ALGORITHMS FOR COMBINATORIAL OPTIMIZATION IN REAL TIME AND THEIR AUTOMATED REFINEMENTS BY GENETICS-BASED LEARNING

BY

LON-CHAN CHU

B.S., National Taiwan University, 1985
M.S., University of Illinois at Urbana-Champaign, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

ALGORITHMS FOR COMBINATORIAL OPTIMIZATION IN REAL TIME AND THEIR AUTOMATED REFINEMENTS BY GENETICS-BASED LEARNING

Lon-Chan Chu, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1994
Benjamin W. Wah, Advisor

The goal of this research is to develop a systematic, integrated method of designing efficient search algorithms that solve optimization problems in real time. Search algorithms studied in this thesis comprise meta-control and primitive search. The class of optimization problems addressed are called combinatorial optimization problems, examples of which include many NP-hard scheduling and planning problems, and problems in operations research and artificial-intelligence applications. The problems we have addressed have a well-defined problem objective and a finite set of well-defined problem constraints. In this research, we use state-space trees as problem representations. The approach we have undertaken in designing efficient search algorithms is an engineering approach and consists of two phases: (a) designing generic search algorithms, and (b) improving by genetics-based machine learning methods parametric heuristics used in the search algorithms designed. Our approach is a systematic method that integrates domain knowledge, search techniques, and automated learning techniques for designing better search algorithms. Knowledge captured in designing one search algorithm can be carried over for designing new ones.

ACKNOWLEDGEMENTS

I express my sincere gratitude to all the people who have helped me in the course of my graduate study. My thesis advisor, Professor Benjamin W. Wah, was always available for discussions and encouraged me to explore new ideas. I am deeply grateful to the committee members, Professor Prithviraj Banerjee, Professor Jane Liu, and Professor Benjamin W. Wah. I sincerely thank Professor David E. Goldberg and Professor Kwei-Jay Lin for their services on the committee of my preliminary examination.

I am deeply grateful to Ms. Yi-Wen Guu for her helpful discussions in complex mathematical problems and in combinatorial algorithms.

I am grateful to Dr. Akiko Aizawa, Dr. Yao-Jen Chang, Mr. Kumar Ganapathy, Dr. Mark Gooley, Mr. Arthur Ieumwananonthachai, Dr. Pankaj Mehra, Mr. Yi Shang, Mr. Chin-Chi Teng, and all my colleagues in the Center for Reliable and High-performance Computing for their invaluable suggestions and helpful discussions.

I thank my wife, Yi-Wen Guu, and my family for many years of love and support.

The learning program, TEACHER, is coauthored by Professor Benjamin W. Wah, Mr. Arthur Ieumwananonthachai, and me. The circuit testing program, CRIS, was written by Professor Daniel Saab.

I wish to thank Professor Michael C. Loui and Professor Errol Lloyd for their clarification on graph partition problems. I also wish to thank Dr. Guo-Jie Li for his helpful discussions in search.

When I spent three summers at Ames Research Center, National Aeronautics and Space Administration, Dr. Jerry Yan was always available for helpful discussions. I want to thank him here.

Finally, I express my appreciation for the financial support provided by the National Science Foundation Grants MIP 88-10584 and MIP 92-18715, National Aeronautics and Space Administration Contracts NCC 2-481, NAG 1-613, and Joint Services Electronics Program N00014-90-J-1270.

TABLE OF CONTENTS

CHAPTER PAGE

1. INTRODUCTION 1
   1.1. Combinatorial Optimization Problems 2
   1.2. Computational Resources 4
   1.3. Combinatorial Search Algorithms 5
   1.4. Genetics-Based Learning 7
   1.5. Research Problems Addressed and Assumptions 8
   1.6. Approach 9
      1.6.1. First phase of design process 10
      1.6.2. Second phase of design process 12
   1.7. Summary of Contributions 13
   1.8. Organization of This Thesis 14

2. COMBINATORIAL SEARCH ALGORITHMS: BACKGROUND, MODELING, AND RESOURCE CONSTRAINTS 15
   2.1. Background on State-Space Search 16
      2.1.1. Preliminary 16
      2.1.2. Assessment of solution quality 18
      2.1.3. Basic selection strategies 20
      2.1.4. Pruning mechanisms 22
      2.1.5. Representations of search spaces 25
      2.1.6. Iterative searches 27
   2.2. Modeling of State-Space Search Algorithms 31
      2.2.1. Previous work 32
      2.2.2. Proposed model 34
      2.2.3. Illustrative examples 38
   2.3. Real-Time Approximation Searches 42
      2.3.1. Characteristics of real-time systems 43
      2.3.2. Anytime algorithms and imprecise computation 44
   2.4. Survey of Resource-Constrained Search Algorithms 48
      2.4.1. Search algorithms with resource-constrained guidance evaluation 49
      2.4.2. Time-constrained search algorithms 50
      2.4.3. Memory-constrained search algorithms 52
      2.4.4. Searches without bounds on solution quality 55
   2.5. Summary 57

3. REAL-TIME SEARCH 58
   3.1. Preliminary 60
   3.2. Framework of Real-Time Search 68
   3.3. Strategies for Real-Time Search with Approximation 71

3.3.1. Empirical model of search complexity 72
3.3.2. Approximation by linear gradient 75
3.3.3. Approximation by first-order regression 79
3.4. Strategies for Real-Time Search with Thresholding 84
3.4.1. Empirical model of thresholding and its justification 85
3.4.2. Thresholding by linear gradient 87
3.4.3. Thresholding by first-order regression 92
3.5. Strategies for Real-Time Search with Approximation and Thresholding 96
3.5.1. Moving-grid strategy 97
3.5.2. Pruning-ratio strategy 99
3.6. Experimental Results 101
3.7. Summary 118

4. BAND SEARCH 122
4.1. Generic Band Search 123
4.2. Variants of Band Search 132
4.2.1. Static band search 133
4.2.2. Dynamic band search 134
4.2.3. Iterative band search 135
4.3. Properties of Band Search 136
4.4. Band-Width Anomalies 139
4.5. Experimental Results 143
4.5.1. Band search without time constraint 145
4.5.2. Time-constrained band search 159
4.6. Summary 183

5. AUTOMATED LEARNING OF PARAMETRIC SEARCH HEURISTICS 184
5.1. Survey of Related Studies in Learning Heuristics 186
5.2. Parametric Search Heuristics 191
5.2.1. Form of heuristics 191
5.2.2. Type of heuristics 192
5.2.3. Performance of heuristics 194
5.3. Learning Framework 197
5.4. Learning Phase 201
5.4.1. Performance evaluation in learning 202
5.4.2. Nonparametric scheduling of experiments in genetics-based learning 207
5.5. Verification Phase 214
5.5.1. Full evaluation 215
5.5.2. Domain clustering 216
5.5.3. Performance evaluation 228
5.6. TEACHER: A Prototype for Automated Learning 232
5.7. Experimental Results 235
5.7.1. Overview of learning experiments 235
5.7.2. Predefined problem subdomains 238
5.7.3. Experimental parameters 242
5.7.4. Parameters used in heuristics 243

5.7.5. Comparison of scheduling strategies 248
5.7.6. Learned real-time search 251
5.7.7. Learned dynamic-band search 258
5.8. Learning Search Heuristics for Circuit Testing 279
5.9. Summary 281

6. CONCLUSIONS 285

REFERENCES 287

APPENDIX A. EXTREME-CASE ANALYSES FOR REAL-TIME SEARCH..... 296
A.1. Approximation by Linear-Gradient 296
A.1.1. Best-case analysis 296
A.1.2. Average-case analysis 297
A.2. Approximation by First-Order-Regression 300
A.2.1. Best-case analysis 300
A.2.2. Average-case analysis 301
A.3. Thresholding by Linear-Gradient 303
A.3.1. Best-case analysis 303
A.3.2. Average-case analysis 304
A.4. Thresholding by First-Order-Regression 305
A.4.1. Best-case analysis 305
A.4.2. Average-case analysis 305

APPENDIX B. EXPERIMENTAL RESULTS OF LEARNING SEARCH HEURISTICS 307
B.1. Experimental Results of Learning Search Heuristics for the KS Problems 307
B.1.1. Learning decomposition heuristics used in GDFS 307
B.1.2. Learning selection heuristics used in GDFS 311
B.2. Experimental Results of Learning Search Heuristics for the VC Problems ... 315
B.2.1. Learning decomposition heuristics used in GDFS 315
B.2.2. Learning selection heuristics used in GDFS 319
B.3. Experimental Results of Learning Search Heuristics for the PP Problems 323
B.4. Experimental Results of Learning Search Heuristics for the TSPs 328
B.4.1. Learning decomposition heuristics used in GDFS 328
B.4.2. Learning selection heuristics used in band search 333
B.4.3. Learning selection heuristics used in band search to find a solution 336
B.4.4. Learning selection heuristics used in GDFS 340
B.4.5. Learning pruning heuristics used in band search 342
B.5. Experimental Results of Learning Search Heuristics for the Maze Problems 347
B.5.1. Learning selection heuristics used in band search to find a solution 347
B.5.2. Learning selection heuristics used in band search 352
B.5.3. Learning pruning heuristics used in band search 356

VITA 362

CHAPTER 1.

INTRODUCTION

Optimization problems exist in many disciplines, and solving them efficiently is an important issue. When their solution spaces are *enumerable* and possibly infinite, they are known as *combinatorial optimization problems* (COP). In many cases, COPs are NP-hard, especially those in scheduling and planning, operations research, artificial intelligence, and some grand challenge application problems [10]. Moreover, solving COPs becomes very difficult when time allowed is limited.

The goal of this research is to *design efficient search algorithms that solve COPs in real time.*

Search algorithms considered consist of meta-control and primitive search methods [105]. We consider the following aspects of search algorithms: (a) framework of search algorithms; (b) decomposition strategy that defines a branching mechanism and granularity of search nodes, (c) selection strategy that defines a sequence of selection keys used to select the node to explore next, and (d) pruning strategies that comprise one or more of the following mechanisms: approximation [49], thresholding [37], and dominance [29].

The approach we have undertaken in this research is an engineering one. The design of a search algorithm consists of two phases: (a) designing a generic search algorithm, and (b) improving its parametric heuristics by automated learning techniques. In the first phase, we design and study search algorithms, including *Real-Time Search* and *Band Search*. In the second phase, we apply genetics-based learning methods to learn new and better parametric heuristics for these algorithms.

We can apply these two phases in conjunction or individually. For real-time search and band search studied in this thesis, we apply both phases in conjunction. For genetic search (applied to find an optimal test pattern in sequential circuit testing [83]), we only apply the second phase to improve the search algorithm. In the latter case, we assume that the original (genetic) search algorithm is as designed by experts in the field.

Our approach to design better search algorithms is a systematic method that integrates domain knowledge, search techniques, and automated learning techniques. Knowledge captured in designing search algorithms we have studied can be carried over for designing new ones.

In the following sections, we first characterize COPs, resource constraints, search algorithms, and genetics-based learning. We state our research problem and justify the assumptions we have made in this research. We then discuss our two-phase design approach in detail. Finally, we summarize the contributions and the organization of this thesis.

1.1. Combinatorial Optimization Problems

One major feature of a COP is that its solution space is enumerable and possibly infinite. The seminal book written by Garey and Johnson contains a good collection of COPs and defines COPs as follows [18].

> ‘‘A *combinatorial optimization problem* Π is either a *minimization problem* or a *maximization problem*, and consists of the following three parts:
>
> (a) a set D_Π of instances;
>
> (b) for each instance I *member* D_Π, a finite set $S_\Pi(I)$ of *candidate solutions* for

I; and

(c) a function m_Π that assigns to each instance $I \; member \; D_\Pi$ and each candidate solution $\sigma \; member \; S_\Pi(I)$ a positive rational number $m_\Pi(I, \sigma)$ called the *solution value* for σ.'' [18]

Many COPs are NP hard; examples include production planning, multiprocessor scheduling, knapsack packing, partition into cliques, and traveling-salesman routing. On the other hand, some COPs are not NP hard and can be solved by efficient polynomial-time algorithms; examples include the shortest-path problem and the network-flow problem.

In general, a COP can be formulated as a state-space search. In the state-space search, a problem is decomposed into subproblems (states), and the problem is solved by exploring all subproblems. State-space search is commonly used in most search problem representations. The search space is enumerable and can be finite or unbounded. A search space is considered unbounded if it is so huge that enumerating in any reasonable amount of time is prohibitive, whereas problems with a finite search space can be enumerated. Note that a problem with an unbounded search space can be transformed by setting thresholds into a (possibly infinite) sequence of searches with finite search space [37]. In this research, we only address problems with finite search space.

A COP in this research has a well-defined problem objective and a finite set of problem constraints. A well-defined problem objective can be formulated algebraically in terms of problem parameters, while an ill-defined problem objective can only be measured but the relation between the problem parameters and the objective value is unknown. Problem constraints can be checked for feasibility by using algebraic equations, rules, and procedures.

In this research, our primary objective is to find optimal solutions under limited time. For problems whose optimal solutions can be found in the time allowed, we are interested to find solutions as close to the optimal solution as possible. For some COPs, we can find feasible solutions by polynomial-time algorithms; these include the traveling salesman problem with fully connected cities and the knapsack problem. For other COPs, there are no existing polynomial-time algorithms (such as the fifteen-puzzle problem and the traveling salesman problem with sparsely connected cities). For problems with existing polynomial-time algorithms, their solution space is referred to as *dense* [92]. On the other hand, for problems without polynomial-time algorithms, their search space is referred to as *sparse*; in this case, feasible solutions may have to be obtained by backtracking.

For brevity, we will use ‘‘dense COPs’’ to refer to COPs whose solution space is dense, and ‘‘sparse COPs’’ to refer to COPs whose solution space is sparse.

1.2. Computational Resources

Resources are limited for computing in most practical systems. In general, resource constraints can include *time*, *memory space*, and *number of processors*, as are shown in Figure 1.1. Time is a constraint imposed by the application requirements; in contrast, memory space and number of processors are architectural constraints imposed by the computer system in which COPs are solved.

Resource constraints can be hard or soft. A hard constraint cannot be violated, whereas a soft constraint may be violated but the utility of the solution degrades gracefully. In this research, time and number of processor are hard constraints; in particular, we are interested in sequential algorithms, that is, the case when only a processor is available as a computing resource. Memory space is treated as a soft constraint.

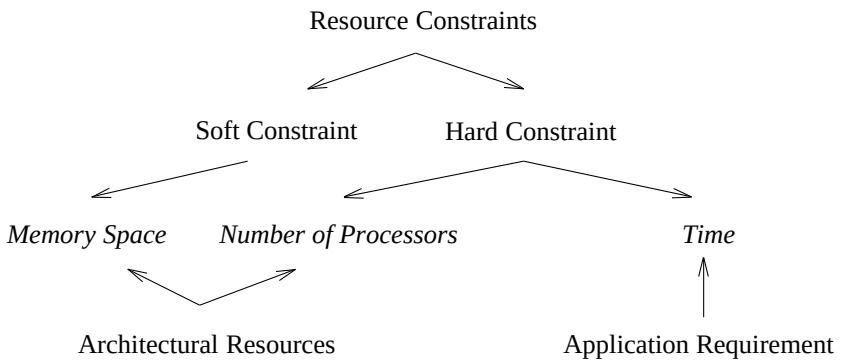


Figure 1.1. Classification of resource constraints posted in this thesis.

Time constraint is different from architectural constraints in many aspects. Time is a resource that cannot be used and released; namely, time usage is monotonically increasing and cannot decrease in any way. In contrast, processing resources can be used and released. Moreover, computation continues as long as the architectural resources used do not exceed the constraints; namely, computation can continue even when the architectural resources used are exactly at the levels of the constraints. In contrast, computation must stop when time used is equal to the time constraint.

1.3. Combinatorial Search Algorithms

Methods for solving COPs are referred to as *search algorithms*, whose structure with flow of information is shown in Figure 1.2. A search algorithm in general comprises two levels of search control [105]: high-level meta-control and low-level primitive search control. The meta-control schedules resources and assigns parameters used in primitive searches. One possible form of such an algorithm is a sequence of primitive searches with meta-control in between. One typical example is IDA* [37] that uses depth-first search as its primitive search, and its meta-control selects thresholds used in the primitive searches.

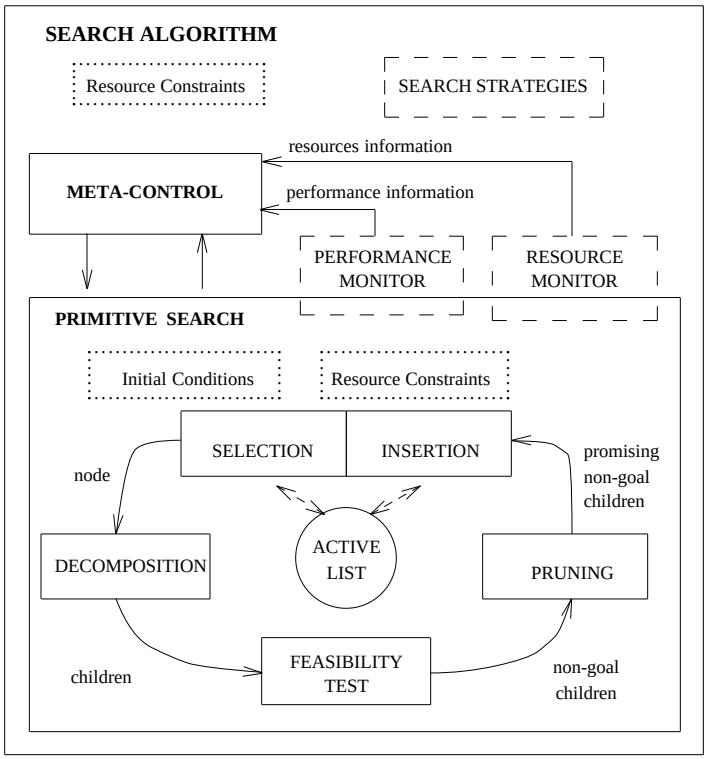


Figure 1.2. Structure of a search algorithm.

The inputs to a search algorithm include the problem instance to solve and the amount of resources that can be consumed during the search. The performance monitor and the resource monitor acquire run-time information of search progress and resources; in many cases, this information is incomplete. As a result, a search algorithm becomes non-deterministic because it uses incomplete information to make decisions. Due to this non-determinism, anomalies may happen in parallel search and approximation search, which have been extensively studied by others [51, 53].

Meta-control schedules resources available to and selects pruning mechanisms for the next primitive search, based on the information supplied by the performance monitor and the resource monitor. Meta-control also defines the initial conditions of the primitive search. A primitive search uses the following strategies:

(a) *Selection Strategy.* This specifies selection keys used for ordering active search nodes. The boxes labeled selection and insertion in Figure 1.2 are related to the selection strategy.

(b) *Decomposition Strategy.* This defines the operator that expands (or transforms) a search node into its child nodes. The decomposition strategy is highly problem dependent.

(c) *Pruning Strategy.* This employs pruning mechanisms to trim subtrees leading to potentially poor solutions. Traditional pruning mechanisms include approximation [49], thresholding [37], and dominance [29].

In this thesis, we design meta-control and search strategies. Moreover, we improve the search strategies we have developed by genetics-based learning.

1.4. Genetics-Based Learning

Genetics-based learning, like genetic algorithm, is an evolving process consisting of multiple generations. In a generation, a finite set of heuristics are generated or retained from the previous generation, and then are tested. At the end of a generation, all heuristics in this generation are associated with their respective fitness values, and are selected randomly with a probability directly proportional to their respective fitness values. The heuristics selected are used in the next generation to generate new heuristics to be tested in the next generation.

The fitness value of a heuristic can be computed as follows. Consider m heuristics. The estimated performance value of heuristic i is denoted as x_i. Given the estimated performance values of all heuristics, x_i for $i = 1, ..., m$, the fitness value f_i of heuristic i is defined as

$$f_i = del\ x_i - \min_{j=1,m} x_j\ .$$

Note that the fitness value of a heuristic is directly proportional to its performance value.

Generation operators used in a genetic algorithm for generating new heuristics include *crossover* and *mutation*. For heuristics that can be expressed as a tree (such as a tree representing the operators and operands in an arithmetic expression), the crossover operator can swap subtrees between two trees, and the mutation operator can replace a subtree with a variable or a random subtree. These operators are most useful when the domain is knowledge-lean as the new heuristics are generated without using domain knowledge.

In our research, search heuristics are algebraic expressions that have strong syntactic requirements such as balanced parentheses and precedence of arithmetic operators. As a result, we adopt parse trees rather than bit strings as for representing heuristics. In this thesis, we use genetics-based learning methods to learn better search heuristics that are expressed in the form of arithmetic expressions. We do not study learning methods that learn complex symbolic rules and macro-operators.

1.5. Research Problems Addressed and Assumptions

The research problem we have addressed in this thesis is to *design efficient search algorithms that solve COPs in real time, where the COPs have well-defined problem objectives and are represented by state-space OR trees*. We decompose this general research problem into two parts: (a) designing efficient search algorithms that solve dense COPs in real time, and (b) designing efficient search algorithms that solve sparse COPs in real time. We address these two research subproblems separately because they have different characteristics.

In most real-time systems, solution quality trades with time in one or more dimensions [50]: *completeness* (some solution components are neglected), *precision* (some solution components are not in their exact forms), *certainty* (confidence on some solution components is not high), and *approximation* (solution is not optimal). In this research, we are interested in the last dimension, namely, approximation, because we are interested in exact and complete solutions that are as close as possible to the optimal solution.

We have made the following three general assumptions for the problem domain we have studied.

Assumption 1.1. *The COPs we study have a well-defined problem objective and a finite set of problem constraints. They are formulated as state-space search in the form of OR trees, and their search spaces are bounded.*

Assumption 1.2. *Resources are measured in logical units; specifically, time is measured as the number of nodes expanded and memory space as the maximum number of active nodes at any particular time. There is only one processor available as the computational resource.*

Assumption 1.3. *There is not a good model relating the parametric heuristics to be learned and their performance. Further, the parametric form of the heuristics can result in a large number and possibly an infinite number of heuristics to be instantiated, too many to be evaluated in any reasonable amount of time.*

1.6. Approach

The approach undertaken to address our research problem consists of two phases and is shown in Figure 1.3. In the first phase, we develop the hierarchical search process, which models activities in search algorithms as a hierarchy of primitive activities. We then design

new search algorithms that perform well for their respective target classes of COPs. In the second phase, we apply genetics-based learning methods to improve search heuristics used in the algorithms designed. In this phase, we only focus on tuning parameters of heuristics expressed in a standard parametric form of arithmetic expressions. We did not attempt to learn other forms of heuristics used in search algorithms.

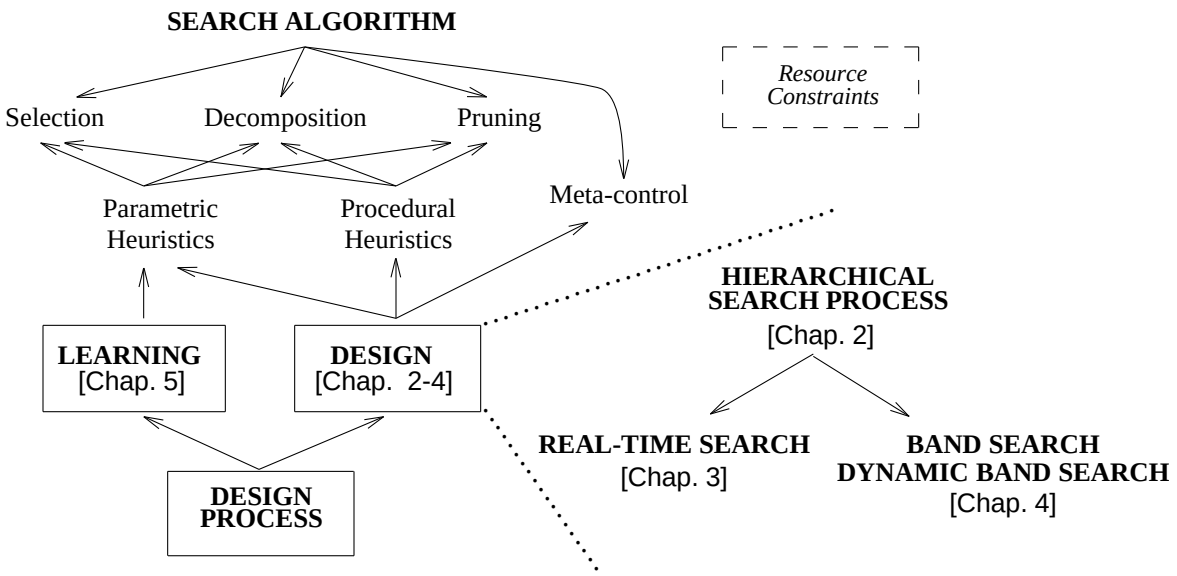


Figure 1.3. Two-phase design process.

1.6.1. First phase of design process

In the first phase, we design search algorithms and address the following design issues.

(a) *Framework of Search Algorithm.* We first develop the framework of the search algorithm including meta-control and primitive search, where the framework may iterate on primitive searches, or simply a single primitive search. In general, the framework may iterate on a variety of primitive searches. For example, in real-time search, the framework iterates on guided depth-first searches with approximation and

thresholding.

(b) *Selection Strategy.* The selection strategy employs a sequence of selection keys to select promising active nodes to expand in primitive searches. One of the problems is to determine the selection keys. For example, in band search to be discussed in Chapter 4, two selection keys are used: one for determining the scope of the bands, and the other, heuristic values for ordering active search nodes.

(c) *Decomposition Strategy.* The decomposition strategy defines the expansion operators and the granularity of search nodes used in primitive searches. Both operators and granularity used are problem dependent.

(d) *Pruning Strategy.* The pruning strategy has a significant effect on search performance in terms of resources used and solution quality. It is part of the primitive search. Basic pruning mechanisms commonly used include approximation, thresholding, and dominance. The pruning strategy is particularly useful when we want to find a solution very fast without guarantee on its quality. We also can design a composite pruning function consisting of basic pruning mechanisms. In the second phase, we focus on learning pruning rules that lead to heuristic solutions (with no guarantee on solution quality).

Specifically, we develop a new search algorithm, called *Real-Time Search*, that is able to solve dense COPs in real time. The algorithm employs guided depth-first search as a primitive search and applies three pruning mechanisms: approximation, thresholding, and dominance.

We also develop a new search algorithm, called *Band Search*, that is a hybrid between best-first search and guided depth-first search. This algorithm acquires nice properties from both, namely, less number of nodes expanded than a guided depth-first search and a linear amount of memory as is in a guided depth-first search. We develop an extension called *Dynamic Band Search* that solves sparse COPs in real time.

1.6.2. Second phase of design process

In the second phase, we apply genetics-based learning methods to learn knowledge-lean parametric heuristics in search algorithms. By ‘‘knowledge lean,’’ we mean that there is no domain knowledge relating parameters used in heuristics and their performance. We first develop a learning system that can learn knowledge-lean heuristics for problems of various characteristics. The learning system accepts a baseline heuristic and a set of parameters as inputs and discovers new heuristics comprising of assignment of values to a subset of these parameters.

The parametric heuristics to be learned include (but are not limited to) parametric heuristics for decomposition, parametric heuristics for selection, and parametric heuristics for pruning. They are briefly described here.

(a) *Parametric Heuristics for Decomposition.* In decomposition, a heuristic is used for selecting the best attribute to decompose a search node into child nodes. Usually, the heuristic can be represented as a function of parameters of a search node.

(b) *Parametric Heuristics for Selection Keys.* Each selection key is a function of parameters of a search node. The selection key may not necessarily be the lower bound or the depth as is commonly used; instead, they can be a function of the lower bound,

the upper bound, the depth, and other performance-related attributes.

(c) *Parametric Heuristics for Pruning.* We learn heuristic pruning rules for finding a feasible solution quickly. These learning rules are expressed as algebraic functions that are used to determine whether a node dominates another so that the one dominated can be pruned.

1.7. Summary of Contributions

In short, the contributions of this thesis are as follows:

(a) development of a hierarchical search process that models activities of search algorithms in a systematic and hierarchical fashion;

(b) design of a modular software package for prototyping efficient resource-constrained search algorithms;

(c) design of a real-time search algorithm that solves dense COPs in real time and obtain solutions that are close to the optimal solutions and that have good bounds on quality;

(d) design of Band Search that generalizes best-first and guided depth-first searches and solves COPs efficiently, and that uses a limited and predictable amount of memory;

(e) design of Dynamic Band Search that finds good/optimal solutions quickly for sparse COPs under time constraints;

(f) design of a learning framework for learning parametric search heuristics for knowledge-lean application problems; and

(g) design of a non-parametric schedule for testing heuristics learned by genetics-based machine learning methods.

1.8. Organization of This Thesis

This thesis is organized as follows. Chapter 2 gives background on search, models of search algorithms, and previous work on resource-constrained search algorithms. Chapter 3 describes Real-Time Search, followed by its analysis and experimental results. Chapter 4 describes Band Search, Dynamic Band Search, and their analysis and experimental results. Chapter 5 gives a brief survey on methods for learning search heuristics and describes the learning framework and system. We also study a non-parametric schedule for testing heuristics, experiment setups, and experimental results. Finally, we draw conclusions in Chapter 6.

CHAPTER 2.

COMBINATORIAL SEARCH ALGORITHMS: BACKGROUND, MODELING, AND RESOURCE CONSTRAINTS

Combinatorial optimization problems fall in a general class of problems known as combinatorial search problems, which include optimization, feasibility, decision, and constraint satisfaction. A feasibility problem is to find *k* feasible solutions regardless of their solution quality, where *k* is a prescribed constant. A decision problem returns an answer ‘‘yes’’ or ‘‘no’’ to the question raised. A constraint-satisfaction problem is to find a problem state that satisfies all prescribed symbolic problem constraints and propagated constraints [24, 80]. In this thesis, we only address optimization problems.

A COP has an objective to be achieved, and solving it is to accomplish this objective. However, due to limited time, it may not be possible to find an optimal solution. In this case, a feasible but suboptimal solution has to be found.

For brevity, we study optimization of minimization search problems. Generality is not lost because maximization is equivalent to minimization by negating the objective or by solving its dual problem.

In this chapter, we give the background on search algorithms and address their modeling, and then survey related studies in resource-constrained search algorithms. We survey related studies in learning search heuristics in Chapter 5.

2.1. Background on State-Space Search

In this section, we first describe basic concepts in branch-and-bound search. Then, we discuss important issues in state-space search, including quality assessment, basic selection strategies, pruning mechanisms, representation of search spaces, iterative searches, and approximation searches.

2.1.1. Preliminary

A branch-and-bound (B&B) search is a general search algorithm [49] shown to be a general formulation of many search algorithms [43]. In this thesis, without loss of generality, we discuss search in terms of the B&B search.

A B&B search decomposes a problem into smaller subproblems and repeatedly decomposes them until a solution is found or infeasibility is proved. Each subproblem is represented by a node in the search tree. The search consists of four components: (a) a selection rule that selects a search node for expansion; (b) a pruning rule that removes inferior nodes according to prescribed pruning mechanisms; (c) a decomposition rule that expands a search node; and (d) a termination rule that determines when the search should stop.

A node i in the search tree is associated with an upper bound u_i and a lower bound ι_i. Let x_i be the best solution in the subtree rooted at node i. Then, $\iota_i \le x_i \le u_i$. In a monotone admissible heuristic search [72], ι_i satisfies the following properties: (a) $\iota_i \le x_i$; (b) $\iota_i = x_i$ if node i is a leaf, and (c) $\iota_i \le \iota_j$ if node i is an ancestor of node j. In the original description of the B&B algorithm, upper bounds are not used [49]. In this thesis, we compute upper bounds of search nodes whenever possible because we can assess the quality of feasible

solutions found using upper and lower bounds (to be discussed in the next subsection). Note that a lower bound is obtained by solving a relaxed problem [75], while an upper bound is evaluated by an efficient algorithm based on methods such as gradient descent and hill climbing.

An *incumbent* is the best solution found so far during the search. Any node with lower bound no less than the incumbent is obviously inferior and can be pruned. Without loss of generality, we assume that solution values, upper bounds, and lower bounds are either all positive or all negative.

There are three types of nonterminal search nodes in a search tree. *Essential nodes* are ones that have lower bound values less than the optimal solution value [54]. On the other hand, *nonessential nodes* are ones that have lower bounds greater than the optimal solution value [54]. Sometimes, there exist nodes that have lower bounds equal to the optimal solution value. They are *semiessential nodes*.

A search for an optimal solution must involve expanding all essential nodes and may require expanding some semiessential and nonessential nodes. When we only want to find a good solution, we may expand some essential nodes, some semiessential nodes, and some nonessential nodes. Let N_e, N_{se}, and N_{ne} denote the numbers of essential, semiessential, and nonessential nodes, respectively. Let $M_{so}(S)$ and $M_{fo}(S)$ denote the numbers of search nodes expanded by search algorithm S for solving an optimal solution and for finding a feasible solution, respectively. Note that $M_{so}(S) \geq M_{fo}(S)$ because finding a feasible solution is part of solving an optimal solution. In terms of N_e, N_{se}, and N_{ne}, for every possible search algorithm S we have the following relation.

$$N_e \le M_{so}(S) \le N_e + a_1 N_{se} + a_2 N_{ne} ,$$

where $0 \le a_1 , a_2 \le 1$. For every possible search algorithm *S*, we also have a relation as follows.

$$1 \le M_{fo}(S) \le a'_1 N_e + a'_2 N_{se} + a'_3 N_{ne} ,$$

where $0 \le a'_1 , a'_2 , a'_3 \le 1$. The best search algorithm S^* that can be designed is one that achieves $M_{so}(S^*) = N_e$. Unfortunately, the best search algorithm does not exist generally.

2.1.2. Assessment of solution quality

An optimization search optimizes the quality of the solution obtained in terms of a prescribed measure. There are four popular measures for assessing solution quality: approximation, accuracy, completeness, and belief.

(a) *Degree of Approximation.* The degree of approximation ε indicates the deviation of *z*, the value of feasible solution obtained, from z^*, the optimal solution value. It is defined as follows.

$$\varepsilon =_{del} \left| \frac{z - z^*}{z^*} \right| . \qquad (2.1)$$

Note that $0 \le \varepsilon < \infty$. The optimal solution has a degree of approximation equal to 0.

(b) *Degree of Accuracy.* The degree of accuracy α indicates how close *z* is to z^* and is defined as

$$\alpha =_{del} \frac{\min \{ z^*, z \}}{\max \{ z^*, z \}} .$$

Note that $0 \le \alpha \le 1$. The optimal solution has a degree of accuracy equal to 1.

(c) *Degree of Completeness.* The degree of completeness indicates the completeness of the solution obtained, which may be incomplete because all resources were expended and the search has to be terminated prematurely. Typical examples can be found in time-constrained robotic routing [50]. The degree of completeness is a value between 0 and 1. A complete solution has a completeness degree equal to 1.

(d) *Degree of Belief.* The belief degree indicates heuristically or statistically the belief that the solution found is the one desired. This is useful in searching in a probabilistic environment. The degree of belief is a value between 0 and 1. A totally confident solution has a belief degree equal to 1.

In this thesis, we are interested in exact and complete solutions; therefore, the degree of completeness and the degree of belief are irrelevant. Approximation and accuracy are duals because one can be derived from the other. In this thesis, we choose the degree of approximation because it is commonly used in approximation searches.

During a search, we may not know the optimal solution value. In this case, the approximation degree of a solution is defined by a function of incumbent z and least lower bound ι.

$$\varepsilon = del\left|\frac{z-\iota}{\iota}\right|. \tag{2.2}$$

This degree of approximation is called the *asymptotic approximation degree* (AAD). In contrast, the one defined using the optimal solution value (Eq. (2.1)) is called the *exact approximation degree* (EAD). The AAD is asymptotic because AAD equals EAD when an optimal solution is solved. For problems with positive solution values and bounds, an approximation degree is between 0 and ∞. For negative solution values and bounds, an

approximation degree is between 0 and 1. Without loss of generality, we discuss concepts only in terms of positive solution values and bounds in this thesis.

2.1.3. Basic selection strategies

A state-space search has an active list that contains active search nodes to be selected for expansion. Node expansion generates new search nodes that are then inserted into the active list. The strategy of selecting for expansion a search node in the active list is called a *selection strategy*. In practice, most search algorithms differ only in their selection strategies. In the following, we give background on three basic selection strategies.

(a) *Best-First Strategy.* The best-first strategy always selects for expansion the best search node in the active list, where the goodness of a search node is based on a heuristic function of parameters of this node. Typical examples include the A* algorithm [73] in graph search and the best-first search [108] in tree search. It has been proved that the best-first strategy expands the least number of search nodes in finding an optimal solution [13, 19, 73] when there is no search node whose lower bound is equal to the optimal solution. The extension of the A* algorithm for searching AND/OR graphs is the AO* algorithm [7, 73]. Due to this optimality, the A* algorithm and its variants have been extensively studied [7, 13, 14, 19, 25, 64, 76, 77]. However, they require memory space for storing active search nodes that is exponential with the problem size. As a result, it is impractical to use these algorithms to solve moderately large problems in real-world applications. Efforts have been attempted to bound the memory required for best-first style searches. Examples include depth-m search [30] and band search [9] described in Chapter 4.

(b) *Depth-First Strategy*. The depth-first strategy always selects for expansion the deepest search node in the active list. The depth-first search has two basic forms. During expansion, if only one child is generated, the search is a depth-first search. If all the children are generated and sorted based on their heuristic values, and the child with best heuristic value is always selected for expansion, then the search is called a *guided depth-first search*. A guided depth-first search is more informed than a depth-first search. On the other hand, a guided depth-first search has a larger memory requirement than a depth-first search. The major advantage of a (guided) depth-first search is that it requires less memory than a best-first search. A (guided) depth-first search goes deep into the search graph/tree and, by chance, may find a good solution very quickly. However, it may be trapped deep in the search graph/tree and may have to backtrack after spending substantial efforts in a fruitless subtree. There are two types of backtracking in (guided) depth-first searches: chronological backtracking and dependency-directed backtracking [108]. *Chronological backtracking* can undo the state of a search node to that of its parent node. In contrast, *dependency-directed* backtracking can undo the state of a search node to that of its ancestor node and involves tracing back through dependencies and identifying the attributes contributing to the dead end. It can skip over several levels of the search graph/tree and can ignore the intermediate search nodes. This concept was first proposed by Stallman and Sussman [88]. In terms of the branch-and-bound algorithm, dependency-directed backtracking is similar to pruning by dominance [29] since the intermediate search nodes skipped in the search can be considered to be pruned by dominance.

(c) *Breadth-First Strategy.* The breadth-first strategy always selects for expansion the search node in the highest level of the search tree. Breadth-first search does not get trapped deep in the search graph/tree because it goes down level by level. However, it goes very slowly to the bottom of the search graph/tree where solutions reside. With a brute-force implementation, a breadth-first search requires an amount of memory exponential in the problem size. However, with a proper encoding of the current search path in a search tree, a breadth-first search only requires a constant amount of memory.

From the properties of these three basic selection strategies, a plausible heuristic search should (a) require limited amount of memory such as a (guided) depth-first search, (b) expand an amount of search nodes comparable to that of a best-first search, and (c) be able to deliver a solution when the search is stopped at any time after an initial interval. In this thesis, our goal is to design search algorithms with these three properties.

2.1.4. Pruning mechanisms

Pruning is an important element in search algorithms that solve COPs in real time because it can reduce search time. In this section, we give background on four popular pruning mechanisms: pruning by approximation, pruning by thresholding, pruning by dominance, and heuristic pruning. They can be applied individually or in conjunction. They are described as follows.

(a) *Pruning by Approximation.* Approximation prunes a search node when the best possible solution in the subtree rooted at this search node is larger than a prescribed deviation from the (unknown) optimal solution value. During a search, we can use the incumbent as an estimator of the optimal solution. A larger approximation degree ε

will generally cause more nodes to be pruned and the search completed in less time. The following pruning rule (P-RULE I) prunes node *i* (with lower bound ι_i) when

$$\textit{P-RULE I}: \quad \iota_i \geq \frac{z}{1+\varepsilon} , \tag{2.3}$$

where *z* is the incumbent and ε is the prescribed approximation degree. The final best solution obtained by applying P-RULE I has an achieved AAD equal to ε. Typical examples of using pruning by approximation include IRA* [55], TCA* [100], and TCGD [101].

(b) *Pruning by Thresholding.* Thresholding can be applied to prune search nodes to allow the search to complete faster [37]. Given a threshold θ, any node *i* with lower bound $\iota_i \geq \theta$ is pruned. A smaller threshold will generally cause more nodes to be pruned and the search completed in less time; however, the solution obtained may be worse. Consider the following pruning rule (P-RULE II):

$$\textit{P-RULE II}: \quad \iota_i \geq \theta . \tag{2.4}$$

The final best solution *z* obtained by applying P-RULE II has an achieved AAD as

$$\begin{aligned} \varepsilon(\theta) &= \frac{z-\theta}{\theta} = \frac{z}{\theta} - 1 \quad \text{if } 0 < \theta \leq z^* , \\ &= 0 \quad \text{if } \theta > z^* , \\ &= \varepsilon_0 \quad \text{if } \theta \leq 0 , \end{aligned} \tag{2.5}$$

where ε_0 is the AAD obtained by generating the root node. Typical examples of using pruning by thresholding include IDA* [37], MIDA* [103], IDA*_CR [84], and DFS* [92].

(c) *Pruning by Dominance.* Pruning by dominance is a powerful, problem dependent mechanism to prune search nodes [29]. A dominance relation may exist between any two search nodes: node *i* is said to dominate node *j* if we can conclude that the best solution in the subtree rooted at node *i* is no worse than the best solution in the subtree rooted at node *j* without expanding both nodes. Consider the knapsack problem with a sack with a weight limit and a set of objects, each having a profit and a weight. The knapsack problem is to maximize the total profit of the objects in the sack such that the total weight of these objects in the sack does not exceed the weight limit of the sack. Consider two search nodes *i* and *j* having examined the same subset of objects. If the sack of node *i* has more profit and less weight than that of node *j*, then we can conclude that node *i* dominates node *j* because the best solution derived from node *i* cannot be worse than that from node *j*. A special form of dominance is that an incumbent dominates a search node with lower bound larger than this incumbent.

(d) *Heuristic Pruning.* Pruning can also be done heuristically. When heuristic pruning is applied, node *x* prunes node *y* if a relation $f(x,y) \geq g(x,y)$ holds, where f and g are two real-valued functions and ''≥'' is the numerical ''greater than or equal to'' operator. Note that node *x* prunes node *y* implies that node *y* cannot prune node *x*.

All of these mechanisms can be used individually or in conjunction. However, combining them may affect the correctness of search. For example, we cannot guarantee the bound on solution quality when applying both approximation and heuristic pruning at the same time. When approximation and thresholding are applied in conjunction, the bound on solution quality may not be equal to the prescribed degree of approximation and may have to be calculated based on the threshold prescribed. A powerful algorithm should exercise as

many mechanisms as possible to prune search nodes without violating the application requirements, though it is hard to have a systematic way of combining these mechanisms efficiently.

2.1.5. Representations of search spaces

In this thesis, we study the OR-tree representations of COPs. In order to give a picture of various representations of search spaces, we give background on them in the following.

A state-space search can be a graph search, a tree search, or a game-tree search. The major difference between graph searches and tree searches is that a graph search may reach a search node through different paths, and a tree search reaches a node through a unique path. This implies that in a graph search a search node can have multiple parent nodes, but in a tree search a node can have only one parent node. One drawback in graph searches is that we have to store all nodes visited to check whether they have been visited before. This makes graph search impractical for solving large COPs. If the search does not care whether a node has been traversed before, then a graph search becomes a tree search. Both graph and tree searches involve only one agent or player who is the problem solver. In game-tree searches, there are two (or even multiple) agents hostile to each other. In a game-tree search, an optimal move is to minimize possible damage that the other agent can cause. Traditional techniques in game-tree searches include minimax search [73] and alpha-beta search [2, 36]. In this thesis, we address problems that can be solved efficiently by tree searches. A state-space tree search can be either an AND-tree search, an OR-tree search, or an AND/OR-tree search.

An AND-tree search implements a divide-and-conquer method [3, 27] because all the branches of a search node must be completed before this node can be completed. Therefore, the solution of the AND-tree search is the entire search tree. The performance issues include the ordering of branches [22].

An OR-tree search can take any one of the branches of a search node. The solution of an OR-tree search is a path from the root node to a leaf goal node. Related performance issues include the decomposition of a search node, the ordering of nodes in the active list, and the pruning of inferior search nodes.

An AND/OR-tree search is a generalization of an AND-tree search and an OR-tree search. An AND/OR tree can be viewed as a search tree consisting of AND nodes and OR nodes in alternating levels. The solution of an AND/OR-tree is a subtree from the root. Related performance issues include the ordering of AND child nodes, in addition to the issues related to OR-tree searches.

The direction of a state-space search can either be forward, backward, or bidirectional [73]. A forward search starts from the root node and continues to the goal node. A backward search starts from the goal node and continues to the root node. A bidirectional search starts from both the root and the goal and stops when a node in the forward subtree meets one in the backward subtree [46]. Bidirectional search is particularly useful in solving problems that have dual problems, where the forward tree represents the original problem and the backward tree represents the dual problem. Since different problem representations may incur different computational overheads, this scheme can bound the worst-case execution time in the sense that the worst-case time is double the minimum of execution times of these two representations. The major drawback of a bidirectional search is that a

large amount of memory is needed to store frontier nodes in both the forward and backward subtrees.

In this thesis, we only consider problems represented by forward search trees. We do not consider bidirectional searches because they require large memory space and are not commonly used in solving combinatorial search problems.

2.1.6. Iterative searches

An iterative search iterates on simple searches. The distinction between the two is that a node may be expanded multiple times in an iterative search, whereas every node is expanded at most once in a simple search such as depth-first search.

Iterative searches are very useful in search algorithms that solve COPs in real time because they can collect information on search progress and schedule what to search in the remaining time. In the following, we give background on iterative searches.

Existing iterative searches (such as IDA*, MIDA*, IDA*_CR, IRA*, and DFS*) start with a (guided) depth-first search and a threshold for pruning. Assuming a minimization search, all search nodes found during the search with lower bounds greater than the prescribed threshold are pruned. When all nodes are either pruned or expanded, the best feasible solution is returned and the process stops. If no feasible solution is found, then a new (guided) depth-first search is started with a larger threshold. This process is repeated until one feasible solution is found. It can be proved that an optimal solution is the best among these feasible solutions. These iterative searches differ in their mechanisms of determining the threshold for the next search. In the following, we describe methods for determining the thresholds for various iterative searches.

(a) *IDA**. The IDA* algorithm uses the smallest lower-bound value that is just greater than this current threshold as the threshold for the next search. The IDA* algorithm is a breakthrough in heuristic search because it simulates a best-first search with limited memory, and the number of search nodes expanded is comparable to that of a best-first search. Moreover, it does not get trapped deep in the search tree as is in a (guided) depth-first search, and uses memory space linear in the height of the search tree. The earliest version of the IDA* algorithm is the iterative-deepening depth-first search algorithm. The threshold here is the depth that the current search can go to; that is, all search nodes with depths greater than this threshold are pruned. The next threshold is the least depth of all pruned search nodes. The iterative-deepening depth-first search is equivalent to the IDA* algorithm in solving some discrete problems whose solution value is the same as the depth of the goal node. Good examples of such problems include the fifteen-puzzle problem and the maze problem. The IDA* algorithm also performs very well in solving discrete-valued COPs, but for continuous-valued COPs the thresholds across iterations increase too slowly. The original IDA* algorithm by itself is not suitable for solving COPs in real time because it does not adjust thresholds based on the remaining time.

(b) *MIDA**. The MIDA* algorithm predicts the threshold used in the next guided depth-first search by regressing the relation between the distribution of lower-bound values and the number of nodes expanded. To regress the relation in the beginning, MIDA* starts a limited best-first search and then switches to a sequence of (guided) depth-first searches with thresholding. MIDA* performs very well in both discrete-valued and continuous-valued COPs. The major advantage of MIDA* is that it can dynamically

adjust the increment of the threshold to accommodate the characteristics of different levels in the search tree. MIDA* is not suitable for solving COPs in real time because it does not adjust thresholds based on the remaining time.

(c) *IDA*_CR.* The IDA*_CR algorithm predicts the threshold in the following way. Assume the current threshold is θ and the heuristic value of search node x is $f(x)$. Assume that we want to have a growth rate r across iterations and that the current iteration is the k-th search. First, we put all the pruned search nodes in 100 buckets in such a way that node x is put in bucket i when the following is satisfied.

$$\theta\left[1+\frac{i}{100}\right] < f(x) \le \theta\left[1+\frac{i+1}{100}\right].$$

After all nodes are either expanded or pruned, we consider the first 50 buckets to see whether there exists a minimum index j such that

$$\sum_{i=1}^{j} c_i \ge r^k,$$

where c_i is the number of search nodes in bucket i. If such an index j exists, then set the next threshold equal to $\theta(1+j/100)$. If such an index j does not exist, then set the next threshold equal to $\theta(1+m/100)$, where m is the index of the last nonempty bucket. The idea of the IDA*_CR algorithm is to control re-expansion of search nodes. It works well for some discrete-valued and continuous-valued COPs. However, the resolution of the ranges corresponding to buckets may cause problems when these ranges are comparably large to the change of lower-bound values. It is possible that all pruned search nodes fall in the first several buckets or, even worse, in the first bucket. Consequently, the next threshold overshoots and many more search nodes than $r^{k-1}+r^k$ will be expanded. On the other hand, when the branching degree is small,

then there are not enough search nodes in the buckets to achieve the desired growth rate *r*. This problem has been observed in the maze problem [92].

It is possible to cope with overshooting of thresholds in solving dense COPs. We can assign the range of the buckets to be one percent of the difference between the incumbent (the best solution found so far) and the threshold. That is,

$$\theta + \frac{i\,(z-\theta)}{100} < f\,(x) \le \theta + \frac{(i+1)(z-\theta)}{100}\ ,$$

where z is the incumbent. Let $n\,(\theta)$ be the number of search nodes with lower bounds less than or equal to θ. Based on the statistics collected in the *k*-th search, we can predict the growth profile by regressing over the following two pairs: $(\theta, n\,(\theta))$ and $\left(\theta + \frac{m\,(z-\theta)}{100}, n\,(\theta) + \sum_{i=1}^{m} c_i\right)$, where m is the index of the last nonempty bucket.

The IDA*_CR algorithm is not suitable for solving COPs in real time because it does not adjust thresholds based on the remaining time.

(d) *DFS**. To overcome the problem in which the threshold is extended insufficiently in successive searches, DFS* always doubles the current threshold for the next search. However, it performs similarly to a (guided) depth-first search when the threshold overshoots. Further, it performs slightly worse than a (guided) depth-first search when the search tree is very narrow and deep, as in the X-window maze problem [26] where the maze has its entrance and exit on the opposite boundaries. A restriction of the DFS* algorithm is that it uses a fixed schedule for thresholding by doubling the threshold every time; consequently, it is not suitable for solving COPs in real time.

Among the algorithms based on thresholding for solving continuous-valued COPs, MIDA* generally performs well; for solving discrete-valued COPs with a reasonable branching degree, IDA* is generally the best; and for solving discrete-valued COPs with a narrow branching degree, MIDA* generally performs well. However, all these algorithms are designed for solving search problems under unlimited time, and do not adjust thresholds based on the remaining time; therefore, they are not suitable for solving COPs under limited time.

2.2. Modeling of State-Space Search Algorithms

In this section, we discuss previous work on modeling search and discuss our modeling of state-space search algorithms. Our model is designed to model search algorithms with meta-control and allows resource constraints, approximation, and parallel processing to be incorporated easily in the search process.

The basic building block of our model is a hierarchical search process, which consists of context-free and context-sensitive components classified according to problem-independent and problem-dependent parts. The context-sensitive components are used mainly for evaluating decision parameters and for resolving decision rules in the context-free component.

With this model, we have developed a software tool ISE (acronym for *I*ntegrated *S*earch *E*nvironment), that implements hierarchical searches with meta-control. By separating problem-dependent and problem-independent components in ISE, we can easily prototype new search algorithms and new search applications by reusing existing software in ISE.

2.2.1. Previous work

An early model for representing the process of optimization is due to Karp and Held [35]. They developed the discrete decision process using finite automata for solving discrete optimization problems. Kumar and Kanal later developed the composite decision process that models a search space by a context-free grammar [45].

The *discrete decision process* (DDP) developed by Karp and Held [35] is a ternary tuple $DDP = (\Sigma, \Omega, \phi)$, where Σ is a finite alphabet, Ω is the set of all feasible strings from alphabet Σ, and ϕ is an assessment function defined over Ω. Function ϕ can be real-valued or integer-valued for optimization problems, or can give a ''yes'' or ''no'' answer for decision problems. In traditional optimization problems, Σ represents the set of *solution entities* that are elements for constructing solutions, Ω corresponds to the space of feasible solutions, and ϕ is a cost or utility function. Note that DDP is a model of problems rather than a search process, since the method for generating Ω is not defined.

To model the generation of Ω, Kumar and Kanal proposed the *composite decision process* (CDP) [45], consisting of a context-free grammar as the key component. A context-free grammar is a 4-ary tuple $G = (N, T, P, S)$, where N is a finite set of nonterminal symbols, T is a finite set of terminal symbols, P is a finite set of production rules, and S is the start symbol. Each production rule is of the form $\alpha \rightarrow \beta$, where α is a symbol in N, and β is a string of symbols from $(N \cup T)$. A CDP is a ternary tuple

$$CDP = (G(N, T, P, S), \phi^N, \phi^T),$$

where ϕ^N and ϕ^T are assessment functions associated with nonterminal and terminal symbols, respectively. Note that the search process is primarily specified by the generating

grammar *G*, and that the terminal symbols in the generating grammar are solution entities. The search process can be characterized by the derivation tree derived from the generating grammar such that each nonterminal node in the derivation tree corresponds to a nonterminal symbol in the grammar, and that branching from this nonterminal node corresponds to a production rule in which the nonterminal symbol appears on the left-hand side. Likewise, each terminal node corresponds to a terminal symbol. Without ambiguity, nodes and symbols are used interchangeably in this section. Consider nonterminal node n and its corresponding production rule $p_n \rightarrow a_1\ a_2\ \cdots\ a_k$, the assessment $\phi^N(n)$ is defined as

$$\phi^N(n) = del\ \ f_{p_n}(\phi^N(a_1), \phi^N(a_2), ..., \phi^N(a_k)),$$

where f_{p_n} is the cost function associated with production rule p_n.

The previous models can express either explicit enumeration (enumerating solutions one by one) or implicit enumeration (enumerating solutions via a search graph/tree). Explicit enumeration may require much more time than implicit enumeration because the former cannot apply pruning to reduce the search space. The DDP represents explicit enumeration since enumeration is carried out over set Ω. In contrast, the CDP is more like implicit enumeration since the feasible space can be derived systematically by the generating grammar. In this sense, the DDP is a model for representing problems, and the CDP is for representing search spaces.

CDPs are, however, inadequate in modeling search algorithms, especially those with meta-control. A CDP has one context-free grammar for characterizing context-free search control, including the problem-dependent and problem-independent parts. Hence, in a search with meta-control in which one component may call other basic search algorithms, a single

context-free grammar implementing the entire control may result in complex interactions between the context-free and context-sensitive parts.

A better model to represent search algorithms with meta-control is hierarchical in which components associated with each control mechanism are partitioned.

2.2.2. Proposed model

In this section, we propose a model for search algorithms with meta-control and illustrate the model by showing the formulations of familiar search algorithms.

We develop the *hierarchical search process* (*HSP*) and use it as a building block of search algorithms with meta-control. A search algorithm in general can be considered as a hierarchy of search processes, each controlled by processes up in the hierarchy. We represent such search algorithms as a hierarchy of invocations of HSPs. This representation is natural because many new and efficient search algorithms can be built by composing existing search algorithms. Instead of representing such composite search as a single task, modeling using HSPs provides a uniform way for representing meta-control of primitive searches and for providing standard interfaces between calls and returns of primitive searches. Such a representation corresponds to the notion of software reusability, in which complex searches can be built by calling primitive searches in a library.

We model the basic building block of a search algorithm as an HSP that consists of four components: (a) a problem-independent context-free (PI/CF) component, (b) a problem-independent context-sensitive (PI/CS) component, (c) a problem-dependent context-free (PD/CF) component, and (d) a problem-dependent context-sensitive (PD/CS) component. Figure 2.1 shows the four components and their interactions in terms of flow of information

from one component to another. The PI/CF component is the entry point to the search, representing the definition (or framework) of the search algorithm involved. It invokes the PI/CS and PD/CS components. The former uses specific problem-independent control information in the search, such as the status of resources, while the latter represents the definition of the application to be solved. The last component, PD/CS, involves application-specific information, such as assessments and past performance obtained in the application.

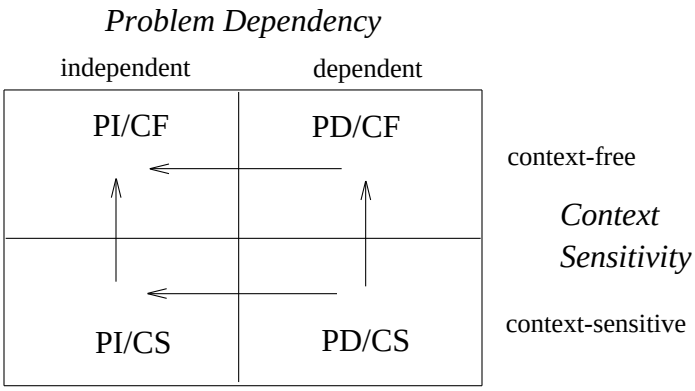


Figure 2.1. The four components of a hierarchical search process, where arrows indicate interactions in terms of flow of information from one component to another.

The context-sensitive component in the definition of the HSP can be viewed as a mechanism for selecting decision rules specified in the context-free components. In other words, the search isolates the context-free components from the context-sensitive search controls by employing the latter to carry out context-sensitive resolution on the former. A search can, therefore, be viewed as a tree derived from the generating grammar specifying the PI/CF search control resolved according to the PI/CS component. A leaf of this tree consists of a context-free grammar specifying the PD/CF search control, the PD/CS component, and a finite set of initial conditions.

The execution of an HSP may result in invocations of other HSPs. If an HSP does not invoke other HSPs, it is called a *primitive search*. Examples of primitive searches are depth-first and best-first searches. Figure 2.2 shows an example of the sequence of HSPs called when $HSP_{1,1}$ is called. $HSP_{1,1}$ invokes four other HSPs, where $HSP_{2,1}$ and $HSP_{2,3}$ are primitive searches, and each of $HSP_{2,2}$ and $HSP_{2,4}$ invokes two other primitive searches.

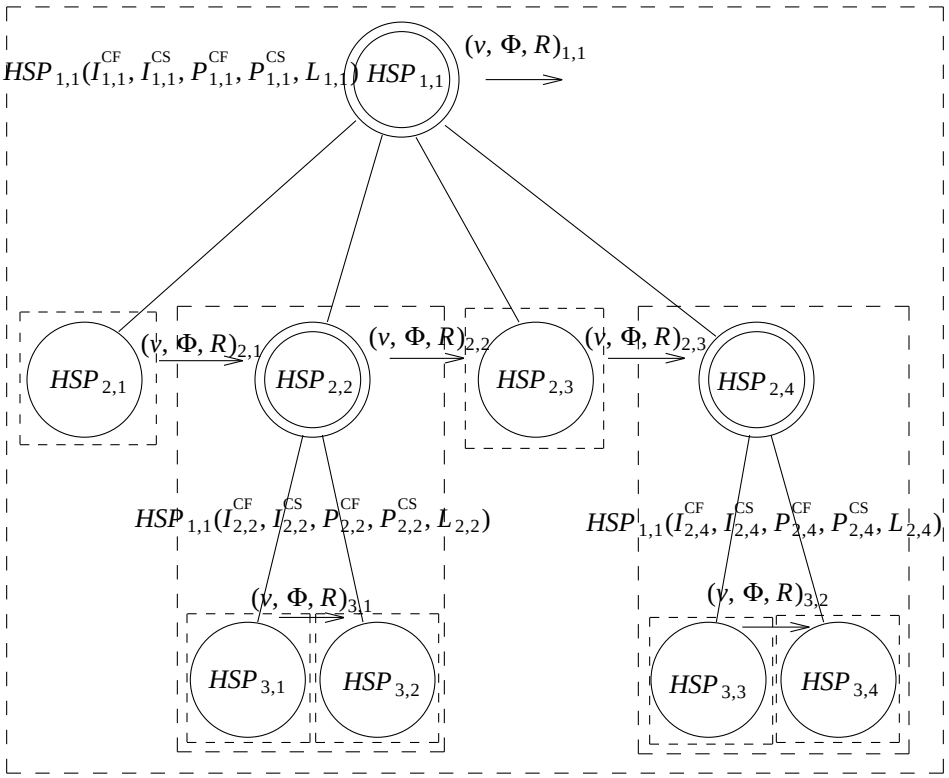


Figure 2.2. Hierarchy of calls generated when $HSP_{1,1}$ is invoked. (A dotted box indicates all the HSPs invoked when the root of the subtree is invoked. A horizontal arrow indicates the flow of results from one HSP to another.)

Formally, an HSP is defined as a mapping from a quintuplet to a triplet,

$$HSP: \quad (v, \Phi, R) \leftarrow HSP(I^{CF}, I^{CS}, P^{CF}, P^{CS}, L),$$

where I^{CF} and P^{CF} specify, respectively, the PI/CF and PD/CF grammars, I^{CS} and P^{CS} specify, respectively, the PI/CS and PD/CS components, and L is a finite set of initial conditions specifying the environment, such as the initial conditions of the resources and the problem parameters when the search is started. Note that T can be considered as part of the context-sensitive component of the HSP. However, we separate it from the context-sensitive component because we wish to distinguish the context-sensitive component from inputs supplied initially by users.

Each of the context-free grammars is defined by a 4-ary tuple (N^G, T^G, P^G, S^G), where N^G is a finite set of nonterminal symbols, T^G is a finite set of terminal symbols that are solution entities used in constructing a solution, P^G is a finite set of production rules, and S^G is the start symbol.

The PI/CF grammar defines the generation of child HSPs. For example, in Figure 2.2, the PI/CF grammar of $HSP_{1,1}$ ‘‘calls’’ the PI/CF grammars of $HSP_{2,1}$, $HSP_{2,2}$, $HSP_{2,3}$, and $HSP_{2,4}$.

The context-sensitive components include PD/CS elements that make decisions based on performance, such as the branching function, and those that schedule the search according to a predefined schedule, such as the depth of the lookahead horizon in the RTA* search. They also include PI/CS elements that schedule the search according to the resources available and past performance, such as the strategy used in MIDA* [103] that sets thresholds based on resources remaining and past performance.

An HSP returns a triplet (v, Φ, R), where v is the solution if available, Φ is a finite set of assessment values, such as quality of the solution and any performance estimates, and R is a specification of the resources remaining. The returned triplet can be used by a subsequent HSP as inputs. The arrows in Figure 2.2 indicate the flow of results from one HSP to another.

This modeling simplifies the development of new search algorithms and the support of new search applications. In developing a new search algorithm, only the problem-independent parts have to be added, and all existing applications can be used to test the new search algorithm. On the other hand, in developing a new search application, only the problem-dependent parts have to be enhanced, and all existing search algorithms can be applied and tested.

2.2.3. Illustrative examples

In this section, we model a number of familiar search algorithms using HSPs. These include a simple search (depth-first search), an iterative search (IDA*) [37], a resource-constrained search (RTA*) [42], and a resource-constrained iterative search (RTS) [8]. In each case, we model the primitive search and its meta-control as HSPs and describe the PI/CF, PI/CS, PD/CF, and PD/CS components and ternary tuple returned in each HSP. We use the traveling salesman problem (TSP) as the target application.

The depth-first search is as an example of a simple search. Figure 2.3 shows a depth-first search modeled using an HSP in solving a TSP instance. In this example, we have initial conditions that include the pruning mechanism, solution requirements, resource constraints, and inputs. The depth-first search is divided into four components according to

the problem dependence and context sensitivity. The returned triplet is not shown in the example.

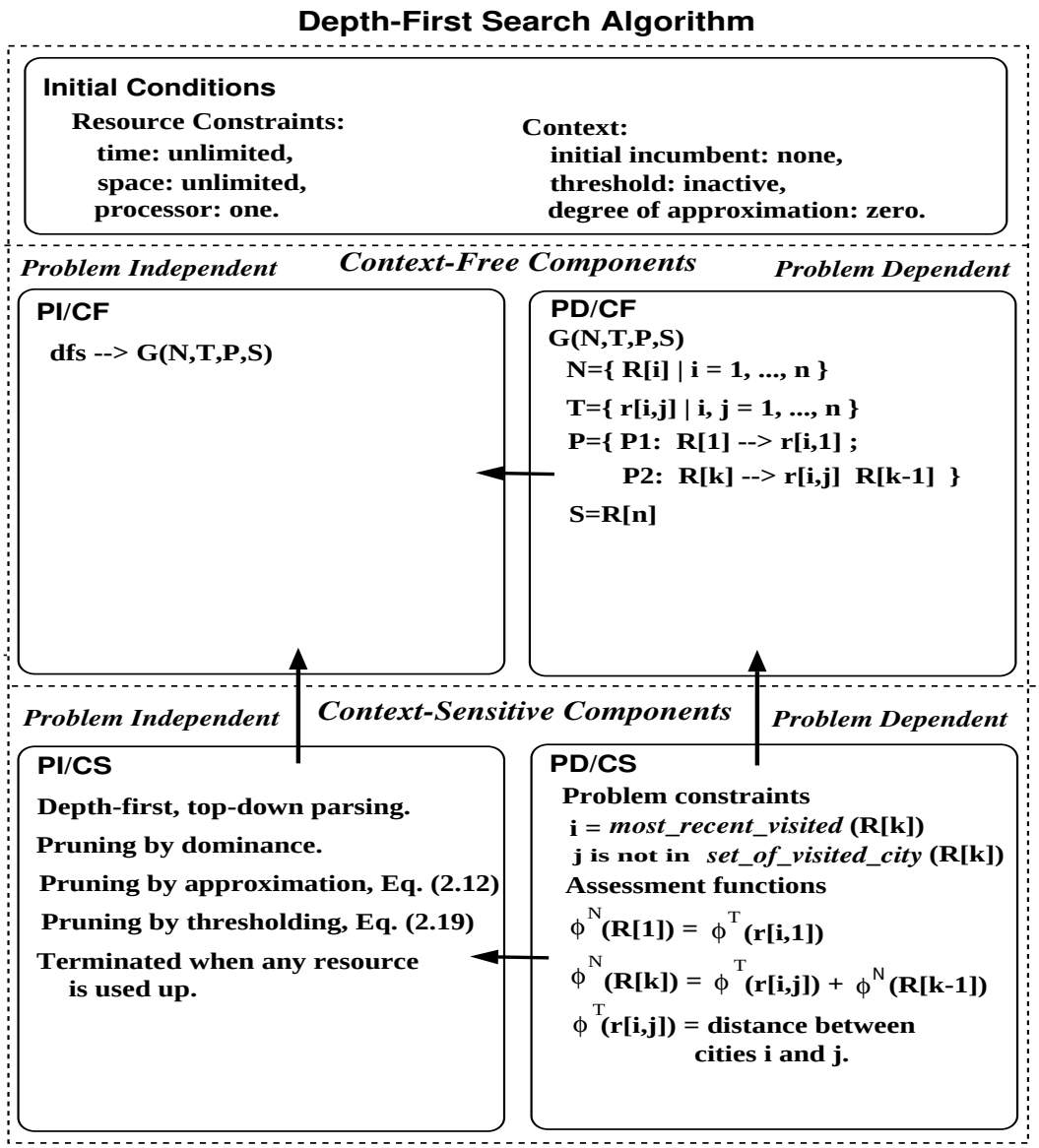


Figure 2.3. Example of showing an HSP of depth-first search in solving an *n*-city TSP instance.

We use IDA* as an example of iterative search. In IDA*, the PI/CF component of IDA*'s meta-control HSP consists of a grammar that defines a sequence of depth-first searches. In Figure 2.4 we show the HSP used to model the meta-control of IDA*. The pruning mechanism selects a threshold based on a value in the previous depth-first search such that the threshold is equal to the minimum of the lower bounds of children of nodes pruned by thresholding in the previous iteration. The primitive search, or depth-first, search, is similar to that shown in Figure 2.3, except for the initial conditions, which are shown in Figure 2.4.

We use the RTA* as an example to illustrate a time-constrained search. The RTA* itself works like a depth-first search, but at each search node, it looks ahead several levels deeper to obtain a better heuristic estimate for guiding the search. Therefore, during the depth-first search, each node is associated with another secondary depth-first search that corresponds to the lookahead. The specification of the secondary search is the same as that in the primary search except for one more pruning mechanism that prunes nodes by depth. This extra pruning mechanism defines the horizon of lookahead. Figure 2.5 shows the HSP for modeling RTA* in solving an *n*-city TSP. Only the initial conditions of the RTA* search and the lookahead search are shown because the rest is the the same as the depth-first search shown in Figure 2.3. Parameter *D* defines the depth of the lookahead horizon, and *B* is the average branching degree. The product of *D* and *B* defines the amount of time to be expended in the lookahead search.

Finally, we use RTS as an example to illustrate a resource-constrained iterative search. RTS [8], a search designed to be used in real-time applications, is a generalization of IDA* using approximation and/or thresholding. It is an iterative search that dynamically selects

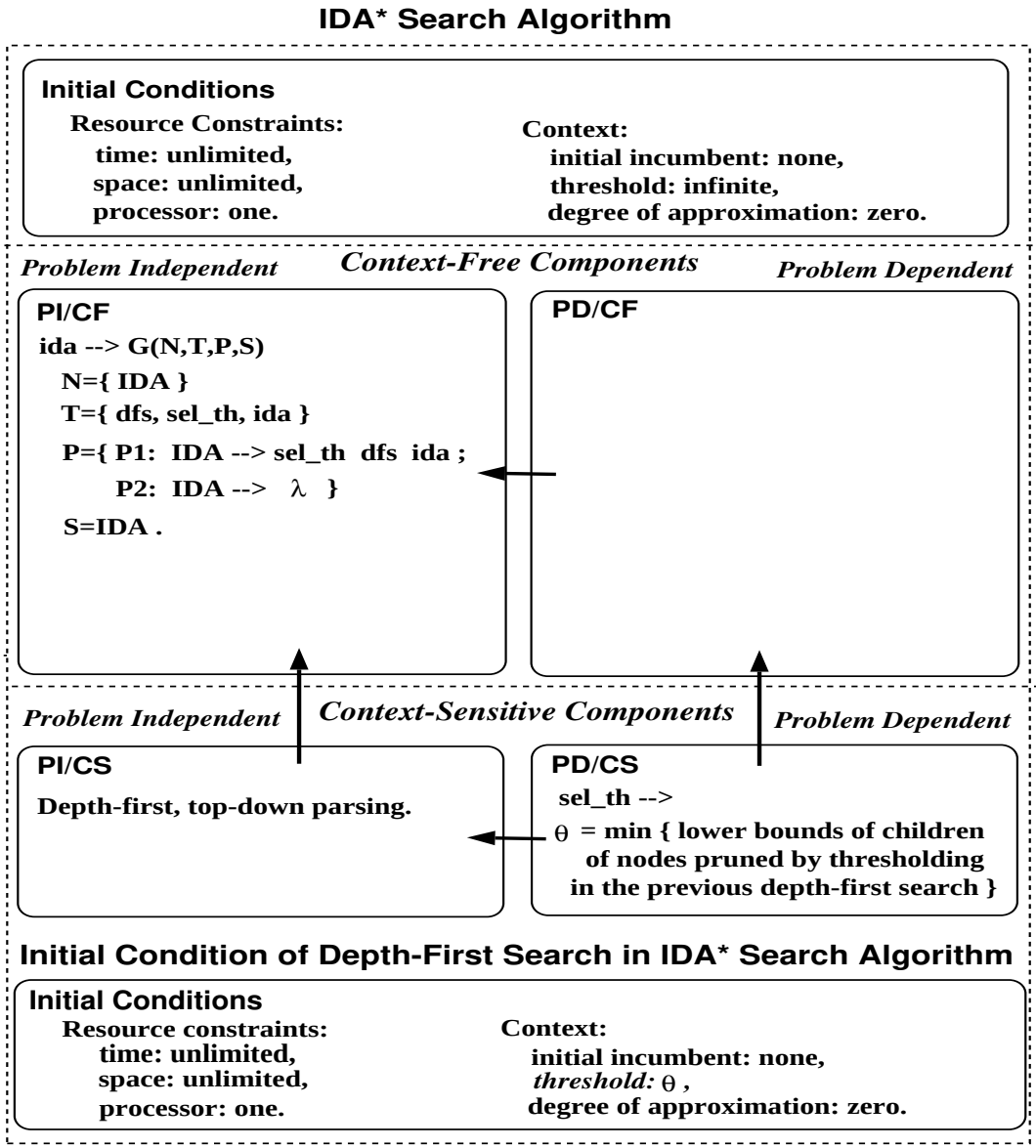


Figure 2.4. Example of showing an HSP of meta-control in IDA* for solving an *n*-city TSP instance.

---

approximation degrees and/or thresholds based on past performance and resources remaining. The following example shows the HSP model of using RTS to solve a TSP instance against a deadline. In this example, only the time constraint is considered because RTS uses bounded amount of memory. As in the formulation of IDA*, RTS is a sequence of

Initial Conditions of RTA* Search Algorithm

Initial Conditions

Resource Constraints:
time: unlimited,
space: unlimited,
processor: one.

Context:
initial incumbent: none,
threshold: inactive,
degree of approximation: zero.

Initial Conditions of Lookahead Search in RTA* Search Algorithm

Initial Conditions

Resource Constraints:
time: D*B units of logical time,
space: unlimited,
processor: one.

Context:
initial incumbent: none,
threshold: inactive,
degree of approximation: zero.

Figure 2.5. Example of showing initial conditions of HSPs of RTA* for solving a TSP.

GDFS; hence, its PI/CF grammar in the meta-control defines this sequence. A noted difference with other searches is that the selection of pruning mechanisms is problem-dependent and context-sensitive because the solution depends not only on resources remaining but also on past performance observed in the search. In Figure 2.6, we only show the initial conditions of the GDFS; the rest of the GDFS is similar to that shown in Figure 2.3, except for the selection function.

2.3. Real-Time Approximation Searches

Our research is to design efficient search algorithms that solve COPs in real time. In this section, we present background on real-time approximation, in particular, characteristics of real-time systems, anytime algorithms, and imprecise computation. They are helpful in understanding the issues in time-constrained searches.

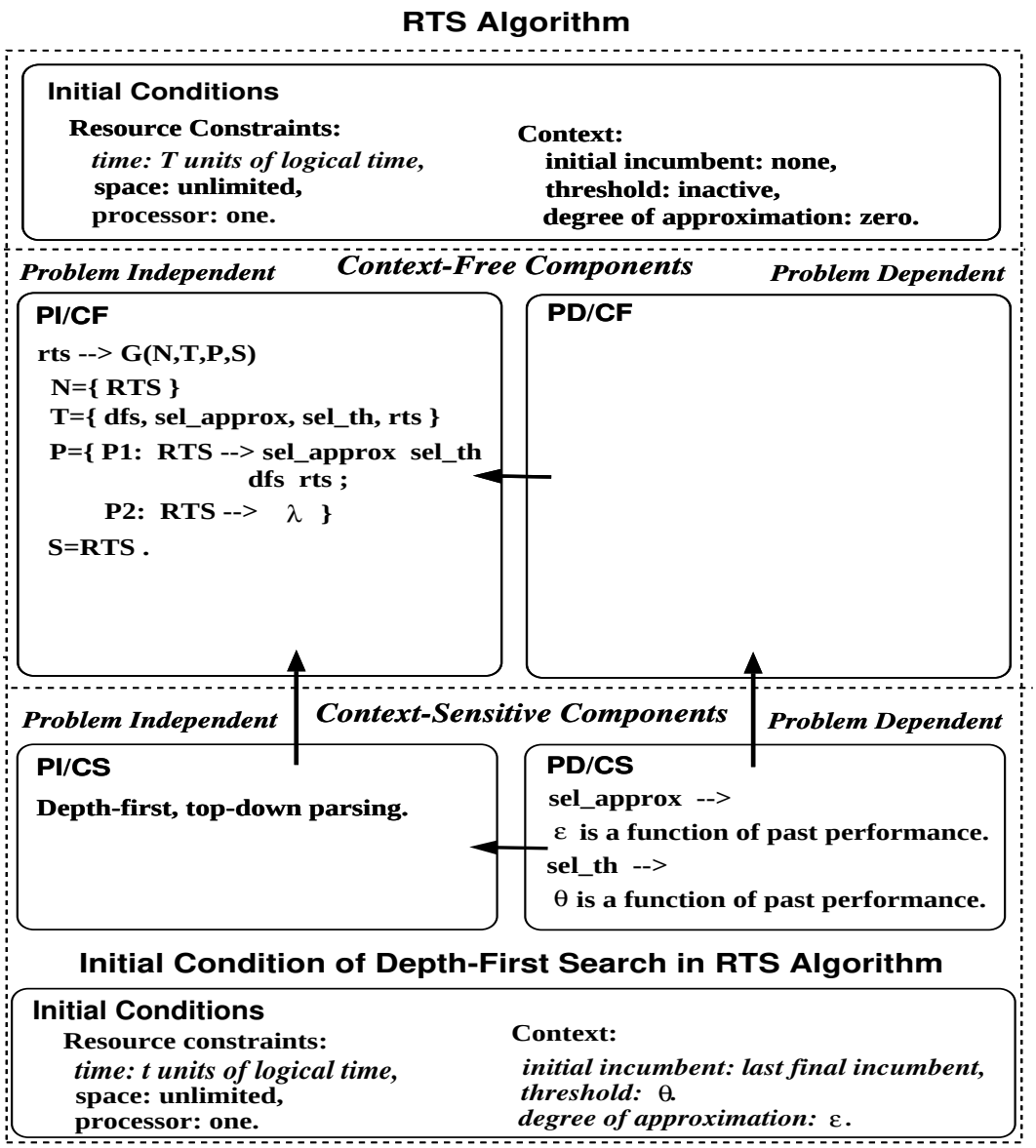


Figure 2.6. Example of showing an HSP of meta-control in RTS for solving a TSP.

2.3.1. Characteristics of real-time systems

There are many definitions of real-time systems and most of them involve timeliness, responsiveness, and degradation. *Hard* real-time systems are characterized by *severe* degradation when timing constraints are violated. On the other hand, *soft* real-time systems are characterized by *graceful* degradation.

Stankovic defined real-time computing as ‘‘*that type of computing where the correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced* [89]. Dodhiawala et al. suggested that a real-time system should have the following four aspects of real-time performance [15]: speed of task execution, responsiveness, timeliness, and graceful degradation. Besides these, memory management also plays an important role in a practical real-time system [34]. Therefore, in addition to meeting timing requirements, a real-time system has to make trade-offs among these five features [85]: speed, responsiveness, memory for data space, approximation, and graceful degradation, where speed is greatly dependent on the underlying hardware, but the other features are greatly dependent on the underlying scheduling algorithm. Most real-time systems differ in the trade-offs among these features.

Lesser et al. have identified three types of approximation to make these trade-offs [50]: approximation search, data approximation, and knowledge approximation. They also identified that reduction of solution quality can be along one or more of the three dimensions [50]: (a) completeness (some components of results may be ignored); (b) precision (some components may not be exact); and (c) certainty (some supporting evidence may be neglected). In this thesis, we investigate another dimension, namely, approximation, which means results may not be optimal.

2.3.2. Anytime algorithms and imprecise computation

In this section, we give background on anytime algorithms and imprecise computation, and discuss their relations.

Dean and Boddy proposed the concept of *anytime algorithms* in 1988 [12], which characterizes a class of algorithms that can be interrupted at any time and deliver the results, where the quality of the result is a function of time. The more time is available, the better the quality of the result achieved. In real-time computing, as we have indicated, there are two major issues: establishing a result (or solution) and refining the solution. A fundamental assumption behind anytime algorithms is that a solution can be established easily. The major issue in anytime algorithms involves only refining the solution because it already has a solution and can deliver this solution at any time when it is interrupted.

Lin et al. [56] and Lin et al. [58] proposed the concept of *imprecise computation*. Traditional schedulers for real-time systems assume that timing attributes of all jobs are known ahead of time. The timing attributes include release time, starting time, maximum execution time, and deadline. A feasible schedule is one in which all jobs can meet their respective deadlines. However, a feasible schedule may not be possible when the maximum times for executing jobs are large, when the number of jobs is large, when the deadlines are tight, or when computing resources are limited. In this case, partial execution of these jobs become necessary. Imprecise computation is to find a feasible schedule, assuming that partial execution of these jobs is allowed. To allow partial execution, a job is decomposed into two types of tasks: *mandatory tasks* and *optional tasks* [56, 58, 59]. Mandatory tasks must be executed, while optional tasks need not be executed but their executions can improve the quality of the result. Therefore, partial execution means that all mandatory tasks and a subset of optional tasks are executed. The feasibility of a schedule in imprecise computation is relaxed to that the mandatory tasks of all jobs can meet their timing requirements.

Imprecise computation can find an optimal feasible schedule, where the objective is defined as the overall quality of partial execution across the entire set of jobs. The fundamental assumptions include (a) the timing attributes of all jobs and all tasks are known, (b) the quality profiles of all jobs can be characterized, and (c) the objective that combines the quality of possibly different measures and scales is well-defined.

To allocate time dynamically to multiple jobs while preemption is allowed, progress of job execution must be known or estimated at run time. In many cases, especially in solving COPs, quality profiles are unknown. This is part of the reason that in this thesis we emphasize the importance of run-time estimation of solution quality.

An anytime algorithm executes a singe job against a time constraint. As is indicated previously, it has a fundamental assumption that a solution can be established easily. Establishing a solution corresponds to executing mandatory tasks in imprecise computation. Refining the solution corresponds to executing optional tasks. Note that imprecise computation also has a similar implicit assumption that execution of mandatory tasks can meet their respective timing requirements.

Imprecise computation usually assumes prior knowledge on execution profiles of all jobs in order to schedule *multiple* jobs. In contrast, an anytime algorithm does not have to know the execution profile because it executes *one* job only. However, with an estimated execution profile, an anytime algorithm can perform better [82].

Imprecise-computation schedulers and anytime algorithms usually assume a monotonic trend of the quality profile; that is, the solution becomes better if we have more time. For example, linear, convex, and concave profiles are monotonic. This property is important in

imprecise computation in the sense that the profiles may not be known but the scheduler can estimate their trends. Nonmonotonic profiles make prediction difficult in that overshooting and undershooting may happen.

There is a distinction between time constraint (imposed in an anytime algorithm) and deadline (imposed in imprecise computation). Imprecise computation schedules jobs such that all mandatory tasks of these jobs meet their respective timing requirements, while an anytime algorithm executes a job within a time constraint. A time constraint defines the maximum time a job can use, and the job can expend some of the time or all of it. A deadline defines the time when a job must be completed. A deadline can be translated to a time constraint, but the job may not be able to expend all the time defined by the translated time constraint because time usage is dependent on other jobs competing with the same computing resources. With good translation from deadlines to time constraints, imprecise computation and anytime algorithms can be combined in a two-level hierarchy in which an imprecise-computation scheduler schedules multiple jobs at the higher level, and anytime algorithms are responsible for executing these jobs at the lower level.

In this thesis, the concept of anytime algorithms is useful because the search algorithms designed are expected to deliver, if possible, a solution when time runs out. The techniques used in imprecise computation is not useful in our research because we are dealing with a single job; however, the distinction between mandatory and optional tasks in imprecise computation is useful in that mandatory tasks of the search algorithms designed are to find a feasible solution and their optional tasks are to refine the incumbent.

2.4. Survey of Resource-Constrained Search Algorithms

Resource constraints include time, memory space, and number of processors. They can be enforced individually or in conjunction. These constraints can be hard or soft.

When the number of processors is constrained, we can exercise parallel search. Parallel search has been extensively studied, including general concepts [44, 97, 102, 109], anomalies in parallel search [53, 97], concurrent access to active lists [33], parallel game tree search [16, 17, 62, 63], and computer models for parallel search [52, 68, 93-96, 99]. In this thesis, we do not study issues related to parallel search; only one processor is assumed.

In this section, we survey time-constrained and memory-constrained searches. A simple classification of searches is shown in Figure 2.7. Resource constraints can be posted in two levels of the search process: (a) evaluation of guidance value and (b) the entire search process. In this research, we design efficient search algorithms that have resource constraints posted on the entire search process and that give a bound on solution quality.

In this thesis, we are interested in general search algorithms that use a minimal amount of domain knowledge; namely, we are interested in weak methods [48]. There are real-time systems that utilize domain knowledge for approximation and dominance in their search strategies, and most of them are not designed to give a bound on solution quality [50, 74, 87].

In order to give a global picture, besides surveying related studies in the case we are interested in, we also concisely survey studies in other cases. In this section, we discuss searches with resource-constrained guidance evaluation and survey time-constrained and memory-constrained searches. Finally, we discuss some basic methods used in real-time systems that utilize domain knowledge for approximation.

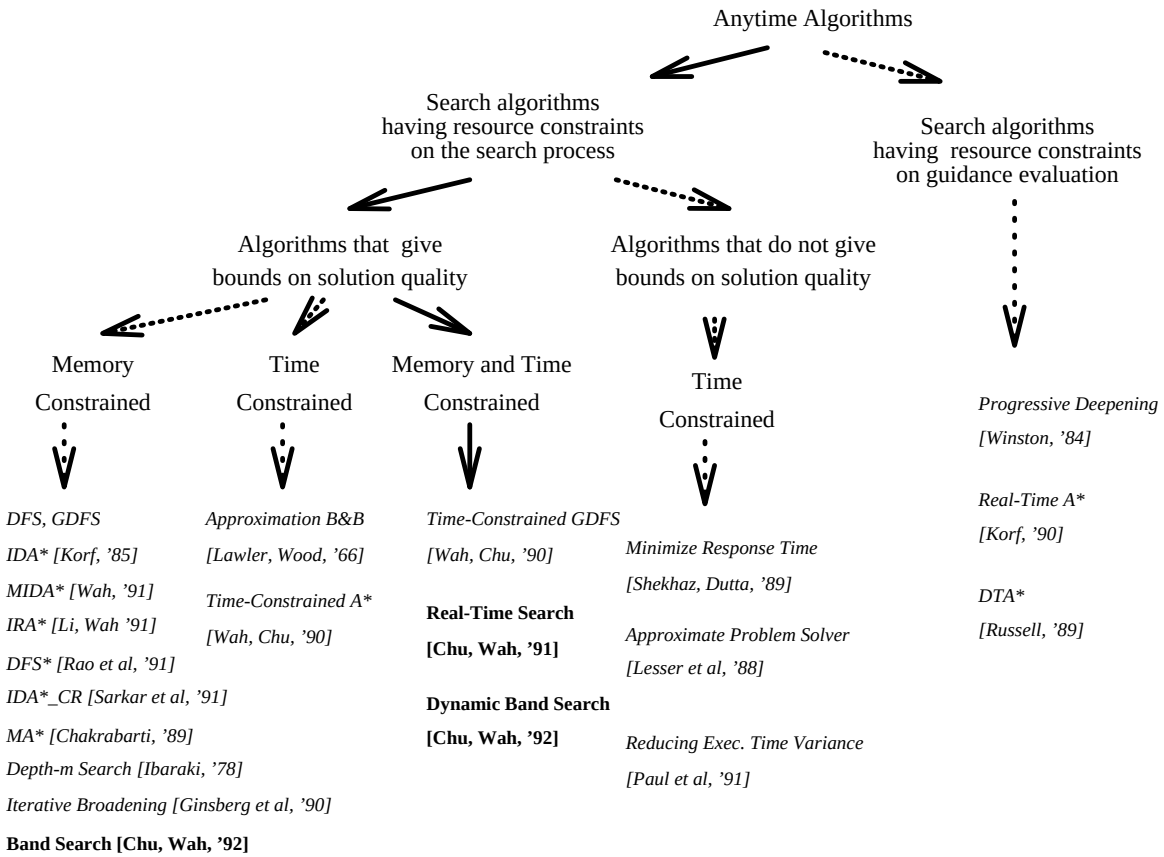


Figure 2.7. Classification of time-constrained and memory-constrained searches.

2.4.1. Search algorithms with resource-constrained guidance evaluation

Resource constraints in game-tree searches are often posted in the level of evaluating guidance values. Examples include Winston's time-constrained progressive deepening [108], the RTA* algorithm [40-42], and the DTA* algorithm [81, 107].

Winston described a time-constrained progressive deepening method for game-tree search [108]. The idea is to analyze (search) situations to depth 1 of the search tree, then to depth 2, then to depth 3, and so on. This is continued until time runs out. Progressive deepening is actually a breadth-first search. Note that progressive deepening is different from iterative deepening, in that the latter involves re-expansions of search nodes.

The RTA* algorithm is designed to interleave planning and execution [40-42]. It makes locally optimal decisions and guarantees to find a solution if such a solution exists. The RTA* algorithm posts the time constraint in the evaluation of its lookahead estimation in hoping that the estimation can guide the search to a promising subtree. During its evaluation, it activates a depth-first search to refine lookahead estimation. RTA* can find a solution if a solution is reachable from any state such as in the puzzle problem, and will propose a move that is along a locally optimal path.

DTA* is similar to RTA* except that it is a hardest-first strategy instead of depth-first strategy [81, 107]. It always expands the current best leaf from the current move and stops when the expected net value (defined as the gain minus the cost) becomes smaller than a value that is calculated at run time.

In short, time-constrained progressive deepening is breadth-first; RTA* is depth-first to a prescribed depth; and DTA* is similar to best-first. These algorithms are not useful in our research because the time constraint is imposed over the entire search process instead of guidance evaluation.

2.4.2. Time-constrained search algorithms

In this thesis, we are interested in the case in which the time constraint is posted over the entire search. Examples of posting a hard time constraint on the entire search include Lawler and Wood's algorithm [49], Mutchler's optimal allocation [70], and our previous work TCA* [100] and TCGD [101]. These examples are general search algorithms that use a minimal amount of domain knowledge and are usually referred to weak methods [48].

Lawler and Wood proposed a time-constrained approximation search in 1966 [49]. Their algorithm is a stepwise refinement search and works as follows. Given a time constraint T, it schedules $T/2$ units of time to solve an optimal solution. If it fails, then it schedules another $T/4$ units of time to find an approximate solution of approximation degree 0.05. After k failures, we schedule $T/2^k$ units of time to solve an approximate solution of approximation degree $0.05 \times k$. This process is repeated until the time constraint is violated. One major drawback in this schedule is that hard problems (finding more accurate solutions) are explored first and expend most of the time, while easy problems (finding less accurate solutions) are explored later and use only a small portion of the time. In our research, we are interested in search algorithms that have stepwise refinements from finding approximate solutions first to finding more accurate solutions after less accurate solutions are found.

Mutchler proposed a probabilistically optimal allocation of very limited search moves for finding a minimum-cost path from the root to a leaf in a special binary search tree of finite depth [70]. The arcs of this binary tree are assigned costs randomly as 1 with probability p or as 0 with probability 1-p. Mutchler developed two theorems: the bad-news theorem and the good-news theorem. The bad-news theorem says that there exists a case that the best-first (or least-cost-first) strategy is not optimal regardless of the time constraint. The good-news theorem says that the best-first strategy is optimal when $p \leq 0.5$ and suboptimal when $p > 0.5$. Note that the bad-news theorem does not contradict the optimality of A* because the goal of this search is to *find* an optimal solution by A*, where solving an optimal solution consists of both finding it and proving its optimality. This technique cannot be applied to our research because the search trees in general cannot be modeled by the binary tree with costs of 0 and 1.

In our previous work, we used approximation in A* search called *time-constrained A** (TCA*) [100] and in guided depth-first search called *time-constrained guided depth-first search* (TCGD) [101]. The basic idea in TCA* and TCGD is inspired from IDA* [37], and they exercise approximation iteratively and progressively to search deeper in the search tree in hoping that a good solution can be found quickly. In this thesis, we design Real-Time Search that enhances TCGD.

2.4.3. Memory-constrained search algorithms

In this thesis, we are interested in search algorithms that operate under a soft memory constraint. There are several successful attempts to memory-constrained search. The basic approaches can be classified into one of the following three cases. In our discussion, we assume that the memory constraint is at least more than that is required to store the shortest path from the root to an optimal solution.

(a) *Retraction and Re-expansion.* This approach is to encode (i.e., retract) search nodes that cannot be stored in memory into ancestor nodes for future exploration. One example is MA* [5, 6]. The MA* algorithm simulates A* under a hard memory constraint. It begins with an A* search. When memory space is running out, it retracts search nodes into their ancestor nodes in order to reclaim (deallocate) the memory occupied by the retracted nodes, and uses the memory newly deallocated to expand the best active node. The major drawbacks are that retractions and re-expansions can incur prohibitively large overheads and that sorting in best-first search is also time consuming. The techniques of retraction and re-expansion used in MA* are not useful in our research because we are interested in search algorithms that require only a limited amount of memory and operate under soft memory constraints.

(b) *Iterative Refinement.* This approach is to refine iteratively the objective of search. Typical examples include IDA* [37] and its variants such as MIDA* [103], IDA*_CR [84], DFS* [92], IRA* [55], and MREC [86]. All of these algorithms except MREC have been described earlier. The MREC algorithm begins with an IDA* search and stores all expanded nodes in memory without redundancy. When memory space is running out, it stops storing search nodes and switches to an ordinary IDA*. After the switch, it does not have to re-expand the search nodes stored in memory and can directly use the information in these stored search nodes. Strictly speaking, only IDA* and MREC can operate under hard memory constraints, while the others operate under soft memory constraints, since they may overshoot to a threshold beyond the value of the optimal solution. The concept of iterative deepening is useful in our research because iterative deepening can avoid being trapped in a poor subtree, and corresponds to stepwise refinement from approximation to accuracy. Also, we can collect information of search progress in these iterations. However, their techniques for setting thresholds are not useful in our research because they do not adjust thresholds based on the remaining time.

(c) *Scope Limiting.* This approach is used to limit the active scope of search. Typical examples include depth-m search [30], iterative broadening [20], and block depth-first search [61].

The depth-m search algorithm simulates best-first search by using a predictable amount of memory [30]. It maintains a priority queue and an ordinary list, where the former contains at most *m* search nodes. It only expands the best node in the priority queue, and all its children are inserted in the priority queue. When the priority queue has more than *m* nodes,

the tail of the priority queue is moved to the front of the ordinary list so that the priority queue has exactly *m* nodes. When the priority queue has less than *m* nodes, it moves the nodes in the front of the ordinary list to the priority queue and sorts the priority queue again.

The depth-m algorithm uses a bounded amount of memory, namely, $O(D^{m+1})$, where *D* is the maximum depth of the search tree. This bound is large because *m* is in the exponent.

The technique of limiting memory space used in the depth-m search is not very useful in our research because the bound on memory space is still too large. We are interested in algorithms whose memory requirements are linear with that of GDFS.

The iterative broadening algorithm starts a search with a branching degree equal to one (ignoring other branches in the search tree) and iteratively increases the branching degree in order to search progressively the entire search space until a goal node is found [20]. The execution times of searches grow exponentially with the iteration count. This algorithm has a flavor similar to IDA*. It performs well in a search tree with a uniform branching degree and in solving decision problems. The technique of iterative broadening is not useful in our research because the search algorithms we design are used to solve COPs rather than decision problems. Moreover, the search trees representing COPs may not have a uniform branching degree.

The block depth-first search algorithm works by grouping all search nodes with equal lower bounds at the same depth into a macro node and by performing guided depth-first search on these macro nodes [61]. It uses a secondary key to resolve ties inside a macro node. It performs well for some discrete-valued problems such as the puzzle problem.

However, it may not perform well for other discrete-valued problems in which many search nodes at the same depth have the same lower bounds, such as the vertex cover problem. In the latter case, it performs similarly to a breadth-first search, which implies that its memory requirement may grow exponentially with the problem size. Also, it becomes a guided depth-first search when solving continuous-valued problems in which search nodes have distinct lower bounds, and all blocks (i.e., macro nodes) only contain a node. The techniques used in a block depth-first search is not very useful in our research for the following reasons: (a) the memory requirement may grow exponentially with the problem size when solving some discrete-valued problems, and (b) a block depth-first search becomes a guided depth-first search when solving continuous-valued problems.

In this research, the search algorithms we design operate under soft memory constraints. The concept of iterative refinement in other search algorithms is useful; however, their techniques for setting thresholds cannot be applied directly because they do not adjust thresholds based on the remaining time. Further, the concept of scope limiting in depth-m and block depth-first searches is useful; however, memory requirements in these algorithms are generally too high.

2.4.4. Searches without bounds on solution quality

There are real-time systems that utilize domain knowledge for approximation and dominance in their search strategies, and most of them are not designed to give a bound on solution quality because they can exercise domain knowledge to have a qualitative assessment of solution quality.

Laffey et al. have a comprehensive survey on real-time AI systems in aerospace applications, communication applications, medical applications, process control applications, and robotics applications, where these systems are evaluated in terms of attributes such as speed, response, temporal reasoning, asynchronous input, interrupt handling, continuous operation, noisy data, and focus of attention [47].

Real-time knowledge-based systems and expert systems in general involve planning and execution of actions. For example, Shekhar and Dutta [87] combined search and execution into planning, and optimized the total time spent in both. Approximation search, data approximation, and knowledge approximation are exercised in planning. These approximations greatly depend on the application domains because they extensively exercise domain knowledge. For example, Lesser et al. proposed incomplete-solution search for vehicle guidance [50]. An incomplete-solution search neglects certain unimportant elements of calculations; specifically, it eliminates corroborating support and competing interpretation, approximates data and knowledge, processes events incompletely, and clusters processing.

Most existing real-time schedulers schedule real-time jobs based on their worst-case execution times such that they can avoid the situation when timing properties of jobs change. Paul et al. reduced problem-solving variance (namely, the worst-case performance) by ignoring certain unimportant elements to improve predictability [74]. Improving prediction is very important to avoid misleading the scheduler because the scheduler may allocate too much time to a job that has large worst-case execution time but small average execution time. The reduction of execution time variance also requires extensive domain knowledge.

In our research, the search algorithms designed are general algorithms that do not require extensive domain knowledge. Moreover, we are interested in algorithms that give

bounds on solution quality. As a result, the techniques used in other methods cannot be applied directly in our research.

2.5. Summary

In this chapter, we describe fundamentals of search and modeling of search algorithms by using the hierarchical search process as a building block. We also discuss the characteristics of real-time systems, anytime algorithms and imprecise computation. Finally, we survey related studies in time-constrained and memory-constrained search algorithms and conclude that existing time-constrained and memory-constrained search algorithms are not directly applicable to our research.

CHAPTER 3.

REAL-TIME SEARCH

In this chapter, we develop a search algorithm called *Real-Time Search* (RTS) that solves NP-hard dense COPs under hard real-time constraints. Our algorithm schedules a sequence of primitive searches, each searching for a more accurate solution than the previous one. By selecting properly the degree of approximation and the threshold in each primitive search, RTS finds solutions very close to the best solution that can be found under the given time constraint.

We are interested in a search algorithm that *finds the best feasible solution in a given time constraint*. This objective is generally not achievable since there is insufficient run-time information before the search begins. We use an alternative heuristic objective for solving *the solution with the best ascertained approximation degree*. Note that the new objective may not be consistent with the original objective in the following sense. To obtain better feasible solutions, the search process has to focus on improving the best solution found so far. On the other hand, to obtain good solutions with small approximation degrees, one has to refine both the minimum lower bound and feasible solution so that the gap in between is smaller. This means that time available may have to be spent in achieving both of these intermediate objectives.

For the dense COPs studied, we empirically observe that *the execution time increases exponentially as the degree of approximation achieved decreases linearly*, though anomalies may happen. Figure 3.1 shows an ideal execution profile. RTS schedules a sequence of GDFSs, each searching for a better solution than the previous one. These GDFSs expends

exponentially increasing amount of time. By selecting thresholds and degrees of approximation properly, RTS can perform very well.

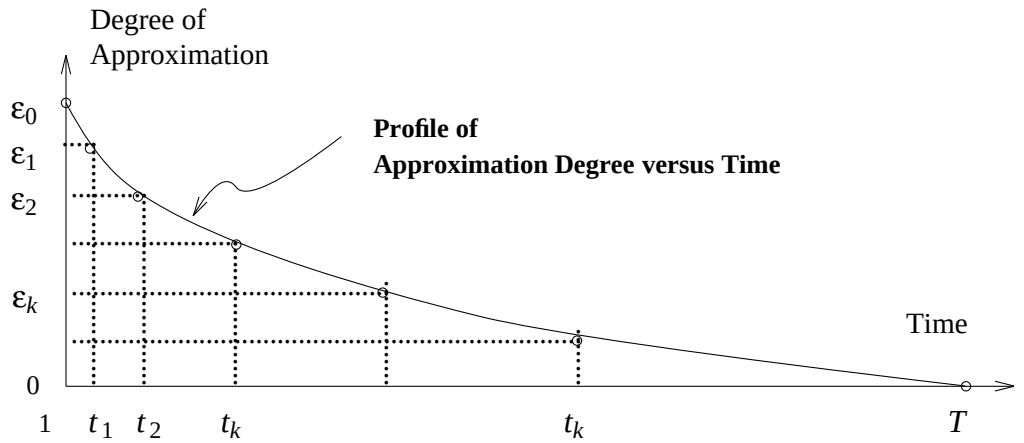


Figure 3.1. An ideal execution profile of approximation degree versus time.

In the RTS algorithm, we employ guided depth-first search as the primitive search since it performs well in the class of target problems. We study the behavior of RTS with static and dynamic strategies in selecting degrees of approximation and thresholds. These strategies are based on a performance model that is justified empirically.

Six versions of the RTS algorithm are formulated, experimented, and analyzed, where each version uses a particular strategy. The performance of RTS is evaluated by simulations using the symmetric traveling-salesman, knapsack, production planning, and vertex cover problems. Our results show that the RTS algorithm finds better solutions than a guided depth-first search and Lawler and Wood's approximation branch-and-bound search (discussed in Chapter 2) in the time allowed.

3.1. Preliminary

In this section, we describe the basic idea behind our real-time search and explain some terminologies. Notations used in this chapter are summarized as follows.

(a) Time is denoted as t or τ, and T is the time constraint.

(b) A degree of approximation is denoted as ε.

(c) A threshold is denoted as θ.

(d) A solution value is denoted as z, and the optimal solution value is z^*.

(e) Predicted values (incumbent, threshold, and degree of approximation) are represented by symbols with hats.

In most real-time systems, solution quality compromises with time in one or more dimensions: *completeness* (some solution components are neglected), *precision* (some solution components are not in their exact forms), *certainty* (confidence on some solution components is not full), and *approximation* (solution is not optimal) [50]. In solving dense COPs, only approximation is relevant and important because feasible solutions can be obtained easily.

The common procedure of a real-time algorithm consists of two parts [59]: (a) finding a feasible solution as fast as possible; and (b) refining the feasible solution as fast as possible. For dense COPs, a solution is easy to find by using a polynomial-time approximation algorithm that is used for computing the upper bound. Hence, we only focus on the second part in our research.

The RTS algorithm falls in the class of the anytime algorithms [12]. An anytime algorithm can be stopped at any time and returns a solution when time is used up, and the quality of this solution is a function of execution time.

To improve the RTS algorithm, two dimensions of improvement are possible: improving the schedule of approximation and thresholding, and improving the prediction of execution profiles.

(a) *Improving the schedule of approximation and thresholding*. RTS uses heuristic procedures to select degrees of approximation and thresholds, and these procedures have certain parameters that affect the performance of RTS. Analysis can be done in optimizing the performance of RTS by choosing appropriate parameters.

(b) *Improving the prediction of execution profiles*. RTS uses thresholding to prune search nodes as in other iterative searches with thresholding (IDA* [37], MIDA* [103], and IDA*_CR [84]). Under limited time, RTS schedules thresholds based on the remaining time. However, the previous algorithms using thresholding are designed to solve search problems without a time constraint and do not schedule thresholds based on the remaining time.

Recall that there are two degrees of approximation: EAD and AAD. The EAD indicates a relative deviation (or error) from the optimal solution value. However, it is impractical to use because the optimal solution value is unknown until we solve an optimal solution. During the search, we use, instead, AAD, which is a function of the incumbent and the least lower bound. To tighten the AAD, we need to improve the least lower bound as well as the incumbent.

One problem with using AAD is that it may not be consistent with EAD. This inconsistency occurs when a smaller lower bound results in a larger AAD, and when a larger lower bound results in a smaller AAD, assuming the incumbent is fixed. The advantage in using the AAD is that it can be computed at run time, while the EAD cannot be computed until the optimal solution value is known. In this thesis, AAD is used as a heuristic measure of EAD. Moreover, we use AAD to guide a search and use both AAD and EAD to evaluate the search.

Both AAD and EAD are important in real-time search: EAD evaluates the solution quality, and AAD serves as an estimate of EAD before the optimal solution is found. If we have neither AAD nor EAD, then we cannot predict the solution quality. Moreover, a real-time scheduler can make its scheduling decisions using AAD as a performance estimate.

In terms of both EAD and AAD, the best algorithm that solves dense COPs under a given time constraint is OPTA*, which ‘‘knows’’ the optimal solution value before the search begins (that is, the incumbent is an optimal solution throughout the search) and behaves like A* during the search. OPTA* is the best in terms of EAD because its EAD is zero all the time. OPTA* also has the best AAD because the minimum lower bound in A* is always no less than that of any other search algorithm at any particular logical time; consequently, AAD of OPTA* is an absolute lower bound of those of other search algorithms. However, OPTA* is obviously not practical and can only serve as a benchmark for comparison because it requires the knowledge of the optimal solution value before the search begins.

Approximation and thresholding are related to pruning. A larger approximation degree and a smaller threshold will generally cause more nodes to be pruned and the search completed in less time. Recall that we have defined rules used in pruning by approximation and pruning by thresholding in Eqs. (2.3) and (2.4).

In the following, we use the knapsack problem as a running example to illustrate the operations of approximation and thresholding pruning. The knapsack problem is to find a maximal-profit set of objects such that their total weight does not exceed a given bound. Given n objects and a weight bound W, the objective of the knapsack problem is to

$$\underset{S}{maximize} \sum_{i member S} p_i \quad \text{such that} \quad \sum_{i member S} w_i \le W ,$$

where p_i and w_i are profit and weight, respectively, of object i. Note that the knapsack problem is a maximization problem and can be transformed into a minimization problem by either negating the objective or by solving its dual problem. Since we study approximation in terms of positive solution values and bounds in this thesis, we choose to solve the dual problem. The objective of the dual problem is

$$\underset{S}{minimize} \sum_{i nomem S} p_i \quad \text{such that} \quad \sum_{i member S} w_i \le W .$$

Note that the objective of the dual problem is the total profit of unselected objects, which can be treated as a penalty; as a result, this dual problem is to minimize the penalty. The lower bound in this dual problem is evaluated by finding the penalty of packing into the remaining space of the sack a virtual object whose profit-to-weight ratio is equal to the maximum of those of unselected objects. The upper bound is computed by a steepest-descent hill-climbing algorithm.

Consider a ten-object knapsack problem whose parameters are listed in Table 3.1 (this knapsack problem is generated by the software tool ISE by using a random seed of 1). The search tree representing this problem is shown in Figure 3.2, where the lower bound *x* and the upper bound *y* of a search node are labeled *x*,*y*, respectively, and the number labeling an edge indicates selection (+) or no selection (-) of an object into the sack. This search tree has a branching degree of 2 because an object considered can be selected or not selected to be put into the sack. The lower bound, upper bound, and AAD of the root node are 2663, 2895, and 0.087, respectively. For reference, the optimal solution value is 2799, which is obtained by using a best-first search that expands 46 nodes.

Assuming a threshold of 2716, then only nodes 1, 2, and 4 are expanded. Next, assuming AAD to be 0.065, pruning by approximation will prune search nodes with lower bounds larger than or equal to $z/(1+0.065)$, where z is updated whenever a better solution appears. In the beginning, the root node is generated, and the incumbent value is 2895. As a result, all search nodes with lower bounds no less than 2718.3 (=2895/1.065) are pruned. Nodes 2 and 3 can be generated by expanding the root node, and the incumbent becomes 2799 after node 3 is expanded. The pruning threshold becomes 2628.2 (=2799/1.065) accordingly. The current active nodes 2 and 3 are then pruned because their lower bounds are greater than the pruning threshold.

It is interesting to note that the upper bound of node 3 is equal to the optimal solution value. Without evaluating upper bounds, a GDFS may have to explore the entire left subtree before it can find an optimal solution that is in the right subtree.

Table 3.1. Profits and weights of objects in a 10-object knapsack problem that is generated with random seed of 1.

Object	Profit	Weight	Object	Profit	Weight
1	559	514	6	100	84
2	391	309	7	329	278
3	1029	948	8	1247	984
4	782	703	9	1013	766
5	525	495	10	1067	768

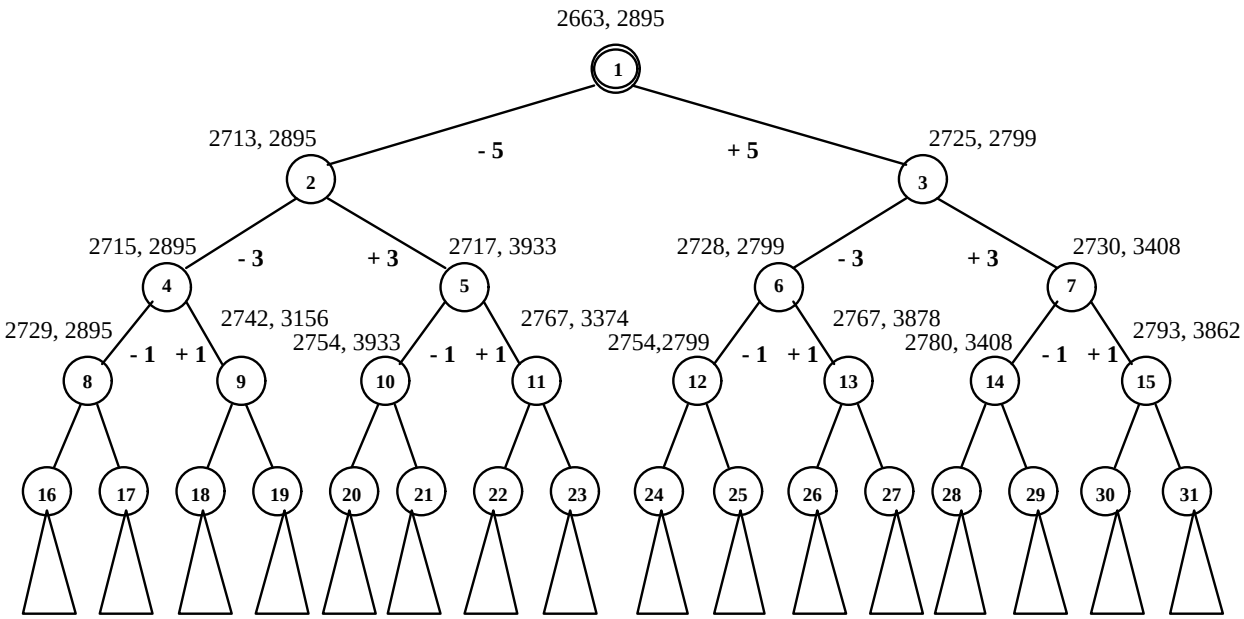


Figure 3.2. A simple search tree for illustrating approximation and thresholding as well as for illustrating various strategies used in the RTS algorithm, where the lower bound x and the upper bound y of a search node are labeled x,y, respectively, and the number labeling an edge indicates selection (+) or no selection (-) of an object into the sack.

In the following, we discuss the interaction between approximation and thresholding.

Consider a GDFS with initial AAD ε, threshold θ, and time constraint τ. An ε-pruning threshold $p_\varepsilon(\varepsilon, \theta, t)$ at time t is defined as the value $z(\varepsilon, \theta, t)/(1+\varepsilon)$. A θ-pruning threshold $p_\theta(\varepsilon, \theta, t)$ at time t is equal to θ. Note that the ε-pruning threshold is nonincreasing, because

the incumbent value is nonincreasing when ε is fixed. Both pruning thresholds delimit the search space so that all nodes with lower bounds no less than the minimum of both pruning values are pruned.

An aggregate pruning threshold $p_{\varepsilon,\theta}(\varepsilon, \theta, t)$ is the minimum of the ε-pruning and θ-pruning thresholds, namely,

$$p_{\varepsilon,\theta}(\varepsilon, \theta, t) = \min \{ p_{\varepsilon}(\varepsilon, \theta, t), p_{\theta}(\varepsilon, \theta, t) \} \quad (3.1)$$

Note that the aggregate pruning threshold is nonincreasing because the ε-pruning threshold is nonincreasing when the θ-pruning threshold is constant. Also note that $p_{\varepsilon}(\varepsilon, \theta, t) \leq z^{*}$ and $p_{\varepsilon}(\varepsilon, \theta, t) \leq p_{\varepsilon}(\varepsilon, \theta, 0)$. The relation between z^{*} and $p_{\varepsilon}(\varepsilon, \theta, 0)$ is not clear because the former can be greater than, equal to, or less than the latter. The final AAD achieved can be computed as

$$\varepsilon_{GDFS(\varepsilon, \theta, t)} = \frac{z(\varepsilon, \theta, t)}{p_{\varepsilon,\theta}(\varepsilon, \theta, t)} - 1 . \quad (3.2)$$

In terms of the ε-pruning and θ-pruning thresholds, the interaction between approximation and thresholding can be in one of the following three cases, as shown in Figure 3.3.

(a) $\theta > p_{\varepsilon}(\varepsilon, \theta, 0)$

In this case, approximation dominates thresholding because all search nodes pruned by thresholding are also pruned by approximation. The search degenerates into one using approximation alone.

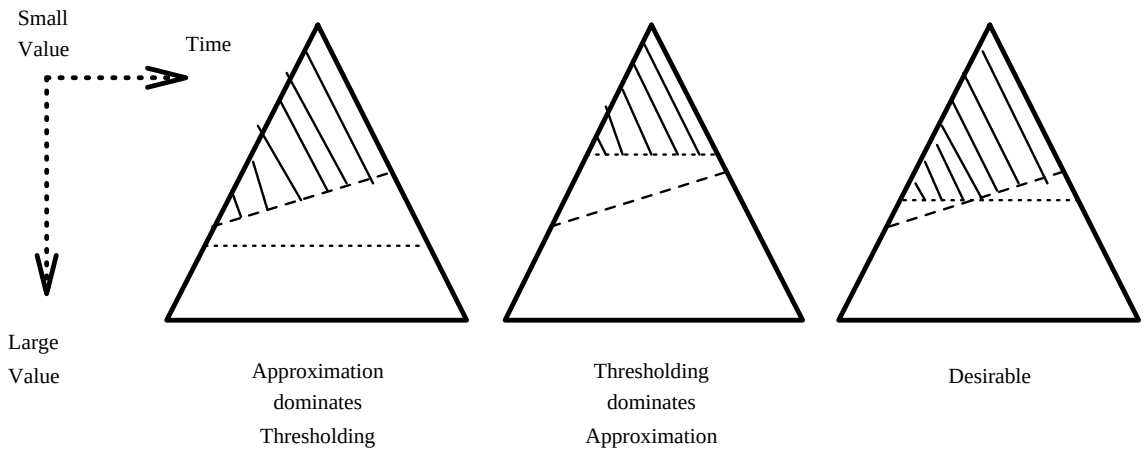


Figure 3.3. Interaction between approximation and thresholding, where the shaded area indicates the part of the search tree explored in a search. (Dashed lines contain approximation pruning thresholds, and dotted lines, thresholding pruning thresholds.)

(b) $\theta < p_{\varepsilon}(\varepsilon, \theta, t)$

In this case, thresholding dominates approximation because all search nodes pruned by approximation are also pruned by thresholding. The search degenerates into one using thresholding alone.

(c) $p_{\varepsilon}(\varepsilon, \theta, 0) \geq \theta \geq p_{\varepsilon}(\varepsilon, \theta, t)$

In this case, neither dominates the other. In the beginning, thresholding dominates approximation; but later, approximation dominates thresholding. This is desirable because (a) a search can achieve the same AAD as one using approximation alone but generally expends less time than the latter, and (b) it expends less time than one using thresholding alone. Note that a search using thresholding alone may explore more nodes and achieve a better AAD than a search using both approximation and thresholding. The reason for the achieved AAD to be equal to ε can be explained as

follows. When the search stops, the final global lower bound is equal to $p_{\varepsilon,\theta}(\varepsilon, \theta, t)$ because all nodes with lower bounds no less than $p_{\varepsilon,\theta}(\varepsilon, \theta, t)$ are pruned. According to Eqs. (3.1) and (3.2), we have $p_{\varepsilon,\theta}(\varepsilon, \theta, t) = p_{\varepsilon}(\varepsilon, \theta, t)$. Hence, the achieved AAD is equal to ε according to the definition of AAD in Eq. (2.2) and P-RULE I in Eq. (2.3).

In general, approximation is more informed than thresholding in the sense that approximation utilizes both incumbent and lower bounds to limit the scope of search, but thresholding only utilizes the lower bound.

3.2. Framework of Real-Time Search

RTS schedules a sequence of approximation GDFSs to find a feasible solution close to the optimal solution (namely, to minimize the achieved EAD) in the time allowed and to minimize the achieved AAD. Figure 3.4 shows the RTS algorithm as a closed-loop control. The parameters for each GDFS scheduled are dependent of the intermediate performance and amount of time used by the previous GDFSs scheduled.

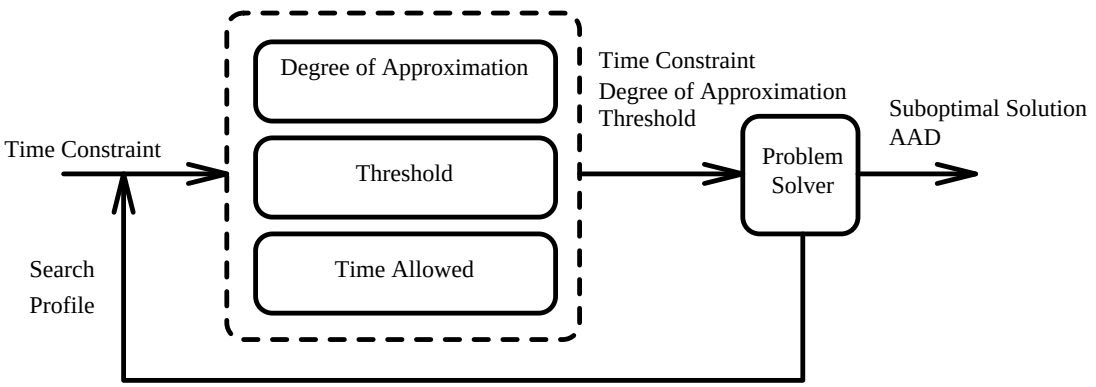


Figure 3.4. The framework of RTS.

A decision element d is a ternary tuple (S_d, P_d, R_d), where S_d is the decision strategy (the mechanism for allocating resources and selecting pruning element), P_d is the pruning element (approximation and/or thresholding), and R_d is the time constraint.

Under a time constraint T, the decision element d allocates time τ_i to $GDFS_i$, the i-th GDFS, and assigns approximation degree ε_i and threshold θ_i. This i-th GDFS is denoted as $GDFS_i(\pi_i)$ where $\pi_i = <\varepsilon_i, \theta_i, \tau_i>$ is the *constraint tuple* that specifies the AAD to be achieved, the threshold, and the time constraint. A *schedule* is a sequence of these constraint tuples. Note that $\varepsilon_0 > \varepsilon_1 > \cdots > \varepsilon_n$, and $\sum_{i=0}^{n} t_i \le T$.

The eventual AAD achieved by RTS under limited time T is the minimum of the AADs achieved in the last and the second-to-last GDFS, namely, $\varepsilon_{RTS(T)} = \min\{\varepsilon_{n-1}, \varepsilon_{GDFS_n(\pi_n)}\}$. The solution achieved is the incumbent of the last GDFS. Note that RTS allows the incumbent obtained in the previous GDFSs to be used in the new GDFS, so the incumbent of the last one is always the best. The propagation of incumbent was first proposed and studied in the IRA* algorithm [55].

In this thesis, we use RTS[A,B](T) to denote the RTS algorithm that solves dense COPs under time constraint T, based on approximation degrees selected by strategy A and thresholds by strategy B. Inactive strategy (I) denotes the degenerate strategy that sets either $\text{AAD} = 0$ or $\theta = \infty$.

A pruning mechanism is degenerate if it is inactive or is dominated by another. Consider GDFS(ε, θ, T). Approximation is degenerate (denoted as $\varepsilon \sim 0$) if $\varepsilon = 0$ or $z(\varepsilon, \theta, t)/(1+\varepsilon) > \theta$ *oppA t member* $[1, T]$. On the other hand, thresholding is degenerate (denoted as $\theta \sim \infty$) if $\theta = \infty$ or $\theta > z(\varepsilon, \theta, t)/(1+\varepsilon)$ *oppA t member* $[1, T]$.

A desirable schedule $\Lambda = \{ <\varepsilon_i, \theta_i, \tau_i> \mid i = 0,...,n \}$ satisfies the following conditions for every possible $i < j$.

$$\begin{aligned} &\text{(a) } \varepsilon_i > \varepsilon_j \text{ and } \theta_i, \theta_j \sim \infty, \text{ or} \\ &\text{(b) } \theta_i < \theta_j \text{ and } \varepsilon_i, \varepsilon_j \sim 0, \text{ or} \\ &\text{(c) } \varepsilon_i > \varepsilon_j \text{ and } \theta_i < \theta_j \text{ and neither } \varepsilon \text{ nor } \theta \text{ dominates the other.} \end{aligned} \tag{3.3}$$

A desirable execution of RTS makes progress in each primitive search, that is, $\varepsilon_i > \varepsilon_j$ for completed $GDFS_i$ and $GDFS_j$, *oppA* $i < j$.

It may happen that the completion time of a GDFS using larger AAD is greater than that using smaller AAD, that is, $t(\varepsilon_i) > t(\varepsilon_j)$ when $\varepsilon_i > \varepsilon_j$. This phenomenon is called *approximation anomaly*. Approximation anomaly has been well-studied [51, 53, 97], and happens when nodes x (resp. y) can be pruned by y (resp. x) under different conditions. In an approximation GDFS, AAD is not consistent with the incumbent nor is EAD; therefore, there exist nodes that may be pruned in terms of EAD but may not be pruned in terms of AAD.

In contrast, approximation anomaly cannot happen when thresholding is used alone. The reason is that all search nodes whose lower bounds are less than or equal to the threshold specified must be expanded.

Approximation anomalies are intractable because they depend on run-time information unavailable during a search. In this thesis, we develop real-time search strategies that do not account for these anomalies. Readers should note that these anomalies rarely result in poor performance with our strategies.

Conditions (a) and (b) in Eq. (3.3) can easily be satisfied by monotonically increasing and decreasing, respectively, thresholds and degrees of approximation. Condition (c) can be

satisfied by proper selection of thresholds or degrees of approximation. In this thesis, we design the RTS strategies such that RTS satisfies one of these three conditions.

RTS is asymptotically complete with respect to time in the sense that RTS can always find an optimal solution when time is unlimited. The achieved AAD in RTS is monotonically decreasing; therefore, RTS will eventually schedule a GDFS using $\varepsilon_n = 0$ when approximation is active, or RTS will eventually schedule a GDFS using $\theta \geq z^*$ when thresholding is active. Consequently, RTS can always find an optimal solution given unlimited time.

RTS can be considered an anytime algorithm. At any point in time, RTS can be stopped to return a solution. If more time is allowed, then RTS can be resumed from where it was stopped.

The RTS with static strategies can guarantee a better solution when the time allowed is larger. However, for the RTS with dynamic strategies, anomalies may happen such that the solution obtained may be worse given a larger time constraint.

3.3. Strategies for Real-Time Search with Approximation

RTS using approximation alone selects AADs progressively and uses inactive thresholding ($\theta=\infty$) in the schedule. The resulting RTS is a sequence of approximation GDFSs with incumbent propagation. Two decision strategies are studied: *linear gradient* (RTS[LG,I]) and *first-order regression* (RTS[FR,I]). By ‘‘linear gradient,’’ we mean that the scheduled AAD is decreased in a linear fashion. By ‘‘first-order regression,’’ we mean that the scheduled AAD is predicted by regressing a linear function of $\log t$. The linear-gradient strategy is static; that is, the scheduled AADs are decreased in a predetermined fashion. The

first-order-regression strategy is dynamic in that the scheduled AADs are predicted based on the performance of previous searches.

3.3.1. Empirical model of search complexity

We empirically observed that the completion time in solving a dense COP grows exponentially as the scheduled AAD decreases linearly. An intuitive explanation is that the COPs we study are NP-hard and that the completion time grows exponentially. In this section, we propose a model for this behavior and verify it empirically.

The profiles studied in this thesis can be classified into one of two types: actual and parametric. An *actual profile* shows the relationship between the execution time expended and the final AAD achieved. It is unknown until the search has been carried out. On the other hand, a *parametric profile* is an estimate of the actual profile and is obtained by regressing the partial actual profile. A problem-independent closed-form profile is desirable so it can be manipulated easily.

Two general algebraic profiles have been studied in our previous work [100]. We found empirically that high-order regression may be intractable in prediction, though it incurs less error in fitting; hence, we use a simple parametric profile in this thesis. Let

$$\varepsilon = \beta_0 + \beta_1 \log t, \tag{3.4}$$

with two boundary conditions: $t(0) = t_d$ and $t(\varepsilon_0) = 1$, where t_d is the execution time for a GDFS to find an optimal solution, and ε_0 as the AAD of the root node in the search tree. Note that t_d is a problem-dependent constant. Substituting the boundary conditions into the above equation, we have $\beta_0 = \varepsilon_0$ and $\beta_1 = -\varepsilon_0 / \log t_d$. Based on these conditions, the above equation can be rewritten as

$$\varepsilon(t) = \varepsilon_0 \left(1 - \log_{t_d} t\right) \tag{3.5}$$

Rewriting the above equation in terms of ε, we have

$$t(\varepsilon) = t_d^{\,1 - \frac{\varepsilon}{\varepsilon_0}}. \tag{3.6}$$

This empirical model is called a *simple exponential model* (SE model) for approximation, which is characterized by Eqs. (3.5) and (3.6).

Table 3.2 shows the statistical justification for the SE model on fifty random instances of each of the symmetric traveling salesperson problems satisfying triangular inequality, the knapsack problem, the production planning problem, and the vertex-cover problem. The statistical analysis is based on the *coefficient of determination* (R) [71], which is a common method for testing linearity of regression. If an actual profile (ε versus $\log t$) is perfectly linear (except horizontal), then $R = 1$. In contrast, $R = 0$ if the actual profile is horizontal. If an actual profile is almost linear, then R is close to one.

Table 3.2. Statistical analysis on linear regression of actual profiles (ε versus $\log t$) for fifty random instances of each of the symmetric traveling salesperson problem satisfying the triangular inequality, the knapsack problem, the production planning problem, and the vertex cover problem.

Application Problem	Number of Instances	Sizes of Instances	$\mu(R)$	$\sigma(R)$
Traveling Salesman	50	16 ~ 18	0.970	0.041
Production Planning	50	16 ~ 20	0.938	0.032
Knapsack	50	71 ~ 75	0.964	0.028
Vertex Cover	50	31 ~ 35	0.975	0.021

A point in an actual profile is a pair ε and $\log t$. This point means that the search with approximation degree ε takes t units of time to complete. Regression is performed on an

actual profile that excludes *transient points* and *saturation points*, where transient points are those with very small *t*, and saturation points are those whose *t* is very close to t_d. These points are excluded because they are potentially outliers.

As we can see in Table 3.2, the average *R* values for all these four problems are greater than 0.90 and their variations are quite small, which statistically implies that linear regression is meaningful and suitable.

It may be inappropriate for us to conclude from our limited experiments that the profiles of other COPs can be parameterized by the simple exponential function. In general, the distribution of lower bounds with respect to the number of search nodes in a finite search space is *bell-shaped*.[1] The exponential function used in modeling execution time fits well when the optimal solution value lies on the rising edge of the bell-shaped curve, which is the case for the traveling-salesman and the production-planning problems. The exponential model fails when (a) the optimal solution value is near the peak or on the falling edge of the bell-shaped profile, or (b) the profile function is growing faster than an exponential function. Both cases happen frequently in hard-constrained COPs (such as general integer programming problems). For these problems, feasible solutions cannot be derived easily. Case (a) also happens when the lower-bound function is loose in predicting the optimal solution value.

The dense COPs that do not satisfy the SE model can, in general, fall into one of two cases: (a) the problems are too complex to be modeled sufficiently by the SE model; and (b) the problems are so easy that they do not have exponential behavior. For the problems in the

[1] The statement that the distribution of lower bounds with respect to the number of search nodes is bell-shaped is not true in a COP with an infinite search space, such as the 15-puzzle problem.

first case, RTS still can perform well, though it may perform better by using polynomial fits of data. For the problems in the second case, RTS may not perform satisfactorily since the overheads in the previous iterations are relatively high. Further, since we are only interested in the trend of growth of the profile, the estimated profile does not have to be exact or very accurate.

In terms of both EAD and AAD, the best algorithm that solves dense COPs under limited time is OPTA*, which knows the optimal solution value when it starts. The AAD at the root node of OPTA* is, therefore,

$$\varepsilon_{0,OPTA^*} = del\ \frac{z^* - b_0}{b_0} = \frac{z^*}{b_0} - 1.$$

The time for completing an OPTA* search is the time for completing an A* search, that is, t_{a^*}. Assume that OPTA* satisfies the SE model [98]. Then the boundary conditions of the profile of OPTA* are $t(0) = t_{a^*}$ and $t(\varepsilon_{0,OPTA^*}) = 1$, where t_{a^*} is the execution time of A*. By applying the boundary conditions to Eq. (3.4), we obtain the parametric profile of OPTA* as

$$\varepsilon_{OPTA^*(T)}(t) = \varepsilon_{0,OPTA^*}\left(1 - \log_{t_{a^*}} t\right). \tag{3.7}$$

In Section 3.4.2, we describe approximation strategies and derive the worst-case performance bound. Approximate analysis of the average case is shown in Appendix A.

3.3.2. Approximation by linear gradient

Linear-gradient approximation works by decreasing the scheduled AADs in a linear fashion in hoping that the time expended will increase in an exponential fashion according to the SE model. The *k*-th GDFS is scheduled with an approximation degree ε_k

$$\varepsilon_k = \varepsilon_0 (1 - k\, g), \tag{3.8}$$

where g is a *gradient factor* that controls the linear step of approximation degrees. When the k-th GDFS is completed, $\varepsilon_k = \varepsilon_k$.

An example showing the linear-gradient strategy with $g = 0.1$ for solving a 10-object knapsack problem (as is seen in Figure 3.2) is shown in Figure 3.5 and Table 3.3. The root node has a lower bound of 2663 and an upper bound of 2895, hence, having an AAD of 0.0871. In this example, degrees of approximation are chosen based on the equation $\varepsilon_i = 0.0871 * (1 - k * 0.1)$. When the number of search nodes expanded is small, the search state is transient. In this case, the search is in a transient state when the number of nodes expanded is less than the problem size. The analysis of linear-gradient approximation is based on the SE model, assuming inactive thresholding and no approximation anomaly.

The execution time expended in GDFS(ε_k, ∞, ∞) is exponential with k, that is,

$$t(\varepsilon_k) = t_d^{\,k\,g}.$$

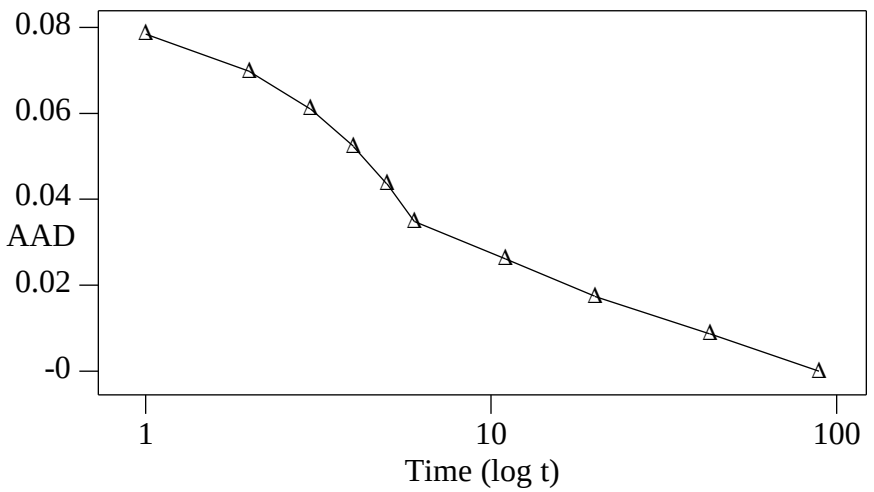


Figure 3.5. An example for approximation by linear gradient with $g = 0.1$, where this profile is obtained by solving the 10-object knapsack problem shown in Figure 3.2.

Table 3.3. Listing of search parameters used in Figure 3.5.

Iteration Index	Degree of Approximation	Incumbent	Global Lower Bound	Time Spent in This GDFS	Total Time Spent
1	0.078	2799	2596	1	1
2	0.070	2799	2617	1	2
3	0.061	2799	2639	1	3
4	0.052	2799	2660	1	4
5	0.044	2799	2683	1	5
6	0.035	2799	2705	1	6
7	0.026	2799	2728	5	11
8	0.017	2799	2752	9	20
9	0.009	2799	2775	23	43
10	0.000	2799	2799	46	89

This can be derived by applying Eq. (3.8) to Eq. (3.6) and by noting that $\varepsilon = \varepsilon_k$ when the search completes.

Assume that the m-th GDFS is the last *completed* GDFS. This implies that the (m+1)-th GDFS is terminated prematurely due to time constraint T. As a result, time constraint T must be bounded from below by the sum of times expended in all completed GDFSs, and must be bounded from above by the sum of times expended in all completed ones plus the time expended in completing the (m+1)-th GDFS. Namely, $\sum_{k=0}^{m} t_d{}^{kg} \le T < \sum_{k=0}^{m+1} t_d{}^{kg}$. By simplifying the geometric series, we can rewrite it into

$$\frac{[t_d{}^{g(m+1)} - 1]}{[t_d{}^{g} - 1]} \le T < \frac{[t_d{}^{g(m+2)} - 1]}{[t_d{}^{g} - 1]},$$

assuming that g is not equal to 0. Finally, we can derive the bounds of m as

$$m_{\min} = \left\lceil \frac{1}{g} \log_{t_d} \left(T(t_d{}^{g} - 1) + 1 \right) - 2 \right\rceil, \tag{3.9.a}$$

$$m_{\max} = \left\lceil \frac{1}{g} \log_{t_d} \left(T\,(t_d{}^{g} - 1) + 1 \right) - 1 \right\rceil . \qquad (3.9.b)$$

For $T \geq 1$, $m_{\max}$ monotonically increases as T increases. When T is 1, $m_{\max}$ is 0. As a result, $m_{\max}$ is nonnegative for $T \geq 1$. To assure that $m_{\min}$ is nonnegative, we assume that $T \geq 2$, and that g is in the range (0, $\log_{t_d}(T-1)$], where 0 and $\log_{t_d}(T-1)$ are the roots of $m_{\min} = 0$.

Let $\varepsilon_{RTS[LG,I](T)}$ and $\varepsilon_{OPTA^*(T)}$ be the AADs achieved by RTS[LG,I] and OPTA* under time constraint T, respectively, and $\Delta\varepsilon_{RTS[LG,I](T)}$ be the upper bound on their difference.

The worst-case value of ε_m can be calculated by substituting $m_{\min}$ into Eq. (3.8). In terms of the worst-case value of ε_m, the worst-case $\varepsilon_{RTS[LG,I](T)}$ can be obtained by applying the worst-case value of ε_m (due to $m_{\min}$) to Eq. (3.8). By subtracting $\varepsilon_{OPTA^*(T)}$ in Eq. (3.7) from the worst-case $\varepsilon_{RTS[LG,I](T)}$, we can derive $\Delta\varepsilon_{RTS[LG,I](T)}$ as a function of time constraint T and gradient factor g, that is,

$$\varepsilon_{RTS[LG,I](T)} - \varepsilon_{OPTA^*(T)} \;<\; \Delta\varepsilon_{RTS[LG,I](T)} \;=\; \left(\varepsilon_0 - \varepsilon_{0,OPTA^*} \right) + H(T) + F^{worst}_{LG,I}(g,\,T)\,,$$

$$\text{where } H(T) \;=\; \left(\varepsilon_{0,OPTA^*} \log_{t_{a^*}} T - \varepsilon_0 \log_{t_d} T \right),$$

$$F^{worst}_{LG,I}(g,\,T) \;=\; 2\,\varepsilon_0\, g - \varepsilon_0 \log_{t_d} \left(t_d{}^{g} - 1 + T^{-1} \right).$$

Note that the first term and $F^{worst}_{LG,I}(g,\,T)$ are positive, Also note that $H(T)$ can be positive or negative. For instance, when $t_{a^*} = t_d$, $H(T) = -L \log_{t_{a^*}} T \leq 0$ due to $T \leq t_d$; but when $t_{a^*} \ll t_d$, $H(T) \approx -\varepsilon_0 \log_{t_d} T \geq 0$. In general, when $t_{a^*} \approx t_d$, $H(T)$ is likely to be negative. This implies that when GDFS performs well, this bound becomes small.

$\Delta\varepsilon_{RTS[LG,I](T)}$ is roughly logarithmic with respect to T (due to $H(T)$), since the effect of T in $F^{worst}_{LG,I}(g,T)$ is very small and negligible. Also, $\Delta\varepsilon_{RTS[LG,I](T)}$ is roughly linear with respect to g when g is large, since $F^{worst}_{LG,I}(g,T) \approx \varepsilon_0\, g$; it is independent of g when g is small, since

$F^{worst}_{LG,I}(g,T) \approx \varepsilon_0 \log_{t_d} T$ as $g \approx 0$. When g is in between 0 and 1, the bound is curvilinear with g, so that there exists a best choice of g such that the bound is minimal. The following lemma shows the best choice of g for minimizing $\Delta\varepsilon_{RTS[LG,I](T)}$.

Lemma 3.1. *In terms of minimizing* $\Delta\varepsilon_{RTS[LG,I](T)}$ *in the worst case, the best gradient factor is*

$$g^{*worst} = \log_{t_d}\left[2\left(1-T^{-1}\right)\right] \approx \log_{t_d} 2. \tag{3.10}$$

Proof. This lemma can be proved by taking the partial derivative of $\Delta\varepsilon_{RTS[LG,I](T)}$ with respect to g, equating the partial derivative to zero, and solving the equation for the root, g^{*worst}. The approximation is acceptable as T usually is much larger than 1. □

Theorem 3.1. *In terms of* g^{*worst}, $\Delta\varepsilon_{RTS[LG,I](T)}$ *in the worst case is bounded from above by*

$$\Delta\varepsilon_{RTS[LG,I](T)}(g^{*worst}) < \left(\varepsilon_0 - \varepsilon_{0,OPTA^*}\right) + \left(\varepsilon_{0,OPTA^*}\log_{t_{a^*}} T - \varepsilon_0 \log_{t_d} T\right) + 2\,\varepsilon_0 \log_{t_d} 2 .$$

Proof. By applying g^{*worst}, $\Delta\varepsilon_{RTS[LG,I](T)}$ can be rewritten as

$$\begin{aligned}\Delta\varepsilon_{RTS[LG,I](T)}(g^{*worst}) &= \left(\varepsilon_0 - \varepsilon_{0,OPTA^*}\right) + \left(\varepsilon_{0,OPTA^*}\log_{t_{a^*}} T - \varepsilon_0 \log_{t_d} T\right) \\ &\quad + 2\,\varepsilon_0 \log_{t_d}\left[2\left(1-T^{-1}\right)\right] - \varepsilon_0 \log_{t_d}\left(1-T^{-1}\right) \\ &< \left(\varepsilon_0 - \varepsilon_{0,OPTA^*}\right) + \left(\varepsilon_{0,OPTA^*}\log_{t_{a^*}} T - \varepsilon_0 \log_{t_d} T\right) + 2\,\varepsilon_0 \log_{t_d} 2 .\end{aligned}$$ □

Note that the bound in the above theorem is *the best bound in the worst case*. This bound will be used for bounding the performance of RTS later.

3.3.3. Approximation by first-order regression

RTS[FR,I] works by regressing the relationship between the achieved AAD and the execution time expended by the previous GDFSs and by using the relation to predict the AAD to be used in the next GDFS. Specifically, to predict ε_k, the relationship in Eq. (3.4) is regressed over all the pairs (ε_i, t_i) for $i = 0,...,k-1$.

The rationale of RTS[FR,I] is that the time ratio in completing successive GDFSs should be a constant r (called the *growth rate*). After the k-th GDFS is completed, we have

$$t_k = r\, t_{k-1}. \tag{3.11}$$

As a result, the time expended in a GDFS grows exponentially with the number of GDFSs evaluated.

A GDFS scheduled later should expend more time than previous ones. As a result, the last GDFS is expected to produce the best solution. However, if the last GDFS cannot produce a better solution, then the time scheduled for it is wasted. Therefore, the prediction of the last completed GDFS is essential, provided that prediction is accurate. Consider the k-th GDFS to be scheduled. This iteration is expected to expend $\hat{t}_k$ time units. To avoid the last iteration to be wasted, the remaining time after completing this iteration must be large enough so that the remaining time is no less than $\hat{t}_{k+1}$. Before the k-th GDFS is scheduled, we need to check to make sure that

$$\sum_{i=0}^{k-1} t(\varepsilon_i) + \hat{t}_k + \hat{t}_{k+1} \le T,$$

where $\hat{t}_k = r\, t_{k-1}$ and $\hat{t}_{k+1} = r^2\, t_{k-1}$. If this inequality is false, then the k-th GDFS is not granted, and the GDFS with an AAD that uses up all remaining time is scheduled.

The execution time in completing GDFS(ε_k, ∞, ∞) is exponential with k, that is,

$$t(\varepsilon_k) = r^k. \tag{3.12}$$

Equation (3.12) can be derived by applying Eq. (3.11) iteratively until $t_0 = 1$ (where only the root node is expanded). Based on Eq. (3.12), we can derive the AAD achieved after completing GDFS(ε_k, ∞, ∞). The AAD achieved in the k-th GDFS can be derived easily by applying Eq. (3.12) to Eq. (3.5).

$$\varepsilon_k = \varepsilon_0 \left(1 - k \log_{t_d} r\right). \tag{3.13}$$

An example showing the first-order-regression strategy with $r = 3.513$ for solving a 40-object knapsack problem (with random seed of 1) is shown in Figure 3.6 and Table 3.4.

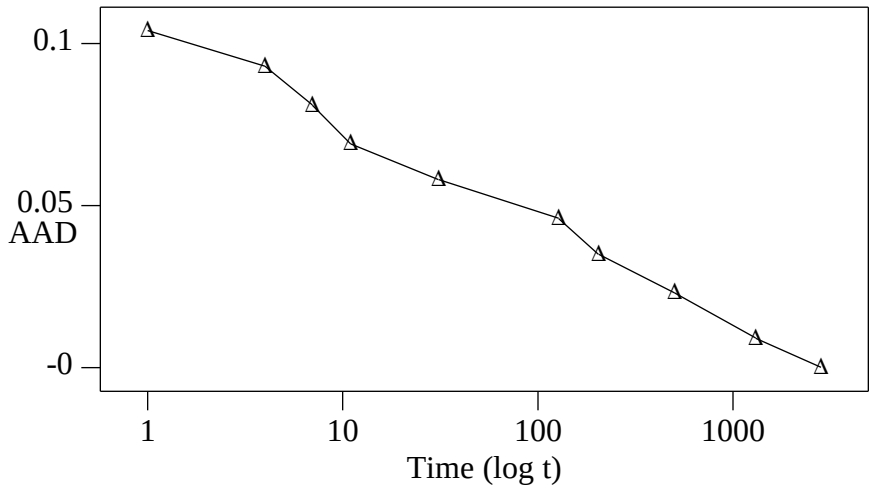


Figure 3.6. An example showing approximation by first-order regression with $r = 3.513$, where the profile is obtained by solving a 40-object knapsack problem (with random seed of 1).

Table 3.4. Search parameters for the example in Figure 3.6.

Iteration Index	Degree of Approximation	Incumbent	Global Lower Bound	Time Spent in This GDFS	Total Time Spent
1	0.104	10749	9735	1	1
2	0.093	10749	9839	3	4
3	0.081	10749	9944	3	7
4	0.069	10715	10020	4	11
5	0.058	10644	10062	20	31
6	0.046	10644	10174	97	128
7	0.035	10493	10141	77	205
8	0.023	10493	10256	298	503
9	0.009	10493	10401	877	1380
10	0.000	10434	10434	1449	2829

We do not show the example of the 10-object knapsack problem (as is seen in Figure 3.2) because the problem is so small that the profile of the first-order regression is exactly the same as that of the linear gradient. The root node has a lower bound of 9634, an upper bound of 10749, and an achieved AAD of 0.1157. In this example, the search is in a transient state when the number of search nodes expanded is less than the problem size.

Assume that the *m*-th GDFS is the last completed GDFS. By following similar derivation for Eq. (3.9), *m* is bounded from below by $m_{\min}$ and from above by $m_{\max}$, where $m_{\min}$ and $m_{\max}$ are functions of time constraint *T* and growth rate *r*:

$$m_{\min} = \log_r \left(T(r-1)+1 \right) - 2 ; \tag{3.14.a}$$

$$m_{\max} = \log_r \left(T(r-1)+1 \right) - 1 . \tag{3.14.b}$$

For $T \geq 1$, $m_{\max}$ increases monotonically as *T* increases. When *T* is 1, $m_{\max}$ is 0. As a result, $m_{\max}$ is nonnegative. To assure that $m_{\min}$ is nonnegative, we can assume that $T \geq 2$, and that *r* is in the range $(1, T-1]$, where 1 and $T-1$ are the roots of *r* for $m_{\min} = 0$.

Let $\varepsilon_{RTS[FR,I](T)}$ be the AAD achieved by RTS[FR,I] in time constraint *T*; also, let $\Delta\varepsilon_{RTS[FR,I](T)}$ be the upper bound on the difference in AAD between $\varepsilon_{RTS[FR,I](T)}$ and $\varepsilon_{OPTA^*(T)}$.

The worst-case value of ε_m can be calculated by substituting $m_{\min}$ into Eq. (3.13). In terms of the worst-case value of ε_m, the worst-case $\varepsilon_{RTS[FR,I](T)}$ can be obtained by applying the worst-case value of ε_m (due to $m_{\min}$) in Eq. (3.13). By subtracting $\varepsilon_{OPTA^*(T)}$ in Eq. (3.7) from the worst-case $\varepsilon_{RTS[FR,I](T)}$, we can derive $\Delta\varepsilon_{RTS[FR,I](T)}$ as a function of time constraint *T* and growth rate *r*, that is,

$$\varepsilon_{RTS[FR,I](T)} - \varepsilon_{OPTA^*(T)} < \Delta\varepsilon_{RTS[FR,I](T)} = \left(\varepsilon_0 - \varepsilon_{0,OPTA^*} \right) + H(T) + F^{worst}_{FR,I}(r, T),$$

$$\text{where } H(T) = \left(\varepsilon_{0,OPTA^*} \log_{t_{a^*}} T - \varepsilon_0 \log_{t_d} T \right),$$

$$F_{FR,I}^{worst}(r, T) = 2\,\varepsilon_0 \log_{t_d} r - \varepsilon_0 \log_{t_d} \left(r - 1 + T^{-1} \right).$$

Note that $F_{FR,I}^{worst}(r,T)$ is positive. $\Delta\varepsilon_{RTS[FR,I](T)}$ is roughly logarithmic with respect to T (due to $H(T)$), since the effect of T in $F_{FR,I}^{worst}(r,T)$ is very small and negligible. As a result, as T becomes larger, $\Delta\varepsilon_{RTS[FR,I](T)}$ will become larger. Also, $\Delta\varepsilon_{RTS[FR,I](T)}$ is roughly logarithmic with r when r is large, since $F_{FR,I}^{worst}(r,T) \approx \varepsilon_0 \log_{t_d} r$ as $r \gg 1$; it is independent of r when r is small, since $F_{FR,I}^{worst}(r,T) \approx \varepsilon_0 \log_{t_d} T$ as $r \approx 1$. When r is larger than 1, the bound is curvilinear with r; consequently, there exists a best choice of r such that the bound is minimal. The following lemma shows the best choice of r for minimizing $\Delta\varepsilon_{RTS[FR,I](T)}$.

Lemma 3.2. *In terms of minimizing $\Delta\varepsilon_{RTS[FR,I](T)}$ in the worst case, the best growth rate is*

$$r^{*worst} = 2\left(1 - T^{-1}\right) \approx 2. \tag{3.15}$$

Proof. The lemma is proved by taking the partial derivative of $\Delta\varepsilon_{RTS[FR,I](T)}$ with respect to r, equating the partial derivative to zero, and solving the equation for the root, r^{*worst}. This approximation is acceptable as T usually is much larger than 1. □

Theorem 3.2. *In terms of r^{*worst}, $\Delta\varepsilon_{RTS[FR,I](T)}$ in the worst case is bounded from above by*

$$\Delta\varepsilon_{RTS[FR,I](T)}(r^{*worst}) < \left(\varepsilon_0 - \varepsilon_{0,OPTA^*}\right) + \left(\varepsilon_{0,OPTA^*} \log_{t_{a^*}} T - \varepsilon_0 \log_{t_d} T\right) + 2\,\varepsilon_0 \log_{t_d} 2.$$

Proof. By applying r^{*worst} in Eq. (3.15), $\Delta\varepsilon_{RTS[FR,I](T)}$ can be rewritten as

$$\begin{aligned}\Delta\varepsilon_{RTS[FR,I](T)}(r^{*worst}) &= \left(\varepsilon_0 - \varepsilon_{0,OPTA^*}\right) + \left(\varepsilon_{0,OPTA^*} \log_{t_{a^*}} T - \varepsilon_0 \log_{t_d} T\right) \\ &\quad + 2\,\varepsilon_0 \log_{t_d} \left[2\left(1 - T^{-1}\right)\right] - \varepsilon_0 \log_{t_d} \left(1 - T^{-1}\right) \\ &< \left(\varepsilon_0 - \varepsilon_{0,OPTA^*}\right) + \left(\varepsilon_{0,OPTA^*} \log_{t_{a^*}} T - \varepsilon_0 \log_{t_d} T\right) + 2\,\varepsilon_0 \log_{t_d} 2.\end{aligned}$$

□

Theorem 3.3. *Using approximation pruning alone, RTS using the first-order-regression strategy has the same worst-case behavior as RTS using the linear gradient strategy when*

$$g = \log_{t_d} r. \tag{3.16}$$

Proof. By substituting Eq. (3.16) into Eq. (3.10), we can conclude $r = 2$, which is the same situation as derived in Eq. (3.15). □

For a dynamic decision strategy such as the first-order-regression strategy, it is hard to schedule time and approximation degrees for GDFSs early in the search, because there is insufficient information for prediction. Fortunately, the above theorem shows that the approximation degrees of GDFSs early in RTS can be scheduled by a linear-gradient strategy. When performance statistics are sufficient for prediction, first-order regression can then be employed.

3.4. Strategies for Real-Time Search with Thresholding

RTS using thresholding alone is denoted as RTS[I,H]. It selects thresholds for pruning by strategy H and uses no (or inactive) approximation pruning in the schedule. The resulting RTS is a sequence of GDFSs with thresholding. The traditional IDA* is similar to RTS[I,H] except the new incumbent found in one GDFS is propagated to the next in RTS. (Each GDFS in IDA* always starts with the same incumbent.) This propagation of incumbent allows a tight AAD. A sequence of GDFSs with incumbent propagation was first proposed by Li and Wah [55].

Two decision strategies for thresholding are studied: *linear gradient* (RTS[I,LG]) and *first-order regression* (RTS[I,FR]). Linear gradient means that thresholds are scheduled to increase in a linear fashion, whereas first-order regression means that thresholds are

predicted based on regression of the relation between thresholds and log(t). The linear-gradient strategy is static, while the first-order-regression strategy is dynamic. In this section, these two decision strategies are described and analyzed.

3.4.1. Empirical model of thresholding and its justification

We empirically observe that the completion time for solving a COP grows exponentially as the threshold is increased linearly [37, 84, 103]. The intuitive explanation of this observation is that the COPs we study are NP-hard and that the distribution of lower bounds is exponential with the threshold values. Consequently, completion times grow exponentially with thresholds.

Consider the following two boundary conditions: $t(0, b_0) = 1$ and $t(0, z^*) = t_{a^*}$, where t_{a^*} is the execution time for A* to find an optimal solution, and b_0 is the lower bound of the root node in the search tree. The second boundary condition exists because all nodes with lower bounds less than the threshold (set equal to the optimal solution value z^*) must be expanded. This only happens in an A* search when there is no node whose lower bound is equal to the optimal solution value. Based on the boundary conditions, the parametric profile becomes

$$t(0, \theta) = t_{a^*}^{\frac{\theta - b_0}{z^* - b_0}}. \tag{3.17}$$

Rewriting Eq. (3.17) in terms of the threshold, we have

$$\theta(0, t) = b_0 + (z^* - b_0) \log_{t_{a^*}} t. \tag{3.18}$$

The above two equations characterize the SE model for thresholding.

Two observations can be made on these two equations. First, they characterize searches for θ *member* $[b_0, z^*]$ only, and the behavior outside this range may be very different. Second, a new incumbent found cannot prune nodes with lower bounds in this range.

When a GDFS with thresholding stops, the AAD achieved can be calculated by Eq. (2.5) using the incumbent z obtained and the threshold θ used. This means that it is impossible to analyze the achieved AAD without knowing z. However, as found empirically, the behavior of the incumbent is hard to model. To be able to analyze the performance of thresholding, we assume that the SE model for approximation still holds. This assumption is reasonable for the dense COPs we study because the incumbent changes infrequently during a search and pruning by thresholding becomes similar to pruning by approximation. However, the boundary conditions here are slightly different from those for approximation alone: $\varepsilon(0, b_0, 1) = \varepsilon_0$ and $\varepsilon(0, z^*, t_{a^*}) = 0$. The first boundary condition says that when $\theta = b_0$, then the time to complete the GDFS is 1 since only the root node is expanded; as a result, the AAD achieved is ε_0. The second boundary condition says that when $\theta = z^*$, then the time to complete the GDFS is equal to the time to complete the A* search, that is, t_{a^*}. Obviously, the AAD achieved is 0. The modified equations for modeling approximation degrees achieved become

$$\varepsilon(t) = \varepsilon_0 \left(1 - \log_{t_{a^*}} t\right), \tag{3.19}$$

$$t(\varepsilon) = t_{a^*}^{\,1 - \frac{\varepsilon}{\varepsilon_0}}, \tag{3.20}$$

where $t = t(0, \theta)$ and $\varepsilon = \varepsilon(0, \theta)$ for threshold θ.

Table 3.5 shows the statistical analysis of the SE model for thresholding over actual profiles (θ versus $\log t$) for fifty random instances of each of the symmetric traveling

salesperson problems satisfying triangular inequality, the knapsack problem, the production planning problem, and the vertex cover problem. Recall that R, the coefficient of determination, is close to one when linear regression is meaningful and suitable. As we can see in Table 3.5, the average R values for all these four problems are greater than 0.90 and their variations are quite small, which statistically implies that linear regression is meaningful and suitable.

Table 3.5. Statistical analysis on linear regression of actual profiles (θ versus $\log t$) for fifty random instances for each of the symmetric traveling salesperson problems satisfying triangular inequality, the knapsack problem, the production planning problem, and the vertex cover problem.

Application Problem	Number of Instances	Sizes of Instances	$\mu(R)$	$\sigma(R)$
Traveling Salesman	50	16 ~ 18	0.952	0.058
Production Planning	50	16 ~ 20	0.964	0.040
Knapsack	50	71 ~ 75	0.956	0.031
Vertex Cover	50	31 ~ 35	0.911	0.062

In Section 3.4.2, we will describe thresholding strategies and derive the worst-case performance bound. Approximate analysis of the average case is shown in Appendix A.

3.4.2. Thresholding by linear gradient

RTS[I,LG] is similar to RTS[LG,I]. Thresholds are selected in a linear fashion in hoping that the time to complete successive GDFSs increases in an exponential fashion according to the SE model. The threshold scheduled in the k-th GDFS is as follows:

$$\theta_k = b_0 + k\,g\,(z_0 - b_0), \tag{3.21}$$

where g is the gradient factor that controls the linear step of thresholding, b_0 is the lower bound of the root node, and z_0 is the upper-bound solution at the root node. When the k-th GDFS($0, \theta_k, t_k$) is completed, the AAD achieved can be obtained by combining Eqs. (2.5) and (3.21)

$$\varepsilon_k(0, \theta_k) = \frac{z_k - \theta_k}{\theta_k} = \frac{z_k}{k\,g\,(z_0 - b_0) + b_0} - 1,$$

where z_k is the final incumbent in the k-th GDFS.

It is interesting to compare linear-gradient thresholding with linear-gradient approximation. The AAD achieved in GDFS($\varepsilon_k, \infty, t_k$) of RTS[LG,I] is

$$\varepsilon_k = \varepsilon_0\,(1 - k\,g) = \frac{(z_0 - b_0)(1 - k\,g)}{b_0} = \frac{z_0 - k\,g\,(z_0 - b_0)}{b_0} - 1.$$

We observe that thresholding primarily moves the global lower bound to the optimal solution value, whereas approximation primarily moves the incumbent to the optimal one. As a result, approximation and thresholding are duals, although approximation is generally preferred since it results in a better incumbent.

An example showing the linear-gradient strategy with $g = 0.1$ for solving the 10-object knapsack problem (as is seen in Figure 3.2) is shown in Figure 3.7 and Table 3.6. Note that the root node has a lower bound of 2663 and an upper bound of 2895. In this example, thresholds are set based on the formula $\theta_i = 2663 + k * 0.1 * (2895 - 2663)$. We plot in Figure 3.7 the profile of AADs achieved versus times and that of thresholds versus times. Both curves in the graph are approximately linear.

The following analysis is based on the SE model for thresholding and the modified SE model for approximation, assuming that approximation pruning is not applied concurrently.

By applying Eq. (3.21) to Eq. (3.17), the execution time expended in GDFS(0, θ_k, ∞) can be shown to be exponential with k, that is,

$$t(0, \theta_k) = t_{a^*}^{\lambda k g}, \tag{3.22}$$

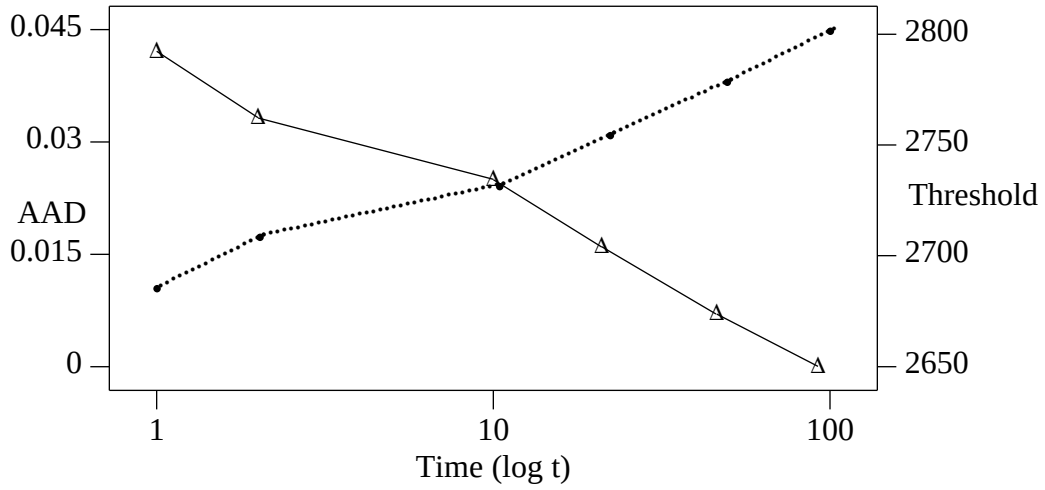


Figure 3.7. An example showing thresholding by linear gradient with $g = 0.1$, where the profile is obtained by solving the 10-object knapsack problem shown in Figure 3.2. (The solid curve is for the profile of AADs achieved versus times, and the dotted curve, the profile of thresholds versus times.)

Table 3.6. Search parameters used for the example in Figure 3.7.

Iteration Index	Threshold	AAD Achieved	Incumbent	Time Spent in This GDFS	Total Time Spent
1	2686	0.0421	2799	1	1
2	2709	0.0332	2799	1	2
3	2732	0.025	2799	8	10
4	2755	0.016	2799	11	21
5	2779	0.007	2799	25	46
6	2802	0.000	2799	46	92

where $\lambda = del\ (z_0 - b_0) / (z^* - b_0)$. Note that $\lambda \geq 1$ due to $z_0 \geq z^*$. λ is a constant because z_0, z^*, and b_0 are constants. This constant can be considered as an indicator of the quality of the heuristic algorithm applied to find a feasible solution with value z_0.

It is interesting to note that $t_{a^*}{}^{\lambda}$ is the extrapolated execution time for completing a GDFS(0, z_0, ∞) according to Eq. (3.17). In this case, thresholding is inactive and does not result in pruning of nodes ($z \leq z_0 = \theta$, or $\theta = z_0 \sim \infty$). Therefore, GDFS(0, z_0, ∞) behaves like a pure GDFS, that is, GDFS(0, ∞, ∞). Although it may be reasonable to conjecture that $t_{a^*}{}^{\lambda} = t_d$, the extrapolation of the SE model beyond the region [b_0,z^*] is invalid for two reasons.

(a) Nodes with lower bounds larger than z^* usually occur close to the leaves of the search tree, and their lower bounds do not change as fast as those close to the root or as those in the middle of the search tree. As a result, the distributions of lower bound values are different in general between nodes with lower bounds less than z^* and those larger than z^*.

(b) Search nodes with lower bounds larger than z^* may be pruned by new incumbents found. Therefore, the time expended in the search can be less than that predicted.

Heuristically, $t_{a^*}{}^{\lambda}$ is an estimate of t_d; hence, as an approximation, we can assume that they are equal in our analysis of the behavior of RTS with thresholding.

$$t_{a^*}{}^{\lambda} = t_d \tag{3.23}$$

Equation (3.22) can be rewritten as

$$t(0, \theta_k) = t_d{}^{kg}. \tag{3.24}$$

By applying Eq. (3.21) to Eq. (3.17), and combining it with Eq. (3.19), the AAD achieved after completing the *k*-th GDFS is

$$\varepsilon_k = \varepsilon_0 (1 - \lambda k g).$$

By following a similar derivation as in Eq. (3.9), we can have the bounds of *m* as

$$m_{\min} = del \left[\frac{1}{g} \log_{t_d} \left(T (t_d{}^{g} - 1) + 1 \right) - 2 \right] < m \le \left[\frac{1}{g} \log_{t_d} \left(T (t_d{}^{g} - 1) + 1 \right) - 1 \right] = del\ m_{\max}.$$

Note that the above equation is exactly the same as Eq. (3.9).

The worst-case value of ε_m can be calculated by substituting $m_{\min}$ into Eq. (3.8). The worst-case $\varepsilon_{RTS[I,LG](T)}$ can be obtained by applying the worst-case value of ε_m (due to $m_{\min}$) to Eq. (3.20). By subtracting $\varepsilon_{OPTA^*(T)}$ in Eq. (3.7) from the worst-case $\varepsilon_{RTS[I,LG](T)}$ we can derive $\Delta\varepsilon_{RTS[I,LG](T)}$ as a function of time constraint *T* and gradient factor *g*, that is,

$$\varepsilon_{RTS[I,LG](T)} - \varepsilon_{OPTA^*(T)} < \Delta\varepsilon_{RTS[I,LG](T)} = \left(\varepsilon_0 - \varepsilon_{0,OPTA^*} \right) \left(1 - \log_{t_{a^*}} T \right) + F^{worst}_{I,LG}(g,T),$$

$$\text{where } F^{worst}_{I,LG}(g,T) = 2\lambda\varepsilon_0 g - \varepsilon_0 \log_{t_{a^*}} \left(t_{a^*}{}^{\lambda g} - 1 + T^{-1} \right),$$

Note that $F^{worst}_{I,LG}(g,T)$ is positive. $\Delta\varepsilon_{RTS[I,LG](T)}$ is roughly logarithmic with respect to *T*, since the effect of *T* in $F^{worst}_{I,LG}(g,T)$ is very small and negligible. As a result, as *T* becomes larger, $\Delta\varepsilon_{RTS[I,LG](T)}$ will become larger. Also, $\Delta\varepsilon_{RTS[I,LG](T)}$ is roughly linear with respect to *g* when *g* is large, since $F^{worst}_{I,LG}(g,T) \approx \lambda\varepsilon_0 g$; it is independent of *g* when *g* is small, since $F^{worst}_{I,LG}(g,T) \approx \varepsilon_0 \log_{t_{a^*}} T$ as $g \approx 0$. When *g* is in between 0 and 1, the bound is curvilinear with *g*, so that there exists a best *g* such that the bound is minimal. The following lemma shows the best *g* for minimizing $\Delta\varepsilon_{RTS[I,LG](T)}$.

Lemma 3.3. *In terms of minimizing* $\Delta\varepsilon_{RTS[I,LG](T)}$ *in the worst case, the best gradient factor is*

$$g^{*worst} = \frac{1}{\lambda}\log_{t_{a^*}}\left[2\left(1-T^{-1}\right)\right] = \log_{t_d}\left[2\left(1-T^{-1}\right)\right] \approx \log_{t_d} 2\,. \quad (3.25)$$

Proof. This lemma can be proved by taking the partial derivative of $\Delta\varepsilon_{RTS[I,LG](T)}$ with respect to *g*, equating the partial derivative to zero, and solving the equation for the root, g^{*worst}. The second equality holds due to Eq. (3.23). This approximation is acceptable as *T* usually is much larger than 1. □

Theorem 3.4. *In terms of* g^{*worst}, $\Delta\varepsilon_{RTS[I,LG](T)}$ *in the worst case is bounded from above by*

$$\Delta\varepsilon_{RTS[I,LG](T)}(g^{*worst}) < \left(\varepsilon_0-\varepsilon_{0,OPTA^*}\right)\left(1-\log_{t_{a^*}} T\right) + 2\,\varepsilon_0\log_{t_{a^*}} 2\,.$$

Proof. By applying g^{*worst} to $\Delta\varepsilon_{RTS[I,LG](T)}$, $F_{I,LG}^{worst}(g,T)$ can be rewritten as

$$F_{I,LG}^{worst}(g,T) = 2\,\varepsilon_0\log_{t_{a^*}}\left[2\left(1-T^{-1}\right)\right] - \varepsilon_0\log_{t_{a^*}}\left(1-T^{-1}\right) = 2\,\varepsilon_0\log_{t_{a^*}} 2. \quad \square$$

3.4.3. Thresholding by first-order regression

RTS[I,FR] works by regressing the relationship between thresholds θ_i and times t_i expended in completing GDFSs, $i=0,...,k-1$, and by using the regressed relation to predict θ_k. The rationale of RTS[I,FR] is that the ratio of times spent in completing successive GDFSs is based on the growth rate *r* as is defined in Eq. (3.11). After the *k*-th GDFS is completed, t_k is defined as is in Eq. (3.12). As a result, the time expended in a GDFS grows exponentially with the number of GDFSs evaluated.

The execution time for solving GDFS(0, θ_k, ∞) is exponential with *k*, that is,

$$t(0,\theta_k) = r^k. \quad (3.26)$$

Equation (3.26) can be derived by applying Eq. (3.12) iteratively until $t_0 = 1$ (only the root node is expanded). By applying Eq. (3.26) to Eq. (3.18), we can derive the threshold to be scheduled for the *k*-th GDFS as

$$\theta_k = b_0 + (z^* - b_0)\, k \log_{t_{a^*}} r.$$

By applying Eq. (3.26) to Eq. (3.19), we can derive the AAD achieved in the *k*-th GDFS as

$$\varepsilon_k = \varepsilon_0 \left(1 - k \log_{t_{a^*}} r\right). \tag{3.27}$$

An example showing the first-order-regression strategy with $r = 3.513$ for solving a 40-object knapsack problem (with random seed of 1) is shown in Figure 3.8 and Table 3.7. The root node has a lower bound of 9634, an upper bound of 10749, and an achieved AAD of 0.1157.

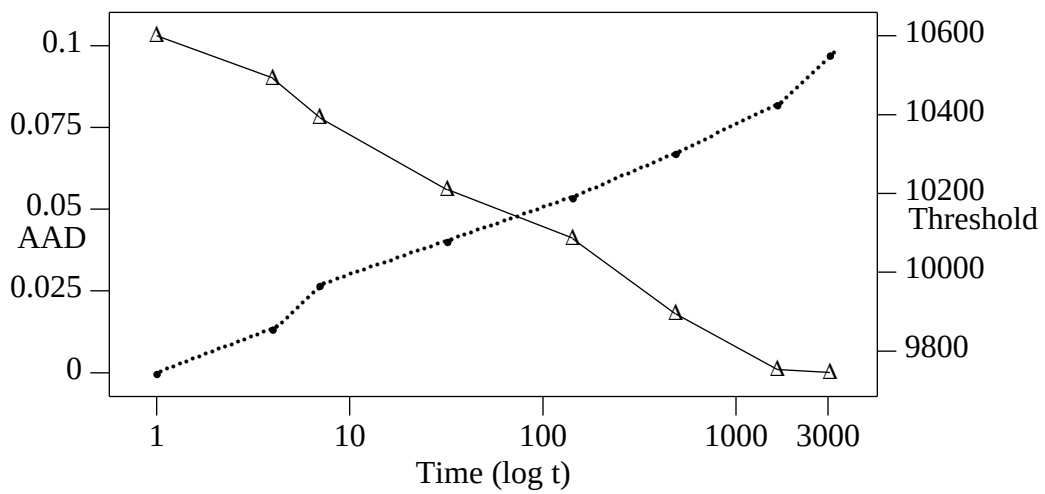


Figure 3.8. An example showing thresholding by first-order regression with $r = 3.513$, where the profile is obtained by solving a 40-object knapsack problem (with random seed of 1).

Table 3.7. Search parameters used for the example in Figure 3.8.

Iteration Index	Threshold	AAD Achieved	Incumbent	Time Spent in This GDFS	Total Time Spent
1	9745	0.103	10749	1	1
2	9857	0.090	10749	3	4
3	9968	0.078	10749	3	7
4	10080	0.056	10644	25	32
5	10191	0.041	10609	111	143
6	10303	0.018	10493	343	486
7	10427	0.001	10436	1148	1634
8	10553	0.000	10434	1436	3070

Assume that the *m*-th GDFS is the last completed GDFS. We can derive the bounds on *m* by following similar derivation for Eq. (3.24). These bounds are the same as those in Eq. (3.14).

The worst-case value of ε_m can be calculated by substituting $m_{\min}$ into Eq. (3.27). The worst-case $\varepsilon_{RTS[I,FR](T)}$ can be obtained by applying the worst-case value of ε_m (due to $m_{\min}$) to Eq. (3.27). By subtracting $\varepsilon_{OPTA^*(T)}$ in Eq. (3.7) from the worst-case $\varepsilon_{RTS[I,FR](T)}$, we can derive $\Delta\varepsilon_{RTS[I,FR](T)}$ as a function of time constraint *T* and growth rate *r*, that is,

$$\varepsilon_{RTS[I,FR](T)} - \varepsilon_{OPTA^*(T)} < \Delta\varepsilon_{RTS[I,FR](T)} = \left(\varepsilon_0 - \varepsilon_{0,OPTA^*}\right)\left(1 - \log_{t_{a^*}} T\right) + F^{worst}_{I,FR}(r, T),$$

$$\text{where } F^{worst}_{I,FR}(r, T) = 2\,\varepsilon_0 \log_{t_{a^*}} r - \varepsilon_0 \log_{t_{a^*}} \left(r - 1 + T^{-1}\right).$$

Note that $F^{worst}_{I,FR}(g,T)$ is positive. $\Delta\varepsilon_{RTS[I,FR](T)}$ is roughly logarithmic with respect to *T* since the effect of *T* in $F^{worst}_{I,FR}(g,T)$ is very small and negligible. As a result, as *T* becomes larger, $\Delta\varepsilon_{RTS[I,FR](T)}$ will become larger. Also, $\Delta\varepsilon_{RTS[I,FR](T)}$ is roughly logarithmic with *r* when *r* is large since $F^{worst}_{I,FR}(g,T) \approx \varepsilon_0 \log_{t_{a^*}} r$ when $r \gg 1$; it is independent of *r* when *r* is small since $F^{worst}_{I,FR}(g,T) \approx \varepsilon_0 \log_{t_{a^*}} T$ when $r \approx 1$. When *r* is larger than 1, the bound is curvilinear

with r; consequently, there exists a best choice of r such that the bound is minimal. The following lemma shows the best choice of r for minimizing $\Delta\varepsilon_{RTS[I,FR](T)}$.

Lemma 3.4. *In terms of minimizing* $\Delta\varepsilon_{RTS[I,FR](T)}$ *in the worst case, the best growth rate is*

$$r^{*worst} = 2\left(1 - T^{-1}\right) \approx 2. \tag{3.28}$$

Proof. The lemma is proved by taking the partial derivative of $\Delta\varepsilon_{RTS[I,FR](T)}$ with respect to r, equating the partial derivative to zero, and solving the equation for the root, r^{*worst}. The approximation is acceptable because T is usually much larger than 1. □

Theorem 3.5. *In terms of* r^{*worst}, $\Delta\varepsilon_{RTS[I,FR](T)}$ *in the worst case is bounded above by*

$$\Delta\varepsilon_{RTS[I,FR](T)}(r^{*worst}) < \left(\varepsilon_0 - \varepsilon_{0,OPTA^*}\right)\left(1 - \log_{t_{a^*}} T\right) + 2\,\varepsilon_0 \log_{t_{a^*}} 2\,.$$

Proof. By using r^{*worst}, $\Delta\varepsilon_{RTS[I,FR](T)}$ can be rewritten as

$$\begin{aligned}\Delta\varepsilon_{RTS[I,FR](T)}(r^{*worst}) &= \left(\varepsilon_0 - \varepsilon_{0,OPTA^*}\right)\left(1 - \log_{t_{a^*}} T\right) \\ &\quad + 2\,\varepsilon_0 \log_{t_{a^*}}\left[2\left(1 - T^{-1}\right)\right] - \varepsilon_0 \log_{t_{a^*}}\left(1 - T^{-1}\right) \\ &< \left(\varepsilon_0 - \varepsilon_{0,OPTA^*}\right)\left(1 - \log_{t_{a^*}} T\right) + 2\,\varepsilon_0 \log_{t_{a^*}} 2.\end{aligned}$$ □

Theorem 3.6. *Using thresholding alone, RTS using the first-order regression strategy has the same worst-case behavior as RTS using the linear gradient strategy when*

$$g = \log_{t_d} r. \tag{3.29}$$

Proof. By substituting Eq. (3.29) into Eq. (3.25), we conclude that $r = 2$, the same situation as in Eq. (3.28). □

For a dynamic decision strategy, it is hard to schedule time, approximation degrees, and thresholds for GDFSs early in the search since there is insufficient information for prediction. Fortunately, the above theorem shows that the earlier GDFSs can be scheduled

by the linear-gradient strategy. When the performance statistics are sufficient for prediction, the threshold for the next GDFS can then be set by the first-order regression strategy.

3.5. Strategies for Real-Time Search with Approximation and Thresholding

RTS using approximation and thresholding, denoted as RTS[A,B], selects, for the next GDFS, target AAD using strategy A and target threshold using strategy B. The resulting RTS is a sequence of approximation GDFSs with thresholding. Although such a combination is potentially beneficial, it is very difficult to analyze the behavior of approximation GDFSs with thresholding.

Selections of approximation degrees and thresholds can affect each other, and the proper choice of one cannot be made independently of the other. In this thesis, we adopt a heuristic strategy that uses approximation pruning as a primary pruning mechanism and thresholding as a secondary mechanism. The reason for this choice is that approximation pruning tends to improve the incumbent while thresholding tends to improve the minimum lower bound.

In our study, we only consider the linear-gradient strategy as the primary decision strategy in approximation pruning because it is difficult to select a proper threshold when a dynamic approximation pruning strategy is used. Two secondary decision strategies are studied: *moving-grid* (MG) thresholding and *pruning-ratio* (PR) thresholding. The corresponding RTSs are denoted as RTS[LG,MG] and RTS[LG,PR]. They are difficult to analyze due to the lack of a good model; therefore, in this section we briefly discuss the rationale of these strategies.

3.5.1. Moving-grid strategy

The basic idea of the moving-grid strategy is to select a threshold between the previous final pruning value and the current initial pruning value. A new threshold is computed from information on the previous threshold, the final incumbent, and the AAD achieved.

Define width δ_k in the k-th GDFS as the width of the range between the previous final pruning value, $z_{k-1}/(1+\varepsilon_{k-1})$, and the current initial pruning value, $z_{k-1}/(1+\varepsilon_k)$. That is,

$$\delta_k = del\ \frac{z_{k-1}}{1+\varepsilon_k} - \frac{z_{k-1}}{1+\varepsilon_{k-1}}.$$

If the $(k-1)$-th GDFS is scheduled with a threshold θ_{k-1} that is too small such that approximation dominates thresholding, then θ_k should be selected to be larger than the previous final pruning value, that is,

$$\theta_k = \frac{z_{k-1}}{1+\varepsilon_{k-1}} + k\ g\ \delta_k \tag{3.30}$$

where g is the gradient factor. The first term on the right-hand side is the previous final pruning value, while the second term forces the threshold to the k-th grid of width δ_k. The underlying idea is to set the threshold closer to the initial pruning value as the number of completed GDFSs increases. At that point, we put less weight on thresholding.

If neither approximation nor thresholding dominates the other in the $(k-1)$-th GDFS, then θ_k can be set by adding an increment that depends on width δ_k and gradient g.

$$\theta_k = \theta_{k-1} + g\ \delta_k. \tag{3.31}$$

Note that it is impossible for thresholding to dominate approximation since we select thresholds after approximation degrees are selected.

An example showing RTS[LG,MG] with $g = 0.1$ for solving the 40-object knapsack problem (with random seed of 1) is shown in Figure 3.9 and Table 3.8. The root node has a lower bound of 9634, an upper bound of 10749, and an achieved AAD of 0.1157. In the beginning, transient data are not used in prediction; as a result, thresholds in the earlier GDFSs are inactive. When the degree of approximation is zero, the threshold is also inactive, so we can find an optimal solution. The profile of RTS[LG,MG] is similar to that of RTS[LG,I] since thresholding serves as minor tuning.

It is interesting to note that θ_k computed in Eq. (3.30) and that in Eq. (3.31) are equivalent in the following sense. Consider the case when the width is divided into $\lceil 1/g \rceil$ grids. Threshold θ_k defined in Eq. (3.30) is simply selected to be the k-th grid, whereas θ_k defined in Eq. (3.31) is moved from the $(k-1)$-th grid to the k-th grid.

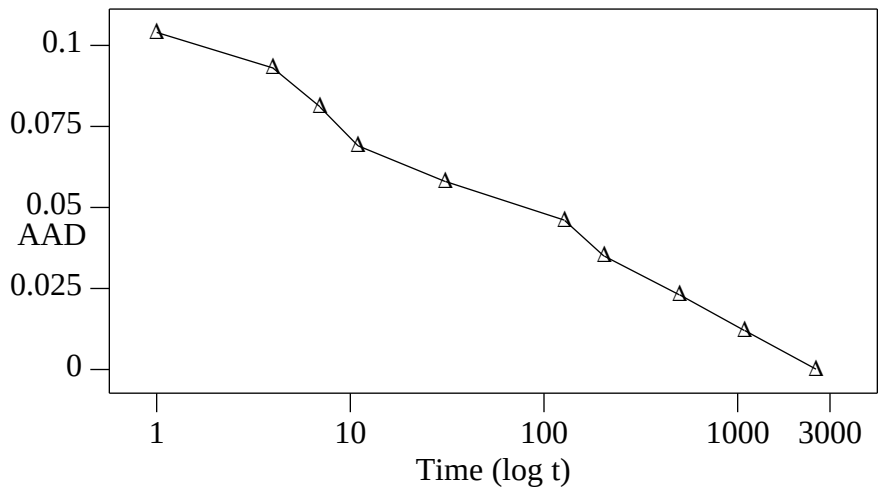


Figure 3.9. An example to illustrate RTS[LG,MG] with $g = 0.1$, where the profile is obtained by solving a 40-object knapsack problem (with random seed of 1).

Table 3.8. Search parameters used for the example in Figure 3.9.

Iteration Index	AAD Achieved	Threshold	Incumbent	Time Spent in This GDFS	Total Time Spent
1	0.104	∞	10749	1	1
2	0.093	∞	10749	3	4
3	0.081	∞	10749	3	7
4	0.069	∞	10715	4	11
5	0.058	∞	10644	20	31
6	0.046	∞	10644	97	128
7	0.035	10286	10493	77	205
8	0.023	10255	10493	297	502
9	0.012	10361	10493	584	1086
10	0.000	∞	10434	1449	2535

3.5.2. Pruning-ratio strategy

The basic idea of the pruning-ration strategy is to select a threshold to be in between the current initial pruning value and the predicted current final pruning value, where the prediction is based on the first-order regression on the relation between the incumbents obtained at the end of each of the previous GDFSs and the times taken in these GDFSs. The threshold for the *k*-th GDFS is

$$\theta_k = w_1 \frac{\hat{z}_k}{1+\varepsilon_k} + w_2 \frac{z_{k-1}}{1+\varepsilon_k}$$

where $\hat{z}_k$ is the predicted final incumbent in the *k*-th GDFS, and w_1 and w_2 are weights satisfying $0 \le w_1, w_2 \le 1$ and $w_1 + w_2 = 1$. Note that the first term (excluding the weight) in the above equation is the predicted final pruning value of the *k*-th GDFS, while the second term (excluding the weight) is the initial pruning value of the *k*-th GDFS. If we want thresholding to prune more search nodes, w_1 should be large, and θ_k, small; on the other hand, if we want approximation to prune more nodes, w_2 should be large, and θ_k, small.

Proper selection of w_1 and w_2 is problem dependent. Define a *pruning ratio p* to be the ratio of the time when approximation is dominant to the total execution time of this GDFS. When $p = 0$, the threshold should be the expected final pruning value of the *k*-th GDFS, that is, $\theta_k = \hat{z}_k / 1 + \varepsilon_k$. When $p = 1$, the threshold should be the initial pruning value of the *k*-th GDFS, that is, $\theta_k = z_{k-1} / 1 + \varepsilon_k$. Moreover, the threshold value is linear with logarithmic time. After extensive experimentation, we found the following choices of w_1 and w_2.

$$w_1 = 1 - \frac{\log(1 - p + p\,r)}{\log r},$$

$$w_2 = \frac{\log(1 - p + p\,r)}{\log r}.$$

In our experiments, we do not put any bias on approximation or thresholding, and select the pruning ratio *p* to be 0.5.

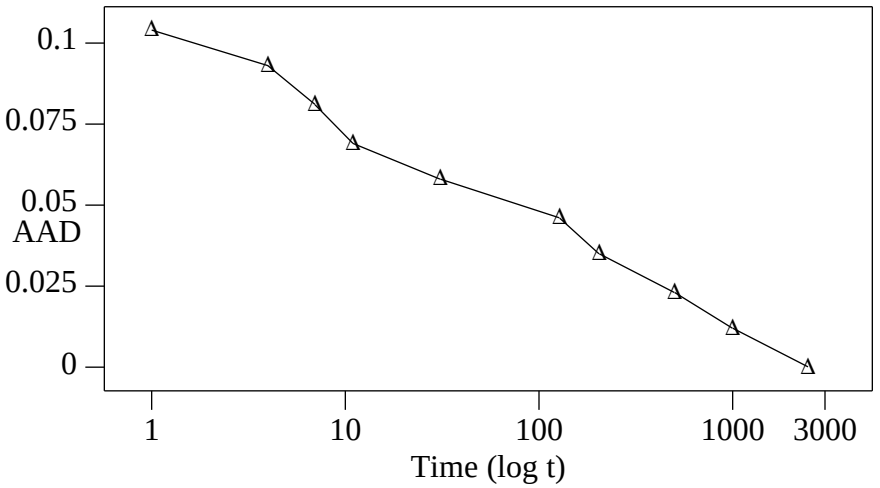


Figure 3.10. An example to illustrate RTS[LG,PR] with $g = 0.1$, where the profile is obtained by solving a 40-object knapsack problem (with random seed of 1).

Table 3.9. Search parameters used for the example in Figure 3.10.

Iteration Index	AAD Achieved	Threshold	Incumbent	Time Spent in This GDFS	Total Time Spent
1	0.104	∞	10749	1	1
2	0.093	∞	10749	3	4
3	0.081	∞	10749	3	7
4	0.069	∞	10715	4	11
5	0.058	∞	10644	20	31
6	0.046	∞	10644	97	128
7	0.035	10286	10493	77	205
8	0.023	10255	10493	297	502
9	0.012	10347	10493	497	999
10	0.000	∞	10434	1449	2448

An example to illustrate RTS[LG,PR] with $g = 0.1$ for solving a 40-object knapsack problem (with random seed of 1) is shown in Figure 3.10 and Table 3.9. The root node has a lower bound of 9634, an upper bound of 10749, and an achieved AAD of 0.1157. In the beginning, transient data are not used in prediction; as a result, thresholds in the earlier GDFSs are inactive. When the degree of approximation is zero, thresholding is also inactive, so we can find an optimal solution. It is interesting to notice that the profile of RTS[LG,PR] is similar to that of RTS[LG,MR] in Figure 3.9 and Table 3.8, and only the data for iteration 9 are different.

3.6. Experimental Results

The performance of RTS with static and dynamic strategies is studied by simulations of the symmetric traveling-salesman problem satisfying the triangular inequality, the production-planning problem, the knapsack problem, and the vertex-cover problem [18]. Due to space limitation, we cannot present all of our experimental results. Instead, we will

only show the profiles of RTS solving a random instance of each of these four problems, and show a listing of AADs and EADs achieved by RTS under various time constraints for solving fifty random instances of each of these problems.

In this section, we first describe the test problems and their implementations. We then show and discuss experimental results for all of the strategies studied in this thesis. We also show empirically the sensitivity of the parameters chosen in our strategies.

The traveling salesman problem is to find the shortest cyclic tour covering all cities. In the traveling salesman problem studied, all cities are mapped onto a Cartesian plane. For every two cities, a path is then created with a probability depending on the connectivity of the cities. The lower bound is computed using the well-known spanning-tree heuristics, whereas the upper bound is computed using a steepest-descent hill-climbing algorithm. Let TSP(x;y,z) denote a random traveling salesman problem instance with y cities, connectivity of x, and random seed z. When z is not specified, it denotes a set of random instances. Note that TSP(1;y,z) has fully connected cities and is a dense COP because any tour is a feasible solution.

The production-planning problem is to find a minimal-cost production schedule. In the production planning problem studied, all problem parameters, including capacities, requirements, production costs, inventory costs, and start-up costs, are generated randomly. The lower bound is evaluated by finding the minimal-cost schedule and by ignoring the set-up costs. The upper bound is computed using a steepest-descent hill-climbing algorithm. Let PP(x;y,z) denote a random production-planning problem instance with requirements x, y time frames to schedule, and random seed z. When z is not specified, it denotes a set of random instances.

The knapsack problem is to find a maximal-profit set of objects such that their total weight does not exceed a given bound. In the knapsack problem studied, the profit of an object is generated based on its weight, $p_i = k_i\ w_i$, where p_i, k_i, and w_i are the profit, random constant, and random weight of object i, respectively. Note that the knapsack problem is a maximization problem and is transformed into its dual minimization problem in this thesis. The lower bound in this dual problem is evaluated by finding the penalty of packing into the remaining space of the sack, a virtual object whose profit-to-weight ratio is equal to the maximum of those of unselected objects. The upper bound is computed by a steepest-descent hill-climbing algorithm. Let KS(x;y,z) be a knapsack problem instance with y objects and random seed z. The weight limit of the sack is a fraction x of the total weight of all objects. When z is not specified, it denotes a set of random instances.

The vertex-cover problem is to find a minimal set of vertices such that all edges have at least one of their vertices covered by this set. In the vertex-cover problem studied, an edge is created with a certain probability for each possible pair of two vertices. The lower bound is evaluated by finding the minimal number of vertices so that the total number of their edges is equal to the number of uncovered edges. The upper bound is computed using a steepest-descent hill-climbing algorithm. Let VC(x;y,z) be a random vertex-cover problem instance with y vertices and random seed z. The probability of creating an edge between two vertices is equal to x. When z is not specified, it denotes a set of random instances.

The bounding functions used in our test problems may not be the best; however, they do not affect the validity of our experimental results, because all algorithms use the same bounding function and the effects on them are similar. Also, in order for us to carry out

extensive tests on different problem instances, the problem size used in our experiments may not be the largest possible.

In the following, we show our experimental results on a random instance of each of the four problems: KS(0.5;80,1), PP(3;18,2), TSP(1;22,3), and VC(0.1;40,4). The search statistics of these problem instances are summarized in Table 3.10. Several observations and comments are made regarding the random problem instances and their statistics are shown in Table 3.10.

Table 3.10. Strategic parameters used in RTS and the search statistics of four random problem instances.

Problem	Strategic Parameters		Search Statistics					
	g	r	t_{a^*}	t_d	z^*	z_0	b_0	λ
KS(0.5;80,1)	.111	3.513	47053	83573	25456	25800	22988	1.139
PP(3;18,2)	.121	3.513	21506	33175	1314	1633	363	1.335
TS(1;22,3)	.117	3.513	32370	44946	405.379	503.684	331.894	2.338
VC(0.1;40,4)	.098	3.513	13602	353008	22	31	14	2.125

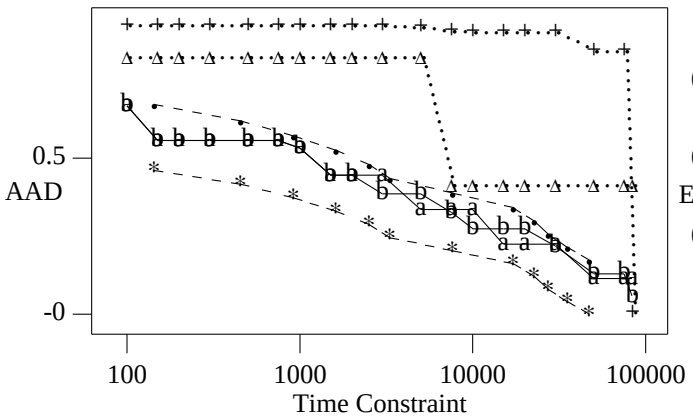
(a) For KS(0.5;80,1), PP(3;18,2), and TSP(1;22,3), GDFS performs well in the sense that the execution time of GDFS is close to that of an A* search. For VC(0.1;40,4), GDFS does not perform well since the VC problem is a discrete problem, and many nodes have equal lower bounds so that a GDFS can easily be misled and trapped in a poor subtree. One way to cope with this problem is to use a better guidance function instead of lower bounds as is in GDFS.

(b) We notice that the initial upper bounds (that is, the upper bounds at the root node) for KS(0.5;80,1), PP(3;18,2) and TSP(1;22,3) are closer to their respective optimal solution values than the initial lower bounds. This can partly explain why GDFS performs well for them.

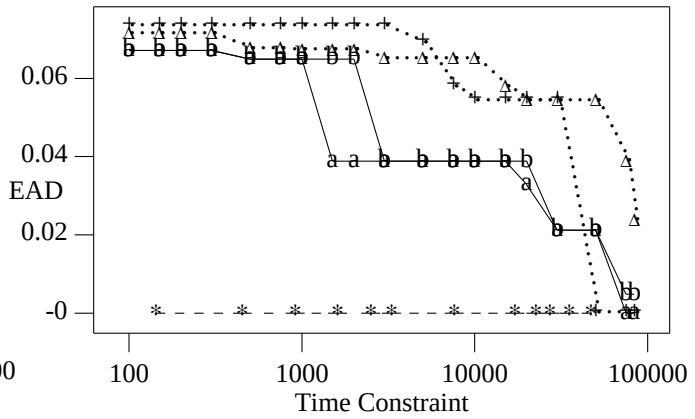
(c) We notice that $t_{a^*}^{\lambda} \gg t_d$ in all these four instances. This implies that the analysis in Section 3.4.2 should use t_d to replace $t_{a^*}^{\lambda}$; recall that $t_{a^*}^{\lambda}$ is the extrapolated execution time of GDFS, that is, the extrapolated t_d. (The explanation for $t_{a^*}^{\lambda} > t_d$ can be found in Section 3.4.2.) Therefore, we use t_d to replace $t_{a^*}^{\lambda}$ in calculating parameters used.

(d) The linear-gradient factor *g* is calculated based on the average-case analysis in Appendix A. The growth rate *r* is calculated based on the average-case analysis in Appendix A. These two strategic parameters are calculated based on the average case rather than the worst case because we empirically found that the average case generally outperforms the worst case. One possible reason is that the worst case does not happen often and can be neglected.

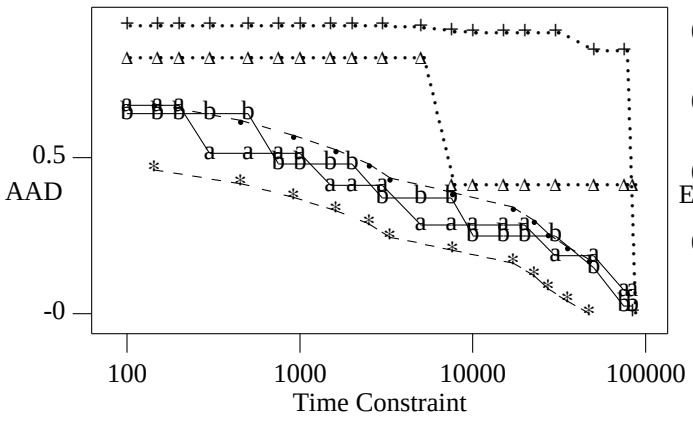
(e) A search tree representing a TSP instance branches on cities that have not been visited and enumerates all unvisited cities in generating child nodes when a node is decomposed. As a result, the search tree is fat when close to the root and narrow in the bottom. In contrast, a search node can be decomposed based on whether a city is visited or not. In this case, the branching degree is linear, representing whether a city is visited or not.



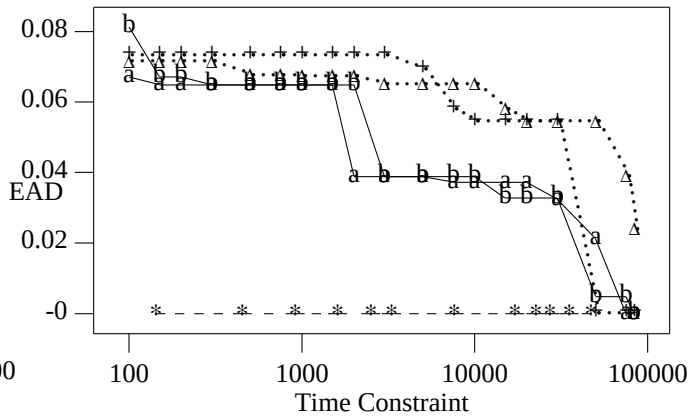
(a) AADs of RTS[LG,I] (‘a’) and RTS[FR,I] (‘b’).



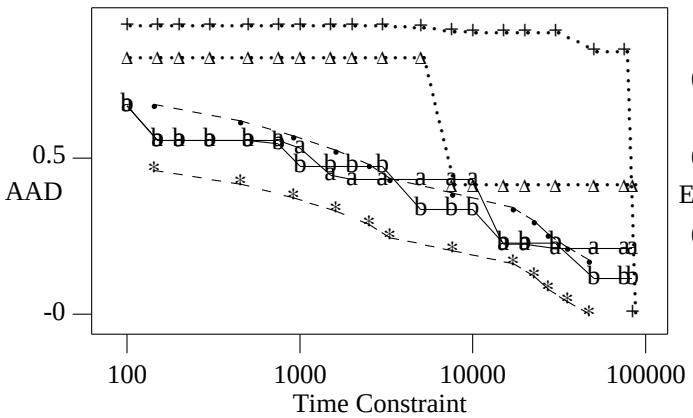
(b) EADs of RTS[LG,I] (‘a’) and RTS[FR,I] (‘b’).



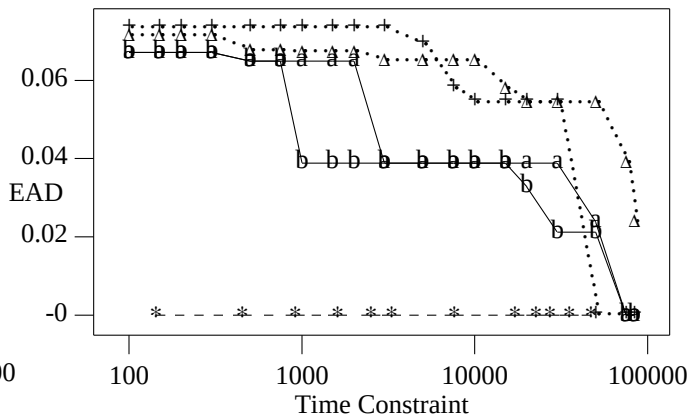
(c) AADs of RTS[I,LG] (‘a’) and RTS[I,FR] (‘b’).



(d) EADs of RTS[I,LG] (‘a’) and RTS[I,FR] (‘b’).

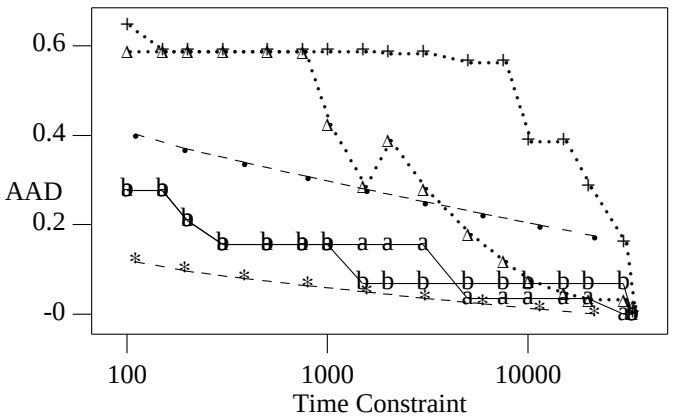


(e) AADs of RTS[LG,MG] (‘a’) and RTS[LG,PR] (‘b’).

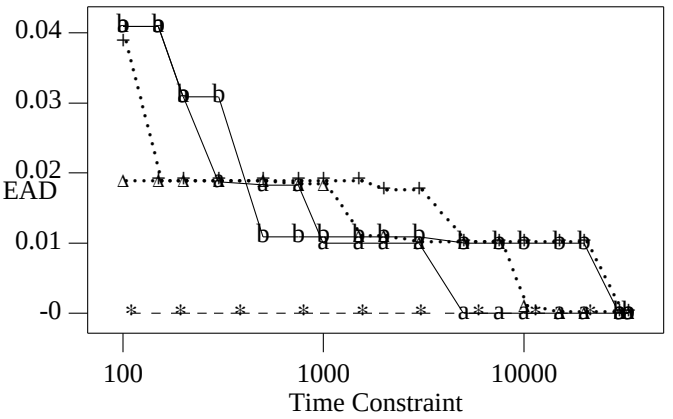


(f) EADs of RTS[LG,MG] (‘a’) and RTS[LG,PR] (‘b’).

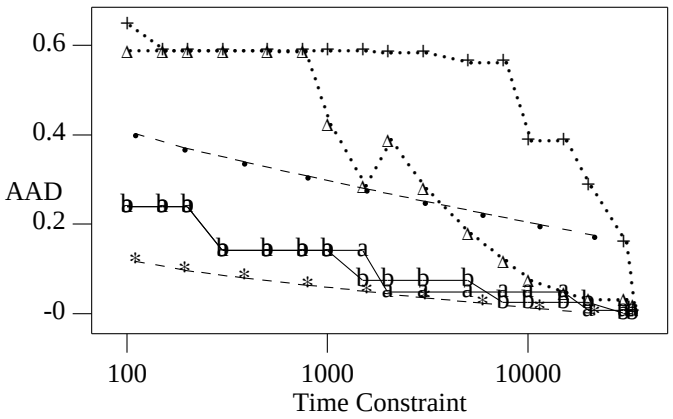
Figure 3.11. Performance of the RTS with static and dynamic strategies in solving KS(0.5;80,1), where the left column is for AADs and the right column for EADs, and ‘+’ for GDFS, ‘Δ’ for Lawler and Wood’s algorithm, ‘*’ for OPTA*, and ‘•’ for the best upper bound in the worst case.



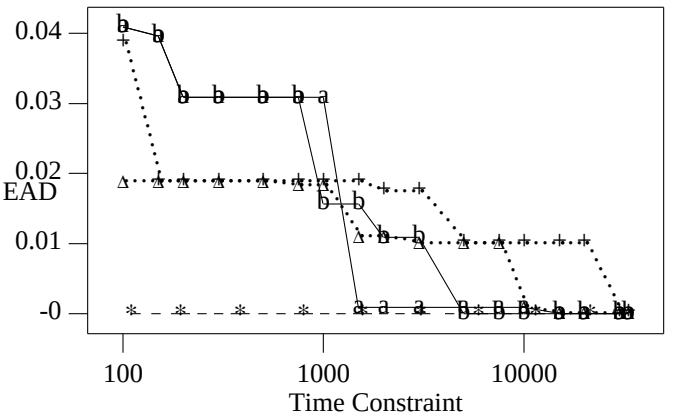
(a) AADs of RTS[LG,I] (‘a’) and RTS[FR,I] (‘b’).



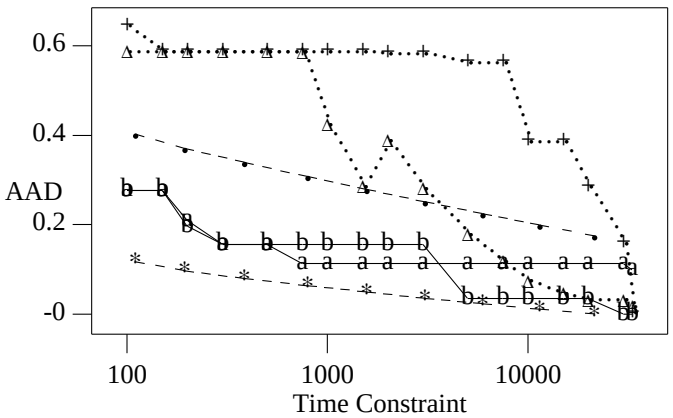
(b) EADs of RTS[LG,I] (‘a’) and RTS[FR,I] (‘b’).



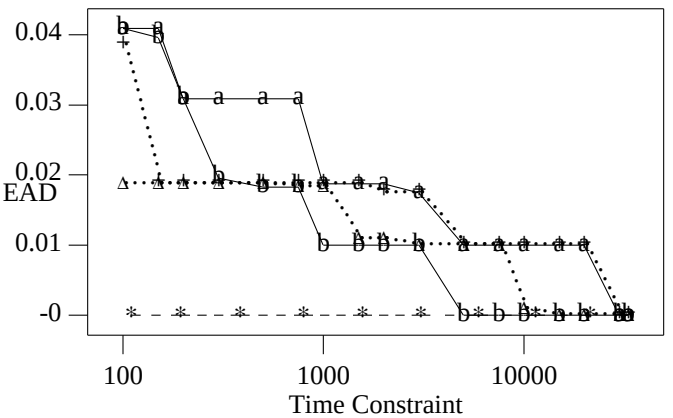
(c) AADs of RTS[I,LG] (‘a’) and RTS[I,FR] (‘b’).



(d) EADs of RTS[I,LG] (‘a’) and RTS[I,FR] (‘b’).

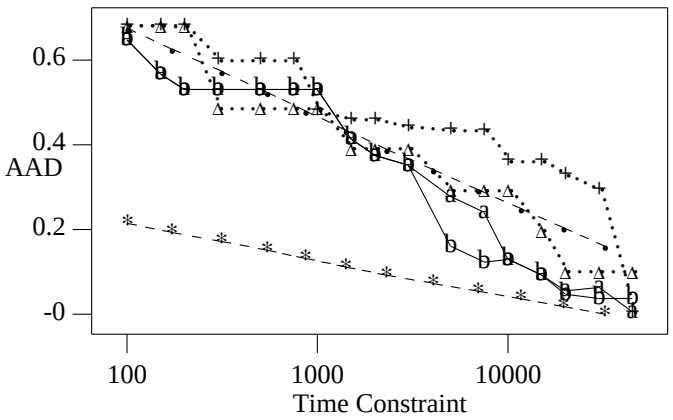


(e) AADs of RTS[LG,MG] (‘a’) and RTS[LG,PR] (‘b’).

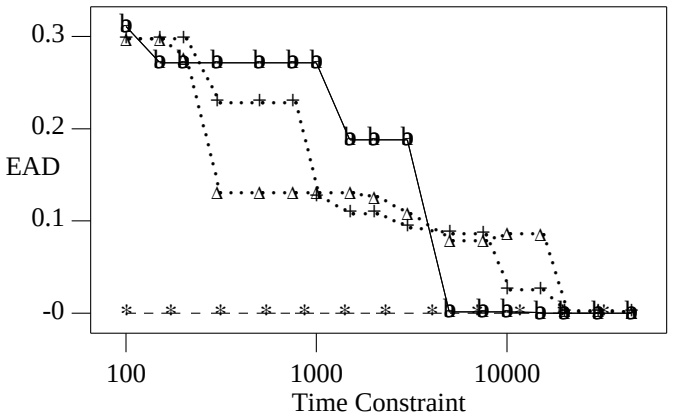


(f) EADs of RTS[LG,MG] (‘a’) and RTS[LG,PR] (‘b’).

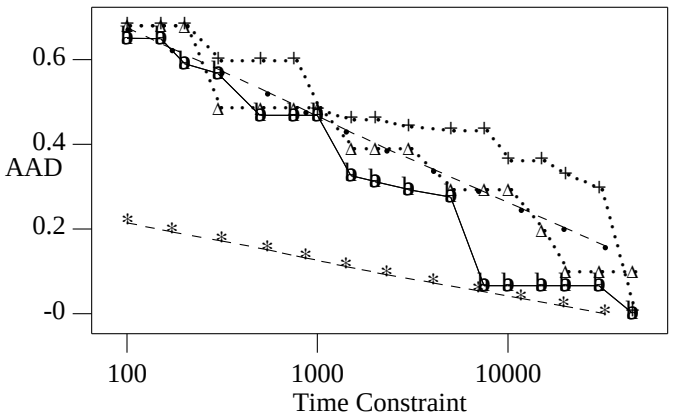
Figure 3.12. Performance of the RTS with static and dynamic strategies in solving PP(3;18,2), where the left column is for AADs and the right column for EADs, and ‘+’ for GDFS, ‘Δ’ for Lawler and Wood’s algorithm, ‘*’ for OPTA*, and ‘•’ for the best upper bound in the worst case.



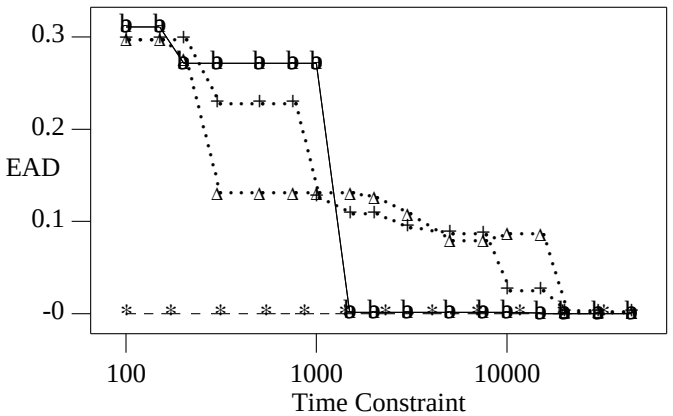
(a) AADs of RTS[LG,I] (‘a’) and RTS[FR,I] (‘b’).



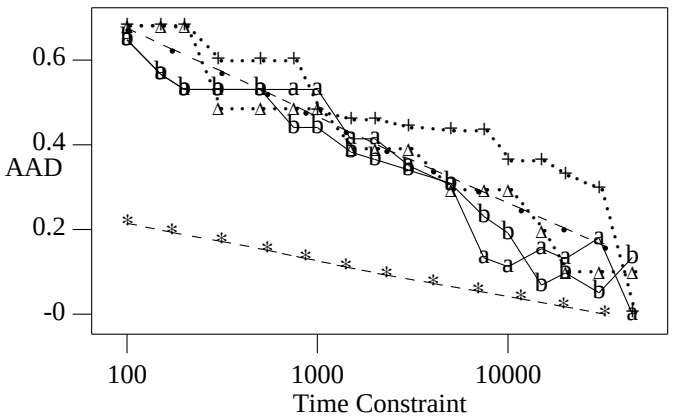
(b) EADs of RTS[LG,I] (‘a’) and RTS[FR,I] (‘b’).



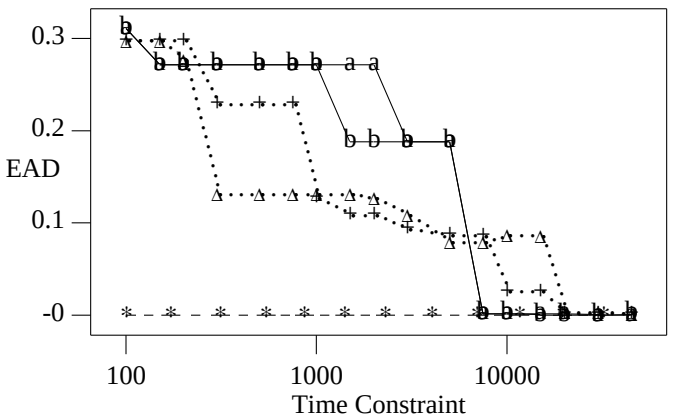
(c) AADs of RTS[I,LG] (‘a’) and RTS[I,FR] (‘b’).



(d) EADs of RTS[I,LG] (‘a’) and RTS[I,FR] (‘b’).

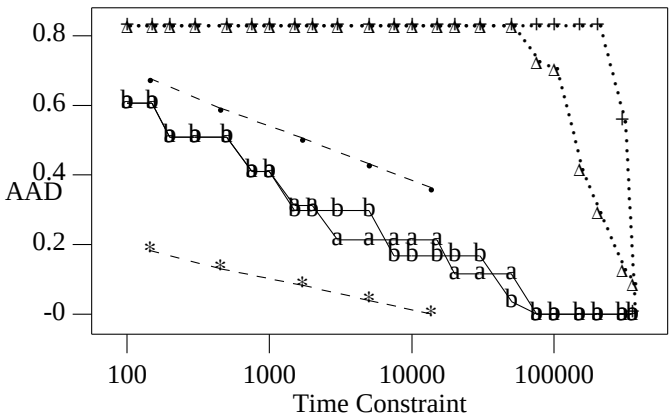


(e) AADs of RTS[LG,MG] (‘a’) and RTS[LG,PR] (‘b’).

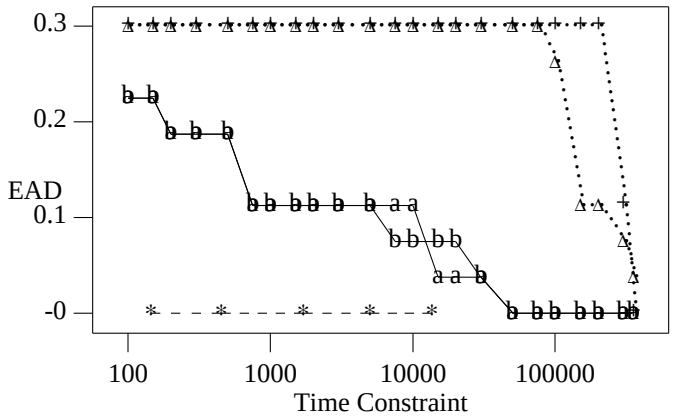


(f) EADs of RTS[LG,MG] (‘a’) and RTS[LG,PR] (‘b’).

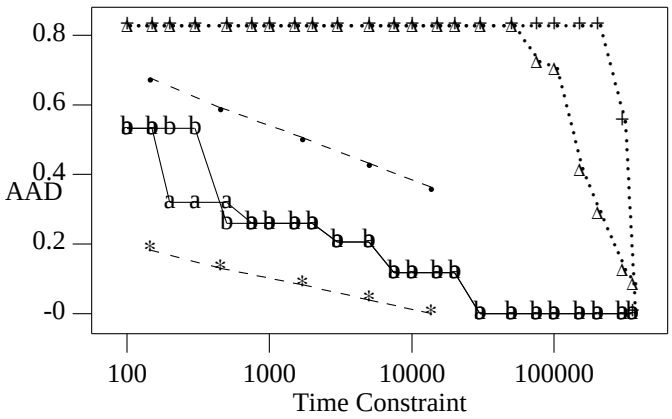
Figure 3.13. Performance of the RTS with static and dynamic strategies in solving TSP(1;22,3), where the left column is for AADs and the right column for EADs, and ‘+’ for GDFS, ‘Δ’ for Lawler and Wood’s algorithm, ‘*’ for OPTA*, and ‘•’ for the best upper bound in the worst case.



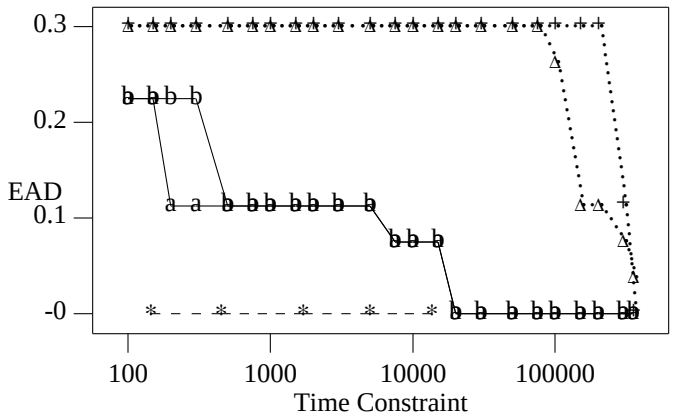
(a) AADs of RTS[LG,I] (‘a’) and RTS[FR,I] (‘b’).



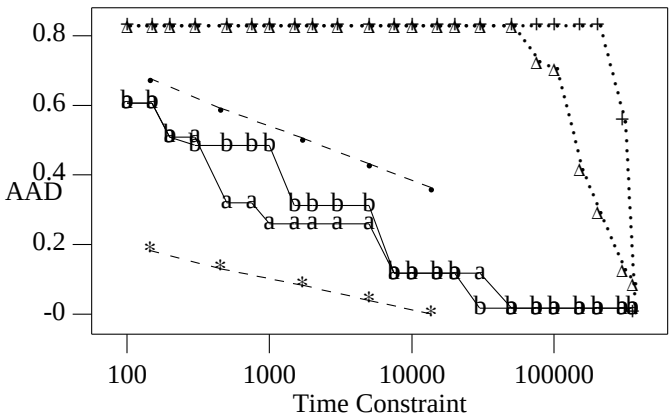
(b) EADs of RTS[LG,I] (‘a’) and RTS[FR,I] (‘b’).



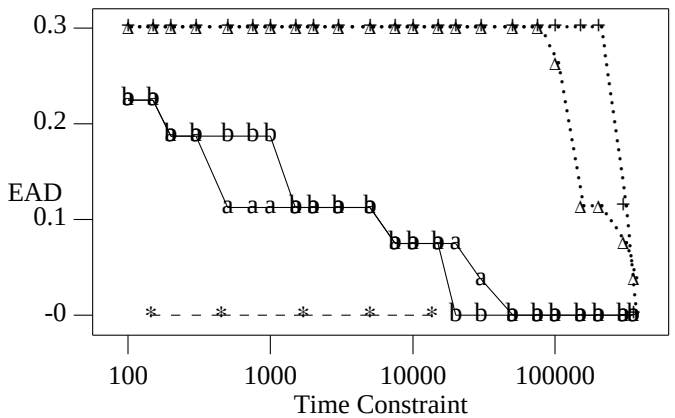
(c) AADs of RTS[I,LG] (‘a’) and RTS[I,FR] (‘b’).



(d) EADs of RTS[I,LG] (‘a’) and RTS[I,FR] (‘b’).



(e) AADs of RTS[LG,MG] (‘a’) and RTS[LG,PR] (‘b’).



(f) EADs of RTS[LG,MG] (‘a’) and RTS[LG,PR] (‘b’).

Figure 3.14. Performance of the RTS with static and dynamic strategies in solving VC(0.1;40,4), where the left column is for AADs and the right column for EADs, and ‘+’ for GDFS, ‘Δ’ for Lawler and Wood’s algorithm, ‘*’ for OPTA*, and ‘•’ for the best upper bound in the worst case.

The performances of RTS using various strategies to solve KS(0.5;80,1), PP(3;18,2), TSP(1;22,3), and VC(0.1;40,4) are shown in Figures 3.11, 3.12, 3.13 and 3.14, respectively. Several important observations and comments regarding these graphs are summarized as follows.

(a) It is important to note in these graphs that the time axis is logarithmically scaled so that the major portion of time is on the right-hand side of each graph. Further, we are particularly interested in the performance in the middle portion of the graph because (i) the left portion corresponds to small time constraints and the search behavior is not stable, and (ii) the right portion corresponds to large time constraints that enable GDFS to almost find an optimal solution.

(b) It is interesting to note that the AAD curves (‘‘*’’) of OPTA* are always lower than those of the others. This happens because the AADs achieved by OPTA* are the *absolute* lower bounds. Also, note that the EAD curves (‘‘*’’) of OPTA* are horizontal lines at $\varepsilon = 0$. This happens because OPTA* finds an optimal solution at the root and its achieved EAD is always 0.

(c) The performance ‘‘upper bound’’ (‘‘•’’) is the best upper bound in the worst case. This ‘‘upper bound’’ is not a true bound and only serves as a reference that is the best in the worst case. Sometimes, the curves will exceed this ‘‘upper bound.’’

(d) Most of the curves lie inside the band delimited by the absolute lower bounds (‘‘*’’) and the ‘‘upper bound’’ (‘‘•’’), especially for PP(3;18,2) and VC(0.1;40,4).

(e) RTS applied to TSP(1;22,3) checks if a GDFS is the last search in the sequence. We empirically found that an RTS performs better than one without checking whether a

GDFS is the last one in the sequence. The reason is that regression, when RTS is applied to solve a TSP, gives a good prediction whether a specific GDFS is the last. Note that good regression is only necessary (but not sufficient) for good prediction. For the other three problems, we do not apply this prediction since it worsens the performance of RTS.

Tables 3.11, 3.12, 3.13, and 3.14 show the performance of RTS using various strategies to solve fifty random instances of each of VC(0.1;30), TSP(1;18), PP(3;18), and KS(0.5;60), respectively, under various time constraints. The random seeds used are obtained by the formula 1000+7*k, k=1,...,50. Several important comments regarding these tables are summarized as follows.

(a) The value *x* in the leftmost column is the logarithmic value of time normalized with respect to time t_d spent by a GDFS for finding an optimal solution. That is, the time τ imposed on a search is equal to $\tau = t_d{}^x$.

(b) The normalized AAD and EAD are normalized with respect to the AAD obtained in the root node. By normalizing AADs and EADs, we can find average solution quality across problem instances because normalized AADs and EADs are between 0 and 1.

(c) The numbers in these tables are averages of normalized AADs or EADs over fifty random instances.

Table 3.11. Summary of averages of normalized AADs and EADs achieved by RTS using various strategies for solving fifty random instances of VC(0.1;30,z) (z=1000+7*k for k=1,...,50) under ten time constraints.

Averages of Normalized AADs								
Time	GDFS	L&W	RTS[LG,I]	RTS[FR,I]	RTS[I,LG]	RTS[I,FR]	RTS[LG,PR]	RTS[LG,MG]
0.45	0.752	0.752	0.507	0.519	0.445	0.445	0.524	0.516
0.50	0.752	0.752	0.460	0.483	0.411	0.411	0.482	0.470
0.55	0.752	0.749	0.407	0.438	0.372	0.372	0.438	0.416
0.60	0.752	0.747	0.360	0.401	0.315	0.315	0.378	0.353
0.65	0.752	0.734	0.315	0.378	0.248	0.248	0.331	0.294
0.70	0.752	0.733	0.269	0.341	0.204	0.204	0.290	0.252
0.75	0.739	0.669	0.174	0.251	0.126	0.117	0.185	0.162
0.80	0.700	0.576	0.124	0.203	0.098	0.082	0.149	0.135
0.85	0.647	0.439	0.077	0.160	0.042	0.032	0.118	0.114
0.90	0.586	0.222	0.044	0.111	0.023	0.022	0.070	0.071
Averages of Normalized EADs								
Time	GDFS	L&W	RTS[LG,I]	RTS[FR,I]	RTS[I,LG]	RTS[I,FR]	RTS[LG,PR]	RTS[LG,MG]
0.45	0.255	0.255	0.154	0.158	0.151	0.151	0.164	0.164
0.50	0.255	0.255	0.143	0.146	0.137	0.137	0.147	0.144
0.55	0.255	0.253	0.121	0.134	0.119	0.119	0.125	0.119
0.60	0.255	0.254	0.101	0.118	0.099	0.099	0.108	0.103
0.65	0.255	0.248	0.083	0.110	0.069	0.069	0.092	0.082
0.70	0.255	0.245	0.068	0.095	0.055	0.054	0.071	0.067
0.75	0.249	0.219	0.028	0.053	0.027	0.023	0.039	0.037
0.80	0.222	0.178	0.013	0.037	0.012	0.004	0.026	0.025
0.85	0.185	0.129	0.004	0.026	0.003	0.001	0.013	0.009
0.90	0.147	0.052	0.001	0.009	0.000	0.000	0.007	0.006

Table 3.12. Summary of averages of normalized AADs and EADs achieved by RTS using various strategies for solving fifty random instances of TSP(1;18,z) (z=1000+7*k for k=1,...,50) under ten time constraints.

Averages of Normalized AADs								
Time	GDFS	L&W	RTS[LG,I]	RTS[FR,I]	RTS[I,LG]	RTS[I,FR]	RTS[LG,PR]	RTS[LG,MG]
0.45	0.645	0.658	0.536	0.536	0.507	0.507	0.534	0.531
0.50	0.645	0.641	0.510	0.512	0.459	0.459	0.499	0.500
0.55	0.644	0.624	0.466	0.474	0.419	0.419	0.452	0.456
0.60	0.638	0.600	0.439	0.444	0.387	0.387	0.417	0.425
0.65	0.630	0.558	0.377	0.401	0.333	0.332	0.358	0.374
0.70	0.623	0.505	0.329	0.365	0.300	0.302	0.319	0.334
0.75	0.594	0.363	0.266	0.292	0.199	0.214	0.241	0.241
0.80	0.568	0.282	0.206	0.244	0.172	0.160	0.196	0.194
0.85	0.511	0.224	0.157	0.199	0.121	0.120	0.149	0.158
0.90	0.443	0.160	0.114	0.135	0.078	0.076	0.109	0.119
Averages of Normalized EADs								
Time	GDFS	L&W	RTS[LG,I]	RTS[FR,I]	RTS[I,LG]	RTS[I,FR]	RTS[LG,PR]	RTS[LG,MG]
0.45	0.205	0.174	0.153	0.153	0.140	0.140	0.148	0.149
0.50	0.205	0.164	0.143	0.143	0.112	0.112	0.134	0.136
0.55	0.204	0.152	0.115	0.119	0.096	0.096	0.100	0.113
0.60	0.198	0.146	0.107	0.104	0.082	0.082	0.093	0.099
0.65	0.191	0.124	0.075	0.085	0.063	0.063	0.073	0.081
0.70	0.185	0.107	0.057	0.073	0.043	0.043	0.056	0.066
0.75	0.162	0.052	0.035	0.048	0.015	0.016	0.025	0.028
0.80	0.141	0.026	0.013	0.022	0.010	0.010	0.014	0.011
0.85	0.096	0.012	0.003	0.008	0.002	0.002	0.004	0.007
0.90	0.057	0.009	0.000	0.000	0.001	0.001	0.001	0.001

Table 3.13. Summary of averages of normalized AADs and EADs achieved by RTS using various strategies for solving fifty random instances of PP(3;18,z) (z=1000+7*k for k=1,...,50) under ten time constraints.

Averages of Normalized AADs								
Time	GDFS	L&W	RTS[LG,I]	RTS[FR,I]	RTS[I,LG]	RTS[I,FR]	RTS[LG,PR]	RTS[LG,MG]
0.45	0.588	0.575	0.313	0.314	0.298	0.298	0.313	0.308
0.50	0.580	0.567	0.265	0.276	0.260	0.256	0.267	0.269
0.55	0.571	0.560	0.238	0.247	0.223	0.227	0.236	0.243
0.60	0.565	0.549	0.201	0.212	0.180	0.185	0.203	0.212
0.65	0.557	0.539	0.171	0.180	0.161	0.157	0.179	0.184
0.70	0.548	0.514	0.144	0.152	0.133	0.128	0.144	0.154
0.75	0.529	0.411	0.108	0.106	0.091	0.087	0.113	0.127
0.80	0.515	0.283	0.102	0.086	0.065	0.067	0.107	0.125
0.85	0.480	0.141	0.100	0.073	0.051	0.049	0.105	0.123
0.90	0.410	0.044	0.100	0.062	0.033	0.030	0.105	0.123
Averages of Normalized EADs								
Time	GDFS	L&W	RTS[LG,I]	RTS[FR,I]	RTS[I,LG]	RTS[I,FR]	RTS[LG,PR]	RTS[LG,MG]
0.45	0.035	0.031	0.034	0.033	0.032	0.032	0.034	0.035
0.50	0.032	0.027	0.029	0.027	0.027	0.027	0.029	0.029
0.55	0.029	0.025	0.023	0.023	0.023	0.023	0.024	0.025
0.60	0.027	0.022	0.018	0.017	0.019	0.020	0.020	0.022
0.65	0.024	0.019	0.016	0.015	0.015	0.016	0.018	0.018
0.70	0.020	0.014	0.014	0.011	0.012	0.012	0.015	0.016
0.75	0.014	0.008	0.010	0.006	0.006	0.006	0.010	0.011
0.80	0.010	0.004	0.007	0.004	0.002	0.003	0.007	0.008
0.85	0.006	0.001	0.004	0.002	0.002	0.002	0.004	0.004
0.90	0.002	0.000	0.001	0.001	0.001	0.001	0.001	0.002

Table 3.14. Summary of averages of normalized AADs and EADs achieved by RTS using various strategies for solving fifty random instances of KS(0.5;60,z) (z=1000+7*k for k=1,...,50) under ten time constraints.

Averages of Normalized AADs								
Time	GDFS	L&W	RTS[LG,I]	RTS[FR,I]	RTS[I,LG]	RTS[I,FR]	RTS[LG,PR]	RTS[LG,MG]
0.45	0.824	0.849	0.659	0.659	0.658	0.658	0.659	0.659
0.50	0.824	0.831	0.615	0.615	0.620	0.620	0.615	0.615
0.55	0.822	0.795	0.570	0.570	0.568	0.566	0.569	0.570
0.60	0.822	0.758	0.527	0.529	0.524	0.526	0.527	0.529
0.65	0.820	0.697	0.484	0.492	0.478	0.484	0.486	0.483
0.70	0.819	0.670	0.442	0.460	0.436	0.455	0.443	0.452
0.75	0.816	0.566	0.346	0.385	0.345	0.356	0.344	0.362
0.80	0.811	0.511	0.292	0.312	0.291	0.294	0.292	0.304
0.85	0.801	0.418	0.234	0.275	0.240	0.245	0.236	0.246
0.90	0.783	0.368	0.176	0.209	0.180	0.183	0.177	0.174
Averages of Normalized EADs								
Time	GDFS	L&W	RTS[LG,I]	RTS[FR,I]	RTS[I,LG]	RTS[I,FR]	RTS[LG,PR]	RTS[LG,MG]
0.45	0.039	0.036	0.046	0.046	0.047	0.047	0.046	0.046
0.50	0.038	0.035	0.040	0.040	0.040	0.040	0.040	0.040
0.55	0.037	0.034	0.039	0.040	0.038	0.038	0.039	0.040
0.60	0.036	0.031	0.035	0.036	0.035	0.037	0.035	0.037
0.65	0.035	0.030	0.032	0.033	0.032	0.034	0.032	0.033
0.70	0.034	0.028	0.031	0.031	0.029	0.030	0.031	0.032
0.75	0.031	0.024	0.027	0.027	0.027	0.026	0.027	0.027
0.80	0.027	0.022	0.025	0.025	0.025	0.024	0.025	0.024
0.85	0.018	0.020	0.021	0.022	0.022	0.021	0.021	0.021
0.90	0.005	0.017	0.018	0.016	0.017	0.016	0.018	0.017

(d) The search trees representing the TSP instances are binary. Note that the search tree for the TSP in Figure 3.13 uses a search tree whose branching degree is the number of unvisited cities.

Lawler and Wood's algorithm outperforms GDFS in terms of AAD, but performs similarly in terms of EAD. As seen in Figures 3.11-3.14 (pp. 106-109) and Tables 3.10-3.13, RTS generally outperforms GDFS and Lawler and Wood's algorithm in terms of both AAD and EAD. Further, AADs achieved by RTS are close to those of OPTA*, the absolute lower bound, for KS(0.5;80,1) and PP(3;18,2), as is seen in Figures 3.11 and 3.12. We also notice that Lawler and Wood's algorithm performs well in solving the KS problems as is seen in Table 3.13.

As found in our experimental results, static strategies perform slightly better than dynamic ones in terms of EAD but not AAD. Further, static strategies are more consistent than dynamic strategies in terms of monotonicity of solution values as search progresses. A static strategy has the property that performance is better for a larger time constraint; in contrast, a dynamic strategy may not have this property due to a lack of accurate run-time information. We also observe in Figures 3.11-3.14 that the solution values obtained by a static strategy are nonincreasing with time. Although, dynamic strategies also have this property as shown in Figures 3.11-3.14, we do not observe this behavior in experimental results not shown here.

We also notice that RTS[LG,PR] performs slightly better than RTS[LG,MG] for solving the TSP, PP, and KS problems. The former balances the significance of approximation pruning and thresholding pruning. Thresholds in the latter search grow slowly as the gap between the incumbent and the global lower bound decreases.

RTSs with approximation pruning performs similarly as RTSs with thresholding, because the change of incumbent is minor as compared to that of the global lower bound across iterations. On the other hand, when the incumbent changes rapidly and the guidance strategy is effective, RTSs with approximation pruning outperform RTSs with thresholding. This behavior is found in the PP problem. For discrete problems, RTSs with thresholding outperform RTSs with approximation because thresholding inherits discrete solution values through the levels of a search tree. This is found in the VC problem. For problems with ineffective guidance, RTSs with thresholding outperform RTSs with approximation, since the approximation strategy can be misled by the poor guidance strategy to spend unnecessary efforts in searching a subtree determined by the guidance strategy. This is found in the TSP problem. The guidance strategy used in the TSP problem is not effective because we do not check whether a search node is dominated by another node (when the same set of cities is visited in a different order).

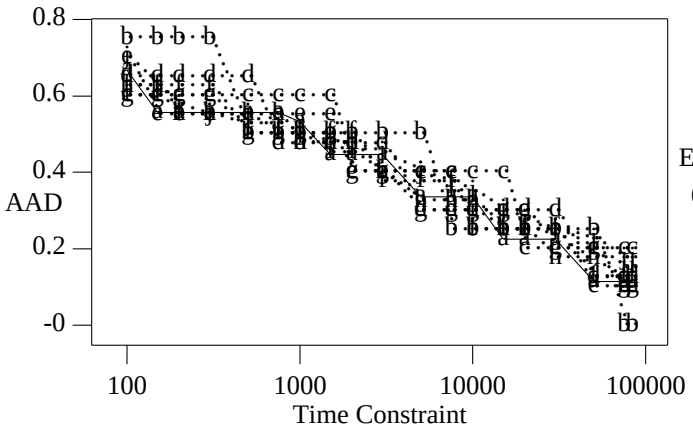
When an RTS checks if the current search is the last iteration, a static strategy becomes dynamic in the sense that it uses run-time information of performance history to predict the future performance. As a result, both static and dynamic strategies perform similarly. This is found in the TSP problem. In fact, such RTSs spend roughly $1/(1+r+r^2)$ portion of the time constraint in collecting the actual profile that is used to predict the performance of the last iteration that uses up the remaining $(r+r^2)/(1+r+r^2)$ portion of the time. When $r=1$, one third of the time is used in collecting the actual profile. When $r=3$, around 7.7% of the time is used in collecting the actual profile. Therefore, the major portion of the time constraint is normally expended in the last iteration. Checking if a GDFS is the last works well when regression is good in predicting the future behavior.

Figures 3.15 and 3.16 (pp. 119 and 120) show the sensitivity of the schedule of approximation and thresholding in KS(0.5;80,1), PP(3;18,2) and VC(0.1;40,4). Each graph consists of ten curves of various linear-gradient factors from $g = 0.25$ to $g = 0.025$. One of these ten curves is RTS[LG,I] or RTS[I,LG] used in Figures 3.11-3.13 (pp. 106-108), and is denoted by a solid curve marked ‘‘a.’’ All the others are dotted curves for easy distinction. We can see that significant amount of the time the solid curve is close to the bottom envelope of the collection of curves, which implies that the tuned RTS has good performance. Recall that the parameters used in RTSs are optimized for the average case; therefore, RTSs using such parameters may not always be the best.

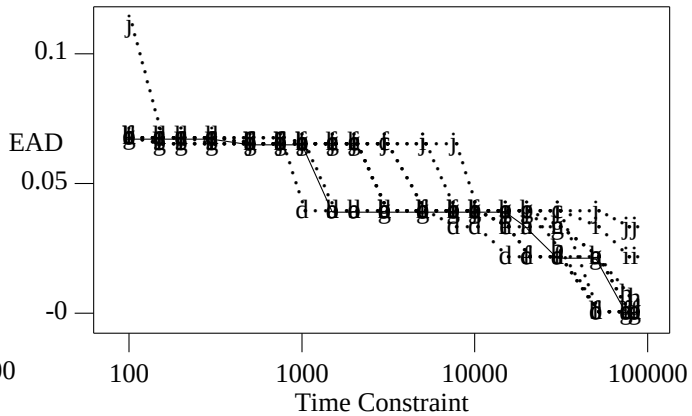
3.7. Summary

In this chapter, we have developed new search algorithms that solve dense COPs under hard time constraints. Our original objective is to find for a specific application problem the best feasible solution in the time constraint. However, this objective is not achievable as we do not have enough information before the search is completed. We have used an alternative heuristic objective that looks for the solution with the best ascertained approximation degree (AAD). (This hopefully is close to the actual approximation degree -- EAD.)

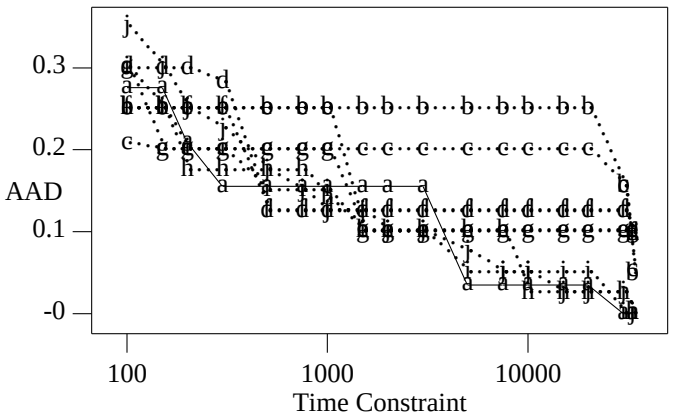
Our RTS algorithm schedules a sequence of approximation GDFSs, each searching for either a more accurate solution or a solution deeper in the search tree (or both). We study static and dynamic strategies for setting approximation degrees and thresholds. We develop an exponential model to characterize the relationship between the approximation degrees achieved and time taken. Our study shows that RTSs with proper schedules can generally outperform GDFSs and Lawler and Wood’s algorithm in terms of both EAD and AAD.



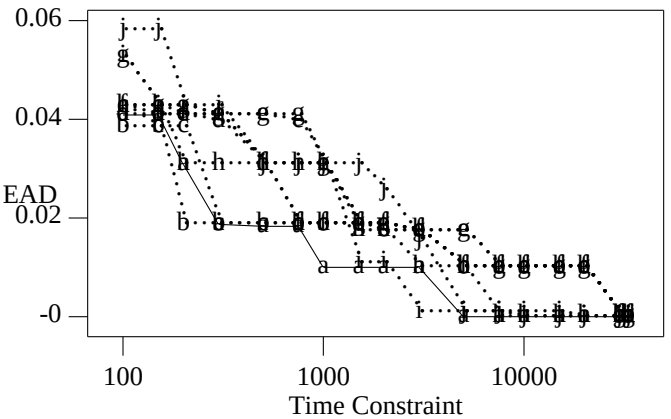
(a) AADs of various RTS[LG,I] on KS(80,1).



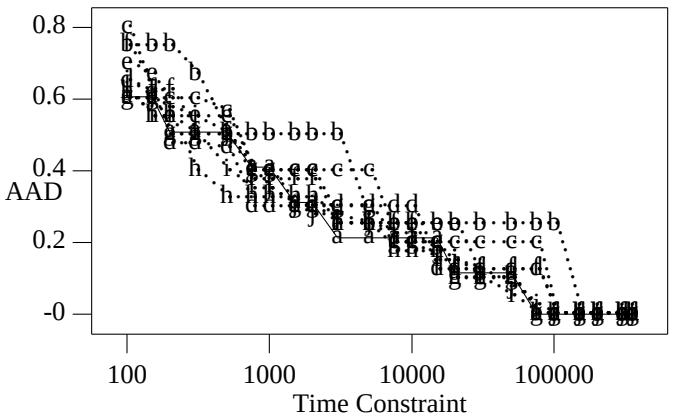
(b) EADs of various RTS[LG,I] on KS(80,1).



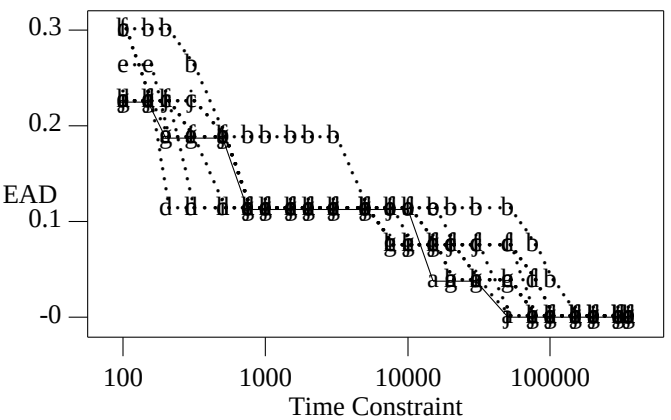
(c) AADs of various RTS[LG,I] on PP(18,2).



(d) EADs of various RTS[LG,I] on PP(18,2).

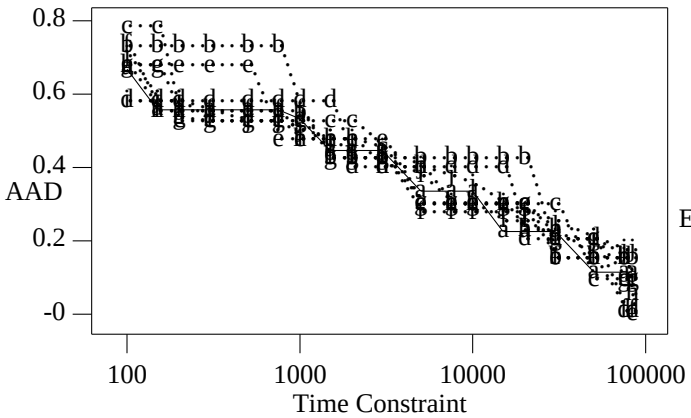


(e) AADs of various RTS[LG,I] on VC(40,4).

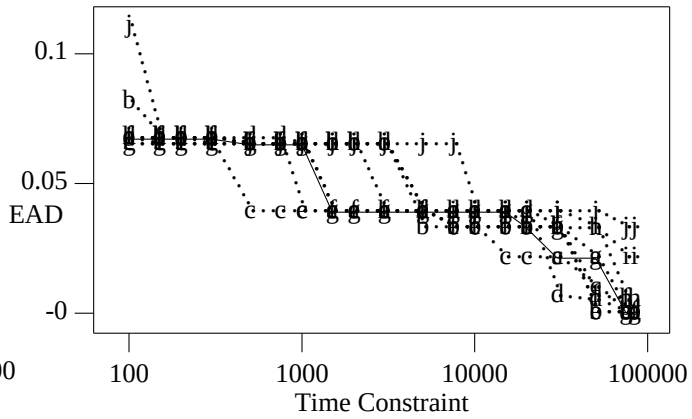


(f) EADs of various RTS[LG,I] on VC(40,4).

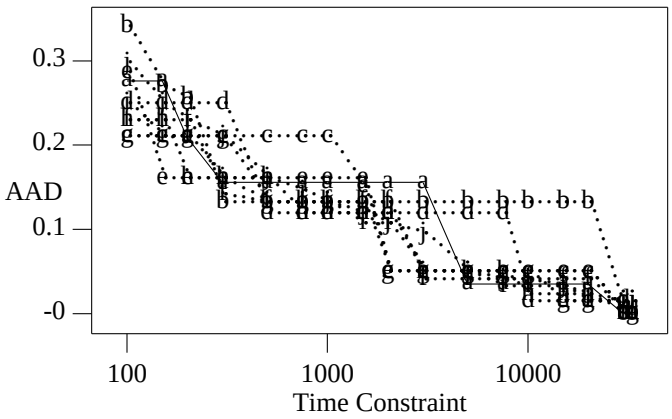
Figure 3.15. Performance of the RTS[LG,I] in solving KS(0.5;80,1), PP(3;18,2), and VC(0.1;40,4) where ‘a’ is for the g used for comparison, ‘b’ for g=0.25, ‘c’ for g=0.2, ‘d’ for g=0.175, ‘e’ for g=0.15, ‘f’ for g=0.125, ‘g’ for g=0.1, ‘h’ for g=0.075, ‘i’ for g=0.05, ‘j’ for g=0.025.



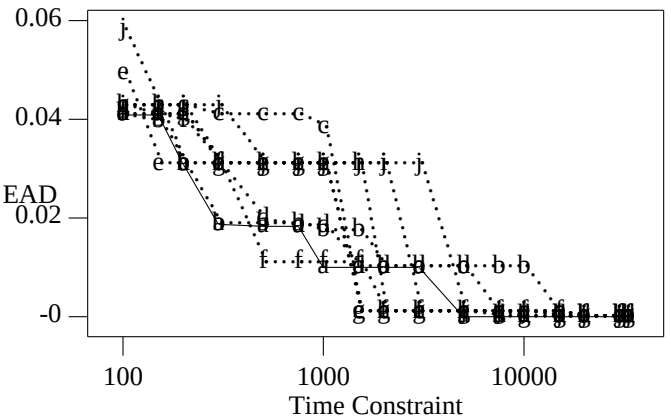
(a) AADs of various RTS[I,LG] on KS(80,1).



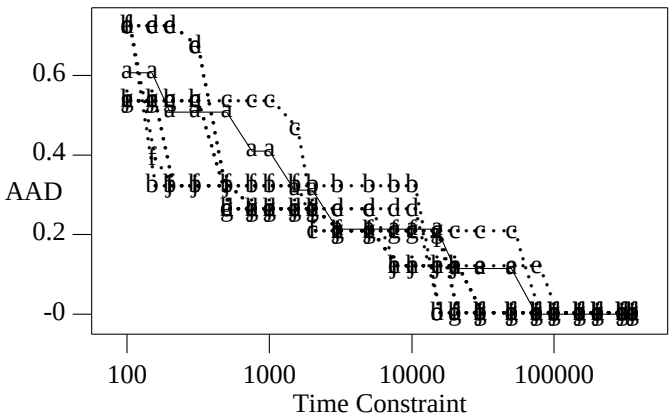
(b) EADs of various RTS[I,LG] on KS(80,1).



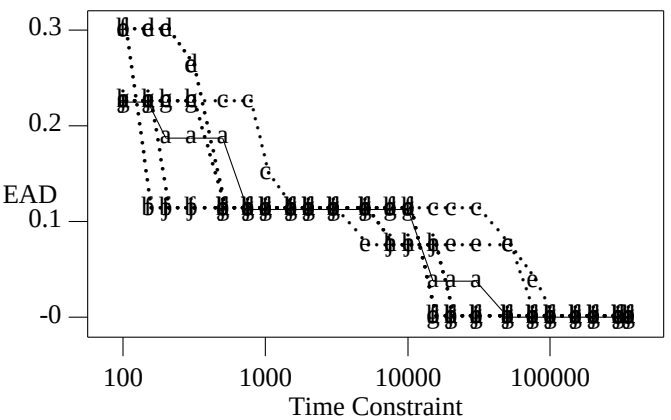
(c) AADs of various RTS[I,LG] on PP(18,2).



(d) EADs of various RTS[I,LG] on PP(18,2).



(e) AADs of various RTS[I,LG] on VC(40,4).



(f) EADs of various RTS[I,LG] on VC(40,4).

Figure 3.16. Performance of the RTS[I,LG] in solving KS(0.5;80,1), PP(3;18,2), and VC(0.1;40,4) where ‘a’ is for the g used for comparison, ‘b’ for g=0.25, ‘c’ for g=0.2, ‘d’ for g=0.175, ‘e’ for g=0.15, ‘f’ for g=0.125, ‘g’ for g=0.1, ‘h’ for g=0.075, ‘i’ for g=0.05, ‘j’ for g=0.025.

Our results are verified by simulations using the KS, PP, TSP, and VC problems.

Finally, we have the following concluding remarks:

(a) For applications with discrete solution values, RTSs with thresholding generally perform well.

(b) For application problems with significant changes of incumbents across iterations and with effective guidance strategies, either RTS[LG,I] or RTS[FR,I] is effective.

(c) For applications with minor changes of incumbents across iterations and with effective guidance strategies, RTSs using either approximation or thresholding perform well.

(d) For application problems with ineffective guidance strategies, RTSs with thresholding generally perform well.

CHAPTER 4.

BAND SEARCH

Simple search methods frequently used include best-first search (BFS) and guided depth-first search (GDFS). BFS expands the least number of nodes in solving an optimal solution (provided that there is no semiessential node - nodes whose lower bounds are equal to the optimal solution value), but requires a large amount of memory that grows exponentially with the problem size or the depth of the search tree. As a result, BFS is practical only in solving small search problems. On the other hand, GDFS requires memory proportional to the product of the depth of the search tree and the branching degree of the search tree; however, it may become trapped deep in the search tree.

More efficient search algorithms have been designed using GDFS as a building block. These include IDA* [37], dynamic-control IDA* (MIDA*) [103], IDA_CR [84], DFS* [92], and iterative refining A* (IRA*) [55]. All these algorithms are iterative and search progressively deeper in the search tree using multiple GDFSs. They have the advantages that they use limited memory and that the number of nodes expanded is close to that of BFS.

Instead of searching deeper in successive iterations, *iterative broadening* [20] increases the number of branches searched in each node progressively until the entire tree is searched. This algorithm is suitable for searching trees with large and uniform branching degrees and for solving decision problems. However, it is not suitable for optimization problems considered in this thesis, whose search trees have small branching degrees.

In this chapter, we study a novel search algorithm called *band search*, which generalizes GDFS and BFS and has the best features of both searches. A band search

allocates a limited amount of space (called the *band*) in each level for storing active nodes in that level, selects for expansion in a *best-first* fashion from nodes in *all* the bands in memory, and backtracks in a depth-first fashion when all these bands become empty. The advantage of the algorithm is that the space it uses is proportional to the product of the depth of the search tree, the branching degree of the search tree, and the maximum width of the bands. The number of nodes expanded by band search is close to that of BFS, and its memory requirement is only $w_{\max}$ times the memory used in GDFS, where $w_{\max}$ is the maximum band width. The latter feature makes band search practical to use in most real-world applications that do not have deep search trees.

4.1. Generic Band Search

Band search is characterized by a band of at most w_i active nodes in each level, where the nodes inside these bands are selected for expansion in a *best-first* fashion. When all the bands are empty, the bands are updated by *depth-first* backtracking. The generic band search is one whose band widths are defined a priori.

The generic band search can be explained conceptually as follows. The active list is decomposed into priority queues, one for each level of the search tree, as is shown in Figure 4.1. A *band* is the top portion of the priority queue. The priority queue in level i is associated with a *band width* w_i and a *band counter* c_i. The band in a level is defined as the top portion of the priority queue in this level, where the top portion is determined by the band counter. The band counter delimits the set of legitimate nodes to be selected for expansion in this level. To simplify the discussion, we assume that all the priority queues are oriented to the left, that is, the best node occurs at the leftmost end of a priority queue.

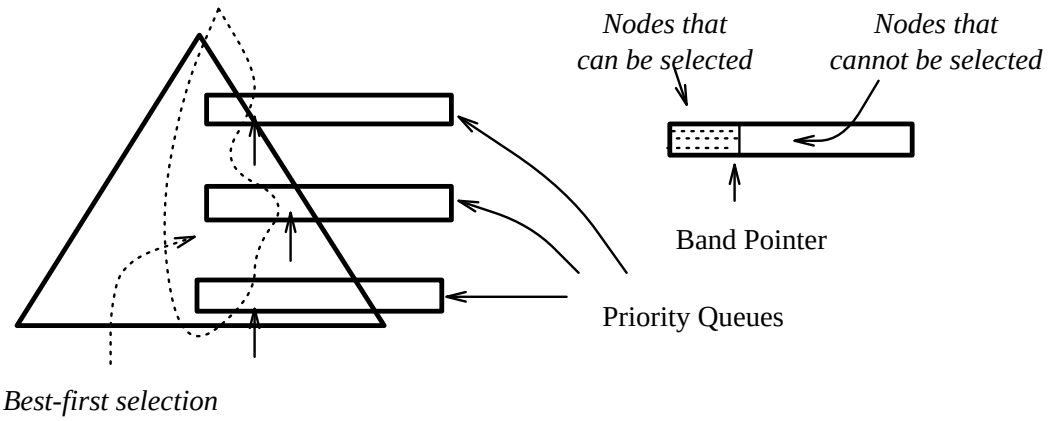


Figure 4.1. Conceptual illustration of band-search operations.

Initially, the band pointer is set to the position defined by its band width, that is, $c_i = w_i$ for every i. During the search, the band pointer can move to the left only, that is, c_i can only be decreased. When a search node in level i is expanded, the band pointer in this level moves by one to the left, that is, $c_i \leftarrow c_i - 1$. Eventually, all the bands become empty. This happens either when all the band pointers have moved to their leftmost ends, that is, $c_i = \text{nil}$ for every i, or when all the nodes in the bands have been expanded. When all the bands are empty, the search finds the deepest level, say level x, where the priority queue is not empty, and resets the band pointer of level x to the position defined by the band width in level x, that is, $c_x = w_x$. The search is then resumed to select new search nodes for expansion. The search stops when all the priority queues are empty.

A band search has three major features: (1) it selects for expansion in a best-first fashion from all the nodes in the bands, (2) it resets band counters in a depth-first fashion, and (3) it restricts backtracking so that at most w_i nodes in level i are fully searched before allowing new nodes in this level into the band. In terms of its operations, band search has five phases.

(1) *Selection Phase.* It selects for expansion in a best-first fashion from all the nodes in the bands and deletes the node selected from its corresponding band. Assume that this node is in level $d-1$. The band counter c_{d-1} is decremented by one when a node in level $d-1$ is removed from its band for expansion.

(2) *Expansion Phase.* It expands the node selected into child nodes in level d. The band counters are not changed in this phase.

(3) *Pruning Phase.* A child node from the node expanded, if any, is pruned when its heuristic value is worse than the incumbent. If the descendent is a feasible solution and its value is better than the incumbent, then the incumbent is updated, and nodes in the priority queues whose heuristic values are worse than the new incumbent are pruned. The band counters are not changed in this phase.

(4) *Insertion Phase.* All the expanded child nodes that are not pruned are inserted into the priority queue in level d. The band counters are not changed in this phase.

(5) *Backtracking Phase.* If the bands are not empty, then it goes to step (1). If all the bands are empty, then it finds the deepest nonempty priority queue (say, in level d') and resets the band counter in level d' to $w_{d'}$. Note that only the band counter (and nodes in this band) in level d' is changed, although there may be nonempty priority queues in levels above d'. Similar to backtracking in GDFS, band search only backtracks to the deepest level with a nonempty priority queue. This step is necessary for bounding the number of nodes in the priority queues. If all the priority queues are empty, then the algorithm stops.

To illustrate the operations in band search, we show a step-by-step example of using band search with uniform band widths equal to 2. Consider a simple search tree shown in Figure 4.2, where the feasible-solution nodes are indicated by double circles. Numbers inside the circles represent both the node identification numbers and the corresponding lower-bound or feasible-solution values. The optimal-solution value is 11. The twelve steps are shown in Figures 4.3-4.6.

In this example, PQ[*i*] denotes the priority queue in level *i*, where a semicolon delimits the scope of the band such that numbers to the left of the semicolon are the identification numbers of nodes in the band. Also, c[*i*] denotes the band counter in level *i*. In each step, we show two boundaries, although sometimes they are the same. The upper boundary delimits the part contained in the bands, and the lower one delimits the part of the search tree contained in the priority queues. Therefore, the part of the search tree in between the two boundaries contains search nodes that are in the priority queues but outside the bands.

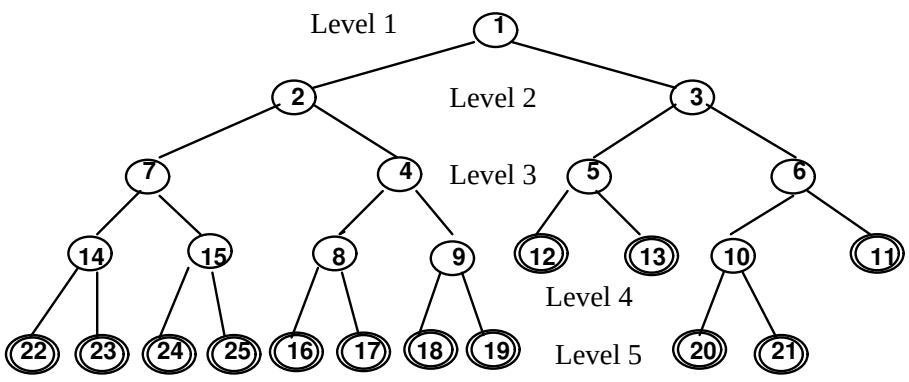


Figure 4.2. A search tree for illustrating the operations of band search.

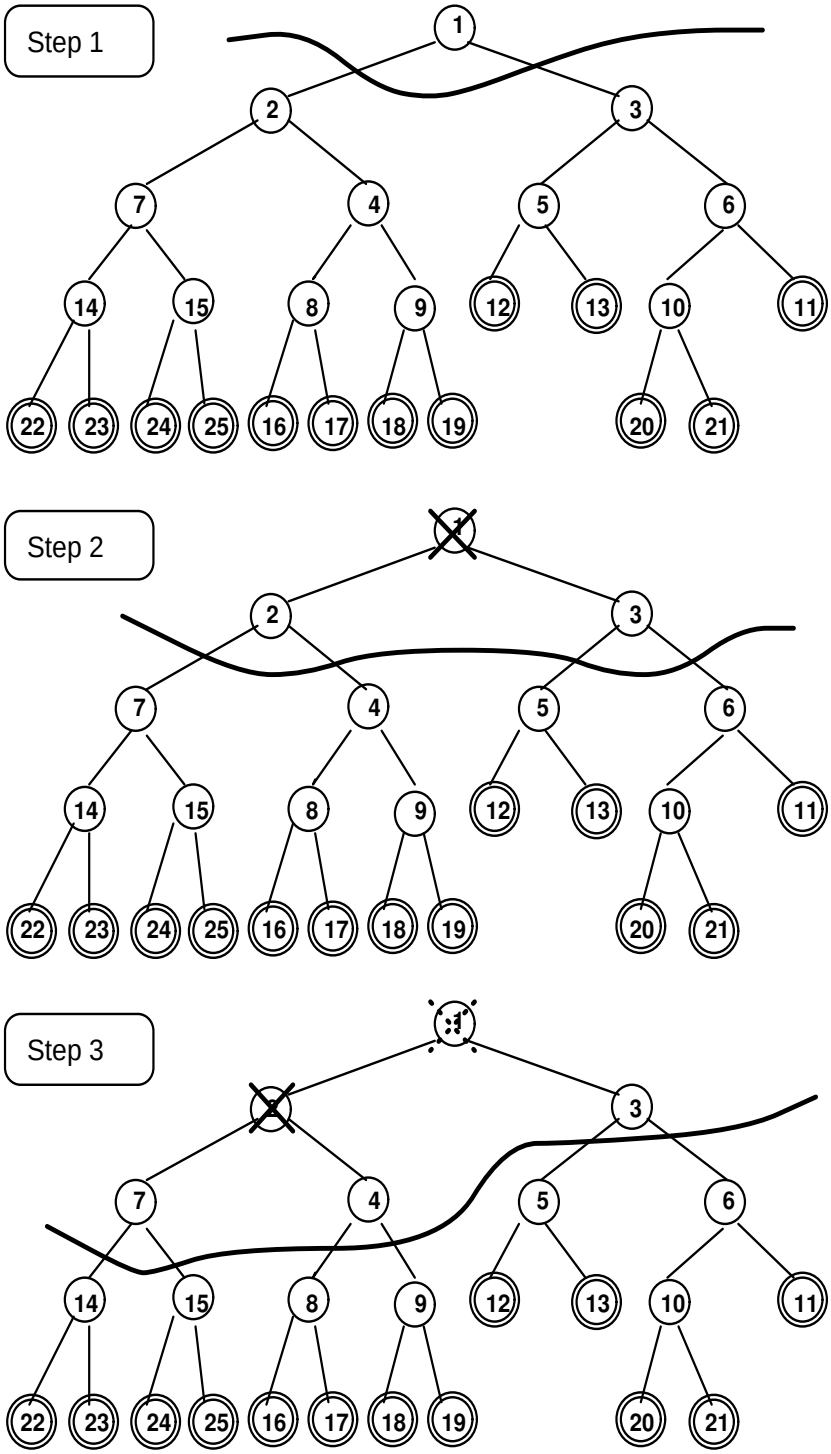


Figure 4.3. Step-by-step illustration of band search (Steps 1-3).

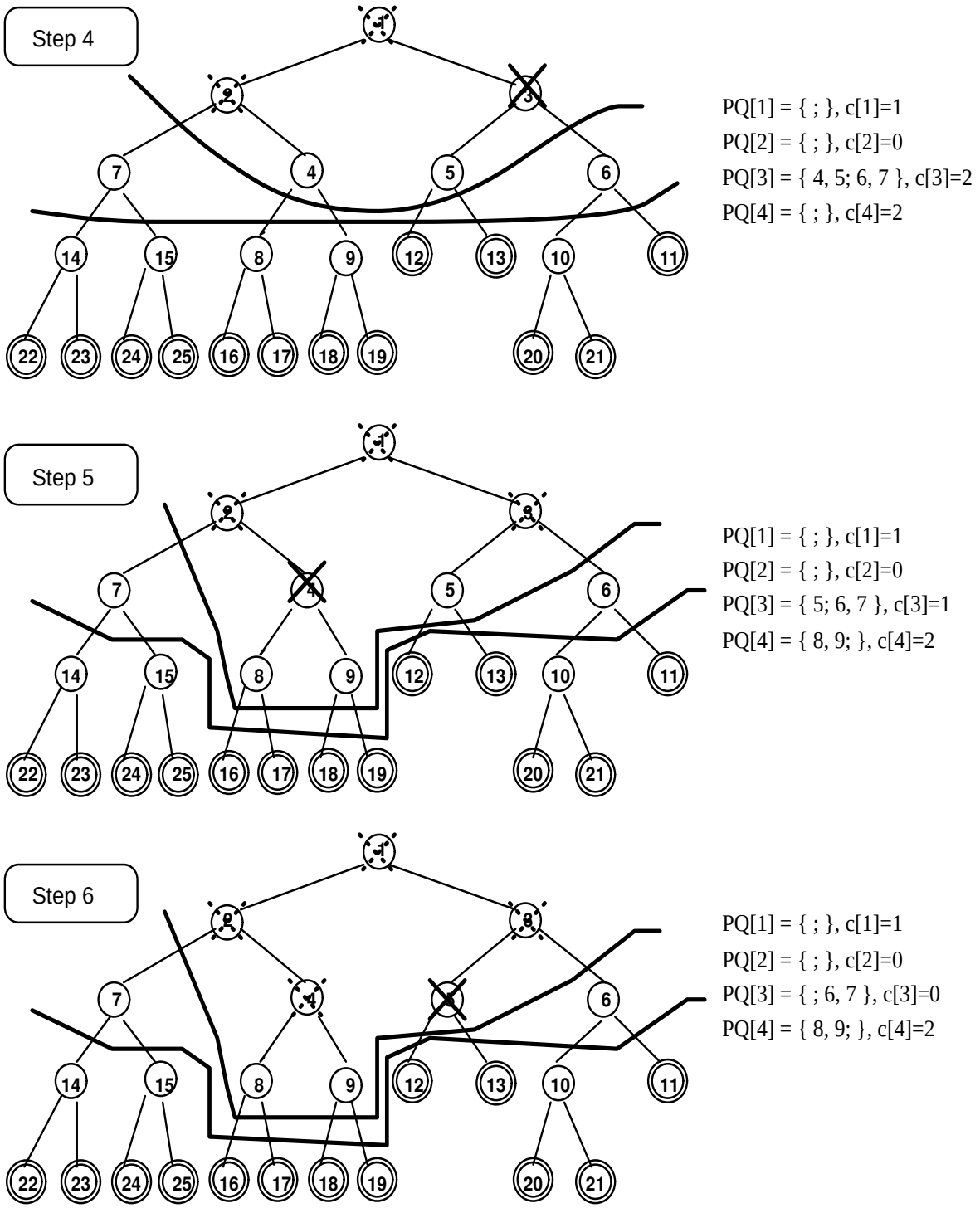


Figure 4.4. Step-by-step illustration of band search (Steps 4-6).

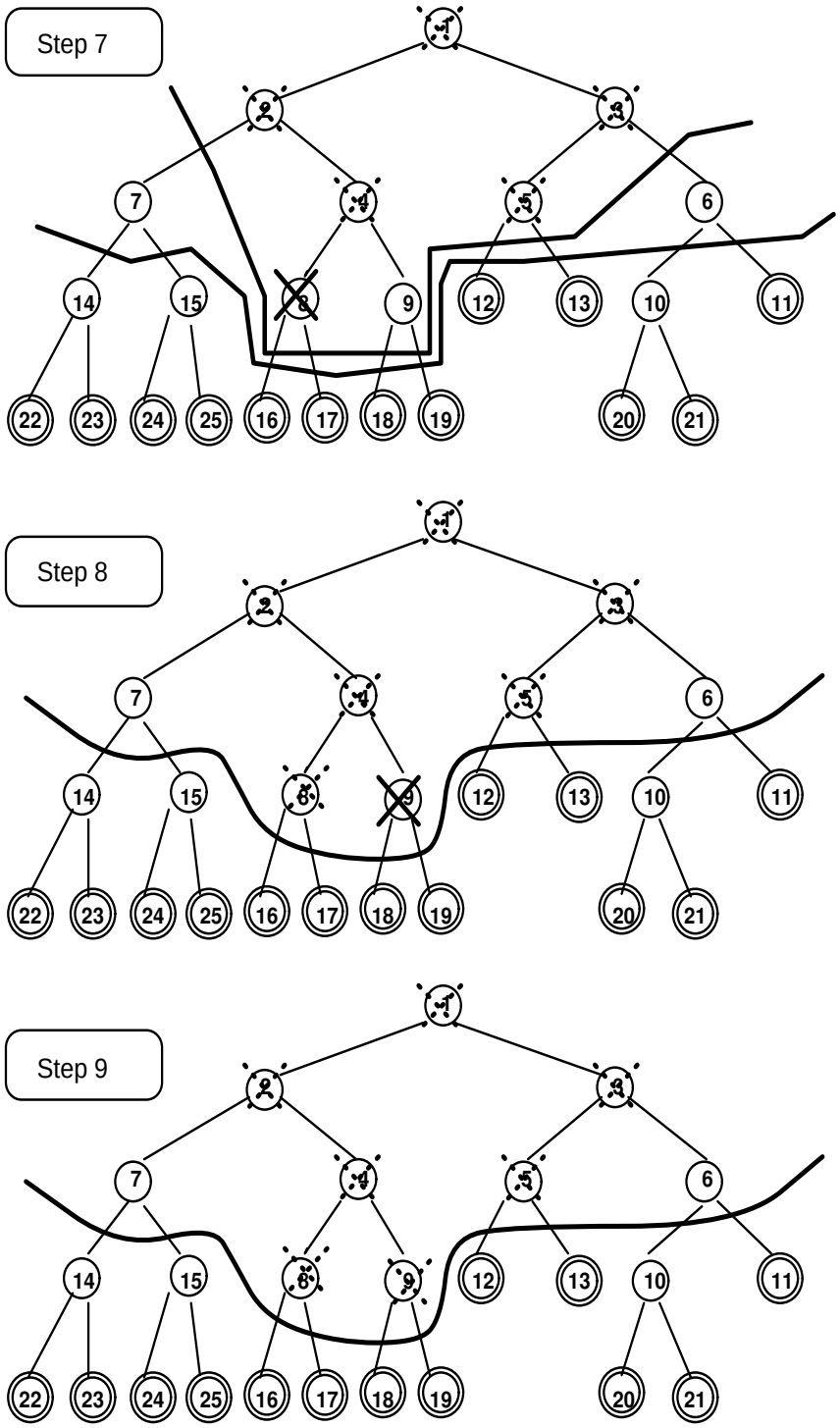


Figure 4.5. Step-by-step illustration of band search (Steps 7-9).

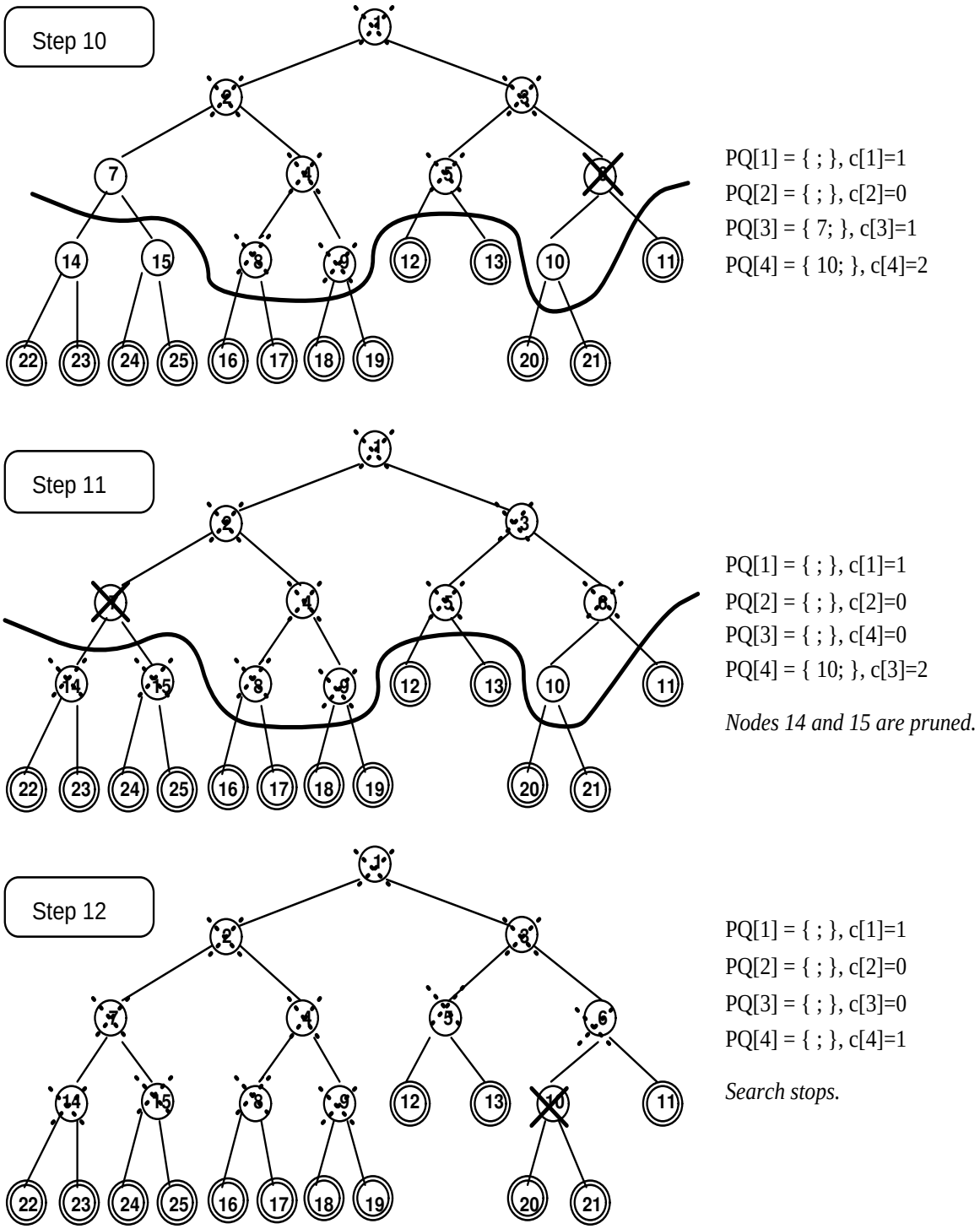


Figure 4.6. Step-by-step illustration of band search (Steps 10-12).

The expansion sequence due to this band search with widths equal to 2 is

$$\text{BS}[2] \Rightarrow 1, 2, 3, 4, 5, 8, 9, 6, 7, 10.$$

For comparison, the expansion sequences due to BFS and GDFS are, respectively,

$$\text{BFS} \Rightarrow 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,$$
$$\text{GDFS} \Rightarrow 1, 2, 4, 8, 9, 7, 14, 15, 3, 5, 6, 10.$$

In this example, band search and BFS expand different sequences of nodes, but they are both optimal because they expand only essential nodes (nodes whose lower bounds are less than the optimal solution). Further, we observe that band search performs better than GDFS.

Band search requires a search tree to have finite depth, as in the case of GDFS; otherwise, the search may become trapped deep in the search tree. However, in general, the depth of the search tree may not be known ahead of time; therefore, space for the priority queue may need to be allocated dynamically.

Band search resembles beam search [108] in the sense that both focus on a band (or beam) of nodes in each level. One major difference, however, is that nodes in a beam search are searched in a breadth-first fashion, whereas nodes in bands are indexed by their heuristic values and searched in a best-first fashion. Another major difference is that a beam search does not allow backtracking, whereas band search does.

A band search specializes to a GDFS when all the band widths are equal to one, and to a BFS when all the band widths are unlimited. By choosing band widths appropriately, we find empirically that band searches can often outperform GDFSs and have performances close to that of BFSs. In most cases, we find that band searches with larger band widths expand less nodes than band searches with smaller band widths; that is, band searches with larger band widths perform closer to BFSs than those with smaller band widths.

Band search can be considered as a B&B search with a binary selection strategy that uses two selection keys, the first indicating whether the search node is inside or outside the band, and the other, the heuristic values of the search nodes. Recall that we only consider minimization searches. For nodes inside the bands, the first selection key is 0, while the key is 1 for nodes outside the bands. Nodes inside the band are further ordered by the secondary key (its heuristic value). In this way, nodes inside the bands are selected for expansion in a best-first fashion. The first selection key of search nodes is updated dynamically based on their positions in the priority queues and the band counters.

There are several possible implementation techniques that can improve the efficiency of band searches in terms of physical execution time.

(1) To avoid the overheads in determining whether a priority queue is empty or not and in selecting the best node across all priority queues, we can maintain two running bounds that delimit the range of active levels in the search tree.

(2) The number of active priority queues used in a band search may not be known ahead of time, hence complicating static allocation of memory space. One way to cope with the problem is to double the size of the memory space whenever the space allocated to the array of active priority queues is filled up.

4.2. Variants of Band Search

Band searches can be static or dynamic. A band search is *static* if its band widths do not change at run time. In contrast, a band search is *dynamic* if its band widths change with time. Band searches can be used as a search algorithm by itself or as a primitive search in an iterative search algorithm such as IDA* and DFS*. Static band searches are very effective

for solving some classes of COPs with unlimited time, whereas dynamic band searches are effective for solving some classes of COPs under time constraints.

In this section, we describe three groups of band searches. The first group is on static band searches, the second, dynamic band searches, and the third, iterative band searches.

4.2.1. Static band search

Band widths in a static band search are static and predefined. The predefined widths can be *uniform* (i.e., all widths are equal) or *functional* (i.e., widths are functions of level numbers in the search tree).

Let $d_{\max}$ be the maximum depth of a search tree. In the case of uniform band widths, band search has $w_i = \omega$ for every $i = 1, ..., d_{\max}$, where ω is predefined.

Band widths can be static and be an increasing or decreasing function with levels in a linear, exponential, or arbitrary fashion. For increasing band widths, we wish to have $w_1 = 1$ and $w_{d_{\max}} = \omega$, where ω is predefined. In contrast, for decreasing band widths, we wish to have $w_1 = \omega$ and $w_{d_{\max}} = 1$. Possible linear and exponential functions that achieve these boundary conditions are

$$w_i = del\left\lceil 1 + (\omega - 1)\left(\frac{i-1}{n-1}\right)\right\rceil \qquad \textit{Linearly Increasing Band Widths}$$

$$w_i = del\left\lceil \omega - (\omega - 1)\left(\frac{i-1}{n-1}\right)\right\rceil \qquad \textit{Linearly Decreasing Band Widths}$$

$$w_i = del\left\lceil \omega^{\frac{i-1}{n-1}}\right\rceil \qquad \textit{Exponentially Increasing Band Widths}$$

$$w_i = del\left\lceil \omega^{1-\frac{i-1}{n-1}}\right\rceil \qquad \textit{Exponentially Decreasing Band Widths}$$

4.2.2. Dynamic band search

Dynamic band searches adapt band widths dynamically based on the effectiveness of the search heuristics used. They are designed especially for problems with sparse solutions. For problems with dense solutions, static band searches perform very well.

The key issues in a dynamic band search include the measurement of the effectiveness of search heuristics and the dynamic adaptation of band widths.

For problems with sparse solutions, heuristics are designed in the search algorithm to find good feasible solutions quickly. One way to measure their effectiveness is to use the frequency of feasible or infeasible solutions encountered. However, such measures are inadequate because the quality of feasible solutions found may have large variations. Consequently, we do not take the quality of feasible solutions as a performance indicator in dynamic band searches.

A dynamic band search adjusts its band widths as soon as the estimated effectiveness of the search strategy changes. The amount of change is difficult to decide. One possible way is to change band widths by a small increment and localize the adjustment of band widths in the levels involved. To avoid increasing band widths without limit, we limit the size of all band widths, hence limiting the memory space used.

For fairly good guidance heuristics, GDFSs usually perform well in finding the first solution, while band searches are good for improving feasible solutions found. For these problems, we can start the search by a GDFS (band search with band widths equal to 1) and switch to a band search when a feasible solution is found.

In summary, a dynamic band search works as follows. Initially, it has band widths equal to one for all levels. After the first feasible solution is found, band widths are allowed to change. During the search, band widths are adjusted when either feasible or infeasible solutions are found.

(1) *Increasing Band Widths.* If all children of a nonterminal node in level d are infeasible, then we interpret that the search heuristics used have not performed well in the band widths allowed. To overcome this problem, we increase the band widths by one for all active levels from 1 to $d-1$. If band widths are larger than the predefined limit, then we reset it to the limit.

(2) *Decreasing band widths.* If a feasible solution is found when expanding a nonterminal node, then we interpret that the search heuristics used have performed well, and we may reduce the band widths without degrading its performance. In this case, we decrease band widths by one for all active levels from 1 to $d-1$, except in the level where the band width is already one.

4.2.3. Iterative band search

Band search can serve as a primitive search in iterative searches such as IDA* and DFS* instead of using GDFSs as primitive searches. The resulting searches are denoted as IDA*_BS and DFS*_BS, respectively.

Band search can also be designed so that backtracking is not allowed. In this way, it only expands search nodes in the bands and discards all search nodes outside the bands. The resulting band search without backtracking is called *first-band search*. First-band search is

similar to beam search, but they differ in that the former exercises best-first selection and the latter, breadth-first selection.

An *iterative first-band search* is an iterative search using the first-band search as a primitive search. The iterative first-band search increases band widths exponentially with the number of iterations. This search is only suitable for time-constrained search in hoping that it can arrive at a good solution quickly.

4.3. Properties of Band Search

In this section, we study the properties of band search and its memory requirement. The following theorem shows the limiting cases of band search.

Fact 4.1. *A band search with all band widths equal to one is equivalent to a GDFS.*

Proof. When all band widths are equal to one, only one node is allowed in the band in every level. When a search node is expanded, only one of its child nodes is put in the band and the others are put in the priority queue (outside the band). Therefore, only the node in the band will be expanded. This is equivalent to GDFS. At the instant when all bands are empty, the deepest nonempty priority queue resets its band counter to 1; therefore, a node in that priority queue can be selected for expansion. This is equivalent to backtracking in GDFS. □

Fact 4.2. *A band search with unlimited band widths is equivalent to a BFS.*

Proof. When all band widths are unlimited, all the search nodes generated can be kept in the bands, and all of them can be selected for expansion. Since nodes in the bands are expanded in a best-first fashion, the selection of active nodes for expansion is, therefore, best first. □

For some cases, band search behaves like a BFS even when band widths are limited. There exists a critical threshold for a given search instance so that a band search with band widths larger than this threshold expands the same set of nodes as BFS (assuming no node has a lower bound value equal to the optimal solution value). The sequence of nodes expanded, however, may be different, as observed in the previous example.

The following theorem shows that the priority queue used in band search requires a bounded amount of memory, assuming a finite band width and a finite branching degree.

Theorem 4.1 (bound on size of priority queues). *Consider a search tree and its maximum depth* $d_{\max}$. *For* $1 \le i < d_{\max}$, *the maximum size of priority queue* Q_{i+1} *in level* $i+1$ *is bounded by the product of its band width* w_i *and branching degree* b_i *in level i. That is,*

$$| Q_{i+1} | \le w_i \times b_i .$$

Proof. For level *i*, at most w_i active nodes can be included in the band, and that no nodes outside the band can be moved to this band until the subtrees rooted at the nodes in this band are searched. Therefore, the maximum number of nodes that can be generated by nodes in the band in level *i* is bounded by $w_i \times b_i$. □

The following theorem says that the maximum space used by band search is bounded by the product of the maximum band width, the maximum depth, and the maximum branching degree.

Theorem 4.2 (bound on memory space requirement). *Consider a search tree and its maximum depth* $d_{\max}$ *and maximum branching degree* $b_{\max}$. *The maximum band width of band search is* $w_{\max}$. *The total number of active nodes in all priority queues is bounded as follows.*

$$\sum_{i=1}^{d_{\max}} | Q_i | \le d_{\max} \times w_{\max} \times b_{\max} .$$

Proof. From the previous theorem, we have

$$\sum_{i=1}^{d_{\max}} | Q_i | \le \sum_{i=1}^{d_{\max}} \left(w_i \times b_i \right) \le d_{\max} \times w_{\max} \times b_{\max} . \qquad \square$$

According to Theorem 4.2, the maximum memory required in a band search is at most $w_{\max}$ times the maximum amount needed in a GDFS. This feature makes band search feasible and useful in real-world applications.

In the following, we discuss the optimality conditions for band search. Define the *first bands* as the set of nodes that are included in the bands before backtracking happens, that is, before the bands become empty for the first time. The following theorem shows a necessary and sufficient condition of the optimality of band search. The condition is in terms of occurrences of nodes with lower bounds less than the optimal solution value.

Theorem 4.3 (necessary and sufficient optimality condition). *A band search is optimal if and only if any of the bands always contains at least one essential node at any moment of node selection, except when all priority queues are empty.*

Proof. (Sufficiency proof) Whenever node selection is done, if any of the bands always contain at least one essential node, then one of the essential nodes must be selected for expansion. When all priority queues are empty, then the search stops and nothing will be selected. Therefore, the nodes selected for expansion must always be essential.

(Necessity proof) On the other hand, when band search is optimal, it only expands essential nodes. Therefore, there must exist at least one essential node that can be selected for expansion at any time. Since only nodes in the bands are eligible to be selected, the essential

nodes selected must exist in the bands; namely, any of the bands must always contain at least one essential node at any moment of node selection. All these can only happen when not all the priority queues are empty; otherwise, the search stops. □

Note that these two conditions do not require the uniqueness of lower bound values of search nodes, which is generally a sufficient condition for BFSs to be optimal. In a BFS, anomalies in performance may happen when there are semiessential nodes (nonsolution nodes with lower bounds equal to the optimal solution value) on the path from the root node to any optimal or infeasible solution. *In this thesis, when we say that band search is optimal with respect to BFS, we assume (1) that no semiessential nodes exist on the path from the root to any optimal or infeasible solution in the search tree, and (2) that dominance is not active.* The first assumption is necessary so that the number of nodes searched by a BFS is the minimum possible. If this assumption does not hold, then it is possible for a band search to find a solution faster than a BFS, as we have observed in our simulations. The second assumption is also necessary since nodes pruned by dominance relations may depend on the order in which nodes are searched.

4.4. Band-Width Anomalies

Band searches with larger band widths in general expand less nodes than those with smaller band widths. However, we have observed anomalies in a limited number of cases. We refer to these as *band-width anomalies*. A special case of band-width anomalies is that a band search may perform worse than a GDFS (which is a band search with band widths equal to one).

Band-width anomalies can be illustrated by the following example. Consider the search tree shown in Figure 4.7, and two band searches, one (denoted as BS[2]) with static uniform band widths equal to 2 and the other (denoted as BS[3]) with static uniform band widths equal to 3. Applying BS[2] and BS[3] results in the following expansion sequences.

$BS[2] \Rightarrow$ 1, 2, 3, 8, 9, 10, 4, 5, 5, 6, 7, stop.

$BS[3] \Rightarrow$ 1, 2, 3, 4, 5, 6, 7, 11, 11, 12, ..., are trapped in a large subtree.

BS[3] is trapped in a large subtree; consequently, it expands more nodes than BS[2]. It is easy to extend this example to show the general case in which BS[w_1] can expand more nodes than BS[w_2] for $w_1 > w_2$.

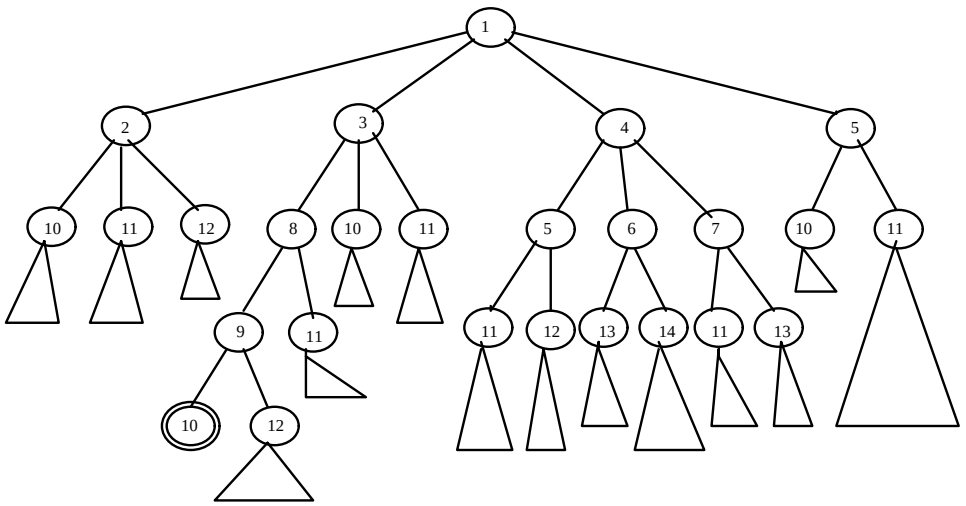


Figure 4.7. A search tree with band-width anomalies.

Note that band-width anomalies can happen when all lower bounds are unique as in Figure 4.7 or when lower bounds are not unique. The reason for these anomalies is that band searches with larger band widths have more nodes of similar quality, but not the same set of nodes as in BFS, to select for expansion. This increases the risk of selecting a node with a

small lower-bound value but leading the search to a large subtree with no solution or inferior solutions.

Fortunately, as empirically observed, anomalies do not happen frequently, especially when band widths are large and the lower bounds are generally ‘‘consistent’’ with the quality of the best solutions found in the subtree rooted at these nodes. We say that lower bounds are consistent between two nodes (say nodes *i* and *j*) in the search tree when the lower bound of node *i* is smaller than that of node *j*, and the best solution obtained by searching the subtree rooted at node *i* is also smaller than that at node *j*. Anomalies may happen in problems that have abundant inconsistent lower bounds, such as in puzzle and maze problems.

Theorem 4.4 (necessary condition for band-width anomalies). *If band-width anomalies happen, then for a band search with larger band widths, at some particular time, the bands do not contain any essential nodes, and there remain essential nodes in some of the priority queues.*

Proof. This can be proved by contradiction. If any of the bands contains at least one essential node all the time, then band search is optimal. In this case, band searches with larger band widths are optimal, and band searches with less band widths cannot be better. Consequently, band-width anomalies cannot happen. A contradiction! □

A band search may perform worse than a GDFS due to band-width anomalies. Further, we cannot guarantee the worst-case performance of band search with respect to the performance of GDFS nor BFS. This is not desirable in some cases when the worst-case performance is important. In the following, we propose a method to cope with the worst-case behavior of band search.

Consider a hybrid method that expends a fraction a of the total time in solving an application problem by a GDFS and $(1-a)$ fraction of the total time for a band search, where $0 \le a \le 1$. The execution time t_h of the hybrid method can be computed based on a, the execution time t_g of GDFS, and the execution time t_b of the band search.

$$t_h = \min\left\{ \frac{t_g}{a}, \frac{t_b}{1-a} \right\} \le \frac{t_g}{a} .$$

The above equation shows that the performance of the hybrid method can be bounded in terms of the execution time of GDFS.

The fraction value a is between 0 and 1 and is chosen based on the relative magnitudes between t_g and t_b. The hybrid search outperforms GDFS if t_h is less than t_g, which happens only when t_h is equal to $t_b / (1-a)$. Therefore, we have

$$t_g > t_h = \frac{t_b}{1-a} .$$

Rearranging the above equation leads to

$$a < 1 - \frac{t_b}{t_g} .$$

The above equation says that the hybrid search outperforms GDFS only when the band search is faster than GDFS by a factor $(1-a)$. For example, if $t_b = t_g / 2$, then $a < 0.5$. This means that we should spend more time on band search when it is twice as fast as GDFS in order for the hybrid search to be faster than GDFS. Although the hybrid search may not have better average performance than GDFS, it limits the worst-case performance of the latter. The value of a can be adjusted based on the importance placed on the worst-case performance.

4.5. Experimental Results

In this section, we evaluate empirically the performance of band searches. We consider two cases: (1) unlimited time and (2) limited time. In the first case, we evaluate static band searches as well as iterative band searches. In the second case, we evaluate dynamic band searches and iterative first-band searches.

For easy reference, the symbols denoting band search are summarized in Table 4.1. The square brackets contain the parameters used in the band searches evaluated. There are two set of parameters: the first set describing the type of band searches, and the second, the band widths.

In the first set of parameters, ‘‘s’’ and ‘‘d’’ denote that the band search is ‘‘static’’ and ‘‘dynamic’’, respectively. Symbols ‘‘u’’ and ‘‘f’’ denote that the band widths are ‘‘uniform’’ and ‘‘functional’’, respectively. Symbols ‘‘li’’ and ‘‘ei’’ denote that the band widths increase linearly and exponentially, respectively, with the levels. Symbols ‘‘ld’’ and ‘‘ed’’ denote that the band widths decrease linearly and exponentially, respectively, with the levels. Symbol ‘‘ii’’ denotes that the band search is an iterative first-band search. Symbol ‘‘bwi’’ denotes that the dynamic band search adapts band widths locally; that is, it adapts band widths one level at a time.

In the second set of parameters, ‘‘w’’ denotes that uniform band widths are equal to w. ‘‘$w_{\min}$’’ and ‘‘$w_{\max}$’’ denote the minimum and the maximum band widths allowed in the band search.

Table 4.1. Summary of the symbols denoting band searches studied, where parameters are inside square brackets.

Notation	Meaning
BS[s,u;w]	*s*tatic band search with *u*niform band widths, where the uniform band width is equal to *w*
BS[s,li;$w_{\min}$,$w_{\max}$]	*s*tatic band search with *l*inearly *i*ncreasing band widths, where $w_{\min}$ and $w_{\max}$ denote, respectively, the minimum and the maximum band widths allowed
BS[s,ld;$w_{\min}$,$w_{\max}$]	*s*tatic band search with *l*inearly *d*ecreasing band widths, where $w_{\min}$ and $w_{\max}$ denote, respectively, the minimum and the maximum band widths allowed
BS[s,ei;$w_{\min}$,$w_{\max}$]	*s*tatic band search with *e*xponentially *i*ncreasing band widths, where $w_{\min}$ and $w_{\max}$ denote, respectively, the minimum and the maximum band widths allowed
BS[s,ed;$w_{\min}$,$w_{\max}$]	*s*tatic band search with *e*xponentially *d*ecreasing band widths, where $w_{\min}$ and $w_{\max}$ denote, respectively, the minimum and the maximum band widths allowed
IDA*_BS[s,u;w]	IDA* using as its primitive search a *s*tatic band search with *u*niform band widths, where the uniform band width is equal to *w*
DFS*_BS[s,u;*w*]	DFS* using as its primitive search a *s*tatic band search with *u*niform band widths, where the uniform band width is equal to *w*
BS[s,u,ii;r]	*i*terative first-band search with *s*tatic, *u*niform band widths, where *r* is the growth rate
BS[d;w_{init},$w_{\max}$]	*d*ynamic band search, where w_{init} and $w_{\max}$ denote, respectively, the initial and the maximum band widths
BS[d,bwi;w_{init},$w_{\max}$]	*d*ynamic band search with *i*solated *b*and*w*idth adaptation where w_{init} and $w_{\max}$ denote, respectively, the initial and the maximum band widths

4.5.1. Band search without time constraint

Static band searches to be evaluated include the ones with uniform band widths and the ones with functional band widths. We empirically evaluate static band search as on numerous search instances for a variety of band widths, and identify the best choice of band width in terms of physical execution time of band search. We evaluate four functional band widths: (1) band widths increasing linearly with levels, (2) band widths decreasing linearly with levels, (3) band widths increasing exponentially with levels, and (4) band widths decreasing exponentially with levels.

The iterative band searches to be evaluated include IDA* and DFS* using band search as their primitive search. Note that IDA* and DFS* traditionally use GDFS as their primitive search.

The test problems for evaluating static band searches include the TSP(1), PP(3), KS(0.5), and VC(0.1) problems. These test problems have been described in Chapter 3 and are not repeated here. The test problems for evaluating iterative band searches include maze and fifteen-puzzle problems.

A maze problem entails the search of the shortest path that connects an entrance to an exit in a maze. The mazes we use are generated by a modified version of the X maze program [26], whose original version generates mazes that only have a single feasible solution. To increase the complexity of finding solutions, we swap the entrance and the exit; that is, the original entrance becomes the new exit and the original exit becomes the new entrance. Moreover, we remove all the walls next to the new entrance. To vary the complexity, we randomly remove interior walls by generating a random number in between

0 and 100 to see whether it is less than a predefined threshold *v*. We denote Maze(*v*;*n*,*s*) as an instance of the *n*-by-*n* maze problem with threshold *v* and random seed *s*. The lower bound of a search node is calculated as its Manhattan distance between the current position and the exit. The upper bound is calculated by a steepest-descent hill-climbing method if it succeeds. If this method does not succeed, the upper bound is defined as infinity.

A fifteen-puzzle problem entails the search of the shortest sequence of steps that transform the initial panel to the goal panel. The panel in the fifteen-puzzle problem is a 4-by-4 square with 16 tile positions, where each position is either empty or has a tile that is uniquely numbered from 1 to 15, and there is only one empty position. Only the tiles neighboring to the empty position can move to the empty position. Further, only one tile can move at a time, and the position it originally occupies becomes empty. The problem instances used are those published by Korf [37]. The lower bound of the search node is calculated using the Manhattan-distance method [37]. Algorithms for computing the upper bound have not been discovered.

Execution time of search algorithms may vary over several orders of magnitude; therefore, we use normalization on a per-instance basis to alleviate the variance. We can normalize execution time of search algorithm A with respect to that of the optimal algorithm OPTD. Consider a search instance *i*. Let $t_{A,i}$ and $t_{OPTD,i}$ denote the execution time of search algorithms A and OPTD, respectively, in solving instance *i*. The normalized time $t_{A,i}^{[OPTD]}$ is defined as

$$t_{A,i}^{[OPTD]} = del\ \frac{t_{A,i}}{t_{OPTD,i}}. \tag{4.1}$$

The superscript denotes the search algorithm which the normalization of performance is based on. The subscript denotes the search algorithm normalized (with respect to the base algorithm denoted in the superscript) and the search instance.

Similarly, memory usage can also be normalized. Let $m_{A,i}$ and $m_{OPTD,i}$ denote the amount of memory required by search algorithms A and OPTD, respectively, in solving instance i. The normalized memory usage $m_{A,i}^{[OPTD]}$ is defined as

$$m_{A,i}^{[OPTD]} = del \frac{m_{A,i}}{m_{OPTD,i}} .$$

The scatter plots of normalized execution times for static band searches of various uniform band widths in solving fifty instances of the TSP(1;20), PP(3;18), KS(0.5;80), and VC(0.1;40) problems are shown in Figures 4.8-4.11, respectively. In these graphs, the execution time (respectively, memory used) for each instance is normalized. Note that the axis representing memory usage is logarithmic. In all these graphs, the averages of normalized times are connected by a solid line, and the vertical and horizontal bars indicate the estimated 90% confidence intervals on the averages, assuming that the averages have a normal distribution.

From the graphs in Figures 4.8-4.11, we find that static band searches expand less nodes than GDFSs in general in finding optimal solutions, although band-width anomalies may happen. We notice that band searches perform very well on the average for small band widths, and that their performance for large band widths is more stable, showing a monotonic improvement in performance and a smaller chance for band-width anomalies to occur. For various problems evaluated, we have the following observations.

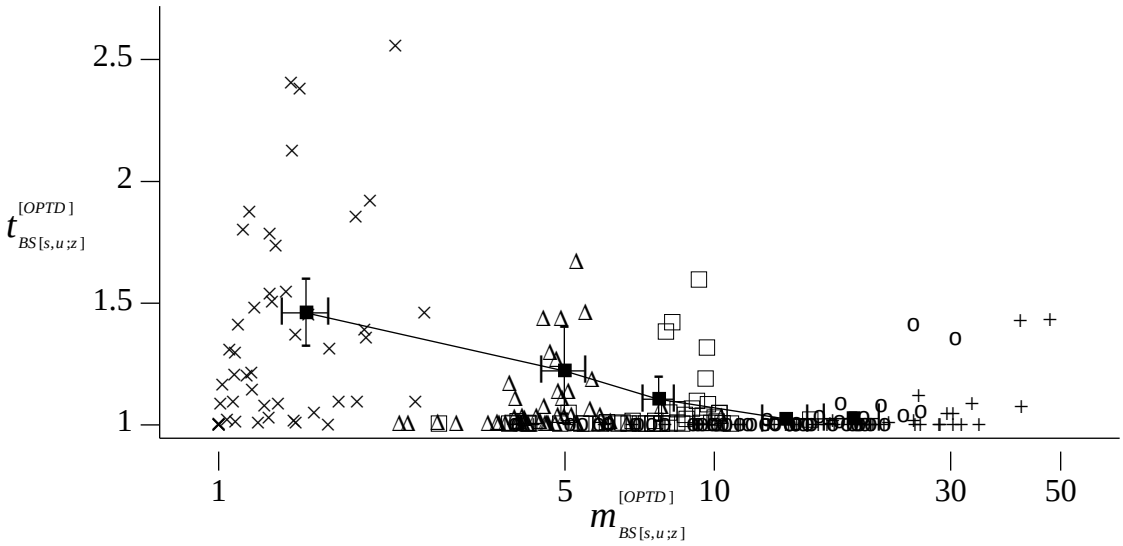


Figure 4.8. Scatter plot of execution time versus memory usage for five static band searches BS[s,u;z], z=1,5,10,30,50, in solving fifty instances of TSP(1;20), where execution time and memory usage are normalized with respect to those of OPTD. (‘‘×’’ for BS[s,u;1] (i.e., GDFS), ‘‘Δ’’ for BS[s,u;5], ‘‘□’’ for BS[s,u;10], ‘‘○’’ for BS[s,u;30]. ‘‘+’’ for BS[s,u;50]).

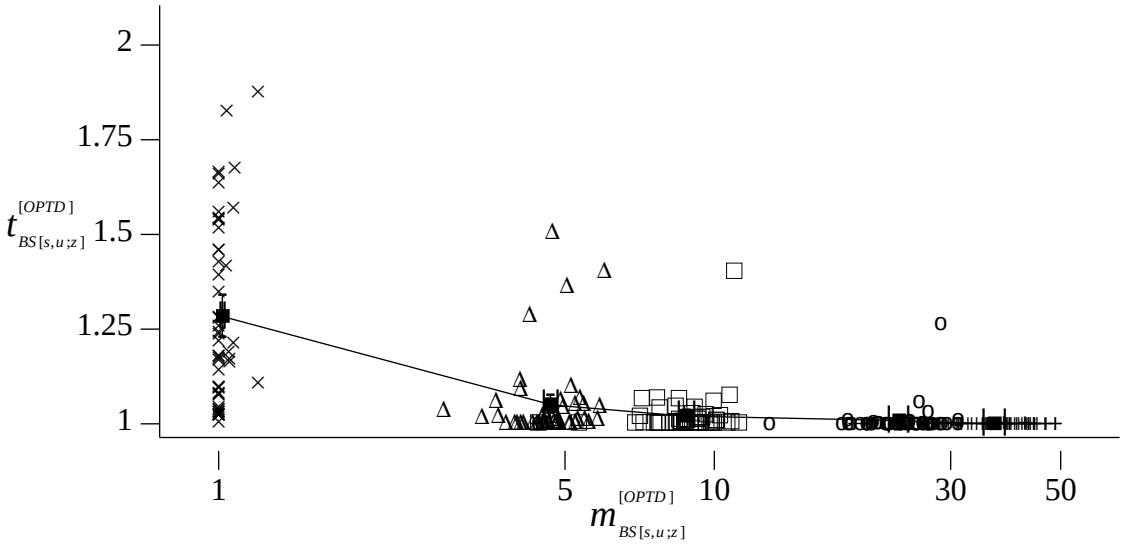


Figure 4.9. Scatter plot of execution time versus memory usage for five static band searches BS[s,u;z], z=1,5,10,30,50, in solving fifty instances of PP(3;18), where execution time and memory usage are normalized with respect to those of OPTD. (‘‘×’’ for BS[s,u;1] (i.e., GDFS), ‘‘Δ’’ for BS[s,u;5], ‘‘□’’ for BS[s,u;10], ‘‘○’’ for BS[s,u;30]. ‘‘+’’ for BS[s,u;50]).

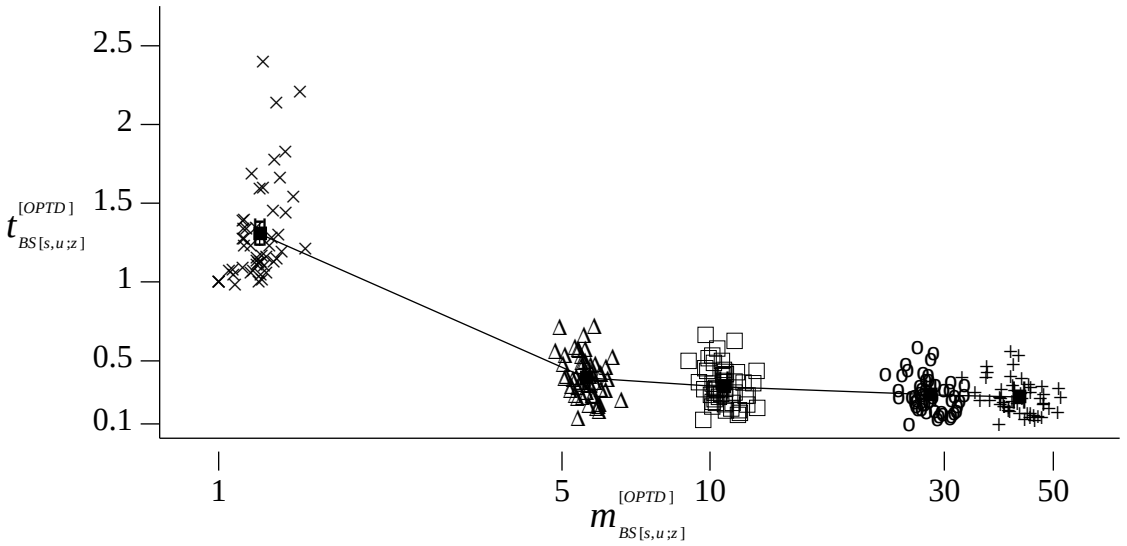


Figure 4.10. Scatter plot of execution time versus memory usage for five static band searches BS[s,u;z], z=1,5,10,30,50, in solving fifty instances of KS(0.5;80), where execution time and memory usage are normalized with respect to those of OPTD. (‘‘×’’ for BS[s,u;1] (i.e., GDFS), ‘‘Δ’’ for BS[s,u;5], ‘‘□’’ for BS[s,u;10], ‘‘○’’ for BS[s,u;30]. ‘‘+’’ for BS[s,u;50]).

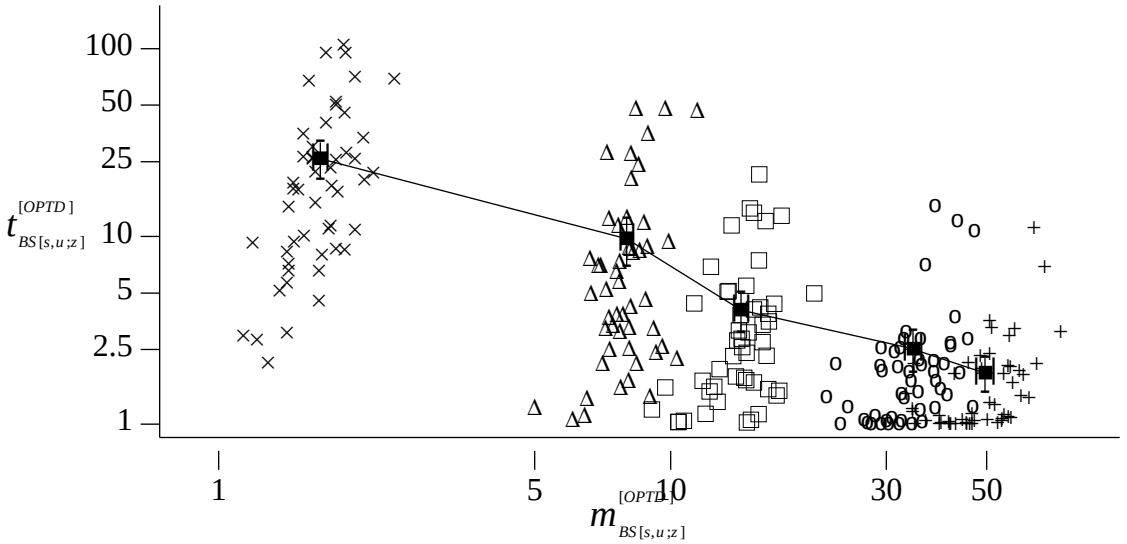


Figure 4.11. Scatter plot of execution time versus memory usage for five static band searches BS[s,u;z], z=1,5,10,30,50, in solving fifty instances of VC(0.1;40), where execution time and memory usage are normalized with respect to those of OPTD. (‘‘×’’ for BS[s,u;1] (i.e., GDFS), ‘‘Δ’’ for BS[s,u;5], ‘‘□’’ for BS[s,u;10], ‘‘○’’ for BS[s,u;30]. ‘‘+’’ for BS[s,u;50]).

(a) For problems where GDFS performs well such as the TSP(1;20) and the PP(3;18) instances, band search performs better and has performance close to that of OPTD. However, its improvement over GDFS is small because GDFS already performs very well for these problems.

(b) For the KS(0.5;80) instances, dominance relations are applied so that search node *i* dominates search node *j* when both nodes have considered the same set of objects, and the total profit (respectively, total weight) of objects assigned to node *i* is larger (respectively, smaller) than that of node *j*. We find that band search performs better than GDFS and even OPTD, because that band search allows more nodes at any level to be considered for selection, allowing dominance relations to be applied more frequently.

(c) For the VC(0.1;40) instances, GDFS does not perform well, and band search improves performance dramatically, while incurring limited overheads in memory usage. The deviations of normalized times are not shown. Note that the time axis in Figure 4.11 is logarithmic.

The scatter plot of normalized execution times of DFS*'s employing static band searches instead of GDFSs in solving fifty random instances of the Maze(0.2;40) problem is shown in Figure 4.12. Also, the scatter plot of normalized execution times of IDA*'s employing static band searches in solving fifty random instances of the fifteen-puzzle problem is shown in Figure 4.13.

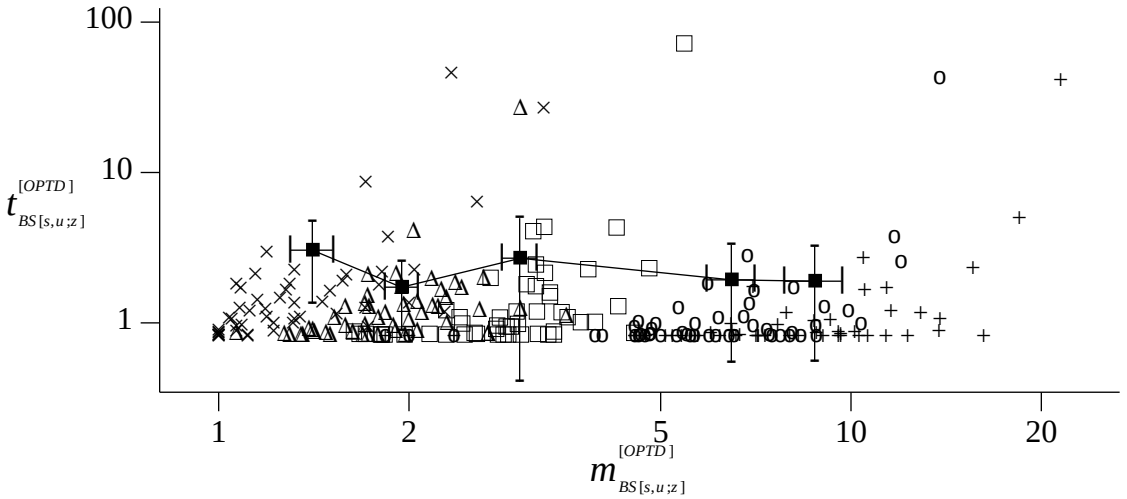


Figure 4.12. Scatter plot of execution time versus memory usage for DFS*_BS[s,u;z], z=1,5,10,30,50, in solving fifty instances of Maze (0.2;40), where execution time and memory usage are normalized with respect to those of OPTD. (‘‘×’’ for DFS*_BS[s,u;1] (i.e., DFS*), ‘‘Δ’’ for DFS*_BS[s,u;5], ‘‘□’’ for DFS*_BS[s,u;10], ‘‘○’’ for DFS*_BS[s,u;30]. ‘‘+’’ for DFS*_BS[s,u;50]).

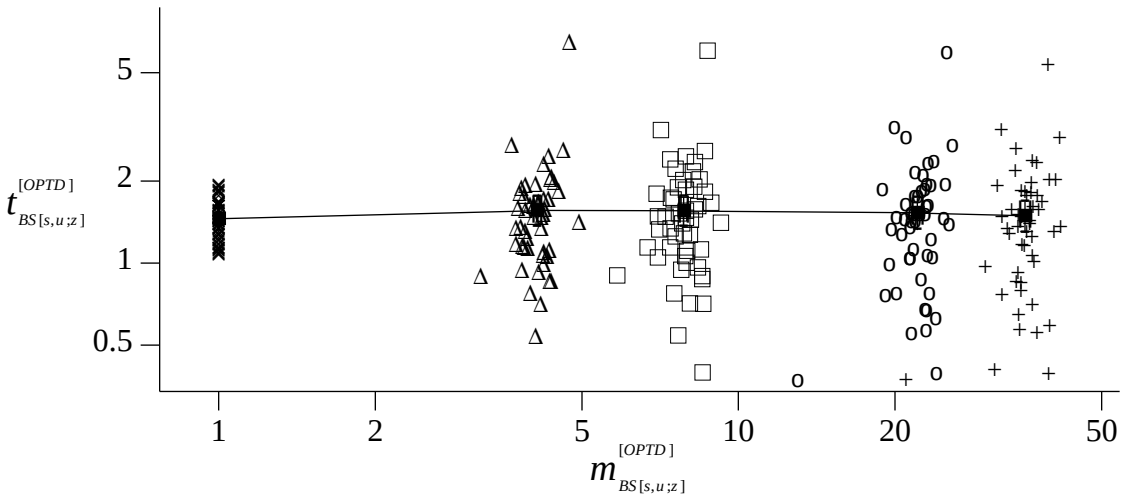


Figure 4.13. Scatter plot of execution time versus memory usage for IDA*_BS[s,u;z], z=1,5,10,30,50, in solving fifty instances of Puzzle (Korf’s;15), where execution time and memory usage are normalized with respect to those of OPTD. (‘‘×’’ for IDA*_BS[s,u;1] (i.e., IDA*), ‘‘Δ’’ for IDA*_BS[s,u;5], ‘‘□’’ for IDA*_BS[s,u;10], ‘‘○’’ for IDA*_BS[s,u;30]. ‘‘+’’ for IDA*_BS[s,u;50]).

From these two graphs, we find that improvements due to band searches are small. This happens because successive searches in IDA* and DFS* employ pruning by thresholds, regardless of whether GDFSs or band searches are used. As a result, the difference in performance between using band searches versus GDFSs is only apparent but minor in the last iteration in IDA* and DFS*. Moreover, the thresholds used in the last iteration of IDA* and DFS* in solving fifteen-puzzle and maze problems are either very close to the optimal solution value or exactly equal to the optimal solution value. This reduces the benefit of band searches, which keep a set of nodes in memory and expands in a best-first fashion. Band search will work well when an accurate estimation of thresholds in the last iteration is difficult.

An average execution time may be biased heavily because speedups (having values between 1 and ∞ with respect to GDFS) carry more weight on the average than slowdowns (having speedups close to or less than 1 with respect to GDFS). The same problem happens when raw execution times or normalized times are used.

One way to cope with this problem is to normalize speedups by taking one search algorithm as a reference (say, REF) and normalizing individual execution time (say, for search instance *i*) of the search algorithm being tested (say, A) so that the sum of normalized times of algorithm A and that of algorithm REF is 1. That is,

$$t_{A,i}^{ewn[A,REF]} = del\ \frac{t_{A,i}}{t_{A,i} + t_{REF,i}}, \tag{4.2.a}$$

$$t_{REF,i}^{ewn[A,REF]} = del\ \frac{t_{REF,i}}{t_{A,i} + t_{REF,i}}, \tag{4.2.b}$$

where $t_{A,i}$ and $t_{REF,i}$ are, respectively, the execution times of algorithms A and REF in solving search instance *i*, and EWN denotes this normalization method. The superscripts in

the above two equations indicate the search algorithm under consideration and the reference search algorithm. The subscripts indicate the search algorithm being normalized and the search instance. The advantage of using this normalization method is that each search instance carries an equal weight in computing the final average value.

$$t_{A,i}^{ewn[A,REF]} + t_{REF,i}^{ewn[A,REF]} = 1 ,$$

$$\bar{t}_{A}^{ewn[A,REF]} + \bar{t}_{REF}^{ewn[A,REF]} = \frac{1}{n}\sum_{i=1}^{n} t_{A,i}^{ewn[A,REF]} + \frac{1}{n}\sum_{i=1}^{n} t_{REF,i}^{ewn[A,REF]}$$

$$= \frac{1}{n}\sum_{1=1}^{n}\left(t_{A,i}^{ewn[A,REF]} + t_{REF,i}^{ewn[A,REF]} \right) = 1 ,$$

where $\bar{t}$ is the average time, and n is the number of instances tested. Note that execution times normalized this way are always between 0 and 1.

We can calculate the average normalized speedup based on the EWN times by using the ratio of the total sums of the EWN times. That is,

$$p_{A}^{ewn[A,REF]} = del \frac{\sum_{i=1}^{n} t_{REF,i}^{ewn[A,REF]}}{\sum_{i=1}^{n} t_{A,i}^{ewn[A,REF]}} = \frac{\bar{t}_{REF}^{ewn[A,REF]}}{\bar{t}_{A}^{ewn[A,REF]}} . \quad (4.3)$$

Note that the average normalized speedup is different from the traditional average speedup. The traditional speedup of algorithm A with respect to algorithm REF based on the raw execution times is

$$\bar{s}_{A}^{[REF]} = del \frac{1}{n}\sum_{i=1}^{n} \frac{t_{REF,i}}{t_{A,i}} = \frac{1}{n}\sum_{i=1}^{n} \frac{t_{REF,i}^{ewn[A,REF]}}{t_{A,i}^{ewn[A,REF]}} . \quad (4.4)$$

If we redefine normalized speedup based on giving equal weight on total time spent to run each problem instance, then we have

$$s_{A,i}^{ewn[A,REF]} = del \frac{t_{REF,i}}{t_{A,i} + t_{REF,i}} . \quad (4.5)$$

Note that the value of $s_{A,i}^{ewn[A,REF]}$ is always between 0 and 1. Also note that $s_{A,i}^{ewn[A,REF]} = t_{REF,i}^{ewn[A,REF]}$ according to Eqs. (4.2.b) and (4.5). The traditional speedup and the EWN speedup have the following relationship.

$$s_{A,i}^{ewn[A,REF]} = \frac{s_{A,i}^{[REF]}}{1 + s_{A,i}^{[REF]}},$$

$$s_{A,i}^{[REF]} = \frac{s_{A,i}^{ewn[A,REF]}}{1 - s_{A,i}^{ewn[A,REF]}}.$$

One interesting feature of the EWN method is that the average normalized speedup defined in Eq. (4.3) has a one-to-one relation to the EWN speedup defined in Eq. (4.5). This feature can be illustrated by the following simple derivations. The traditional definition of speedup based on the EWN speedup can be computed as

$$S = del \frac{\bar{s}_A^{ewn[A,REF]}}{1 - \bar{s}_A^{ewn[A,REF]}}.$$

In fact, S is equal to $p_A^{ewn[A,REF]}$, because

$$S = \frac{\bar{t}_{REF}^{ewn[A,REF]}}{1 - \bar{t}_{REF}^{ewn[A,REF]}} = \frac{\bar{t}_{REF}^{ewn[A,REF]}}{\bar{t}_A^{ewn[A,REF]}} = p_A^{ewn[A,REF]}.$$

This implies that the EWN method can be applied to raw execution times either in the form of EWN time or in the form of EWN speedup because they are equivalent.

The performance values shown in Figures 4.8-4.13 are summarized in Table 4.2. The performance values are averaged by the four methods defined in Eqs. (4.2), (4.3), (4.1), and (4.4). The reason to show all these four is that different methods may propose different best algorithms.

Table 4.2. Average performance of BS[s,u;*] in solving fifty random instances of each of TSP(1;20), KS(0.5;80), PP(3;18), and VC(0.1;40) problems, and Maze(0.2;40), and fifteen-puzzle problems.

Search Instance	Algorithm A	$\bar{t}_A^{ewn[A,GDFS]}$ Eq. (4.2)	$p_A^{ewn[A,GDFS]}$ Eq. (4.3)	$\bar{t}_A^{[OPTD]}$ Eq. (4.1)	$\bar{s}_A^{[GDFS]}$ Eq. (4.4)
ΔTSP(1;20)	GDFS	0.500	1.00	1.46	1.00
	BS[s,u;5]	0.453	1.21	1.22	1.28
	BS[s,u;10]	0.440	1.27	1.10	1.38
	BS[s,u;30)	0.428	1.34	1.02	1.43
	BS[s,u;50]	0.428	1.34	1.02	1.43
PP(3;18)	GDFS	0.500	1.00	1.28	1.00
	BS[s,u;5]	0.453	1.21	1.05	1.23
	BS[s,u;10]	0.447	1.24	1.02	1.26
	BS[s,u;30)	0.444	1.25	1.01	1.27
	BS[s,u;50]	0.442	1.26	1.00	1.28
KS(0.5;80)	GDFS	0.500	1.00	1.31	1.00
	BS[s,u;5]	0.232	3.31	0.39	3.81
	BS[s,u;10]	0.205	3.88	0.33	4.49
	BS[s,u;30)	0.180	4.56	0.28	5.32
	BS[s,u;50]	0.172	4.81	0.27	5.63
VC(0.1;40)	GDFS	0.500	1.00	25.93	1.00
	BS[s,u;5]	0.293	2.41	9.70	6.12
	BS[s,u;10]	0.187	4.35	4.05	11.55
	BS[s,u;30)	0.135	6.41	2.51	15.58
	BS[s,u;50]	0.114	7.77	1.87	18.11
Search Instance	Algorithm A	$\bar{t}_A^{ewn[A,DFS*]}$ Eq. (4.2)	$p_A^{ewn[A,DFS*]}$ Eq. (4.3)	$\bar{t}_A^{[OPTD]}$ Eq. (4.1)	$\bar{s}_A^{[DFS*]}$ Eq. (4.4)
Maze(0.2;40)	DFS*	0.500	1.00	3.73	1.00
	DFS*_BS[s,u;5]	0.449	1.23	2.11	2.78
	DFS*_BS[s,u;10]	0.454	1.20	3.29	3.25
	DFS*_BS[s,u;30)	0.423	1.36	2.36	3.39
	DFS*_BS[s,u;50]	0.417	1.40	2.31	3.45
Search Instance	Algorithm A	$\bar{t}_A^{ewn[A,IDA*]}$ Eq. (4.2)	$p_A^{ewn[A,IDA*]}$ Eq. (4.3)	$\bar{t}_A^{[OPTD]}$ Eq. (4.1)	$\bar{s}_A^{[IDA*]}$ Eq. (4.4)
Fifteen Puzzle	IDA*	0.500	1.00	1.63	1.00
	IDA*_BS[s,u;5]	0.497	1.01	1.75	1.05
	IDA*_BS[s,u;10]	0.493	1.03	1.74	1.10
	IDA*_BS[s,u;30)	0.482	1.07	1.71	1.17
	IDA*_BS[s,u;50]	0.475	1.11	1.67	1.22

From Table 4.2, we find that the average performance, in terms of $\bar{t}_A^{ewn[A,GDFS]}$, $p_A^{ewn[A,GDFS]}$, $\bar{t}_A^{[OPTD]}$, and $\bar{s}_A^{[GDFS]}$, in solving the TSP(1;20), KS(0.5;80), PP(3;18), and VC(0.1;40) problems, shows monotonic trends with band widths. That is, band searches with larger band widths perform better in terms of logical time. However, for DFS*'s using band searches as their primitive searches, we find band-width anomalies in DFS*_BS[s,u;5]. For IDA*'s using band search as their primitive searches, we find anomalies in IDA*_GDFS when $\bar{t}_A^{[OPTD]}$ is used. These happen because band searches with larger band widths have large execution times that bias the average speedup calculated.

The physical execution time of static band searches in solving ten random instances of the TSP(1,20), PP(3;18), KS(0.5;80), and VC(0.1;40) problems are shown in Figures 4.14-4.17. We can find some occurrences of band-width anomalies in these graphs. We also find that the physical execution times decrease as the band widths increase because band searches with larger band widths can find an optimal solution faster in terms of logical time. However, after a certain threshold in the size of band widths, the physical execution times increase as band widths are increased due to the excessive overhead in sorting search nodes in each band. From these graphs in Figures 4.14-4.17, we find that the best choice of the uniform band width is 10 which in general can result in small logical execution time as well as small physical execution time.

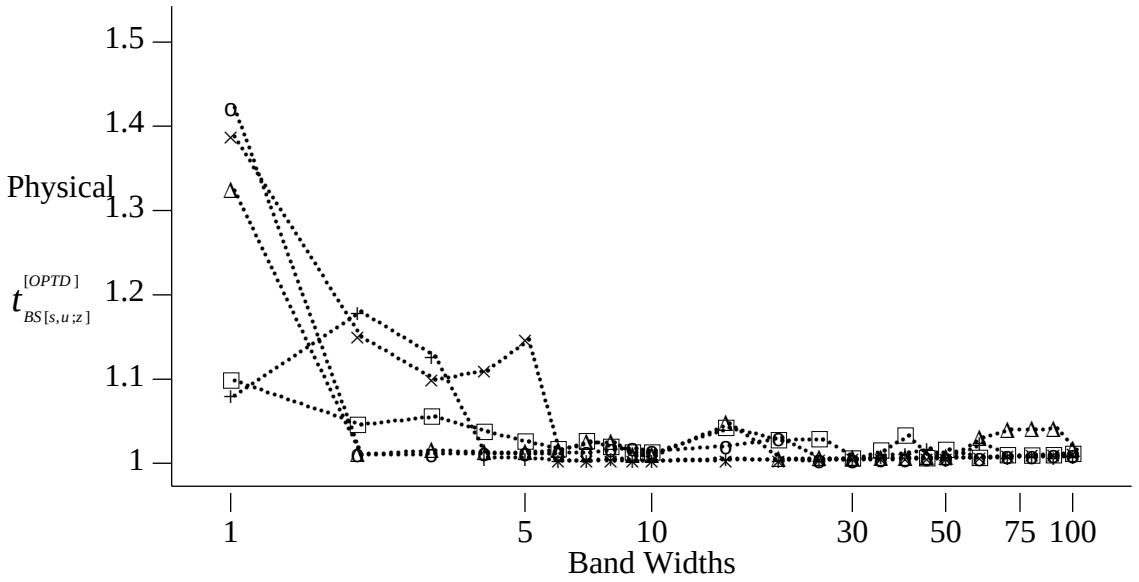


Figure 4.14. Profiles of physical execution times versus band widths of BS[s,u;z], z=1,2,3,4,5,6,7,8,9,10,15,20,25,30,35,40,45,50,60,70,80,90,100, in solving instances TSP(1;20,1) (‘‘□’’), TSP(1;20,2) (‘‘○’’), TSP(1;20,3) (‘‘Δ’’), TSP(1;20,4) (‘‘+’’), and TSP(1;20,5) (‘‘×’’), where physical execution times are normalized with respect to those of OPTD.

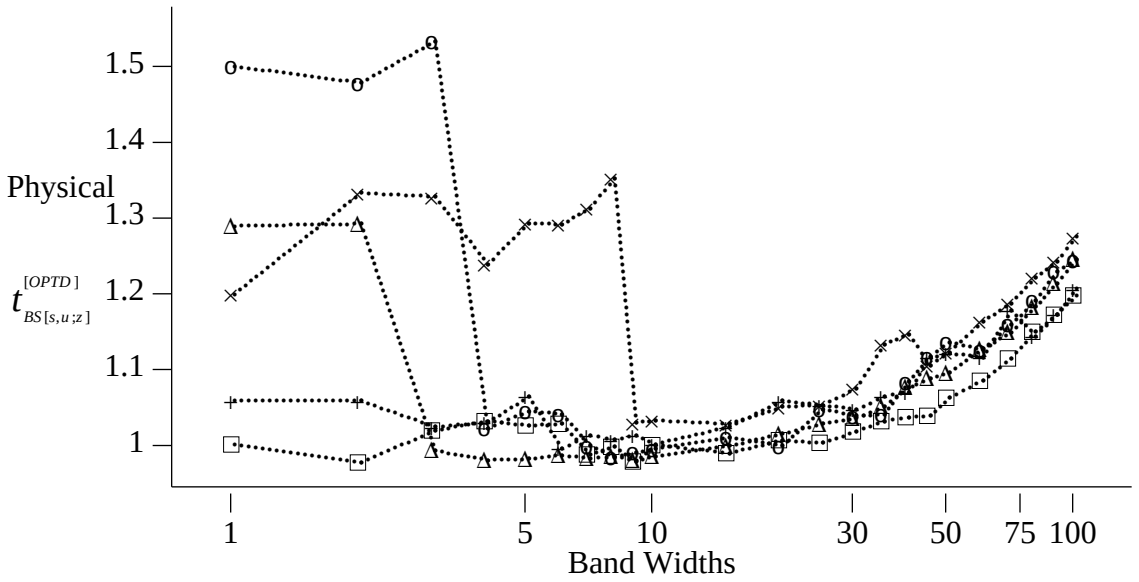


Figure 4.15. Profiles of physical execution times versus band widths BS[s,u;z], z=1,2,3,4,5,6,7,8,9,10,15,20,25,30,35,40,45,50,60,70,80,90,100, PP(3;18,1) (‘‘□’’), PP(3;18,2) (‘‘○’’), PP(3;18,3) (‘‘Δ’’), PP(3;18,4) (‘‘+’’), and PP(3;18,5) (‘‘×’’), where physical execution times are normalized with respect to those of OPTD.

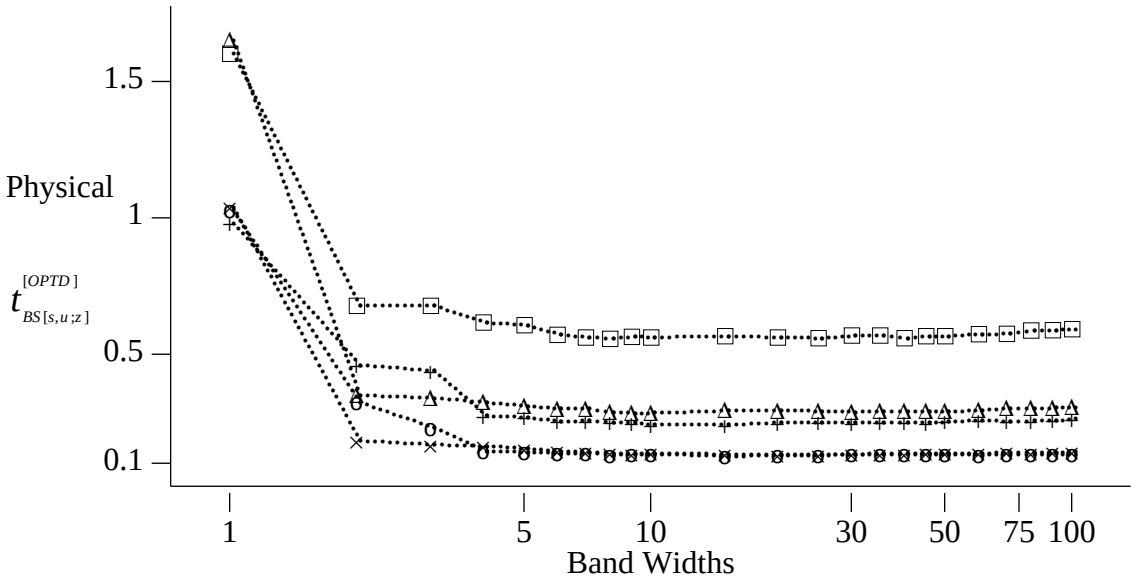


Figure 4.16. Profiles of physical execution times versus band widths BS[s,u;z], z=1,2,3,4,5,6,7,8,9,10,15,20,25,30,35,40,45,50,60,70,80,90,100, KS(0.5;80,1) (‘‘□’’), KS(0.5;80,2) (‘‘○’’), KS(0.5;80,3) (‘‘Δ’’), KS(0.5;80,4) (‘‘+’’), and KS(0.5;80,5) (‘‘×’’), where physical execution times are normalized with respect to those of OPTD.

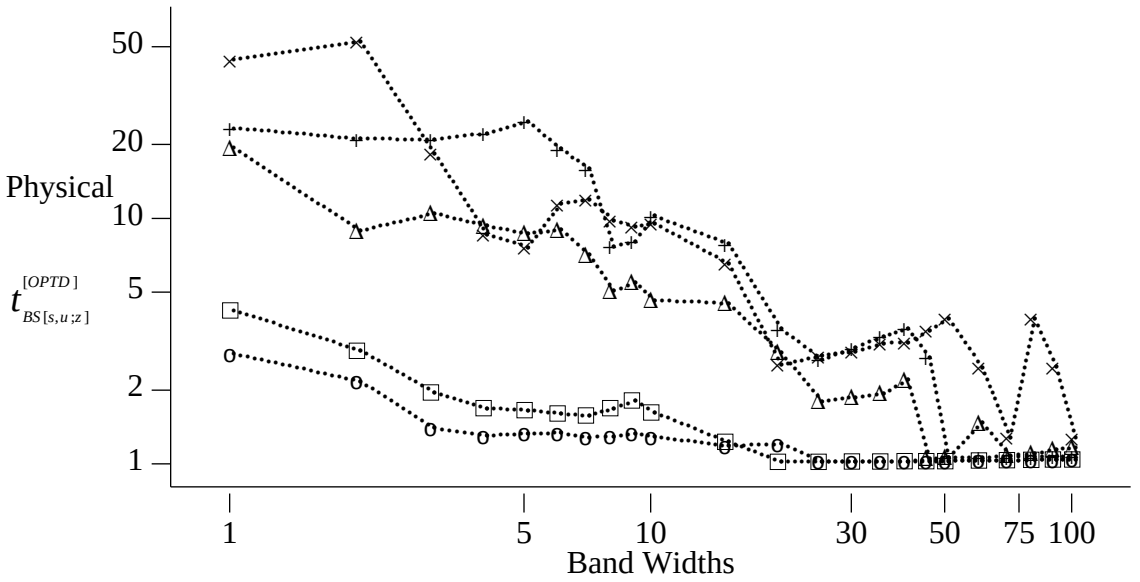


Figure 4.17. Profiles of physical execution times versus band widths BS[s,u;z], z=1,2,3,4,5,6,7,8,9,10,15,20,25,30,35,40,45,50,60,70,80,90,100, VC(0.1;40,1) (‘‘□’’), VC(0.1;40,2) (‘‘○’’), VC(0.1;40,3) (‘‘Δ’’), VC(0.1;40,4) (‘‘+’’), and VC(0.1;40,5) (‘‘×’’), where physical execution times are normalized with respect to those of OPTD.

We also study the performance of static band searches whose band widths are functions of level numbers in the search tree. The performance plots of logical execution times versus various band-width limits for five different static band searches in solving fifty random instances of the TSP(1,20), PP(3;18), KS(0.5;80), and VC(0.1;40) problems are shown, respectively, in Figures 4.18-4.21. These five static band searches include BS[s,u;z], BS[s,li;1,z], BS[s,ld;1,z], BS[s,ei;1,z], and BS[s,ed;1,z], z=1,5,10,30,50. From these graphs, we find that BS[s,u;z] is generally the best. Also, we find that searches with decreasing band widths are generally better than those with increasing band widths. One reason is that less accuracy in predicting goodness of search nodes occurs in levels close to the root so that including more nodes in these levels can result in better performance.

4.5.2. Time-constrained band search

In this section, we study the performance of time-constrained band searches, including dynamic band searches and iterative first-band searches.

The dynamic band searches to be evaluated include those that adjust one band width at a time and those that adjust multiple band widths a time. We empirically evaluate dynamic band searches on numerous search instances in terms of their performance for finding both feasible and optimal solutions, and in terms of their performance in refining feasible solutions obtained under various time limits.

The iterative first-band search to be evaluated uses a growth rate of 2 so that the times spent in successive iterations grow exponentially.

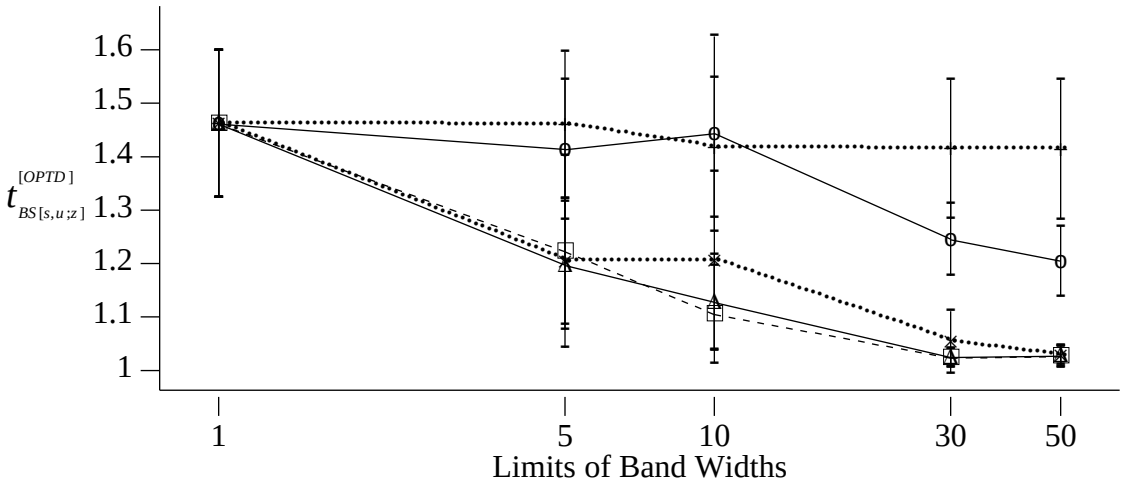


Figure 4.18. Profiles of logical execution times versus band widths for five band searches BS[s,u;z], BS[s,li;1,z], BS[s,ld;1,z], BS[s,ei;1,z], and BS[s,ed;1,z], z=1,5,10,30,50, in solving fifty instances of TSP(1;20), where logical execution time is normalized with respect to those of OPTD, the dashed curve (with symbol ‘‘□’’) is for BS[s,u;z] the solid curves are for BS[s,li;1,z] (with symbol ‘‘○’’) and BS[s,ld;1,z] (with symbol ‘‘Δ’’), and the dotted curves are for BS[s,ei;1,z] (with symbol ‘‘+’’) and BS[s,ed;1,z] (with symbol ‘‘×’’).

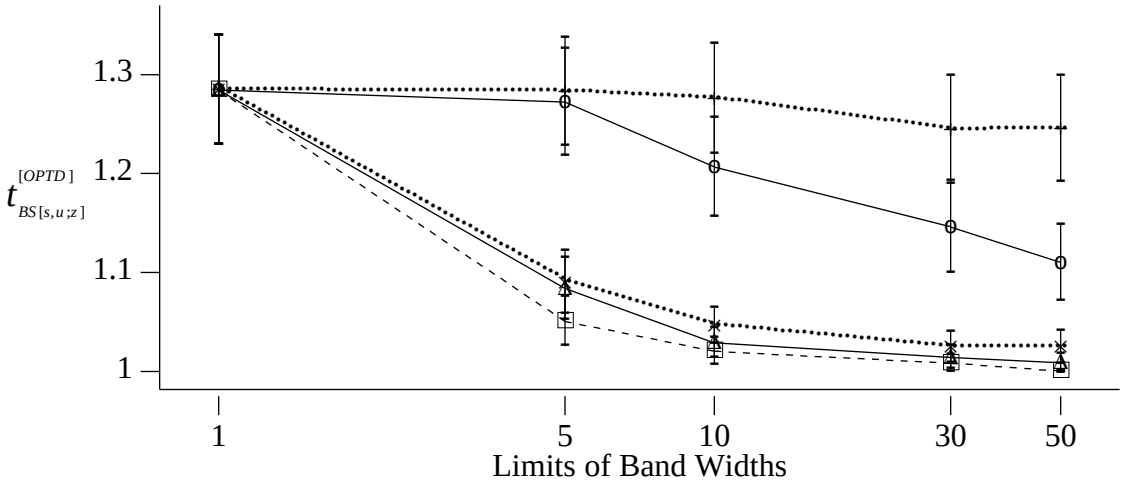


Figure 4.19. Profiles of logical execution times versus band widths for five band searches BS[s,u;z], BS[s,li;1,z], BS[s,ld;1,z], BS[s,ei;1,z], and BS[s,ed;1,z], z=1,5,10,30,50, in solving fifty instances of PP(3;18), where logical execution time is normalized with respect to those of OPTD, the dashed curve (with symbol ‘‘□’’) is for BS[s,u;z] the solid curves are for BS[s,li;1,z] (with symbol ‘‘○’’) and BS[s,ld;1,z] (with symbol ‘‘Δ’’), and the dotted curves are for BS[s,ei;1,z] (with symbol ‘‘+’’) and BS[s,ed;1,z] (with symbol ‘‘×’’).

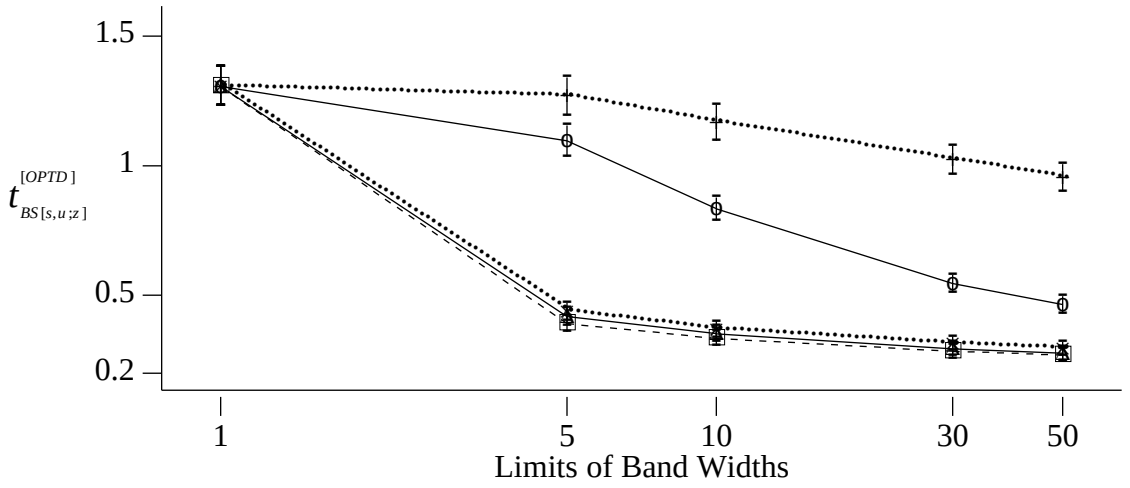


Figure 4.20. Profiles of logical execution times versus band widths for five band searches BS[s,u;z], BS[s,li;1,z], BS[s,ld;1,z], BS[s,ei;1,z], and BS[s,ed;1,z], z=1,5,10,30,50, in solving fifty instances of KS(0.5;80), where logical execution time is normalized with respect to those of OPTD, the dashed curve (with symbol ‘‘□’’) is for BS[s,u;z] the solid curves are for BS[s,li;1,z] (with symbol ‘‘○’’) and BS[s,ld;1,z] (with symbol ‘‘Δ’’), and the dotted curves are for BS[s,ei;1,z] (with symbol ‘‘+’’) and BS[s,ed;1,z] (with symbol ‘‘×’’).

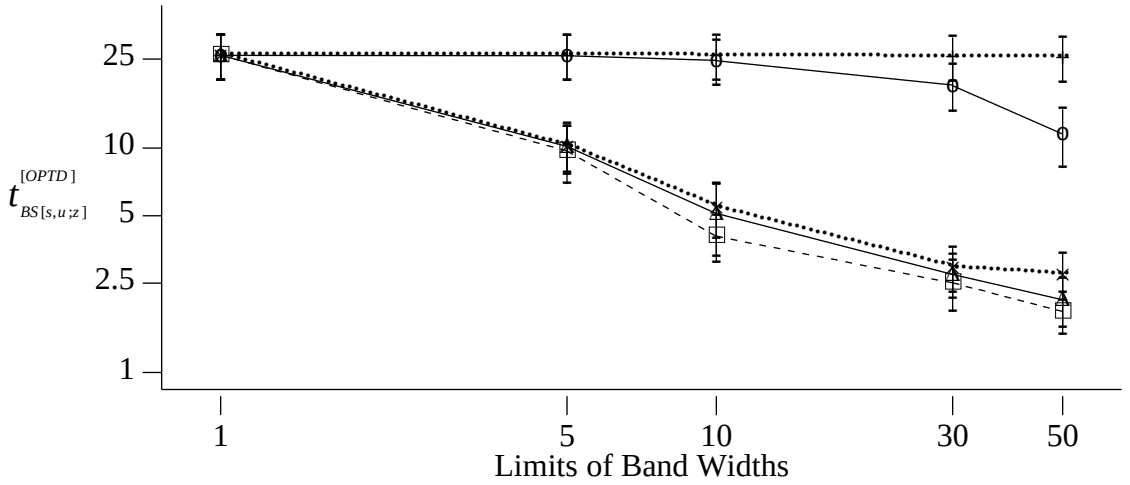


Figure 4.21. Profiles of logical execution times versus band widths for five band searches BS[s,u;z], BS[s,li;1,z], BS[s,ld;1,z], BS[s,ei;1,z], and BS[s,ed;1,z], z=1,5,10,30,50, in solving fifty instances of VC(0.1;40), where logical execution time is normalized with respect to those of OPTD, the dashed curve (with symbol ‘‘□’’) is for BS[s,u;z] the solid curves are for BS[s,li;1,z] (with symbol ‘‘○’’) and BS[s,ld;1,z] (with symbol ‘‘Δ’’), and the dotted curves are for BS[s,ei;1,z] (with symbol ‘‘+’’) and BS[s,ed;1,z] (with symbol ‘‘×’’).

The target search problems are sparse COPs. The test search problems include TSP(0.05), TSP(0.1), TSP(0.2), Maze(0), Maze(0.1), and Maze(0.2).

A dynamic band search starts with a predefined initial band width and adapts its band widths at run time. The initial band width can be small or large, and we empirically find that a good choice of the initial band width depends on the underlying search problems to be solved. To choose a good initial band width for the TSP and maze problems, we empirically evaluate static band searches whose band widths are functions of the level numbers in the search tree. Figures 4.22 and 4.23 show the average normalized times (with respect to those of OPTD), where vertical bars are 90% confidence intervals on the averages, assuming the averages are normally distributed.

Empirically, we find searches with smaller initial band widths perform better than those with larger initial band widths in solving the TSP problem. This happens because the lower bound of search nodes in levels close to the root of the search tree can give misleading information in guiding the search; therefore, seemingly inferior nodes should also be *included* for expansion in these levels. As a result, we choose the maximum band width allowed as the initial band width in solving the TSP problem.

Interestingly, from the graphs in Figure 4.23, we find that searches with increasing band widths by levels perform better than those with decreasing band widths by levels. Note that this behavior is different from that in solving the TSP problem. Therefore, we choose 1 as the initial band width in solving the maze problem.

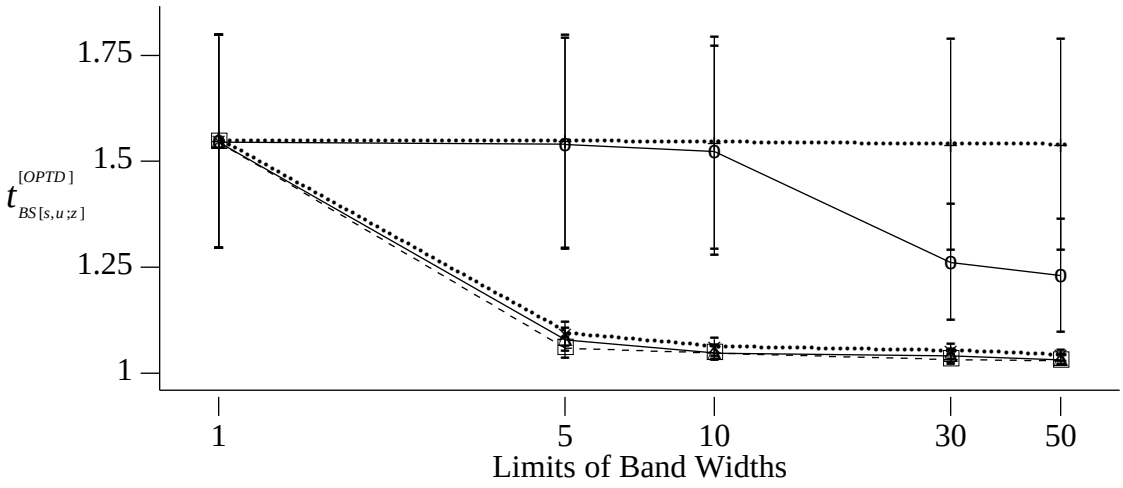


Figure 4.22. Profiles of logical execution times versus band widths for five band searches BS[s,u;z], BS[s,li;1,z], BS[s,ld;1,z], BS[s,ei;1,z], and BS[s,ed;1,z], z=1,5,10,30,50, in solving fifty instances of TSP(0.1;20), where logical execution time is normalized with respect to those of OPTD, the dashed curve (with symbol ‘‘□’’) is for BS[s,u;z] the solid curves are for BS[s,li;1,z] (with symbol ‘‘○’’) and BS[s,ld;1,z] (with symbol ‘‘Δ’’), and the dotted curves are for BS[s,ei;1,z] (with symbol ‘‘+’’) and BS[s,ed;1,z] (with symbol ‘‘×’’).

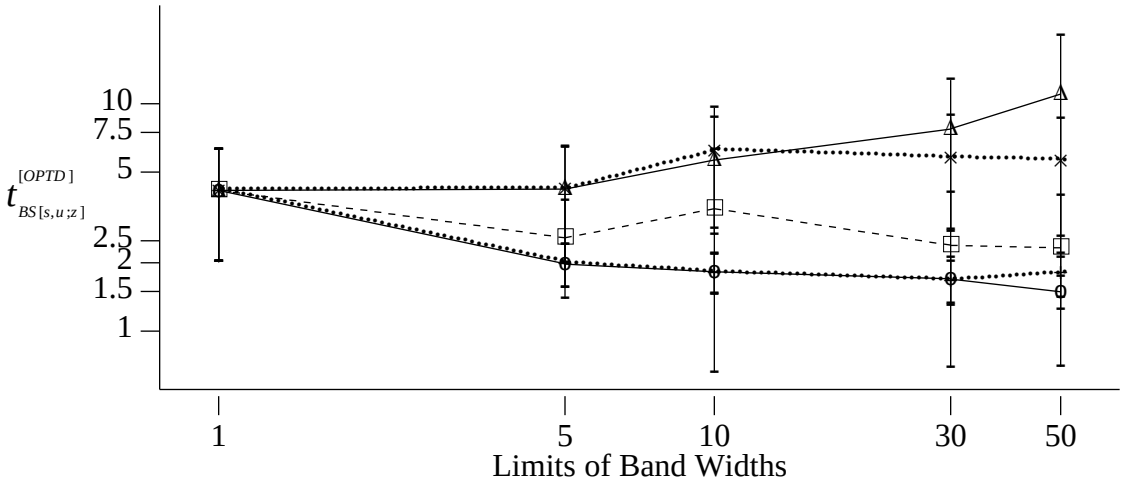


Figure 4.23. Profiles of logical execution times versus band widths for five band searches BS[s,u;z], BS[s,li;1,z], BS[s,ld;1,z], BS[s,ei;1,z], and BS[s,ed;1,z], z=1,5,10,30,50, in solving fifty instances of Maze(0.2;40), where logical execution time is normalized with respect to those of OPTD, the dashed curve (with symbol ‘‘□’’) is for BS[s,u;z] the solid curves are for BS[s,li;1,z] (with symbol ‘‘○’’) and BS[s,ld;1,z] (with symbol ‘‘Δ’’), and the dotted curves are for BS[s,ei;1,z] (with symbol ‘‘+’’) and BS[s,ed;1,z] (with symbol ‘‘×’’).

Empirically, we find that searches with smaller initial band widths perform better than those with larger initial band widths in solving the maze problem. This happens because lower bounds of search nodes in levels close to the root of the search tree are consistent with the optimal solution value and can give correct information in guiding the search; consequently, seemingly inferior nodes should be *excluded* for expansion in these levels.

For sparse COPs, the search algorithm is designed to find good feasible solutions under time limits. The objective of solving constrained minimization problems in limited time is not always well-defined in the sense that in some cases no *practical* algorithm can find a feasible solution under limited time. In this case, we are interested in developing a search algorithm that would require the smallest additional overhead to find a feasible solution.

On the other hand, the objective of solving constrained minimization problems in limited time is well-defined if a practical search algorithm exists for finding a feasible solution in the time limit. In this case, we would like to develop a search algorithm that can find a better solution within the time limit. To ensure that we have a common basis for comparison, we assume that a GDFS guided by lower-bound values of search nodes is used as a reference algorithm.

In short, we identify three possible objectives of an algorithm for searching in limited time.

(1) If a solution can be found by any practical algorithm in the time limit, the algorithm designed should (a) find a solution as fast as possible, and (b) make this first solution as good as possible.

(2) If a solution can be found by any practical algorithm in the time limit, then the algorithm designed should (a) find better solutions as fast as possible, and (b) improve the *estimated quality* of solutions as fast as possible.

(3) If a solution cannot be found by any practical algorithm in the time limit, then the algorithm designed should make the best progress in finding a solution; that is, if the search were continued, the algorithm designed should require the minimum amount of time (among all practical algorithms) to find a solution.

It is very difficult, however, to design a algorithm that achieves all these objectives simultaneously. We anticipate that such an algorithm would have two components, one trying to find a solution, and another trying to refine the incumbent.

Under a strict time limit, finding a feasible solution is difficult. Goal (1.a) is, therefore, more important that Goal (1.b). In this section, we evaluate the performance of search algorithms in terms of the times for finding feasible solutions.

To find the best algorithm that can refine the solutions quickly, we need to evaluate all possible algorithms for all possible time limits. This is prohibitively expensive to do in a reasonable amount of time. Further, one search algorithm that is worse than a second algorithm can outperform the second under different time limits. We evaluate solution refinements of search algorithms by comparing the quality values of solutions found under various time limits. This method can also help evaluate search algorithms for Goal (3).

To accommodate the goal of refining solutions, we can also adopt an approximate goal that minimizes the time in finding an optimal solution. That is, we evaluate the performance of search algorithms in terms of times for finding optimal solutions.

In summary, we will evaluate search algorithms in terms of (a) times for finding feasible solutions, (b) times for finding optimal solutions, and (c) search progress in terms of the quality of the solution delivered within the time limit.

The evaluations of band searches in solving fifty random instances of the TSP(0.05;22), TSP(0.1;22), and TSP(0.2;22) problems based on the first two criteria are summarized in Tables 4.3-4.5. These tables contain two major parts: one evaluating the search for finding a feasible solution and the other evaluating the search for finding optimal solutions. The speedup is computed with respect to GDFS; that is, the average speedup of algorithm A is $\overline{s}_A^{[GDFS]}$ (Eq. (4.4)). The normalized time is computed with respect to the time taken by OPTD, that is, the average normalized time of algorithm A is $\overline{t}_A^{[OPTD]}$ (Eq. (4.1)). We also record the standard deviation of speedups and normalized times in order to examine the spread of these speedups and times. Both average speedups and normalized times are shown because the former may be biased by large speedups (caused by large execution times of GDFSs) and the latter may be biased by large execution times of the target search algorithms. Therefore, by showing both average values, we can have a balanced view of these algorithms. For the various algorithms evaluated, we have the following observations.

(a) GDFS_BS use GDFS to find a solution and switch to band search afterwards. Therefore, its performance in finding a solution is the same as that of GDFS. However, its performance in finding an optimal solution may be different from that of GDFS. From Tables 4.3-4.5, we also find that these hybrid algorithms perform better than GDFS in finding an optimal solution.

Table 4.3. Average performance of various band searches for solving fifty random instances of TSP(0.05;22).

Search Algorithms	Finding First Solution				Finding Optimal Solution			
	Speedup		Normalized Time		Speedup		Normalized Time	
	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation
GDFS	1.000	0.000	0.303	0.213	1.000	0.000	0.367	0.225
GDFS_BS[s,u,ii;2]					0.990	0.824	0.557	0.480
GDFS_BS[d;1,10]					1.071	0.370	0.354	0.214
GDFS_BS[d,bwi;1,10]					1.000	0.000	0.367	0.225
GDFS_BS[d;10,10]					1.106	0.457	0.355	0.243
GDFS_BS[d,bwi;10,10]					1.056	0.136	0.348	0.218
GDFS_BS[s,u;10]					1.055	0.133	0.349	0.217
BS[s,u,ii;2]	1.537	2.659	0.706	0.685	1.165	1.900	0.929	0.717
BS[d;1,10]	1.165	0.643	0.274	0.187	1.314	1.179	0.322	0.204
BS[d,bwi;1,10]	1.000	0.000	0.303	0.213	1.000	0.000	0.367	0.225
BS[d;10,10]	2.745	3.290	0.225	0.213	2.783	3.347	0.279	0.247
BS[d,bwi;10,10]					2.708	3.330	0.284	0.247
BS[s,u;10]					2.722	3.327	0.282	0.247

Table 4.4. Average performance of various band searches for solving fifty random instances of TSP(0.1;22).

Search Algorithms	Finding First Solution				Finding Optimal Solution			
	Speedup		Normalized Time		Speedup		Normalized Time	
	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation
GDFS	1.000	0.000	0.399	0.963	1.000	0.000	0.814	1.133
GDFS_BS[s,u,ii;2]					1.693	0.347	1.102	1.630
GDFS_BS[d;1,10]					1.693	2.177	0.681	1.011
GDFS_BS[d,bwi;1,10]					1.000	0.000	0.814	1.133
GDFS_BS[d;10,10]					1.579	1.884	0.714	1.062
GDFS_BS[d,bwi;10,10]					1.328	1.335	0.727	1.092
GDFS_BS[s,u;10]					1.335	1.338	0.723	1.091
BS[s,u,ii;2]	3.202	3.966	0.287	0.423	3.000	3.798	0.811	1.433
BS[d;1,10]	1.638	1.901	0.251	0.636	2.426	2.659	0.495	0.697
BS[d,bwi;1,10]	1.000	0.000	0.399	0.963	1.000	0.000	0.814	1.133
BS[d;10,10]	3.789	4.120	0.117	0.146	4.168	4.071	0.237	0.218
BS[d,bwi;10,10]					4.186	4.060	0.232	0.211
BS[s,u;10]					4.172	4.044	0.231	0.207

Table 4.5. Average performance of various band searches for solving fifty random instances of TSP(0.2;22).

Search Algorithms	Finding First Solution				Finding Optimal Solution			
	Speedup		Normalized Time		Speedup		Normalized Time	
	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation
GDFS	1.000	0.000	0.232	0.651	1.000	0.000	0.975	1.439
GDFS_BS[s,u,ii;2]					3.125	3.512	0.979	2.006
GDFS_BS[d;1,10]					2.061	2.203	0.572	0.752
GDFS_BS[d,bwi;1,10]					1.000	0.000	0.975	1.439
GDFS_BS[d;10,10]					2.459	2.766	0.556	0.765
GDFS_BS[d,bwi;10,10]					1.476	1.368	0.702	0.915
GDFS_BS[s,u;10]					1.517	1.372	0.677	0.886
BS[s,u,ii;2]	4.016	4.035	0.014	0.026	4.265	4.198	0.807	1.954
BS[d;1,10]	2.875	3.445	0.051	0.149	3.052	3.054	0.365	0.378
BS[d,bwi;1,10]	1.000	0.000	0.232	0.651	1.000	0.000	0.975	1.439
BS[d;10,10]	4.666	4.184	0.013	0.031	5.073	4.145	0.253	0.358
BS[d,bwi;10,10]					5.040	4.123	0.252	0.395
BS[s,u;10]					5.072	4.146	0.253	0.397

(b) BS[d,bwi;1,10] performs the same as GDFS. It can be explained as follows. BS[d,bwi;1,10] increases band widths when and in levels where infeasible solutions are found. However, in TSP, infeasible solutions occur usually at the bottom of the search tree; therefore, the isolated band-width increments usually occur in the levels close to the bottom, where feasible solutions often occur and the band width just increased will soon be decreased.

(c) BS[d;10,10] and BS[d,bwi;10,10] have the same performance as BS[s,u;10] in finding a solution. BS[d;10,10] and BS[d,bwi;10,10] start with initial band widths equal to the maximum band width 10; therefore, the band widths cannot be increased until some band widths are decreased. However, decrements of band widths happen only when solutions are found. In finding a solution, the search stops when a solution is found, which means band widths cannot be changed until the first solution is found. As a result, BS[d;10,10] and BS[d,bwi;10,10] perform the same as BS[s,u;10] in finding a solution.

(d) From Tables 4.3-4.5, we find that BS[d;10,10] and BS[d,bwi;10,10], in general, perform better than the other search algorithms evaluated in terms of both finding a solution and finding an optimal solution. We also find that BS[d;10,10] performs slightly better than BS[d,bwi;10,10].

(e) Standard deviations of speedups and normalized times may be larger than the averages. This implies that the corresponding algorithm may perform very well in terms of speedup because large speedups carry more weight than small speedups and slowdowns. This also implies that the corresponding algorithm may perform very poorly in terms of

normalized times because large normalized times carry more weight than small normalized times.

(f) BS[s,u,ii;2] performs very well in finding a solution for the TSP instances evaluated because these problems have a reasonable number of feasible solutions so that the search algorithm can quickly find a solution deep in the search tree. The search algorithm also has moderately good performance in finding an optimal solution. However, this algorithm may suffer from large memory requirements, especially when finding optimal solutions, hence making it unsuitable for finding optimal solutions under restricted memory.

The evaluations of band searches in solving fifty random instances of the Maze(0;100), Maze(0.1,45), and Maze(0.2,40) problems based on finding feasible and optimal solutions are summarized in Tables 4.6-4.8. The tables also contain two major parts: one evaluating searches for feasible solutions and the other evaluating searches for finding optimal solutions. The speedups and normalized times are computed in the same way as those in Tables 4.3-4.5. For the various algorithms evaluated, we have the following observations.

(a) BS[d,bwi;1,10] does not perform the same as GDFS while solving the TSP. This can be explained as follows. BS[d,bwi;1,10] increases band widths when and in levels where infeasible solutions are found. However, in the maze problem, infeasibilities can occur everywhere in the search tree; therefore, these isolated band-width increments can occur in any level. On the contrary, all feasible and infeasible solutions in a TSP are located at the leaves of the search tree. Therefore, expanding band widths while encountering infeasible solutions cannot improve search in solving a TSP.

Table 4.6. Average performance of various band searches for solving fifty random instances of Maze(0;100).

Search Algorithms	Finding First Solution				Finding Optimal Solution			
	Speedup		Normalized Time		Speedup		Normalized Time	
	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation
GDFS	1.000	0.000	0.421	0.211	1.000	0.000	0.611	0.282
GDFS_BS[s,u,ii;2]					0.831	0.285	0.934	0.689
GDFS_BS[d;1,10]					0.981	0.155	0.636	0.306
GDFS_BS[d,bwi;1,10]					0.996	0.022	0.615	0.286
GDFS_BS[d;10,10]					0.983	0.145	0.635	0.307
GDFS_BS[d,bwi;10,10]					0.966	0.090	0.615	0.286
GDFS_BS[s,u;10]					0.966	0.090	0.643	0.308
BS[s,u,ii;2]	0.392	0.212	1.198	0.447	0.570	0.397	1.237	0.459
BS[d;1,10]	0.586	0.248	0.736	0.263	0.816	0.493	0.816	0.273
BS[d,bwi;1,10]	0.999	0.002	0.421	0.210	0.995	0.025	0.615	0.286
BS[d;10,10]	0.590	0.243	0.732	0.273	0.794	0.449	0.828	0.270
BS[d,bwi;10,10]					0.794	0.449	0.828	0.270
BS[s,u;10]					0.794	0.449	0.828	0.270
BS[s,li;1,10]	0.709	0.245	0.614	0.281	0.791	0.244	0.775	0.276

Table 4.7. Average performance of various band searches for solving fifty random instances of Maze(0.1;45).

Search Algorithms	Finding First Solution				Finding Optimal Solution			
	Speedup		Normalized Time		Speedup		Normalized Time	
	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation
GDFS	1.000	0.000	0.604	1.540	1.000	0.000	1.088	1.670
GDFS_BS[s,u,ii;2]					0.750	0.492	1.667	1.632
GDFS_BS[d;1,10]					1.096	0.712	1.109	1.650
GDFS_BS[d,bwi;1,10]					1.002	0.002	1.088	1.671
GDFS_BS[d;10,10]					1.072	0.689	1.118	1.651
GDFS_BS[d,bwi;10,10]					0.956	0.184	1.104	1.586
GDFS_BS[s,u;10]					0.956	0.184	1.104	1.586
BS[s,u,ii;2]	0.582	0.825	1.069	0.880	0.917	0.988	1.466	0.886
BS[d;1,10]	1.036	1.963	0.533	0.349	1.904	2.529	0.790	0.565
BS[d,bwi;1,10]	0.977	0.111	0.620	1.539	0.985	0.102	1.103	1.667
BS[d;10,10]	1.067	1.973	0.532	0.356	1.884	2.426	0.788	0.552
BS[d,bwi;10,10]					1.884	2.426	0.787	0.552
BS[s,u;10]					1.884	2.426	0.787	0.552
BS[s,li;1,10]	0.635	0.375	0.765	1.526	0.946	0.459	1.112	1.546

Table 4.8. Average performance of various band searches for solving fifty random instances of Maze(0.2;40).

Search Algorithms	Finding First Solution				Finding Optimal Solution			
	Speedup		Normalized Time		Speedup		Normalized Time	
	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation	Average	Standard Deviation
GDFS	1.000	0.000	0.519	1.514	1.000	0.000	1.793	2.393
GDFS_BS[s,u,ii;2]					2.496	3.394	1.511	1.589
GDFS_BS[d;1,10]					2.664	3.111	1.125	1.516
GDFS_BS[d,bwi;1,10]					1.002	0.038	1.791	2.389
GDFS_BS[d;10,10]					2.655	3.170	1.135	1.503
GDFS_BS[d,bwi;10,10]					1.110	0.372	1.745	2.386
GDFS_BS[s,u;10]					1.109	0.371	1.745	2.386
BS[s,u,ii;2]	1.630	2.993	0.328	0.362	3.448	3.967	1.021	0.127
BS[d;1,10]	1.836	3.316	0.344	0.882	3.297	3.674	0.987	1.120
BS[d,bwi;1,10]	0.965	0.155	0.709	2.024	0.989	0.100	1.791	2.389
BS[d;10,10]	1.621	3.155	0.521	1.553	3.439	3.907	1.215	1.736
BS[d,bwi;10,10]					3.431	3.900	1.216	1.735
BS[s,u;10]					3.430	3.898	1.217	1.735
BS[s,li;1,10]	1.481	2.895	0.475	1.491	2.408	3.198	1.233	1.607

(b) From Table 4.6, the best search algorithm in solving Maze(0;100) is GDFS. This happens because Maze(0;100) has only a few feasible solutions and many infeasible solutions that are close to the root (the search tree is narrow and deep). For such a search tree, GDFS is the best and can concentrate its effort on the single main stem of the search tree. In contrast, band search may waste time on other smaller stems that lead to infeasible solutions.

(c) BS[s,u,ii;2] performs poorly in finding feasible solutions and optimal solutions for the Maze(0;100) and Maze(0.1;45) problems evaluated because these problems do not have a reasonable number of feasible solutions. For Maze(0.2;40), BS[s,u,ii;2] performs very well in finding both feasible and optimal solutions because this problem has many feasible solutions. In general, BS[s,u,ii;2] performs well for problems with a moderately large number of feasible solutions and performs poorly for sparse COPs.

(d) From Tables 4.7 and 4.8, we find that BS[d;1,10], BS[d;10,10], BS[d,bwi;10,10], and BS[s,u;1,10] perform very well in finding both feasible and optimal solutions for the Maze(0.1;45) and Maze(0.2;40) problems. In general, these band searches perform well for problems with moderately many and sparse solutions such as the Maze(0.1;45) and Maze(0.2;40) problems, and GDFS performs well for problems with very sparse solutions such as the Maze(0,100) problem.

To demonstrate search progress in terms of the quality of the solutions delivered within a time limit, we show the performance of the search algorithms at various time limits.

The search algorithms of interest include BS[s,u,ii;2], BS[s,u;10], BS[d;1,10], BS[d;10,10], and BS[d,bwi;10,10]. The problems evaluated include the TSP(0.05;22), TSP(0.1;22), TSP(0.2;20), Maze(0;100), Maze(0.1;50), and Maze(0.2;45) problems. The performance profiles are shown in Figures 4.24-4.35.

The quality of a solution is measured by its degree of accuracy instead of its degree of approximation. This is done because we may not find any solution within the time limit. The degree of approximation is infinity when we cannot find a solution. This causes problems when we compare algorithms across search instances. In contrast, the degree of accuracy is 0 when we cannot find a solution. Recall that the degree of accuracy is always between 0 and 1.

Similar to the degree of approximation, the degree of accuracy can be asymptotic or exact. The asymptotic accuracy degree (AAcD) is a lower-bound estimator of the exact accuracy degree (EAcD) at run time. The AAcD is computed based on the incumbent and the least lower bound; in contrast, the EAcD is computed based on the incumbent and the optimal solution value.

From all these graphs, we find that solutions are refined in limited discrete steps. This happens because solutions are sparse and the incumbent is changed only when the solution just found is better than the incumbent. Also, we find that the performance profiles shown in Figures 4.24-4.35 are consistent with average performance values shown in Tables 4.3-4.8.

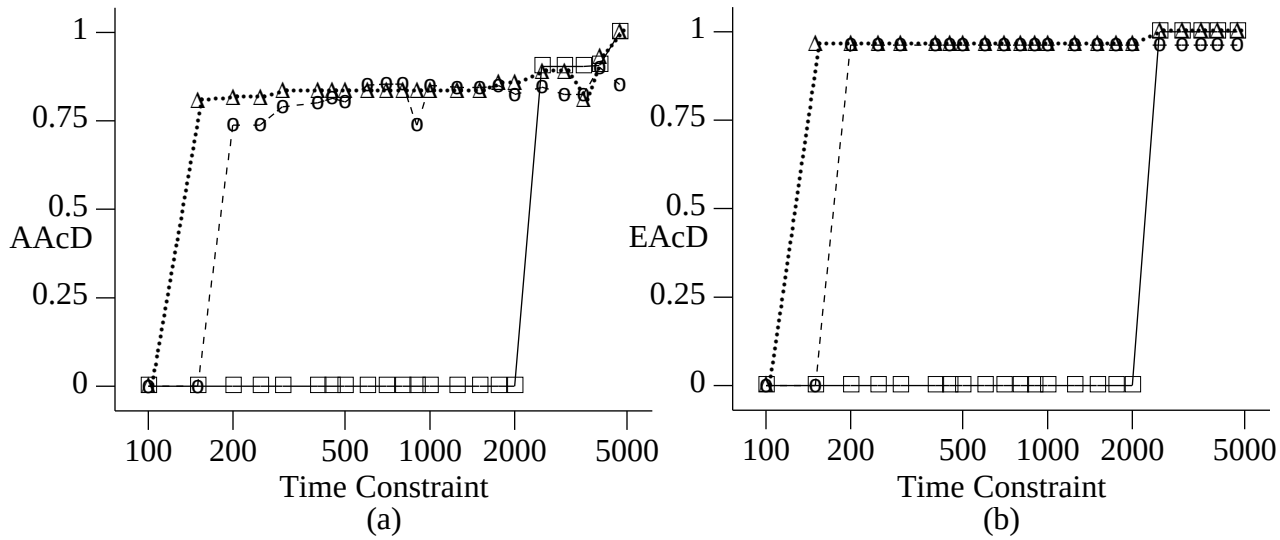


Figure 4.24. Performance profiles of GDFS (‘□’), BS[s,u,ii;2] (‘o’), and BS[s,u;10] (‘Δ’) for solving instances of TSP(0.05;22,15) for various time constraints, where asymptotic accuracy degree (AAcD) is shown in (a) and exact accuracy degree (EAcD) is in (b).

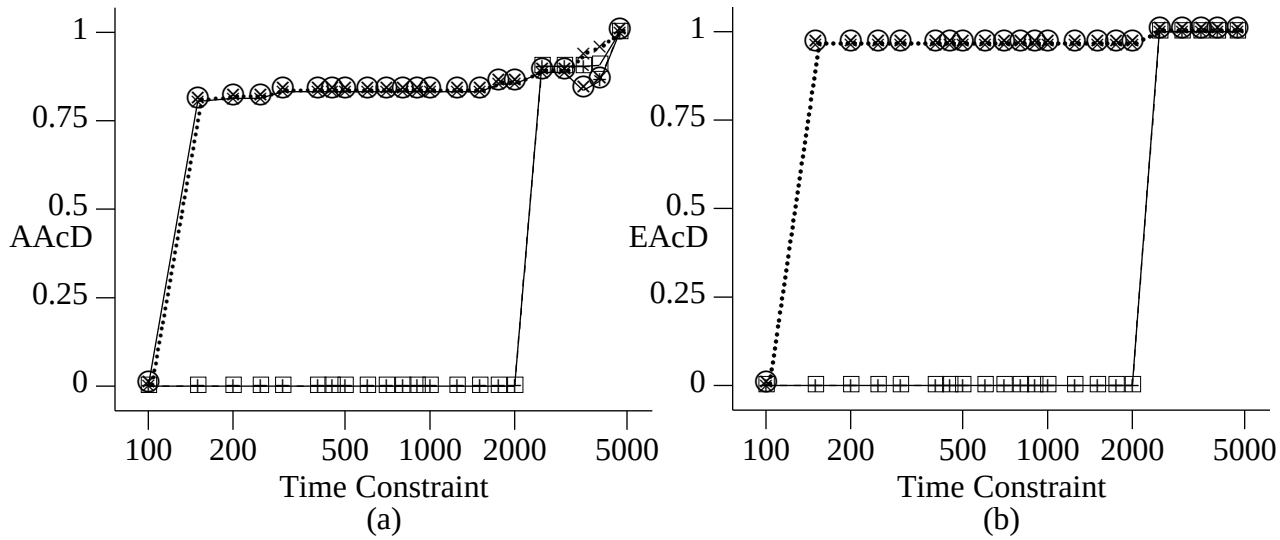


Figure 4.25. Performance profiles of GDFS (‘□’), BS[d;1,10] (‘+’), BS[d;10,10] (‘×’), and BS[d,bwi;10,10] (‘⊗’) for solving instances of TSP(0.05;22,15) for various time constraints, where asymptotic accuracy degree (AAcD) is shown in (a) and exact accuracy degree (EAcD) is in (b).

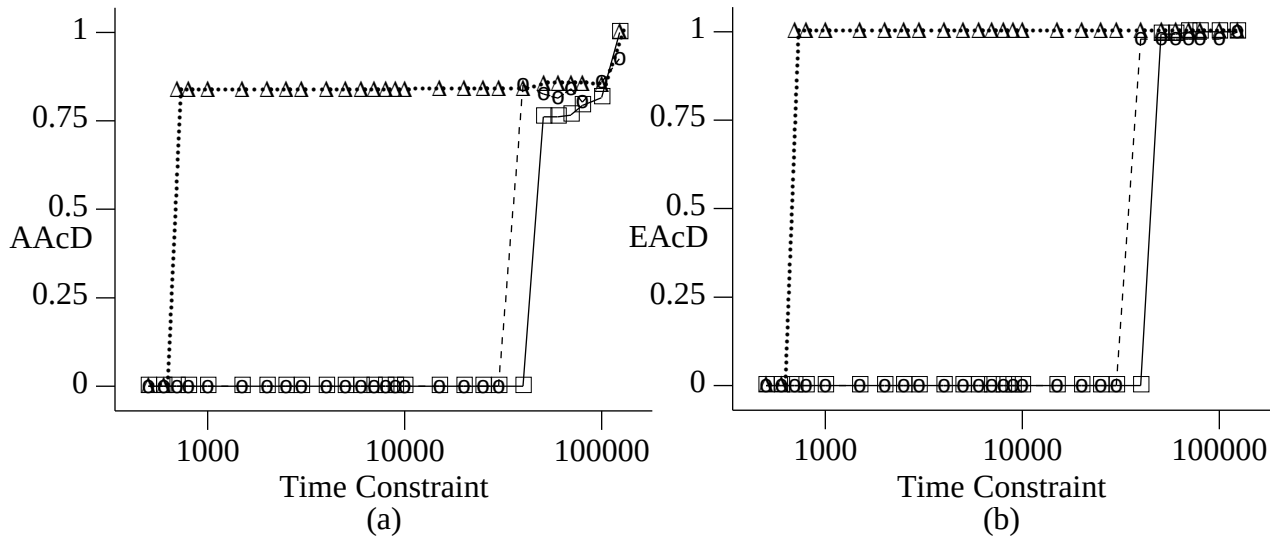


Figure 4.26. Performance profiles of GDFS (‘□’), BS[s,u,ii;2] (‘o’), and BS[s,u;10] (‘Δ’) for solving instances of TSP(0.1;22,5) for various time constraints, where asymptotic accuracy degree (AAcD) is shown in (a) and exact accuracy degree (EAcD) is in (b).

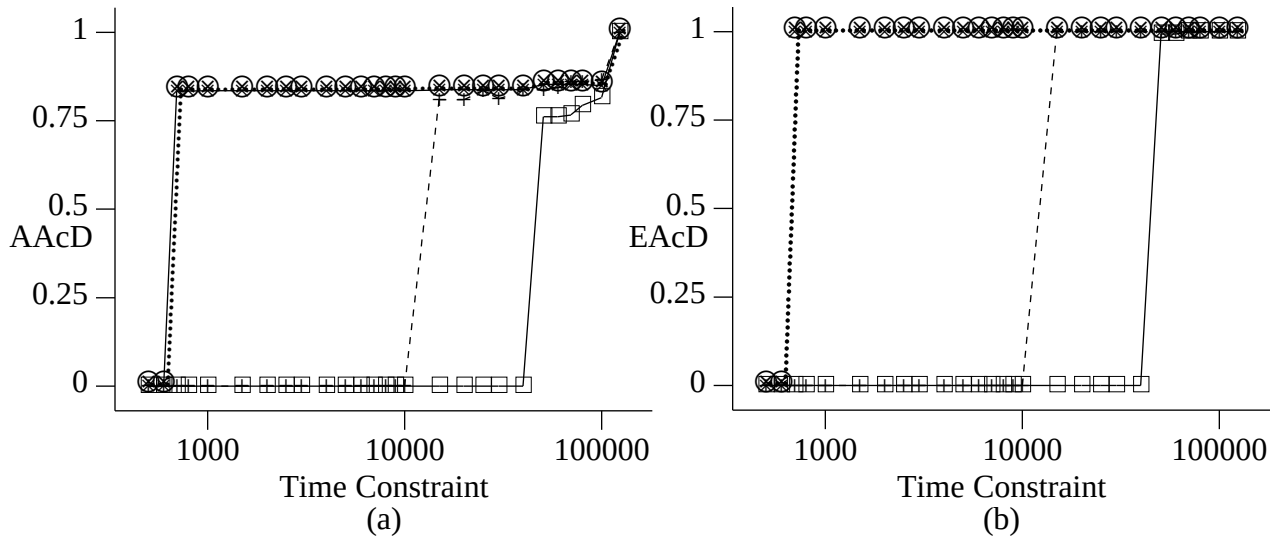


Figure 4.27. Performance profiles of GDFS (‘□’), BS[d;1,10] (‘+’), BS[d;10,10] (‘×’), and BS[d,bwi;10,10] (‘⊗’) for solving instances of TSP(0.1;22,5) for various time constraints, where asymptotic accuracy degree (AAcD) is shown in (a) and exact accuracy degree (EAcD) is in (b).

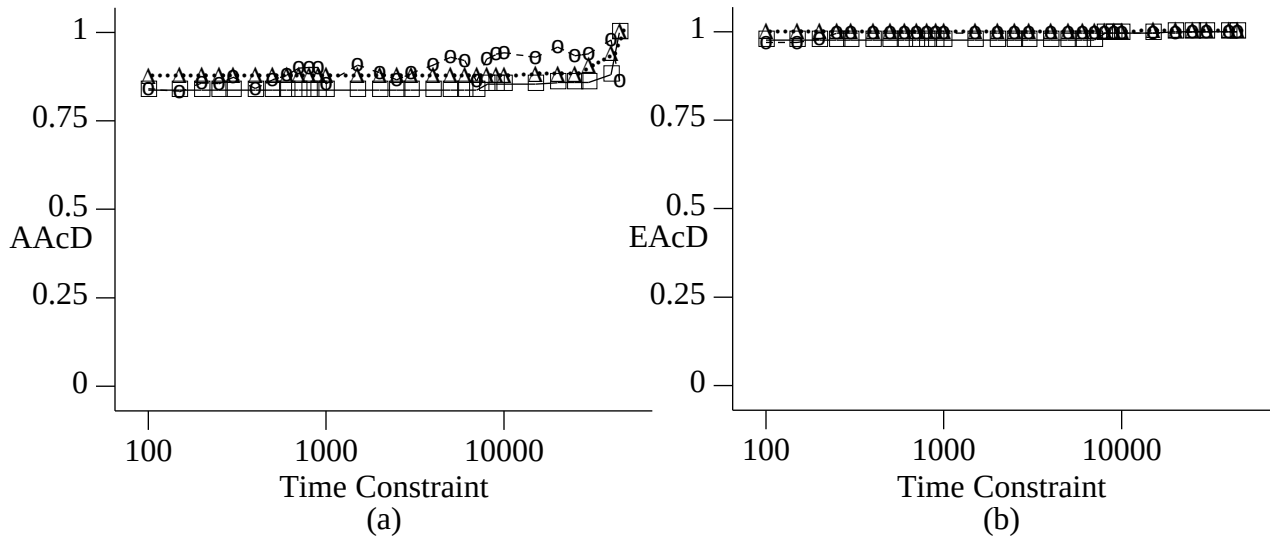


Figure 4.28. Performance profiles of GDFS (‘□’), BS[s,u,ii;2] (‘o’), and BS[s,u;10] (‘Δ’) for solving instances of TSP(0.2;20,25) for various time constraints, where asymptotic accuracy degree (AAcD) is shown in (a) and exact accuracy degree (EAcD) is in (b).

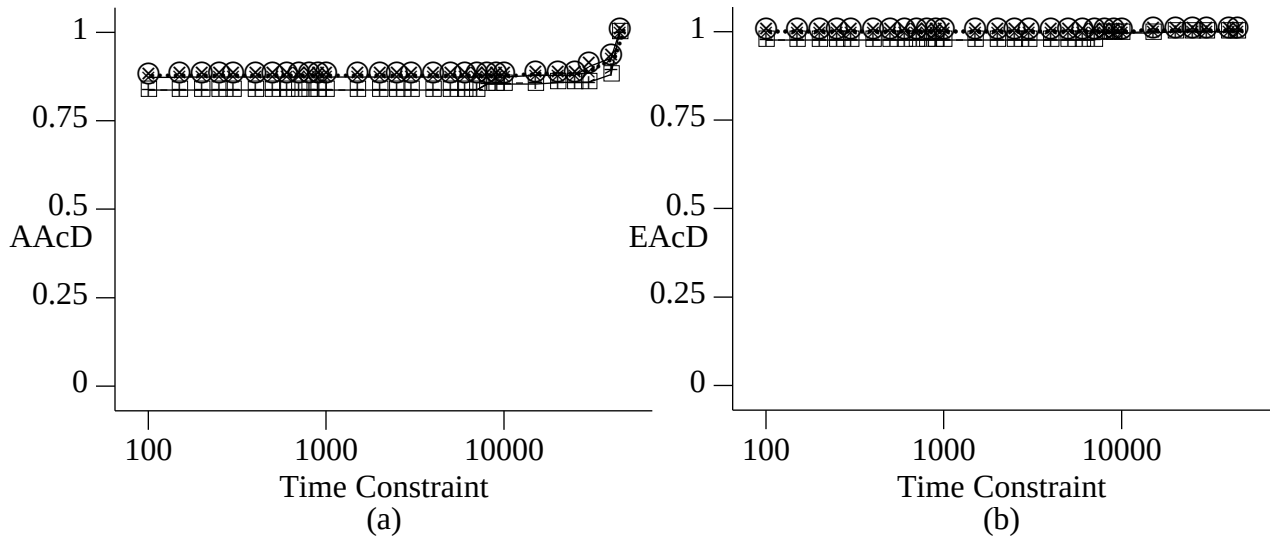


Figure 4.29. Performance profiles of GDFS (‘□’), BS[d;1,10] (‘+’), BS[d;10,10] (‘×’), and BS[d,bwi;10,10] (‘⊗’) for solving instances of TSP(0.2;20,25) for various time constraints, where asymptotic accuracy degree (AAcD) is shown in (a) and exact accuracy degree (EAcD) is in (b).

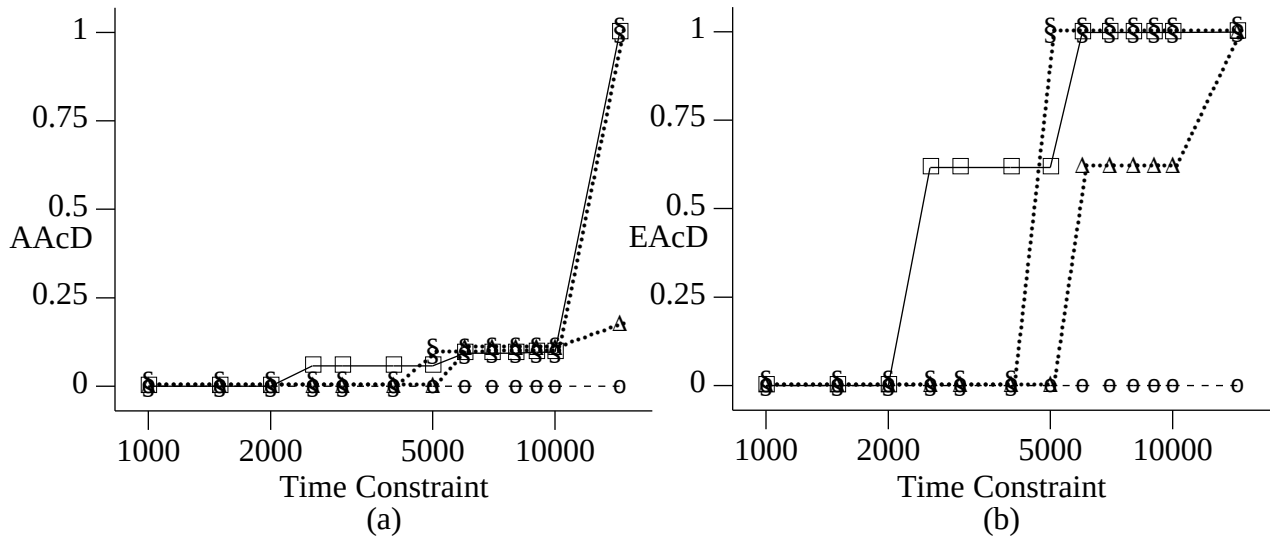


Figure 4.30. Performance profiles of GDFS (‘□’), BS[s,u,ii;2] (‘o’), BS[s,u;10] (‘Δ’), and BS[s,u,li;1,10] (‘§’) for solving instances of Maze(0;100,23) for various time constraints, where asymptotic accuracy degree (AAcD) is shown in (a) and exact accuracy degree (EAcD) is in (b).

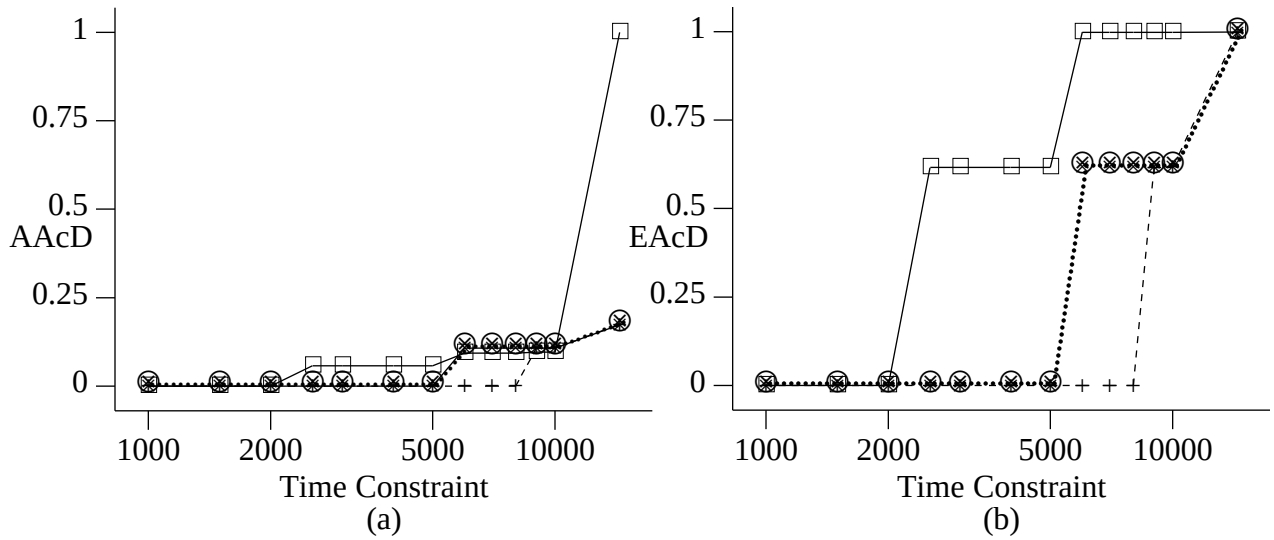


Figure 4.31. Performance profiles of GDFS (‘□’), BS[d;1,10] (‘+’), BS[d;10,10] (‘×’), and BS[d,bwi;10,10] (‘⊗’) for solving instances of Maze(0;100,23) for various time constraints, where asymptotic accuracy degree (AAcD) is shown in (a) and exact accuracy degree (EAcD) is in (b).

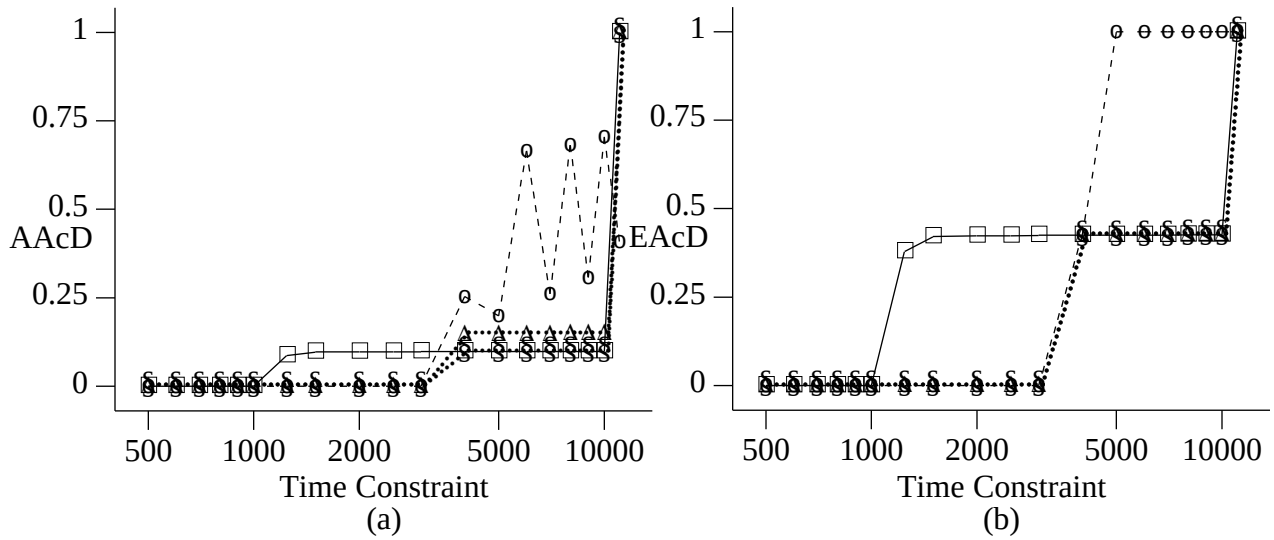


Figure 4.32. Performance profiles of GDFS (‘□’), BS[s,u,ii;2] (‘o’), BS[s,u;10] (‘Δ’), and BS[s,u,li;1,10] (‘§’) for solving instances of Maze(0.1;50,11) for various time constraints, where asymptotic accuracy degree (AAcD) is shown in (a) and exact accuracy degree (EAcD) is in (b).

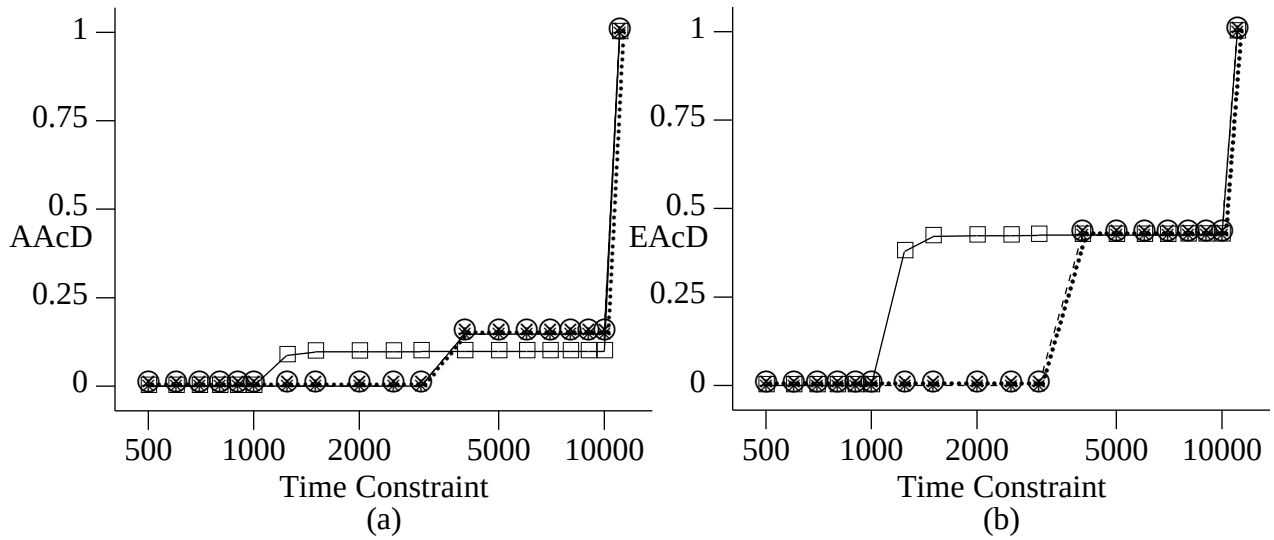


Figure 4.33. Performance profiles of GDFS (‘□’), BS[d;1,10] (‘+’), BS[d;10,10] (‘×’), and BS[d,bwi;10,10] (‘⊗’) for solving instances of Maze(0.1;50,11) for various time constraints, where asymptotic accuracy degree (AAcD) is shown in (a) and exact accuracy degree (EAcD) is in (b).

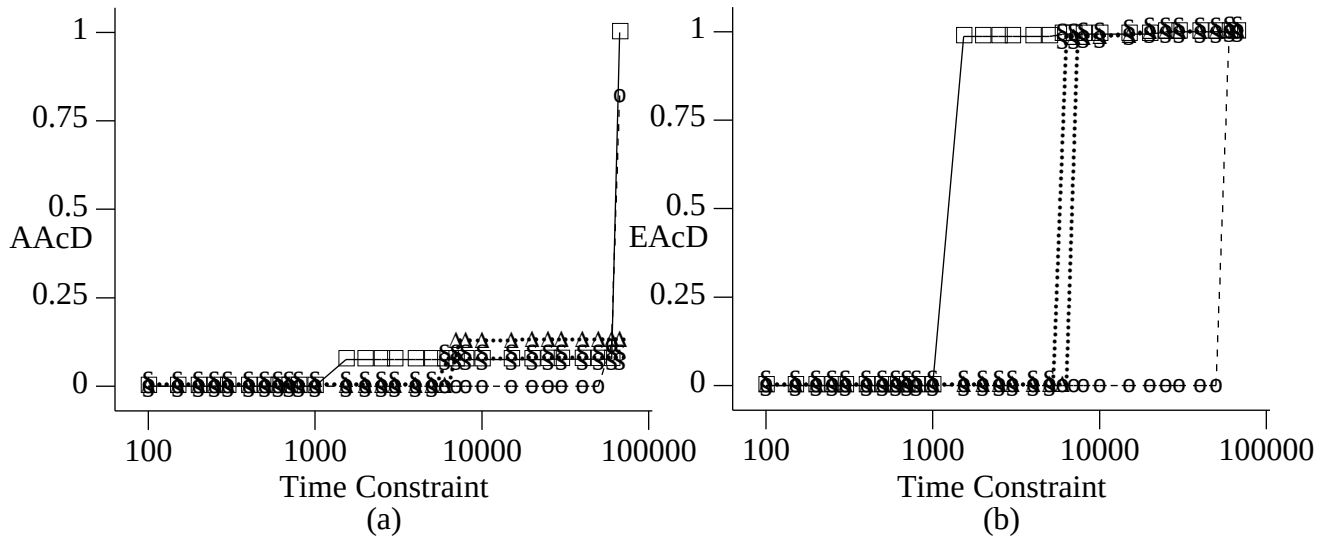


Figure 4.34. Performance profiles of GDFS (‘□’), BS[s,u,ii;2] (‘o’), BS[s,u;10] (‘Δ’), and BS[s,u,li;1,10] (‘§’) for solving instances of Maze(0.2;45,5) for various time constraints, where asymptotic accuracy degree (AAcD) is shown in (a) and exact accuracy degree (EAcD) is in (b).

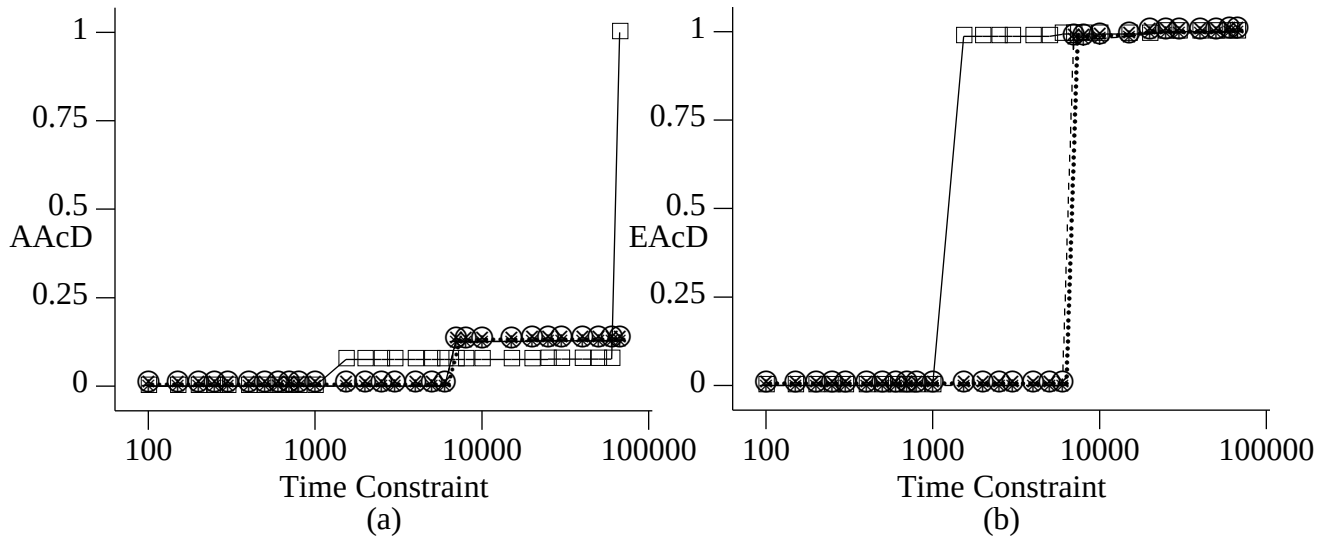


Figure 4.35. Performance profiles of GDFS (‘□’), BS[d;1,10] (‘+’), BS[d;10,10] (‘×’), and BS[d,bwi;10,10] (‘⊗’) for solving instances of Maze(0.2;45,5) for various time constraints, where asymptotic accuracy degree (AAcD) is shown in (a) and exact accuracy degree (EAcD) is in (b).

4.6. Summary

In this chapter, we propose and study a new search algorithm called band search and its behaviors for solving COPs with and without time limits. For solving dense COPs without time limits, we find empirically that band searches with constant band widths generally perform better than those whose band widths vary across levels. We also empirically find that the best choice of band widths is 10 in terms of physical search time. For solving sparse COPs under limited time, we empirically find that BS[d;10,10] and BS[d,bwi;10,10] in general perform very well in solving the evaluated TSPs, and that BS[d;1,10] and GDFS perform well in solving the evaluated maze problems. In particular, GDFS is the best for problems with very sparse solutions and BS[d;1,10] is the best for problems with moderately sparse solutions. For problems in between, the hybrid algorithm GDFS_BS[d;1,10] is the best as seen in Table 4.7.

CHAPTER 5.

AUTOMATED LEARNING OF PARAMETRIC SEARCH HEURISTICS

Machine learning is a general approach to improve heuristics used in problem solvers. For algorithms that solve optimization problems in real time, we classify in Tables 5.1 and 5.2 possible algorithm improvements by machine learning in terms of the density of feasible solutions.

Table 5.1. Search algorithms for solving dense COPs with various objectives and resource constraints, and the applicability of machine learning for algorithm improvement.

Search Objective	Resource Constraints	Machine Learning	Design of Efficient Search Algorithms
Finding first solution	space	no learning	Trivial; use existing algorithms
		learning	Improve existing algorithms in terms of finding better feasible solutions
	space and time	no learning	Trivial, since execution time is predefined
		learning	Improve existing algorithms in terms of finding better feasible solutions
Finding optimal solution	space	no learning	Band search (Chapter 4)
		learning	Improve heuristics used in band search
	space and time	no learning	Real-time search (Chapter 3)
		learning	Improve heuristics used in real-time search

Table 5.2. Search algorithms for solving sparse COPs with various objectives and resource constraints, and the applicability of machine learning for algorithm improvement.

Search Objective	Resource Constraints	Machine Learning	Design of Efficient Search Algorithms
Find first feasible solution or optimal solution	space and time	no learning	MIDA* has better worst-case behavior, while band search is speculative and has better average behavior but poor worst-case behavior
		learning	Reduce time for finding a feasible or optimal solution

In this chapter, we apply genetics-based machine learning methods to learn knowledge-lean search heuristics for problem solvers that can be used for solving a class of problem instances with possibly different characteristics. The search heuristics we are interested can be expressed as parametric functions of problem variables and include decomposition heuristics, selection heuristics, and pruning heuristics.

In this chapter, we first develop a novel learning framework for learning ‘‘good’’ heuristics for solving problems of different characteristics. We then present various strategies for scheduling tests in learning and study different evaluation methods for assessing the performance of heuristics.

In this chapter, we first briefly survey related studies for learning heuristics. We then discuss the characteristics of parametric search heuristics and their performance. This is followed by a description of a learning framework; in particular we present a nonparametric scheduling algorithm for learning and a novel method for generalizing ‘‘good’’ heuristics for

solving search problems of various characteristics. We also describe a learning prototype TEACHER developed in our research group.[1] Finally, we show experimental results on solving application problems by the new search algorithms and on a sequential-circuit testing software tool.

5.1. Survey of Related Studies in Learning Heuristics

The classification of related studies in learning heuristics is shown in Figure 5.1. ‘‘Knowledge lean’’ means that a world model characterizing the relation between heuristics and performance is lacking. In contrast, ‘‘knowledge rich’’ means that such a world model or a good approximate world model exists. Credit assignment is easier for learning knowledge-rich heuristics due to the availability of world models; in contrast, credit assignment is harder for learning knowledge-lean heuristics. In this thesis, the forms of the target parametric heuristics used in search algorithms, such as selection, decomposition and pruning, are all knowledge-lean because there is no good model relating them and the performance of search algorithms. By extensive studies of a particular problem, enough domain knowledge can be gained so that the resulting search algorithms can be called *strong methods*. However, in this research, we focus on learning parametric search heuristics that do not have domain knowledge for performance improvement. (Such heuristics may be called weak methods [48].)

In the following, we concisely survey different learning methods and their perspective domains.

---

1. Most application-independent components of TEACHER were developed earlier by A. Ieumwananonthachai, S. Schwartz, and Professor Wah.

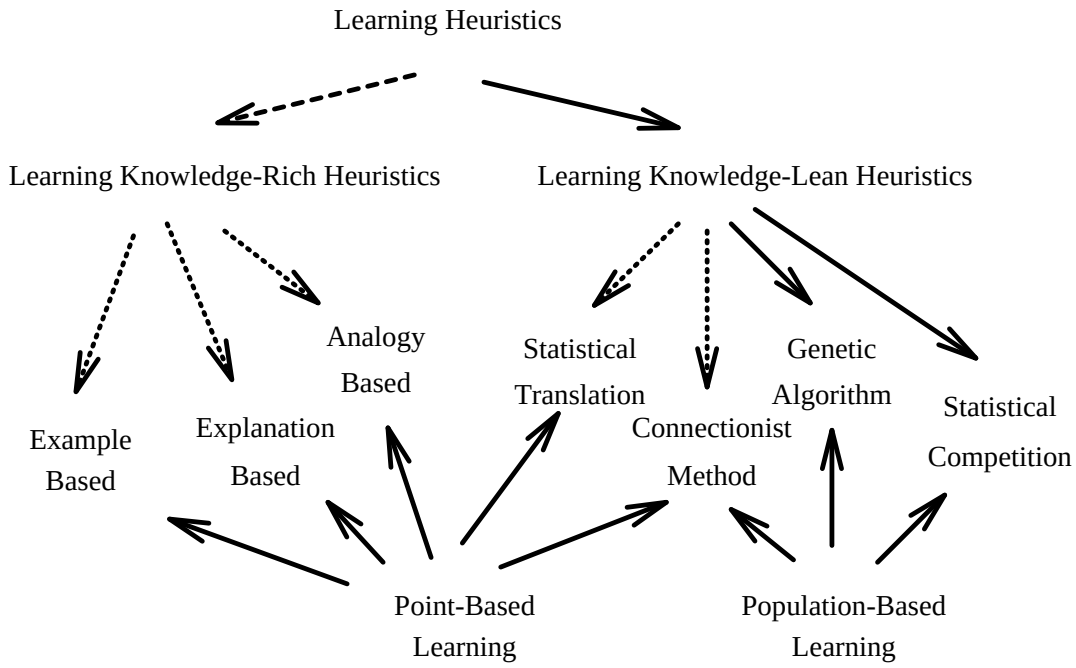


Figure 5.1. Classification of related studies in learning heuristics.

(a) *Learning Knowledge-Rich Heuristics.* Several general methods have been proposed for learning knowledge-rich heuristics. These methods include analytic methods, learning by examples, explanation-based learning, and learning by analogy. Learning by example narrows the scope of possible heuristics by specialization and generalization [48, 67]. Explanation-based learning exercises domain knowledge to explain the relationship between a heuristic and its performance [66, 67]. Learning by analogy translates the knowledge of a problem already solved to a new problem by comparing their problem situations and preconditions. The major problem with learning knowledge-rich heuristics is that extensive domain knowledge must be

available; however, the heuristics to be learned in this research do not have a good world model.

(b) *Learning Knowledge-Lean Heuristics.* Several general methods have been proposed for learning knowledge-lean heuristics. These methods include connectionist methods, genetic algorithms, and statistical methods. Connectionist (or neural-network) methods learn the association between situations and actions in terms of weights of connections [57]. In this research, heuristics have an explicit symbolic representation, rather than an association of numerical values. Genetic algorithms are generate-and-test methods that generate new heuristics to test by applying operators to existing good heuristics. The operators applied include reproduction, crossover, and mutation [21]. The new heuristics are potentially better as they were generated based on good ones. This concept is useful for learning search heuristics in this research. However, the representation of search heuristics may not be the same as in traditional genetic algorithms because search heuristics cannot be encoded as genetic sequences of fixed length. One form of the statistical methods uses statistics to translate a collection of data into separate concepts such that concept learning can be applied [79]. Another form of statistical methods uses statistics to decide which heuristic to test more, given a pool of candidate heuristics [106]. This method is especially useful for learning knowledge-lean heuristics of which the exact performance cannot be determined by a limited number of tests.

(c) *Learning Search Control.* Figure 5.2 shows a classification of methods for learning search heuristics. Methods for learning search control generally fall into one of two categories: *learning while doing* or *learning after doing*. Learning-while-doing

methods exercise learning techniques during a search. In contrast, learning-after-doing methods apply learning techniques only after a search is completed. A learning-while-doing method is usually for learning knowledge-rich heuristics since it requires a good world model for specialization [66] or discrimination [48]. In this case, the search control is represented as rules, and the method learns the conditions of these rules. A learning-after-doing method is usually for learning knowledge-lean heuristics, and it discovers functions that predict the performance of a search, where the discovery can be done statistically [78] or can be done by induction [60]. In this research, we focus on learning simple operators instead of macro operators such as those developed by Iba [28], Korf [38, 39], and Mooney [69], which require extensive domain-specific knowledge.

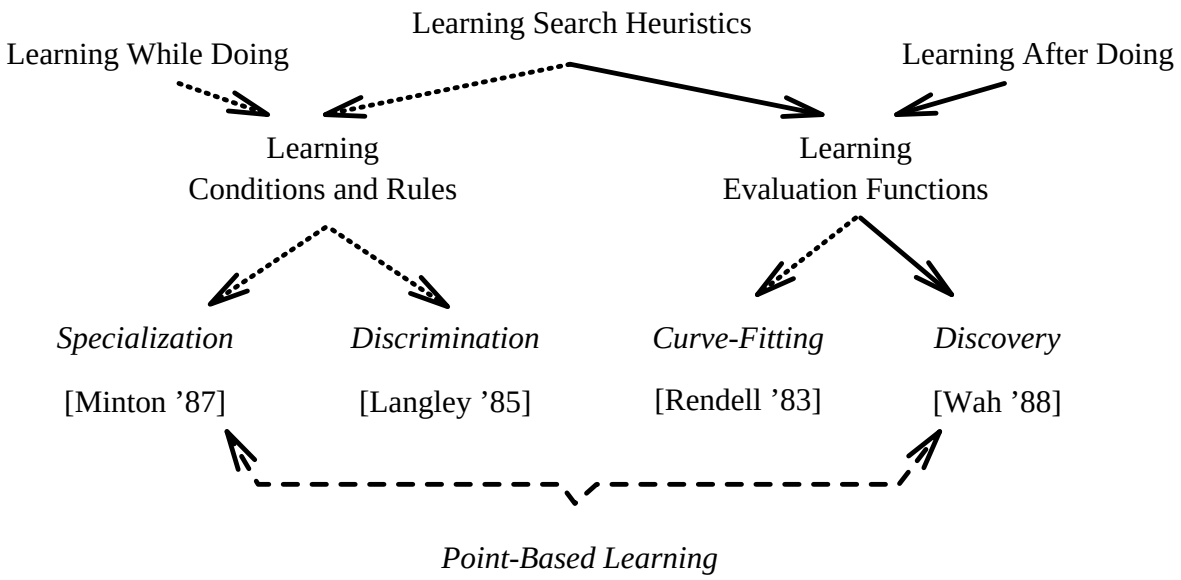


Figure 5.2. Classification of related studies in learning search heuristics.

(d) *Population-Based Learning Paradigm.* In a point-based learning paradigm, an incumbent represents the best heuristic discovered. An operator is applied to this incumbent to generate a number of new ones. The best of these becomes the incumbent. This process is repeated until the termination criteria are met. This learning paradigm works well for learning knowledge-rich heuristics because this paradigm relies greatly on the generator of heuristics. On the other hand, a population-based learning paradigm decides the best heuristic after competitions in a pool of candidate heuristics. During this competition, new heuristics can be added to the pool, and heuristics with poor performance can be removed from the pool. This learning paradigm is useful for learning knowledge-lean heuristics because the performance of a heuristic method cannot be estimated until a reasonable number of tests have been carried out. Note that the methods shown in Figure 5.3 all apply point-based learning. In this research, we apply population-based learning for global adjustments of heuristics and point-based learning for local adjustments.

(e) *Genetics-Based Learning.* Genetics-based machine learning is suitable for learning knowledge-lean heuristics. In this thesis, we use genetics-based learning as the learning method to learn search heuristics. Previous work in this area ignores the costs of tests (or samplings), resulting in a large number of tests. However, in our research, solving a problem instance is costly and a large number of tests is not feasible. In our case, designing a good schedule for drawing samples becomes an important issue. Moreover, an application may have several problem characteristics; for example, a TSP may have fully connected cities or sparsely connected cities. We develop methods that identify conditions under which a specific heuristic method will be most applicable.

5.2. Parametric Search Heuristics

In this section, we first describe the forms and types of heuristics to be learned. We, then, characterize the performance of heuristics.

5.2.1. Form of heuristics

The search heuristic method under discussion is an algebraic expression that can give a value when its parameters are known. A sequence of decisions using the specified heuristic method is made for a given problem instance in order to satisfy the application objective. Heuristics are involved in these decision points because it is difficult to predict the final result before a search is completed due to the well-known horizon effect [108].

In our study, search heuristics are algebraic expressions that must be valid in terms of laws of arithmetic operators and balanced parentheses. In this thesis, we represent heuristics by algebraic expressions in the form of trees where nonterminal nodes are arithmetic operators and terminal nodes are parameters. Two examples are shown in Figure 5.3.

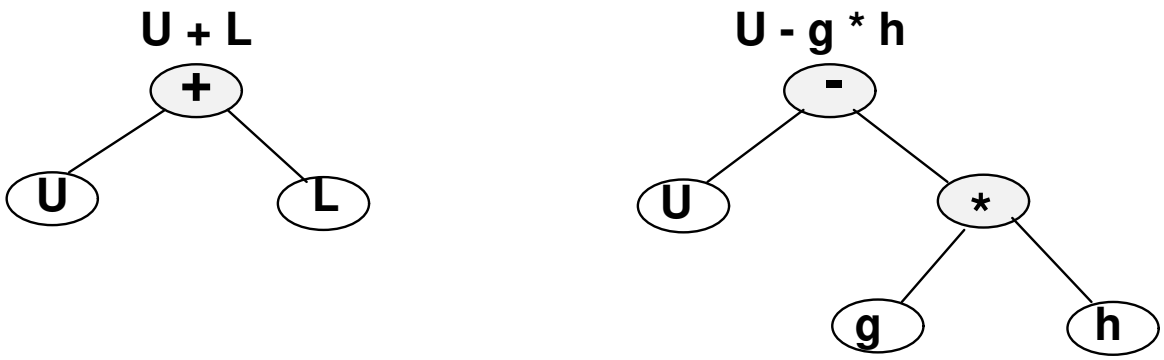


Figure 5.3. Two examples of tree-structured representation of heuristics.

By this tree-structured representation of heuristics, the crossover operator can swap subtrees between two trees. The mutation operator can replace a subtree with a parameter. Examples are shown in Figure 5.4. In this research, we invent a new operator called *coalescing* that joins two trees into a single tree by introducing an arithmetic operator combining these two trees. By this new operator, we can increase the complexity of heuristics.

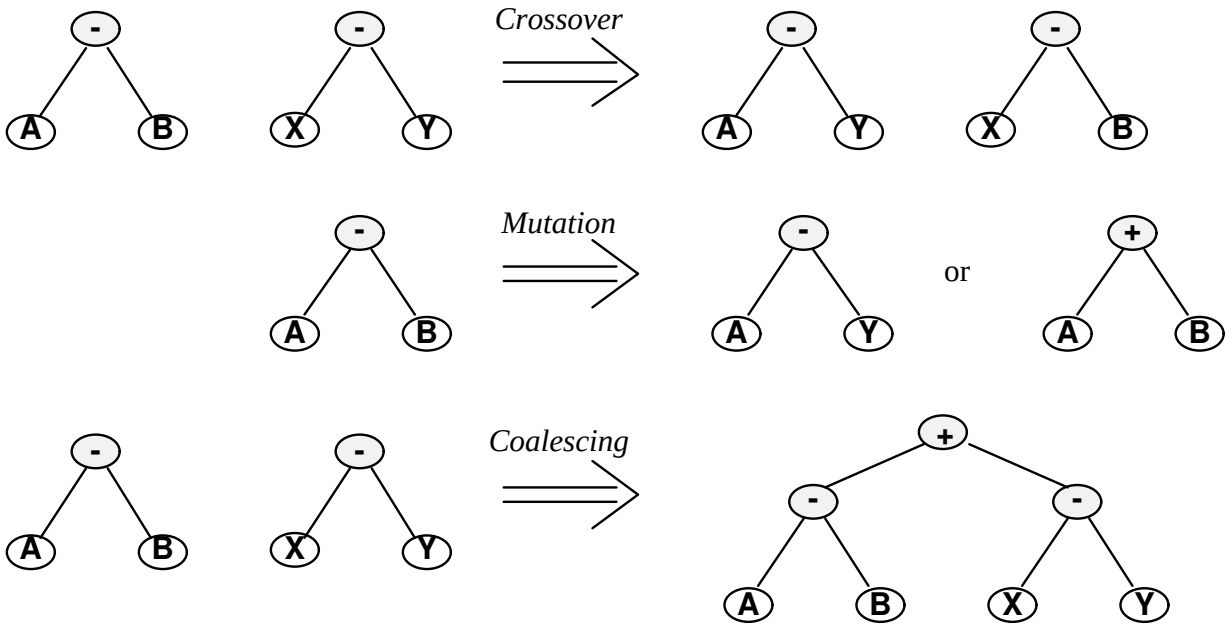


Figure 5.4. Examples of applying crossovers, mutations, and coalescing operators to tree-structured heuristics.

5.2.2. Type of heuristics

A parametric heuristic is an algebraic expression of several parameters or a vector of numeric values. An example of the former one is '$x + y * z$', where x, y, and z include parameters. An example of the latter is (1, 2.05, 3), where values can be integral or real.

Search heuristics studied in this thesis are decomposition heuristics, selection heuristics, and pruning heuristics.

A *decomposition heuristic* studied in this thesis is an algebraic expression of the parameters associated with search nodes and the search state. These parameters are greatly dependent on the problem instance. When a search node is to be expanded, the decomposition heuristic gives a value for each possible way of decomposition; the problem solver then chooses the best way of decomposition based on these values and expands the search node.

A *selection heuristic* studied in this thesis is an algebraic expression of the parameters associated with search nodes and the search state. These parameters include, but are not limited to, the lower bound (l) and the upper bound (u) of a search node, the partial solution value (g) from the root node to this search node, and the estimated partial solution value (h) from this node to an optimal-solution node in the subtree rooted at this search node. A possible algebraic expression is '$l+u$' which measures the average between the lower and upper bounds. During the search, when a search node is generated, this algebraic expression gives a value based on the parameters, and the problem solver selects the best search node for expansion from all search nodes that are candidates for selection.

A *pruning heuristic* studied in this thesis is an algebraic expression of the parameters associated with search nodes and the search state. The pruning heuristic gives a value to each active search node. During pruning, the problem solver prunes all search nodes whose pruning values are no less than (respectively, no greater than) the incumbent for a minimization (respectively, maximization) search.

A search heuristic mainly contains a knowledge-rich part and a knowledge-lean part. As an example consider a selection heuristic $l + u$. Methods for computing the lower bound l and the upper bound u are derived based on extensive domain knowledge; hence, they are considered knowledge-rich. However, the algebraic form '$l + u$' is knowledge-lean because the relationship between this form and its performance cannot be characterized based on known domain knowledge. In this thesis, we are interested in developing heuristics related to the composition of parts either known to be a problem solver or obtained from domain knowledge.

5.2.3. Performance of heuristics

The performance of a heuristic can be measured in many ways but mainly depends on the *quality* of the solution found due to a heuristic and the *cost* of finding this solution.

For solving optimization search without a time limit, the quality is not an issue because the final solution delivered is the optimal solution. In this case, only the cost is related to the performance. In this thesis, the cost we are interested is the execution time, and the performance of a heuristic is its *speedup* over that of the heuristic commonly used. For brevity, we use the term *baseline heuristic* to refer to the heuristic commonly used. We choose speedup as a measure because (a) it is commonly used for comparing two search algorithms; and (b) the raw execution time (or the number of search nodes expanded or generated) may vary over several orders of magnitude while solving different problem instances, and the raw execution times cannot be appropriately averaged or combined.

For time-constrained optimization, the execution time is given and fixed, and quality is the only issue. Ideally, we should learn heuristics for each possible time constraint.

However, the number of possible time constraints is so huge that it forbids such an exhaustive approach. Therefore, we adopt an alternative strategy that has two phases: (a) learning a good heuristic for finding the first solution, and (b) learning a good heuristic for finding the optimal solution. Finding the first solution is an important issue for many time-constrained searches because finding the first solution sometimes is almost as hard as finding the optimal solution. Finding the optimal solution is also important for time-constrained optimization because it indicates the progress of refining the incumbent. For both phases, the goodness criterion is the speed of finding the first solution and the optimal solution; therefore, the performance of a heuristic is the speedup over that of the baseline heuristic. Note that solving the optimal solution is to find the optimal solution and prove its optimality; in contrast, finding the first solution is to simply generate any feasible solution. In general, finding the optimal solution is much harder than finding a feasible solution.

The performance of a heuristic is not exact in the sense that it can perform well on some problem instances but perform poorly on some other problem instances. Statistically, the performance of a heuristic can be characterized statistically by a distribution that may not be known. Therefore, a heuristic must be tested a number of times to find statistical measures such as mean and variance of its performance distribution.

When a heuristic is evaluated across a number of problem instances, the average of speedups can vary up and down. We can plot the profile of the speedup averages against the number of tests. Whenever the profile plots of two heuristics cross each other, inconsistency in predicting which one is better may happen. The situation is even worse when the number of tests on these two heuristics is not equal. Even if we choose another aggregate performance measure instead of the average, such inconsistencies (or anomalies) may

happen. We call this phenomenon as the *inconsistency of aggregate performance*. To cope with this inconsistency when we need to select heuristics at the end of learning, we introduce *full* evaluation on all promising heuristics in order to allow all heuristics under comparison to be evaluated to have more statistically valid performance values.

Another inconsistency in performance evaluation may happen when we compare performance of heuristics across problem instances of different characteristics. For example, the cities in the traveling salesperson problem (TSP) can be fully connected, densely connected, or sparsely connected. A good heuristic for solving a TSP instance with fully connected cities may not perform well for solving an instance with sparsely connected cities. That is, the performance of a heuristic may be inconsistent across problem instances of different characteristics. We call this the *inconsistency of domain performance*.

A *subdomain* of problem instances is a collection of problem instances that have similar characteristics in terms of both search parameters and search performance. However, the boundary between the two subdomains may not be so clear that it can be identified easily. Moreover, it is intractable to learn a spectrum of heuristics for all different (possibly an infinite number of) subdomains. We, therefore, select a collection of subdomains of the problem, learn good heuristics that work well on these subdomains, and validate heuristics learned on subdomains not selected for learning. For example, subdomains for the vertex-cover problem can be classified based on the connectivity of the graph in the problem. Similarly, subdomains for the traveling salesman problem can be classified based on the connectivity and distribution of cities in the problem.

5.3. Learning Framework

The learning process consists of two phases: *learning phase* and *verification phase*. Learning works by alternating between these two phases, beginning with a learning phase and ending with a verification phase, as shown in Figure 5.5.

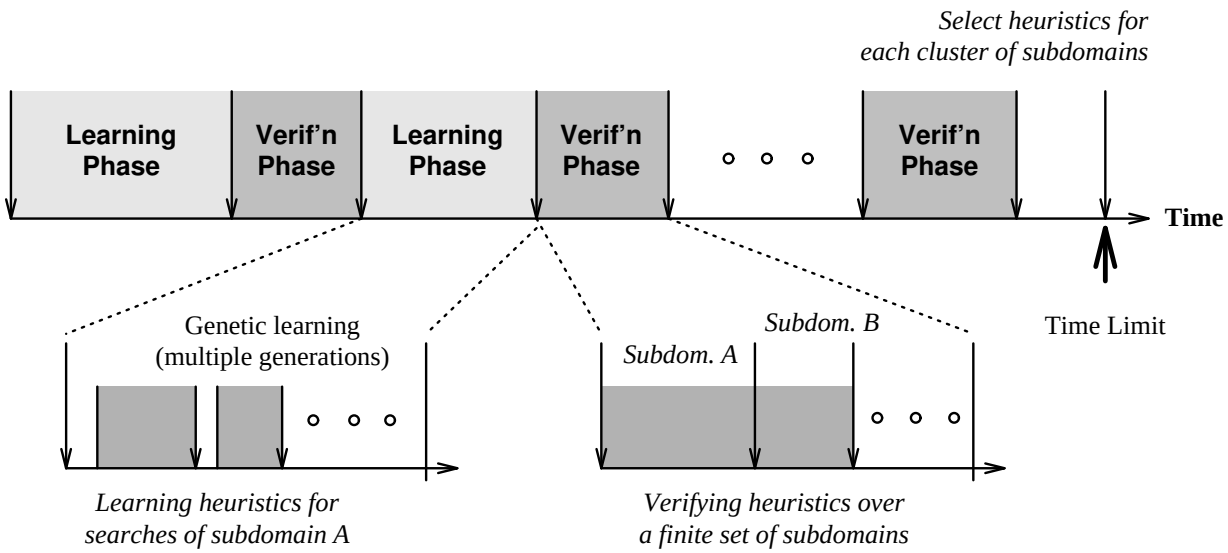


Figure 5.5. The learning process consists of a sequence of alternating learning and verification phases, where genetic programming is exercised in a learning phase and full evaluation is exercised over a set of subdomains in a verification phase. (In the last verification phase, domain clustering is performed and a heuristic is proposed for each cluster of subdomains.)[2]

During a learning phase, we exercise genetic programming, which is a genetics-based learning algorithm that generates and tests knowledge-lean heuristics. A genetic algorithm contains multiple generations. In each generation, we generate and test a finite set of

2. This diagram is modified from an original version due to A. Ieumwananonthachai and Professor Wah.

heuristics. At the end of a generation, we associate heuristics in this generation with their respective fitness values. Then we select heuristics randomly with a probability directly proportional to its fitness value. Finally, we retain the selected heuristics in the next generation and use them to generate by genetic operators new heuristics for the next generation.

During a verification phase, we fully evaluate promising heuristics on subdomains whose heuristics have been learned and tested on (these subdomains are called *verified subdomains*). At the end of a verification phase, each verified subdomain is associated with a sequence of heuristics sorted in descending order based on their performance values. Then the top heuristics from all verified subdomains are selected and used in the first generation of the following learning phase.

In the last verification phase, we fully test the heuristics over all specified subdomains. We do not have to select heuristics for the next learning phase since this is the last phase in the entire process. We, instead, conduct *domain clustering*. In domain clustering, heuristics are ranked on each verified subdomain. We employ an affinity measure to see whether two subdomains can be combined (or clustered). A set of subdomains can be clustered if their pair-wise affinity measures satisfy a given criterion. We then select and deliver the very top heuristic for each subdomain or each cluster of subdomains.

There are several important issues in the learning and verification phases that we study in this thesis. In the learning phase, three important issues arise:

(1) *Heuristics Generation.* In order to find good heuristics, we must be able to generate effectively good heuristics. However, the heuristics studied in this thesis are knowledge

lean, namely, we do not have effective domain-specific information about relating heuristics to their performance. Therefore, we adopt genetic programming as our underlying learning algorithm, which only exercises weak operators, such as crossover and mutation, in our heuristic generator to generate new heuristics.

(2) *Performance Evaluation.* In order to generate new heuristics from promising ones, we have to find good heuristics during genetic programming. The performance of heuristics is in terms of speedup; therefore, we use the average speedup as an aggregate performance for heuristics, though we notice that such a measure may have performance that is inconsistent with other performance measures. Also, predicting the aggregate performance of a heuristic based on unequal numbers of tests can incur inconsistency regardless of the aggregate performance measure used. Another reason we use the average speedup is that it is suitable for computing a fitness value that is used as a performance measure in genetics-based learning methods, in which the performance measure used must be able to (a) sort all heuristics and (b) assign to heuristics values that have a linear relation to performance values of these heuristics.

(3) *Test Scheduling.* In order to efficiently use limited time to test heuristics, we have to have a good strategy of scheduling tests. We empirically found that the performance of heuristics does not comply with any well-known distribution; therefore, we need to use a nonparametric strategy for scheduling tests.

In a verification phase, we fully evaluate heuristics on subdomains; therefore, test scheduling is not an issue in verification. However, there still exist two important issues:

(1) *Generalization of Heuristics.* There are a significant number of possible subdomains and a tremendous amount of time needed to learn a good heuristic for each subdomain. Therefore, we learn heuristics for a limited number of subdomains and generalize them to the other subdomains. Also, these subdomains are manually classified and may overlap with each other; therefore, we can combine (or cluster) some subdomains into a single one based on the performance of heuristics tested on these subdomains.

(2) *Performance Evaluation.* After combining the subdomains into a single one, we have to select a good heuristic that works well over a cluster of subdomains. The performance of a heuristic on a problem instance is defined as the speedup with respect to that of the baseline heuristic. The performance distribution of a search heuristic studied, as shown in Figure 5.6, does not comply with any well-known distribution. Moreover, the distribution of performance across different subdomains may not have the same range nor the same shape as seen in Figure 5.6. Improper aggregate performance across different subdomains may incur inconsistency of domain performance as discussed earlier. The method used should avoid the inconsistency of domain performance.

The input to the learning system includes the following information: (a) a problem solver using a heuristic supplied by a learning system, (b) a baseline heuristic that is commonly used in the problem solver, (c) a finite set of subdomains manually classified, (d) the *test universe* that contains a finite set of problem instances for each manually classified subdomain, and (e) a finite set of learning parameters. The learning parameters required include the number of learning phases (i.e., the number of verification phases), the number of generations per learning phase, the scheduling strategy used in the learning phase, the

generation duration, the generation size, and the percentage of heuristics retained from the previous generation in the generation size.

The output of learning includes the clusters of subdomains and a good heuristic for each cluster of subdomains.

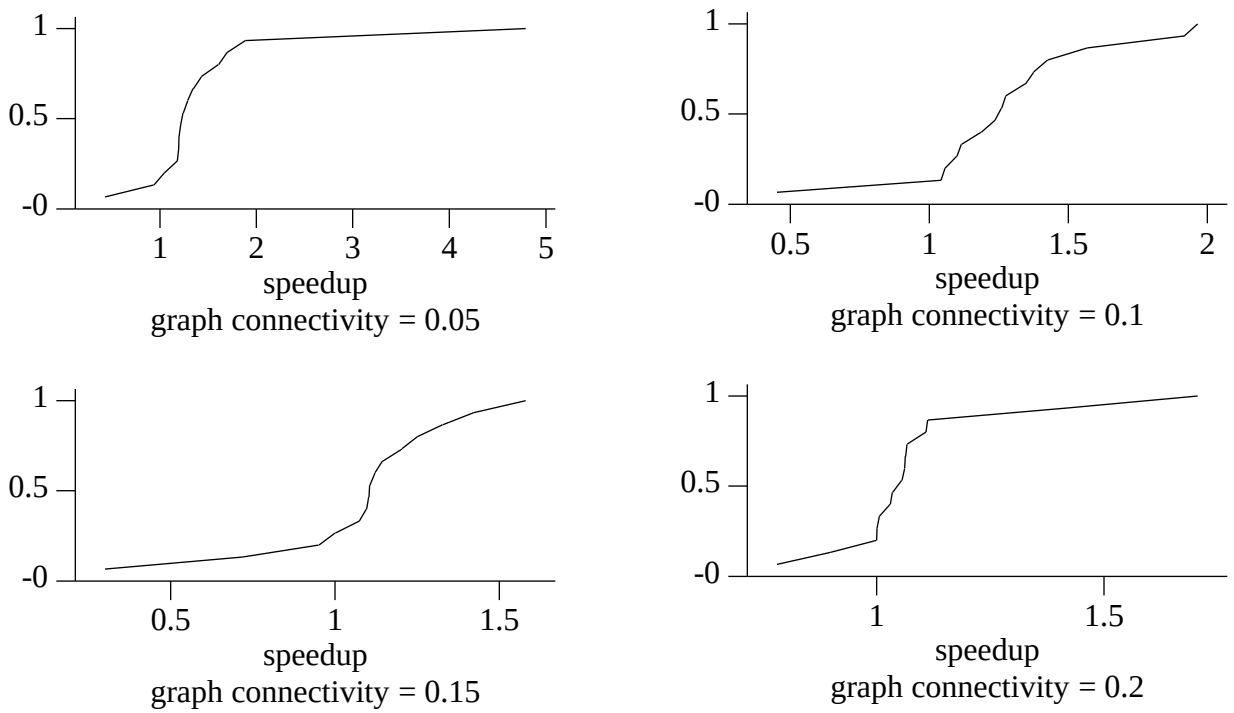


Figure 5.6. Approximate cumulative distribution functions of the performance of a decomposition heuristic used in a GDFS for solving fifteen random instances of each of the VC problems with various graph connectivities, where the heuristic used is learned and its ID is 778 (see Appendix B).

5.4. Learning Phase

During the learning phase, we use genetic programming to learn search heuristics. In this section, we address performance evaluation and scheduling issues in learning.

5.4.1. Performance evaluation in learning

The scope of performance evaluation in a learning phase is within a generation, where a fixed number of heuristics are active and their respective numbers of tests may be different. The scheduling algorithm in learning is incremental, and the performance evaluation method must be able to evaluate incremental performance data.

There are five requirements of the performance evaluation method used in the learning phase: (1) the measure must be able to sort all candidate heuristics, (2) the measure must allow fitness values to be computed that can be translated to performance values of heuristics, (3) the measure can be evaluated incrementally, (4) the measure must not depend on a large number of samples, and (5) the measure can lead to improved speedup. In this section, we study three methods for performance evaluation: the pure average (PA) method, the equal-weight normalization (EWN) method, and the probability of win (PW) method.

To facilitate the description, we assume the following situation at the time of performance evaluation. Assume there are m heuristics. Let $x_{i,j}$ denote the performance value (i.e., speedup, in this thesis) of heuristic i on problem instance j, and $\overline{x}_{i,j} = del\ \frac{1}{j}\sum_{k=1}^{j} x_{i,k}$. Assume heuristic i has n_i tests at the present time, then we have $\overline{x}_i = \overline{x}_{i,n_i}$.

5.4.1.1. The pure average method

The performance of heuristic i is simply the average of the performance value, (i.e., speedup) of solving each of the problem instances it has been tested on so far, i.e., $\overline{x}_i$. The advantage of this method is that it is consistent with the objective of learning, namely, the

performance of the heuristic. The major disadvantage is that large speedups carry more weight than slowdowns such that the average speedup of a heuristic looks better than the baseline heuristic when it has a few large speedups and many slowdowns. For example, consider a heuristic with speedups (or slowdowns) 10, 0.1, 0.2, 0.25, and 0.4, then its average speedup is 2.19, which implies that this heuristic is better than the baseline heuristic. However, if we view this heuristic as the reference, then the speedups (or slowdowns) of the baseline heuristic are 0.1, 10, 5, 4, and 2.5 and its average is 4.32, which implies that the baseline heuristic is better than this heuristic. Therefore, the decision of the better heuristic depends on the reference used. We call this phenomenon the *speedup-slowdown anomaly*. This anomaly has been noticed in the literature [9, 104].

As another example, consider two heuristics *i* and *j*. Assume the speedups of heuristic *i* are 30, 2, 2, and 2; the speedups of heuristic *j*, 4, 5, 6, and 5. The average speedups of heuristics *i* and *j* are 9 and 5, respectively, which implies that heuristic *i* is better than heuristic *j*. However, this conclusion is misleading because heuristic *i* is better only because it has *one* large speedup. In contrast, heuristic *j* consistently has moderate speedups. Therefore, for a random instance to be solved, heuristic *j* is better than heuristic *i*. This phenomenon is called the *overweight anomaly*. Note that this example illustrates the overweight anomaly, though we are aware that the anomalous situation may be solved by detecting and removing outliers (i.e., 20, the first speedup of heuristic *i*). However, the overweight anomaly may not be completely avoided by handling outliers because this anomaly can happen when limited samples are drawn from a distribution of performance values with two unbalanced modes, one of which is relatively larger than the other. This

anomaly has also been noticed in the literature [9, 104].

5.4.1.2. The equal-weight normalization method

The EWN method removes speedup-slowdown anomalies. Let $t_{i,j}$ denote the execution time of using heuristic *i* to solve problem instance *j*. For brevity, we denote the baseline heuristic as heuristic *b*. The equal-weight speedup of heuristic *i* on test *j* is defined as $x_{i,j}^{ewn} = del\ \frac{t_{b,j}}{t_{b,j}+t_{i,j}}$. Note that $0 \le x_{i,j}^{ewn} \le 1$. When $x_{i,j}^{ewn} > 0.5$, it is a speedup. When $x_{i,j}^{ewn} < 0.5$, it is a slowdown. All performance values are between 0 and 1, which removes the speedup-slowdown anomaly. Moreover, the EWN de-emphasizes large speedups because the transformations are nonlinear due to $x_{i,j}^{ewn} = \frac{x_{i,j}}{1+x_{i,j}}$ and $x_{i,j} = \frac{x_{i,j}^{ewn}}{1-x_{i,j}^{ewn}}$; therefore, the overweight anomaly is also alleviated because all the performance values are between 0 and 1.

The EWN method is especially good for pair-wise comparison. Consider heuristics *i* and *b*. Let $x_{i,j}^{ewn} = del\ \frac{t_{b,j}}{t_{b,j}+t_{i,j}}$ and $x_{b,j}^{ewn} = del\ \frac{t_{i,j}}{t_{b,j}+t_{i,j}}$. Then, we have $x_{i,j}^{ewn} + x_{b,j}^{ewn} = 1$ for every *j*. Also, we have $\bar{x}_{i,j}^{ewn} + \bar{x}_{b,j}^{ewn} = 1$ for each possible *j*, which removes the overweight and speedup-slowdown anomalies as well as removes the bias against the reference. The equality $\bar{x}_{i,j}^{ewn} + \bar{x}_{b,j}^{ewn} = 1$ is true because

$$\bar{x}_{i,j}^{ewn} + \bar{x}_{b,j}^{ewn} = \frac{1}{j}\sum_{k=1}^{j} x_{i,j}^{ewn} + \frac{1}{j}\sum_{k=1}^{j} x_{b,j}^{ewn} = \frac{1}{j}\sum_{k=1}^{j}\left(\frac{t_{b,j}}{t_{b,j}+t_{i,j}} + \frac{t_{i,j}}{t_{b,j}+t_{i,j}}\right) = \frac{1}{j}\sum_{k=1}^{j} 1 = 1 .$$

A major drawback of the EWN method is that it eliminates slowdowns by amplifying the weights of slowdowns. On the other hand, it does not try to improve speedup since less weights are put on speedups. Consequently, the EWN method is not useful for performance

evaluation in the learning phase because the method used cannot lead to improve speedup, as is the fifth requirement described in Section 5.4.1, namely, the measure can lead to improved speedup.

5.4.1.3. The probability of win method

According to the central limit theorem, the average ($\hat{\mu}$) of performance values behaves approximately as a normal distribution for a large number ($\hat{n}$) of samples given its mean (μ) and variance (σ^2). That is, $\frac{\hat{\mu}-\mu}{\sigma/\sqrt{n}}$ *wig* $N(0,1)$. In practice, we do not know the exact mean μ and variance σ, but only know the sampled mean and variance. Also, the number of samples may not be sufficiently large to apply the central limit theorem.

In this thesis, we are interested in estimating the mean μ based on the average $\hat{\mu}$ and the empirical variance $\hat{\sigma}$; in particular, we are interested in knowing the probability that a mean is better than another one. Consider two heuristics *i* and *j*. We are interested in knowing the probability $Pr[\,\mu_i > \mu_j \mid \hat{\mu}_i, \hat{\sigma}_i, n_i, \hat{\mu}_j, \hat{\sigma}_j, n_j\,]$. We refer to this probability as the *probability of win*, i.e., the probability that heuristic *i* wins over heuristic *j*.

Statistically, the mean μ behaves as the *t* distribution with $n-1$ degrees of freedom given $\hat{\mu}$, $\hat{\sigma}$, and *n*. However, it is difficult to derive the probability of win based on the *t* distribution. Therefore, in this thesis we compute the probability *asymptotically*, which implies that the probability becomes accurate as the number of samples is very large.

Consider heuristic *i*. Assume that we know σ_i. Define a random variable X_i as

$$X_i \;=del\; \frac{\hat{\mu}_i - \mu_i}{\sigma_i / \sqrt{n_i}}\,.$$

Note that X_i behaves as N(0,1). If we consider μ_i as a random variable, then we have

$$\mu_i = \hat{\mu}_i - \frac{X_i \sigma_i}{\sqrt{n_i}} .$$

Note that μ_i behaves as $N\left(\hat{\mu}_i, \frac{\sigma_i^2}{n_i}\right)$. Consider two heuristics *i* and *j*. The difference $\Delta\mu_{i,j}$ of performance means of heuristics *i* and *j* can be written as

$$\Delta\mu_{i,j} = del \ \mu_i - \mu_j = \hat{\mu}_i - \hat{\mu}_j - \frac{X_i \sigma_i}{\sqrt{n_i}} + \frac{X_j \sigma_j}{\sqrt{n_j}} .$$

Note that $\Delta\mu_{i,j}$ behaves as $N\left(\hat{\mu}_i - \hat{\mu}_j, \frac{\sigma_i^2}{n_i} + \frac{\sigma_j^2}{n_j}\right)$. The probability that heuristic *i* wins over heuristic *j* becomes

$$Pr[\mu_i > \mu_j \mid \hat{\mu}_i, \hat{\sigma}_i, n_i, \hat{\mu}_j, \hat{\sigma}_j, n_j] = Pr[\Delta\mu_{i,j} > 0 \mid \hat{\mu}_i, \hat{\sigma}_i, n_i, \hat{\mu}_j, \hat{\sigma}_j, n_j] .$$

We can compute the probability based on the fact that $\Delta\mu_{i,j}$ behaves as $N\left(\hat{\mu}_i - \hat{\mu}_j, \frac{\sigma_i^2}{n_i} + \frac{\sigma_j^2}{n_j}\right)$. During this computation, we replace σ_i and σ_j with $\hat{\sigma}_i$ and $\hat{\sigma}_j$, respectively. This replacement is asymptotically correct because $\hat{\sigma}_i$ converges to σ_i with probability one and so does $\hat{\sigma}_j$ to σ_j when the number of samples drawn is large.

In this thesis, we assume all heuristics are independent and that all test cases equally likely to occur. Based on the assumptions, the probability of win for a heuristic can be defined as the average of all probabilities of win for this heuristic over all other heuristics. The major advantage of the PW method is that it takes, besides the average, the number of samples and the estimated variance into account while comparing heuristics. Its major drawback is that the method needs a large number of tests, which is impractical in the learning phase due to time limitation, as is the fourth requirement described in Section 5.4.1

(namely, the measure must not depend on a large number of samples).

5.4.1.4. Summary

In this thesis, we choose the PA method as the method for performance evaluation because (a) the EWN method eliminates slowdowns but does not try to improve speedup, (b) the PW method requires a large number of tests that are impractical in the learning phase, and (c) the PA method is the most popular and widely accepted method for performance evaluation among all these three methods; we are fully aware that the PA method is prone to anomalies with respect to other evaluation methods. To assure the correctness of speedups, the sequence of problem instances used for testing heuristics is identical.

5.4.2. Nonparametric scheduling of experiments in genetics-based learning

In a traditional genetic algorithm, the generation size and the generation duration are fixed and determined ahead of time. Moreover, all heuristics are tested in a *round-robin* (RR) fashion; as a result, the number of tests per heuristic are fixed.

In this thesis, we propose a new test scheduling strategy that schedules the heuristics to be tested based on run-time performance information of heuristics.

Previous work [1, 31, 32] on test scheduling assumes the knowledge of performance distributions of all heuristics. In this thesis, the assumption of performance distributions is *not* required, and nonparametric statistical techniques are used, since we empirically found that the performance distributions of search heuristics are hard to parameterize, as are seen in Figure 5.6. This nonparametric scheduler is inspired by the previous work by Aizawa and Wah [1].

The objective of scheduling is to *minimize under limited time the risk that the heuristics selected to generate new ones in the next generation are not the best*. In this thesis, we interpret time in terms of its logical units that are measured in terms of the number of tests performed.

Consider a generation of m heuristics among which m' heuristics are selected to generate new ones used in the next generation at the end of a generation. Let w denote the worst heuristic that serves as a base for calculating fitness values of other heuristics (all fitness values are nonnegative). Heuristic i is characterized by information such as n_i, the number of tests performed, mean μ_i (unknown), variance σ_i (unknown), estimated mean $\hat{\mu}_i$, estimated variance $\hat{\sigma}_i$, fitness value F_i, and estimated fitness value f_i, where $F_i =del\ \mu_i - \mu_w$ and $f_i =del\ \hat{\mu}_i - \hat{\mu}_w$. Note that f_i is an unbiased estimator of F_i since $\hat{\mu}_i$ and $\hat{\mu}_w$, respectively, are unbiased estimators of μ_i and μ_w. The loss L_i of the belief that f_i is accurate can be defined as

$$L_i =del\ \mathrm{E}[(f_i - F_i)^2]$$

for $i \neq w$. For heuristic w, its fitness value is zero according to its definition; therefore, the loss is equal to the loss of the belief that its estimated mean is accurate; namely,

$$L_w = \mathrm{E}[(\hat{\mu}_w - \mu_w)^2] .$$

We will describe how to compute L_i and L_w later. The probability that heuristic i will be selected to generate new ones in genetic programming is defined as

$$P_i =del\ \frac{f_i}{\sum_{j=1}^{m} f_j} .$$

The objective of test scheduling can be formulated as

$$\Psi = \sum_{i=1}^{m} P_i \, L_i$$

under a constraint $\sum_{i=1}^{m} n_i = t$, where t is the current number of tests that has been performed in this generation. Note that the goal is to minimize Ψ. By applying a Lagrangian multiplier, we have

$$\Psi' = \Psi + \lambda \left(\sum_{i=1}^{m} n_i - t \right).$$

By equating $\frac{\partial \Psi'}{\partial n_i}$ to zero, we have the optimality criterion as

$$-\lambda = \left(\frac{\partial P_i}{\partial n_i} \right) L_i + P_i \left(\frac{\partial L_i}{\partial n_i} \right) = \left(\frac{\partial P_j}{\partial n_j} \right) L_j + P_j \left(\frac{\partial L_j}{\partial n_j} \right) \tag{5.1}$$

for $i \neq j$. Hence, test scheduling is to maintain the optimality criterion. Note that the Lagrangian procedure is valid only in the continuous domain, but not in the discrete domain. In our case, the target domain is discrete, and this procedure incurs approximation. The partial derivative of P_i can be extended as

$$\begin{aligned} \frac{\partial P_i}{\partial n_i} &= \frac{\partial}{\partial n_i} \left(\frac{f_i}{f_i + \sum_{j \neq i} f_j} \right) \\ &= \left(\frac{\partial f_i}{\partial n_i} \right) \left(f_i + \sum_{j \neq i} f_j \right)^{-1} - f_i \left(f_i + \sum_{j \neq i} f_j \right)^{-2} \left(\frac{\partial f_i}{\partial n_i} \right) \\ &= \left(\frac{\partial f_i}{\partial n_i} \right) \left(\frac{1 - P_i}{\sum_{j=1}^{n} f_j} \right). \end{aligned}$$

In a generation, the fitness value of a heuristic usually does not change significantly after a number of tests; therefore, the value of $\frac{\partial f_i}{\partial n_i}$ is small. The second term is close to 1 for a moderate number of heuristics, say 10. Therefore, it is reasonable to neglect the

derivative of P_i, that is, $\frac{\partial P_i}{\partial n_i} \approx 0$. We can rewrite the optimality criteria in Eq. (5.1) as

$$-\lambda = P_i \frac{\partial L_i}{\partial n_i} = P_j \frac{\partial L_j}{\partial n_j} . \tag{5.2}$$

The following lemmas and theorems provide a statistical foundation for computing the losses L_i and L_w.

Lemma 5.1. *Let X_1, ..., X_n be n independent random variables. Random variable X_i has known mean μ_i and variance σ_i. Let $\rho_{i,j}$, $i \neq j$, denote the correlation coefficient between two random variables X_i and X_j. Then, the mean and the variance of the linear statistic $Y = \sum_{i=1}^{n} a_i X_i$ are*

$$\mu_Y = \sum_{i=1}^{n} a_i \mu_i \quad \text{and}$$
$$\sigma_Y^2 = \sum_{i=1}^{n} a_i^2 \sigma_i^2 + \sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j \rho_{i,j} \sigma_i \sigma_j ,$$

respectively, where a_1, ..., a_n are real-valued constants.

Proof. Let ϕ_i denote the p.d.f. of X_i for $i = 1, ..., n$. Without loss of generality, we assume that these random variables are continuous. According to the definition of Y, we can compute its mean as

$$\begin{aligned}
\mu_Y &= \int_{-\infty}^{\infty} \cdots \int \left(\sum_{i=1}^{n} a_i x_i \right) \left(\prod_{j=1}^{n} \phi_j \right) dx_1 \cdots dx_n \\
&= \sum_{i=1}^{n} a_i \left[\int_{-\infty}^{\infty} \cdots \int x_i \left(\prod_{j=1}^{n} \phi_j \right) dx_1 \cdots dx_n \right] \\
&= \sum_{i=1}^{n} a_i \left[\int_{-\infty}^{\infty} \cdots \int \left(\int_{-\infty}^{\infty} x_i \phi_i \, dx_i \right) \left(\prod_{j \neq i} \phi_j \right) dx_1 \cdots dx_{i-1} \, dx_{i+1} \, dx_n \right] \\
&= \sum_{i=1}^{n} a_i \left[\int_{-\infty}^{\infty} \cdots \int \mu_i \left(\prod_{j \neq i} \phi_j \right) dx_1 \cdots dx_{i-1} \, dx_{i+1} \, dx_n \right]
\end{aligned}$$

$$= \sum_{i=1}^{n} a_i \, \mu_i \, .$$

The variance of Y can be computed as

$$\begin{aligned}
\sigma_Y^2 &= \mathrm{E}\left[\left(\sum_{i=1}^{n} a_i X_i - a_i \mu_i \right)^2 \right] \\
&= \mathrm{E}\left[\sum_{i=1}^{n} a_i^2 (X_i - \mu_i)^2 + \sum_{i=1}^{n} \sum_{j=1}^{n} \left(a_i a_j (X_i - m_i)(X_j - m_j) \right) \right] \\
&= \sum_{i=1}^{n} a_i^2 \, \mathrm{E}[(X_i - \mu_i)^2] + \sum_{i=1}^{n} \sum_{j=1}^{n} \left(a_i a_j \, \mathrm{E}[(X_i - m_i)(X_j - m_j)] \right) \\
&= \sum_{i=1}^{n} a_i^2 \, \sigma^2 + \sum_{i=1}^{n} \sum_{j=1}^{n} \left(a_i a_j \, \rho_{i,j} \, \sigma_i \, \sigma_j \right)
\end{aligned}$$

□

The assumption of Lemma 5.1 is the knowledge of μ_i and σ_i for every $i = 1, ..., n$. If X_i and X_j are mutually independent, then their correlation coefficient becomes zero, i.e., $\rho_{i,j} = 0$.

Lemma 5.2. *Let $X_1, ..., X_n$ be n independent random samples from a distribution with mean μ and variance σ. If we use the average of X_i for $i = 1, ..., n$ as an estimator $\hat{\mu}$ of the mean μ, that is, $\hat{\mu}$ =del $\bar{X}$, then the loss L in treating $\hat{\mu}$ as the estimator is*

$$L = \textit{del } \mathrm{E}[(\hat{\mu} - \mu)^2] = \frac{\sigma^2}{n}$$

Proof. Let $Y = \hat{\mu} = \bar{X}$ and $a_i = \frac{1}{n}$ for $i = 1, ..., n$, then we have $Y = \sum_{i=1}^{n} a_i X_i$. According to Lemma 5.1, we have $\mu_Y = \mu$ and $\sigma_Y^2 = \frac{\sigma^2}{n}$. Loss L can be computed as

$$L = \mathrm{E}[(\bar{X} - \mu)^2] = \mathrm{E}[\,(Y - \mu)^2] = \sigma_Y^2 = \frac{\sigma^2}{n} \, .$$

□

Loss L_w can then be computed as

$$L_w = \mathrm{E}[\hat{\mu}_w - m_w] = \frac{\sigma_w^2}{n_w} .$$

The derivative of the loss becomes

$$\frac{\partial L_w}{\partial n_w} = \frac{\sigma_w^2}{n_w^2} . \tag{5.3}$$

Theorem 5.1. *Assume that the performance of all heuristics is mutually independent. Loss L_i of the belief that fitness value f_i is accurate is*

$$L_i = \frac{\sigma_i^2}{n_i} + \frac{\sigma_w^2}{n_w}$$

for $i \neq w$, where heuristic w is the base heuristic for computing fitness values.

Proof. Loss L_i is computed as the expected variance of the fitness values. According to the definition of fitness values, we have

$$\begin{aligned}
L_i &= \mathrm{E}[(f_i - F_i)^2] \\
&= \mathrm{E}\left[\left((\hat{\mu}_i - \hat{\mu}_w) - (\mu_i - \mu_w) \right)^2 \right] \\
&= \mathrm{E}\left[\left((\hat{\mu}_i - \mu_i) - (\hat{\mu}_w - \mu_w) \right)^2 \right] \\
&= \mathrm{E}[(\hat{\mu}_i - \mu_i)^2 + (\hat{\mu}_w - \mu_w)^2 - 2(\hat{\mu}_i - \mu_i)(\hat{\mu}_w - \mu_w)] \\
&= \mathrm{E}[(\hat{\mu}_i - \mu_i)^2] + \mathrm{E}[(\hat{\mu}_w - \mu_w)^2] - 2\,\mathrm{E}[(\hat{\mu}_i - \mu_i)(\hat{\mu}_w - \mu_w)] .
\end{aligned}$$

According to the assumption that the performance of all heuristics is mutually independent, we have

$$\mathrm{E}[(\hat{\mu}_i - \mu_i)(\hat{\mu}_w - \mu_w)] = 0$$

for $i \neq w$. According to Lemma 5.2, we, therefore, have

$$L_i = \frac{\sigma_i^2}{n_i} + \frac{\sigma_w^2}{n_w} .$$

□

We, then, compute the derivative of the loss as

$$\frac{\partial L_i}{\partial n_i} = \frac{\sigma_i^2}{n_i^2} \,. \tag{5.4}$$

Note that the form in Eq. (5.4) for the derivative of the loss of heuristic i for $i \neq w$ is the same as that in Eq. (5.3) for heuristic w. Therefore, in the following, we do not distinguish them explicitly when we are discussing the derivative of the loss. Finally, we can complete the optimality criteria in Eq. (5.2) as

$$P_i \frac{\sigma_i^2}{n_i^2} = P_j \frac{\sigma_j^2}{n_j^2} \,.$$

Since $\frac{\sigma_i^2}{n_i^2}$ is monotonically decreasing as n_i is increasing, in order to satisfy the optimality criteria, we, therefore, can select the heuristic that has the largest value of $P_i \frac{\sigma_i^2}{n_i^2}$ and test it. That is, the test scheduling strategy selects a heuristic z if this heuristic satisfies

$$P_z \frac{\sigma_z^2}{n_z^2} = \max_{i=1,m} P_i \frac{\sigma_i^2}{n_i^2} \,. \tag{5.5}$$

This test scheduling strategy minimizes the risk of selecting wrong heuristics to generate new ones for the next generation; therefore, we call it the *minimum-risk (MR) strategy*. The *characteristic equation* of the minimum-risk strategy is defined in Eq. (5.5). Recall that it is derived based on an assumption that the mean and the variance are known. However, the mean and the variance are generally not known during learning. In this thesis, we replace them by their unbiased estimators when we compute the values in Eq. (5.5).

The MR strategy is sensitive to the order of the test sequence. If the test sequence is ordered in such a way that speedups of heuristics are decreasing, then the MR strategy will

work similar to the RR strategy because evaluating more tests can result in a decrease of the estimated mean $\hat{\mu}_i$ and equivalently result in the decrease of f_i and P_i. If the test sequence is ordered in a way that speedups of heuristics are increasing, then the MR strategy will spend too much time in testing a few good heuristics, where the goodness is based on the performance values of limited tests.

The statistics used in the MR strategy can all be computed incrementally; that is, the estimated averages and the estimated variances can be computed incrementally rather than re-computed every time from the beginning. Assume that the current statistics for a certain heuristic include an estimated average $\hat{\mu}_{old}$, an estimated variance $\hat{\sigma}_{old}$, a number n_{old} of instances that have been tested, and a fitness value f_{new}. Assume that the new performance value is x_{new}. The incremental computation of statistics is based on the following equalities.

$$n_{new} = n_{old} + 1 ,$$

$$\hat{\mu}_{new} = \frac{\hat{\mu}_{old}\, n_{old} + x_{new}}{n_{new}} ,$$

$$\hat{\sigma}^2_{new} = \frac{n_{new}}{n_{new} - 1} \left(\frac{x^2_{new} + (n_{old} - 1)\,\hat{\sigma}^2_{old} + n_{old}\,\hat{\mu}_{old}}{n_{new}} - \hat{\mu}_{new} \right) ,$$

where $(n_{old} - 1)\,\hat{\sigma}^2_{old} + n_{old}\,\hat{\mu}_{old}$ is the sum of the squares of each of the previous performance values. The MR strategy is not used until all heuristics have a certain number of tests. After this minimum number of tests, the worst heuristic, say heuristic w, will serve as a base for computing fitness values of heuristics. Because the fitness value f_w of heuristic w is zero, it will not be selected for testing hereafter. Therefore, the base for computing fitness values will not change, and the new fitness value becomes

$$f_{new} = f_{old} + \hat{\mu}_{new} - \hat{\mu}_{old} .$$

5.5. Verification Phase

An application domain can be divided into several subdomains, each having different characteristics. During learning, we wish to learn a set of heuristics, each of which is chosen for a specific subdomain or a specific set of subdomains. We can envision the whole application domain as comprising a number of manually classified subdomains that may overlap with each other, as discussed earlier in Section 5.2.3.

During a learning phase, we generate and test heuristics and propose a set of heuristics that are good for solving problem instances in each subdomain. During a verification phase, we select a single set of heuristics that come from the good heuristics (proposed by the learning phase) in all subdomains. Then, we fully evaluate these selected heuristics such that they have the same amount of tests on all tested subdomains. Finally, we apply a statistical method to cluster subdomains in a way that the performance of heuristics for solving problem instances in the subdomains of a cluster are consistent in terms of the ranking of their performance.

5.5.1. Full evaluation

When a heuristic is evaluated across a number of problem instances, the average of speedups can vary up and down. We can plot the profile of the speedup averages against the number of tests. Whenever the profile plots of two heuristics cross each other, inconsistency in predicting which one is better may happen. The situation is even worse when the number of tests on these two heuristics is not equal. Therefore, there is a need to fully evaluate heuristics when learning is completed.

Full evaluation is to assure that all interested heuristics have the same and reasonably

large number of tests, so the performance comparison can be done in high confidence. A subdomain d_i has a finite set $\mathbf{I_i}$ of problem instances. Full evaluation is done to assure that all the heuristics are tested over the entire set $\mathbf{I_i}$ of problem instances if these heuristics are to be tested on this subdomain d_i.

The input to full evaluation is a number of sets of heuristics, each set corresponding to a subdomain. Assume that there are n subdomains that heuristics have been tested on. Let $\mathbf{H_i}$ be the set of heuristics that are good for solving problem instances in subdomain d_i. Let $\mathbf{H}$ be the set of heuristics, namely, $\mathbf{H} = \bigcup_{i=1}^{n} \mathbf{H_i}$. On the other hand, $\mathbf{H_i}$ is associated with set $\mathbf{D_i}$ of subdomains that heuristics in set $\mathbf{H_i}$ have been tested on.

In full evaluation, we first find the total set $\mathbf{D}$ of all tested subdomains

$$\mathbf{D} = \bigcup_{i=1}^{n} \mathbf{D_i} ,$$

and assure that all heuristics in $\mathbf{H}$ are fully tested on the set $\mathbf{D}$ of subdomains.

In a verification phase, we can also include other subdomains that heuristics have not been tested on. Let $\mathbf{D_u}$ be the set of such subdomains. Then, the set $\mathbf{D}$ of subdomains to be tested becomes

$$\mathbf{D} = \left(\bigcup_{i=1}^{n} \mathbf{D_i} \right) \cup \mathbf{D_u} .$$

The output of full evaluation is the set $\mathbf{D}$ of subdomains and $|\mathbf{D}|$ sets of heuristics, where set $\mathbf{H_i}$ of heuristics is good for solving problem instances in subdomain d_i *member* $\mathbf{D}$. Also, all heuristics are fully tested on problem instances in all subdomains in set $\mathbf{D}$. Note that set $\mathbf{H_i}$ of heuristics is associated with a new set ($=\mathbf{D}$) of subdomains.

5.5.2. Domain clustering

A subdomain is characterized by a collection of problem instances that have similar characteristics in terms of both parameters and the complexity of finding the desired solution. However, the boundary between two subdomains may not be so clear. Moreover, it is intractable to learn a spectrum of heuristics for all different (possibly an infinite number of) subdomains. We, therefore, manually select subdomains of the application and learn good heuristics that work well on these subdomains.

The manually selected subdomains may not reflect exactly the actual subdomains because the latter ones are unknown in general. Also, the manually selected subdomains may overlap with each other in an unknown fashion since the boundary is vague. As a result, some of the manually selected subdomains may be combined into a single subdomain. When learning is completed, we cluster subdomains in a way that heuristics perform consistently in terms of the ranking of their performance across subdomains in a cluster.

In the last verification phase, we fully test heuristics over all specified subdomains. We do not have to select good heuristics for the next learning phase because it is the last step in the process. We, instead, conduct *domain clustering*, in which heuristics are ranked on each verified subdomain and we use an affinity measure to see whether two subdomains can be combined (or clustered). A set of subdomains can be clustered if their pair-wise affinity measures satisfy a given criterion. Finally, we select and deliver the very top heuristic for each subdomain or each cluster of subdomains.

Domain clustering is important when we want to combine subdomains that have similar search performance and when we want to use clusters of manually selected subdomains to

simulate actual subdomains.

Ideally, domain clustering should be done with respect to a specified distribution of problem instances. When the distribution of problem instances is known, the probability density function is denoted as $f(x)$, where x is the parameter that characterizes subdomains. Without loss of generality, we assume the domain of x is continuous. Let $s_i(x)$ denote the (average) speedup of heuristic i on subdomain x, where the speedup is with respect to a baseline heuristic. Then, the very best heuristic h^* is selected to

$$\underset{i}{maximize} \int f(x)\, s_i(x)\, \mathrm{d}x \,. \tag{5.6}$$

When the distribution of problem instances is not specified, we are interested in the heuristic that generally performs well across all subdomains interested and for all possible distribtions of problem instances. In this case, we apply the probability-of-win method to predict the goodness of a heuristic; therefore, Eq. (5.6) can be rewritten as

$$\underset{i}{maximize} \int p_{i\,|H}(x)\, \mathrm{d}x \,, \tag{5.7}$$

where $p_{i\,|H}(x)$ denote the probability of win of heuristic i among a set H of heuristics on subdomain x. In this study, we do not assume any prior distribution of problem instances; therefore, we use Eq. (5.7) to select the best heuristic when we are only interested in a heuristic that works well across all subdomains.

When we allow multiple heuristics for different subdomains, we can use the best heuristic from each subdomain tested. The resulting performance curve is the envelope of the performance curves of the top heuristics from all subdomains. Figure 5.7 shows an example of the envelope (the solid curve with symbol □), where dotted curves indicate the performance values of the top heuristics across different subdomains. Another solid curve

with symbol Δ gives the performance values of the best heuristic (across all subdomains). These curves are obtained from the learning experiment that learns decomposition heuristics used for solving the VC problem, where the random seed is 61801 and the scheduling strategy used is MR.

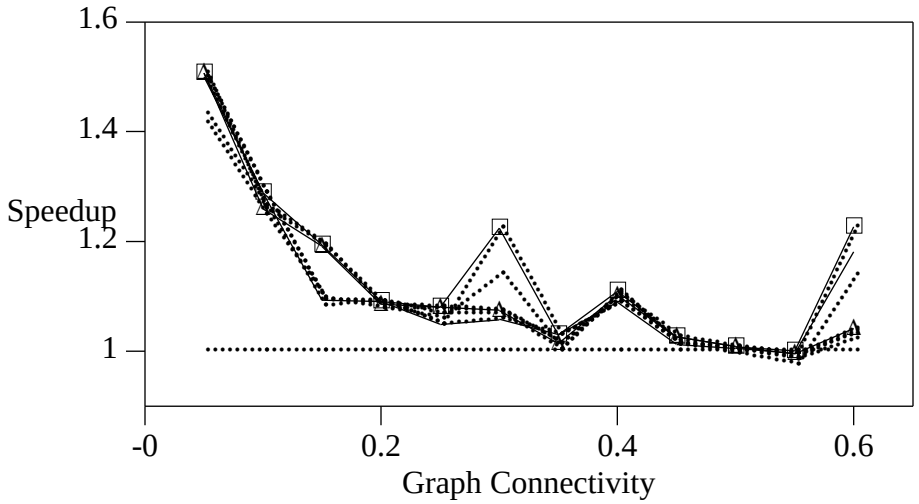


Figure 5.7. The envelope (denoted by □) of the performance values of the top heuristics of all subdomains and the performance values (denoted by Δ) of the very best heuristic across all subdomains, where dotted curves indicate performance values of the top heuristics of all subdomains.

When the distribution of problem instances is not specified and when we allow multiple heuristics, domain clustering can be a crucial method for characterizing similarity of search performance in subdomains. Different subdomains with similar search performance can be clustered and only a single heuristic is proposed for this cluster of subdomains; in contrast, we propose a heuristic for each subdomain when we know the specified distribution of problem instances. In short, our approach is to cluster subdomains first and then select the best heuristic for each cluster, where the goodness of heuristics is in terms of the probability of win.

Our approach in domain clustering is not restricted to any specific statistical method. We can use any affinity measure that is consistent between two subdomains and that can form an equivalence class.

In this section, we first describe the requirement of the affinity measure and, without loss of generality, we suggest a well-known statistical index to serve this purpose. After these, we formulate the domain clustering problem as an NP-hard graph partitioning problem, namely, the *partition-into-cliques* problem [18]. Finally, we choose a very effective approximation algorithm published in the literature to solve the problem [4].

5.5.2.1. Affinity measure

The affinity measure for domain clustering can be any one that can provide statistically sound information on the consistency or the inconsistency of performance of heuristics used to solve problem instances in any two subdomains.

To design a statistical method to check the consistency or inconsistency of performance of heuristics used to solve problem instances in two subdomains, we have to define or assume the relation between consistency (or inconsistency) and performance of heuristics. In this thesis, we define consistency as follows.

Domain Consistency. Two subdomains are said to be consistent under a given set of heuristics and under two sets of problem instances (one from each subdomain) if the ranking of performance of these heuristics for solving problem instances in one set of problem instances is highly correlated to that of the performance of solving problem instances in the other set of problem instances.

From this definition, if the rankings of heuristics on two subdomains are highly correlated, then we can infer that these heuristics perform consistently on these two subdomains. With this information, we can combine subdomains that are highly correlated into a single cluster of subdomains.

The correlation of two ranking systems can be checked statistically. After full evaluation, a set of heuristics is proposed, and these heuristics with their performance information are the input to domain clustering. Since we are dealing with the same set of heuristics, the range of the two rankings on the two subdomains is the same.

In this thesis, we propose to use the *coefficient of rank correlation* (CRC) as an index to indicate the correlation of two ranking systems on the same set of objects.

Consider a set **H** of *n* heuristics and two subdomains d_1 and d_2. Assume that the ranks of heuristic *k* on subdomains d_1 and d_2 are $r_{k,1}$ and $r_{k,2}$, respectively. If the difference $\delta_{k,1,2} = del\ |r_{k,1} - r_{k,2}|$ is small for every possible *k*, then this implies these two ranking systems are correlated.

The CRC $r_{i,j}^{\mathbf{H},\mathbf{I_i},\mathbf{I_j}}$ of two ranking systems of a set *H* of *n* heuristics for solving a set $\mathbf{I_i}$ of instances in subdomain d_i and for solving a set $\mathbf{I_j}$ of instances in subdomain d_j is defined as

$$r_{i,j}^{\mathbf{H},\mathbf{I_i},\mathbf{I_j}} = del\ 1 - \frac{2\sum_{k=1}^{n} \delta_{k,i,j}^2}{D(n)},$$

where $D(n)$ is the worst-case value of $\sum_{k=1}^{n} \delta_{k,i,j}^2$ such that CRC is a value in between -1 and 1.

When every heuristic has a unique rank in a ranking system, we call this ranking a *unique ranking*. It happens when the performances of all heuristics are distinct. In this case, $D(n) = del\ n\,(n^2 - 1)/3$ [65].

In the learning system, the heuristics retained in the verification phase usually perform equally well; namely, the heuristics are competitive. Therefore, a large rank difference may occur for a small difference in performance values, which alleviates the credibility of CRC.

To overcome this adverse effect, we propose *segment ranking* that gives the same rank to all heuristics whose performance values are in close vicinity to each other. In the case of segment ranking, $D(n) = del\, n\,(m^2 - 1)$, where m is the number of segments. We empirically found that CRC generally increases as the number of segments decreases for small numbers of segments. In an extreme case, CRC is one when $m = 1$; that is, all heuristics are in the same segment and, therefore, of the same rank. In another special case, when $m = n$, $D(n)$ in segment ranking is different from $D(n)$ in unique ranking because segment ranking allows multiple heuristics to have the same rank values.

To avoid too many empty segments due to different ranges of performance values for different subdomains, we normalize performance values in a way that we align the maximum performance values. Let α denote the maximum of all performance values; that is,

$$\alpha = \min_x \left(\max_i s_i(x) \right),$$

where $s_i(x)$ is the speedup of heuristic i on subdomain x. Then a normalized speedup $s_i'(x)$ is computed as

$$s_i'(x) = s_i(x) \frac{\alpha}{\max_y s_i(y)}.$$

The idea of this normalization is to align the maxima of the performance values of all subdomains. The minima of the performance values are not aligned because in some subdomains performance values are all similar and all heuristics under verification can be assigned the same rank.

The segment ranking is controlled by the number of segments n_{seg}. The resolution interval is the interval within which performance values are assigned to the same rank. To

tighten the resolution interval, we use α and β, the minimum of the minimum performance values of all subdomains, to define the resolution interval δ_{seg}. That is,

$$\delta_{seg} = \frac{\alpha - \beta}{n_{seg}}, \tag{5.8}$$

where $\beta = \min_x \min_i s_i(x)$.

We define a CRC threshold γ that defines the necessary strength of correlation of two ranking systems for two subdomains to be clustered. If a set of subdomains can be clustered, then every possible pair of subdomains in this set has its CRC no less than threshold γ.

The result of clustering is dependent on the CRC threshold and the number of segments; however, we do not have a sound theory to determine their values. Instead, we plot in Figure 5.8 the distribution of the resulting clusters for different possible values of thresholds and numbers of segments. We call this plot a *cluster plot*. In a cluster plot, a symbol represents a possible cluster set.

We found that the resulting clusters in general are not sensitive to the number of segments, as are seen in Appendix B. It happens because $D(n)$ and $\delta_{k,i,j}$ are enlarged roughly in the same scale when the number of segments increases. On the other hand, the resulting cluster sets generated are moderately sensitive to the CRC threshold around the range [0.75,0.95] which we are interested in, but clustering is very sensitive to the threshold when it is close to 1.

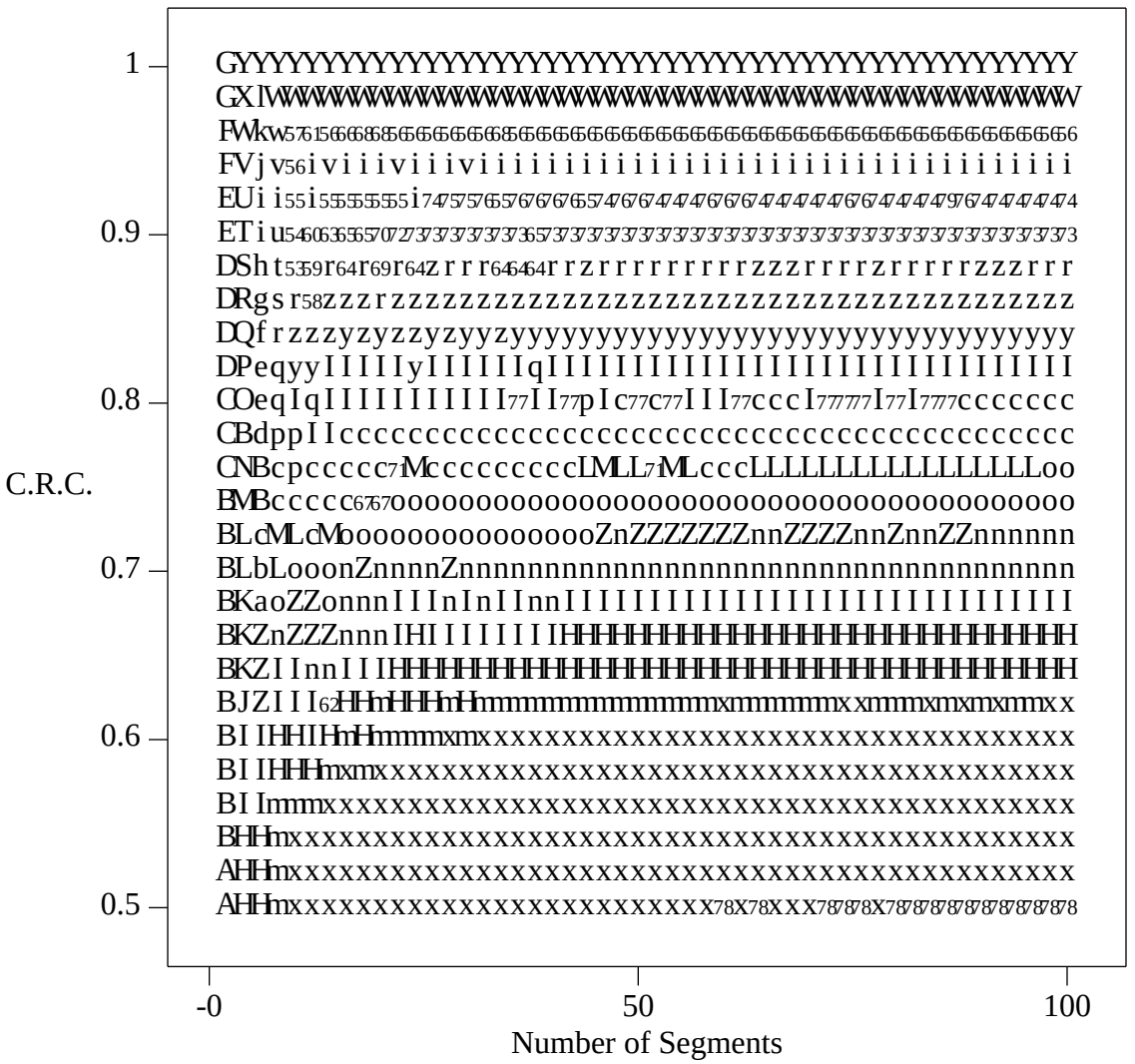


Figure 5.8. An example of a cluster plot in which each symbol represents a possible cluster set, where a cluster set on positions x and y is obtained by using *x* segments for segment ranking and using CRC *y* for domain clustering. (This cluster plot is obtained from learning selection heuristics used in dynamic band search to find a feasible solution of the maze problem, where the learning is performed with the MR strategy and a random seed 61801.)

The cluster plot is useful when parameters used to define subdomains are discrete or a combination of discrete and continuous because it is hard to define neighoring subdomains. For continuous parameters, we may not need the cluster plot because (a) clustering is not

sensitive to the number of segments, and (b) neighboring subdomains are well-defined. We can use a graph to illustrate the clustering by showing curves that connect the worst CRC values of clusters of various sizes. Figure 5.9 shows an example of such curves. These curves are obtained in learning decomposition heuristics used for solving the VC problem, where the learning is with a random seed of 94035 and MR scheduling strategy. A point (denoted as *k*) at positions *x* and *y* indicates that the worst CRC values in a cluster containing *k* neighboring subdomains starting from the subdomain with parameter *x* is equal to *y*. We can use this graph to see the number of clusters that can be formed by reducing the CRC threshold gradually.

It is interesting to note that subdomain 12 (graph connectivity is 0.6) is very different from the other subdomains because all attempts to include it into a cluster result in a low CRC value. This is illustrated by a row of low CRC values.

We can imagine the CRC threshold is a horizontal line. At the beginning, the horizontal threshold line is at CRC equal to 1. When we move the line downwards, we can find the resulting clusters that satisfy the CRC threshold; that is, we can find the points above the threshold line.

Note that cluster curves are only useful for continuous parameters. When parameters are discrete or a combination of continuous and discrete, cluster curves carry no meaning unless we know how to define neighboring subdomains.

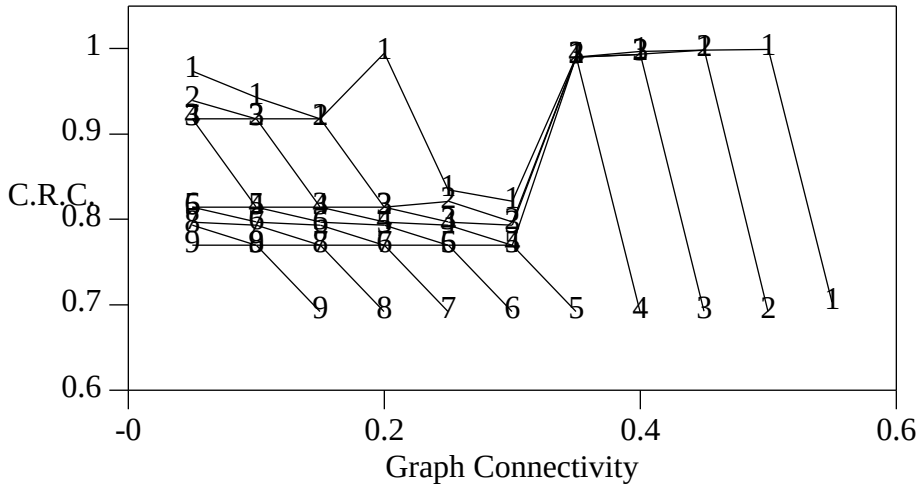


Figure 5.9. An example of cluster curves, in which each curve denoted by *k* indicates the worst CRC values for possible clusters of size $k+1$. (These curves are obtained in learning decomposition heuristics used for solving the VC problem, where learning is with a random seed of 94035 and MR scheduling strategy. The number of segments is 40 and the number of heuristics is about 60.)

5.5.2.2. Formulation of domain clustering problem

In this section, we formulate the domain clustering problem as the *partition-into-cliques* problem [18], in which the corresponding optimization problem is to minimize the number of cliques (i.e., subdomain clusters).

Consider a complete graph $G_0 = (\mathbf{V_0}, \mathrm{E_0})$, where $\mathbf{V_0}$ is the set of vertices and $\mathrm{E_0}$ is the set of edges. A vertex represents a single subdomain. An edge *e* is labeled by its corresponding CRC value r_e due to the ranking of heuristics for solving instances in two subdomains represented by the two vertices connected by this edge.

Consider another graph $G = (\mathbf{V}, \mathrm{E}; G_0, \gamma)$, where $\mathbf{V} = \mathbf{V_0}$ and $\mathrm{E} = \{ e \mid e \textit{ member } \mathrm{E_0}$ and $r_e \geq \gamma \}$. In graph *G*, an edge between two vertices means that two subdomains represented by these two vertices can be combined. A cluster of subdomains is then represented as a

clique (i.e., a complete sub-graph) because all possible pairs of subdomains (or vertices) in a cluster (or clique) can be combined (or are connected).

The domain clustering problem is, therefore, exactly equivalent to the problem of partitioning the graph G into cliques. The decision problem of partitioning a graph into cliques is NP-complete [18]. In this thesis, we are interested in finding the partitioning such that the number of cliques is minimum. This optimization problem is NP-hard.

The problem of partitioning a graph $G=(\mathbf{V}, E)$ into cliques is equivalent to the problem of partitioning the complement of G into independent sets. Moreover, the problem of partitioning a graph into independent sets is exactly the problem of coloring the graph.

Graph-coloring problems have been extensively studied, and there are many good approximation algorithms [4, 11, 91]. Turner reported the no-choice algorithm and Brelaz's algorithm can almost find the optimal graph coloring for almost all random graphs [91]. However, the no-choice algorithm requires prior knowledge of the k-colorability of the graph, and the chromatic number k is one of the inputs to the algorithm. However, in our case, the chromatic number is not known ahead of time. Therefore, we employ Brelaz's algorithm [4] to partition a graph into independent sets (or equivalently, partition a graph into cliques).

Besides minimizing the number of subdomain clusters, we also wish to improve the worst CRC in each subdomain cluster; namely, we wish to have the best domain clustering. Our primary objective is to minimize the number of clusters and second, maximize the worst CRC in each subdomain cluster.

We apply a greedy algorithm to improve the worst CRC in clusters, given a minimized number of subdomain clusters. We first find the cluster that has the worst CRC in all possible clusters. After that, we try to move one of the two subdomains associated with the worst CRC into other clusters without decreasing the worst CRC in all clusters after the move. This process is repeated until it fails. This process is greedy because it does not allow backtracking.

5.5.3. Performance evaluation

For each cluster of subdomains, we have to assess heuristics in order to propose the best one for this cluster. The assessment is based on the performance of heuristics in solving problem instances in this cluster of subdomains. The average of performance values may not be a good measure because the ranges of performance values in all subdomains in a cluster may be different, as is seen in Figure 5.6. For example, the range of speedups in subdomain one may be from 1 to 2, but the range in subdomain two may be from 4 to 7. Note that subdomains are clustered when the performances on these subdomains are *consistent* in terms of ranking correlation, rather than in terms of their raw performance values.

At the end of domain clustering, we test performance evaluation methods to evaluate a set of heuristics that are equally tested. Since all heuristics are fully evaluated, a significant number of performance values are available for each heuristic. Also, since only the very top heuristic of each cluster has to be identified, we do not need to sort all the heuristics in terms of their performance values. Moreover, we can compute performance values of heuristics in a batch rather than in an incremental fashion because no more tests are required.

There are three requirements of the performance evaluation method used in the learning phase: (1) the measure must be able to propose the heuristic with the best performance value, (2) the measure includes the effects on the number and variance of performance data, and (3) the measure must be able to combine performance values of different ranges.

In the following, we describe our methods for performance evaluation, which include the pure average (PA) method, the average rank (AR) method, the worst rank (WR) method, the number of wins (NW) method, the probability of win (PW) method, the speedup normalization (SN) method, and the equal-weight normalization (EWN) method.

To facilitate the description, let's assume the following situation. Assume there are m heuristics and n subdomains. Let $x_{i,j}$ denote the average performance value (i.e., average speedup, in this thesis) of heuristic i after κ_j tests in subdomain d_j. Similarly, let $\delta_{i,j}$ denote the standard deviation. Let $r_{i,j}$ denote the rank of heuristic i for its performance on subdomain d_j, where $1 \le r_{i,j} \le m$, and smaller rank values represent better heuristics.

(1) *Pure Average.* We can use the average of average speedups of heuristics for solving problem instances in all these n subdomains to assess heuristics, namely, use $\frac{1}{n}\sum_{j=1}^{n} x_{i,j}$ as the performance value of heuristic i over these n subdomains. It suffers from both overweight and speedup-slowdown anomalies in the same way when the PA method is used in the learning phase. Moreover, averages do not take into account the number of tests and the variance of performance values. Further, the ranges of average speedups on different subdomains may be so distinct that averages become meaningless.

(2) *Average Rank.* We can use the average of all the rank values of heuristics for solving problem instances in all these n subdomains to assess heuristics, namely, use $\frac{1}{n}\sum_{j=1}^{n} r_{i,j}$ as the performance value of heuristic i over these n subdomains. The advantage is that it avoids the speedup-slowdown anomaly. However, it suffers from the overweight anomaly. Average rank does not take into account the number of tests and the variance of performance values.

(3) *Worst Rank.* We can use the worst-case rank value to assess heuristics, namely, use $\max_{j=1,\,n} r_{i,j}$ as the performance value of heuristic i over these n subdomains. The major advantage is that it can give a picture of the worst-case rank of a heuristic, which can be viewed as its worst-case behavior. It inherits the advantages and disadvantages of the average-rank method. It can be proved that the worst rank value of a heuristic becomes larger after more tests because the worst rank value is nondecreasing. Also, after a large number of tests, almost all heuristics may have the same worst rank value that is approximately equal to the number of heuristics. This phenomenon can be illustrated by the following simple calculation. Assume heuristic i has a p.d.f. $f_i(r)$ (c.d.f. $F_i(r)$) for rank values, where r is a rank value and $1 \le r \le m$. Then, $\sum_{r=1}^{m} f_i(r) = 1$ for every $i = 1, ..., m$. Let r_i^w denote the worst rank value for heuristic i over n subdomains.

$$F_{i,n}^{w}(r) = Pr[r_i^w \le r] = Pr[r_{i,1} \le r \;\&\; r_{i,2} \le r \;\&\; \cdots \;\&\; r_{i,n} \le r] = \prod_{j=1}^{n} Pr[r_{i,j} \le r] = (F_i(r))^n \ .$$

Assume the distribution is uniform, then after n_i tests the expected rank value is $\gamma_{i,n}$ that satisfies $F_{i,n}^{w}(\gamma_{i,n}) = 0.5$. After simple calculations, we have $\gamma_{i,n} = \frac{m}{\sqrt[n]{2}}$. Note that $\lim_{n \to \infty} \gamma_{i,n} = m$. This implies that over a large number of subdomains, almost all

heuristics may have the same worst rank value, which implies the worst-rank method is not scalable.

(4) *Number of Wins.* Consider two heuristics p and q. Let $a_{p,q} = del \sum_{j=1}^{n} u(x_{p,j} - x_{q,j})$ and $a_{q,p} = del \sum_{j=1}^{n} u(x_{q,j} - x_{p,j})$, where $u()$ is a step function, namely, $u(y) = 1$ for $y > 0$ and $u(y) = 0$ otherwise. Heuristic p wins if $a_{p,q} > a_{q,p}$. On the other hand, heuristic q wins if $a_{q,p} > a_{p,q}$. This comparison is performed for every possible pair of heuristics p and q. We use the number of wins as the performance value of a heuristic over these subdomains. This can avoid the speedup-slowdown anomaly and the overweight anomaly; however, it does not take into account the number of tests and the variance of performance values.

(5) *Probability of Win.* This PW method is similar to that proposed for the learning phase. Moreover, the probabilities on all subdomains are averaged. The major advantage is that it takes into account, in addition to the average speedups, the variance of speedups and the number of tests when comparing two heuristics. Moreover, it can avoid the speedup-slowdown anomaly and the overweight anomaly. Further, it is independent of the range of performance values across different subdomains.

(6) *Speedup Normalization.* For subdomain d_j, we can find the best average speedup and the worst average speedup among average speedups of all heuristics, and then normalize all average speedups to be between 0 and 1, where 0 indicates the worst average speedup and 1 indicates the best average speedup. That is,

$$x'_{i,j} = del \frac{x_{i,j} - \min_{k=1,m} x_{k,j}}{\max_{k=1,m} x_{k,j} - \min_{k=1,m} x_{k,j}}.$$

By this SN method, the speedup-slowdown and the overweight anomalies can be alleviated; however, it suffers from the amplification of negligibly minor differences when the best and the worst differ very little for some subdomains. Moreover, it does not take into account the number of tests and the variance of performance values.

(7) *Equal-Weight Normalization.* This EWN method is the same as the one proposed in the learning phase, except that the performance to be normalized is the average speedup across subdomains in this case. A disadvantage is that it does not take into account the variance of speedups and the number of tests when comparing two heuristics.

In our study, we choose the probability-of-win method as our method for performance evaluation because this method meets the three requirements posted in Section 5.4.1.

5.6. TEACHER: A Prototype for Automated Learning

TEACHER is a realization of the automated learning method we have described earlier in this chapter. The block diagram of the organization of TEACHER is shown in Figure 5.10. The realization of TEACHER is a common project in our research group led by Professor Wah. The application-independent part of TEACHER is mainly due to A. Ieumwananonthachai, S. Schwartz, and B. Wah who designed an early prototype. The work on verification is mainly due to B. Wah, L.-C. Chu, and A. Ieumwananonthachai.

From the view point of TEACHER, there are four major modules: (1) population-based learning system, (2) point-based learning system, (3) test-instance manager, and (4) application problem solver. Because we have already discussed many issues and functions of automated learning, in this section we concisely describe the components of TEACHER.

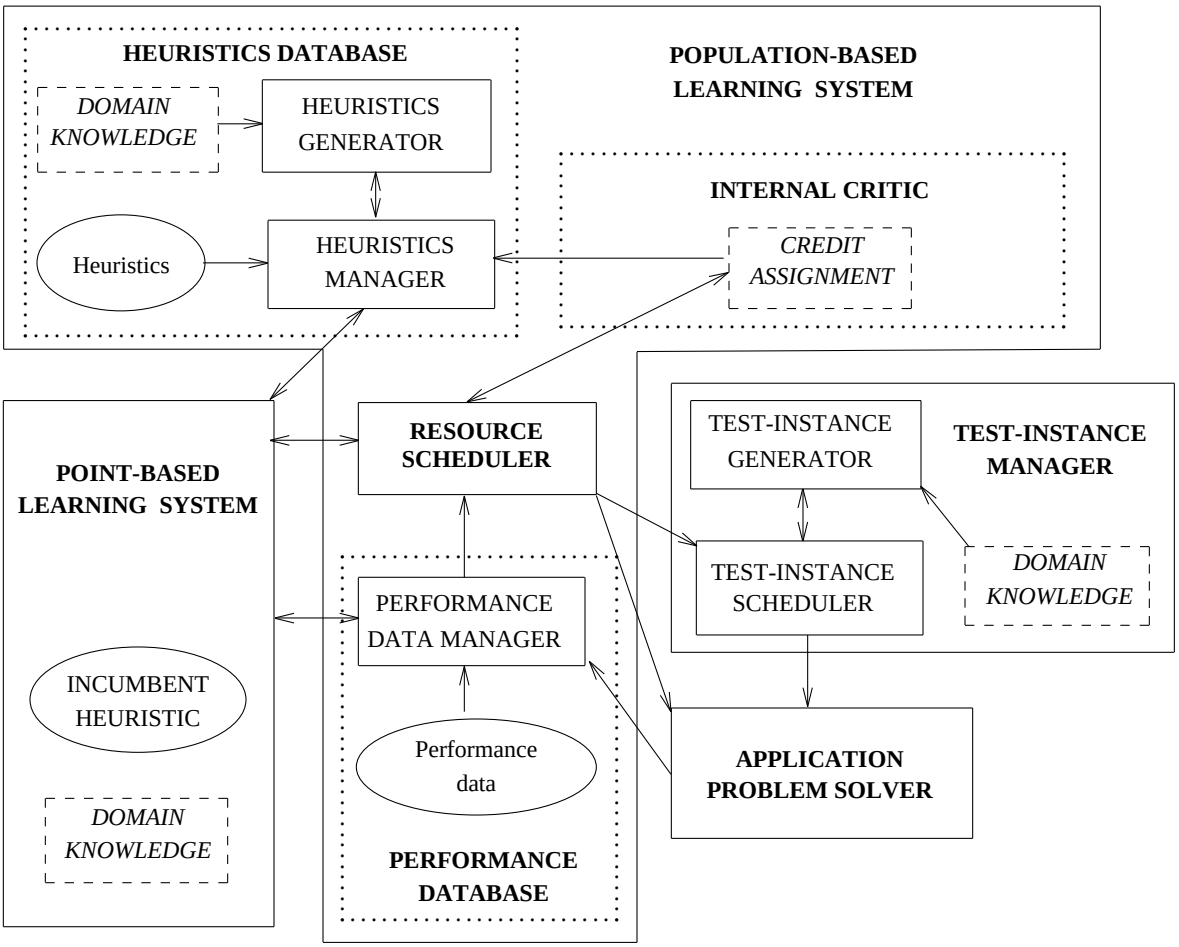


Figure 5.10. Framework of learning system TEACHER.

The population-based learning system is the main core of TEACHER and is the realization of the automated learning method we have discussed earlier in this chapter. The point-based learning system modifies heuristics in place and tests them in order to learn new heuristics. Note that the major difference between population-based learning and point-based learning is that the former maintains multiple heuristics at a time and generates new heuristics based on old ones, while the latter maintain a very limited number of heuristics and modifies old ones in place into new ones (that is, the new ones occupies the memory used by the old ones). There is a good analogy in search methods. Population-based

learning is like a beam search that maintains multiple choices at a time, while point-based learning is like a hill-climbing search that maintains a single thread of search. The test-instance manager handles test instances in multiple subdomains. The application problem solver can be reconfigured to accommodate new heuristics learned.

The internal critic evaluates the performance of heuristics and supplies the performance value to the resource scheduler and the heuristics generator. The core of the internal critic implements *credit assignment* [23] that assigns credit and blame to the causes leading to good heuristics and bad heuristics, respectively. Sutton proposed two types of credit assignment [90]: *structural credit assignment* and *temporal credit assignment*.

Structural credit assignment translates performance feedbacks associated with operators of generating heuristics into modifications of heuristics. Temporal credit assignment translates performance feedbacks associated with a past sequence of generation operations into modifications of heuristics. In this research, temporal credit assignment is *not* important because there is no delayed performance feedback. Therefore, we mostly neglect temporal credit assignment. To carry out structural credit assignment, we can use a rule-based system to generate new heuristics.

The generator of heuristics exercises genetic operators such as crossover, mutation, and coalescing, and creates new heuristics from a set of base heuristics selected from the previous generation. It is important to assure that the space of new heuristics that can be generated is ‘‘complete’’ in the sense that all semantically meaningful heuristics can be generated. That is, for a feasible heuristic, a sequence of operations exists that can be applied to base heuristics to generate this heuristic. Also, it is desirable that the newly generated heuristics are not functionally or semantically equivalent to any existing ones.

The resource scheduler is the main controller and decides when to generate new heuristics and how many, which heuristic to be tested, and when to end a generation. In this thesis, we mainly emphasize the scheduling of tests, that is, which heuristic to be selected for testing. The scheduling of tests is not important in the verfication phase, because we fully evaluate all remaining heuristics. In the learning phase, the scheduling of tests is important because heuristics are all partially tested and an effective schedule of tests is important.

5.7. Experimental Results

In this section, we describe the experiments we perform to demonstrate the learning framework and the learning schedule. Also, we describe the parameters used in our experiments that generate new heuristics and then demonstrate these new heuristics. We leave the details of our experiments in Appendix B for reference.

5.7.1. Overview of learning experiments

The overview of all learning experiments is shown in Figure 5.11, where solid lines indicate the relations between the search heuristics to learn and the target search algorithms, and dotted lines indicate the relation between the heuristics to learn and the applications which these heuristics are used to solve. The target search algorithms to be improved include real-time search and dynamic band search.

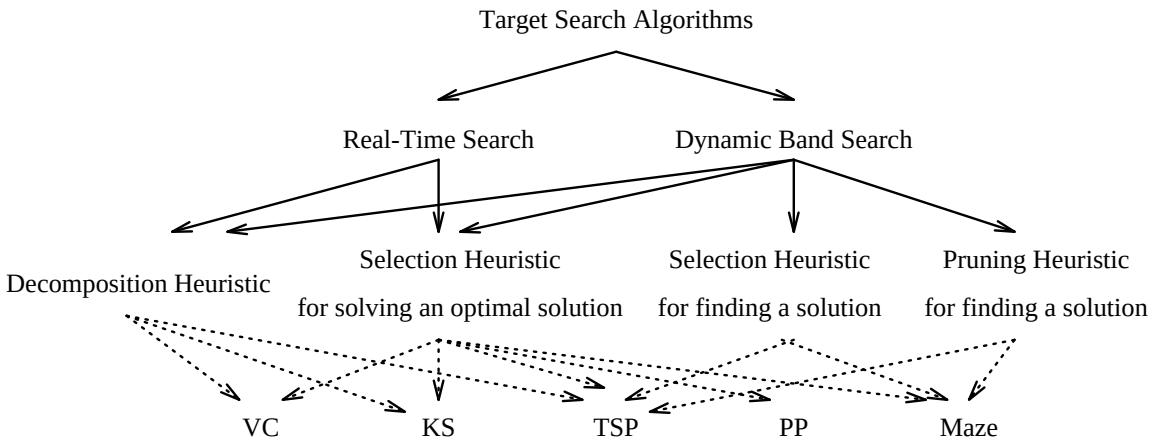


Figure 5.11. Overview of learning experiments.

In an RTS, we can learn decomposition and selection heuristics. The original objective of learning is that the new heuristics can lead to the best RTS that finds the best solution as quickly as possible in every possible time constraint. However, there are numerous possible time constraints; as a result, it may not be possible to learn heuristics under all possible time constraints. In this thesis, we adopt an approximation objective that minimizes the time for a primitive search of RTS to solve an optimal solution.

For dynamic band search, we can learn decomposition heuristics and selection heuristics. We are interested in not only finding the optimal solution as fast as possible, but also in finding a feasible solution as fast as possible. Therefore, we learn two selection heuristics: one for finding a feasible solution and the other for finding the optimal solution.

For decomposition heuristics, we apply learning to problems that involve choosing an attribute and decomposing the current state into descendent state (e.g., TSP, KS, and VC problems). For problems such as the maze and PP, multiple descendent states are created

from a selected state by enumerating all possible alternatives. In this thesis, we do not learn new decomposition heuristics.

For selection heuristics, we adopt two objectives: (1) finding a feasible solution and (2) finding an optimal solution. To improve RTSs, we use the second goal, while to improve dynamic band searches, we use both goals.

In heuristic pruning, nodes are pruned to limit the search space and memory requirement. This is not possible when bounds on solution quality are needed. Consequently, heuristic pruning is not appropriate for RTSs. Therefore, we only learn pruning heuristics for dynamic band searches. Further, heuristic pruning generally results in heuristic solutions.

The heuristic we study in this thesis is learned one at a time. The decomposition heuristics are learned first. The selection heuristics or the pruning heuristics are then learned. We choose to learn heuristics in this order because decomposition is done before selection or pruning in search.

In short, there are five combinations of search algorithms, search heuristics, and objectives of learning. These combinations and their notations are summarized in Table 5.3 for easy reference.

Table 5.3. Combinations of search algorithms, search heuristics, and objectives of learning studied in this thesis.

Notation	Search Heuristic	Learning Objective	Algorithm	Applications
$H_{D,sos,gdfs}$	Decomposition	Finding an optimal sol.	GDFS	KS, VC, TSP
$H_{S,sos,gdfs}$	Selection	Finding an optimal sol.	GDFS	KS, VC, PP, TSP
$H_{S,ffs,dbs}$	Selection	Finding a feasible sol.	BS[d]	TSP, Maze
$H_{S,sos,dbs}$	Selection	Finding an optimal sol.	BS[d]	TSP, Maze
$H_{P,ffs,dbs}$	Pruning	Finding a feasible sol.	BS[d]	TSP, Maze

5.7.2. Predefined problem subdomains

In this section, we describe our design of predefined problem subdomains used in our experiments. The idea is to select a finite and small subset of problem instances of different characteristics from a space of possibly infinite number of heuristics.

The predefined subdomains in the KS problem are designed based on the range of object weights and the range of the profit-weight ratio. The 12 subdomains used in our design are summarized in Table 5.4.

Table 5.4. Summary of predefined subdomains in the knapsack problem.

Sub-domain	Object Weights	Profit-Weight Ratio	Comments
1	[100,1000]	[1,1.5]	Large variance of weights and small variance of profit-weight ratios
2	[100,1000]	[1,10]	Large variance of weights and large variance of profit-weight ratios
3	[100,1000]	[1,1.05]	Large variance of weights and small variance of profit-weight ratios
4	[100,1000]	[1,100]	Large variance of weights and random profit-weight ratios
5	[100,200]	[1,1.5]	Medium variance of weights and medium variance of profit-weight ratios
6	[100,200]	[1,10]	Medium variance of weights and large variance of profit-weight ratios
7	[100,200]	[1,1.05]	Medium variance of weights and small variance of profit-weight ratios
8	[100,200]	[1,100]	Medium variance of weights and random profit-weight ratios
9	[100,105]	[1,1.5]	Small variance of weights and medium variance of profit-weight ratios
10	[100,105]	[1,10]	Small variance of weights and large variance of profit-weight ratios
11	[100,105]	[1,1.05]	Small variance of weights and small variance of profit-weight ratios
12	[100,105]	[1,100]	Small variance of weights and random profit-weight ratios

The predefined subdomains in the VC problem are designed based on the connectivity of the graph. There are also twelve subdomains in our experiments, and their connectivities are between 0.05 and 0.6.

The predefined subdomains in the TSP problem are designed based on the distributions of cities and connectivities of these cities. There are three different distributions of cities: (1) cities are uniformly distributed in both horizontal and vertical directions, (2) cities are uniformly distributed in one direction and distributed with a truncated normal distribution in the other direction, (3) cities are distributed with truncated normal distributions in both directions. The connectivities in our experiments include 1, 0.1, 0.2, and 0.3. Therefore, there are totally twelve subdomains. In the first four subdomains, cities are uniformly distributed in both directions, while in the last four subdomains, cities are normally distributed.

The predefined subdomains in the PP problem are designed based on the ranges of setup costs, production costs, and inventory costs. The 12 subdomains used in our design are summarized in Table 5.5.

The predefined subdomains in the maze problem are designed based on the pattern of the entrance and the exit, and the probability of walls being removed in our test problems. There are two patterns: one with the entrance and the exit on the opposite boundaries of the maze, which is the maze generated by the maze program in the X window environment [26], and the other with the entrance in the middle of the maze and the exit on a boundary, which is the maze used by some researchers to increase the difficulty of finding a solution [92]. The probabilities of removing walls are between 0 and 0.3. The first seven subdomains are in the first pattern and the others are in the second pattern.

Table 5.5. Summary of predefined subdomains in the production planning problem.

Sub-domain	Setup Cost	Production Cost	Inventory Cost	Comments
1	[50,100]	[10,20]	[5,10]	Medium setup cost, medium production cost, and medium inventory cost
2	[10,20]	[10,20]	[5,10]	Low setup cost, medium production cost, and medium inventory cost
3	[100,200]	[10,20]	[5,10]	High setup cost, medium production cost, and medium inventory cost
4	[50,100]	[10,20]	[10,20]	Medium setup cost, medium production cost, and high inventory cost
5	[50,100]	[10,20]	[2,5]	Medium setup cost, medium production cost, and low inventory cost
6	[100,200]	[10,20]	[10,20]	High setup cost, medium production cost, and high inventory cost
7	[100,200]	[10,20]	[2,5]	High setup cost, medium production cost, and low inventory cost
8	[10,20]	[10,20]	[10,20]	Low setup cost, medium production cost, and high inventory cost
9	[10,20]	[10,20]	[2,5]	Low setup cost, medium production cost, and low inventory cost
10	[10,20]	[100,200]	[2,5]	Low setup cost, high production cost, and low inventory cost
11	[100,200]	[2,5]	[10,20]	High setup cost, low production cost, and high inventory cost
12	[10,200]	[2,200]	[2,20]	Random setup cost, random production cost, and random inventory cost

These predefined subdomains are selected points in the entire continuous space of problem instances. We use these predefined subdomains as sample points to learn new heuristics and verify the performance of heuristics learned on the rest of the problem space.

5.7.3. Experimental parameters

To compare the minimum-risk strategy and the round-robin strategy, we perform three experiments for each strategy on learning decomposition heuristics where each experiment starts with a different random seed. These random seeds used are 61801, 92010, and 94035. For experiments on learning the other heuristics, we only perform one experiment per strategy with random seed 61801.

For all these experiments, we learn heuristics on three subdomains and generalize good heuristics learned to the other subdomains. Therefore, the entire learning process contains three learning phases and three verification phases. In a learning phase, there are ten generations. The number of generations is small because evaluating test cases is time consuming and is prohibitively expensive to have a large number of generations. In each generation, we consider 40 heuristics. The percentage of good heuristics retained from the previous generation to the current is 0.25. During a generation, we limit the number of test cases evaluated to be 160.

In generating new heuristics, the cross-over operator is applied with a probability of 1/3, the mutation operator, 1/6, the coalescing operator, 1/6, and the random generation operator, 1/3. The random operator is used to allow us explore space outside the local optima, and the coalescing operator is used to increase the size of heuristics. The experimental parameters are summarized in Table 5.6 for easy reference.

Table 5.6. Summary of all learning parameters used in this thesis, where $N_{r/s}$ is the number of experiments run per strategy (the minimum-risk strategy or the round-robin strategy), $T_{d,L}$ is the time constraint (number of tests allowed) for all learning phases, $N_{g/d}$ is the number of generations per learning phase, $T_{g,d}$ is the time constraint (number of tests allowed) in a generation, $N_{h/g}$ is the number of heuristics per generation, N_d is the total number of predefined subdomains, $N_{d,L}$ is the number of subdomains to learn heuristics on, and $N_{d,G}$ is the total number of subdomains to generalize in all verification phases.

Type of Heuristics	Application Problem	$N_{r/s}$	$T_{d,L}$	$N_{g/d}$	$T_{g/d}$	$N_{h/g}$	N_d	$N_{d,L}$	$N_{d,G}$
$H_{D,sos,gdfs}$	KS	3	1600	10	160	40	12	3	9
	VC								
	TSP								
$H_{S,sos,gdfs}$	KS	1	1600	10	160	40	12	3	9
	VC								
	PP								
	TSP						3		0
$H_{S,ffs,dbs}$	TSP	1	1600	10	160	40	9	3	6
	Maze						14		11
$H_{S,sos,dbs}$	TSP	1	1600	10	160	40	9	3	6
	Maze						14		11
$H_{P,ffs,dbs}$	TSP	1	1600	10	160	40	9	3	6
	Maze						14		11

5.7.4. Parameters used in heuristics

In this section, we describe the potential parameters used in heuristics to be learned. For different types of heuristics, we may use different sets of parameters. Tables 5.7, 5.8, and 5.9 show the parameters used for constructing decomposition heuristics that are used by GDFSs for solving the VC, TSP, and KS problems, respectively.

Tables 5.10, 5.11, 5.12, and 5.13 show the parameters for constructing selection heuristics that are used by GDFSs for solving the VC, TSP, KS, and PP problems, respectively.

Tables 5.14 and 5.15 show the parameters used for constructing selection heuristics that are used by dynamic band searches for solving the TSP and maze problems, respectively.

Table 5.7. Parameters used for constructing decomposition heuristics used by GDFSs for solving the VC problem.

Symbol	Meaning
f	degree of vertex selected in terms of number of edges connected
l	live degree of this vertex, that is, the degree in terms of uncovered edges
d	dead degree of this vertex, that is, the degree in terms of covered edges
n	average live degree of all neighbors of this vertex
Δ l	difference of the live degrees of this vertex in the parent search node and the current search node

Table 5.8. Parameters used for constructing decomposition heuristics used by GDFSs for solving the TSP.

Symbol	Meaning
c	length of the road connecting this city and the terminal city of the current partial tour
m	minimum length of the roads connecting unvisited cities and the terminal city of the current partial tour
a	average length of the roads connecting unvisited cities and the terminal city of the current partial tour
l	the number of neighboring cities that have not been visited
d	the number of neighboring cities that have been visited

Table 5.9. Parameters used for constructing decomposition heuristics used by GDFSs for solving the KS problem.

Symbol	Meaning
p	profit of this object
w	weight of this object
s	weight slack, defined as the weight bound minus the current weight in the sack
$p_{\max}$	maximum profit of unselected objects
$w_{\max}$	maximum weight of unselected objects
$p_{\min}$	minimum profit of unselected objects
$w_{\min}$	minimum weight of unselected objects

Table 5.10. Parameters used for constructing selection heuristics used by GDFSs for solving the VC problem.

Symbol	Meaning
l	lower bound of this search node
u	upper bound of this search node
g	cost from the root node to this node
h	underestimated cost from this node to an optimal solution; note that $h =del\ l - g$
Δ u	$=del\ u_{parent} - u_{this}$
Δ l	$=del\ l_{this} - l_{parent}$
Δ g	$=del\ l_{this} - l_{parent}$
Δ h	$=del\ h_{parent} - h_{this}$
h_1	degree of the vertex selected by this node in terms of connecting edges
h_2	live degree of the vertex selected by this node, that is, the degree in terms of uncovered edges
h_3	dead degree of the vertex selected by this node, that is, the degree in terms of covered edges
h_4	average live degree of all neighbors of the vertex selected by this node

Table 5.11. Parameters used for constructing selection heuristics used by GDFSs for solving the TSP.

Symbol	Meaning
l	lower bound of this search node
u	upper bound of this search node
g	cost from the root node to this node
h	underestimated cost from this node to an optimal solution; note that $h = del\ l - g$
Δ u	$=del\ u_{parent} - u_{this}$
Δ l	$=del\ l_{this} - l_{parent}$
Δ g	$=del\ l_{this} - l_{parent}$
Δ h	$=del\ h_{parent} - h_{this}$
h_1	minimum length of the roads (perhaps, a tree) connecting unvisited cities and the terminal city of the current partial tour
h_2	average length of the roads (perhaps, a tree) connecting unvisited cities and the terminal city of the current partial tour
h_3	the number of neighboring cities that have not been visited
h_4	the number of neighboring cities that have been visited

Table 5.12. Parameters used for constructing selection heuristics used by GDFSs for solving the KS problem.

Symbol	Meaning
l	lower bound of this search node
u	upper bound of this search node
g	cost from the root node to this node
h	underestimated cost from this node to an optimal solution; note that $h = del\ l - g$
Δ u	$=del\ u_{parent} - u_{this}$
Δ l	$=del\ l_{this} - l_{parent}$
Δ g	$=del\ l_{this} - l_{parent}$
Δ h	$=del\ h_{parent} - h_{this}$
h_1	profit of the object selected by this node
h_2	weight of the object selected by this node
h_3	weight slack, defined as the weight bound minus the current weight in the sack

Table 5.13. Parameters used for constructing selection heuristics used by GDFSs for solving the PP problem.

Symbol	Meaning
l	lower bound of this search node
u	upper bound of this search node
g	cost from the root node to this node
h	underestimated cost from this node to an optimal solution; note that $h = del\ l - g$
Δ u	$= del\ u_{parent} - u_{this}$
Δ l	$= del\ l_{this} - l_{parent}$
Δ g	$= del\ l_{this} - l_{parent}$
Δ h	$= del\ h_{parent} - h_{this}$

Table 5.14. Parameters used for constructing selection heuristics used by dynamic band searches for solving the TSP.

Symbol	Meaning
l	lower bound of this search node
u	upper bound of this search node
g	cost from the root node to this node
h	underestimated cost from this node to an optimal solution; note that $h = del\ l - g$
Δ u	$= del\ u_{parent} - u_{this}$
Δ l	$= del\ l_{this} - l_{parent}$
Δ g	$= del\ l_{this} - l_{parent}$
Δ h	$= del\ h_{parent} - h_{this}$
h_1	minimum length of the roads (perhaps, a tree) connecting unvisited cities and the terminal city of the current partial tour
h_2	average length of the roads (perhaps, a tree) connecting unvisited cities and the terminal city of the current partial tour
h_3	the number of neighboring cities that have not been visited
h_4	the number of neighboring cities that have been visited

Table 5.15. Parameters used for constructing selection heuristics used by dynamic band searches for solving the maze problem.

Symbol	Meaning
l	lower bound of this search node
g	cost from the root node to this node
h	underestimated cost from this node to an optimal solution; note that $h = del\ l - g$
Δ l	$= del\ l_{this} - l_{parent}$
Δ h	$= del\ h_{parent} - h_{this}$
h_1	number of move alternatives at the current position
h_2	Manhattan distance between the entrance and the current position

5.7.5. Comparison of scheduling strategies

In this section, we empirically compare two scheduling strategies: the minimum-risk (MR) strategy and the round-robin (RR) strategy. To make the comparison, we run all the learning experiments outlined in Table 5.6.

There are two possible criteria for comparison: (1) the comparison based on the best heuristics of each subdomain in the best run from each of the strategies, and (2) the comparison based on the best heuristics of each subdomain out of all runs of the strategies. Note that these two criteria are equivalent when there is only one run per strategy. Therefore, in our experiments, these two criteria have different conclusions only in learning decomposition heuristics in which we have multiple runs.

The empirical comparison of the MR and RR strategies is summarized in Table 5.16, where the performance measure in this comparison is the average probability of wins of all the best heuristics in each subdomain. To have a balanced view, we report the average probabilities of wins for learned subdomains, generalized subdomains, and all subdomains.

Note that the learning of selection heuristics used in the guided depth-first search for solving the TSP contains three subdomains that are learned; therefore, there is no generalized subdomain in the experiment, and we denote it as NA.

Table 5.16. Comparison of the minimum-risk strategy and the round-robin strategy on various learning experiments, where the comparison is based on the average probability of wins of the best heuristics in each subdomain.

Comparison Based on Best Run of Strategy (Criterion (1))							
Type of Heuristics	Problem	Learned Subdomains		Generalized Subdomains		All Subdomains	
		MR	RR	MR	RR	MR	RR
$H_{D,sos,gdfs}$	KS	.567	.433	.399	.601	.441	.559
	VC	.542	.458	.500	.500	.509	.491
	TSP	.408	.592	.472	.528	.456	.543
$H_{S,sos,gdfs}$	KS	.582	.418	.570	.430	.573	.427
	VC	.582	.418	.557	.443	.563	.437
	PP	.444	.556	.487	.513	.476	.524
	TSP	.520	.480	NA	NA	.520	.480
$H_{S,ffs,dbs}$	TSP	.456	.544	.483	.517	.474	.526
	Maze	.569	.431	.433	.567	462	.538
$H_{S,sos,dbs}$	TSP	.498	.510	.500	.500	.497	.503
	Maze	.561	.439	.426	.574	.455	.545
$H_{P,ffs,dbs}$	TSP	.471	.529	.233	.767	.312	.688
	Maze	.760	.240	.419	.581	.492	.508
Comparison Based on Strategy's Best Heuristic per Subdomain (Criterion (2))							
$H_{D,sos,gdfs}$	KS	.641	.359	.470	.530	.513	.487
	VC	.510	.490	.508	.492	.509	.491
	TSP	.477	.523	.519	.481	.508	.492

The direct effect of the strategy depends on the learned subdomains. We find that there is no single one that can win over the others all the time. We also find that the RR strategy works better for a particular set of problems such as the TSP, regardless of which heuristic is being learned. The RR strategy also performs better in learning decomposition heuristics of the PP problem. For others, the MR strategy performs better. We find that the MR strategy may generate heuristics that are very special for the particular subdomain being learned on; for example, the probability of wins for learning $H_{P,ffs,dbs}$ of the maze problem is 0.760 when considering the learned subdomains, but the probability of wins becomes 0.492 when considering all subdomains. This happens especially when the performance of heuristics varies significantly such that we have to do more tests on heuristics in order to distinguish them. The MR strategy will concentrate testing good heuristics and take into account the variance of performance at the same time.

We also find that the results based on these criteria are agreeable on learned subdomains but not agreeable on all subdomains. In fact, both criteria are valid and the decision on which one to use is dependent on how learning experiments are run. We should use the first criterion when we are concerned about the performance of a single run of learning, and we compare the strategies based on a complete run. On the other hand, we use the second criterion when we emphasize the performance of multiple runs of learning. In this case, we can collect the best heuristic for each subdomain among all runs.

In short, we find that the MR strategy generally outperforms the RR strategy on the KS, the VC, and the maze problems, while the RR strategy generally outperforms the MR strategy on the TSP and the PP problems.

5.7.6. Learned real-time search

In this section, we will empirically demonstrate the improvement of RTS with the new heuristics learned.

To demonstrate the heuristics learned, we need to obtain the appropriate clustering of subdomains. Note that the clusters found are based on the consistency in ranking of heuristics. Therefore, it is possible for two clusters to conclude on the same best heuristic. Moreover, the subdomain concluding the baseline heuristic as the best may be grouped with other subdomains. To deal with these problems, the procedure of obtaining the appropriate clustering of subdomains is as follows.

(1) Group subdomains into a cluster in which the best heuristic is the baseline heuristic.

(2) Examine the cluster plot and choose clusters with c.r.c. around 0.75 ~ 0.95 such that those clusters have the largest area.

(3) Group the clusters that suggest the same best heuristics.

(4) Examine the speedups of the best heuristics suggested in all clusters and replace the best heuristic with the baseline one when its speedup is less than 1.02; in such a case, the superiority of the learned heuristic is small and may be due to noise.

There are two extreme cluster sets: one in which the most generalized cluster set groups all subdomains into a single cluster, and the other in which each subdomain is a cluster by itself.

The decomposition heuristics are obtained by running our learning experiments with the MR strategy and a random seed of 94035, except that the results for the TSP are obtained using the RR strategy and random seed of 61801. The selection heuristics are obtained by running our learning experiments with the MR strategy and random seed of 61801, except that the results for the TSP are obtained using the RR strategy and random seed of 61801.

The cluster plots of the learning results for decomposition heuristics and selection heuristics of the KS, TSP, VC, and PP problems are shown in Appendix B. Besides the cluster plots, we also show the most generalized cluster set, the most specialized cluster set, and the chosen cluster set.

In the following, we show the performance of the learned RTS in terms of both EAD and AAD for various time constraints. Remember that smaller AADs and EADs mean better performance. The clusters of subdomains and their best heuristics in terms of the probability of win are summarized in Appendix B.

In Tables 5.17-5.20, we show the improvement of the learned RTS over the original RTS for solving the VC problems and can see the improvement of the learned RTS in terms of both EAD and AAD.

In Table 5.21, we show the performance of the learned RTS for solving the TSPs. The learned RTS improves the original RTS in subdomain 1 only. In subdomains 5 and 9, we do not learned anything better; therefore, we reject the heuristics learned on subdomains 5 and 9. This happens when the evaluation criterion used is different from that used in learning. In RTS, we use both EAD and AAD to evaluate search algorithms over various time constraints. However, during learning, it is hard to learn heuristics over various time

constraints in terms of both EAD and AAD. Note that in learning we attempt to find the best heuristics in terms of least execution time spent in solving an optimal solution.

Table 5.17. RTS for solving the VC problems of subdomains 1, 2, and 3.

Performance Comparison Based on Normalized AAD Average									
Subdomain	1			2			3		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.712	0.410	0.137	0.690	0.392	0.031	0.768	0.464	0.259
0.50	0.712	0.358	0.117	0.690	0.378	0.031	0.768	0.402	0.194
0.55	0.712	0.330	0.072	0.690	0.358	0.014	0.768	0.378	0.124
0.60	0.712	0.294	0.035	0.690	0.323	0.014	0.768	0.326	0.098
0.65	0.712	0.250	0.011	0.690	0.296	0.000	0.768	0.266	0.061
0.70	0.712	0.199	0.000	0.690	0.216	0.000	0.768	0.245	0.031
0.75	0.712	0.141	0.000	0.690	0.123	0.000	0.760	0.139	0.011
0.80	0.689	0.107	0.000	0.690	0.123	0.000	0.705	0.093	0.000
0.85	0.656	0.041	0.000	0.690	0.123	0.000	0.665	0.083	0.000
0.90	0.596	0.033	0.000	0.625	0.116	0.000	0.535	0.035	0.000
Performance Comparison Based on Normalized EAD Average									
Subdomain	1			2			3		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.183	0.083	0.035	0.401	0.207	0.014	0.245	0.131	0.074
0.50	0.183	0.078	0.016	0.401	0.202	0.014	0.245	0.117	0.054
0.55	0.183	0.065	0.004	0.401	0.187	0.000	0.245	0.100	0.049
0.60	0.183	0.057	0.004	0.401	0.144	0.000	0.245	0.079	0.025
0.65	0.183	0.049	0.000	0.401	0.138	0.000	0.245	0.079	0.014
0.70	0.183	0.049	0.000	0.401	0.077	0.000	0.245	0.067	0.009
0.75	0.183	0.020	0.000	0.401	0.000	0.000	0.240	0.029	0.000
0.80	0.169	0.000	0.000	0.401	0.000	0.000	0.203	0.015	0.000
0.85	0.145	0.000	0.000	0.401	0.000	0.000	0.177	0.009	0.000
0.90	0.115	0.000	0.000	0.348	0.000	0.000	0.097	0.004	0.000

Table 5.18. RTS for solving the VC problems of subdomains 4, 5, and 6.

Performance Comparison Based on Normalized AAD Average									
Subdomain	4			5			6		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.720	0.449	0.248	0.787	0.519	0.328	0.821	0.578	0.351
0.50	0.720	0.441	0.165	0.787	0.460	0.249	0.821	0.515	0.322
0.55	0.720	0.341	0.130	0.787	0.424	0.214	0.821	0.506	0.264
0.60	0.720	0.267	0.076	0.787	0.389	0.142	0.821	0.436	0.221
0.65	0.720	0.259	0.057	0.787	0.338	0.094	0.821	0.345	0.157
0.70	0.720	0.208	0.023	0.787	0.287	0.060	0.821	0.325	0.096
0.75	0.683	0.139	0.000	0.744	0.196	0.015	0.754	0.212	0.018
0.80	0.650	0.088	0.000	0.702	0.159	0.007	0.746	0.177	0.000
0.85	0.607	0.053	0.000	0.632	0.126	0.000	0.662	0.137	0.000
0.90	0.530	0.053	0.000	0.521	0.065	0.000	0.592	0.087	0.000
Performance Comparison Based on Normalized EAD Average									
Subdomain	4			5			6		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.203	0.096	0.084	0.199	0.137	0.076	0.170	0.112	0.089
0.50	0.203	0.096	0.060	0.199	0.121	0.069	0.170	0.112	0.082
0.55	0.203	0.062	0.036	0.199	0.112	0.054	0.170	0.106	0.073
0.60	0.203	0.057	0.017	0.199	0.102	0.024	0.170	0.076	0.044
0.65	0.203	0.049	0.017	0.199	0.081	0.019	0.170	0.051	0.030
0.70	0.203	0.016	0.000	0.199	0.069	0.007	0.170	0.051	0.012
0.75	0.178	0.011	0.000	0.184	0.048	0.007	0.128	0.023	0.000
0.80	0.156	0.005	0.000	0.167	0.031	0.000	0.123	0.023	0.000
0.85	0.127	0.005	0.000	0.122	0.031	0.000	0.079	0.013	0.000
0.90	0.085	0.000	0.000	0.046	0.010	0.000	0.051	0.006	0.000

Table 5.19. RTS for solving the VC problems of subdomains 7, 8, and 9.

Performance Comparison Based on Normalized AAD Average									
Subdomain	7			8			9		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.785	0.574	0.420	0.897	0.599	0.423	0.895	0.645	0.430
0.50	0.785	0.506	0.360	0.897	0.532	0.394	0.895	0.578	0.412
0.55	0.785	0.458	0.343	0.897	0.504	0.350	0.895	0.529	0.332
0.60	0.785	0.414	0.293	0.897	0.474	0.284	0.895	0.500	0.323
0.65	0.785	0.378	0.259	0.897	0.422	0.231	0.895	0.408	0.245
0.70	0.769	0.360	0.175	0.897	0.355	0.180	0.895	0.361	0.237
0.75	0.769	0.249	0.059	0.865	0.254	0.061	0.862	0.247	0.093
0.80	0.731	0.197	0.038	0.825	0.215	0.033	0.846	0.226	0.071
0.85	0.685	0.143	0.000	0.808	0.171	0.004	0.771	0.165	0.011
0.90	0.575	0.091	0.000	0.615	0.121	0.000	0.606	0.150	0.000
Performance Comparison Based on Normalized EAD Average									
Subdomain	7			8			9		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.167	0.128	0.132	0.147	0.089	0.079	0.139	0.102	0.077
0.50	0.167	0.104	0.126	0.147	0.077	0.079	0.139	0.093	0.077
0.55	0.167	0.090	0.121	0.147	0.077	0.057	0.139	0.078	0.077
0.60	0.167	0.085	0.106	0.147	0.072	0.051	0.139	0.063	0.073
0.65	0.167	0.075	0.079	0.147	0.057	0.042	0.139	0.047	0.052
0.70	0.167	0.065	0.056	0.147	0.040	0.022	0.139	0.037	0.048
0.75	0.167	0.037	0.013	0.128	0.022	0.006	0.129	0.005	0.024
0.80	0.142	0.028	0.000	0.103	0.012	0.000	0.119	0.005	0.000
0.85	0.113	0.015	0.000	0.093	0.006	0.000	0.085	0.005	0.000
0.90	0.053	0.006	0.000	0.030	0.006	0.000	0.028	0.005	0.000

Table 5.20. RTS for solving the VC problems of subdomains 10, 11, and 12.

Performance Comparison Based on Normalized AAD Average									
Subdomain	10			11			12		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.819	0.596	0.507	0.856	0.658	0.539	0.919	0.730	0.607
0.50	0.819	0.577	0.471	0.856	0.609	0.508	0.919	0.649	0.506
0.55	0.819	0.529	0.386	0.856	0.567	0.422	0.919	0.603	0.462
0.60	0.819	0.480	0.362	0.856	0.505	0.413	0.919	0.591	0.462
0.65	0.819	0.461	0.311	0.856	0.484	0.364	0.919	0.544	0.396
0.70	0.819	0.424	0.262	0.856	0.429	0.298	0.919	0.447	0.368
0.75	0.811	0.333	0.167	0.856	0.342	0.203	0.919	0.329	0.261
0.80	0.802	0.272	0.128	0.847	0.278	0.153	0.893	0.311	0.220
0.85	0.802	0.248	0.071	0.772	0.240	0.090	0.834	0.278	0.164
0.90	0.667	0.166	0.016	0.634	0.175	0.039	0.666	0.223	0.090
Performance Comparison Based on Normalized EAD Average									
Subdomain	10			11			12		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.089	0.065	0.059	0.118	0.093	0.084	0.108	0.092	0.061
0.50	0.089	0.060	0.059	0.118	0.078	0.084	0.108	0.076	0.061
0.55	0.089	0.060	0.054	0.118	0.067	0.062	0.108	0.076	0.061
0.60	0.089	0.060	0.054	0.118	0.062	0.062	0.108	0.066	0.061
0.65	0.089	0.060	0.043	0.118	0.062	0.057	0.108	0.056	0.053
0.70	0.089	0.054	0.038	0.118	0.046	0.047	0.108	0.035	0.042
0.75	0.084	0.038	0.028	0.118	0.037	0.031	0.108	0.010	0.036
0.80	0.078	0.032	0.012	0.113	0.026	0.031	0.093	0.010	0.021
0.85	0.078	0.027	0.000	0.068	0.022	0.026	0.057	0.010	0.021
0.90	0.031	0.016	0.000	0.027	0.011	0.005	0.027	0.010	0.011

Table 5.21. RTS for solving the TSP of subdomains 1, 5, and 9.

Performance Comparison Based on Normalized AAD Average									
Subdomain	1			5			9		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.547	0.460	0.476	0.708	0.583	0.641	0.652	0.556	0.540
0.50	0.547	0.384	0.417	0.708	0.569	0.574	0.652	0.504	0.492
0.55	0.547	0.364	0.363	0.708	0.541	0.553	0.652	0.488	0.486
0.60	0.530	0.343	0.336	0.708	0.440	0.540	0.652	0.397	0.435
0.65	0.523	0.316	0.330	0.708	0.369	0.437	0.643	0.366	0.374
0.70	0.519	0.247	0.261	0.702	0.372	0.364	0.641	0.321	0.336
0.75	0.491	0.184	0.157	0.666	0.233	0.264	0.615	0.227	0.236
0.80	0.474	0.125	0.129	0.633	0.177	0.212	0.567	0.165	0.198
0.85	0.403	0.103	0.093	0.586	0.167	0.185	0.520	0.141	0.112
0.90	0.352	0.079	0.065	0.536	0.115	0.131	0.451	0.099	0.105
Performance Comparison Based on Normalized EAD Average									
Subdomain	1			5			9		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.177	0.128	0.135	0.149	0.142	0.141	0.147	0.110	0.075
0.50	0.177	0.107	0.126	0.149	0.133	0.133	0.147	0.061	0.065
0.55	0.177	0.069	0.071	0.149	0.119	0.128	0.147	0.058	0.063
0.60	0.161	0.050	0.047	0.149	0.105	0.094	0.147	0.048	0.048
0.65	0.155	0.046	0.045	0.149	0.083	0.084	0.140	0.032	0.035
0.70	0.151	0.030	0.026	0.144	0.056	0.084	0.138	0.023	0.024
0.75	0.127	0.005	0.003	0.116	0.016	0.016	0.122	0.012	0.012
0.80	0.113	0.001	0.001	0.089	0.010	0.013	0.084	0.008	0.012
0.85	0.063	0.000	0.000	0.058	0.010	0.013	0.054	0.007	0.007
0.90	0.032	0.000	0.000	0.022	0.008	0.007	0.032	0.000	0.000

In Tables 5.22-5.25, we show the performance of the learned RTS for solving the KS problems. We find that the learned RTS has improvement on most subdomains, except subdomain 7. In subdomain 11, the learned RTS is better than the original RTS for some large time constraints. In subdomain 9, EADs achieved by the learned RTS is better than those by the original RTS; on the other hand, AADs achieved by the learned RTS is worse than those by the original RTS. It is interesting to note that the polynomial-time algorithm we used for computing upper bounds of search nodes performs well in subdomains 10 and 12 because this algorithm almost always find the optimal solution, as seen in Table 5.25.

In Tables 5.26-5.29, we show the performance of the learned RTS for solving the PP problems. We find that the learned RTS has improvement on most subdomains, except subdomains 2, 9, and 12.

5.7.7. Learned dynamic-band search

In this section, we will empirically demonstrate the improvement of dynamic band search with new heuristics learned. The performance is evaluated in terms of speedup in finding a feasible solution when the goal of the search is to find a solution. On the other hand, we also use AAcD and EAcD to evaluate algorithms when we are interested in good solutions under a given time constraint. We use AAcD and EAcD instead of AAD and EAD because we may not find any solution under the time constraint and the AAD and EAD becomes infinity which makes comparison difficult. In contrast, the AAcD and EAcD are zero when there is no solution. Recall that larger AAcDs and EAcDs mean better performance.

Table 5.22. RTS for solving the KS problems of subdomains 1, 2, and 3.

Performance Comparison Based on Normalized AAD Average									
Subdomain	1			2			3		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.761	0.640	0.640	0.445	0.212	0.212	0.510	0.359	0.357
0.50	0.760	0.570	0.570	0.445	0.190	0.190	0.510	0.324	0.322
0.55	0.759	0.530	0.530	0.445	0.160	0.160	0.510	0.296	0.294
0.60	0.759	0.500	0.500	0.445	0.160	0.160	0.509	0.279	0.270
0.65	0.757	0.480	0.480	0.444	0.130	0.120	0.509	0.263	0.246
0.70	0.757	0.440	0.440	0.444	0.120	0.120	0.508	0.237	0.232
0.75	0.754	0.379	0.379	0.444	0.097	0.095	0.503	0.188	0.126
0.80	0.746	0.330	0.330	0.443	0.087	0.085	0.498	0.171	0.085
0.85	0.733	0.303	0.270	0.438	0.078	0.070	0.480	0.134	0.029
0.90	0.723	0.227	0.227	0.435	0.078	0.040	0.441	0.096	0.000
Performance Comparison Based on Normalized EAD Average									
Subdomain	1			2			3		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.037	0.050	0.048	0.002	0.002	0.002	0.130	0.100	0.081
0.50	0.036	0.037	0.038	0.002	0.002	0.001	0.130	0.083	0.066
0.55	0.035	0.037	0.036	0.002	0.002	0.001	0.130	0.082	0.056
0.60	0.035	0.037	0.031	0.002	0.002	0.001	0.130	0.075	0.042
0.65	0.034	0.035	0.029	0.002	0.002	0.001	0.130	0.071	0.038
0.70	0.033	0.032	0.025	0.002	0.002	0.001	0.129	0.065	0.035
0.75	0.031	0.030	0.022	0.002	0.002	0.000	0.123	0.047	0.020
0.80	0.023	0.025	0.022	0.001	0.001	0.000	0.119	0.040	0.003
0.85	0.012	0.022	0.017	0.001	0.001	0.000	0.100	0.029	0.000
0.90	0.002	0.016	0.014	0.000	0.001	0.000	0.062	0.006	0.000

Table 5.23. RTS for solving the KS problems of subdomains 4, 5, and 6.

Performance Comparison Based on Normalized AAD Average									
Subdomain	4			5			6		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.318	0.145	0.145	0.926	0.638	0.631	0.380	0.150	0.150
0.50	0.318	0.100	0.100	0.926	0.604	0.600	0.380	0.140	0.135
0.55	0.318	0.080	0.080	0.926	0.540	0.540	0.380	0.120	0.120
0.60	0.318	0.080	0.080	0.926	0.520	0.510	0.380	0.103	0.110
0.65	0.318	0.080	0.080	0.926	0.500	0.480	0.380	0.094	0.090
0.70	0.318	0.080	0.080	0.926	0.429	0.413	0.380	0.090	0.090
0.75	0.314	0.080	0.080	0.924	0.317	0.289	0.380	0.080	0.080
0.80	0.313	0.080	0.080	0.923	0.287	0.187	0.378	0.080	0.070
0.85	0.313	0.080	0.080	0.915	0.197	0.074	0.378	0.080	0.040
0.90	0.310	0.080	0.043	0.874	0.180	0.002	0.376	0.080	0.010
Performance Comparison Based on Normalized EAD Average									
Subdomain	4			5			6		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.001	0.001	0.001	0.077	0.049	0.049	0.001	0.001	0.001
0.50	0.001	0.001	0.001	0.077	0.049	0.049	0.001	0.001	0.001
0.55	0.001	0.001	0.001	0.077	0.049	0.047	0.001	0.001	0.001
0.60	0.001	0.001	0.000	0.077	0.047	0.045	0.001	0.001	0.001
0.65	0.001	0.001	0.000	0.077	0.047	0.045	0.001	0.001	0.000
0.70	0.001	0.001	0.000	0.077	0.045	0.033	0.001	0.001	0.000
0.75	0.000	0.000	0.000	0.076	0.036	0.021	0.001	0.001	0.000
0.80	0.000	0.000	0.000	0.075	0.034	0.008	0.001	0.001	0.000
0.85	0.000	0.000	0.000	0.067	0.024	0.001	0.001	0.001	0.000
0.90	0.000	0.000	0.000	0.047	0.022	0.000	0.000	0.000	0.000

Table 5.24. RTS for solving the KS problems of subdomains 7, 8, and 9.

Performance Comparison Based on Normalized AAD Average									
Subdomain	7			8			9		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.629	0.396	0.454	0.353	0.140	0.140	0.940	0.755	0.949
0.50	0.629	0.344	0.459	0.353	0.110	0.110	0.940	0.730	0.942
0.55	0.619	0.285	0.445	0.353	0.090	0.100	0.940	0.671	0.932
0.60	0.616	0.252	0.426	0.353	0.090	0.080	0.940	0.610	0.910
0.65	0.602	0.235	0.427	0.353	0.070	0.070	0.940	0.550	0.886
0.70	0.557	0.200	0.385	0.353	0.070	0.070	0.940	0.510	0.868
0.75	0.525	0.155	0.318	0.353	0.070	0.070	0.940	0.434	0.827
0.80	0.494	0.103	0.304	0.353	0.070	0.060	0.929	0.340	0.778
0.85	0.441	0.103	0.266	0.301	0.070	0.020	0.910	0.339	0.713
0.90	0.383	0.072	0.252	0.301	0.070	0.010	0.877	0.300	0.633
Performance Comparison Based on Normalized EAD Average									
Subdomain	7			8			9		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.271	0.143	0.109	0.001	0.001	0.001	0.013	0.013	0.009
0.50	0.271	0.111	0.109	0.001	0.001	0.001	0.013	0.013	0.009
0.55	0.262	0.099	0.109	0.001	0.001	0.001	0.013	0.013	0.008
0.60	0.258	0.078	0.109	0.001	0.001	0.001	0.013	0.013	0.008
0.65	0.245	0.070	0.109	0.001	0.001	0.000	0.013	0.013	0.008
0.70	0.243	0.067	0.071	0.001	0.001	0.000	0.013	0.013	0.006
0.75	0.211	0.039	0.048	0.001	0.001	0.000	0.013	0.012	0.006
0.80	0.180	0.028	0.042	0.001	0.001	0.000	0.012	0.011	0.006
0.85	0.128	0.018	0.026	0.001	0.001	0.000	0.011	0.010	0.006
0.90	0.075	0.004	0.022	0.001	0.001	0.000	0.008	0.009	0.006

Table 5.25. RTS for solving the KS problems of subdomains 10, 11, and 12.

Performance Comparison Based on Normalized AAD Average									
Subdomain	10			11			12		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.247	0.088	0.088	0.882	0.707	0.831	0.291	0.180	0.180
0.50	0.247	0.078	0.068	0.882	0.688	0.820	0.291	0.120	0.120
0.55	0.247	0.068	0.048	0.882	0.660	0.817	0.291	0.090	0.080
0.60	0.247	0.048	0.048	0.882	0.630	0.782	0.291	0.060	0.060
0.65	0.247	0.048	0.048	0.882	0.595	0.768	0.291	0.060	0.060
0.70	0.247	0.048	0.048	0.882	0.574	0.715	0.291	0.060	0.060
0.75	0.247	0.048	0.048	0.878	0.486	0.567	0.291	0.060	0.050
0.80	0.247	0.048	0.028	0.874	0.433	0.348	0.291	0.060	0.040
0.85	0.247	0.048	0.028	0.856	0.350	0.196	0.291	0.060	0.030
0.90	0.173	0.048	0.000	0.700	0.286	0.125	0.291	0.060	0.000
Performance Comparison Based on Normalized EAD Average									
Subdomain	10			11			12		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.000	0.000	0.000	0.331	0.221	0.282	0.000	0.000	0.000
0.50	0.000	0.000	0.000	0.331	0.220	0.276	0.000	0.000	0.000
0.55	0.000	0.000	0.000	0.331	0.206	0.276	0.000	0.000	0.000
0.60	0.000	0.000	0.000	0.331	0.195	0.270	0.000	0.000	0.000
0.65	0.000	0.000	0.000	0.331	0.178	0.261	0.000	0.000	0.000
0.70	0.000	0.000	0.000	0.331	0.165	0.239	0.000	0.000	0.000
0.75	0.000	0.000	0.000	0.327	0.139	0.191	0.000	0.000	0.000
0.80	0.000	0.000	0.000	0.323	0.122	0.042	0.000	0.000	0.000
0.85	0.000	0.000	0.000	0.306	0.043	0.014	0.000	0.000	0.000
0.90	0.000	0.000	0.000	0.152	0.034	0.005	0.000	0.000	0.000

Table 5.26. RTS for solving the PP problems of subdomains 1, 2, and 3.

Performance Comparison Based on Normalized AAD Average									
Subdomain	1			2			3		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.565	0.287	0.285	0.819	0.527	0.539	0.378	0.224	0.220
0.50	0.556	0.238	0.230	0.818	0.480	0.480	0.373	0.210	0.209
0.55	0.549	0.223	0.223	0.818	0.448	0.448	0.369	0.175	0.179
0.60	0.541	0.213	0.213	0.814	0.375	0.388	0.359	0.151	0.150
0.65	0.535	0.183	0.175	0.814	0.370	0.370	0.357	0.123	0.123
0.70	0.521	0.128	0.127	0.811	0.292	0.298	0.348	0.102	0.106
0.75	0.509	0.099	0.112	0.797	0.224	0.219	0.328	0.083	0.079
0.80	0.489	0.078	0.081	0.793	0.201	0.195	0.316	0.070	0.070
0.85	0.456	0.071	0.066	0.770	0.118	0.129	0.288	0.070	0.070
0.90	0.391	0.070	0.066	0.679	0.108	0.107	0.222	0.061	0.044
Performance Comparison Based on Normalized EAD Average									
Subdomain	1			2			3		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.042	0.033	0.025	0.013	0.017	0.017	0.029	0.032	0.028
0.50	0.038	0.024	0.016	0.013	0.015	0.016	0.027	0.028	0.022
0.55	0.035	0.021	0.011	0.013	0.015	0.016	0.026	0.024	0.021
0.60	0.032	0.015	0.011	0.011	0.008	0.015	0.023	0.022	0.019
0.65	0.030	0.014	0.006	0.011	0.006	0.013	0.022	0.021	0.016
0.70	0.023	0.010	0.003	0.010	0.006	0.007	0.019	0.019	0.012
0.75	0.018	0.005	0.002	0.004	0.006	0.005	0.013	0.011	0.002
0.80	0.010	0.003	0.002	0.002	0.006	0.005	0.009	0.005	0.001
0.85	0.009	0.002	0.000	0.000	0.003	0.004	0.007	0.004	0.001
0.90	0.005	0.000	0.000	0.000	0.000	0.003	0.003	0.001	0.001

Table 5.27. RTS for solving the PP problems of subdomains 4, 5, and 6.

Performance Comparison Based on Normalized AAD Average									
Subdomain	4			5			6		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.462	0.297	0.290	0.453	0.298	0.288	0.340	0.208	0.208
0.50	0.459	0.250	0.250	0.438	0.254	0.254	0.332	0.180	0.180
0.55	0.457	0.240	0.240	0.431	0.229	0.219	0.327	0.158	0.158
0.60	0.449	0.193	0.194	0.422	0.197	0.200	0.324	0.126	0.117
0.65	0.443	0.164	0.164	0.417	0.153	0.145	0.321	0.105	0.105
0.70	0.437	0.149	0.147	0.410	0.133	0.135	0.313	0.101	0.101
0.75	0.422	0.099	0.089	0.401	0.089	0.104	0.303	0.081	0.081
0.80	0.416	0.082	0.083	0.389	0.089	0.104	0.287	0.075	0.075
0.85	0.409	0.075	0.075	0.377	0.081	0.096	0.285	0.063	0.052
0.90	0.350	0.069	0.066	0.308	0.067	0.058	0.227	0.039	0.038
Performance Comparison Based on Normalized EAD Average									
Subdomain	4			5			6		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.023	0.023	0.022	0.040	0.055	0.037	0.020	0.023	0.022
0.50	0.021	0.022	0.022	0.034	0.043	0.028	0.017	0.019	0.015
0.55	0.020	0.018	0.017	0.030	0.032	0.022	0.015	0.014	0.012
0.60	0.017	0.016	0.015	0.026	0.029	0.020	0.014	0.011	0.008
0.65	0.015	0.015	0.011	0.024	0.025	0.014	0.013	0.007	0.005
0.70	0.012	0.013	0.009	0.021	0.017	0.011	0.010	0.006	0.004
0.75	0.007	0.005	0.003	0.017	0.007	0.003	0.007	0.004	0.003
0.80	0.005	0.002	0.002	0.013	0.006	0.003	0.006	0.003	0.001
0.85	0.004	0.002	0.000	0.008	0.002	0.000	0.005	0.002	0.001
0.90	0.001	0.000	0.000	0.003	0.001	0.000	0.004	0.001	0.000

Table 5.28. RTS for solving the PP problems of subdomains 7, 8, and 9.

Performance Comparison Based on Normalized AAD Average									
Subdomain	7			8			9		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.402	0.243	0.242	0.667	0.596		0.773	0.537	0.540
0.50	0.401	0.215	0.215	0.644	0.536		0.772	0.479	0.479
0.55	0.393	0.207	0.184	0.629	0.490		0.770	0.403	0.414
0.60	0.390	0.143	0.142	0.620	0.440		0.768	0.366	0.368
0.65	0.380	0.136	0.135	0.611	0.413		0.764	0.329	0.331
0.70	0.374	0.112	0.105	0.604	0.382		0.763	0.288	0.278
0.75	0.357	0.074	0.072	0.585	0.293		0.759	0.202	0.218
0.80	0.344	0.074	0.072	0.578	0.236		0.753	0.182	0.186
0.85	0.336	0.074	0.072	0.542	0.179		0.748	0.182	0.174
0.90	0.325	0.052	0.039	0.458	0.159		0.699	0.118	0.121
Performance Comparison Based on Normalized EAD Average									
Subdomain	7			8			9		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.024	0.035	0.030	0.004	0.006		0.017	0.040	0.044
0.50	0.024	0.032	0.026	0.004	0.006		0.016	0.032	0.035
0.55	0.022	0.025	0.020	0.003	0.005		0.015	0.025	0.027
0.60	0.021	0.021	0.016	0.002	0.004		0.014	0.022	0.024
0.65	0.017	0.016	0.011	0.001	0.004		0.011	0.017	0.015
0.70	0.015	0.012	0.009	0.001	0.004		0.011	0.014	0.011
0.75	0.010	0.007	0.002	0.001	0.003		0.008	0.007	0.008
0.80	0.006	0.005	0.001	0.001	0.002		0.006	0.004	0.006
0.85	0.004	0.002	0.000	0.001	0.002		0.005	0.002	0.004
0.90	0.002	0.001	0.000	0.000	0.001		0.001	0.002	0.004

Table 5.29. RTS for solving the PP problems of subdomains 10, 11, and 12.

Performance Comparison Based on Normalized AAD Average									
Subdomain	10			11			12		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.416	0.563		0.133	0.093	0.091	0.466	0.493	0.493
0.50	0.407	0.515		0.132	0.093	0.091	0.463	0.463	0.473
0.55	0.403	0.454		0.132	0.093	0.090	0.451	0.411	0.405
0.60	0.403	0.431		0.129	0.093	0.090	0.443	0.380	0.382
0.65	0.387	0.406		0.126	0.092	0.088	0.434	0.338	0.343
0.70	0.387	0.361		0.123	0.091	0.088	0.421	0.281	0.288
0.75	0.383	0.286		0.118	0.090	0.088	0.400	0.215	0.251
0.80	0.380	0.247		0.107	0.088	0.087	0.383	0.183	0.214
0.85	0.372	0.184		0.100	0.083	0.085	0.378	0.168	0.157
0.90	0.360	0.151		0.090	0.077	0.077	0.361	0.109	0.117
Performance Comparison Based on Normalized EAD Average									
Subdomain	10			11			12		
Time	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]	GDFS	RTS[FR,I]	LRTS[FR,I]
0.45	0.039	0.206		0.005	0.005	0.002	0.019	0.102	0.105
0.50	0.030	0.173		0.005	0.004	0.002	0.018	0.096	0.096
0.55	0.027	0.125		0.005	0.004	0.002	0.016	0.082	0.087
0.60	0.027	0.125		0.004	0.004	0.002	0.014	0.071	0.075
0.65	0.012	0.122		0.004	0.003	0.001	0.012	0.054	0.062
0.70	0.012	0.104		0.003	0.003	0.001	0.011	0.039	0.051
0.75	0.008	0.077		0.002	0.002	0.001	0.010	0.016	0.035
0.80	0.007	0.057		0.001	0.001	0.000	0.006	0.016	0.026
0.85	0.002	0.022		0.001	0.001	0.000	0.006	0.000	0.009
0.90	0.000	0.022		0.000	0.000	0.000	0.005	0.000	0.005

In Tables 5.30 and 5.31, we show the performance of the learned dynamic band search in finding a feasible solution of the TSPs. The performance values are average speedups and their 90% confidence intervals.

In Tables 5.32-5.34, we show the performance of the learned dynamic band search in finding good solutions of the TSPs for various time constraints. The performance is evaluated in terms of both AAcD and EAcD. We find that the learned dynamic band search is better than the original dynamic band search in most subdomains, except subdomains 8 and 11. In subdomain 12, both perform similarly.

In Tables 5.35 and 5.36, we show the performance of the learned dynamic band search in finding a feasible solution of the maze problems. The performance values are represented by average speedups and their 90% confidence intervals. We find we may not be able to find good heuristics in some subdomains.

In Tables 5.37-5.41, we show the performance of the learned dynamic band search in finding good solutions of the maze problems for various time constraints. The performance is evaluated in terms of both AAcD and EAcD. We find that the learned dynamic band search is better than the original dynamic band search in all subdomains most of the time. We find that in many cases the GDFS may find a solution very quickly, as seen in Tables 5.37-5.41. However, the GDFS may be trapped in a poor subtree and find a solution very slowly. This phenomenon cannot be seen in Tables 5.37-5.41 but can be seen in Tables 5.35 and 5.36. This happens because the dynamic band search is more consistent than the GDFS in the sense that the time for the GDFS to find a solution may vary significantly.

Table 5.30. Dynamic band search for solving a solution of the TSP of all subdomains studied.

Subdomain	Problem Size	Number of Instances	BS[d;] Average Speedup	LBS[d;] Average Speedup
2	20	50	[5.584 ± 1.835]	[8.285 ± 2.093]
3	22	50	[7.089 ± 2.013]	[10.009 ± 2.022]
4	18	50	[4.864 ± 1.790]	[5.862 ± 1.785]
6	20	50	[5.966 ± 1.738]	[6.010 ± 1.863]
7	20	50	[7.157 ± 2.053]	[8.350 ± 2.110]
8	20	50	[6.207 ± 1.985]	[6.293 ± 1.987]
10	20	50	[2.627 ± 1.140]	[6.452 ± 1.782]
11	20	50	[7.655 ± 2.054]	[6.695 ± 2.002]
12	16	50	[4.715 ± 1.600]	[4.940 ± 1.656]

Table 5.31. Dynamic band search using new pruning heuristics for solving a solution of the TSP of all subdomains studied.

Subdomain	Problem Size	Number of Instances	BS[d;] Average Speedup	LBS[d;] Average Speedup
2	20	50	[5.584 ± 1.835]	[15.015 ± 1.827]
3	22	50	[7.089 ± 2.013]	[12.287 ± 2.064]
4	18	50	[4.864 ± 1.790]	[6.878 ± 1.909]
6	20	50	[5.966 ± 1.738]	[17.285 ± 1.253]
7	20	50	[7.157 ± 2.053]	[11.953 ± 2.111]
8	20	50	[6.207 ± 1.985]	[6.271 ± 1.964]
10	20	50	[2.627 ± 1.140]	[15.608 ± 1.688]
11	20	50	[7.655 ± 2.054]	[7.167 ± 1.999]
12	16	50	[4.715 ± 1.600]	[7.099 ± 1.871]

Table 5.32. Dynamic band search for solving the TSP of subdomains 2, 3, and 4.

Performance Comparison Based on Normalized AAcD Average									
Subdomain	2			3			4		
Time	GDFS	BS[d;10,10]	LBS[d;10,10]	GDFS	BS[d;10,10]	LBS[d;10,10]	GDFS	BS[d;10,10]	LBS[d;10,10]
0.45	0.075	0.258	0.163	0.321	0.455	0.617	0.460	0.701	0.619
0.50	0.076	0.247	0.229	0.384	0.644	0.706	0.460	0.764	0.750
0.55	0.076	0.257	0.228	0.443	0.647	0.748	0.462	0.765	0.760
0.60	0.315	0.257	0.314	0.451	0.651	0.773	0.471	0.770	0.769
0.65	0.381	0.414	0.475	0.461	0.652	0.774	0.478	0.773	0.785
0.70	0.455	0.416	0.563	0.472	0.735	0.794	0.540	0.780	0.785
0.75	0.468	0.513	0.593	0.559	0.821	0.812	0.554	0.791	0.787
0.80	0.481	0.687	0.680	0.627	0.822	0.824	0.626	0.795	0.790
0.85	0.565	0.839	0.753	0.705	0.829	0.819	0.639	0.800	0.801
0.90	0.578	0.869	0.773	0.736	0.841	0.850	0.760	0.818	0.819
Performance Comparison Based on Normalized EAcD Average									
Subdomain	2			3			4		
Time	GDFS	BS[d;10,10]	LBS[d;10,10]	GDFS	BS[d;10,10]	LBS[d;10,10]	GDFS	BS[d;10,10]	LBS[d;10,10]
0.45	0.098	0.291	0.192	0.393	0.544	0.745	0.592	0.850	0.767
0.50	0.100	0.291	0.277	0.466	0.763	0.849	0.592	0.941	0.930
0.55	0.100	0.291	0.277	0.542	0.766	0.898	0.595	0.942	0.941
0.60	0.375	0.291	0.378	0.551	0.771	0.927	0.607	0.948	0.951
0.65	0.461	0.479	0.579	0.564	0.772	0.927	0.616	0.953	0.973
0.70	0.556	0.481	0.674	0.576	0.875	0.953	0.690	0.961	0.973
0.75	0.570	0.589	0.698	0.682	0.980	0.975	0.707	0.975	0.976
0.80	0.586	0.792	0.800	0.770	0.981	0.989	0.793	0.979	0.979
0.85	0.691	0.982	0.888	0.865	0.989	0.992	0.806	0.981	0.986
0.90	0.698	0.996	0.895	0.891	1.000	0.997	0.953	0.997	0.986

Table 5.33. Dynamic band search for solving the TSP of subdomains 6, 7, and 8.

Performance Comparison Based on Normalized AAcD Average									
Subdomain	6			7			8		
Time	GDFS	BS[d;10,10]	LBS[d;10,10]	GDFS	BS[d;10,10]	LBS[d;10,10]	GDFS	BS[d;10,10]	LBS[d;10,10]
0.45	0.154	0.161	0.087	0.225	0.226	0.375	0.312	0.478	0.481
0.50	0.154	0.232	0.145	0.225	0.289	0.378	0.362	0.480	0.494
0.55	0.154	0.236	0.212	0.284	0.302	0.382	0.368	0.490	0.594
0.60	0.214	0.238	0.275	0.341	0.415	0.447	0.374	0.638	0.624
0.65	0.289	0.323	0.294	0.344	0.419	0.600	0.438	0.651	0.637
0.70	0.290	0.394	0.430	0.415	0.505	0.653	0.444	0.652	0.682
0.75	0.504	0.479	0.508	0.556	0.610	0.663	0.547	0.736	0.706
0.80	0.512	0.557	0.593	0.646	0.623	0.675	0.566	0.753	0.735
0.85	0.667	0.723	0.752	0.721	0.755	0.685	0.640	0.765	0.745
0.90	0.710	0.830	0.781	0.773	0.786	0.713	0.660	0.777	0.762
Performance Comparison Based on Normalized EAcD Average									
Subdomain	6			7			8		
Time	GDFS	BS[d;10,10]	LBS[d;10,10]	GDFS	BS[d;10,10]	LBS[d;10,10]	GDFS	BS[d;10,10]	LBS[d;10,10]
0.45	0.196	0.183	0.099	0.268	0.276	0.500	0.413	0.626	0.641
0.50	0.196	0.276	0.183	0.268	0.357	0.503	0.491	0.630	0.655
0.55	0.196	0.284	0.276	0.345	0.374	0.503	0.500	0.641	0.772
0.60	0.284	0.286	0.343	0.426	0.532	0.591	0.508	0.825	0.809
0.65	0.381	0.395	0.364	0.430	0.538	0.778	0.596	0.842	0.827
0.70	0.382	0.493	0.561	0.522	0.644	0.845	0.604	0.844	0.894
0.75	0.669	0.596	0.664	0.718	0.769	0.856	0.740	0.960	0.927
0.80	0.679	0.695	0.772	0.846	0.784	0.871	0.764	0.983	0.967
0.85	0.877	0.883	0.973	0.942	0.956	0.879	0.858	0.999	0.970
0.90	0.898	1.000	0.984	0.961	0.988	0.894	0.879	1.000	0.971

Table 5.34. Dynamic band search for solving the TSP of subdomains 10, 11, and 12.

Performance Comparison Based on Normalized AAcD Average									
Subdomain	10			11			12		
Time	GDFS	BS[d;10,10]	LBS[d;10,10]	GDFS	BS[d;10,10]	LBS[d;10,10]	GDFS	BS[d;10,10]	LBS[d;10,10]
0.45	0.174	0.162	0.503	0.282	0.451	0.290	0.409	0.376	0.509
0.50	0.234	0.257	0.540	0.283	0.613	0.375	0.536	0.382	0.519
0.55	0.317	0.256	0.645	0.365	0.611	0.504	0.538	0.464	0.585
0.60	0.318	0.256	0.648	0.429	0.621	0.517	0.544	0.533	0.588
0.65	0.319	0.338	0.652	0.495	0.622	0.522	0.551	0.683	0.716
0.70	0.422	0.570	0.670	0.499	0.626	0.588	0.552	0.778	0.734
0.75	0.593	0.585	0.739	0.518	0.712	0.596	0.656	0.794	0.739
0.80	0.678	0.678	0.747	0.529	0.718	0.605	0.679	0.800	0.740
0.85	0.751	0.690	0.761	0.603	0.733	0.749	0.684	0.813	0.763
0.90	0.749	0.866	0.781	0.698	0.825	0.798	0.729	0.833	0.778
Performance Comparison Based on Normalized EAcD Average									
Subdomain	10			11			12		
Time	GDFS	BS[d;10,10]	LBS[d;10,10]	GDFS	BS[d;10,10]	LBS[d;10,10]	GDFS	BS[d;10,10]	LBS[d;10,10]
0.45	0.192	0.197	0.582	0.354	0.575	0.365	0.513	0.461	0.638
0.50	0.277	0.297	0.667	0.355	0.770	0.469	0.676	0.469	0.652
0.55	0.377	0.297	0.774	0.453	0.770	0.660	0.678	0.562	0.730
0.60	0.378	0.297	0.774	0.535	0.775	0.675	0.687	0.651	0.733
0.65	0.378	0.386	0.775	0.625	0.777	0.680	0.695	0.855	0.886
0.70	0.485	0.671	0.790	0.630	0.782	0.760	0.696	0.957	0.905
0.75	0.692	0.686	0.882	0.654	0.885	0.767	0.821	0.968	0.918
0.80	0.797	0.786	0.889	0.663	0.890	0.770	0.850	0.975	0.920
0.85	0.889	0.792	0.894	0.779	0.894	0.962	0.855	0.988	0.942
0.90	0.889	1.000	0.897	0.887	0.999	0.997	0.869	0.995	0.959

Table 5.35. Dynamic band search for solving a solution of the maze problem of all subdomains studied.

Subdomain	Problem Size	Number of Instances	BS[d;10,10] Average Speedup	LBS[d;10,10] Average Speedup
1	40	50	[1.743 ± 1.116]	[2.088 ± 1.013]
2	50	50	[0.641 ± 0.089]	[0.993 ± 0.129]
3	50	50	[0.885 ± 0.267]	[1.189 ± 0.195]
4	40	50	[1.222 ± 0.553]	[1.704 ± 0.544]
5	40	50	[1.406 ± 0.867]	[1.442 ± 0.443]
6	35	50	[2.163 ± 1.237]	[2.459 ± 1.219]
7	35	50	[1.729 ± 1.096]	[1.161 ± 0.178]
8	50	50	[2.078 ± 0.611]	[0.988 ± 0.027]
9	50	50	[1.644 ± 0.710]	[1.014 ± 0.066]
10	40	50	[1.117 ± 0.334]	[1.011 ± 0.049]
11	50	50	[1.110 ± 0.398]	[1.002 ± 0.007]
12	50	50	[1.105 ± 0.249]	[0.979 ± 0.053]
13	50	50	[1.546 ± 0.510]	[1.088 ± 0.101]
14	50	50	[1.928 ± 0.948]	[0.995 ± 0.052]

Table 5.36. Dynamic band search using new pruning heuristics for solving a solution of the maze problem of all subdomains studied.

Subdomain	Problem Size	Number of Instances	BS[d;10,10] Average Speedup	LBS[d;10,10] Average Speedup
1	40	50	[1.743 ± 1.116]	[1.638 ± 1.111]
2	50	50	[0.641 ± 0.089]	[0.697 ± 0.090]
3	50	50	[0.885 ± 0.267]	[0.957 ± 0.277]
4	40	50	[1.222 ± 0.553]	[1.222 ± 0.553]
5	40	50	[1.406 ± 0.867]	[1.216 ± 0.730]
6	35	50	[2.163 ± 1.237]	[2.198 ± 1.257]
7	35	50	[1.729 ± 1.096]	[2.628 ± 1.314]
8	50	50	[2.078 ± 0.611]	[2.078 ± 0.611]
9	50	50	[1.644 ± 0.710]	[1.644 ± 0.710]
10	40	50	[1.117 ± 0.334]	[1.038 ± 0.311]
11	50	50	[1.110 ± 0.398]	[1.928 ± 0.705]
12	50	50	[1.105 ± 0.249]	[1.105 ± 0.249]
13	50	50	[1.546 ± 0.510]	[1.194 ± 0.301]
14	50	50	[1.928 ± 0.948]	[1.635 ± 0.740]

Table 5.37. Dynamic band search for solving the maze problem of subdomains 1, 2, and 3.

Performance Comparison Based on Normalized AAcD Average									
Subdomain	1			2			3		
Time	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]
0.45	0.027	0.000	0.035	0.000	0.000	0.000	0.000	0.000	0.000
0.50	0.046	0.000	0.057	0.000	0.000	0.000	0.000	0.000	0.000
0.55	0.048	0.000	0.057	0.000	0.000	0.000	0.000	0.000	0.000
0.60	0.073	0.000	0.083	0.000	0.000	0.000	0.000	0.000	0.000
0.65	0.073	0.000	0.091	0.000	0.000	0.000	0.000	0.000	0.000
0.70	0.119	0.036	0.161	0.000	0.000	0.000	0.000	0.000	0.000
0.75	0.256	0.115	0.316	0.051	0.000	0.021	0.025	0.000	0.064
0.80	0.293	0.150	0.348	0.064	0.000	0.057	0.081	0.000	0.082
0.85	0.297	0.426	0.361	0.113	0.000	0.110	0.113	0.000	0.171
0.90	0.405	0.435	0.484	0.137	0.054	0.136	0.135	0.179	0.329
Performance Comparison Based on Normalized EAcD Average									
Subdomain	1			2			3		
Time	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]
0.45	0.100	0.000	0.100	0.000	0.000	0.000	0.000	0.000	0.000
0.50	0.173	0.000	0.173	0.000	0.000	0.000	0.000	0.000	0.000
0.55	0.181	0.000	0.173	0.000	0.000	0.000	0.000	0.000	0.000
0.60	0.279	0.000	0.270	0.000	0.000	0.000	0.000	0.000	0.000
0.65	0.279	0.000	0.280	0.000	0.000	0.000	0.000	0.000	0.000
0.70	0.422	0.099	0.417	0.000	0.000	0.000	0.000	0.000	0.000
0.75	0.575	0.291	0.561	0.299	0.000	0.089	0.100	0.000	0.200
0.80	0.678	0.370	0.627	0.399	0.000	0.189	0.400	0.000	0.200
0.85	0.680	0.770	0.647	0.699	0.000	0.388	0.700	0.000	0.586
0.90	0.692	0.773	0.726	0.900	0.185	0.399	0.800	0.500	0.799

Table 5.38. Dynamic band search for solving the maze problem of subdomains 4, 5, and 6.

Performance Comparison Based on Normalized AAcD Average									
Subdomain	4			5			6		
Time	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]
0.45	0.000	0.000	0.000	0.000	0.000	0.000	0.056	0.043	0.058
0.50	0.000	0.000	0.000	0.000	0.000	0.000	0.091	0.043	0.058
0.55	0.000	0.000	0.000	0.017	0.000	0.033	0.118	0.054	0.119
0.60	0.000	0.000	0.000	0.017	0.000	0.033	0.164	0.084	0.187
0.65	0.000	0.000	0.000	0.037	0.034	0.052	0.192	0.147	0.192
0.70	0.044	0.000	0.056	0.091	0.034	0.069	0.195	0.181	0.225
0.75	0.076	0.057	0.190	0.169	0.131	0.171	0.239	0.273	0.307
0.80	0.091	0.097	0.213	0.191	0.157	0.198	0.252	0.349	0.347
0.85	0.123	0.232	0.287	0.201	0.217	0.234	0.259	0.453	0.373
0.90	0.161	0.353	0.369	0.218	0.399	0.252	0.370	0.482	0.408
Performance Comparison Based on Normalized EAcD Average									
Subdomain	4			5			6		
Time	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]
0.45	0.000	0.000	0.000	0.000	0.000	0.000	0.161	0.096	0.161
0.50	0.000	0.000	0.000	0.000	0.000	0.000	0.307	0.096	0.161
0.55	0.000	0.000	0.000	0.100	0.000	0.098	0.375	0.141	0.314
0.60	0.000	0.000	0.000	0.100	0.000	0.098	0.552	0.228	0.466
0.65	0.000	0.000	0.000	0.151	0.100	0.163	0.647	0.378	0.467
0.70	0.198	0.000	0.198	0.445	0.100	0.253	0.658	0.473	0.544
0.75	0.398	0.175	0.562	0.722	0.440	0.586	0.777	0.688	0.679
0.80	0.468	0.283	0.563	0.799	0.539	0.631	0.820	0.797	0.762
0.85	0.636	0.582	0.711	0.829	0.715	0.692	0.838	0.919	0.812
0.90	0.795	0.800	0.896	0.852	0.981	0.693	0.869	0.962	0.868

Table 5.39. Dynamic band search for solving the maze problem of subdomains 7, 8, and 9.

Performance Comparison Based on Normalized AAcD Average									
Subdomain	7			8			9		
Time	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]
0.45	0.035	0.000	0.066	0.063	0.063	0.063	0.000	0.000	0.000
0.50	0.091	0.000	0.102	0.063	0.063	0.063	0.000	0.000	0.000
0.55	0.152	0.000	0.173	0.063	0.063	0.063	0.000	0.000	0.000
0.60	0.153	0.040	0.200	0.063	0.063	0.063	0.000	0.000	0.000
0.65	0.220	0.244	0.266	0.063	0.063	0.063	0.000	0.000	0.000
0.70	0.318	0.260	0.404	0.063	0.063	0.063	0.000	0.000	0.000
0.75	0.383	0.341	0.461	0.063	0.063	0.063	0.000	0.000	0.000
0.80	0.394	0.520	0.520	0.063	0.063	0.063	0.000	0.000	0.000
0.85	0.408	0.524	0.568	0.073	0.063	0.095	0.013	0.000	0.045
0.90	0.527	0.618	0.601	0.105	0.228	0.221	0.024	0.000	0.127
Performance Comparison Based on Normalized EAcD Average									
Subdomain	7			8			9		
Time	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]
0.45	0.114	0.000	0.206	0.100	0.100	0.100	0.000	0.000	0.000
0.50	0.281	0.000	0.271	0.100	0.100	0.100	0.000	0.000	0.000
0.55	0.448	0.000	0.417	0.100	0.100	0.100	0.000	0.000	0.000
0.60	0.452	0.096	0.450	0.100	0.100	0.100	0.000	0.000	0.000
0.65	0.562	0.517	0.546	0.100	0.100	0.100	0.000	0.000	0.000
0.70	0.769	0.547	0.794	0.100	0.100	0.100	0.000	0.000	0.000
0.75	0.801	0.628	0.794	0.100	0.100	0.100	0.000	0.000	0.000
0.80	0.829	0.812	0.821	0.100	0.100	0.100	0.000	0.000	0.000
0.85	0.841	0.819	0.866	0.200	0.100	0.300	0.100	0.000	0.100
0.90	0.844	0.898	0.866	0.400	0.300	0.400	0.500	0.000	0.400

Table 5.40. Dynamic band search for solving the maze problem of subdomains 10, 11, and 12.

Performance Comparison Based on Normalized AAcD Average									
Subdomain	10			11			12		
Time	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]
0.45	0.000	0.000	0.000	0.000	0.000	0.000	0.028	0.028	0.028
0.50	0.000	0.000	0.000	0.000	0.000	0.000	0.028	0.028	0.028
0.55	0.000	0.000	0.000	0.000	0.000	0.000	0.028	0.028	0.028
0.60	0.000	0.000	0.000	0.000	0.000	0.000	0.028	0.028	0.028
0.65	0.000	0.000	0.000	0.033	0.048	0.036	0.028	0.028	0.028
0.70	0.000	0.000	0.000	0.033	0.048	0.048	0.028	0.028	0.028
0.75	0.000	0.000	0.000	0.044	0.048	0.063	0.072	0.028	0.028
0.80	0.000	0.000	0.000	0.089	0.048	0.160	0.089	0.028	0.028
0.85	0.014	0.000	0.073	0.106	0.048	0.281	0.118	0.036	0.084
0.90	0.021	0.074	0.216	0.124	0.157	0.469	0.153	0.221	0.254
Performance Comparison Based on Normalized EAcD Average									
Subdomain	10			11			12		
Time	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]
0.45	0.000	0.000	0.000	0.000	0.000	0.000	0.100	0.100	0.100
0.50	0.000	0.000	0.000	0.000	0.000	0.000	0.100	0.100	0.100
0.55	0.000	0.000	0.000	0.000	0.000	0.000	0.100	0.100	0.100
0.60	0.000	0.000	0.000	0.000	0.000	0.000	0.100	0.100	0.100
0.65	0.000	0.000	0.000	0.100	0.100	0.100	0.100	0.100	0.100
0.70	0.000	0.000	0.000	0.100	0.100	0.100	0.100	0.100	0.100
0.75	0.000	0.000	0.000	0.200	0.100	0.200	0.200	0.100	0.100
0.80	0.000	0.000	0.000	0.500	0.100	0.500	0.400	0.100	0.100
0.85	0.300	0.000	0.300	0.600	0.100	0.600	0.500	0.100	0.300
0.90	0.400	0.100	0.500	0.700	0.200	0.700	0.600	0.300	0.400

Table 5.41. Dynamic band search for solving the maze problem of subdomains 13 and 14.

Performance Comparison Based on Normalized AAcD Average						
Subdomain	13			14		
Time	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]
0.45	0.000	0.000	0.000	0.000	0.000	0.000
0.50	0.000	0.000	0.000	0.000	0.000	0.000
0.55	0.000	0.000	0.000	0.000	0.000	0.000
0.60	0.000	0.000	0.000	0.000	0.000	0.000
0.65	0.000	0.000	0.000	0.000	0.000	0.000
0.70	0.000	0.000	0.000	0.000	0.000	0.000
0.75	0.000	0.000	0.000	0.000	0.000	0.000
0.80	0.012	0.000	0.058	0.012	0.000	0.023
0.85	0.061	0.080	0.175	0.015	0.000	0.060
0.90	0.088	0.082	0.284	0.026	0.078	0.201
Performance Comparison Based on Normalized EAcD Average						
Subdomain	13			14		
Time	GDFS	BS[d;1,10]	LBS[d;1,10]	GDFS	BS[d;1,10]	LBS[d;1,10]
0.45	0.000	0.000	0.000	0.000	0.000	0.000
0.50	0.000	0.000	0.000	0.000	0.000	0.000
0.55	0.000	0.000	0.000	0.000	0.000	0.000
0.60	0.000	0.000	0.000	0.000	0.000	0.000
0.65	0.000	0.000	0.000	0.000	0.000	0.000
0.70	0.000	0.000	0.000	0.000	0.000	0.000
0.75	0.000	0.000	0.000	0.000	0.000	0.000
0.80	0.200	0.000	0.200	0.100	0.000	0.100
0.85	0.500	0.200	0.500	0.200	0.000	0.200
0.90	0.600	0.200	0.500	0.300	0.100	0.300

5.8. Learning Search Heuristics for Circuit Testing

In this section, we describe our results on learning heuristics that are used in the genetic search for circuit testing. The genetic search package for circuit testing, called CRIS, was developed by experts in the area [83], who identified seven parameters that can be tuned, including five integer-valued parameters and two real-valued parameters. Each of these parameters has its range specified. The ranges can be large or small.

A heuristic is a set of these seven parameters. The numerical characteristics of these seven parameters are summarized as follows.

Parameter	Parameter Type	Parameter Range
x_1	integer-valued	[1,5]
x_2	integer-valued	[4,20]
x_3	integer-valued	[1,20]
x_4	real-valued	[0.1,4.0]
x_5	integer-valued	[50,500]
x_6	integer-valued	[1,20]
x_7	real-valued	[0.1,1.0]

These parameters are control parameters used in genetic search for finding the best test pattern. The performance of genetic search is characterized by fault coverage. The relation between control parameters and fault coverage is unknown.

The goal of the learning is to find a heuristic (a set of seven parameters) that can improve the fault coverage of the genetic search. This goal is important when we are

interested in finding a set of parameters that work well across all circuits, instead of using different sets of parameters for different circuits.

The baseline heuristics are *unknown*. The baseline performance is obtained from the literature [83], where different circuits may use different heuristics and the heuristics used are not shown in the literature. Moreover, the fault coverage values are the largest among several runs using different random seeds; however, the number of runs is unknown.

During the learning, we treat all circuits as in the same subdomain. The fault coverage values for different circuits vary significantly. Therefore, we choose the *gain* of fault coverage as the performance metric. The gain of fault coverage for heuristic *i* over heuristic *j* is defined as the ratio of the fault coverage FC_i of heuristic *i* to the fault coverage FC_j of heuristic *j*. The concept of fault coverage gain is similar to speedup. The larger the fault coverage gain, the better. The performance of a heuristic is the average fault coverage gain over a given set of test circuits.

Another approach is to treat each individual circuit as a subdomain and to learn heuristics for all individual circuits. This implies that we need to perform *N* learnings when we have *N* circuits. This approach is time consuming but can come to better heuristics. In this thesis, we do not take this approach because the main goal of this work is to demonstrate the idea and both approaches are based on the same idea.

The learning process consists of 30 generations, each having 40 heuristics and retaining 25%. Genetic generators used include crossover and mutation, as well as gradient descent and random generations. Each generation is allowed to perform 160 tests.

We conduct two learning experiments for different random seeds (61801 and 98052) under the minimum-risk scheduling strategy. The results are shown in Tables 5.42-5.43, where a fault coverage ratio is defined as the ratio of a fault coverage due to the learned heuristic to that due to one of the baseline heuristics.

From Tables 5.42 and 5.43, the unknown baseline heuristics perform well because the baseline ones have been tuned over several months. However, different circuits may require different heuristics; consequently, whenever a new circuit is to be tested, we may require a significant amount of time to find a new heuristic.

From Table 5.42, the learned heuristic (obtained from the learning with a random seed of 61801) seems to perform well across most of the circuits. Therefore, we may simply apply the learned heuristic to find the best test pattern for a new circuit, or we can also re-run the learning to obtain a specialized heuristic for the new circuit.

5.9. Summary

In this chapter, we apply genetic programming to learn knowledge-lean search heuristics for problem solvers that can be used for solving a class of instances of problems with different characteristics. The search heuristics we are interested in can be expressed as parametric functions of problem variables and include decomposition heuristics, selection heuristics, and pruning heuristics.

In this research, we first develop a novel framework, consisting of learning phases and verification phases, for creating good heuristics for solving problems of different characteristics. We present various strategies for scheduling tests in learning and study different evaluation methods for assessing the performance of heuristics.

Table 5.42. Results of learning a heuristic to be used in CRIS, where the learning is with a random seed of 61801 and the MR scheduling strategy. (Data under F.C.(61801) and F.C.(98052) are fault coverage values using this heuristic with random seeds, respectively, 61801 and 98052; data under F.C.(max) are the maximum of data under F.C.(61801) and F.C. (98052).) (The average fault coverage of the unknown baseline is 79.63.)

Heuristic ID = 605, Form = (3, 20, 23, 2.347, 500, 1, 0.1)				
Circuit	F.C.(61801)	F.C.(98052)	F.C.(max)	F.C. Gain
s386	78.65	77.86	78.65	1.03
s344	95.61	95.61	95.61	1.02
s382	86.72	85.21	86.72	1.26
s526	78.56	73.51	78.56	1.02
s298	85.06	83.12	85.06	1.04
s444	81.65	86.08	86.08	1.03
s641	83.08	84.58	84.58	0.99
s400	84.80	82.19	84.80	1.00
s713	81.93	81.24	81.93	1.00
s820	46.00	46.12	46.00	0.87
s832	44.37	43.68	44.37	1.04
s1238	89.15	87.60	89.15	0.98
s1494	92.76	95.02	95.02	1.05
s1488	94.62	93.74	94.62	1.04
s1196	93.40	94.61	94.61	1.00
Average	81.09	80.68	81.72	1.02

Table 5.43. Results of learning a heuristic used in CRIS, where the learning is with a random seed of 98052 and the MR scheduling strategy. (Data under F.C.(61801) and F.C.(98052) are fault coverage values using this heuristic with random seeds, respectively, 61801 and 98052; data under F.C.(max) are the maximum of data under F.C.(61801) and F.C. (98052).) (The average fault coverage of the unknown baseline is 79.63.)

Heuristic ID = 667, Form = (3, 20, 20, 3.85, 442, 1, 0.1)				
Circuit	F.C.(61801)	F.C.(98052)	F.C.(max)	F.C. Gain
s386	77.60	77.60	77.60	1.02
s344	95.61	95.61	95.61	1.02
s382	86.72	85.21	86.72	1.26
s526	78.56	73.51	78.56	1.02
s298	85.06	83.12	85.06	1.04
s444	81.65	86.08	86.08	1.03
s641	83.08	84.58	84.58	0.99
s400	84.80	82.19	84.80	1.00
s713	81.93	80.90	81.93	1.00
s820	40.24	42.82	42.82	0.81
s832	39.89	51.26	51.26	1.21
s1238	89.45	91.22	91.22	1.01
s1494	92.56	93.36	93.36	1.04
s1488	93.14	91.52	93.14	1.02
s1196	94.12	92.11	94.12	0.99
Average	80.29	80.74	81.79	1.03

We demonstrated in this chapter the improvement of the learned search algorithms that use learned heuristics. We show the output of the learning in Appendix B.

CHAPTER 6.

CONCLUSIONS

In this thesis, we have developed a systematic, integrated method of designing efficient search algorithms that solve COPs in real time. These problems addressed have a well-defined problem objective and a finite set of well-defined problem constraints. Search algorithms in this thesis comprise meta-control and primitive search. We have used state-space trees as problem representations.

The design of search algorithms consists of two phases: (a) designing generic search algorithms, and (b) improving by genetics-based machine learning parametric heuristics used in the search algorithms designed. This research is a systematic method that integrates domain knowledge, search techniques, and automated learning techniques to design better search algorithms. The approach we have undertaken in designing search algorithms is an engineering approach for finding good heuristics in limited time.

In the first phase, we have designed a real-time search algorithm that solves dense COPs in real time, and designed band search that solves efficiently COPs when time is not limited. Moreover, we have designed dynamic band search that solves efficiently sparse COPs in real time. We found that RTS and band search can generally outperform GDFS in the problems we studied.

In the second phase, we have used genetics-based machine learning methods to learn parametric search heuristics used in the algorithms designed to improve search performance. To accommodate possibly various problem characteristics in an application, we have developed a novel framework that can learn and generalize heuristics over a set of

subdomains. We have also developed a nonparametric schedule for allocating costly sampling during our learning algorithm. For the problems studied, we found that the heuristics learned can generally improve the performance of the search algorithms designed.

REFERENCES

[1] A. N. Aizawa and B. W. Wah, ‘‘A Sequential Sampling Procedure for Genetic Algorithms,’’ *Proc. 5th Int’l Workshop of the Bellman Continum*, Waikoloa, Hawaii, Jan. 1993.

[2] G. Baudet, ‘‘On the Branching Factor of the Alpha-Beta Pruning Algorithm,’’ *Artificial Intelligence*, vol. 10, no. 2, pp. 173-199, North Holland, 1978.

[3] J. L. Bentley, ‘‘Multidimensional Divide-and-Conquer,’’ *Communications of the ACM*, vol. 23, no. 51-59, ACM, 1980.

[4] D. Brelaz, ‘‘New Methods to Color the Vertices of a Graph,’’ *Communications of the ACM*, vol. 22, no. 4, pp. 251-256, April 1979.

[5] P. P. Chakrabarti, S. Ghose, A. Acharya, and S. C. DeSarkar, ‘‘Tree Searching in Restricted Memory,’’ *International Symposium on Electronic Devices Circuits and Systems*, India, Dec. 1987.

[6] P. P. Chakrabarti, S. Ghose, A. Acharya, and S. C. de Sarkar, ‘‘Heuristic Search in Restricted Memory,’’ *Artificial Intelligence*, vol. 41, pp. 197-221, 1989.

[7] C. L. Chang and J. R. Slagle, ‘‘An Admissible and Optimal Algorithm for Searching AND/OR Graphs,’’ *Artificial Intelligence*, vol. 2, no. 2, pp. 117-128, North-Holland, 1971.

[8] L.-C. Chu and B. W. Wah, ‘‘Optimization in Real Time,’’ *Proc. Real Time Systems Symp.*, pp. 150-159, IEEE, Nov. 1991.

[9] L.-C. Chu and B. W. Wah, ‘‘Band Search: An Efficient Alternative to Guided Depth-First Search,’’ *Proc. Int’l Conf. on Tools for Artificial Intelligence*, pp. 154-161, IEEE Computer Society, Nov. 1992.

[10] Committee on Physical, Mathematical, and Engineering Sciences, *Grand Challenges: High Performance Computing and Communications (To Supplement the President’s Fiscal Year 1992 Budget*, 1992.

[11] J. C. Culberson, ‘‘Iterated Greedy Graph Coloring and the Difficulty Landscape,’’ Technical Report, Department of Computer Science, University of Alberta, Edmonton, Canada T6G 2H1, June 1992..

[12] T. Dean and M. Boddy, ‘‘An Analysis of Time Dependent Planning,’’ *Proc. of AAAI*, pp. 49-54, 1988.

[13] R. Dechter and J. Pearl, ‘‘The Optimality of A^* Revisited,’’ *Proc. of AAAI*, pp. 95-99, 1983.

[14] R. Dechter and J. Pearl, ‘‘Generalized Best-First Search and the Optimality of A^*,’’ *J. ACM*, vol. 32, pp. 506-536, 1985.

[15] R. Dodhiawala, N. S. Sridharan, P. Raulefs, and C. Pickering, ‘‘Real-Time AI Systems: A Definition and An Architecture,’’ *Proc. of AAAI*, pp. 256-261, 1989.

[16] P. A. Finkel and J. P. Fishburn, ‘‘Proved Speedup Bounds of Parallel Alpha-Beta Search,’’ *Trans. Pattern Analysis and Machine Intelligence*, vol. PAMI-5, no. 1, pp. 89-92, IEEE, Jan. 1983.

[17] R. A. Finkel and J. P. Fishburn, ‘‘Parallelism in Alpha-Beta Search,’’ *Artificial Intelligence*, vol. 19, no. 1, pp. 89-106, North-Holland, 1982.

[18] M. R. Garey and D. S. Johnson, *Computers and Intractability,* Freeman, San Francisco, CA, 1979.

[19] D. Gelperin, ‘‘On the Optimality of A*,’’ *Artificial Intelligence*, vol. 8, no. 1, pp. 69-76, North-Holland, 1977.

[20] M. L. Ginsberg and W. D. Harvey, ‘‘Iterative Broadening,’’ *Proc. National Conf. on Artificial Intelligence*, pp. 216-220, AAAI, 1990.

[21] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Pub. Co., 1989.

[22] M. M. Gooley and B. W. Wah, ‘‘Efficient Reordering of Prolog Programs,’’ *IEEE Trans. on Knowledge and Data Engineering*, vol. 1, no. 4, pp. 470-482, IEEE, Dec. 1989.

[23] J.J. Grefenstette, ‘‘Credit Assignment in Rule Discovery Systems Based on Genetic Algorithms,’’ *Machine Learning*, vol. 3, no. 2/3, pp. 225-246, Kluwer Academic Pub., Boston, MA, Oct. 1988.

[24] R. M. Haralick and G. L. Elliot, ‘‘Increasing Tree Search Efficiency for Constraint Satisfaction Problems,’’ *Artificial Intelligence*, vol. 14, pp. 263-313, North Holland, 1980.

[25] L. R. Harris, ‘‘The Heuristic Search Under Conditions of Error,’’ *Artificial Intelligence*, vol. 5, pp. 217-234, North-Holland, 1974.

[26] R. Hess, D. Lemke, and M. Weiss, ‘‘Maze,’’ *X Version 11 Release 4*, July 1991.

[27] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms,* Computer Science Press, Rockville, MD, 1978.

[28] G.A. Iba, ‘‘A Heuristic Approach to the Discovery of Macro-Operators,’’ *Machine Learning*, vol. 3, no. 4, pp. 285-318, Kluwer Academic Pub., Boston, MA, March 1989.

[29] T. Ibaraki, ‘‘The Power of Dominance Relations in Branch-and-Bound Algorithms,’’ *J. of the ACM*, vol. 24, no. 2, pp. 264-279, April 1977.

[30] T. Ibaraki, ‘‘Depth-m Search in Branch-and-Bound Algorithms,’’ *Int’l. J. of Computer and Information Sciences*, vol. 7, no. 4, pp. 315-373, Plenum Press, 1978.

[31] A. Ieumwananonthachai, A. N. Aizawa, S. R. Schwartz, B. W. Wah, and J. C. Yan, ‘‘Intelligent Mapping of Communicating Processes in Distributed Computing Systems,’’ *Proc. Supercomputing 91*, pp. 512-521, ACM/IEEE, Albuquerque, NM, Nov. 1991.

[32] A. Ieumwananonthachai, A. Aizawa, S. R. Schwartz, B. W. Wah, and J. C. Yan, ‘‘Intelligent Process Mapping Through Systematic Improvement of Heuristics,’’ *J. of Parallel and Distributed Computing*, vol. 15, pp. 118-142, Academic Press, June 1992.

[33] D. W. Jones, ‘‘Concurrent Operations on Priority Queues,’’ *Communications of the ACM*, pp. 132-137, Jan. 1989.

[34] M. Joseph and A. Goswami, ‘‘What’s ‘Real’ about Real-time Systems ?,’’ *Proc. of the 9th Real-Time Systems Symposium*, pp. 78-85, IEEE, Dec. 1988.

[35] R. Karp and M. Held, ‘‘Finite State Processes and Dynamic Programming,’’ *SIAM J. on Appl. Math.*, vol. 15, pp. 693-718, 1967.

[36] D. E. Knuth and R. W. Moore, ‘‘An Analysis of Alpha-Beta Pruning,’’ *Artificial Intelligence*, vol. 6, no. 4, pp. 293-326, North-Holland, 1975.

[37] R. E. Korf, ‘‘Depth-First Iterative Deepening: An Optimal Admissible Tree Search,’’ *Artificial Intelligence*, vol. 27, pp. 97-109, North-Holland, 1985.

[38] R. E. Korf, *Learning to Solve Problems by Searching for Macro-Operators*, Pitman, Boston, 1985.

[39] R. E. Korf, ‘‘Macro-Operators: A Weak Method for Learning,’’ *Artificial Intelligence*, vol. 26, pp. 35-77, North-Holland, 1985.

[40] R. E. Korf, ‘‘Real-Time Heuristic Search: First Results,’’ *Proc. National Conf. on Artificial Intelligence*, pp. 133-8, AAAI, Inc., Seattle, Washington, June 1987.

[41] R. E. Korf, ‘‘Real-Time Heuristic Search: New Results,’’ *Proc. of AAAI-88*, pp. 139-144, St. Paul, MN, 1988.

[42] R. E. Korf, ‘‘Real-Time Heuristic Search,’’ *Artificial Intelligence*, vol. 42, pp. 189-211, Elsevier Science Publishers, 1990.

[43] V. Kumar and L. N. Kanal, ‘‘A General Branch and Bound Formulation for Understanding and Synthesizing And/Or Tree Search Procedures,’’ *Artificial Intelligence*, vol. 21, no. 1-2, pp. 179-198, North-Holland, 1983.

[44] V. Kumar and L. Kanal, ‘‘Parallel Branch-and-Bound Formulations for AND/OR Tree Search,’’ *Trans. Pattern Analysis and Machine Intelligence*, vol. PAMI-6, IEEE, 1984.

[45] V. Kumar and L. Kanal, ‘‘The CDP: A Unifying Formulation for Heuristic Search, Dynamic Programming, and Branch-and-Bound,’’ *Search in Artificial Intelligence (eds L. Kanal, V. Kumar)*, pp. 1-27, Springer-Verlag, New York, 1988.

[46] J. B. H. Kwa, ‘‘BS*: An Admissible Bidirectional Staged Heuristic Search Algorithm,’’ *Artificial Intelligence*, vol. 38, pp. 95-109, 1989.

[47] T. J. Laffey, P. A. Cox, J. L. Schmidt, S. M. Kao, and J. Y. Read, ‘‘Real-Time Knowledge-Based Systems,’’ *AI Magazine*, pp. 27-45, Spring 1988.

[48] P. Langley, ‘‘Learning to Search: From Weak Methods to Domain-Specific Heuristics,’’ *Cognitive Science*, vol. 9, pp. 217-260, Ablex Pub. Co., Norwood, NJ, 1985.

[49] E. L. Lawler and D. W. Wood, ‘‘Branch and Bound Methods: A Survey,’’ *Operations Research*, vol. 14, pp. 699-719, ORSA, 1966.

[50] V. R. Lesser, J. Pavlin, and E. Durfee, ‘‘Approximate Processing in Real-Time Problem Solving,’’ *AI Magazine*, pp. 49-61, AAAI, Spring 1988.

[51] G. J. Li and B. W. Wah, ‘‘How to Cope with Anomalies in Parallel Approximate Branch-and-Bound Algorithms,’’ *Proc. National Conf. on Artificial Intelligence*, pp. 212-215, AAAI, Aug. 1984.

[52] G.-J. Li and B. W. Wah, ‘‘MANIP-2: A Multicomputer Architecture for Evaluating Logic Programs,’’ *Proc. Int’l Conf. on Parallel Processing*, pp. 123-130, IEEE, Aug. 1985. Also in *Tutorial: Computers for Artificial Intelligence Applications*, ed. B. W. Wah, IEEE Computer Society, 1986, pp. 392-399.

[53] G.-J. Li and B. W. Wah, ‘‘Coping with Anomalies in Parallel Branch-and-Bound Algorithms,’’ *IEEE Trans. on Computers*, vol. C-34, no. 6, pp. 568-573, IEEE, June 1986.

[54] G. J. Li and B. W. Wah, ‘‘Computational Efficiency of Combinatorial OR-Tree Searches,’’ *IEEE Trans. on Software Engineering*, vol. 16, no. 1, pp. 13-31, IEEE, Jan. 1990.

[55] G.-J. Li and B. W. Wah, ‘‘Parallel Iterative Refining A*: An Efficient Search Scheme for Solving Combinatorial Optimization Problems,’’ *Proc. Int’l Conf. on Parallel Processing*, pp. 608-615, Pennsylvania State Univ. Press, University Park, PA, Aug. 1991.

[56] K. J. Lin, S. Natarajan, and J. W. S. Liu, ‘‘Imprecise Result: Utilizing Partial Computations in Real-Time Systems,’’ *Proc. of the Eighth Real-Time System Symposium*, pp. 210-217, Dec. 1987.

[57] R. P. Lippmann, ‘‘An Introduction to Computing with Neural Nets,’’ *Accoustics, Speech and Signal Processing Magazine*, vol. 4, no. 2, pp. 4-22, IEEE, April 1987.

[58] J. W. S. Liu, K. J. Lin, and S. Natarajan, ‘‘Scheduling Real-Time, Periodic Jobs Using Imprecise Results,’’ *Proc. of the Eighth Real-Time System Symposium*, pp. 252-260, Dec. 1987.

[59] J. W. S. Liu, K.-J. Lin, W.-K. Shih, A. C.-S. Yu, J.-Y. Chung, and W. Zhao, ‘‘Algorithms for Scheduling Imprecise Computations,’’ *Computer*, pp. 58-68, IEEE, May 1991.

[60] M. B. Lowrie and B. W. Wah, ‘‘Learning Heuristic Functions for Numeric Optimization Problems,’’ *Proc. Computer Software and Applications Conf.*, pp. 443-450, IEEE, Chicago, IL, Oct. 1988.

[61] A. Mahanti, S. Ghosh, and A. K. Pal, ‘‘A High-Performance Limited-Memory Admissible and Real Time Search Algorithms for Networks,’’ *CS-TR-2858 (UMIACS-TR-92-34)*, Dept. of Computer Science, Univ. of Maryland, College Park, Maryland, March 1992.

[62] T. A. Marsland and M. Campbell, ‘‘Parallel Search of Strongly Ordered Game Trees,’’ *Computing Surveys*, vol. 14, no. 4, pp. 533-551, ACM, Dec. 1982.

[63] T. A. Marsland and F. Popowich, ‘‘Parallel Game-Tree Search,’’ *Trans. on Pattern Analysis and Machine Intelligence*, vol. PAMI-7, no. 4, pp. 442-452, IEEE, July

1985.

[64] A. Martelli, ‘‘On the Complexity of Admissible Search Algorithms,’’ *Artificial Intelligence*, vol. 8, no. 1, pp. 1-13, North-Holland, 1977.

[65] I. Miller and J. E. Freund, *Probability and Statistics for Engineers (second edition)*, Prentice-Hall, Englewood Cliffs, NJ, 1977.

[66] S. Minton, J.G. Carbonell, and O. Etzioni, ‘‘Acquiring Effective Search Control Rules: Explanation-Based Learning in the PRODIGY System,’’ *Machine Learning*, pp. 122-133, Kluwer Academic Pub., Boston, MA, 1987.

[67] T. M. Mitchell, ‘‘Learning and Problem Solving,’’ *Proc. 8th Int’l Joint Conf. on Artificial Intelligence*, pp. 1139-1151, William Kaufman, Los Altos, CA, Aug. 1983.

[68] D. A. Moon, ‘‘Symbolics Architecture,’’ *Computer*, vol. 20, no. 1, pp. 43-52, IEEE, Jan. 1987.

[69] R.J. Mooney, ‘‘Generalizing the Order of Operators in Macro-Operators,’’ *Machine Learning*, pp. 270-283, Kluwer Academic Pub., Boston, MA, 1988.

[70] D. Mutchler, ‘‘Optimal Allocation of Very Limited Search Resources,’’ *Proc. of AAAI-86*, pp. 467-471, 1986.

[71] J. Neter, W. Wasserman, and M. H. Kutner, *Applied Linear Statistical Models*, Irwin, Inc., Homewood, IL, 1985.

[72] N. J. Nilsson, *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, NY, 1971.

[73] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga, 1980.

[74] C. J. Paul, A. Acharya, B. Black, and J. K. Strosnider, ‘‘Reducing Problem-Solving Variance to Improve Predictability,’’ *Communications of the ACM*, vol. 34, no. 8, pp. 80-93, Aug. 1991.

[75] J. Pearl, ‘‘On the Discovery and Generation of Certain Heuristics,’’ *The AI Magazine*, pp. 23-33, AAAI, Winter/Spring 1983.

[76] J. Pearl, ‘‘Knowledge versus Search: A Quantitative Analysis Using A^*,’’ *Artificial Intelligence*, vol. 20, pp. 1-13, Elsevier Science Pub., New York, 1983.

[77] J. Pearl, *Heuristics--Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, MA, 1984.

[78] L. A. Rendell, *An Adaptive Plan for State-Space Problems*, Tech. Rep. CS-81-13, Univ. of Waterloo, Ontario, Canada, March 1981.

[79] L. A. Rendell, ‘‘A New Basis for State-Space Learning Systems and a Successful Implementation,’’ *Artificial Intelligence*, vol. 20, pp. 369-392, North-Holland, 1983.

[80] E. Rich and K. Knight, *Artificial Intelligence*, McGraw Hill, New York, 1991.

[81] S. Russell and E. Wefald, ‘‘Principles of Metareasoning,’’ *Artificial Intelligence*, vol. 49, pp. 361-395, Elsevier, Amsterdam, 1991.

[82] S. J. Russell and S. Zilberstein, ‘‘Composing Real-Time Systems,’’ *Proc. of the 12th IJCAI*, Sydney, Australia, Aug. 1991.

[83] D. G. Saab, Y. G. Saab, and J. A. Abraham, ‘‘CRIS: A Test Cultivation Program for Sequential VLSI Circuits,’’ *Proc. of Int’l Conf. on Computer Aided Design*, pp. 216-219, IEEE, Santa Clara, CA, Nov. 8-12, 1992.

[84] U. K. Sarkar, P. P. Chakrabarti, S. Ghose, and S. C. De Sarkar, ‘‘Reducing Reexpansions in Iterative-Deepening Search by Controlling Cutoff Bounds,’’ *Artificial Intelligence*, vol. 50, pp. 207-221, Elsevier Science Publishers, 1991.

[85] M. Schoppers, ‘‘Real-Time Knowledge-Based Control Systems,’’ *Communications of the ACM*, vol. 34, no. 8, August 1991.

[86] A. K. Sen and A. Bagchi, ‘‘Fast Recursive Formulations for Best-Fist Search That Allow Controlled Use of Memory,’’ *Proc. Int’l Joint Conf. on Artificial Intelligence*, pp. 297-302, IJCAI, Inc., Detroit, MI, Aug. 1989.

[87] S. Shekhar and S. Dutta, ‘‘Minimizing Response Times in Real Time Planning and Search,’’ *Proc. of AAAI-89*, pp. 238-242, 1989.

[88] R. M. Stallman, G. J. Sussman, and Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis, *Artificial Intelligence*, vol. 9, no. 2, 1977.

[89] J. A. Stankovic, ‘‘Real-Time Computing Systems: The Next Generation,’’ in *Tutorial in Hard Real-Time Systems*, ed. K. Ramamritham, pp. 14-37, IEEE, 1988.

[90] R. S. Sutton, *Temporal Credit Assignment in Reinforcement Learning*, Ph.D. Thesis, Univ. of Massachusetts, Amherst, MA, Feb. 1984.

[91] J. S. Turner, ‘‘Almost All k-Colorable Graphs Are Easy to Color,’’ *Journal of Algorithms*, vol. 9, pp. 63-82, Academic Press, Inc., 1988.

[92] N. Rao Vempaty, V. Kumar, and R. E. Korf, ‘‘Depth-First vs Best-First Search,’’ *Proc. National Conf. on Artificial Intelligence*, AAAI, Anaheim, CA, July 1991.

[93] B. W. Wah and Y. W. Ma, ‘‘MANIP - A Parallel Computer System for Implementing Branch and Bound Algorithms,’’ *Proc. 8th Int’l Symp. on Computer Architecture*, pp. 259-262, ACM/IEEE, May 1981.

[94] B. W. Wah and Y. W. Ma, ‘‘The Architecture of MANIP - A Parallel Computer System for Solving Non-deterministic Polynomial Complete Problems,’’ *Proc. National Computer Conf.*, pp. 149-161, AFIPS Press, May 1981.

[95] B. W. Wah, G. J. Li, and C. F. Yu, ‘‘The Status of MANIP — A Multi-Computer Architecture for Solving Combinatorial Extremum-Search Problems,’’ *Proc. 11’th Int’l Symp. on Computer Architecture*, pp. 56-63, ACM/IEEE, June 1984.

[96] B. W. Wah and Y. W. Ma, ‘‘MANIP--A Multicomputer Architecture for Solving Combinatorial Extremum Problems,’’ *IEEE Trans. on Computers*, vol. C-33, no. 5, pp. 377-390, IEEE, May 1984. Also in *Tutorial: Computer Architecture*, ed. D. D. Gajski, et al., IEEE Computer Society, 1987, pp. 578-591; Also in *Tutorial: Parallel Architecture for Database Systems*, ed. A. R. Hurson, et al., IEEE Computer Society,

1988.

[97] B. W. Wah, G.-J. Li, and C.-F. Yu, ‘‘Multiprocessing of Combinatorial Search Problems,’’ *Computer*, vol. 18, no. 6, pp. 93-108, IEEE, 1985.

[98] B. W. Wah and C. F. Yu, ‘‘Stochastic Modeling of Branch-and-Bound Algorithms with Best-First Search,’’ *IEEE Trans. on Software Engineering*, vol. SE-11, no. 9, pp. 922-934, IEEE, Sept. 1985.

[99] B. W. Wah, M. B. Lowrie, and G.-J. Li, ‘‘Computers for Symbolic Processing,’’ *Proc. of the IEEE*, vol. 77, no. 4, pp. 507-540, April 1989. Also in *Architectures for Data/Knowledge Base Systems*, ed. L. L. Miller, S. H. Pakzad and A. R. Hurson, IEEE Computer Society Press, 1994.

[100] B. W. Wah and L.-C. Chu, ‘‘TCA*--A Time-Constrained Approximate A* Search Algorithm,’’ *Proc. Int’l Workshop on Tools for Artificial Intelligence*, pp. 314-320, IEEE, Nov. 1990.

[101] B. W. Wah and L.-C. Chu, ‘‘TCGD: A Time-Constrained Approximate Guided Depth-First Search Algorithm,’’ *Proc. Int’l Computer Symp.*, pp. 507-516, Tsing Hua Univ., Hsinchu, Taiwan, Dec. 1990.

[102] B. W. Wah and G.-J. Li, ‘‘Optimal Parallel Evaluation of AND-Trees,’’ *J. Parallel and Distributed Computing*, vol. 10, no. 1, pp. 1-17, Academic Press, Oct. 1990.

[103] B. W. Wah, *MIDA*: An IDA* Search with Dynamic Control*, Research Report CRHC-91-09, Center for Reliable and High Performance Computing, Coordinated Science Laboratory, Univ. of Illinois, Urbana, IL, April 1991.

[104] B. W. Wah, ‘‘Population-Based Learning: A New Method for Learning from Examples under Resource Constraints,’’ *IEEE Trans. on Knowledge and Data Engineering*, vol. 4, no. 5, pp. 454-474, IEEE, Oct. 1992.

[105] B. W. Wah and L.-C. Chu, ‘‘Combinatorial Search Algorithms with Meta-Control: Modeling and Implementations,’’ *Int’l J. of Artificial Intelligence Tools*, vol. 1, no. 3, pp. 369-397, World Scientific Publishers, Sept. 1992.

[106] B. W. Wah, A. Ieumwananonthachai, L. C. Chu, and A. Aizawa, ‘‘Genetics-Based Learning of New Heuristics: Rational Scheduling of Experiments and Generalization,’’ *IEEE Trans. on Knowledge and Data Engineering*, vol. 7, no. 5, pp. 763-785, IEEE, Oct. 1995.

[107] E.H. Wefald and S.J. Russell, ‘‘Adaptive Learning of Decision-Theoretic Search Control Knowledge,’’ *Machine Learning*, pp. 408-411, Kluwer Academic Pub., Boston, MA, 1989.

[108] P. H. Winston, *Artificial Intelligence*, Second Edition, Addison-Wesley, Reading, MA, 1984.

[109] Y. Zhang, ‘‘Parallel Algorithms for Combinatorial Search Problems,’’ Technical Report No. UCB/CSD 89/543, Univ. of California, Berkeley, CA, Nov. 1989.

APPENDIX A

EXTREME-CASE ANALYSES FOR REAL-TIME SEARCH

In this appendix, we show the best-case and worst-case analyses for various strategies used in the RTS algorithm.

A.1. Approximation by Linear-Gradient

A.1.1. Best-case analysis

The best-case value of ε_m can be calculated by substituting $m_{\max}$ into Eq. (4.5). The best-case $\Delta\varepsilon_{RTS[LG,I](T)}$ can be derived accordingly as

$$\varepsilon_{RTS[LG,I](T)} - \varepsilon_{OPTA^*(T)} < \Delta\varepsilon_{RTS[LG,I](T)} = \left(\varepsilon_0 - \varepsilon_{0,OPTA^*}\right) + H(T) + F^{best}_{LG,I}(g,\ T),$$

$$\text{where } H(T) = \left(\varepsilon_{0,OPTA^*} \log_{t_{a^*}} T - \varepsilon_0 \log_{t_d} T\right),$$

$$F^{best}_{LG,I}(g,\ T) = \varepsilon_0\, g - \varepsilon_0 \log_{t_d}\left(t_d{}^{g} - 1 + T^{-1}\right).$$

$F^{best}_{LG,I}(g,T)$ differs from $F^{worst}_{LG,I}(g,T)$ by $\varepsilon_0\, g$, which is quite small for small g. This implies that for small g the best-case bound is almost equal to the worst-case bound. However, when g is too small, overheads incurred in evaluating previous (not the last completed) GDFSs become significant, though the worst-case and best-case bounds do not differ much.

Taking the partial derivative of $\Delta\varepsilon_{RTS[LG,I](T)}$ with respect to g, and equating the derivative to zero, we have $t_d{}^{g} = t_d{}^{g} - 1 + 1/T$, which does not have real roots. As a result, there is no convex or concave optimality in $\Delta\varepsilon_{RTS[LG,I](T)}$. One of the extreme cases of g values is the best, and is found to be $g^{*best} = \log_{t_d}(T-1)$. This bound indicates that g should be set to be as large as possible if the last GDFS always completes (the best case). However, in general this

is impractical, because we do not know when a GDFS can complete. Since the best case is impractical, we will not analyze it further.

A.1.2. Average-case analysis

Comparing the worst and best case values of $\Delta\varepsilon_{RTS[LG,I](T)}$, we find that the only difference between $F^{worst}_{LG,I}(g,T)$ and $F^{best}_{LG,I}(g,T)$ is $\varepsilon_0\, g$. For analysis, we can parameterize this function for the general case.

$$F^{avg}_{LG,I}(\rho, g, T) = (\rho + 1)\,\varepsilon_0\, g - \varepsilon_0 \log_{t_d}\left(t_d{}^{g} - 1 + T^{-1}\right),$$

where $0 \le \rho \le 1$, and $\rho = 1$ in $F^{worst}_{LG,I}(g,T)$, $\rho = 0$ in $F^{best}_{LG,I}(g,T)$ is $\varepsilon_0\, g$.

Lemma A.1. *In terms of minimizing $\Delta\varepsilon_{RTS[LG,I](T)}$ in the average case, the best gradient factor is*

$$g^{*avg} = \log_{t_d}\left[\left(1 + \rho^{-1}\right)\left(1 - T^{-1}\right)\right] \approx \log_{t_d}(1 + \rho^{-1}). \qquad \text{(A.1)}$$

Proof. Taking the partial derivative of $\Delta\varepsilon_{RTS[LG,I](T)}$ with respect to g, equating the partial derivative to zero, and solving the equation for the root, we have the result in Eq. (A.1). This approximation is acceptable as T usually is much larger than 1. □

In the last unfinished GDFS, the AAD achieved in the past GDFSs may be better than the AAD achieved in the last completed GDFS, especially when the time scheduled for the last GDFS is less than the time expended in the last complete GDFS. Therefore, in general, we assume that $\rho = 1$ when $\tau_{k+1} < t_k$, where τ_{k+1} is the amount of time scheduled for the (m+1)-th GDFS. When $\tau_{k+1} \ge t_k$, the AAD achieved in the last unfinished GDFS is generally no worse than the AAD achieved in the last completed GDFS. The SE model says that the AAD achieved is logarithmic with time expended, namely, ρ is assumed to be logarithmic

with τ_{k+1}. Hence, it is reasonable to assume that the bound of AAD achieved is also logarithmic with time expended. To summarize, ρ can be formulated as

$$\rho(\tau_{k+1}) \;=\!del\; \begin{cases} 1 & \text{if } 0 < \tau_{k+1} \le t_k \;\text{ (worst case)}, \\ (k+1) - \dfrac{1}{g}\log_{t_d} \tau_{k+1} & \text{if } t_k < \tau_{k+1} \le t_{k+1} \;\text{ (desired case)}, \end{cases} \tag{A.2}$$

which satisfies the boundary conditions $\rho(0) = \rho(t_k) = 1$ and $\rho(t_{k+1}) = 0$. Note that it is impossible that $\tau_{k+1} > t_{k+1}$ because the *k*-th GDFS is the last one. Let $\phi(t)$ be the probability density function of τ_{k+1}; we have

$$\phi(t) \;=\!del\; \frac{1}{t_{k+1}}$$

for uniform distribution of τ_{k+1}. E[g^{*avg}] is too difficult to be solved analytically. It cannot be solved numerically since we do not know t_d ahead of time. Here, we adopt an approximation of E[g^{*avg}] that is independent of t_d. This is done by substituting E[ρ] for ρ in Eq. (A.1). In the following lemma, we derive the approximate E[g^{*avg}].

Lemma A.2. *Assuming a uniform distribution of* τ_{k+1} *in* [0, t_{k+1}], *the expected value of* ρ *is*

$$\mathrm{E}[\rho^{avg}] \;=\; \frac{1 - t_d^{\,-g}}{g \ln t_d}\,. \tag{A.3}$$

Proof. $\mathrm{E}[\rho] = \int_{-\infty}^{\infty} \rho(t)\,\phi(t)\,\mathrm{d}t = \frac{1}{t_{k+1}}\left[\int_0^{t_k} \rho(t)\,\mathrm{d}t + \int_{t_k}^{t_{k+1}} \rho(t)\,\mathrm{d}t\right] = \frac{1 - t_d^{\,-g}}{g \ln t_d}.$ □

Theorem A.1. *In terms of* E[ρ^{avg}], *the expected value of the best* g^* *in the average case is*

$$\mathrm{E}[g^{*avg}] \;\approx\; \log_{t_d} 2.537.$$

Proof. Applying Eq. (A.3) to Eq. (A.1), we have

$$\mathrm{E}[g^{*avg}] \approx \log_{t_d}\left(1+\mathrm{E}[\rho^{avg}]^{-1}\right) = \log_{t_d}\left(1+\frac{\bar{g}\ln t_d}{1-t_d^{-\bar{g}}}\right),$$

where $\bar{g}=\mathrm{E}[g^{*avg}]$. By rearranging the previous equation, we have $t_d^{\bar{g}}+t_d^{-\bar{g}}-\ln t_d^{\bar{g}}-2=0$. Solving the roots numerically for $t_d^{\bar{g}}$, we have $t_d^{\bar{g}}$ = 1 or 2.537, where $t_d^{\bar{g}}=1$ is infeasible because it requires $\tau_i=1$ for every *i*. □

The above theorem provides a good approximation to estimate g^{*avg} because changes of g^{*avg} for a large spectrum of t_d are small. For instance, for t_d *member* $[10^4, 10^{10}]$, g^{*avg} *member* $[0.040, 0.101]$.

In Eq. (A.2), the desired case occurs when the last GDFS is not wasted and makes progress in improving the feasible solution already found. Due to incumbent propagation, the desired case is more likely to occur because the approximation pruning value becomes small. The following lemma and theorem are concentrated on the desired case.

Lemma A.3. *Assuming that* τ_{k+1} *can only happen in* $[t_k, t_{k+1}]$ *and* τ_{k+1} *is uniformly distributed in this interval, the expected value of desired* ρ *is*

$$\mathrm{E}[\rho^{desired}] = \frac{t_d^{\,g}-g\ln t_d-1}{g\,(t_d^{\,g}-1)\ln t_d}. \tag{A.4}$$

Proof. By definition, we have

$$\mathrm{E}[\rho^{desired}] = \frac{1}{t_{k+1}-t_k}\int_{t_k}^{t_{k+1}}\rho(t)\,\mathrm{d}t.$$

This lemma can be proved by simple calculus. □

Theorem A.2. *In terms of* $\mathrm{E}[\rho^{desired}]$*, the expected value of the best desired-case gradient factor in the average case is*

$$\mathrm{E}[g^{*desired}] \approx \log_{t_d} 3.513 .$$

Proof. Applying Eq. (A.4) to Eq. (A.1), we have

$$\mathrm{E}[g^{*desired}] \approx \log_{t_d}\left(1+\mathrm{E}[\rho]^{-1}\right) = \log_{t_d}\left(1+[\bar{g}\,(t_d^{\,\bar{g}}-1)\ln t_d]/[t_d^{\,\bar{g}}-\bar{g}\ln t_d - 1]\right),$$

where $\bar{g}=\mathrm{E}[g^{*desired}]$. By rearranging the previous equation, we have $t_d^{\,\bar{g}} - 2\ln t_d^{\,\bar{g}} - 1 = 0$. Solving the roots numerically for $t_d^{\,\bar{g}}$, we have $t_d^{\,\bar{g}}$ = 1 or 3.513, where $t_d^{\,\bar{g}} = 1$ is infeasible because it requires $\tau_i = 1$ for every i. □

The above theorem provides a very good approximation to estimate $g^{*desired}$ in the average case, because changes of g^* are small for a large spectrum of t_d. For instance, for t_d *member* [10^4, 10^{10}], $g^{*desired}$ *member* [0.055, 0.136]. In this thesis, we use $\mathrm{E}[g^{*desired}]$ to evaluate RTS with approximation alone.

A.2. Approximation by First-Order-Regression

A.2.1. Best-case analysis

The best-case value of ε_m can be calculated by substituting $m_{\max}$ into Eq. (4.9). The worst-case bound $\Delta\varepsilon_{RTS[FR,I](T)}$ in the best case can be derived accordingly as

$$\varepsilon_{RTS[FR,I](T)} - \varepsilon_{OPTA^*(T)} < \Delta\varepsilon_{RTS[FR,I](T)} = \left(\varepsilon_0 - \varepsilon_{0,OPTA^*}\right) + H(T) + F^{best}_{FR,I}(r,\,T),$$

$$\text{where } H(T) = \left(\varepsilon_{0,OPTA^*}\log_{t_{a^*}} T - \varepsilon_0 \log_{t_d} T\right),$$

$$F^{best}_{FR,I}(r,\,T) = \varepsilon_0 \log_{t_d} r - \varepsilon_0 \log_{t_d}\left(r - 1 + T^{-1}\right).$$

$F^{best}_{FR,I}(r, T)$ differs from $F^{worst}_{FR,I}(r, T)$ by $\varepsilon_0 \log_{t_d} r$ which is quite small for small r. This implies that for small r the best-case bound is almost equal to the worst-case bound. However, when r is too small, overheads incurred in solving previous (not the last completed) GDFSs become significant and performance may worsen.

Taking the partial derivative of $\Delta\varepsilon_{RTS[FR,I](T)}$ with respect to r, and equating the derivative to zero, the equation becomes $r = r - 1 + 1/T$, which does not have real roots. As a result, there is no convex or concave optimality in $\Delta\varepsilon_{RTS[FR,I](T)}$. Therefore, one of the extreme cases of r values is the best. In terms of minimizing $\Delta\varepsilon_{RTS[FR,I](T)}$ in the best case, an optimal choice of the growth rate is $r^* = T - 1$, which indicates that r should be set as large as possible, if the last GDFS always completes (which occurs in the best case). In general, this condition is impossible because we do not know when a specific GDFS can be completed. Since the best case is impractical, we will not analyze it further.

A.2.2. Average-case analysis

Comparing the worst-case and best-case values of $\Delta\varepsilon_{RTS[FR,I](T)}$, we find that the only difference is $\varepsilon_0 \log_{t_d} r$ between $F^{worst}_{FR,I}(r, T)$ and $F^{best}_{FR,I}(r, T)$. For analysis, we can parameterize this function for the general case.

$$F^{avg}_{FR,I}(\rho, r, T) = (\rho + 1)\,\varepsilon_0 \log_{t_d} r - \varepsilon_0 \log_{t_d} \left(r - 1 + T^{-1} \right),$$

where $0 \le \rho \le 1$, and $\rho = 1$ for the worst case, while $\rho = 0$ for the best case.

Lemma A.4. *The best growth rate for minimizing $\Delta\varepsilon_{RTS[FR,I](T)}$ in the average case is*

$$r^{*avg} = \left(1 + \rho^{-1} \right) \left(1 - T^{-1} \right) \approx 1 + \rho^{-1}. \qquad \text{(A.5)}$$

Proof. Taking the partial derivative of $\Delta\varepsilon_{RTS[FR,I](T)}$ with respect to r, equating the partial derivative to zero, and solving the equation for the root, we have the result shown in Eq. (A.5). This approximation is acceptable as T usually is much larger than 1. □

The distribution of ρ is described in Eq. (A.2). However, it is very unlikely that $t_k > \tau_{k+1}$, as the incumbent used in the kth GDFS is used as a starting point, the $(k+1)$st GDFS. Therefore, it is reasonable to assume that it is unlikely for $\rho = 1$; that is, the probability that $\sum_{i=0}^{k} t_i < T \le \sum_{i=0}^{k} t_i + t_k$ is negligible. In other words, we only consider the desired case in Eq. (A.2).

Lemma A.5. *Assuming that* τ_{k+1} *can only be in the range* $[t_k, t_{k+1}]$ *and is uniformly distributed in this interval, the expected value of* ρ *in the desired case is*

$$\mathrm{E}[\rho] = \frac{r - \ln r - 1}{(r-1)\ln r}. \tag{A.6}$$

Proof. $\mathrm{E}[\rho] = \int_{-\infty}^{\infty} \rho(t)\,\phi(t)\,\mathrm{d}t = \frac{1}{t_{k+1} - t_k} \int_{t_k}^{t_{k+1}} \rho(t)\,\mathrm{d}t.$ The lemma can be proved by simple calculus. □

Theorem A.3. *The expected value of the best* r^* *in the average case is*

$$\mathrm{E}[r^{*avg}] \approx 3.513. \tag{A.7}$$

Proof. Applying Eq. (A.6) to Eq. (A.5), we have

$$\mathrm{E}[r^{*avg}] \approx 1 + \mathrm{E}[\rho]^{-1} = 1 + \frac{(\bar{r}-1)\ln\bar{r}}{\bar{r} - \ln\bar{r} - 1},$$

where $\bar{r} = \mathrm{E}[r^{*avg}]$. By rearranging the previous equation, we have $\bar{r} - 2\ln\bar{r} - 1 = 0$. Solving the roots numerically for $\bar{r}$, we have $\bar{r}$ = 1 or 3.513. However, $\bar{r} = 1$ is infeasible because it requires $\tau_i = 1$ for every i. □

The above theorem provides a very good approximation to estimate r^* in the average case. Unlike the estimate for the linear-gradient strategy, this estimate is independent of t_d.

A.3. Thresholding by Linear-Gradient

A.3.1. Best-case analysis

The best-case value of ε_m can be calculated by substituting $m_{\max}$ into Eq. (4.18). The worst bound $\Delta\varepsilon_{RTS[I,LG](T)}$ in the best case can be derived accordingly as

$$\varepsilon_{RTS[I,LG](T)} - \varepsilon_{OPTA^*(T)} < \Delta\varepsilon_{RTS[I,LG](T)} = \left(\varepsilon_0 - \varepsilon_{0,OPTA^*}\right)\left(1 - \log_{t_{a^*}} T\right) + F^{best}_{I,LG}(g, T),$$

$$\text{where } F^{best}_{I,LG}(g, T) = \lambda\,\varepsilon_0\, g - \varepsilon_0 \log_{t_{a^*}}\left(t_{a^*}{}^{\lambda g} - 1 + T^{-1}\right).$$

Note that $F^{best}_{I,LG}(g,T)$ in the above equation differs from the worst-case one by $\lambda\,\varepsilon_0\, g$, which is small for small g. This implies that for small g the best-case bound is almost equal to the worst-case bound. However, when g is too small, overheads incurred in solving previous (not the last completed) GDFSs become significant, and the final solution may become worse.

By following similar derivations in Section A.2.1, we can have $g^{*best} = \frac{1}{\lambda}\log_{t_{a^*}}(T-1) = \log_{t_d}(T-1)$, which is impractical as described in that section.

A.3.2. Average-case analysis

Comparing the worst-case $\Delta\varepsilon_{RTS[I,LG](T)}$ and the best-case one, we find that the only difference in $F_{I,LG}^{worst}(g,T)$ and $F_{I,LG}^{best}(g,T)$ is $\lambda\,\varepsilon_0\,g$. For analysis, we can parameterize this function for the general case.

$$F_{I,LG}^{avg}(\rho, g, T) = (\rho + 1)\,\lambda\,\varepsilon_0\,g - \varepsilon_0 \log_{t_{a^*}} \left(t_{a^*}{}^{\lambda g} - 1 + T^{-1} \right),$$

where $0 \le \rho \le 1$, and $\rho = 1$ for the worst case while $\rho = 0$ for the best case.

Lemma A.6. *In terms of minimizing $\Delta\varepsilon_{RTS[I,LG](T)}$ in the general case, the best gradient factor is*

$$\begin{aligned} g^{*avg} &= \frac{1}{\lambda} \log_{t_{a^*}} \left[\left(1 + \rho^{-1} \right) \left(1 - T^{-1} \right) \right] \\ &= \log_{t_d} \left[\left(1 + \rho^{-1} \right) \left(1 - T^{-1} \right) \right] \\ &\approx \log_{t_d} \left(1 + \rho^{-1} \right). \end{aligned} \tag{A.8}$$

Proof. Taking the partial derivative of $\Delta\varepsilon_{RTS[I,LG](T)}$ with respect to *g*, equating the partial derivative to zero, and solving the equation for the root, we have the result in Eq. (A.8). The second equality holds due to Eq. (4.17). The approximation is acceptable as *T* usually is much larger than 1. □

Note that Eq. (A.8) is the same as Eq. (A.1). Hence, we can follow the same analysis in Section A.1.2., and the analysis is not shown here.

A.4. Thresholding by First-Order-Regression

A.4.1. Best-case analysis

The best-case value of ε_m can be calculated by substituting $m_{\max}$ into Eq. (4.20). The best-case bound $\Delta\varepsilon_{RTS[I,FR](T)}$ can be derived accordingly as

$$\varepsilon_{RTS[I,FR](T)} - \varepsilon_{OPTA^*(T)} < \Delta\varepsilon_{RTS[I,FR](T)} = \left(\varepsilon_0 - \varepsilon_{0,OPTA^*}\right)\left(1 - \log_{t_{a^*}} T\right) + F^{best}_{I,FR}(r, T),$$

$$\text{where } F^{best}_{I,FR}(r, T) = \varepsilon_0 \log_{t_{a^*}} r - \varepsilon_0 \log_{t_{a^*}}\left(r - 1 + T^{-1}\right).$$

$F^{best}_{I,FR}(r, T)$ in the above equation differs from $F^{worst}_{I,FR}(r, T)$ by $\varepsilon_0 \log_{t_{a^*}} r$, which is quite small for small r. This implies that for small r the best-case bound is almost equal to the worst-case bound. However, when r is too small, overheads incurred in solving previous (not the last completed) GDFSs become significant and the final solution may become worse.

Taking the partial derivative of $\Delta\varepsilon_{RTS[I,FR](T)}$ with respect to r, equating the derivative to zero, and solving for the root, we can have $r^{*best} = T - 1$, which is not practical as described in Section A.2.1.

A.4.2. Average-case analysis

Comparing the worst-case and best-case values of $\Delta\varepsilon_{RTS[I,FR](T)}$, we find that the only difference between $F^{worst}_{I,FR}(r, T)$ and $F^{best}_{I,FR}(r, T)$ is $\varepsilon_0 \log_{t_{a^*}} r$. For analysis, we can parameterize this function for the average case.

$$F^{avg}_{I,FR}(\rho, r, T) = (\rho + 1)\,\varepsilon_0 \log_{t_{a^*}} r - \varepsilon_0 \log_{t_{a^*}}\left(r - 1 + T^{-1}\right),$$

where $0 \le \rho \le 1$, and $\rho = 1$ for the worst case while $\rho = 0$ for the best case. The following lemma shows the best *r* for minimizing $\Delta\varepsilon_{RTS[I,FR](T)}$ in the average case.

Lemma A.7. *The best r for minimizing* $\Delta\varepsilon_{RTS[I,FR](T)}$ *is*

$$r^{*avg} = \left(1 + \rho^{-1}\right)\left(1 - T^{-1}\right) \approx 1 + \rho^{-1}. \quad \text{(A.9)}$$

Proof. The proof is similar to that for Lemma A.4. □

Note that Eqs. (A.9) and (A.5) are identical. Therefore, we can apply the results we have derived to analyze RTS[FR,I] to RTS[I,FR].

APPENDIX B

EXPERIMENTAL RESULTS OF LEARNING SEARCH HEURISTICS

In this appendix, we show the cluster plots, cluster sets, and heuristics discussed or described in Chapter 5.

B.1. Experimental Results of Learning Search Heuristics for the KS Problems

In this section, we will show the cluster plots, cluster sets, and learned decomposition and selection heuristics used in guided depth-first search for solving the KS problem.

B.1.1. Learning decomposition heuristics used in GDFS

In this section, we show in Figure B.1 the cluster plots of learned decomposition heuristics used in guided depth-first search to solve an optimal solution of the KS problem. The learning strategy is the MR strategy with a random seed 94035. Following Figure B.1, we list cluster sets shown in the plot. In Table B.1, we summarize the most generalized cluster set and the most specialized cluster set, and the chosen cluster set. In Table B.2, we show the learned decomposition heuristics mentioned in Table B.1.

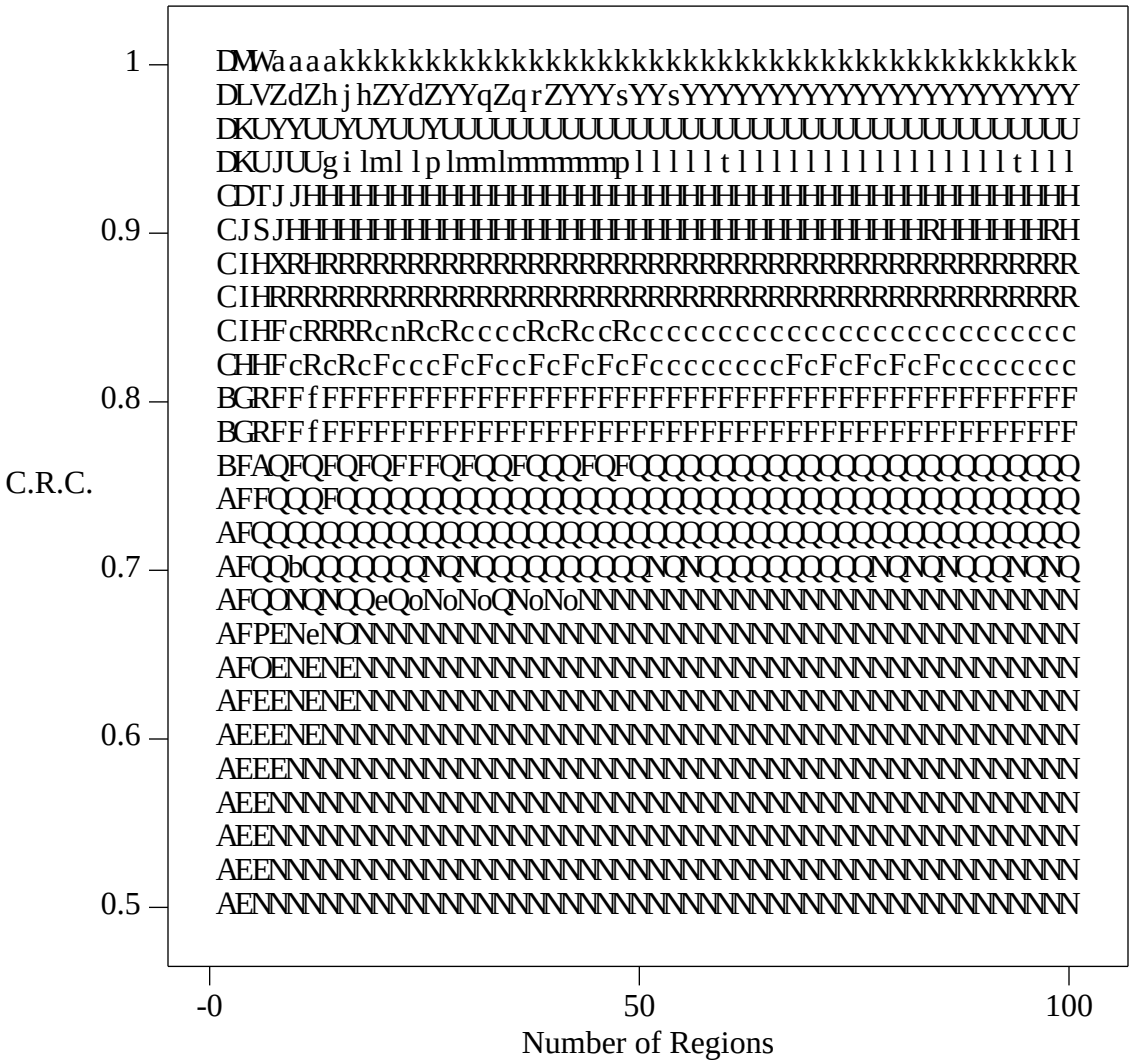


Figure B.1. Cluster plot obtained from learning decomposition heuristics used in guided depth-first search to solve an optimal solution of the KS problem, where the learning is performed with the MR strategy and a random seed 94035.

Listing of Cluster Sets

Cluster Set A: ((11) (7 9 10) (1 2 3 4 5 6 8 12))
Cluster Set B: ((11) (10) (7 9) (1 2 3 4 5 6 8 12))
Cluster Set C: ((11) (7) (8 12) (9 10) (1 2 3 4 5 6))
Cluster Set D: ((11) (7) (9) (10) (8 12) (1 2 3 4 5 6))
Cluster Set E: ((9 11) (1 2 3 4 5 6 7 8 10 12))
Cluster Set F: ((9 11) (1 2 4 5 6) (3 7 8 10 12))

Cluster Set G: ((11) (3 8 12) (7 9 10) (1 2 4 5 6))
Cluster Set H: ((11) (7 9) (8 10 12) (1 2 3 4 5 6))
Cluster Set I: ((11) (9) (3 7) (8 10 12) (1 2 4 5 6))
Cluster Set J: ((11) (7) (9) (8 10 12) (1 2 3 4 5 6))
Cluster Set K: ((11) (3) (7) (9) (10) (8 12) (1 2 4 5 6))
Cluster Set L: ((11) (3) (7) (8) (9) (10) (12) (1 2 4 5 6))
Cluster Set M: ((6) (11) (3) (7) (8) (9) (10) (12) (1 2 4 5))
Cluster Set N: ((7 9 11) (1 2 3 4 5 6 8 10 12))
Cluster Set O: ((9 11) (1 2 4) (3 5 6 7 8 10 12))
Cluster Set P: ((9 11) (7 10) (1 2 3 4 5 6 8 12))
Cluster Set Q: ((9 11) (1 2 4 6) (3 5 7 8 10 12))
Cluster Set R: ((11) (7 9) (3 8 10 12) (1 2 4 5 6))
Cluster Set S: ((11) (7 9) (10 12) (1 2 3 4 5 6 8))
Cluster Set T: ((11) (7) (9) (10 12) (1 2 4) (3 5 6 8))
Cluster Set U: ((11) (7) (9) (3 8) (10 12) (1 2 4 5 6))
Cluster Set V: ((11) (7) (8) (9) (10) (12) (3 5) (1 2 4 6))
Cluster Set W: ((6) (11) (3) (5) (7) (8) (9) (10) (12) (1 2 4))
Cluster Set X: ((11) (9) (7 10) (3 8 12) (1 2 4 5 6))
Cluster Set Y: ((11) (7) (9) (10) (3 5) (8 12) (1 2 4 6))
Cluster Set Z: ((11) (7) (8) (9) (3 5) (10 12) (1 2 4 6))
Cluster Set a: ((6) (11) (3) (4) (5) (7) (8) (9) (10) (12) (1 2))
Cluster Set b: ((10) (7 9 11) (1 2 3 4 5 6 8 12))
Cluster Set c: ((9 11) (3 7 10 12) (1 2 4 5 6 8))
Cluster Set d: ((11) (3) (5) (7) (8) (9) (10 12) (1 2 4 6))
Cluster Set e: ((7) (9 11) (1 2 3 4 5 6 8 10 12))
Cluster Set f: ((11) (7 9 10 12) (1 2 3 4 5 6 8))
Cluster Set g: ((11) (7 9) (1 2 4) (3 5 6) (8 10 12))
Cluster Set h: ((11) (3) (5) (7) (9) (10) (8 12) (1 2 4 6))
Cluster Set i: ((11) (7) (9) (3 5) (8 10 12) (1 2 4 6))
Cluster Set j: ((11) (3) (7) (8) (9) (1 5) (10 12) (2 4 6))
Cluster Set k: ((1) (6) (11) (2) (3) (4) (5) (7) (8) (9) (10) (12))
Cluster Set l: ((11) (3) (7) (9) (8 10 12) (1 2 4 5 6))
Cluster Set m: ((11) (7) (9) (1 3 5) (2 4 6) (8 10 12))
Cluster Set n: ((9 11) (7 10) (1 2 4 6) (3 5 8 12))
Cluster Set o: ((9 11) (2 4 6) (1 3 5 7 8 10 12))
Cluster Set p: ((11) (7 9) (1 3 5) (2 4 6) (8 10 12))
Cluster Set q: ((11) (3) (7) (9) (10) (1 5) (8 12) (2 4 6))
Cluster Set r: ((11) (3) (7) (8) (9) (4 6) (10 12) (1 2 5))
Cluster Set s: ((11) (3) (7) (9) (10) (4 6) (8 12) (1 2 5))
Cluster Set t: ((11) (3) (7 9) (8 10 12) (1 2 4 5 6))

Table B.1. Summary of the most generalized cluster set, the most specialized cluster set, and the chosen cluster set (denoted as R') from learning decomposition heuristics of the KS problems, where the cluster plot is shown in Figure B.1. (Cluster set R' is modified from cluster set R by moving subdomain 11 to the cluster containing subdomains 7 and 9, because they conclude the same best heuristic.)

The Most Generalized Cluster Set			
Subdomains	Heuristics ID	Probability of Wins	Average Speedup
1,2,3,4,5,6,7,8,9,10,11,12	1	0.692	1.000
The Most Specialized Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
1	1	0.896	1.000
2	1	0.735	1.000
3	1	0.912	1.000
4	1	0.902	1.000
5	1	0.852	1.000
6	1	0.875	1.000
7	255	0.970	1.288
8	1	0.701	1.000
9	825	0.874	1.651
10	1	0.707	1.000
11	255	0.999	2.908
12	1	0.840	1.000
The Chosen Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
7,9,11	255	0.926	1.885
1,2,3,4,5,6,8,10,12	1	0.754	1.000

Table B.2. Summary of the learned heuristics mentioned in Table B.1.

ID	Learned Heuristics	Simplified Heuristics
1	(p / w)	$\frac{p}{w}$
255	((((p - mp) - (s + w)) - s) / (_zerop ((((p - mp) - (s + w)) - (1 / (_zerop ((((p - mp) - (s + w)) * (((p - mp) - (s + w)) / (_zerop (((LC * s) + w))))))))))))	$\frac{-mp+p-2*s-w}{-mp+p-s-w-\frac{LC*s+w}{mp-p+s+w^2}}$
825	(s * ((p / (_zerop (w))) / (_zerop ((((p / (_zerop (w))) - mw) / (_zerop ((np + ((p / (_zerop (w))) - mw)))))))))	$\frac{p*s*(-mw+np+\frac{p}{w})}{p-mw*w}$

B.1.2. Learning selection heuristics used in GDFS

In this section, we show in Figure B.2 the cluster plots of learned selection heuristics used in a guided depth-first search to solve an optimal solution of the KS problem, where the decomposition heuristics are those in the chosen set shown in Table B.1. The learning strategy is the MR strategy with a random seed 61801. Following Figure B.2, we list cluster sets shown in the plot. In Table B.3, we summarize the most generalized cluster set, the most specialized cluster set, and the chosen cluster set. In Table B.4, we show the learned selection heuristics mentioned in Table B.3.

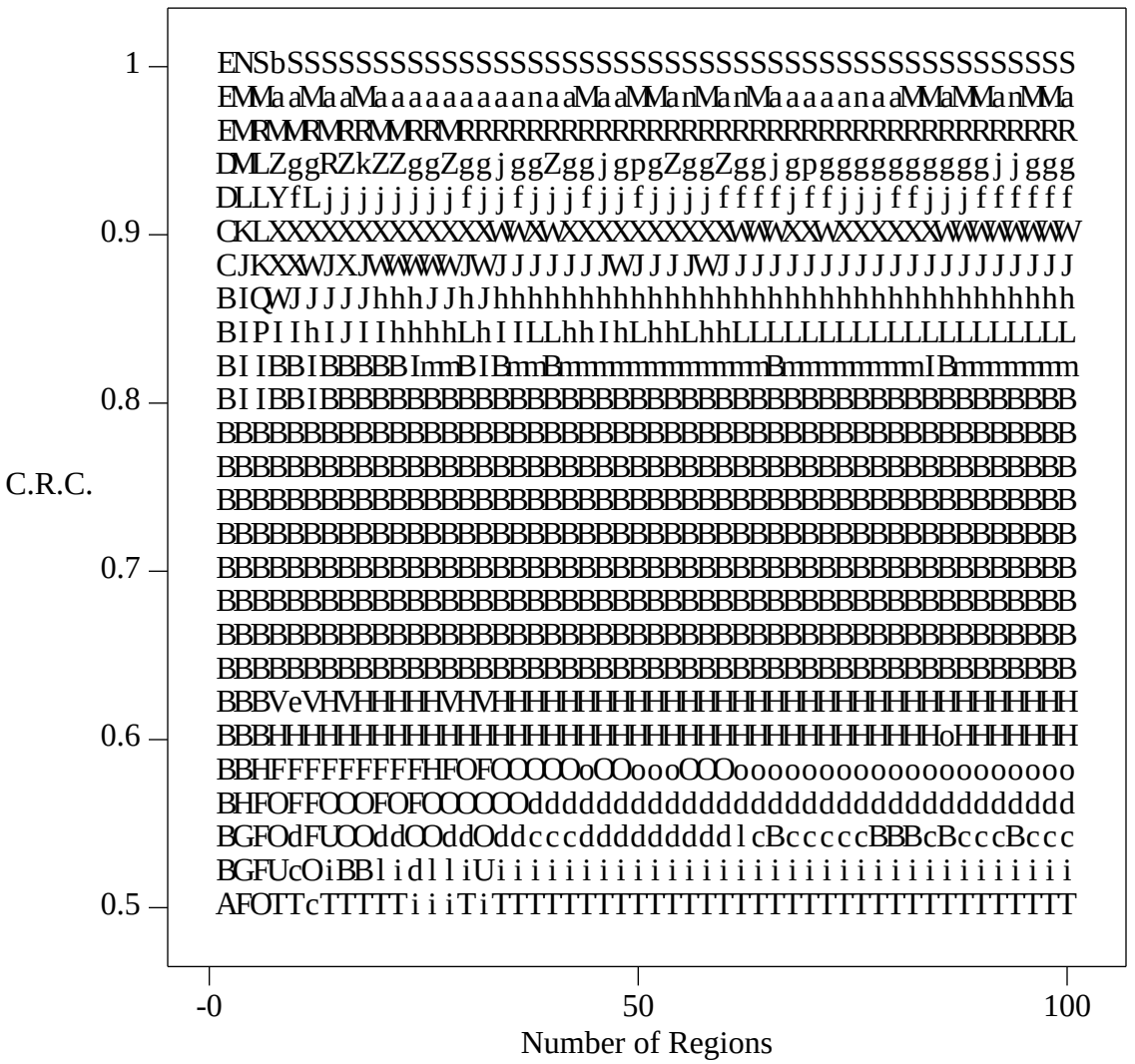


Figure B.2. Cluster plot obtained from learning selection heuristics used in guided depth-first search to solve an optimal solution of the KS problem, where the learning is performed with the MR strategy and a random seed 61801.

Listing of Cluster Sets

Cluster Set A: ((7 9) (1 2 3 4 5 6 8 10 11 12))
Cluster Set B: ((7 9 11) (1 2 3 4 5 6 8 10 12))
Cluster Set C: ((3 6) (7 9 11) (1 2 4 5 8 10 12))
Cluster Set D: ((3) (12) (7 9 11) (1 2 4 5 6 8 10))
Cluster Set E: ((6) (11) (3) (12) (7 9) (1 2 4 5 8 10))

Cluster Set F: ((3 6 10 11 12) (1 2 4 5 7 8 9))
Cluster Set G: ((1 2 3 4 5 8) (6 7 9 10 11 12))
Cluster Set H: ((6 7 9 10 11) (1 2 3 4 5 8 12))
Cluster Set I: ((11) (7 9) (1 2 3 4 5 6 8 10 12))
Cluster Set J: ((11) (7 9) (1 2 4) (3 5 6 8 10 12))
Cluster Set K: ((11) (3 6) (7 9) (1 2 4 5 8 10 12))
Cluster Set L: ((11) (3) (7 9) (1 2 4 5 6 8 10 12))
Cluster Set M: ((11) (3) (7 9) (6 10 12) (1 2 4 5 8))
Cluster Set N: ((6) (11) (3) (10) (12) (5 8) (7 9) (1 2 4))
Cluster Set O: ((1 2 4 5 7 9) (3 6 8 10 11 12))
Cluster Set P: ((11) (2 4) (7 9) (1 3 5 6 8 10 12))
Cluster Set Q: ((11) (7 9) (3 6 8 12) (1 2 4 5 10))
Cluster Set R: ((11) (3) (7 9) (1 2 4 5) (6 8 10 12))
Cluster Set S: ((1) (6) (11) (2) (3) (4) (5) (7) (8) (9) (10) (12))
Cluster Set T: ((9) (1 2 3 4 5 6 7 8 10 11 12))
Cluster Set U: ((1 2 5 9) (3 4 6 7 8 10 11 12))
Cluster Set V: ((7 9 10 11) (1 2 3 4 5 6 8 12))
Cluster Set W: ((11) (7 9) (1 2 4 5) (3 6 8 10 12))
Cluster Set X: ((11) (7 9) (3 6 10 12) (1 2 4 5 8))
Cluster Set Y: ((11) (7 9) (3 6 12) (1 2 4 5 8 10))
Cluster Set Z: ((11) (3) (7 9) (1 2 5) (4 6 8 10 12))
Cluster Set a: ((11) (3) (6 8) (7 9) (10 12) (1 2 4 5))
Cluster Set b: ((6) (11) (3) (4) (5) (7) (8) (9) (10) (12) (1 2))
Cluster Set c: ((1 2 7 9) (3 4 5 6 8 10 11 12))
Cluster Set d: ((1 2 5 7 9) (3 4 6 8 10 11 12))
Cluster Set e: ((6 7 9 11) (1 2 3 4 5 8 10 12))
Cluster Set f: ((11) (7 9) (3 10 12) (1 2 4 5 6 8))
Cluster Set g: ((11) (3) (1 2) (7 9) (4 5 6 8 10 12))
Cluster Set h: ((11) (1 2) (7 9) (3 4 5 6 8 10 12))
Cluster Set i: ((9 11) (1 2 3 4 5 6 7 8 10 12))
Cluster Set j: ((11) (3 12) (7 9) (1 2 4 5 6 8 10))
Cluster Set k: ((11) (3) (12) (7 9) (1 2 4 5 6 8 10))
Cluster Set l: ((1 2 9) (3 4 5 6 7 8 10 11 12))
Cluster Set m: ((9) (7 11) (1 2 3 4 5 6 8 10 12))
Cluster Set n: ((11) (3) (7 9) (10 12) (1 2 5) (4 6 8))
Cluster Set o: ((1 2 3 4 5 12) (6 7 8 9 10 11))
Cluster Set p: ((11) (3) (7 9) (10 12) (1 2 4 5 6 8))

Table B.3. Summary of the most generalized cluster set, the most specialized cluster set, and the chosen cluster set (cluster set Z) from learning selection heuristics of the KS problems, where the cluster plot is shown in Figure B.2.

The Most Generalized Cluster Set			
Subdomains	Heuristics ID	Probability of Wins	Average Speedup
1,2,3,4,5,6,7,8,9,10,11,12	305	0.674	1.546
The Most Specialized Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
1	352	0.535	1.288
2	389	0.577	1.169
3	305	0.781	3.412
4	74	0.673	1.177
5	355	0.611	1.572
6	74	0.709	1.910
7	688	0.881	1.003
8	74	0.679	1.751
9	305	0.914	1.004
10	74	0.864	1.623
11	295	0.978	1.481
12	74	0.899	1.661
The Chosen Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
11	295	0.978	1.481
3	305	0.781	3.412
7,9	688	0.794	1.002
1,2,5	305	0.554	1.336
4,6,8,10,12	74	0.765	1.625

Table B.4. Summary of the learned heuristics mentioned in Table B.3.

ID	Learned Heuristics	Simplified Heuristics
1	l	l
74	(((u - (LC * g)) - dg) / (_zerop ((u + (LC * h)))))	$\frac{-dg-g*LC+u}{h*LC+u}$
295	((dl * ((LC * h) - dg)) - ((l + (- h3)) - dh))	dh−dg*dl+h3−l+dl*h*LC
305	(h + (h * dl))	h+h*dl
352	(((dl * ((LC * h) - dg)) - ((LC * h) + (- h3))) + l)	−(dg*dl)+h3+l−h*LC+dl*h*LC
355	(h1 + ((l + (- h3)) - dh))	−dh+h1−h3+l
389	((dl * ((LC * h) - dg)) - (h + (h * dl)))	−(dg*dl)−h−dl*h+dl*h*LC
688	((h3 + ((dl * ((LC * h) - dg)) - ((l + (- h3)) - dh))) + (((dl * ((LC * h) - dg)) - ((l + (- h3)) - dh)) / (_zerop (dl)) / (_zerop (((LC * h) - dg))) / (_zerop (h3))))	$dh+2*h3-l+dl*(-dg+h*LC)+\frac{dh-dg*dl+h3-l+dl*h*LC}{dl*h3*(-dg+h*LC)}$

B.2. Experimental Results of Learning Search Heuristics for the VC Problems

In this section, we will show the cluster plots, cluster sets, and learned decomposition and selection heuristics used in guided depth-first search for solving the VC problem.

B.2.1. Learning decomposition heuristics used in GDFS

In this section, we show in Figure B.3 the cluster plots of learned decomposition heuristics used in a guided depth-first search to solve an optimal solution of the VC problem. The learning strategy is the MR strategy with a random seed 94035. Following Figure B.3, we list cluster sets shown in the plot. In Table B.5, we summarize the most generalized cluster set, the most specialized cluster set, and the chosen cluster set. In Table B.6, we show the learned decomposition heuristics mentioned in Table B.5.

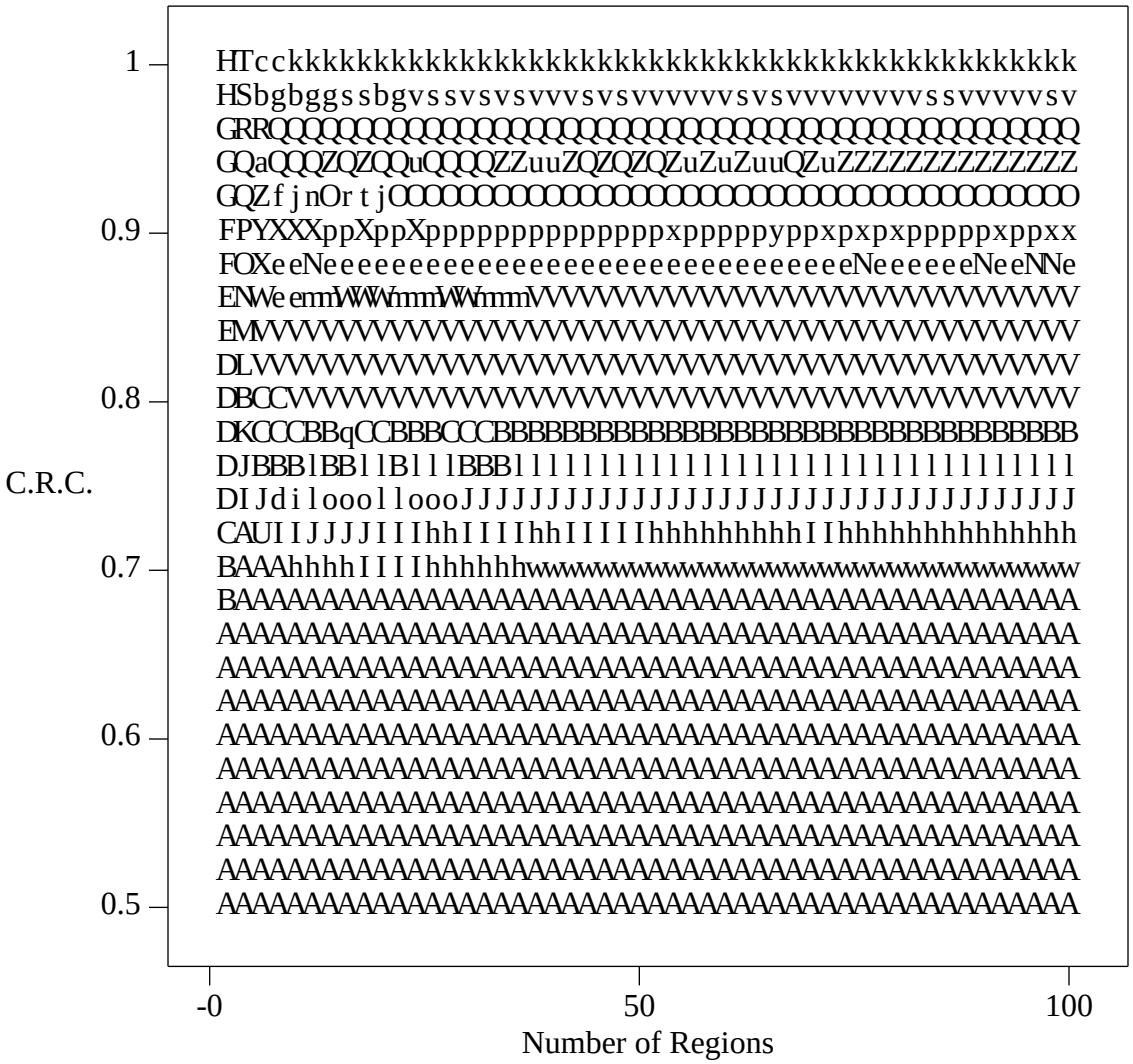


Figure B.3. Cluster plot obtained from learning decomposition heuristics of the VC problems in solving an optimal solution, where the learning is performed with the MR strategy and a random seed 94035.

Listing of Cluster Sets

Cluster Set A: ((1 2 3 4 5 6 7 8 9 10 11 12))
Cluster Set B: ((1 2 3 6 12) (4 5 7 8 9 10 11))
Cluster Set C: ((2 3 6 12) (1 4 5 7 8 9 10 11))
Cluster Set D: ((2 6 12) (1 3 4 5 7 8 9 10 11))
Cluster Set E: ((6 12) (1 2 3) (4 5 7 8 9 10 11))
Cluster Set F: ((12) (2 6) (1 3 4 5 7 8 9 10 11))

Cluster Set G: ((12) (3) (2 6) (1 4 5 7 8 9 10 11))
Cluster Set H: ((1) (6) (12) (2) (3) (4 5 7 8 9 10 11))
Cluster Set I: ((9 10 11) (1 2 3 4 5 6 7 8 12))
Cluster Set J: ((8 9 10 11) (1 2 3 4 5 6 7 12))
Cluster Set K: ((1 2 3 4 6 12) (5 7 8 9 10 11))
Cluster Set L: ((6 12) (1 2 3 4 5 7 8 9 10 11))
Cluster Set M: ((12) (1 2 3 6) (4 5 7 8 9 10 11))
Cluster Set N: ((12) (2 3 6) (1 4 5 7 8 9 10 11))
Cluster Set O: ((12) (3 6) (8 9 10 11) (1 2 4 5 7))
Cluster Set P: ((6) (12) (3) (1 2 4) (5 7 8 9 10 11))
Cluster Set Q: ((6) (12) (3) (1 2) (4 5 7 8 9 10 11))
Cluster Set R: ((6) (12) (2) (3) (1 4 5 7 8 9 10 11))
Cluster Set S: ((6) (12) (2) (3) (1 4 7 8) (5 9 10 11))
Cluster Set T: ((1) (6) (12) (2) (3) (4) (5) (7) (8) (9 10 11))
Cluster Set U: ((11) (1 2 3 4 5 6 7 8 9 10 12))
Cluster Set V: ((3 6 12) (1 2 4 5 7 8 9 10 11))
Cluster Set W: ((2 6) (3 12) (1 4 5 7 8 9 10 11))
Cluster Set X: ((12) (3 6) (9 10 11) (1 2 4 5 7 8))
Cluster Set Y: ((6) (12) (1 2 3 4) (5 7 8 9 10 11))
Cluster Set Z: ((6) (12) (1 2 3) (4 5 7 8 9 10 11))
Cluster Set a: ((6) (12) (2) (1 3) (4 5 7 8 9 10 11))
Cluster Set b: ((6) (12) (2) (3) (1 4 5) (7 8 9 10 11))
Cluster Set c: ((1) (6) (12) (2) (3) (4) (5) (7) (8) (11) (9 10))
Cluster Set d: ((5 8 9 10 11) (1 2 3 4 6 7 12))
Cluster Set e: ((12) (3 6) (1 2 4 5 7 8 9 10 11))
Cluster Set f: ((6) (12) (2 3) (1 4 5 7 8 9 10 11))
Cluster Set g: ((6) (12) (2) (3) (1 4) (5 7 8 9 10 11))
Cluster Set h: ((10 11) (1 2 3 4 5 6 7 8 9 12))
Cluster Set i: ((7 8 9 10 11) (1 2 3 4 5 6 12))
Cluster Set j: ((6) (12) (1 2 3 5) (4 7 8 9 10 11))
Cluster Set k: ((1) (6) (12) (2) (3) (4) (5) (7) (8) (9) (10) (11))
Cluster Set l: ((1 2 3 5 6 12) (4 7 8 9 10 11))
Cluster Set m: ((3 12) (1 2 6) (4 5 7 8 9 10 11))
Cluster Set n: ((12) (1 2) (3 6) (4 5 7 8 9 10 11))
Cluster Set o: ((4 8 9 10 11) (1 2 3 5 6 7 12))
Cluster Set p: ((12) (3 6) (10 11) (1 2 4 5 7 8 9))
Cluster Set q: ((1 3 6 12) (2 4 5 7 8 9 10 11))
Cluster Set r: ((6) (12) (1 2 3 7) (4 5 8 9 10 11))
Cluster Set s: ((6) (12) (2) (3) (1 4 7) (5 8 9 10 11))
Cluster Set t: ((6) (12) (8 9 10 11) (1 2 3 4 5 7))
Cluster Set u: ((6) (12) (1 2) (3 5) (4 7 8 9 10 11))
Cluster Set v: ((6) (12) (2) (3) (1 4 5 7) (8 9 10 11))
Cluster Set w: ((10) (1 2 3 4 5 6 7 8 9 11 12))
Cluster Set x: ((12) (2) (3 6) (1 4 5 7 8 9 10 11))
Cluster Set y: ((12) (11) (3 6) (1 2 4 5 7 8 9 10))

Table B.5. Summary of the most generalized cluster set, the most specialized cluster set, and the chosen cluster set (cluster set I) from learning decomposition heuristics of the VC problems, where the cluster plot is shown in Figure B.3.

The Most Generalized Cluster Set			
Subdomains	Heuristics ID	Probability of Wins	Average Speedup
1,2,3,4,5,6,7,8,9,10,11,12	234	0.621	1.115
The Most Specialized Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
1	817	0.638	1.289
2	234	0.608	1.507
3	231	0.748	1.193
4	639	0.614	1.091
5	234	0.728	1.080
6	500	0.892	1.224
7	778	0.784	1.030
8	364	0.586	1.109
9	234	0.759	1.026
10	234	0.624	1.008
11	1	0.891	1.000
12	778	0.937	1.227
The Chosen Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
9,10,11	1	0.685	1.000
1,2,3,4,5,6,7,8,12	778	0.606	1.158

Table B.6. Summary of the learned heuristics mentioned in Table B.5.

ID	Learned Heuristics	Simplified Heuristics
1	l	l
231	((((f - d) + ((f - d) + l)) + n) * ((((f - d) + l) - dl) + (((f - d) + ((f - d) + l)) - (((f - d) + l) - (f - d)))))	(–3*d–dl+3*f+l)*(–2*d+2*f+l+n) or (4*l–dl)*(3*l+n)
234	((((f - d) + ((f - d) + l)) + n) + ((((f - d) + l) - dl) + (((f - d) + ((f - d) + l)) + (((f - d) + l) - (f - d)))))	–5*d–dl+5*f+4*l+n or 9*l–dl+n
364	((((((f - d) + ((f - d) + l)) + (f - d)) + n) * (((f - d) + ((f - d) + l)) + (f - d))) + f)	f+(–3*d+3*f+l)*(–3*d+3*f+l+n) or f+4*l*(4*l+n)
500	(((((f - d) + ((f - d) + l)) + (f - d)) + n) + ((f - d) + ((f - d) + l)))	–5*d+5*f+2*l+n or 7*l+n
639	(((l * (((f - d) + ((f - d) + l)) + n)) * l) + dl)	$dl+l^2*(-2*d+2*f+l+n)$ or $dl+l^2*(3*l+n)$
778	(((l * (((f - d) + ((f - d) + l)) + n)) * (((f - d) + ((f - d) + l)) + n)) * (((f - d) + ((f - d) + l)) + n))	$l*(-2*d+2*f+l+n)^3$ or $l*(3*l+n)^3$
817	(((l * (((f - d) + ((f - d) + l)) + n)) * (((f - d) + ((f - d) + l)) + n)) * (l * (((f - d) + ((f - d) + l)) + n)))	$l^2*(-2*d+2*f+l+n)^3$ or $l^2*(3*l+n)^3$

B.2.2. Learning selection heuristics used in GDFS

In this section, we show in Figure B.4 the cluster plots of learned selection heuristics used in guided depth-first search to solve an optimal solution of the VC problem, where the decomposition heuristics are those in the chosen set shown in Table B.5. The learning strategy is the MR strategy with a random seed 61801. Following Figure B.4, we list cluster sets shown in the plot. In Table B.7, we summarize the most generalized cluster set and the most specialized cluster set, and the chosen cluster set. In Table B.8, we show the learned selection heuristics mentioned in Table B.7.

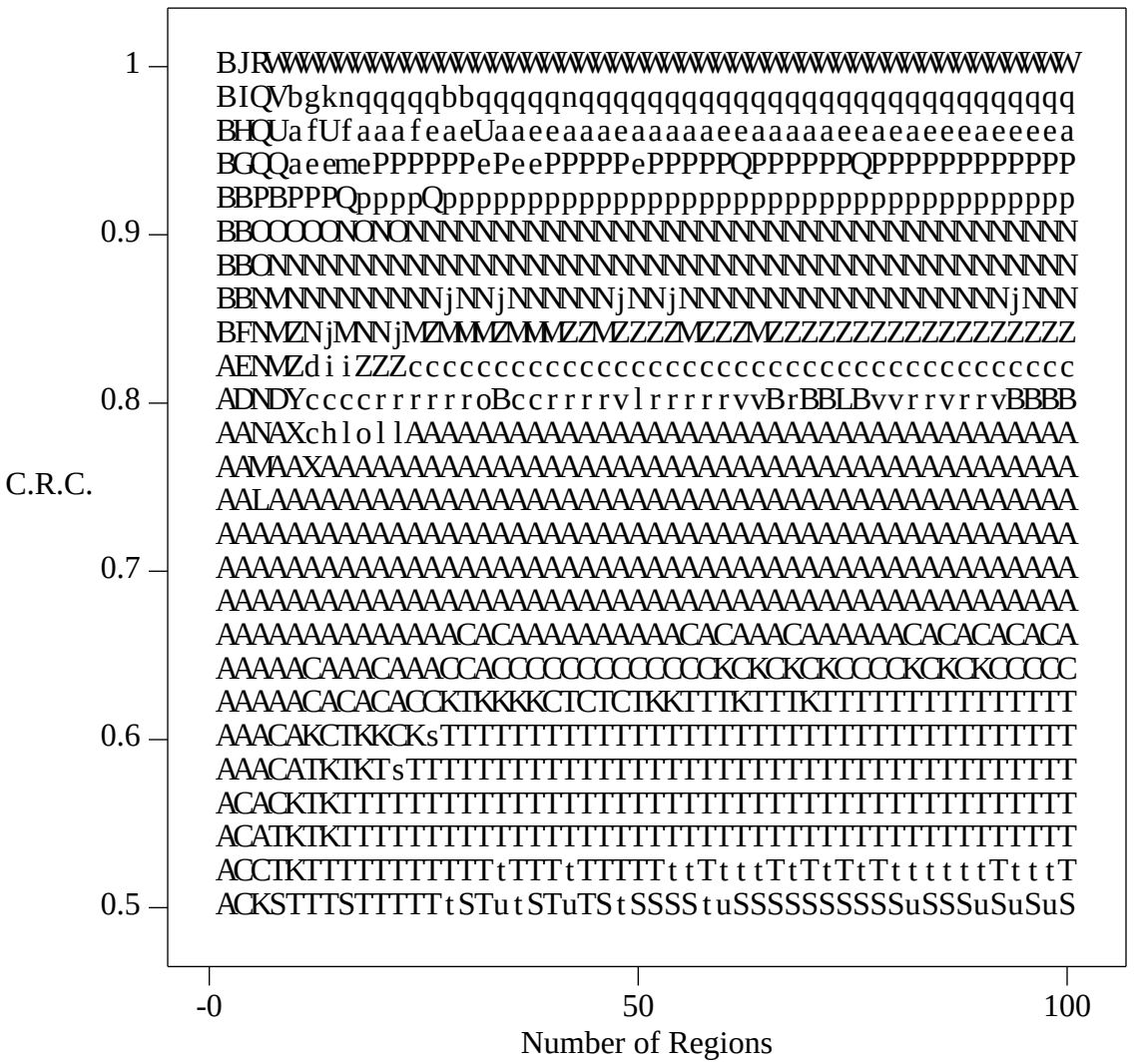


Figure B.4. Cluster plot obtained from learning selection heuristics of the VC problems in solving an optimal solution, where the learning is performed with the MR strategy and a random seed 61801.

Listing of Cluster Sets

Cluster Set A: ((12) (1 2 3 4 5 6 7 8 9 10 11))
Cluster Set B: ((6) (12) (1 2 3 4 5 7 8 9 10 11))
Cluster Set C: ((6 12) (1 2 3 4 5 7 8 9 10 11))
Cluster Set D: ((12) (1 8 10) (2 3 4 5 6 7 9 11))
Cluster Set E: ((12) (1 2 8 10) (3 4 5 6 7 9 11))
Cluster Set F: ((12) (3 5 6 7) (1 2 4 8 9 10 11))

Cluster Set G: ((6) (12) (3) (1 2 4 5 7 8 9 10 11))
Cluster Set H: ((6) (12) (3 7 9 11) (1 2 4 5 8 10))
Cluster Set I: ((6) (12) (3 7) (1 2 8 10) (4 5 9 11))
Cluster Set J: ((1) (6) (12) (2) (3) (4) (5) (7) (8 10) (9 11))
Cluster Set K: ((6 7 12) (1 2 3 4 5 8 9 10 11))
Cluster Set L: ((12) (1 10) (2 3 4 5 6 7 8 9 11))
Cluster Set M: ((12) (3 4 5 6 7) (1 2 8 9 10 11))
Cluster Set N: ((12) (3 6 7) (1 2 4 5 8 9 10 11))
Cluster Set O: ((12) (6 7) (1 2 3 4 5 8 9 10 11))
Cluster Set P: ((6) (12) (1 2 10) (3 4 5 7 8 9 11))
Cluster Set Q: ((6) (12) (3 7) (1 2 4 5 8 9 10 11))
Cluster Set R: ((1) (6) (12) (2) (3) (4) (5) (7) (9) (10) (8 11))
Cluster Set S: ((1 2 8 9 10 11) (3 4 5 6 7 12))
Cluster Set T: ((3 6 7 12) (1 2 4 5 8 9 10 11))
Cluster Set U: ((6) (12) (3 4 5 7) (1 2 8 9 10 11))
Cluster Set V: ((6) (12) (4) (1 2 10) (3 5 7) (8 9 11))
Cluster Set W: ((1) (6) (12) (2) (3) (4) (5) (7) (8) (9) (10) (11))
Cluster Set X: ((12) (10) (1 2 3 4 5 6 7 8 9 11))
Cluster Set Y: ((12) (1 2 10 11) (3 4 5 6 7 8 9))
Cluster Set Z: ((12) (1 2 9 10 11) (3 4 5 6 7 8))
Cluster Set a: ((6) (12) (1 2 9 10 11) (3 4 5 7 8))
Cluster Set b: ((6) (12) (10) (1 2) (3 7) (4 5 8 9 11))
Cluster Set c: ((12) (1 2 9 10) (3 4 5 6 7 8 11))
Cluster Set d: ((12) (3 5 6 7 11) (1 2 4 8 9 10))
Cluster Set e: ((6) (12) (1 2 9 10) (3 4 5 7 8 11))
Cluster Set f: ((6) (12) (1 2 8 9 10) (3 4 5 7 11))
Cluster Set g: ((6) (12) (3 7) (1 2 10) (4 5 8 9 11))
Cluster Set h: ((12) (2 10) (1 3 4 5 6 7 8 9 11))
Cluster Set i: ((12) (1 2 8 9 10) (3 4 5 6 7 11))
Cluster Set j: ((12) (3 4 6 7) (1 2 5 8 9 10 11))
Cluster Set k: ((6) (12) (1 2) (3 7) (9 10) (4 5 8 11))
Cluster Set l: ((12) (2) (1 3 4 5 6 7 8 9 10 11))
Cluster Set m: ((6) (12) (1 2 9) (3 4 5 7 8 10 11))
Cluster Set n: ((6) (12) (4) (1 2) (3 7) (5 8 9 10 11))
Cluster Set o: ((12) (1 2 9) (3 4 5 6 7 8 10 11))
Cluster Set p: ((6) (12) (1 2) (3 4 5 7 8 9 10 11))
Cluster Set q: ((6) (12) (1 2) (3 7) (4 5 8 9 10 11))
Cluster Set r: ((12) (1 2 10) (3 4 5 6 7 8 9 11))
Cluster Set s: ((3 6 12) (1 2 4 5 7 8 9 10 11))
Cluster Set t: ((3 4 6 7 12) (1 2 5 8 9 10 11))
Cluster Set u: ((1 2 9 10 11) (3 4 5 6 7 8 12))
Cluster Set v: ((12) (1 2) (3 4 5 6 7 8 9 10 11))

Table B.7. Summary of the most generalized cluster set, the most specialized cluster set, and the chosen cluster set (cluster set N) from learning selection heuristics of the VC problems, where the cluster plot is shown in Figure B.4.

The Most Generalized Cluster Set			
Subdomains	Heuristics ID	Probability of Wins	Average Speedup
1,2,3,4,5,6,7,8,9,10,11,12	852	0.631	5.556
The Most Specialized Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
1	341	0.563	9.992
2	665	0.551	15.633
3	678	0.757	8.619
4	2	0.681	7.986
5	2	0.693	4.270
6	637	0.836	5.982
7	678	0.766	4.350
8	678	0.687	2.277
9	671	0.626	3.073
10	2	0.642	1.908
11	2	0.665	2.236
12	852	0.872	5.732
The Chosen Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
12	852	0.872	5.732
3,6,7	678	0.688	5.672
1,2,4,5,8,9,10,11	665	0.606	5.754

Table B.8. Summary of the learned heuristics mentioned in Table B.7.

ID	Learned Heuristics	Simplified Heuristics
1	l	l
2	(u / (_zerop (g)))	$\frac{u}{g}$
341	((((u - g) + ((LC * u) - g)) + ((LC * u) - l)) * (u + h))	(h+u)*(−2*g−l+u+2*LC*u)
637	((((((LC * u) + ((LC * u) - g)) - dl) - h2) - ((u + dl) + h2)) + ((((LC * u) + ((LC * u) - g)) - dl) - (u + dl)))	2*(−2*dl−g−h2−u+2*LC*u)
665	((((u - g) + h2) * (((u - g) + ((LC * u) - g)) + (((LC * u) - g) * h))) + u)	u+(−g+h2+u)* (−2*g−g*h+u+LC*u+h*LC*u)
671	(((u - g) + h2) * ((LC * u) + l))	(−g+h2+u)*(l+LC*u)
678	((((u - g) + h2) * (((u - g) + ((LC * u) - g)) + (((LC * u) - g) * h))) / (_zerop (((u - g) + ((LC * u) - g)))))	$\left(\frac{(-g+h2+u)}{-2*g+u+LC*u}\right)$ * (−2*g−g*h+u+LC*u+h*LC*u)
852	(((g + ((((u - g) + ((LC * u) - g)) * u) + (((u - g) + h2) * ((LC * u) + l)))) - ((u - g) + ((LC * u) - g))) / (_zerop ((h2 + h4))))	(3*g−u−LC*u+(−g+h2+u)*(l+LC*u)+ u*(−2*g+u+LC*u))/(h2+h4)

B.3. Experimental Results of Learning Search Heuristics for the PP Problems

In this section, we show in Figure B.5 the cluster plots of learned selection heuristics used in a guided depth-first search to solve an optimal solution of the PP problem. The learning strategy is the MR strategy with a random seed 61801. Following Figure B.5, we list cluster sets shown in the plot. In Table B.9, we summarize the most generalized cluster set, the most specialized cluster set, and the chosen cluster set. In Table B.10, we show the learned selection heuristics mentioned in Table B.9.

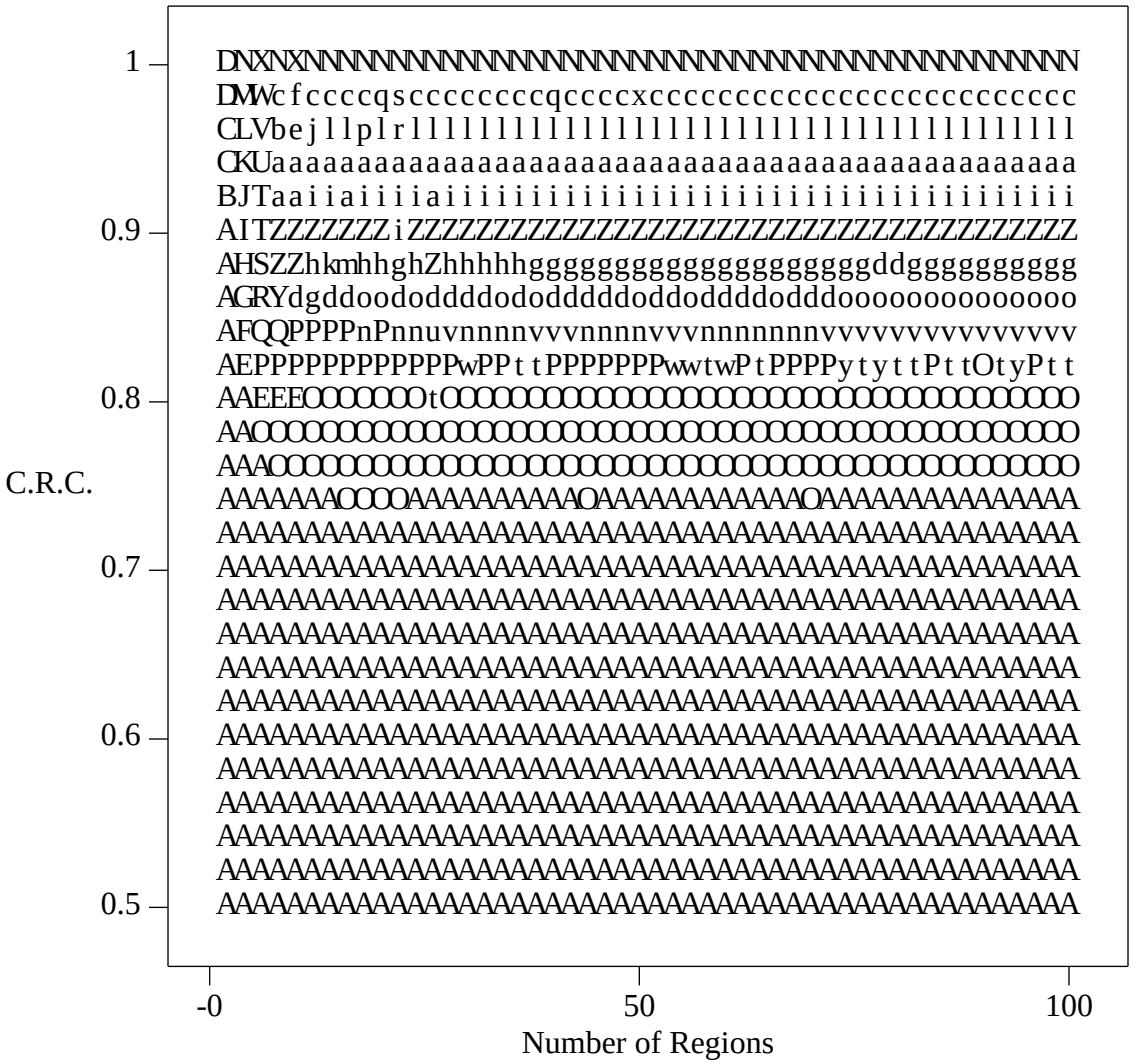


Figure B.5. Cluster plot obtained from learning selection heuristics used in guided depth-first search to solve an optimal solution of the PP problem, where the learning is performed with the MR strategy and a random seed 61801.

Listing of Cluster Sets

Cluster Set A: ((10) (1 2 3 4 5 6 7 8 9 11 12))
Cluster Set B: ((3) (10) (1 2 4 5 6 7 8 9 11 12))
Cluster Set C: ((10) (9 12) (1 2 3 4 5 6 7 8 11))
Cluster Set D: ((9) (3) (10) (12) (1 2 4 5 6 7 8 11))
Cluster Set E: ((10) (6 11) (1 2 3 4 5 7 8 9 12))
Cluster Set F: ((10) (3 6 11) (1 2 4 5 7 8 9 12))

Cluster Set G: ((10) (1 3 6 8 11) (2 4 5 7 9 12))
Cluster Set H: ((10) (7 9 12) (1 2 3 4 5 6 8 11))
Cluster Set I: ((7) (10) (9 12) (1 2 3 4 5 6 8 11))
Cluster Set J: ((10) (9 12) (4 5 7) (1 2 3 6 8 11))
Cluster Set K: ((10) (6 11) (9 12) (2 4) (1 3 5 7 8))
Cluster Set L: ((9) (10) (12) (6 11) (5 7) (1 2 3 4 8))
Cluster Set M: ((6) (9) (3) (4) (5) (7) (10) (11) (12) (1 2 8))
Cluster Set N: ((1) (6) (9) (2) (3) (4) (5) (7) (8) (10) (11) (12))
Cluster Set O: ((10) (11) (1 2 3 4 5 6 7 8 9 12))
Cluster Set P: ((10) (1 6 8 11) (2 3 4 5 7 9 12))
Cluster Set Q: ((10) (1 4 6 8 11) (2 3 5 7 9 12))
Cluster Set R: ((10) (2 5 7 9 12) (1 3 4 6 8 11))
Cluster Set S: ((10) (2 9 12) (5 7 8) (1 3 4 6 11))
Cluster Set T: ((10) (9 12) (3 6 11) (1 2 4 5 7 8))
Cluster Set U: ((3) (10) (6 11) (9 12) (1 2 4 5 7 8))
Cluster Set V: ((10) (11) (3 6) (9 12) (5 7) (1 2 4 8))
Cluster Set W: ((6) (9) (2) (3) (10) (11) (12) (5 7) (1 4 8))
Cluster Set X: ((1) (6) (9) (2) (3) (4) (8) (10) (11) (12) (5 7))
Cluster Set Y: ((5) (10) (2 7 9 12) (1 3 4 6 8 11))
Cluster Set Z: ((10) (3 6 11) (2 9 12) (1 4 5 7 8))
Cluster Set a: ((10) (6 11) (9 12) (1 2 3 4 5 7 8))
Cluster Set b: ((10) (11) (9 12) (4 8) (1 3 6) (2 5 7))
Cluster Set c: ((9) (2) (10) (11) (12) (3 6) (5 7) (1 4 8))
Cluster Set d: ((10) (1 8) (3 4 6 11) (2 5 7 9 12))
Cluster Set e: ((10) (11) (3 6) (9 12) (1 2 4 5 7 8))
Cluster Set f: ((6) (9) (3) (10) (11) (12) (1 8) (2 4) (5 7))
Cluster Set g: ((10) (1 4 8) (3 6 11) (2 5 7 9 12))
Cluster Set h: ((10) (3 6 11) (1 4 5 8) (2 7 9 12))
Cluster Set i: ((10) (6 11) (2 9 12) (1 3 4 5 7 8))
Cluster Set j: ((10) (11) (3 6) (9 12) (1 5 7) (2 4 8))
Cluster Set k: ((10) (1 5 8) (3 4 6 11) (2 7 9 12))
Cluster Set l: ((7) (10) (11) (9 12) (3 4 6) (1 2 5 8))
Cluster Set m: ((10) (2 9 12) (1 5 7 8) (3 4 6 11))
Cluster Set n: ((8) (10) (5 7 9 12) (1 2 3 4 6 11))
Cluster Set o: ((10) (1 8) (5 7 9 12) (2 3 4 6 11))
Cluster Set p: ((10) (11) (9 12) (5 7) (1 2 8) (3 4 6))
Cluster Set q: ((9) (10) (11) (3 6) (2 12) (5 7) (1 4 8))
Cluster Set r: ((7) (10) (11) (3 6) (9 12) (1 2 4 5 8))
Cluster Set s: ((6) (9) (2) (5) (10) (11) (12) (3 7) (1 4 8))
Cluster Set t: ((10) (8 11) (1 2 3 4 5 6 7 9 12))
Cluster Set u: ((10) (1 8) (5 9 12) (2 3 4 6 7 11))
Cluster Set v: ((8) (10) (1 6 11) (2 3 4 5 7 9 12))
Cluster Set w: ((10) (6 8 11) (1 2 3 4 5 7 9 12))
Cluster Set x: ((9) (10) (11) (12) (1 8) (3 6) (2 4) (5 7))
Cluster Set y: ((10) (1 8 11) (2 3 4 5 6 7 9 12))

Table B.9. Summary of the most generalized cluster set, the most specialized cluster set, and the chosen cluster set (cluster set O') from learning selection heuristics of the PP problems, where the cluster plot is shown in Figure B.5. (Cluster Set O' is modified from cluster set O by grouping subdomains 8 and 10 because both conclude the baseline heuristic as the best.)

The Most Generalized Cluster Set			
Subdomains	Heuristics ID	Probability of Wins	Average Speedup
1,2,3,4,5,6,7,8,9,10,11,12	622	0.721	1.135
The Most Specialized Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
1	315	0.590	1.187
2	622	0.859	1.042
3	243	0.765	1.454
4	309	0.854	1.052
5	243	0.638	1.371
6	552	0.752	1.317
7	324	0.662	1.445
8	1	0.935	1.000
9	620	0.838	1.036
10	1	0.999	1.000
11	564	0.756	1.431
12	622	0.840	1.064
The Chosen Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
8,10	1	0.967	1.000
11	564	0.756	1.431
1,2,3,4,5,6,7,9,12	622	0.695	1.193

Table B.10. Summary of the learned heuristics mentioned in Table B.9.

ID	Learned Heuristics	Simplified Heuristics
1	l	l
243	(dl + h)	dl+h
309	((dl + h) - du)	dl+h−du
315	((dl + h) + h)	dl+2*h
324	((dl + h) * (((((u - dh) * (u + h)) + ((u - dh) * u)) + (((u - dh) * (u + h)) * u)) * (dl + ((u - dh) * u))))	$(dl+h)*(-dh+u)*(dl-dh*u+u^2)*(h+2*u+h*u+u^2)$
552	((((((dl + h) + h) + g) + (((dl + h) + h) + dg)) + (((dl + h) + h) + dg)) / (_zerop (l)))	$\frac{2*dg+3*dl+g+6*h}{l}$
564	(dl + (((dl + h) * ((((u - dh) * (u + h)) + ((u - dh) * u)) + dh)) * ((dl + h) - du)))	$dl+(dl+h)*(dl-du+h)*(dh-dh*h-2*dh*u+h*u+2*u^2)$
620	(l + (((((dl + h) + h) + g) + (((dl + h) + h) + dg)) + (((dl + h) + h) + dg)))	2*dg+3*dl+g+6*h+l
622	((((((dl + h) + h) + g) + (((dl + h) + h) + dg)) + (((dl + h) + h) + dg)) * u)	(2*dg+3*dl+g+6*h)*u

B.4. Experimental Results of Learning Search Heuristics for the TSPs

In this section, we will show the cluster plots, cluster sets, learned decomposition, selection, and pruning heuristics used in a guided depth-first search and band search for solving the TSP.

B.4.1. Learning decomposition heuristics used in GDFS

In this section, we show in Figure B.6 the cluster plots of learned decomposition heuristics used in a guided depth-first search to solve an optimal solution of the TSP. The learning strategy is the RR strategy with a random seed 61801. Following Figure B.6, we list cluster sets shown in the plot. In Table B.11, we summarize the most generalized cluster set, the most specialized cluster set, and the chosen cluster set. In Table B.12, we show the learned decomposition heuristics mentioned in Table B.11.

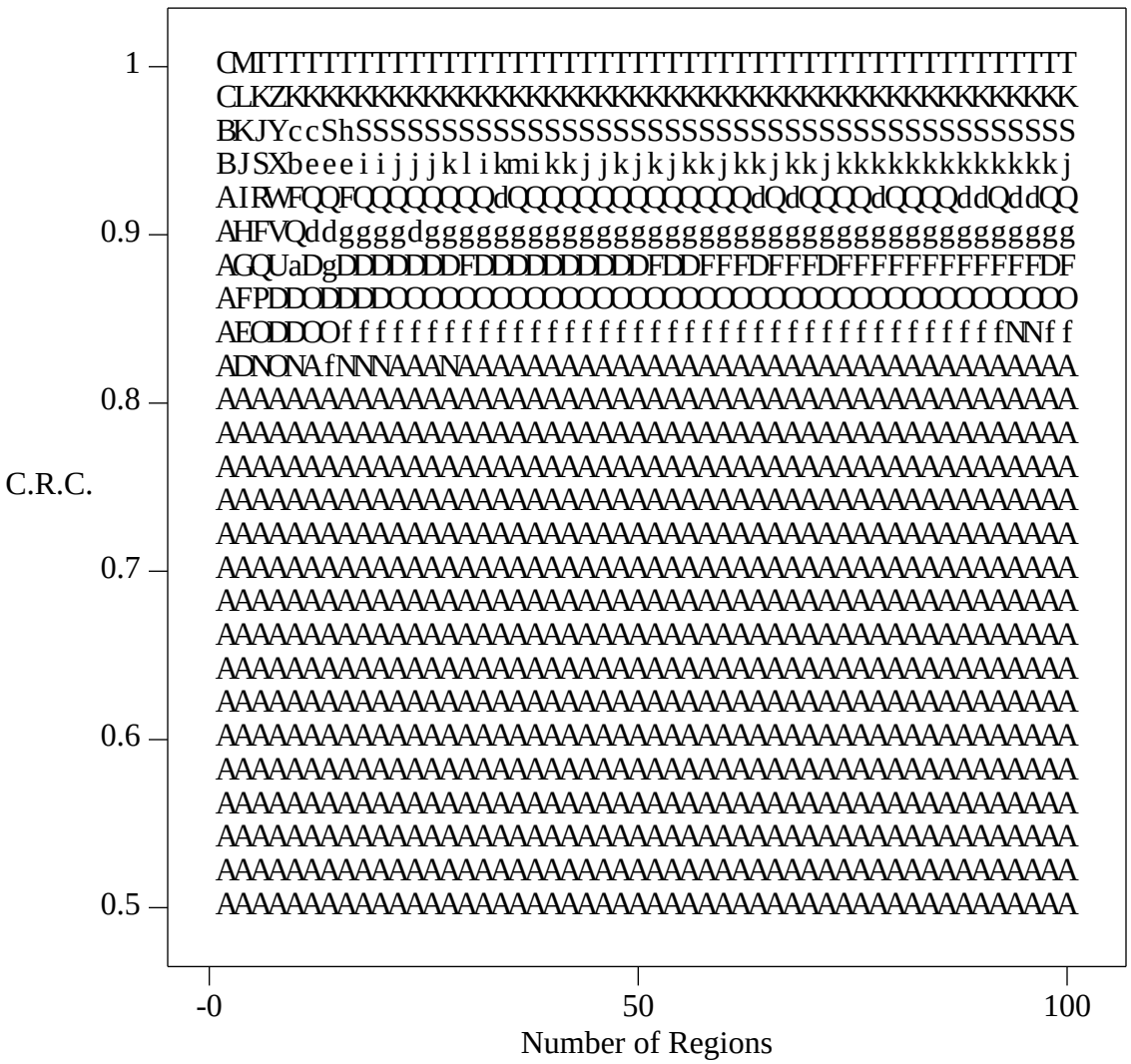


Figure B.6. Cluster plot obtained from learning decomposition heuristics used in guided depth-first search to solve an optimal solution of the TSP, where the learning is performed with the RR strategy and a random seed 61801.

Listing of Cluster Sets

Cluster Set A: ((11) (1 2 3 4 5 6 7 8 9 10 12))
Cluster Set B: ((5) (11) (1 2 3 4 6 7 8 9 10 12))
Cluster Set C: ((4) (5) (11) (1 2 3 6 7 8 9 10 12))
Cluster Set D: ((11) (2 8 10) (1 3 4 5 6 7 9 12))
Cluster Set E: ((11) (2 5 8 10 12) (1 3 4 6 7 9))
Cluster Set F: ((11) (1 4 7) (2 3 5 6 8 9 10 12))
Cluster Set G: ((5) (11) (1 4 7) (2 3 6 8 9 10 12))
Cluster Set H: ((1) (11) (5 7) (3 4 6) (2 8 9 10 12))
Cluster Set I: ((1) (11) (4 7) (5 8) (3 6 12) (2 9 10))
Cluster Set J: ((1) (5) (9) (11) (4 7) (2 3 6 8 10 12))
Cluster Set K: ((1) (4) (5) (7) (9) (11) (3 6 12) (2 8 10))
Cluster Set L: ((1) (4) (5) (7) (9) (11) (3 6) (2 8 10 12))
Cluster Set M: ((1) (6) (12) (3) (4) (5) (7) (8) (9) (11) (2 10))
Cluster Set N: ((7) (11) (1 2 3 4 5 6 8 9 10 12))
Cluster Set O: ((11) (2 8) (1 3 4 5 6 7 9 10 12))
Cluster Set P: ((11) (2 3 8 10) (1 4 5 6 7 9 12))
Cluster Set Q: ((11) (1 4 7 9) (2 3 5 6 8 10 12))
Cluster Set R: ((5) (11) (1 7) (4 6 12) (2 3 8 9 10))
Cluster Set S: ((11) (1 9) (5 6) (4 7) (2 3 8 10 12))
Cluster Set T: ((1) (6) (12) (2) (3) (4) (5) (7) (8) (9) (10) (11))
Cluster Set U: ((11) (2 5 6 8 10) (1 3 4 7 9 12))
Cluster Set V: ((11) (1 3 4 7 9) (2 5 6 8 10 12))
Cluster Set W: ((11) (1 7 9) (4 6 12) (2 3 5 8 10))
Cluster Set X: ((11) (1 9) (4 7) (5 6 12) (2 3 8 10))
Cluster Set Y: ((3) (4) (5) (7) (11) (1 9) (2 6 8 10 12))
Cluster Set Z: ((1) (3) (4) (5) (7) (9) (11) (6 12) (2 8 10))
Cluster Set a: ((11) (2 5 8 10) (1 3 4 6 7 9 12))
Cluster Set b: ((11) (1 9) (4 7) (5 6 8) (2 3 10 12))
Cluster Set c: ((5) (11) (1 9) (4 7) (2 3 6 8 10 12))
Cluster Set d: ((11) (1 4 7 9 12) (2 3 5 6 8 10))
Cluster Set e: ((11) (1 9) (4 7) (2 3 10) (5 6 8 12))
Cluster Set f: ((11) (1 7) (2 3 4 5 6 8 9 10 12))
Cluster Set g: ((11) (2 3 5 8 10) (1 4 6 7 9 12))
Cluster Set h: ((4) (7) (11) (1 9) (5 6) (2 3 8 10 12))
Cluster Set i: ((11) (1 7) (4 6 9) (5 8 12) (2 3 10))
Cluster Set j: ((5) (11) (1 7) (4 6 9) (2 3 8 10 12))
Cluster Set k: ((5) (11) (1 7) (4 6 9 12) (2 3 8 10))
Cluster Set l: ((11) (1 7) (2 3) (5 8 10) (4 6 9 12))
Cluster Set m: ((11) (1 7) (5 8) (2 3 10) (4 6 9 12))

Table B.11. Summary of the most generalized cluster set, the most specialized cluster set, and the chosen cluster set (cluster set A) from learning decomposition heuristics of the TSP, where the cluster plot is shown in Figure B.6.

The Most Generalized Cluster Set			
Subdomains	Heuristics ID	Probability of Wins	Average Speedup
1,2,3,4,5,6,7,8,9,10,11,12	915	0.630	1.060
The Most Specialized Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
1	1231	0.948	1.109
2	949	0.700	1.034
3	915	0.809	1.051
4	957	0.691	1.285
5	1008	0.655	1.217
6	1217	0.752	1.113
7	1217	0.790	1.205
8	1212	0.633	1.141
9	1208	0.911	1.053
10	1015	0.706	1.118
11	1219	0.793	1.734
12	1191	0.723	1.093
The Chosen Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
11	1219	0.793	1.734
1,2,3,4,5,6,7,8,9,10,12	915	0.634	1.067

Table B.12. Summary of the learned heuristics mentioned in Table B.11.

ID	Learned Heuristics	Simplified Heuristics
1	c	c
915	((((c * (m + l)) + d) / (_zerop (c)) / (_zerop ((m + l)))) + ((c * (m + l)) + d))	$\frac{(1+c*l+c*m)*(d+c*l+c*m)}{c*l+c*m}$
949	(((c * (m + l)) + d) * (((c * (m + l)) * (m + l)) - ((c * (m + l)) + d)))	$(d+c*l+c*m)*(-d-c*(l+m)+c*(l+m)^2)$
957	((((c * (m + l)) + d) / (_zerop (c)) / (_zerop ((m + l)))) * ((((c * (m + l)) + d) / (_zerop (c)) / (_zerop ((m + l)))) * ((c * (m + l)) * (m + l))))	$\frac{(d+c*l+c*m)^2}{c}$
1008	(((a + (LC * d)) - d) - (((c * (m + l)) + d) / (_zerop (c)) / (_zerop ((m + l)))))	$a-d+d*LC-\frac{d+c*l+c*m}{c*(l+m)}$
1015	(((a + (LC * d)) - d) + (a * d))	a−d+a*d+d*LC
1191	((l - (((((c * (m + l)) + d) / (_zerop (c)) / (_zerop ((m + l)))) + d) + a)) + ((a * (((((c * (m + l)) + d) / (_zerop (c)) / (_zerop ((m + l)))) + d) + d)) + c))	$-a+c-d+l+2*a*d-(a-1)*\frac{d+c*l+c*m}{c*(l+m)}$
1208	(c - (l - (((((c * (m + l)) + d) / (_zerop (c)) / (_zerop ((m + l)))) + d) + a)))	$a+c+d-l+\frac{d+c*l+c*m}{c*l+c*m}$
1212	(m + ((a * (((((c * (m + l)) + d) / (_zerop (c)) / (_zerop ((m + l)))) + d) + d)) + c))	$c+m+a*\left(2*d+\frac{d+c*l+c*m}{c*l+c*m}\right)$
1217	(c + ((a * (((((c * (m + l)) + d) / (_zerop (c)) / (_zerop ((m + l)))) + d) + d)) + c))	$2*c+a*\left(2*d+\frac{d+c*l+c*m}{c*l+c*m}\right)$
1219	(a / (_zerop ((c - a))))	$\frac{a}{c-a}$
1231	((c + (((((c * (m + l)) + d) / (_zerop (c)) / (_zerop ((m + l)))) - l) - l)) - ((l - (((((c * (m + l)) + d) / (_zerop (c)) / (_zerop ((m + l)))) + d) + a)) + l))	$a+c+d-4*l+\frac{2*(d+c*l+c*m)}{c*l+c*m}$

B.4.2. Learning selection heuristics used in band search

In this section, we show in Figure B.7 the cluster plots of learned selection heuristics used in a dynamic band search to solve an optimal solution of the TSP, where the decomposition heuristics are those in the chosen set shown in Table B.11. The learning strategy is the RR strategy with a random seed 61801. Following Figure B.7, we list cluster sets shown in the plot. In Table B.13, we summarize the most generalized cluster set, the most specialized cluster set, and the chosen cluster set. In Table B.14, we show the learned selection heuristics mentioned in Table B.13.

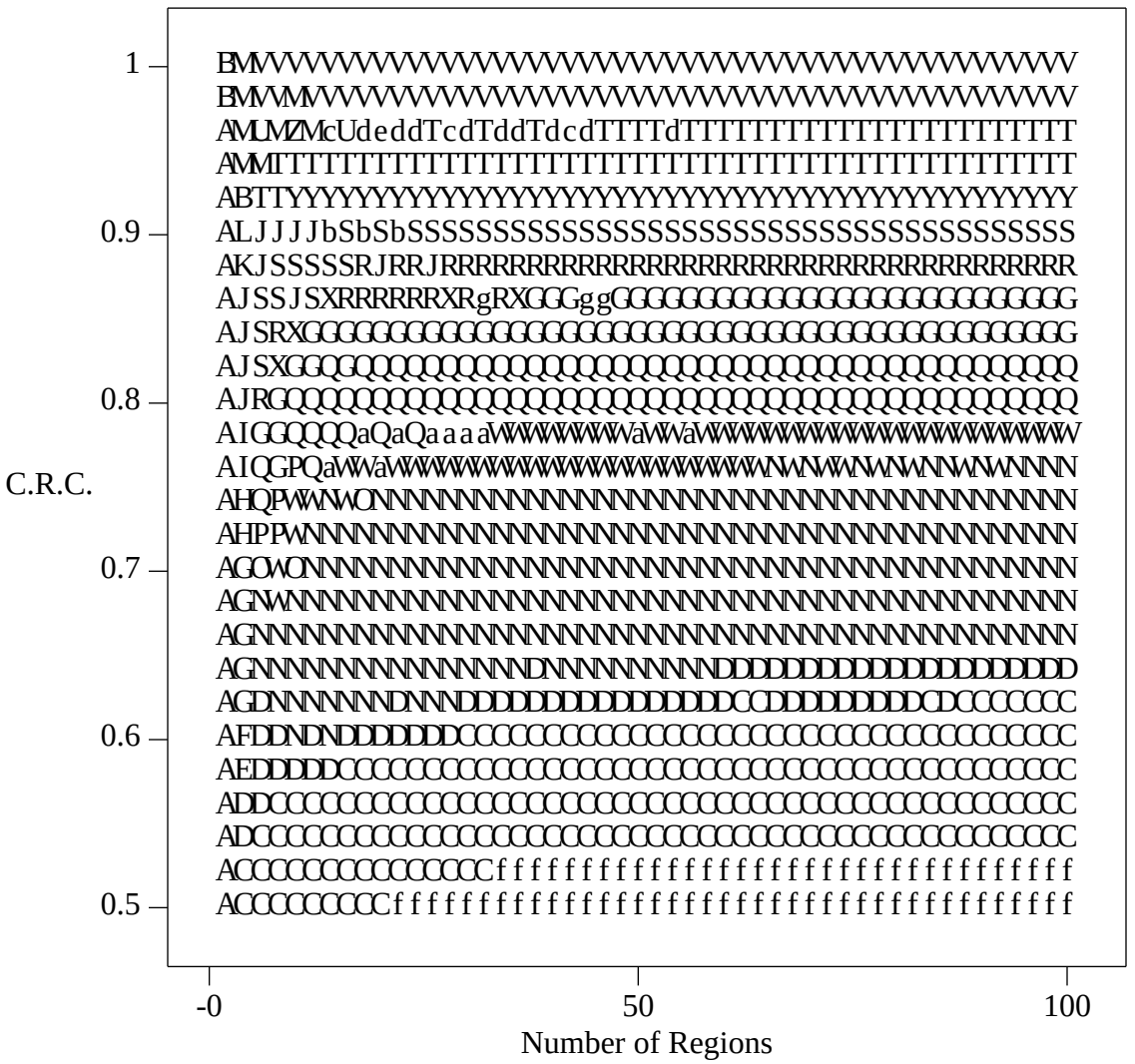


Figure B.7. Cluster plot obtained from learning selection heuristics used in dynamic band search to solve an optimal solution of the TSP, where the learning is performed with the RR strategy and a random seed 61801.

Listing of Cluster Sets

Cluster Set A: ((7) (12) (4) (8) (11) (2 3 6 10))
Cluster Set B: ((7) (12) (3) (4) (8) (11) (2 6 10))
Cluster Set C: ((7 8 11) (2 3 4 6 10 12))
Cluster Set D: ((4 7 8 11) (2 3 6 10 12))
Cluster Set E: ((7) (4 8 11) (2 3 6 10 12))
Cluster Set F: ((8) (2 3 6 10) (4 7 11 12))

Cluster Set G: ((2 6 10) (3 7 12) (4 8 11))
Cluster Set H: ((7 12) (4 8 11) (2 3 6 10))
Cluster Set I: ((7) (4 12) (8 11) (2 3 6 10))
Cluster Set J: ((7) (8 11) (2 6 10) (3 4 12))
Cluster Set K: ((7) (3) (4 12) (8 11) (2 6 10))
Cluster Set L: ((7) (12) (3) (8) (4 11) (2 6 10))
Cluster Set M: ((7) (12) (3) (4) (6) (8) (11) (2 10))
Cluster Set N: ((2 3 6 10) (4 7 8 11 12))
Cluster Set O: ((11) (2 3 6 10) (4 7 8 12))
Cluster Set P: ((8 11) (4 7 12) (2 3 6 10))
Cluster Set Q: ((8 11) (2 6 10) (3 4 7 12))
Cluster Set R: ((7) (3) (2 6 10) (4 8 11 12))
Cluster Set S: ((7) (3 12) (2 6 10) (4 8 11))
Cluster Set T: ((7) (2 10) (4 12) (3 6) (8 11))
Cluster Set U: ((2) (7) (12) (3) (4) (8) (11) (6 10))
Cluster Set V: ((2) (7) (12) (3) (4) (6) (8) (10) (11))
Cluster Set W: ((7) (2 3 6 10) (4 8 11 12))
Cluster Set X: ((3 7) (2 6 10) (4 8 11 12))
Cluster Set Y: ((7) (6) (2 10) (8 11) (3 4 12))
Cluster Set Z: ((7) (12) (4) (8) (11) (2 10) (3 6))
Cluster Set a: ((7) (2 6 10) (3 4 8 11 12))
Cluster Set b: ((7) (12) (2 10) (3 6) (4 8 11))
Cluster Set c: ((7) (8) (2 10) (11) (4 12) (3 6))
Cluster Set d: ((7) (8) (11) (2 10) (4 12) (3 6))
Cluster Set e: ((7) (3) (6) (8) (11) (2 10) (4 12))
Cluster Set f: ((8 11) (2 3 4 6 7 10 12))
Cluster Set g: ((3) (7 12) (2 6 10) (4 8 11))

Table B.13. Summary of the most generalized cluster set, the most specialized cluster set, and the chosen cluster set (cluster set Q') from learning selection heuristics of the TSP, where the cluster plot is shown in Figure B.7. (Cluster Set Q' is obtained by grouping all clusters into a single cluster because all these clusters conclude that the baseline heuristic is the best.)

The Most Generalized Cluster Set			
Subdomains	Heuristics ID	Probability of Wins	Average Speedup
2,3,4,6,7,8,10,11,12	1	0.904	1.000
The Most Specialized Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
2	265	0.589	1.014
3	1	0.977	1.000
4	1	0.989	1.000
6	1	0.982	1.000
7	1	0.941	1.000
8	1	0.989	1.000
10	1	0.929	1.000
11	1	0.972	1.000
12	1	0.958	1.000
The Chosen Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
2,3,4,6,7,8,10,11,12	1	0.904	1.000

Table B.14. Summary of the learned heuristics mentioned in Table B.13.

ID	Learned Heuristics	Simplified Heuristics
1	l	l
265	((((d + (((- h3) * (- l)) * ((- h3) / (_zerop ((- d)))))) + (((- h3) * (- l)) * ((- h3) / (_zerop ((- d)))))) / (_zerop (d))) + (dg + ((- h3) / (_zerop (d)))))	$1+dg-\frac{h3}{d}+2*l*\left(\frac{h3}{d}\right)^2$

B.4.3. Learning selection heuristics used in band search to find a solution

In this section, we show in Figure B.8 the cluster plots of learned selection heuristics used in a dynamic band search to solve a feasible solution of the TSP, where the decomposition heuristics are those in the chosen set shown in Table B.11. The learning

strategy is the RR strategy with a random seed 61801. Following Figure B.8, we list cluster sets shown in the plot. In Table B.15, we summarize the most generalized cluster set, the most specialized cluster set, and the chosen cluster set. In Table B.16, we show the learned selection heuristics mentioned in Table B.15.

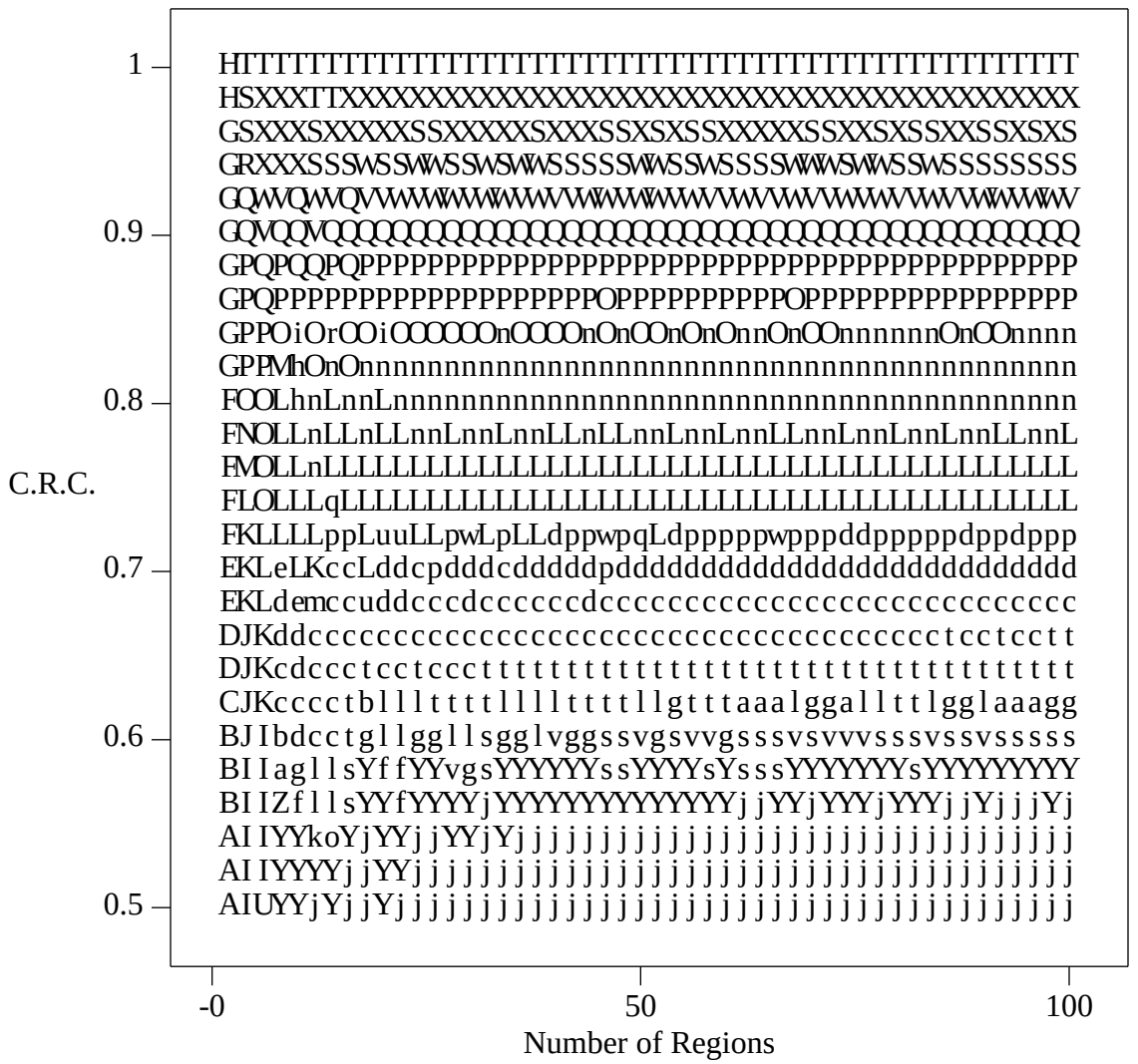


Figure B.8. Cluster plot obtained from learning selection heuristics used in dynamic band search to solve a feasible solution of the TSP, where the learning is performed with the RR strategy and a random seed 61801.

Listing of Cluster Sets

Cluster Set A: ((7) (12) (11) (2 3 4 6 8 10))
Cluster Set B: ((7) (12) (10) (11) (2 3 4 6 8))
Cluster Set C: ((7) (12) (11) (6 10) (2 3 4 8))
Cluster Set D: ((7) (12) (11) (6 8) (2 3 4 10))
Cluster Set E: ((7) (12) (3) (11) (6 10) (2 4 8))
Cluster Set F: ((7) (12) (8) (11) (6 10) (2 3 4))
Cluster Set G: ((7) (12) (6) (8) (10) (11) (2 3 4))
Cluster Set H: ((7) (12) (3) (6) (8) (10) (11) (2 4))
Cluster Set I: ((12) (7 11) (2 3 4 6 8 10))
Cluster Set J: ((12) (6) (7 11) (2 3 4 8 10))
Cluster Set K: ((12) (7 11) (6 8) (2 3 4 10))
Cluster Set L: ((12) (7 11) (2 3 4) (6 8 10))
Cluster Set M: ((7) (12) (6 8) (10 11) (2 3 4))
Cluster Set N: ((7) (12) (11) (2 3 4) (6 8 10))
Cluster Set O: ((7) (12) (11) (3 4) (2 6 8 10))
Cluster Set P: ((7) (12) (8) (11) (3 4) (2 6 10))
Cluster Set Q: ((7) (12) (11) (2 8) (3 4) (6 10))
Cluster Set R: ((7) (12) (6) (8) (11) (2 10) (3 4))
Cluster Set S: ((7) (12) (3) (4) (6) (8) (11) (2 10))
Cluster Set T: ((2) (7) (12) (3) (4) (6) (8) (10) (11))
Cluster Set U: ((11 12) (7 8 10) (2 3 4 6))
Cluster Set V: ((7) (12) (3) (8) (11) (2 4) (6 10))
Cluster Set W: ((7) (12) (3) (4) (11) (2 8) (6 10))
Cluster Set X: ((2) (7) (12) (3) (4) (8) (11) (6 10))
Cluster Set Y: ((2 7) (3 4) (6 8 10 11 12))
Cluster Set Z: ((3 4) (2 7 8) (6 10 11 12))
Cluster Set a: ((2 3 4) (6 7 8) (10 11 12))
Cluster Set b: ((11 12) (2 3 4) (6 7 8 10))
Cluster Set c: ((12) (2 3 4 10) (6 7 8 11))
Cluster Set d: ((12) (2 3 4) (6 7 8 10 11))
Cluster Set e: ((12) (6) (2 3 4) (7 8 10 11))
Cluster Set f: ((3 4) (10 11 12) (2 6 7 8))
Cluster Set g: ((7 8) (10 11 12) (2 3 4 6))
Cluster Set h: ((7) (12) (2 8) (3 4) (6 10 11))
Cluster Set i: ((7) (12) (3 4) (10 11) (2 6 8))
Cluster Set j: ((12) (3 4) (2 6 7 8 10 11))
Cluster Set k: ((3 4) (8 11 12) (2 6 7 10))
Cluster Set l: ((7 8) (11 12) (2 3 4 6 10))
Cluster Set m: ((12) (6) (7 8 11) (2 3 4 10))
Cluster Set n: ((12) (7 11) (3 4) (2 6 8 10))
Cluster Set o: ((11 12) (3 4) (2 6 7 8 10))
Cluster Set p: ((12) (10) (2 3 4) (6 7 8 11))
Cluster Set q: ((12) (8 10) (2 3 4) (6 7 11))
Cluster Set r: ((7) (12) (3 4) (6 11) (2 8 10))

Cluster Set s: ((7) (2 3 4) (6 8 10 11 12))
Cluster Set t: ((11 12) (6 7 8) (2 3 4 10))
Cluster Set u: ((12) (6 8) (2 3 4) (7 10 11))
Cluster Set v: ((7 8) (2 3 4) (6 10 11 12))
Cluster Set w: ((12) (6 10) (2 3 4) (7 8 11))

Table B.15. Summary of the most generalized cluster set, the most specialized cluster set, and the chosen cluster set (cluster set P) from learning selection heuristics of the TSP, where the cluster plot is shown in Figure B.8.

The Most Generalized Cluster Set			
Subdomains	Heuristics ID	Probability of Wins	Average Speedup
2,3,4,6,7,8,10,11,12	581	0.543	4.371
The Most Specialized Cluster Set			
Cluster	Heuristics	ID	Probability
2	301	0.667	12.567
3	279	0.545	1.754
4	476	0.615	3.876
6	301	0.833	4.289
7	597	0.822	13.181
8	511	0.684	8.181
10	301	0.803	8.145
11	204	0.880	2.913
12	501	0.758	8.754
The Chosen Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
7	597	0.822	13.181
8	511	0.684	8.181
11	204	0.880	2.913
12	501	0.758	8.754
3,4	581	0.571	2.763
2,6,10	301	0.768	8.334

Table B.16. Summary of the learned heuristics mentioned in Table B.15.

ID	Learned Heuristics	Simplified Heuristics
1	l	l
204	(((- d) + h2) - h)	$-d+h2-h$
279	((((g - d) + dg) + (((g - d) - ((- d) / (_zerop ((dg + (LC * h)))))) + ((- d) + h2))) - h)	$-3*d+dg+2*g-h+h2+\frac{d}{dg+h*LC}$
301	(g + ((g - d) - ((- d) / (_zerop ((dg + (LC * h)))))))	$-d+2*g+\frac{d}{dg+h*LC}$
476	((h2 + ((g + ((g - d) - ((- d) / (_zerop ((dg + (LCC * h))))))) + g)) - dh)	$-d-dh+3*g+h2+\frac{d}{dg+h*LC}$
501	(((h4 + ((g - d) - ((- d) / (_zerop ((dg + (LCC * h))))))) + (((- d) + h2) - h)) - h3)	$-2*d+g-h+h2-h3+h4+\frac{d}{dg+h*LC}$
511	((d + ((h4 + ((g - d) - ((- d) / (_zerop ((dg + (LC * h))))))) + (((- d) + h2) - h))) - dh)	$-d-dh+g-h+h2+h4+\frac{d}{dg+h*LC}$
581	((1 / (_zerop (d))) + ((d + ((h4 + ((g - d) - ((- d) / (_zerop ((dg + (LC * h))))))) + (((- d) + h2) - h))) + dh))	$\frac{1}{d}-d+dh+g-h+h2+h4+\frac{d}{dg+h*LC}$
597	((((h4 + ((g - d) - ((- d) / (_zerop ((dg + (LC * h))))))) + (((- d) + h2) - h)) - h3) - (1 / (_zerop (d))))	$-\frac{1}{d}-2*d+g-h+h2-h3+h4+\frac{d}{dg+h*LC}$

B.4.4. Learning selection heuristics used in GDFS

In this section, we show in Figure B.9 the cluster plots of learned selection heuristics used in a guided depth-first search to solve an optimal solution of the TSP, where the decomposition heuristics are those in the chosen set shown in Table B.11. The learning strategy is the RR strategy with a random seed 61801. Following Figure B.9, we list cluster sets shown in the plot. In Table B.17, we summarize the most generalized cluster set, the most specialized cluster set, and the chosen cluster set. In Table B.18, we show the learned selection heuristics mentioned in Table B.17.

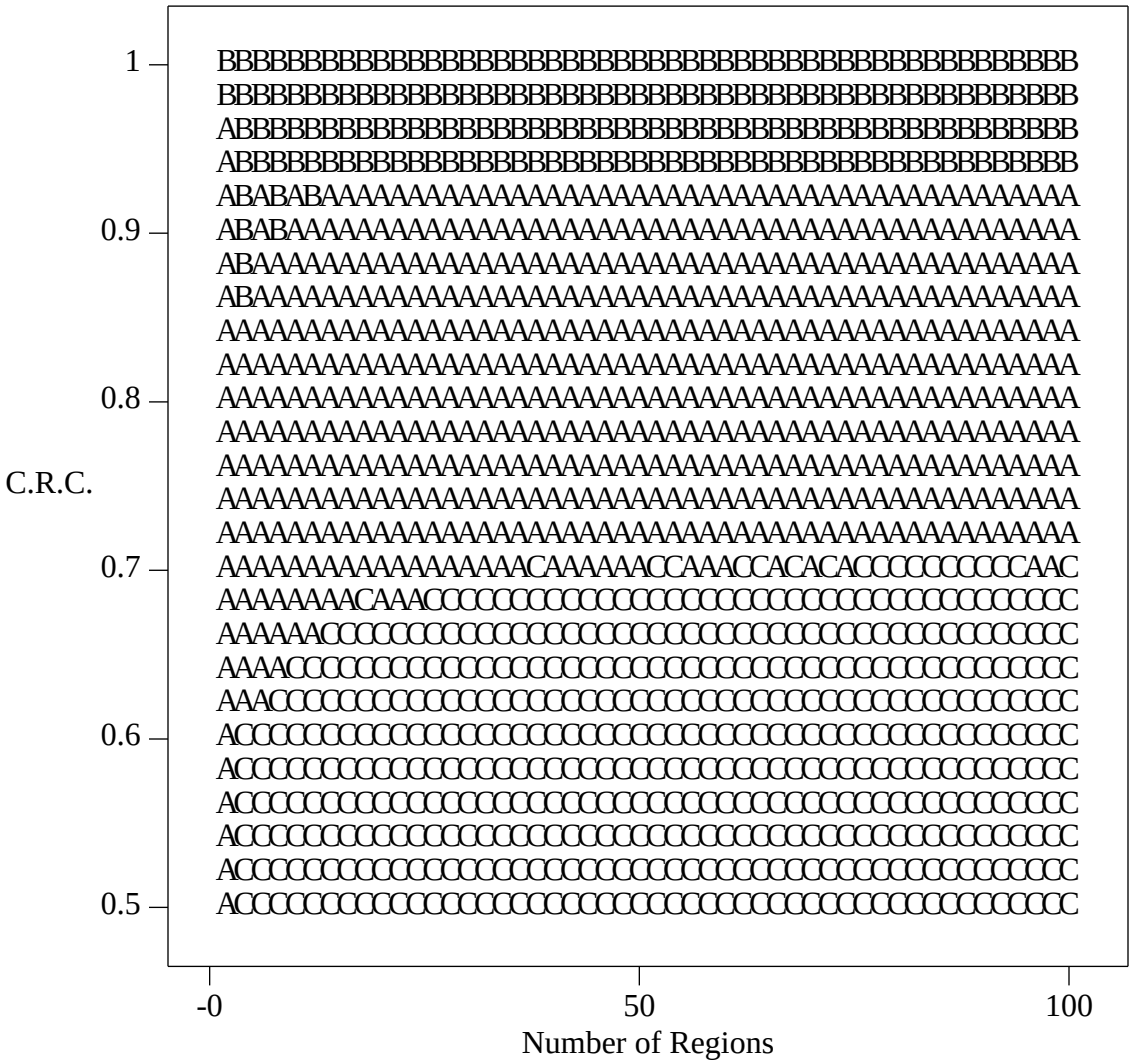


Figure B.9. Cluster plot obtained from learning selection heuristics used in guided depth-first search to solve an optimal solution of the TSP, where the learning is performed with the RR strategy and a random seed 61801.

Listing of Cluster Sets

Cluster Set A: ((5) (1 9))
Cluster Set B: ((1) (5) (9))
Cluster Set C: ((1 5 9))

Table B.17. Summary of the most generalized cluster set, the most specialized cluster set, and the chosen cluster set (cluster set A) from learning selection heuristics of the TSP, where the cluster plot is shown in Figure B.9.

The Most Generalized Cluster Set			
Subdomains	Heuristics ID	Probability of Wins	Average Speedup
1,5,9	471	0.679	1.142
The Most Specialized Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
1	904	0.641	1.201
5	471	0.876	1.145
9	643	0.815	1.162
The Chosen Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
5	471	0.876	1.145
1,9	904	0.691	1.168

Table B.18. Summary of the learned heuristics mentioned in Table B.17.

ID	Learned Heuristics	Simplified Heuristics
1	l	l
471	(((((LC * u) + g) - (u + h1)) * (dg + l)) * (u * l))	l*(dg+l)*u*(g–h1–u+LC*u)
643	((((LC * u) + g) - (u + h1)) * u)	u*(g–h1–u+LC*u)
904	(((((LC * u) + g) - (u + h1)) * u) + (h + ((((((LC * u) + g) - (u + h1)) * (dg + l)) * (u * l)) * l)))	h+u*(g–h1–u+LC*u)+ l^2*(dg+l)*u*(g–h1–u+LC*u)

B.4.5. Learning pruning heuristics used in band search

In this section, we show in Figure B.10 the cluster plots of learned pruning heuristics used in a dynamic band search to solve a feasible solution of the TSP, where the decomposition heuristics are those in the chosen set shown in Table B.11. The learning strategy is the RR strategy with a random seed 61801. Following Figure B.10, we list cluster sets shown in the plot. In Table B.19, we summarize the most generalized cluster set, the

most specialized cluster set, and the chosen cluster set. In Table B.20, we show the learned pruning heuristics mentioned in Table B.19.

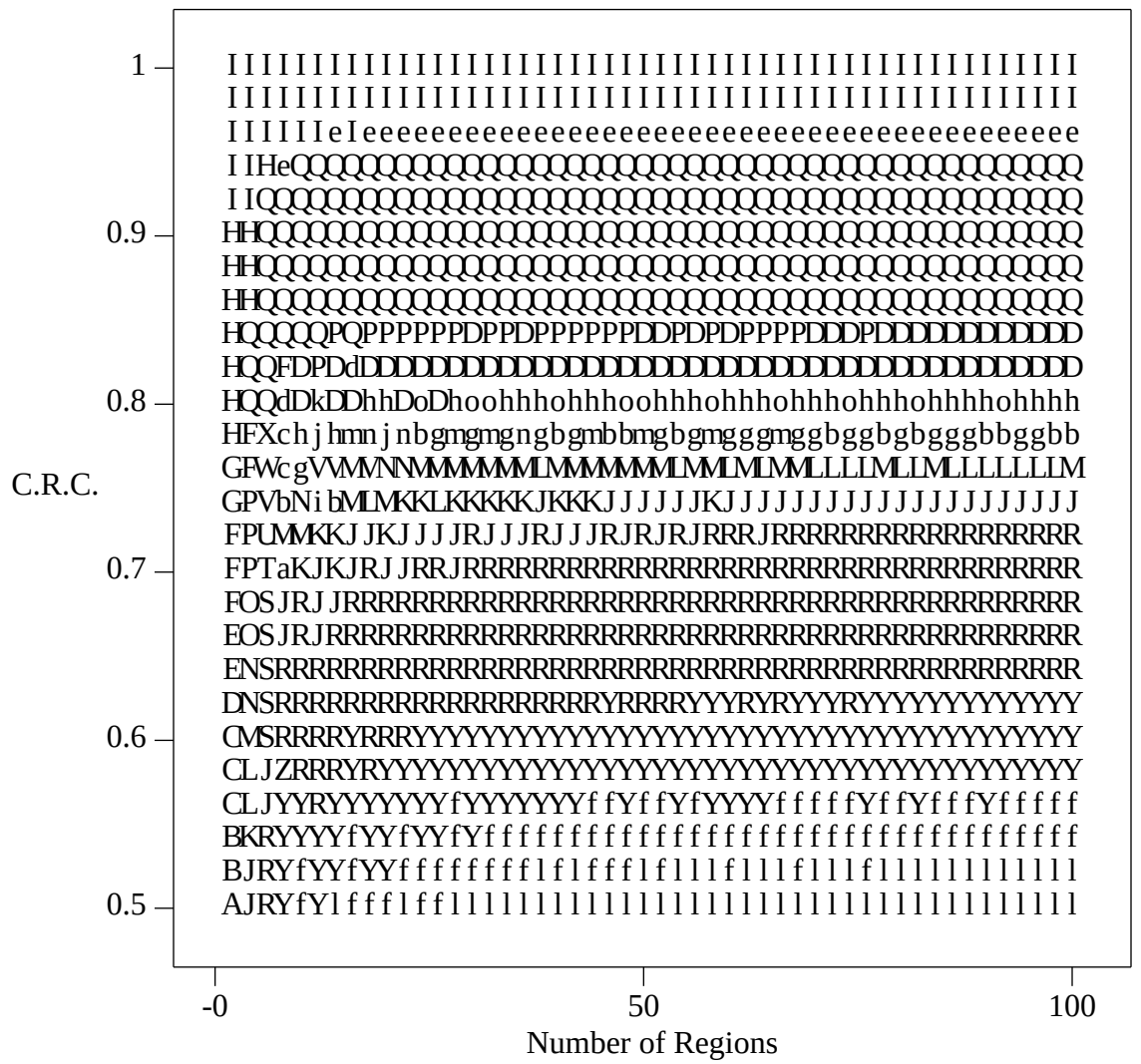


Figure B.10. Cluster plot obtained from learning pruning heuristics used in dynamic band search to solve a feasible solution of the TSP, where the learning is performed with the RR strategy and a random seed 61801.

Listing of Cluster Sets

Cluster Set A: ((7) (2 4) (8 11) (3 6 10 12))
Cluster Set B: ((7) (8 11) (2 4 6) (3 10 12))
Cluster Set C: ((7) (3 12) (8 11) (2 4 6 10))
Cluster Set D: ((7) (12) (3 6) (8 11) (2 4 10))
Cluster Set E: ((7) (12) (6) (2 3) (4 10) (8 11))
Cluster Set F: ((7) (12) (4) (8) (11) (2 10) (3 6))
Cluster Set G: ((2) (7) (12) (4) (8) (11) (3 6 10))
Cluster Set H: ((2) (7) (12) (4) (8) (10) (11) (3 6))
Cluster Set I: ((2) (7) (12) (3) (4) (6) (8) (10) (11))
Cluster Set J: ((8 11) (2 7 10) (3 4 6 12))
Cluster Set K: ((6) (8 11) (2 7 10) (3 4 12))
Cluster Set L: ((3 12) (4 6) (8 11) (2 7 10))
Cluster Set M: ((12) (8 11) (2 7 10) (3 4 6))
Cluster Set N: ((2 7) (10 12) (8 11) (3 4 6))
Cluster Set O: ((7) (12) (3) (8) (11) (2 4 6 10))
Cluster Set P: ((7) (12) (8) (11) (3 6) (2 4 10))
Cluster Set Q: ((7) (12) (8) (10) (11) (2 4) (3 6))
Cluster Set R: ((8 11) (3 6 12) (2 4 7 10))
Cluster Set S: ((7 10) (8 11) (2 3 4 6 12))
Cluster Set T: ((12) (7 10) (8 11) (2 3 4 6))
Cluster Set U: ((7) (4 6) (8 11) (2 3 10 12))
Cluster Set V: ((7) (8 11) (2 10 12) (3 4 6))
Cluster Set W: ((7) (8) (11) (2 10 12) (3 4 6))
Cluster Set X: ((7) (12) (8) (11) (4 10) (2 3 6))
Cluster Set Y: ((7 8 11) (2 3 4 6 10 12))
Cluster Set Z: ((3 12) (8 11) (2 4 6 7 10))
Cluster Set a: ((3) (8 11) (2 7 10) (4 6 12))
Cluster Set b: ((2 12) (7 10) (8 11) (3 4 6))
Cluster Set c: ((12) (2 4) (7 10) (3 6) (8 11))
Cluster Set d: ((7) (12) (4) (2 10) (3 6) (8 11))
Cluster Set e: ((7) (12) (3) (6) (8) (10) (11) (2 4))
Cluster Set f: ((2 7 8 11) (3 4 6 10 12))
Cluster Set g: ((12) (2 7) (8 11) (3 4 6 10))
Cluster Set h: ((7) (12) (3 10) (8 11) (2 4 6))
Cluster Set i: ((7 10) (4 6) (8 11) (2 3 12))
Cluster Set j: ((7) (12) (10) (8 11) (2 3 4 6))
Cluster Set k: ((7) (12) (8) (11) (2 10) (3 4 6))
Cluster Set l: ((3 6 10 12) (2 4 7 8 11))
Cluster Set m: ((12) (10) (2 7) (8 11) (3 4 6))
Cluster Set n: ((7) (10) (2 12) (8 11) (3 4 6))
Cluster Set o: ((7) (12) (2 10) (8 11) (3 4 6))

Table B.19. Summary of the most generalized cluster set, the most specialized cluster set, and the chosen cluster set (cluster set P) from learning pruning heuristics of the TSP, where the cluster plot is shown in Figure B.10.

The Most Generalized Cluster Set			
Subdomains	Heuristics ID	Probability of Wins	Average Speedup
2,3,4,6,7,8,10,11,12	257	0.563	4.039
The Most Specialized Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
2	622	0.706	10.139
3	577	0.676	5.075
4	622	0.802	4.185
6	256	0.704	11.346
7	528	0.820	10.576
8	211	0.743	11.164
10	622	0.832	6.702
11	241	0.903	4.542
12	635	0.713	7.426
The Chosen Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
7	528	0.820	10.576
12	635	0.713	7.426
8	211	0.745	11.164
11	241	0.903	4.542
3,6	577	0.650	7.443
2,4,10	622	0.780	7.009

Table B.20. Summary of the learned heuristics mentioned in Table B.19.

ID	Learned Coefficient	Learned Heuristics	Simplified Heuristics
1	1.0e-12	l	l
211	4	(h4 + ((dh - (l / (_zerop (h)))) * dh))	$dh^2+h4-\frac{dh*l}{h}$
241	16	(h1 + (dh - h3))	h1+dh−h3
256	8	((dh - (l / (_zerop (h)))) * (((dh - (l / (_zerop (h)))) * dh) - h))	$\left(dh-\frac{l}{h}\right)*\left(dh^2-h-\frac{dh*l}{h}\right)$
257	8	((h4 + (dh - h3)) * ((dh - h) + (dh - (l / (_zerop (h))))))	$(dh-h3+h4)*(2*dh-h-\frac{l}{h})$
528	4	((((dh - (l / (_zerop (h)))) * d) * (h1 + d)) - (((dh - h) + (dh - h3)) - (g - h2)))	$-2dh+g+h-h2+h3+d*(d+h1)*(dh-\frac{l}{h})$
577	8	((((((dh - (l / (_zerop (h)))) * d) * (h1 + d)) - (((dh - (l / (_zerop (h)))) * (d + (dh - h3))) / (_zerop (h2)))) - g) / (_zerop ((dg - h2))))	$\left[g+d*(d+h1)*(dh-\frac{l}{h})-\frac{(d+dh-h3)*(dh-\frac{l}{h})}{h2}\right]/(dg-h2)$
622	problem_size	((d - h2) / (_zerop ((l / (_zerop (h))))))	$\frac{h*(d-h2)}{l}$
635	problem_size	(((((dh - (l / (_zerop (h)))) * d) * (h1 + d)) - (((dh - (l / (_zerop (h)))) * (d + (dh - h3))) / (_zerop (h2)))) - l)	$-l+d*(d+h1)*(dh-\frac{l}{h})-\frac{(d+dh-h3)*(dh-\frac{l}{h})}{h2}$

B.5. Experimental Results of Learning Search Heuristics for the Maze Problems

In this section, we will show the cluster plots, cluster sets, and learned selection and pruning heuristics used in band search for solving the maze problem.

B.5.1. Learning selection heuristics used in band search to find a solution

In this section, we show in Figure B.11 the cluster plots of learned selection heuristics used in a dynamic band search to solve a feasible solution of the maze problem. The learning strategy is the MR strategy with a random seed 61801. Following Figure B.11, we list cluster sets shown in the plot. In Table B.21, we summarize the most generalized cluster set, the most specialized cluster set, and the chosen cluster set. In Table B.22, we show the learned selection heuristics mentioned in Table B.21.

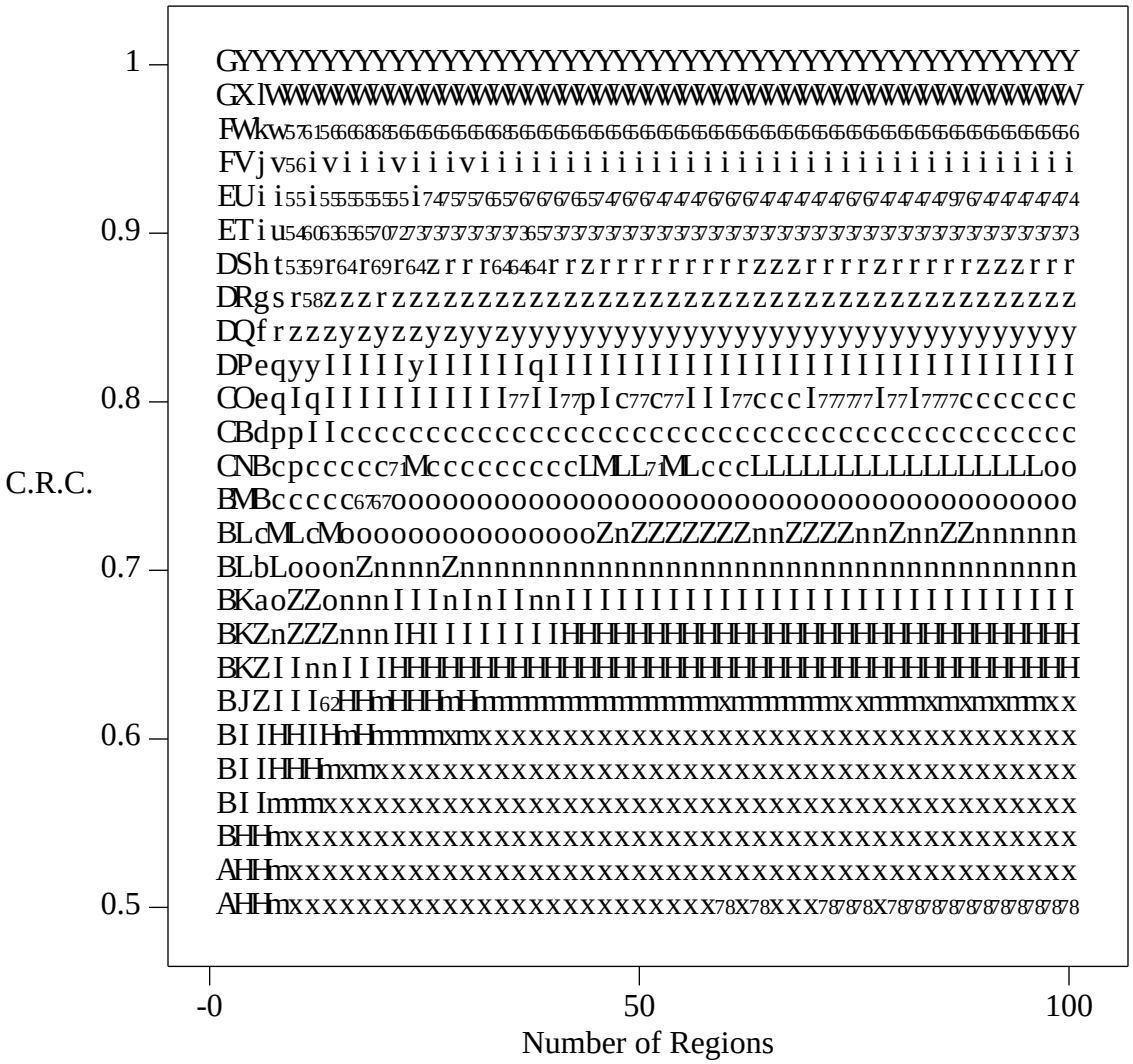


Figure B.11. Cluster plot obtained from learning selection heuristics used in dynamic band search to solve a feasible solution of the maze problem, where the learning is performed with the MR strategy and a random seed 61801.

Listing of Cluster Sets

Cluster Set A: ((11) (2 4) (5 10 12) (1 3 6 7 8 9 13 14))
Cluster Set B: ((11) (2 4) (6 7) (1 3 5 8 9 10 12 13 14))
Cluster Set C: ((2) (11) (4) (6 7) (1 3 5 8 9 10 12 13 14))
Cluster Set D: ((2) (11) (4) (6) (7) (1 3 5 8 9 10 12 13 14))
Cluster Set E: ((2) (11) (4) (6) (7) (10 12) (1 3 5 8 9 13 14))
Cluster Set F: ((2) (11) (4) (5) (6) (7) (1 3 8 9 10 12 13 14))

Cluster Set G: ((2) (11) (4) (5) (6) (7) (10 12) (1 3 8 9 13 14))
Cluster Set H: ((2 4 11) (1 3 5 6 7 8 9 10 12 13 14))
Cluster Set I: ((7) (2 4 11) (1 3 5 6 8 9 10 12 13 14))
Cluster Set J: ((2 4 11) (3 5 10 14) (1 6 7 8 9 12 13))
Cluster Set K: ((2 4 11) (3 5 10 13 14) (1 6 7 8 9 12))
Cluster Set L: ((2 4 11) (1 6 7 8 9) (3 5 10 12 13 14))
Cluster Set M: ((2 4 11) (1 6 7 9) (3 5 8 10 12 13 14))
Cluster Set N: ((11) (2 4) (1 6 7) (3 5 8 9 10 12 13 14))
Cluster Set O: ((11) (9) (2 4) (6 7) (1 3 5 8 10 12 13 14))
Cluster Set P: ((11) (7) (2 4) (3 5 13 14) (1 6 8 9 10 12))
Cluster Set Q: ((11) (7) (2 4) (1 6 8 9 12) (3 5 10 13 14))
Cluster Set R: ((2) (11) (4) (7) (1 6 8 9 12) (3 5 10 13 14))
Cluster Set S: ((2) (11) (4) (7) (1 13 14) (3 5 10) (6 8 9 12))
Cluster Set T: ((1) (2) (11) (4) (5) (7) (3 10 13 14) (6 8 9 12))
Cluster Set U: ((2) (11) (4) (5) (7) (6 12) (3 10 14) (1 8 9 13))
Cluster Set V: ((2) (11) (4) (5) (6) (7) (1 14) (3 10) (8 13) (9 12))
Cluster Set W: ((1) (2) (11) (4) (5) (6) (7) (12) (3 10) (8 9) (13 14))
Cluster Set X: ((1) (2) (11) (3) (4) (5) (6) (7) (8) (9) (10) (12) (13 14))
Cluster Set Y: ((1) (2) (11) (3) (4) (5) (6) (7) (8) (9) (10) (12) (13) (14))
Cluster Set Z: ((2 4 11) (3 10 12 14) (1 5 6 7 8 9 13))
Cluster Set a: ((2 4 11) (3 5 10 12 14) (1 6 7 8 9 13))
Cluster Set b: ((2 4 11) (1 6 7 9 13) (3 5 8 10 12 14))
Cluster Set c: ((1 6 7) (2 4 11) (3 5 8 9 10 12 13 14))
Cluster Set d: ((11) (6) (7) (2 4) (1 3 5 8 9 10 12 13 14))
Cluster Set e: ((11) (7) (2 4) (5 14) (1 3 6 8 9 10 12 13))
Cluster Set f: ((11) (7) (2 4) (5 13 14) (1 3 6 8 9 10 12))
Cluster Set g: ((2) (11) (4) (7) (3 5 13 14) (1 6 8 9 10 12))
Cluster Set h: ((2) (11) (4) (7) (1 10 13) (3 5 14) (6 8 9 12))
Cluster Set i: ((2) (11) (4) (5) (7) (6 8 9 12) (1 3 10 13 14))
Cluster Set j: ((2) (11) (4) (5) (6) (7) (14) (3 8 12) (1 9 10 13))
Cluster Set k: ((2) (11) (4) (5) (6) (7) (14) (3 10) (8 12) (1 9 13))
Cluster Set l: ((1) (2) (11) (3) (4) (5) (6) (7) (10) (12) (13) (14) (8 9))
Cluster Set m: ((2 4 7 11) (1 3 5 6 8 9 10 12 13 14))
Cluster Set n: ((2 4 11) (3 10 12) (1 5 6 7 8 9 13 14))
Cluster Set o: ((2 4 11) (3 10 12 13 14) (1 5 6 7 8 9))
Cluster Set p: ((6 7) (2 4 11) (1 3 5 8 9 10 12 13 14))
Cluster Set q: ((6) (7) (2 4 11) (1 3 5 8 9 10 12 13 14))
Cluster Set r: ((11) (7) (2 4) (6 9 12) (1 3 5 8 10 13 14))
Cluster Set s: ((11) (7) (2 4) (1 6 9 12 13) (3 5 8 10 14))
Cluster Set t: ((2) (11) (4) (7) (13) (3 5 14) (1 6 8 9 10 12))
Cluster Set u: ((2) (11) (4) (5) (7) (10 13 14) (1 3 6 8 9 12))
Cluster Set v: ((2) (11) (4) (5) (7) (1 13 14) (3 10 12) (6 8 9))
Cluster Set w: ((2) (11) (3) (4) (5) (6) (7) (1 9) (10 12) (8 13 14))
Cluster Set x: ((2 4 5 7 11) (1 3 6 8 9 10 12 13 14))
Cluster Set y: ((7) (6 12) (2 4 11) (1 3 5 8 9 10 13 14))
Cluster Set z: ((11) (7) (2 4) (6 12) (1 3 5 8 9 10 13 14))

Cluster Set 53: ((11) (7) (2 4) (3 5 14) (1 6 8 9 10 12 13))
Cluster Set 54: ((2) (11) (4) (5) (7) (12 14) (1 3 6 8 9 10 13))
Cluster Set 55: ((2) (11) (4) (5) (7) (1 3 13 14) (6 8 9 10 12))
Cluster Set 56: ((2) (11) (4) (5) (7) (10 12) (6 8 9) (1 3 13 14))
Cluster Set 57: ((2) (11) (4) (5) (7) (1 13) (3 14) (10 12) (6 8 9))
Cluster Set 58: ((11) (7) (2 4) (6 8 9 12) (1 3 5 10 13 14))
Cluster Set 59: ((11) (7) (14) (2 4) (1 3 5) (6 8 9 10 12 13))
Cluster Set 60: ((2) (11) (4) (5) (7) (14) (1 3 6 8 9 10 12 13))
Cluster Set 61: ((2) (11) (4) (5) (6) (7) (3 14) (1 9 13) (8 10 12))
Cluster Set 62: ((10 12) (2 4 11) (1 3 5 6 7 8 9 13 14))
Cluster Set 63: ((2) (11) (4) (7) (14) (1 3 5) (6 8 9 10 12 13))
Cluster Set 64: ((11) (7) (2 4) (6 9 12 13) (1 3 5 8 10 14))
Cluster Set 65: ((2) (11) (4) (7) (1 3 5 14) (6 8 9 10 12 13))
Cluster Set 66: ((2) (11) (3) (4) (5) (7) (6 8) (1 13 14) (9 10 12))
Cluster Set 67: ((2 4 11) (1 5 6 7) (3 8 9 10 12 13 14))
Cluster Set 68: ((2) (11) (3) (4) (5) (7) (10 12) (1 13 14) (6 8 9))
Cluster Set 69: ((11) (7) (2 4) (1 3 5 10 14) (6 8 9 12 13))
Cluster Set 70: ((2) (11) (4) (7) (3 5 14) (1 6 8 9 10 12 13))
Cluster Set 71: ((2 4 11) (1 6 7 8) (3 5 9 10 12 13 14))
Cluster Set 72: ((2) (11) (4) (7) (3 5 10 14) (1 6 8 9 12 13))
Cluster Set 73: ((2) (11) (4) (7) (1 3 5 10 14) (6 8 9 12 13))
Cluster Set 74: ((2) (11) (4) (7) (3 5) (12 13 14) (1 6 8 9 10))
Cluster Set 75: ((2) (11) (4) (7) (3 5) (1 13 14) (6 8 9 10 12))
Cluster Set 76: ((2) (11) (4) (5) (7) (3 12 13 14) (1 6 8 9 10))
Cluster Set 77: ((1 7) (2 4 11) (3 5 6 8 9 10 12 13 14))
Cluster Set 78: ((1 2 4 5 7 11) (3 6 8 9 10 12 13 14))
Cluster Set 79: ((2) (11) (4) (5) (7) (12 13 14) (1 3 6 8 9 10))

Table B.21. Summary of the most generalized cluster set, the most specialized cluster set, and the chosen cluster set (cluster set z') from learning selection heuristics of the maze problems, where the cluster plot is shown in Figure B.11. (Cluster Set z' is modified from cluster set z by merging two clusters into one because both conclude the same best heuristic.)

The Most Generalized Cluster Set			
Subdomains	Heuristics ID	Probability of Wins	Average Speedup
1,2,3,4,5,6,7,8,9,10,11,12,13,14	135	0.648	2.469
The Most Specialized Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
1	185	0.688	4.721
2	404	0.906	3.920
3	568	0.755	2.367
4	767	0.820	5.583
5	497	0.786	3.384
6	295	0.720	3.810
7	135	0.872	2.851
8	842	0.753	1.700
9	257	0.747	1.710
10	595	0.759	1.653
11	587	0.940	4.488
12	540	0.633	1.410
13	135	0.771	2.000
14	257	0.794	1.756
The Chosen Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
11	587	0.940	4.488
7	135	0.872	2.851
2,4	404	0.715	3.334
1,3,5,6,8,9,10,12,13,14	767	0.668	2.305

0

Table B.22. Summary of the learned heuristics mentioned in Table B.21.

ID	Learned Heuristics	Simplified Heuristics
1	l	l
135	(h + ((- g) + ((- g) / (_zerop ((g + (LC * h)))))))	$-g+h-\frac{g}{g+h*LC}$
185	((- g) + ((- g) + ((- g) + ((- g) / (_zerop ((g + (LC * h))))))))	$-3*g-\frac{g}{g+h*LC}$
257	(((- g) + ((- g) + ((- g) / (_zerop ((g + (LC * h))))))) + ((- g) + ((- g) / (_zerop ((g + (LC * h)))))))	$-3*g-\frac{2*g}{g+h*LC}$
295	(dl + ((((- g) + ((- g) + ((- g) / (_zerop ((g + (LC * h))))))) + (((- g) + ((- g) + ((- g) / (_zerop ((g + (LC * h))))))) - ((- g) + h2))) - h2))	$dl-3*g-2*h2-\frac{2*g}{g+h*LC}$
404	(h2 - g)	h2−g
497	((h2 - g) / (_zerop (l)))	$\frac{h2-g}{l}$
540	(((h2 - g) / (_zerop (l))) + (((h2 - g) / (_zerop (l))) + ((((- g) + ((- g) + ((- g) / (_zerop ((g + (LC * h))))))) - g) - h2)))	$-3*g-h2+\frac{2*(h2-g)}{l}-\frac{g}{g+h*LC}$
568	(((h2 - g) / (_zerop (l))) / (_zerop (h)))	$\frac{-g+h2}{h*l}$
587	(l - (LC * g))	l−LC*g
595	((l - (LC * g)) + ((- g) + ((- g) + h2)))	−2*g+h2+l−g*LC
767	(h1 + ((LC * g) + ((l - (LC * g)) + ((l - (LC * g)) + ((l - (LC * g)) + ((- g) + ((- g) + h2)))))))	−2*g+h1+h2+3*l−2*g*LC
842	((l - (LC * g)) / (_zerop (((LC * g) + ((LC * g) + ((l - (LC * g)) + ((l - (LC * g)) + ((l - (LC * g)) + ((- g) + ((- g) + h2))))))))))	$\frac{l-g*LC}{-2*g+h2+3*l-g*LC}$

B.5.2. Learning selection heuristics used in band search

In this section, we show in Figure B.12 the cluster plots of learned selection heuristics used in a dynamic band search to solve an optimal solution of the maze problem. The learning strategy is the MR strategy with a random seed 61801. Following Figure B.12, we

list cluster sets shown in the plot. In Table B.23, we summarize the most generalized cluster set, the most specialized cluster set, and the chosen cluster set. In Table B.24, we show the learned selection heuristics mentioned in Table B.23.

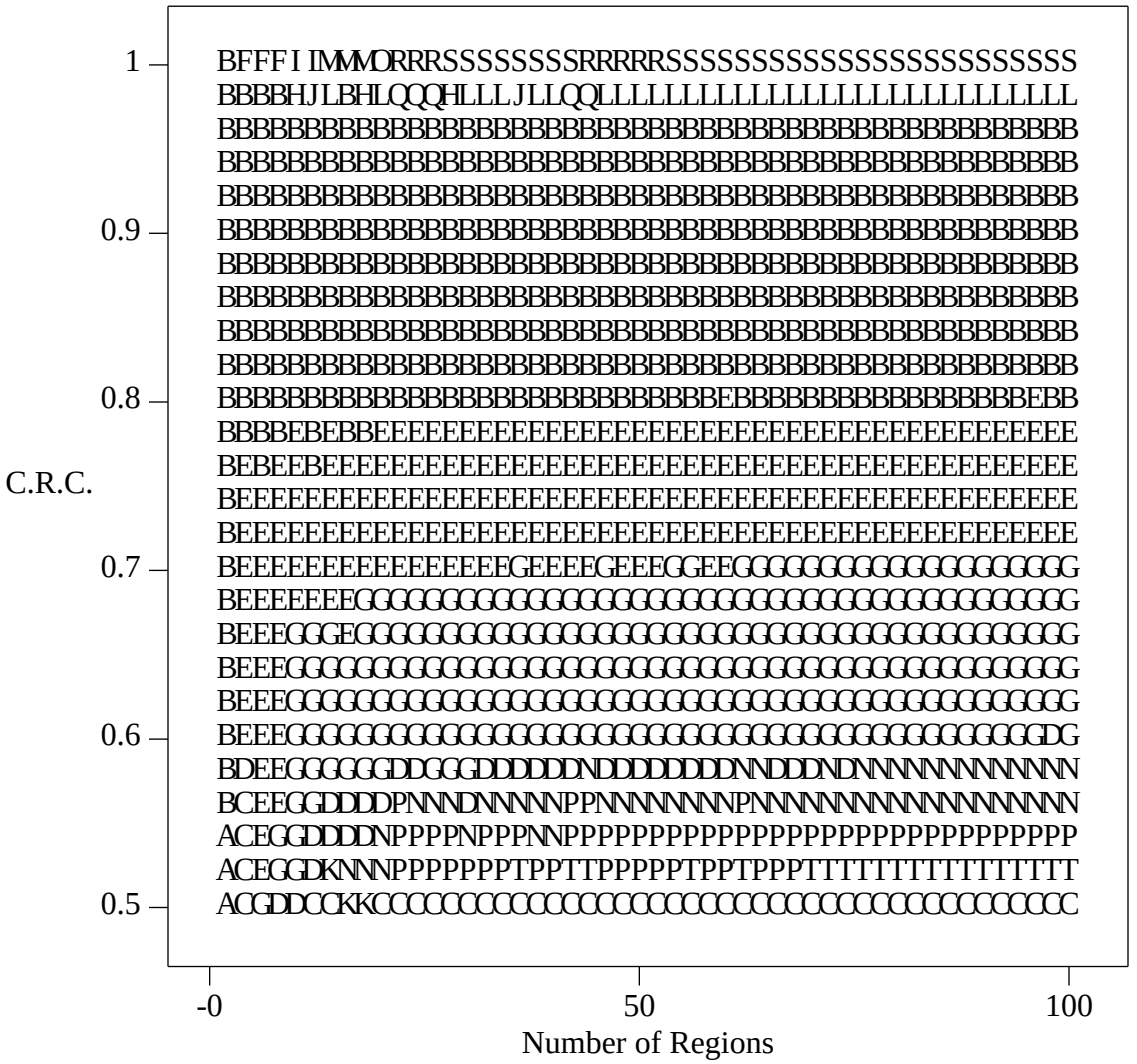


Figure B.12. Cluster plot obtained from learning selection heuristics used in dynamic band search to solve an optimal solution of the maze problems, where the learning is performed with the MR strategy and a random seed 61801.

Listing of Cluster Sets

Cluster Set A: ((6) (1 2 3 4 5 7 8 9 10 11 12 13 14))
Cluster Set B: ((6) (7) (1 2 3 4 5 8 9 10 11 12 13 14))
Cluster Set C: ((1 2 3 4 5 6 7 8 9 10 11 12 13 14))
Cluster Set D: ((1 5 6 7) (2 3 4 8 9 10 11 12 13 14))
Cluster Set E: ((6 7) (1 2 3 4 5 8 9 10 11 12 13 14))
Cluster Set F: ((1) (5) (6) (7) (2 3 4 8 9 10 11 12 13 14))
Cluster Set G: ((5 6 7) (1 2 3 4 8 9 10 11 12 13 14))
Cluster Set H: ((5) (6) (7) (1 2 3 4 8 9 10 11 12 13 14))
Cluster Set I: ((1) (2) (5) (6) (7) (3 4 8 9 10 11 12 13 14))
Cluster Set J: ((6) (7) (1 5) (2 3 4 8 9 10 11 12 13 14))
Cluster Set K: ((1 2 4 5 6 7) (3 8 9 10 11 12 13 14))
Cluster Set L: ((6) (7) (1 2 5) (3 4 8 9 10 11 12 13 14))
Cluster Set M: ((1) (2) (4) (5) (6) (7) (3 8 9 10 11 12 13 14))
Cluster Set N: ((1 2 5 6 7) (3 4 8 9 10 11 12 13 14))
Cluster Set O: ((1) (2) (4) (5) (6) (7) (9) (3 8 10 11 12 13 14))
Cluster Set P: ((1 2 5 6 7 11) (3 4 8 9 10 12 13 14))
Cluster Set Q: ((6) (7) (1 2 5 11) (3 4 8 9 10 12 13 14))
Cluster Set R: ((1) (2) (11) (4) (5) (6) (7) (3 9) (8 10 12 13 14))
Cluster Set S: ((1) (2) (11) (3) (4) (5) (6) (7) (9) (8 10 12 13 14))
Cluster Set T: ((1 2 4 5 6 7 11) (3 8 9 10 12 13 14))

Table B.23. Summary of the most generalized cluster set, the most specialized cluster set, and the chosen cluster set (cluster set B) from learning selection heuristics of the maze problems, where the cluster plot is shown in Figure B.12.

The Most Generalized Cluster Set			
Subdomains	Heuristics ID	Probability of Wins	Average Speedup
1,2,3,4,5,6,7,8,9,10,11,12,13,14	340	0.568	1.001
The Most Specialized Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
1	660	0.776	1.015
2	498	0.861	1.012
3	266	0.852	1.000
4	391	0.858	1.001
5	654	0.825	1.004
6	134	0.911	1.082
7	654	0.804	1.127
8	1	0.500	1.000
9	1	0.532	1.000
10	1	0.500	1.000
11	733	0.830	1.011
12	1	0.562	1.000
13	266	0.851	1.000
14	1	0.500	1.000
The Chosen Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
6	134	0.911	1.082
7	654	0.804	1.127
1,2,3,4,5,8,9,10,11,12,13,14	1	0.577	1.000

0

Table B.24. Summary of the learned heuristics mentioned in Table B.23.

ID	Learned Heuristics	Simplified Heuristics
1	l	l
134	((l + (h - h1)) + (((LC * g) + h) * (LC * g)))	$h-h1+l+g*h*LC+g^2*LC^2$
266	((h1 - ((l + (h - h1)) + dh)) / (_zerop (((l + (h - h1)) + (((LC * g) + h) * (LC * g))))) / (_zerop (h)))	$\frac{-dh-h+2*h1-l}{h*(h-h1+l+g*h*LC+g^2*LC^2)}$
340	((((l + (h - h1)) + (((LC * g) + h) * (LC * g))) - (((LC * g) + h) * (LC * g))) * (l + g))	(g+l)*(h−h1+l)
391	(g + ((l + (h - h1)) + (l - h1)))	g+h−2*h1+2*l
498	(((((l + (h - h1)) + (l - h1)) - dl) * ((l + (h - h1)) + (l - h1))) + (((l + (h - h1)) + (l - h1)) - h2))	h−2*h1−h2+2*l+ (h−2*h1+2*l)*(−dl+h−2*h1+2*l)
654	(((((l + (h - h1)) + (l - h1)) - h2) * g) + (((l + (h - h1)) + (l - h1)) - h2))	(1+g)*(h−2*h1−h2+2*l)
660	((((l + (h - h1)) + (l - h1)) - h2) * (g * ((dl + ((l + (h - h1)) + (l - h1))) - h2)))	g*(h−2*h1−h2+2*l)* (dl+h−2*h1−h2+2*l)
733	((((((l + (h - h1)) + (l - h1)) - h2) * g) + dl) * (dl + ((LC * g) + ((dl + ((l + (h - h1)) + (l - h1))) - h2))))	(dl+g*(h−2*h1−h2+2*l))* (2*dl+h−2*h1−h2+2*l+g*LC)

B.5.3. Learning pruning heuristics used in band search

In this section, we show in Figure B.13 the cluster plots of learned pruning heuristics used in a dynamic band search to solve a feasible solution of the maze problem. The learning strategy is the MR strategy with a random seed 61801. Following Figure B.13, we list cluster sets shown in the plot. In Table B.25, we summarize the most generalized cluster set, the most specialized cluster set, and the chosen cluster set. In Table B.26, we show the learned pruning heuristics mentioned in Table B.25.

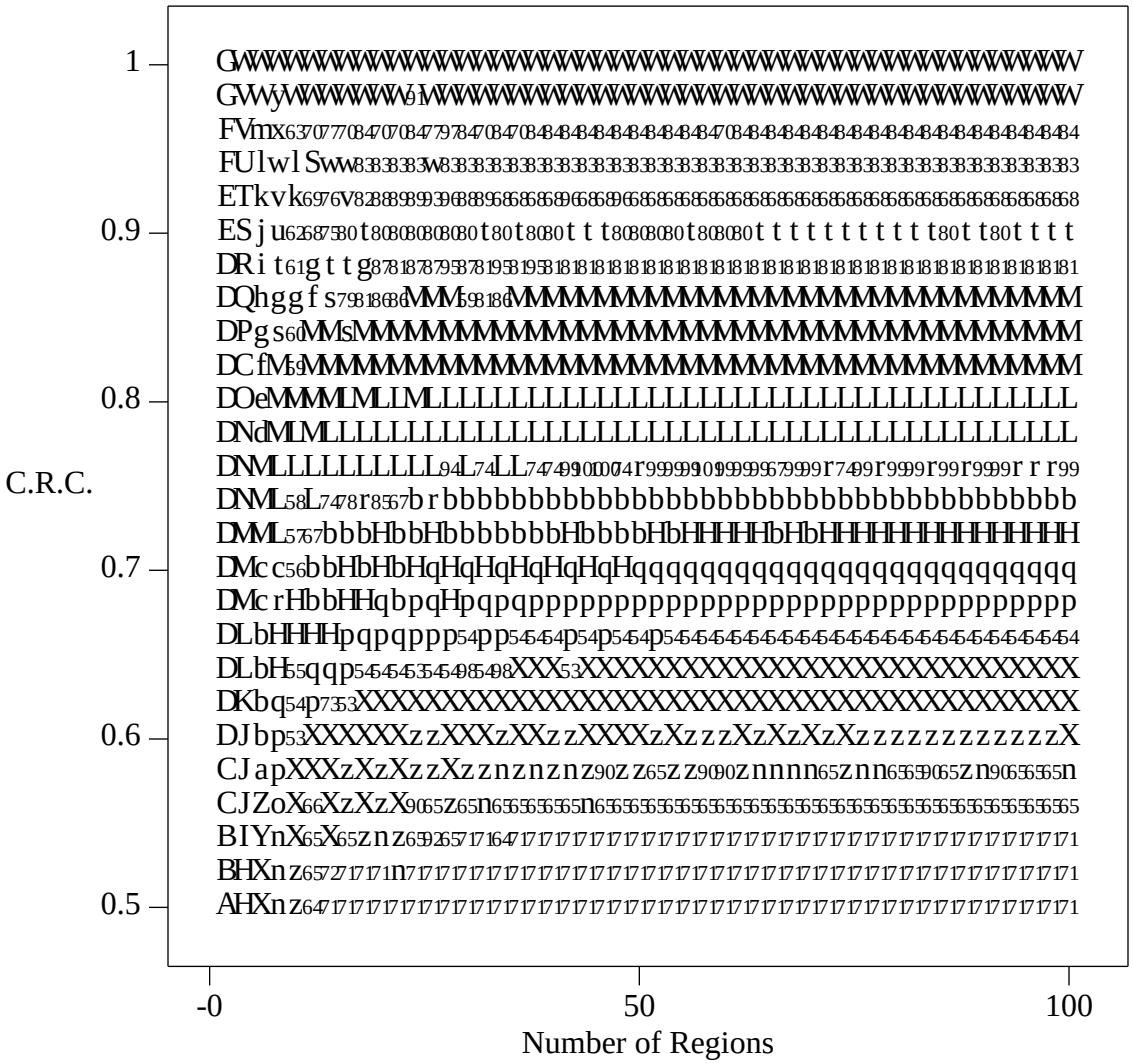


Figure B.13. Cluster plot obtained from learning pruning heuristics used in dynamic band search to solve a feasible solution of the maze problem, where the learning is performed with the MR strategy and a random seed 61801.

Listing of Cluster Sets

Cluster Set A: ((2) (11) (12) (7 14) (1 3 4 5 6 8 9 10 13))
Cluster Set B: ((2) (11) (12) (13) (7 14) (1 3 4 5 6 8 9 10))
Cluster Set C: ((2) (11) (12) (7 14) (10 13) (1 3 4 5 6 8 9))
Cluster Set D: ((2) (11) (12) (7 14) (9 10 13) (1 3 4 5 6 8))
Cluster Set E: ((2) (11) (12) (13) (7 14) (9 10) (1 3 4 5 6 8))
Cluster Set F: ((1) (2) (11) (7) (9) (12) (14) (10 13) (3 4 5 6 8))

Cluster Set G: ((1) (2) (11) (7) (9) (10) (12) (13) (14) (3 4 5 6 8))
Cluster Set H: ((2 12) (7 8 11 13 14) (1 3 4 5 6 9 10))
Cluster Set I: ((2 12) (7 8 11) (9 10 13 14) (1 3 4 5 6))
Cluster Set J: ((2) (7 11 12) (1 3 4 5 6) (8 9 10 13 14))
Cluster Set K: ((2) (7 11 12) (8 10 13 14) (1 3 4 5 6 9))
Cluster Set L: ((2) (7 11) (12 13 14) (1 3 4 5 6 8 9 10))
Cluster Set M: ((2) (11) (7 12 13 14) (1 3 4 5 6 8 9 10))
Cluster Set N: ((2) (11) (7 8) (12 13 14) (1 3 4 5 6 9 10))
Cluster Set O: ((2) (11) (7 13) (12 14) (1 3 4 5 6 8 9 10))
Cluster Set P: ((2) (11) (12) (13) (7 14) (8 9 10) (1 3 4 5 6))
Cluster Set Q: ((2) (11) (10) (12) (13) (7 14) (8 9) (1 3 4 5 6))
Cluster Set R: ((2) (11) (6) (12) (13) (4 8) (7 14) (9 10) (1 3 5))
Cluster Set S: ((2) (11) (8) (12) (13) (4 6) (7 14) (9 10) (1 3 5))
Cluster Set T: ((2) (11) (7) (8) (12) (13) (14) (4 6) (9 10) (1 3 5))
Cluster Set U: ((1) (2) (11) (6) (7) (8) (12) (13) (14) (9 10) (3 4 5))
Cluster Set V: ((1) (2) (11) (3) (6) (7) (8) (12) (13) (14) (4 5) (9 10))
Cluster Set W: ((1) (2) (11) (3) (4) (5) (6) (7) (8) (9) (10) (12) (13) (14))
Cluster Set X: ((1 5 8 11) (2 7 12 13 14) (3 4 6 9 10))
Cluster Set Y: ((1 5 11) (2 7 12 13 14) (3 4 6 8 9 10))
Cluster Set Z: ((2 12 14) (1 5 7 11 13) (3 4 6 8 9 10))
Cluster Set a: ((2 12 14) (7 11 13) (1 3 4 5 6 8 9 10))
Cluster Set b: ((2 12) (7 11 13 14) (1 3 4 5 6 8 9 10))
Cluster Set c: ((2) (12 13) (7 11 14) (1 3 4 5 6 8 9 10))
Cluster Set d: ((2) (11) (7 12 14) (1 3 4 6) (5 8 9 10 13))
Cluster Set e: ((2) (11) (9 13) (7 12 14) (1 3 4 5 6 8 10))
Cluster Set f: ((2) (11) (9) (7 13) (12 14) (1 3 4 5 6 8 10))
Cluster Set g: ((2) (11) (12) (9 10) (7 13 14) (1 3 4 5 6 8))
Cluster Set h: ((2) (11) (12) (13) (7 14) (1 3 5) (4 6 8 9 10))
Cluster Set i: ((1) (2) (11) (12) (13) (7 14) (8 9 10) (3 4 5 6))
Cluster Set j: ((2) (11) (12) (13) (4 6) (7 14) (9 10) (1 3 5 8))
Cluster Set k: ((2) (11) (12) (13) (7 14) (9 10) (1 5 8) (3 4 6))
Cluster Set l: ((2) (11) (3) (7) (12) (4 6) (9 10) (13 14) (1 5 8))
Cluster Set m: ((1) (2) (11) (6) (7) (12) (13) (14) (3 4) (5 8) (9 10))
Cluster Set n: ((4 9 10) (2 7 12 13 14) (1 3 5 6 8 11))
Cluster Set o: ((2 7 12 13) (4 6 9 10) (1 3 5 8 11 14))
Cluster Set p: ((2 12) (3 4 6 9 10) (1 5 7 8 11 13 14))
Cluster Set q: ((2 12) (1 3 4 6 9 10) (5 7 8 11 13 14))
Cluster Set r: ((2) (7 11 12 13 14) (1 3 4 5 6 8 9 10))
Cluster Set s: ((2) (11) (9) (7 12 13 14) (1 3 4 5 6 8 10))
Cluster Set t: ((2) (11) (12) (7 13 14) (8 9 10) (1 3 4 5 6))
Cluster Set u: ((2) (11) (6) (12) (13) (7 14) (1 3 5) (4 8 9 10))
Cluster Set v: ((2) (11) (3) (12) (13) (1 8) (7 14) (9 10) (4 5 6))
Cluster Set w: ((2) (11) (12) (13) (1 3) (4 6) (5 8) (7 14) (9 10))
Cluster Set x: ((1) (2) (11) (7) (8) (12) (3 5) (4 6) (9 10) (13 14))
Cluster Set y: ((1) (2) (11) (3) (4) (5) (6) (7) (8) (12) (13) (14) (9 10))
Cluster Set z: ((4 6 9 10) (1 3 5 8 11) (2 7 12 13 14))

Cluster Set 53: ((2 7 12 13) (1 5 8 11 14) (3 4 6 9 10))
Cluster Set 54: ((2 7 12) (3 4 6 9 10) (1 5 8 11 13 14))
Cluster Set 55: ((2 12) (1 7 8 11 13 14) (3 4 5 6 9 10))
Cluster Set 56: ((14) (2 12) (7 11 13) (1 3 4 5 6 8 9 10))
Cluster Set 57: ((2 12) (7 11) (1 3 4 5 6) (8 9 10 13 14))
Cluster Set 58: ((2) (7 11 12) (8 13 14) (1 3 4 5 6 9 10))
Cluster Set 59: ((2) (11) (7 12 14) (8 9 13) (1 3 4 5 6 10))
Cluster Set 60: ((2) (11) (7 12) (8 13 14) (1 3 4 5 6 9 10))
Cluster Set 61: ((2) (11) (12) (7 13 14) (1 3 5 8) (4 6 9 10))
Cluster Set 62: ((2) (11) (12) (7 14) (8 13) (1 3 5) (4 6 9 10))
Cluster Set 63: ((1) (2) (11) (4) (6) (7) (8) (12) (3 5) (9 10) (13 14))
Cluster Set 64: ((11) (2 7 12 13 14) (1 3 4 5 6 8 9 10))
Cluster Set 65: ((9 10) (2 7 12 13 14) (1 3 4 5 6 8 11))
Cluster Set 66: ((3 6 9 10) (1 4 5 8 11) (2 7 12 13 14))
Cluster Set 67: ((2) (13) (7 11 12 14) (1 3 4 5 6 8 9 10))
Cluster Set 68: ((2) (11) (6) (12) (1 3 5) (7 13 14) (4 8 9 10))
Cluster Set 69: ((2) (11) (12) (13) (1 8) (7 14) (3 5 6) (4 9 10))
Cluster Set 70: ((2) (11) (3) (7) (8) (12) (1 5) (4 6) (9 10) (13 14))
Cluster Set 71: ((2 7 11 12 13 14) (1 3 4 5 6 8 9 10))
Cluster Set 72: ((3 9 10) (2 7 12 13 14) (1 4 5 6 8 11))
Cluster Set 73: ((2 7 12 13) (5 8 11 14) (1 3 4 6 9 10))
Cluster Set 74: ((2) (7 11 12) (1 3 4 6 10) (5 8 9 13 14))
Cluster Set 75: ((2) (11) (9) (12) (1 8 10) (7 13 14) (3 4 5 6))
Cluster Set 76: ((2) (11) (12) (1 8) (9 10) (7 13 14) (3 4 5 6))
Cluster Set 77: ((1) (2) (11) (6) (12) (13) (3 4) (5 8) (7 14) (9 10))
Cluster Set 78: ((2) (8 9 14) (7 11 12 13) (1 3 4 5 6 10))
Cluster Set 79: ((2) (11) (12) (8 9) (7 13 14) (1 3 4 5 6 10))
Cluster Set 80: ((2) (11) (12) (6 9 10) (7 13 14) (1 3 4 5 8))
Cluster Set 81: ((2) (11) (9) (12) (7 13 14) (1 3 4 5 6 8 10))
Cluster Set 82: ((2) (11) (12) (1 8) (3 5 6) (4 9 10) (7 13 14))
Cluster Set 83: ((2) (11) (3) (12) (13) (4 6) (7 14) (9 10) (1 5 8))
Cluster Set 84: ((2) (11) (3) (8) (12) (13) (1 5) (4 6) (7 14) (9 10))
Cluster Set 85: ((2 12) (7 11 13) (8 9 10 14) (1 3 4 5 6))
Cluster Set 86: ((2) (11) (12) (7 13 14) (1 3 4 5 6 8 9 10))
Cluster Set 87: ((2) (11) (12) (1 3 8) (7 13 14) (4 5 6 9 10))
Cluster Set 88: ((2) (11) (6) (12) (13) (7 14) (4 9 10) (1 3 5 8))
Cluster Set 89: ((1) (2) (11) (12) (7 13 14) (8 9 10) (3 4 5 6))
Cluster Set 90: ((6 9 10) (2 7 12 13 14) (1 3 4 5 8 11))
Cluster Set 91: ((1) (2) (11) (3) (4) (5) (6) (7) (8) (9) (10) (12) (13 14))
Cluster Set 92: ((10) (2 7 12 13 14) (1 3 4 5 6 8 9 11))
Cluster Set 93: ((2) (11) (12) (3 6) (1 5 8) (4 9 10) (7 13 14))
Cluster Set 94: ((2) (7 11 12) (8 9 13 14) (1 3 4 5 6 10))
Cluster Set 95: ((2) (11) (12) (1 3) (7 13 14) (4 5 6 8 9 10))
Cluster Set 96: ((2) (11) (6) (12) (4 9 10) (7 13 14) (1 3 5 8))
Cluster Set 97: ((1) (2) (11) (3) (12) (13) (4 6) (5 8) (7 14) (9 10))
Cluster Set 98: ((2 7 12 14) (1 5 8 11 13) (3 4 6 9 10))

Cluster Set 99: ((2) (7 11 12 13) (5 8 9 14) (1 3 4 6 10))
Cluster Set 100: ((2) (13 14) (7 11 12) (1 3 4 5 6 8 9 10))
Cluster Set 101: ((2) (14) (7 11 12 13) (1 3 4 5 6 8 9 10))

Table B.25. Summary of the most generalized cluster set, the most specialized cluster set, and the chosen cluster set (cluster set M') from learning pruning heuristics of the maze problems, where the cluster plot is shown in Figure B.13. (Cluster Set R' is modified from cluster set M by merging the subdomains that conclude the baseline heuristic as the best.)

The Most Generalized Cluster Set			
Subdomains	Heuristics ID	Probability of Wins	Average Speedup
1,2,3,4,5,6,7,8,9,10,11,12,13,14	724	0.689	1.095
The Most Specialized Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
1	407	0.660	2.022
2	407	0.883	2.022
3	407	0.736	1.242
4	645	0.688	1.076
5	627	0.727	1.317
6	478	0.739	1.277
7	1	0.978	1.000
8	1	0.869	1.000
9	162	0.809	1.003
10	262	0.725	1.058
11	704	0.842	2.022
12	1	0.997	1.000
13	262	0.945	1.010
14	627	0.899	1.079
The Chosen Cluster Set			
Cluster	Heuristics ID	Probability of Wins	Average Speedup
2	407	0.883	2.022
11	704	0.842	2.022
13,14	627	0.917	1.056
7,8,12	1	0.948	1.000
1,3,4,5,6,9,10	639	0.636	1.203

0

Table B.26. Summary of the learned heuristics mentioned in Table B.25.

ID	Learned Coefficient	Learned Heuristics	Simplified Heuristics
1	1.0e-12	l	l
162	4	((h - (((pv * g) + h) - (g + h))) / (_zerop (((h - (((pv * g) + h) - (g + h))) + ((pv * g) + h)))))	$\frac{g+h-g*pv}{g+2*h}$
262	8	((h - (((pv * g) + h) - (g + h))) / (_zerop (l)))	$\frac{g+h-g*pv}{l}$
407	16	(((l - (((pv * g) + h) - (g + h))) - g) / (_zerop ((h + g))))	$\frac{l-g*pv}{g+h}$
478	8	(((h - (((pv * g) + h) - (g + h))) - ((h - (((pv * g) + h) - (g + h))) / (_zerop (l)))) - g)	$\frac{-g-h+h*l+g*pv-g*l*pv}{l}$
627	8	(((h - (((pv * g) + h) - (g + h))) / (_zerop (l))) - (dh - (((pv * g) + h) - (g + h))))	$\frac{g+h-dh*l-g*l-g*pv+g*l*pv}{l}$
639	16	(((dh - (((pv * g) + h) - (g + h))) / (_zerop (l))) - h2)	$\frac{g+h-dh*l-g*l-g*pv+g*l*pv}{l}$
645	8	((dh - (((pv * g) + h) - (g + h))) + l)	dh+g+l−g*pv
704	16	(((h - (((pv * g) + h) - (g + h))) / (_zerop (l))) - h2)	$\frac{g+h-h2*l-g*pv}{l}$
724	8	(dl + ((h - (((pv * g) + h) - (g + h))) / (_zerop (l))))	$\frac{g+h+dl*l-g*pv}{l}$

VITA

Lon-Chan Chu was born in Keelung, Taiwan, on April 27, 1963. He obtained his B.S. degree in Electrical Engineering at National Taiwan University in June 1985. He obtained his M.S. degree in Electrical Engineering at the University of Illinois at Urbana-Champaign in May 1991. He is currently a candidate for the Ph.D. degree in Electrical Engineering at the University of Illinois at Urbana-Champaign. He joined the Digital Office Systems Division at Microsoft Corporation in August 1993.

Mr. Chu has published and presented papers at various conferences and journals on parallel processing, real-time systems, artificial intelligence, and neural networks. His research interests include real-time systems, parallel processing, embedded systems, and artificial intelligence.