University of California
Los Angeles

Incremental Search Methods for Real-Time Decision Making

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Joseph Carl Pemberton

1995

© Copyright by
Joseph Carl Pemberton
1995

The dissertation of Joseph Carl Pemberton is approved.

Reza Ahmadi

Eli Gafni

John Mamer

Judea Pearl

Richard Korf, Committee Chair

University of California, Los Angeles
1995

To Tanya and Paul.

Table of Contents

1 Introduction and Overview 1

1.1 Example: Manufacturing Scheduling 1

1.2 Example: Airplane-Takeoff Scheduling 3

1.3 Example: Robot Traveling Salesperson Problem 4

1.4 General Problem Characteristics 4

1.5 Summary and Overview 7

1.5.1 Chapter 2: A Problem Space Formulation of Real-Time Decision Making 8

1.5.2 Chapter 3: Exploring the Problem Space and Deciding When to Stop 8

1.5.3 Chapter 4: Making an Optimal Incremental Decision . . . 9

1.5.4 Chapter 5: Making a Single Incremental Decision - A Comparison of MINIMIN and $E(MPC)$ 9

1.5.5 Chapter 6: Approximating the Optimal Decision Method . 9

1.5.6 Chapter 7: Real-Time Search - Making a Sequence of Incremental Decisions 10

1.5.7 Chapter 8: Learning the Completion-Cost Distribution . . 11

1.5.8 Chapter 9: Incremental Search and Flowshop Scheduling . 11

1.5.9 Chapter 10: Related Work 11

1.5.10 Chapters 11 and 12: Future Work, Contributions and Conclusions 12

2 A Problem-Space Formulation of Real-Time Decision Making 13

2.1 Problem-Space Formulation 13

2.2 General Random-Tree Problem Description 16

2.3 An Argument for Reproducible Random Trees 17

2.4 How Should We Generate Random Trees? 18

2.5 The Incremental Decision Problem 20

2.6 The Incremental Search Approach 25

2.6.1 What is Incremental Search? 25

2.6.2 The Three Sub-Tasks of Incremental Search 26

3 Exploring the Problem Space and Deciding When to Stop . . 28

3.1 The Exploration Problem 28

3.2 Background on Exploration 29

3.3 Heuristic Exploration Methods 32

3.3.1 Depth-First Branch-and-Bound Exploration 32

3.3.2 Best-First Exploration 35

3.4 The Stopping Problem 36

3.5 Summary 38

4 Making an Optimal Incremental Decision 39

4.1 Analysis of Incremental Decision Making 39

4.2 Making Optimal Last Incremental Decisions 46

4.2.1 Calculating the Expected Minimum Path Cost ($E(MPC)$) 46

4.2.2 How Difficult is it to Calculate $E(MPC)$? 49

5 Making a Single Incremental Decision: A Comparison of MINIMIN and $E(MPC)$ 54

5.1 Making a Last Incremental Decision 54

5.1.1 The MINIMIN Decision Method 55

5.1.2 The Decision-Theoretic Approach ($E(MPC)$) 56

5.1.3 How Much Can We Expect to Gain by Using $E(MPC)$? . 56

5.2 Experimental Comparison: $E(MPC)$ versus MINIMIN. 59

5.2.1 Balanced Exploration 60

5.2.2 Best-First Exploration 61

5.3 Evaluation of Larger Search Trees 65

5.3.1 Using Monte-Carlo Simulation to Evaluate Larger Search Trees 67

5.3.2 Monte-Carlo Simulation Results 68

6 Approximating the Optimal Decision Method 69

6.1 k-best: An Approximate $E(MPC)$ Algorithm 69

6.1.1 Description of the k-best Algorithm 70
6.1.2 How Good is the k-best Approximation? 70
6.1.3 Exploring to Find the k-best Frontier Nodes 72
6.1.4 Experimental Comparison of k-best and MINIMIN 73
6.1.5 Discussion 80
6.2 Alpha-bound k-best 81
6.2.1 Description of the Alpha-Bound k-best Algorithm 81
6.2.2 Results Comparing αk-best with MINIMIN 82
6.2.3 Discussion 82

7 Real-Time Search: Making a Sequence of Incremental Decisions 86
7.1 Approximating Each Decision as if it Were the Last 87
7.2 Comparison of MINIMIN and αk-best 87
7.3 Real-Time Search Algorithms 93
7.3.1 Real-Time Branch-and-Bound Algorithms 93
7.3.2 Real-Time Traditional Best-First Algorithms 96
7.3.3 Alternative Real-Time Best-First Algorithms 99
7.4 Results for Random-Tree Experiments 101
7.5 Discussion of Other Experiments 107

8 Learning the Completion-Cost Distribution 111
8.1 How to Sample the Completion-Cost Distribution 111
8.2 Results for Sampling Random Trees 115

9 Incremental Search and Flowshop Scheduling 119
9.1 What is Flowshop Scheduling? 119
9.2 Real-Time Flowshop Scheduling 123
9.3 Three Approaches to Real-Time Scheduling 124
9.4 Experiment and Results 128

10 Related Work 134
10.1 Background 134
10.1.1 Searching Random Trees with Limited Resources 134

10.1.2 Real-Time Search . . . 135
10.1.3 Decision-Theoretic Search (DTA*) . . . 136
10.2 Other Related Work . . . 139
10.2.1 Random Trees . . . 139
10.2.2 Resource-Bounded Decision Making . . . 139
10.2.3 Scheduling . . . 141
10.2.4 Related Search Methods . . . 142

11 Open Problems and Future Work . . . 143
11.1 Extensions . . . 143
11.2 Open Issues . . . 147
11.3 Future Plans . . . 147

12 Contributions and Conclusions . . . 151

A Generating Reproducible Incremental Random Search Trees . 155
A.1 The Method . . . 155
A.2 Random Seed Code . . . 157

B Expected Minimum-Path-Cost Equations . . . 159
B.1 Calculating the Frontier-Node Completion-Cost Distribution . . . 159
B.1.1 E(Single Level Greedy Decision) . . . 160
B.1.2 General Recursive Equations for Expected Minimum Path Cost . . . 161
B.1.3 Detailed Example: $E(MPC)$ for a Depth-Two Binary Tree. 162
B.2 Calculating the Expected Minimum Path Cost . . . 167
B.3 $E(MPC(T(b=2,s=3,u=1)))$ . . . 171
B.3.1 The Algebraic Equation . . . 171
B.3.2 Maple Session to Produce $E(MPC(B_1))$ for $[0,1]$ Edge Costs 172
B.3.3 $E(MPC)$ When the Completion-Cost Cumulative Distribution is of the Form $(1-e^{-\lambda x})$ . . . 176

C Monte-Carlo Simulation Results . . . 177

D Support Code for Sampling the Completion-Cost Distribution 182

E Estimating the Cost of a Flowshop Schedule 185

E.1 Estimating the Complete Schedule Cost 185

E.2 Code for Ignall and Schrage's Flowshop Scheduling Heuristic Evaluation Function 188

References 191

List of Figures

1.1 Simple flowshop scheduling problem, (a) initial set of jobs, (b) processing times, and (c) a sample schedule. 2

1.2 Mutchler's example exploration problem. 6

1.3 Example of the real-time decision task: Which node is a better decision, B_1 or B_2? 7

2.1 Problem-space representation for a four-job, two-machine flowshop scheduling problem. 14

2.2 Example of a random tree problem. 17

2.3 Example random trees: (a) b=2, and (b) branching factor = b. . 19

2.4 Example binary tree with all but one leaf node generated. . . . 21

2.5 Example of Mutchler's decision problem. 23

2.6 Example of the general incremental decision problem. 24

2.7 Example of the general incremental search process. 26

3.1 Mutchler's very limited resource search problem. 30

3.2 Example of depth-first exploration. 33

3.3 Example of iterative-deepening depth-first branch-and-bound exploration. 34

3.4 General best-first search algorithm. 35

4.1 Example of the last incremental decision problem. Which node is a better decision, B_1 or B_2? 40

4.2 Example where the optimal last incremental decision and second-to-last incremental decision differ. 42

4.3 The difference between the probability of finding a zero-cost path below B_2, and the probability of finding a zero-cost path below B_1, given the probability, p, of a zero-cost edge in the unexplored part of the example tree. 44

4.4 Completion-cost cumulative distribution functions for root-to-leaf paths through nodes B_1, C_1 and C_2 when (a) $y_2 = y_1 + 0.2$ and (b) $y_2 = y_1 + 0.5$. 48

4.5 Set of recursion equations for calculating the expected minimum path cost in a random tree. 50

4.6 Completion-cost cumulative-distribution functions for the minimum root-to-leaf path cost through nodes B_1, C_1, C_2 and C_3 when $y_2 = y_1 + 0.2$ and $y_3 = y_2 + 0.2$ and the associated search tree (b). 52

5.1 An example of MINIMIN backed-up values. 56

5.2 Example of a choice between one minimum-cost path and many paths with near minimum cost. 57

5.3 Four balanced search trees 61

5.4 Example of a balanced best-first search tree. 62

5.5 Example of an unbalanced search tree. 63

5.6 Four unbalanced search trees 65

6.1 Example of the cost of using the k-best approximation to make incremental decisions. 71

6.2 Example of an exploration to find the 2-best frontier nodes, (a) part of the way through the exploration, and (b) an example of the complete exploration. 74

6.3 Average percent solution cost error relative to optimal versus search depth for MINIMIN and k-best ($k\epsilon\{2,7\}$). 75

6.4 Average percent solution cost error relative to optimal versus k for k-best ($k\epsilon\{1,2,3,4,5,6,7\}$). 76

6.5 Average generations per decision versus search depth for MINIMIN and $E(MPC, k\epsilon\{1,2,7\})$. 79

6.6 Average percent error versus average node generations ($b = 2$). . 80

6.7 Average percent error versus average node generations ($b = 2$). . 82

7.1 Average percent error relative to optimal versus search depth for MINIMIN, α2-best and α7-best. 89

7.2 Percent wins by α7-best and MINIMIN versus tree depth. 92

7.3 Example of best-first *swap-pathology*. 97

7.4 Example of uneven exploration problem. 99

7.5 Average percent error versus node generations available per decision for depth 20 binary trees with real-time node generation constraint. 102
7.6 Percent wins over 20 decisions for α2-best BnB and idMINIMIN versus available generations per decision. 104
7.7 Percent wins over 20 decisions for α7-best BnB and idMINIMIN versus available generations per decision. 105
7.8 Percent wins over 20 decisions for 7-best BFS versus *hybrid BFS*. 106
7.9 Average percent error versus maximum edge cost (generations per decision $\epsilon\{10, 100, 1000\}$). 110

8.1 An example search tree. 112
8.2 Example of buckets for a sampled completion-cost distribution. . 113
8.3 Average percent error relative to optimal versus search depth for MINIMIN BnB and sα2-best BnB. 116

9.1 Simple flowshop scheduling problem, (a) two machine flowshop, (b) jobs and their processing times, and (c) a sample schedule. . . 120
9.2 Problem-space representation for a four-job, two-machine flowshop scheduling problem. 122
9.3 Average schedule cost versus average node generations per job. . 129
9.4 Average schedule cost versus average node generations per job (tBnB, idMINIMIN, and sαk-best BnB). 131
9.5 Average schedule cost per job versus number of jobs in schedule. 132
9.6 Average schedule cost per job versus average run-time used. . . . 133

10.1 Example of DTA* node selection. 138

11.1 Example tree where the proximity of the second best frontier node cost affects the incremental decision. 148

B.1 Summary of recursion equations for calculating the expected minimum path cost in a random tree. 162
B.2 Four choices for the minimum-cost path in a depth 2 binary tree. 163
B.3 Example tree for last incremental decision problem. 167
B.4 Example tree $T(b = 2, s = 3, u = 1)$. 171

List of Tables

5.1 Results comparing $E(MPC)$ decisions with MINIMIN decisions for balanced random trees. 60

5.2 Best-first results comparing distribution-based decisions with MINIMIN decisions for balanced search trees. 63

5.3 Results comparing distribution-based decisions with four different MINIMIN decision strategies on unbalanced best-first search trees. 66

6.1 Percent wins for 2-best versus MINIMIN. 77

6.2 Percent wins for 7-best versus MINIMIN. 78

6.3 Percent wins for α2-best versus MINIMIN. 83

6.4 Percent wins for α7-best versus MINIMIN. 84

7.1 Percent wins over 15 decisions for α2-best versus MINIMIN. . . . 90

7.2 Percent wins over 15 decisions for α7-best versus MINIMIN. . . . 91

7.3 Table of algorithms considered. 101

7.4 Maximum number of frontier nodes (k) in k-best equations for different branching factors. 108

8.1 Percent wins over 20 decisions for sα2-best BnB and MINIMIN BnB versus search depth. 117

9.1 Percent wins for 16-job flowshop schedule for sα2-best versus. MINIMIN. 130

C.1 Results from Monte-Carlo simulation of last incremental decision problem ($b = 2$, $u = 1$). 178

C.2 Results from Monte-Carlo simulation of last incremental decision problem ($b = 3$, $u = 1$). 179

C.3 Results from Monte-Carlo simulation of last incremental decision problem ($b = 4$, $u = 1$). 179

C.4 Results from Monte-Carlo simulation of last incremental decision problem ($b = 2$, $u = 2$). 179

C.5 Percent same decision and average estimated expected difference between MINIMIN and $E(MPC)$ from Monte-Carlo simulations of the last incremental decision problem. 180

E.1 Processing times for four unscheduled jobs. 186

Acknowledgments

There are many people who have helped me get to this point in my life, and I would like to take this opportunity to thank some of them publicly.

First, I would like to thank my advisor, Rich Korf, for his support over the many years that I have been a graduate student. His dedication to high standards in both research and teaching has been an inspiration to me. He has also taught me how to critically evaluate my own work, and his insistence that my papers be accessible to more than just a few people has greatly improved my writing skills.

I would also like to thank the other members of my dissertation committee: Judea Pearl, Eli Gafni, Reza Ahmadi, and John Mamer. Judea taught me a valuable lesson: that people will be interested in my research only if I can get excited about it myself. In addition, he encouraged me to explore the decision-theoretic approach to real-time decision making, which ultimately led to the k-best algorithm. Eli helped me to formalize the theoretical aspects of my research through his diligent questioning of the basic tenets of my work. Reza and John both helped me to better understand the differences between artificial intelligence and operations research. Their different perspectives on scheduling problems has been very instructive to me, and has helped me to broaden the applicability of my research.

At the risk of forgetting someone, I would like to acknowledge and thank several people with whom I have had discussions that have influenced my research: Matt Ginsberg, David Heckerman, Eric Horvitz, David Mutchler, Weixiong Zhang.

I would especially like to thank my former officemate Weixiong Zhang for his friendship and the countless number of research discussions that we have had. Our close working relationship resulted in what was perhaps the most productive two weeks of research in my life. Our collaboration resulted in the epsilon-transformation method that has yielded both a conference and journal paper [ZP94a, ZP94b]. I hope that I have the opportunity to work with Weixiong again in the future.

In addition, a number of other people have had a positive impact on my working environment at UCLA. In particular, I would like to acknowledge the following people: Alex Balke, Max Chickering, Rob Collins, Dan Greening, Sheila Greibach, David Jefferson, Stott Parker, and Sek-Wah Tan. I would also like to thank Valerie Aylett for her friendship, and for keeping the administrative details in their place.

I would like to thank Gerald Estrin for his commitment to the CS202 seminar class. Through this class and his guidance, I have learned how to give a good presentation. In addition, his willingness to schedule a seminar on two days notice to help a student prepare for an exam or conference talk was often a life saver.

I am also very grateful to Verra Morgan. She and her support staff, Shelly and Rosemarie, were always there to help with the administrative nightmares that invariably plague a graduate student's life. In addition, Verra has been a true friend. She was always there to chastise my mistakes, congratulate my successes, console my frustration, and correct my Greek.

I would like to thank my family for their constant support over the years. My parents, my brothers and sisters, grandmother, my wife's parents and sister, and the other members of my extended family have all been very supportive of my academic pursuit.

I am deeply indebted to my wife Tanya. Without her support, understanding, and patience, I would never have gotten this far. Throughout my years in graduate school, she has always been there for me. I would also like to thank my son, Paul, for his patience and understanding during the many nights and weekends that I had to work instead of playing with him or taking him to the park.

Over the years, my research has been supported by an NSF Young Investigator Award, two NSF grants, #IRI-8552925 and #IRI-9119825, a grant from Rockwell International to Rich Korf, and an equipment grant from Hewlett-Packard. I would also like to thank CIRL for supporting me during the last few months of my dissertation research. My work at CIRL was supported by the Air Force Office of Scientific Research under contract 92-0693, by ARPA/Rome Labs under contracts F30602-91-C-0036 and F30602-93-C-00031, by an NSF Grant #IRI-9119825, and by a grant from Rockwell International to CIRL. My education and research at UCLA was also partially supported by a School of Engineering and Applied Science Fellowship (1985-1986).

Finally, I am very grateful to the following people for providing their software free of charge. The figures in this dissertation were prepared using *tgif*, which was written by William Chia-Wei Cheng. The graphs were prepared using both *tgif* and *xgraph* which was written by David Harrison. The text of this dissertation was edited using *gnuemacs*, and the code was developed and debugged using *gcc* and *gdb*, all three of which were provided by Richard Stallman and the Free Software Foundation. The text was also spell checked using *ispell*, which written by Pace Willisson and others. Lastly, this dissertation was formatted using a LaTeX macro package written by Rich Wales. Without the generosity of these people, I would have had a much harder time preparing my dissertation.

VITA

1982	B.S. Electrical Engineering Massachusetts Institute of Technology, Cambridge, MA.
1984–1985	Engineer Charles Stark Draper Laboratories, Cambridge, MA.
1986–1989	Post Graduate Research Engineer and Research Assistant Distributed Machine Intelligence Laboratory (DMI), University of California, Los Angeles.
1988	M.S. Computer Science University of California, Los Angeles.
1989–1994	Post Graduate Research Engineer Computer Science Department, University of California, Los Angeles.
1989, 1992	Teaching Assistant Computer Science Department, University of California, Los Angeles.
1993	Research Engineer Rockwell International Science Center, Palo Alto, CA.
1994–Present	Research Assistant Computational Intelligence Research Laboratory (CIRL), University of Oregon.

PUBLICATIONS

W. Zhang & J. C. Pemberton. *Epsilon-Transformation: Exploiting Phase Transitions to Solve Combinatorial Optimization Problems - Initial Results* In **Proceedings of the** 12^{th} **Annual Conference on Artificial Intelligence (AAAI-94)**, Seattle, WA, July 1994.

J. C. Pemberton & R. E. Korf. *Incremental Search Algorithms for Real-Time Decision Making.* In **Proceedings of the** 2^{nd} **International Conference on AI Planning Systems (AIPS-94)**, Chicago, June 1994.

R. E. Korf, J. C. Pemberton & W. Zhang. *Incremental Random Search Trees.* In **Working Notes of the AAAI Workshop on Experimental Evaluation of Reasoning and Search Methods**, Seattle, WA, July 1994.

J. C. Pemberton & R. E. Korf. *An Incremental Search Approach to Real-Time Decision Making.* In **Proceedings of the AAAI Spring Symposium on Decision Theoretic Planning**, Stanford, CA, March 1994.

W. Zhang & J. C. Pemberton. *Epsilon Transformation: Exploiting Phase Transitions to Solve Combinatorial Optimization Problems.* **UCLA Computer Science Department Technical Report No. UCLA-CSD-940003**, January 1994.

J. C. Pemberton & R. E. Korf. *An Incremental Search Approach to Real-Time Planning: Preliminary Results for a Scheduling Problem.* In **Proceedings of the AAAI Spring Symposium on Foundations of Automatic Planning: The Classical Approach and Beyond**, Stanford, CA, March 1993. **AAAI Press Technical Report No. SS-93-03**.

J. C. Pemberton & R. E. Korf. *An Incremental Search Approach to Real-Time Planning and Scheduling: Preliminary Results.* In **Proceedings of NSF Workshop on Intelligent, Dynamic Scheduling for Manufacturing Systems**, Cocoa Beach, FL, January 1993.

J. C. Pemberton & R. E. Korf. *Incremental Planning on Graphs with Cycles.* In **Proceedings of the First International Conference on AI Planning Systems (AIPS-92)** (Editor: J. Hendler), College Park, MD, June 1992.

J. C. Pemberton & R. E. Korf. *Making Locally Optimal Decisions on Graphs with Cycles.* **UCLA Computer Science Department Technical Report No. UCLA-CSD-920004**, January 1992.

C. M. Reed, W. M. Rabinowitz, N. I. Durlach, L. A., Delhorne, L. D. Braida, J. C. Pemberton, B. D. Mulcahey & D. L. Washington. *Analytic Study of the Tadoma Method: Improving Performance Through the Use of Supplementary Tac-*

tual Displays. **Journal of Speech and Hearing Research**, Volume 35, p. 450-465, April 1992.

J. C. Pemberton & J. J. Vidal. *The Effect of Training Signal Errors on Node Learning.* In **Proceedings of the IEEE International Joint Conference on Neural Networks**, Washington, D.C., June 1989.

J. C. Pemberton. *A Comparison of Continuous and Discrete Learning Through a Geometric Representation.* **(Master's Thesis), UCLA Computer Science Technical Report No. UCLA-CSD-890006**, January 1989.

J. C. Pemberton & J. J. Vidal. *Noise Immunity of Generalized Delta Rule Learning.* In **Proceedings of the International Neural Network Society Conference**, Boston, MA, September 1988.

J. C. Pemberton & J. J. Vidal. *When is the Generalized Delta Rule a Learning Rule?* In **Proceedings of the IEEE International Conference on Neural Networks**, San Diego, CA, July 1988.

J. J. Vidal, J. C. Pemberton & J. M. Goodwin. *Implementing Neural Nets with Programmable Logic.* In **Proceedings of the IEEE First International Conference on Neural Networks**, San Diego, CA, June 1987.

ABSTRACT OF THE DISSERTATION

Incremental Search Methods for Real-Time Decision Making

by

Joseph Carl Pemberton
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1995
Professor Richard Korf, Chair

Many real-world problems, such as air-traffic control and factory scheduling, require that a sequence of decisions be made in real time, without complete information. Since there is typically not sufficient time for traditional methods to find a complete solution before committing to a decision, we propose an incremental search method for making real-time decisions. Our approach is to separate the real-time decision task into three sub-problems: *where to spend limited computational resources?*, *when to stop computing?*, and *how to make decisions given incomplete information*? By interleaving computation with execution, we can use the execution time to improve the solution quality.

We present the last incremental decision problem as a simplification of the general incremental decision problem. We develop and analyze $E(MPC)$, which is an optimal last incremental decision method. We argue that $E(MPC)$ is impractical for large search trees, due to the size and complexity of the expected-value equations. We compare the performance of an existing incremental search algorithm (MINIMIN) to $E(MPC)$ on a set of small search trees. In general, MINIMIN typically makes very good decisions, choosing the optimal last incremental decision almost as frequently as $E(MPC)$.

We then develop an approximation to $E(MPC)$ called k-best. The idea is to approximate the $E(MPC)$ decision using the k-best frontier nodes under each child of the root. k-best makes the same decisions as MINIMIN when k equals one, and makes optimal decisions when each child of the root has k or less frontier nodes. Thus k-best defines a natural continuum between MINIMIN and optimal. k-best produces slightly better quality decisions than MINIMIN on random trees for a given search-depth. We extend the k-best algorithm and apply it to a flowshop-scheduling problem. Our results show that, although it is possible to improve slightly upon the performance of MINIMIN, the decision quality of

MINIMIN is often quite high. Since MINIMIN is efficient and easy to implement, we conclude that MINIMIN will often be the decision algorithm of choice. When we can further process the information that we have gathered, then k-best is a practical way to spend this additional computation.

CHAPTER 1

Introduction and Overview

In this chapter, we introduce the idea of real-time decision making. To illustrate this idea, we present three examples: scheduling jobs on a bottleneck resource in a manufacturing environment, scheduling the take-off or landing order of aircraft, and choosing a path for a robot traveling salesperson. We then summarize the general characteristics of the real-time decision making problems with which we are concerned. Finally, we present an overview of the rest of the dissertation.

1.1 Example: Manufacturing Scheduling

A good example of real-time decision making is factory scheduling. It is typical in a manufacturing environment for a single resource (*e.g.,* a milling machine) to directly impact the production rate (*i.e.*, throughput) of the factory. This resource is referred to as a *bottleneck*. When the objective of the plant manager is to maximize the plant throughput, then the primary concern is to keep a bottleneck resource busy while still minimizing the cost of production. In this case, the amount of time available to decide which job should be processed next is limited by the time required to process the current job on the bottleneck resource. Once a new job is scheduled, the time until the new job finishes processing can be used to decide on the next job. Normally there will not be enough time to find an optimal schedule due to the number of jobs to be scheduled, and the fact that new jobs can be added to the process over time.

For example, consider the problem of scheduling the processing of a set of jobs on two different machines, M_1 and M_2, shown in Figure 1.1a. Each job consists of one task for each machine, and it must be processed on the first machine, M_1, before it can be processed on the second machine, M_2. Associated with each task is the amount of time it takes to process the task on its respective machine (Figure 1.1b). In general, each job has a different processing time on each machine. Furthermore, each machine can only process one job at a time. One example of this problem is a set of typesetting jobs which must be preprocessed on a computer before being sent to a printer. In the static version of this problem, all jobs are available at the beginning, whereas in the dynamic flowshop scheduling

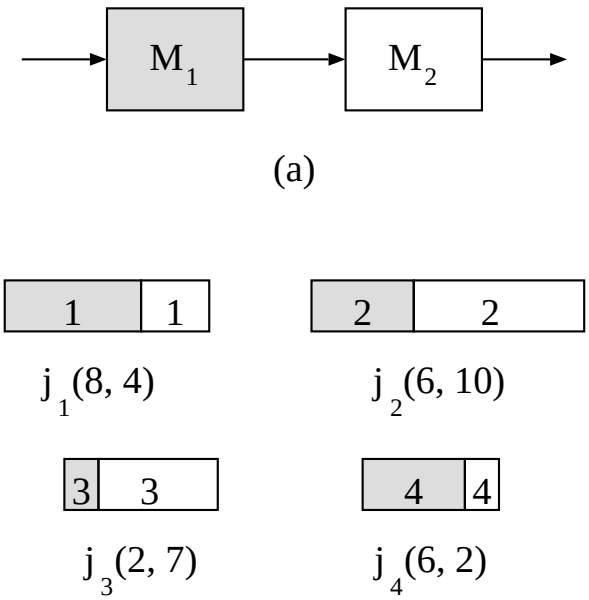


Figure 1.1: Simple flowshop scheduling problem, (a) initial set of jobs, (b) processing times, and (c) a sample schedule.

problem, new jobs may continually enter the system.

The task of the scheduler is to determine in real time the order in which jobs should be processed on the two machines, so that the cost of the resulting schedule is minimized. Example schedule cost functions include minimizing the average production time (*i.e.,* the average of the times that jobs are completed) and minimizing the total completion time (*i.e.,* the time that the last job finishes). For both of these cost functions, computation does not incur any cost as long as a scheduling decision is made before the current job finishes processing. Otherwise, there will be an additional processing cost associated with the idle time of the machines and the in-process inventory while the computation is being performed. If the schedule cost function is the average completion time, then the sample schedule shown in Figure 1.1c is optimal for the problem in Figure 1.1b. We will explore this problem further in Chapter 9.

1.2 Example: Airplane-Takeoff Scheduling

Everyday, airport ground controllers are faced with the problem of deciding what order planes should takeoff. This is a real-time decision problem because any time spent trying to optimize the takeoff sequence results in delays for all the planes that are waiting. There are many factors that affect when a plane will be ready for takeoff, including the time to load the plane, the time to check the mechanical systems, when the plane arrived, when the crew arrived, when connecting passengers arrived, and time to fix minor mechanical problems. Because these factors can vary significantly, it is not possible to optimally solve each takeoff sequencing problem in advance. Thus the problem must be solved in real-time while the planes are waiting for clearance to takeoff. The obvious dual to this problem is the task of sequencing the landing of airplanes.

Anyone who has missed a connecting flight because of too much time spent on a taxiway waiting to takeoff can appreciate the benefit that could be gained from considering connecting flight information in the takeoff decision process. For example, suppose that there are nine planes in line, waiting to takeoff. A tenth plane announces to the ground controllers that it is ready to taxi and takeoff, that it has two-hundred passengers that have to make a connecting flight, and that, because of earlier air-traffic delays, it is behind schedule. If it is allowed to takeoff immediately, then there is still a good chance that the passengers will make their connections. One approach would be to place the tenth plane at the end of the line. A more reasonable approach might be to allow the tenth plane to takeoff sooner, in order to increase the chance that its passengers will make their

connection. It is not immediately obvious what the optimal decision should be, since it depends on many different factors. In addition, the time spent trying to find the optimal decision needs to be factored into the cost of the final decision.

1.3 Example: Robot Traveling Salesperson Problem

Another good example of real-time decision making is a real-time version of the well known traveling salesperson problem (TSP) [Bod91]. The traveling salesperson problem is defined as follows. Given a set of cities, what is the shortest tour that visits each city once and returns to the initial city. This is a difficult problem that typically requires computation that is exponential in the number of cities to solve optimally.

The real-time version of the problem is defined as follows. A robot is given a set of cities, and must traverse a tour that visits each city once and returns to the initial city. The cost of the tour is simply the time between when the list of cities is given to the robot and when it returns to the initial city after visiting all the other cities. For simplicity, we assume that the robot either moves at a constant velocity v, or remains in the same location. The interesting part of this problem is that the robot can think while it is moving, thus the time spent traveling to the next city can be used to further calculate which of the remaining cities is the best next choice.

An anytime approach to this problem [DB88] assumes that there is some initial computation time after which the tour is executed without change. Anytime algorithms are designed to produce a complete solution when the computation time runs out. In addition, the quality of the solution is expected to improve on the average when more computation time is available. Our formulation of the real-time decision making-problem differs in that we explicitly allow the decision algorithm to update the tour as it is executed. Thus an incremental search algorithm will produce the next step of the solution when time runs out, and then start working on the following step.

1.4 General Problem Characteristics

The main characteristics of these sequential real-time decision-making problems are that there is a limited amount of time before a decision must be made, and the opportunity exists to interleave planning and execution. In general, this class of problems requires the problem solver to incrementally generate a sequence of time-limited decisions that result from performing the following three steps:

simulating the effect of future actions (*e.g.*, what is the cost of processing job J next, what are the consequences of a letting plane P_1 takeoff first), deciding when to stop simulating and make a decision (*i.e.*, what the decision deadline is), and making a decision based on the available information and results of the simulations (*e.g.*, what job should be processed next, which plane should takeoff next).

A good example of this approach to real-time decision making is game-playing programs. For example, a program that plays chess cannot explore all possible sequences of moves before committing to the first move. The standard approach is to search to a predetermined depth, backup heuristic values of the nodes at the bottom of the search tree, and use these values to make the next move decision. This is an example of incremental search for the case of two-player games. The three steps for making two-player game moves in real-time are: simulating and evaluating different sequences of moves (*e.g.,* what happens if I move my queen forward by two squares), deciding when to stop evaluating different move sequences and commit to a particular move (*i.e.,* have I evaluated enough game positions to make the current decision), and choosing a move (*i.e.,* given the partially explored game tree, what is the best move to make at this point). The main difference between the single-agent and two-player real-time decision problems is that the the single-agent has control over all decisions, whereas each player has control of their own decisions in a two-player game.

For a single-agent problem-space formulation of real-time decision making, simulating the effects of future actions corresponds to exploring the problem space (*i.e.,* expanding and evaluating nodes in the problem-space tree). We refer to this as the exploration problem. The stopping problem is the task of deciding when to stop exploring the problem space and commit to a decision associated with one of the root children. The stopping problem is difficult, because it requires understanding both the exploration and decision-making problems. The decision-making problem is the task of choosing an action or committing to a move based on the available information. This corresponds to choosing a child of the root node to become the next root node.

A specific case of the exploration problem has been previously examined by Mutchler [Mut86]. An example of Mutchler's exploration problem is shown in Figure 1.2. For this example, the problem solver can learn the actual cost of one more edge (b, c, or d) before it must choose a complete path from the root to a leaf node at the bottom of the tree. The cost of a complete solution path is the sum of the edge costs along the path from the root node to the leaf node, and the objective is to minimize the cost of the solution path. The exploration question at

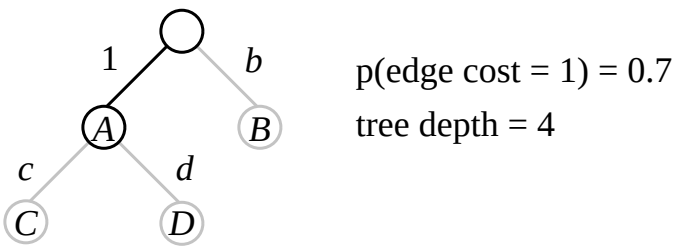


Figure 1.2: Mutchler's example exploration problem.

this point is: Which edge cost should we learn in order to make the best possible decision? Mutchler showed that the obvious best-first exploration strategy (*i.e.,* always expand the best node that hasn't already been expanded, which is node b in this example) is not always optimal. We will discuss this problem in more detail in Chapter 3.

The stopping problem involves deciding when to stop exploring the set of possible decision sequences and commit to a decision. For example, this corresponds to deciding how many partial schedules to consider before scheduling the next job for processing in the flowshop scheduling problem. In order to optimally solve the stopping problem, we first need to know the optimal solution to the exploration and decision-making problems. We have assumed that the stopping condition is part of the problem description, for example when the first flowshop machine finishes processing its current job. For our analysis and experiments, we assume that there is a strict limit on the amount of computation available. We also assume that the available computation is free in that the amount of available computation used does not affect the solution cost. In the flowshop scheduling example, any computation that is performed while the first machine is processing a job is not counted as part of the solution cost. We will briefly discuss the general issues related to the stopping problem in Chapter 3.

An example of the move-decision problem is shown in Figure 1.3. In the figure, the known edge costs are as shown in black, and we know that the gray edge costs will be chosen independently and at random from a uniform distribution over the range $[0, 1]$. The cost of a node is the sum of the edge costs along the path from the root to that node, and the objective is to traverse a path to a leaf node with low cost, subject to the computational constraint that limits the amount of the tree that can be explored. For this example, the edge costs in the remainder of the tree will become known only after a decision is made, and the question at this point is which is a better decision: node B_1 or node B_2? The reader is encouraged to stop at this point and make a choice before continuing. One answer is to use MINIMIN [Kor90], which is an existing real-time decision algorithm. For this

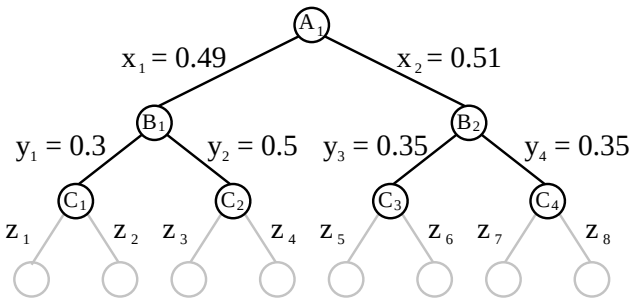


Figure 1.3: Example of the real-time decision task: Which node is a better decision, B_1 or B_2?

example, the MINIMIN move is to node B_1 because it is the first step along the path to the lowest cost frontier node ($cost(C_1) = 0.49 + 0.3 = .79$). The optimal decision, which depends on the known edge costs and the distribution of unknown edge costs, is to move to node B_2. We will return to this example in Chapter 4.

1.5 Summary and Overview

There are four main contributions of this dissertation. The first is our specification of the last incremental decision problem and development of the optimal approach to making the last incremental decision. The second contribution is the result that the optimal approach based on decision-theoretic techniques is impractical for all but a small set of search trees, due to the size and complexity of the equations necessary to make an optimal decision. The third contribution is a set of analytical and experimental results that demonstrate the unexpectedly high level of performance of MINIMIN on randomly generated decision problems. MINIMIN is an incremental search algorithm for real-time decision making previously described in [Kor90]. This high level of performance limits the amount of improvement that any new algorithm could obtain. Our fourth contribution is a new decision method called k-best. k-best is both an approximation of the optimal decision method, and an extension of MINIMIN.

The rest of this section contains a short summary of each of the remaining chapters.

1.5.1 Chapter 2: A Problem Space Formulation of Real-Time Decision Making

This chapter presents a problem-space formulation of real-time decision making. The basic idea is to map decisions into nodes, where arcs emanating from nodes correspond to the choices available at that node. Each edge in the problem-space tree has an associated cost that corresponds to the cost of the action it represents. We model the real-time constraint as a strict limit on the number of nodes that can be generated. A complete solution to the problem corresponds to a path in the problem-space tree from the root at the top of the tree, to a leaf node at the bottom of the tree. Incremental search methods generate a complete solution one step at a time as needed. In this mode, the exploration problem is deciding which nodes to generate, the limit on node generations per decision dictates when to stop, and the decision problem is to choose a child of the current root node to be the next root node.

In the second part of the chapter, we develop a random-tree model that allows us to build problem-space trees to our own specifications. We first describe our approach to generating random tree problems. These random trees are useful because we can control the problem space characteristics in order to better evaluate and compare our real-time search algorithms. Next, we discuss a method that we have developed for efficiently generating reproducible random trees in our experiments.

In the last part of the chapter, we present the incremental decision problem. We present the general incremental decision problem through a series of definitions. In particular, we present the last incremental decision problem, which is a special case of the general incremental decision problem. The general incremental decision is to choose a child of the current root node so that the average cost of the path traversed is minimized. The general incremental decision problem has no restrictions on the number of remaining decisions, whereas the last incremental decision problem focusses on the last incremental decision before the problem space is completely explored. This focus makes it easier to specify the optimal solution for making the last incremental decision.

1.5.2 Chapter 3: Exploring the Problem Space and Deciding When to Stop

Chapter 3 discusses both the exploration and stopping problems. We first define the exploration problem, and then present previous work by Mutchler on a simpler version of the exploration problem. We use Mutchler's results to argue

that our exploration problem is even more difficult, and thus we have focussed on simple, heuristic-based exploration methods: depth-first branch-and-bound and best-first search. The last part of this chapter briefly discusses the optimal stopping problem in the context of incremental search, and the assumptions that we make in lieu of solving the stopping problem.

1.5.3 Chapter 4: Making an Optimal Incremental Decision

In this chapter, we discuss and analyze the problem of making optimal incremental decisions. In the first section, we discuss how to make optimal incremental decisions for the different incremental decision problems described in Chapter 2. In the second section, we then develop the optimal decision method for making the last incremental decision, and argue that making an optimal last incremental decision is impractical in general. The impracticality is due to the size and complexity of the equations that could be necessary to express the expected cost of the incremental decision choices.

1.5.4 Chapter 5: Making a Single Incremental Decision - A Comparison of MINIMIN and $E(MPC)$

This chapter describes our work on how to make a single incremental decision based on partial information. We present the optimal decision-making algorithm for the last incremental decision, which moves to a child of the root node with the lowest expected minimum path cost ($E(MPC)$). We also present MINIMIN [Kor90], which is the best-known existing algorithm for this problem. A MINIMIN decision is to move to a child of the root on a minimum-cost path in the explored part of the problem space. The expected minimum path cost of a root child node is calculated based on the cost of all frontier nodes in the subtree below it. Since MINIMIN makes decisions based on the cost of only a single frontier node, we expected the optimal decision algorithm to produce much better decisions than MINIMIN. However, experimental and analytical results contained in this chapter show that MINIMIN performs very well relative to optimal on a wide variety of random tree problems. This is fortunate because the optimal decision-making algorithm is impractical for all but a few small search trees.

1.5.5 Chapter 6: Approximating the Optimal Decision Method

Since the optimal last incremental decision algorithm is impractical for most search trees, in this chapter we present an approximation to the optimal algo-

rithm. The complexity of expressing the optimal algorithm comes from the fact that the number of integrals necessary to solve for the optimal decision is linear in the number of frontier nodes. Our k-best algorithm is based on the observation that all the frontier nodes do not contribute equally to the expected minimum cost of a path through a child of the root node. Instead, the minimum-cost frontier node contributes the most, followed by the second-lowest-cost frontier node, etc. The k-best algorithm estimates the expected-minimum-cost-path value of a root child by considering only the k minimum-cost frontier nodes.

Our experimental results show that the k-best algorithm is able to outperform MINIMIN when both algorithms can search to the same depth. Unfortunately, the k-best algorithm must generate many more nodes than MINIMIN in order to find the k-best frontier nodes under each child of the root node. This is important because our experimental results on random trees show that the amount of improvement in decision quality produced by k-best for a depth n search tree is less than the improvement MINIMIN gains by exploring to depth $n+1$. Thus, in the case where we are given a fixed number of node generations, MINIMIN will produce better decisions.

To address this efficiency problem, we make a second approximation. Our idea is to make the exploration of k-best as efficient as MINIMIN by forcing k-best to use the same search tree as MINIMIN. The one change is that k-best is allowed to keep track of the k-best frontier nodes seen during the exploration. We call this new algorithm αk-best, because a depth-first branch-and-bound exploration is performed using a single alpha bound. Our results on a set of random tree problems demonstrate that αk-best produces better quality decisions than MINIMIN, while requiring the same number of node generations on average.

1.5.6 Chapter 7: Real-Time Search - Making a Sequence of Incremental Decisions

In this chapter, we first describe our approach to the general incremental decision problem, which is to treat each decision as if it were the last incremental decision. We present experimental results on sequences of incremental decisions with a fixed-depth exploration. Our results further demonstrate a slight improvement of k-best over MINIMIN, although our results also show that MINIMIN continues to perform unexpectedly well over a wide range of problem instances.

In the second half of the chapter, we discuss a set of real-time search algorithms that result from combining depth-first branch-and-bound and best-first exploration, with MINIMIN and αk-best decision making. We present experi-

mental results that compare the average solution cost for a sequence of decisions produced by each algorithm on a set of random-tree problems, when the exploration is bounded by the number of node generations that are available for each decision. The results show that under most conditions, our real-time search algorithms that combine αk-best and either exploration method perform as well or better than MINIMIN with the same exploration method, by producing results that have lower cost on average, and by winning more of the head-to-head competitions, although the difference in performance is very slight. The best-first exploration algorithms outperform the depth-first branch-and-bound exploration algorithms when computation is measured by number of nodes generated. The final choice of exploration methods will ultimately depend on the overhead costs of both methods.

1.5.7 Chapter 8: Learning the Completion-Cost Distribution

The k-best decision-making algorithm assumes that we know the edge-cost distribution. In general, we can't depend on this information being available. In this chapter, we present a method for sampling the edge-cost distribution as the tree is searched, in order to estimate the edge-cost distribution for the decision process. We present experimental results for random-tree problems, when the edge-cost distribution is not known to the search algorithm, and demonstrate the feasibility of this approach.

1.5.8 Chapter 9: Incremental Search and Flowshop Scheduling

In Chapter 9, we provide a detailed description of the flowshop-scheduling problem, and present experimental results for randomly-generated flowshop-scheduling problems. Our results show that incremental search is the clear choice over real-time adaptations of two standard approaches to optimization problems.

1.5.9 Chapter 10: Related Work

In this chapter, we highlight the work that has influenced and is related to our research. The first half of this chapter contains a discussion of three papers that have most directly impacted our research. We first summarize Mutchler's work on allocating limited search resources [Mut86], specifically including a discussion of his random-tree model for evaluating search methods. Next we discuss Korf's presentation of the MINIMIN decision algorithm [Kor90]. The discussion of MINIMIN is important because it serves as both a benchmark and starting point for

our new incremental search algorithms. Finally, we summarize Russell and Wefald's work on DTA*, which provides a general decision-theoretic framework for real-time search.

The second half of this chapter summarizes other related work that can be divided into three areas: random trees, real-time decision making, and algorithms for scheduling problems.

1.5.10 Chapters 11 and 12: Future Work, Contributions and Conclusions

The last two chapters summarize the results of this dissertation. The future work chapter presents the set of problems uncovered but as yet not addressed. In general, these problems stem from the simplifying assumptions necessary for the analysis and development of the optimal decision method. The conclusions chapter summarizes the four main contributions of this dissertation, namely our formulation of the real-time incremental decision-making problem, an analysis of the optimal last incremental decision method, analytical and experimental results that demonstrate that MINIMIN often makes good decisions, and k-best, which is a new method for making real-time decisions. Our results show that although it is possible to slightly improve upon the performance of MINIMIN, the amount of improvement is usually small. In addition, MINIMIN is efficient and easy to implement. Thus, for a wide range of problem instances, MINIMIN is a reasonable method for making real-time search decisions.

CHAPTER 2

A Problem-Space Formulation of Real-Time Decision Making

In this chapter, we present a problem-space formulation of real-time decision-making. The basic idea is to model real-time decision making as exploring and evaluating nodes in a decision tree. In the first section, we motivate this approach by developing a problem-space representation of the flowshop scheduling problem. In the second section, we introduce the general random-tree model and its associated search problem, including a discussion of how a random tree can model the error in a heuristic search. This random-tree problem-space model is used in the rest of this dissertation to analyze and evaluate the performance of real-time search algorithms. In the next two sections, we motivate why we want the random trees to be reproducible and summarize a method to do this, which is presented in detail in Appendix A. In the fifth section, we present the incremental decision problem, which is a simplified version of the general incremental decision problem. The last section describes the incremental search approach to real-time decision making.

2.1 Problem-Space Formulation

Consider the factory scheduling problem we presented in the previous chapter. Recall that there were four jobs to schedule on two machines. It turns out that it is only necessary to consider permutation schedules in order to find an optimal solution to a two-machine flowshop scheduling problem [IS65]. A permutation schedule is a schedule in which the order of job processing is the same on both machines. Thus we only need to specify a single order in which jobs will be processed, and the processing order on the first machine will be the same as the processing order on the second machine.

A problem-space formulation of the flowshop scheduling problem in Figure 1.1 is shown in Figure 2.1. The nodes in the tree correspond to decision points, and the length of the edges below each node indicates the processing time on machine M_1 for the job associated with the parent node. For example the root

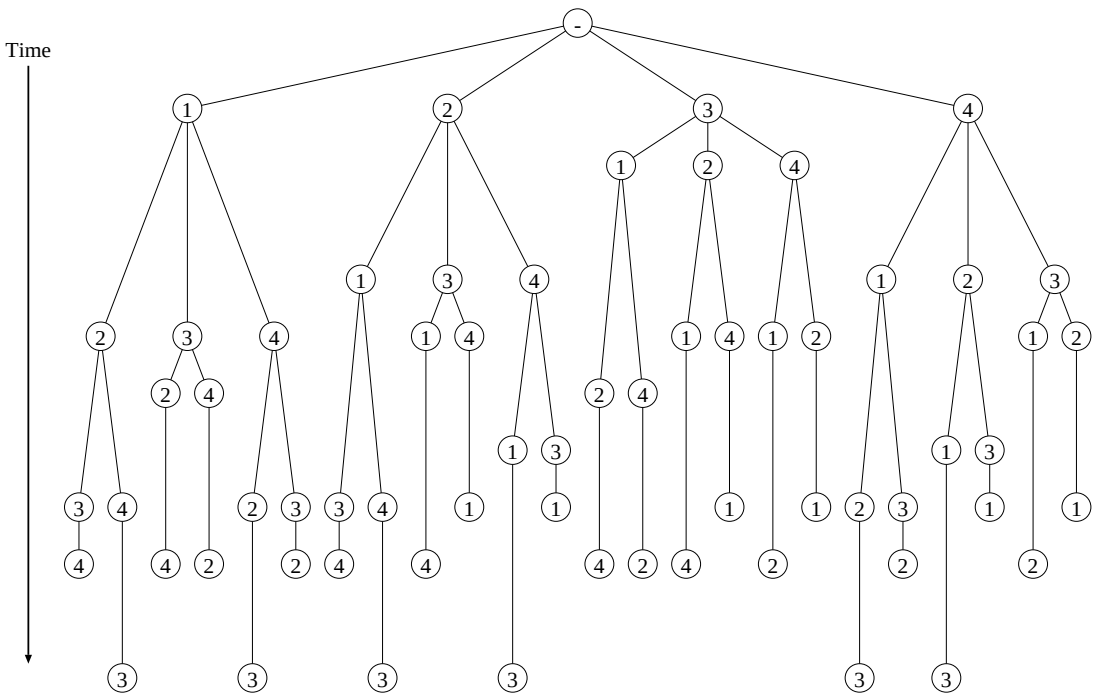


Figure 2.1: Problem-space representation for a four-job, two-machine flowshop scheduling problem.

of the tree represents the situation where no jobs have been scheduled. There are four edges emanating from the root node, one for each scheduling decision that could be made at that point, namely to schedule any one of the four jobs first. The vertical position of a node represents the time that the job associated with the node must be scheduled for processing in order to keep the first machine busy. The vertical distance between the root node and its children represents an initial time period in which to make the first scheduling decision. There are costs associated with each edge that represent either the cost of scheduling the associated job, or the difference between a heuristic estimate of the complete solution cost before the job is expanded, and a heuristic estimate of the complete solution cost after the job is expanded. In the second case, the edge cost is the unexpected or unpredicted cost of scheduling a job that we became aware of when we expanded the node. A complete solution corresponds to a path from the root to a leaf node at the bottom of the problem-space tree, and the schedule that generates this solution is the order in which the jobs appear on the solution path.

Before presenting our random-tree model, we provide the following set of definitions. A *problem space* is a representation of the problem-solving states and sets of actions that can be employed to solve a particular problem. For a real-time

decision problem, the problem space is a decision tree where the nodes correspond to decision points, the edges correspond to the possible operators or actions that apply at a given decision point, and the tree's edge costs correspond to the cost of applying the operator or action, or the penalty associated with a given decision choice. The problem-space tree is used to simulate and evaluate future sequences of decisions. For example, evaluating the cost of a schedule associated with a leaf node in Figure 2.1 is equivalent to answering the question of what the cost would be if the schedule associated with the corresponding solution path was executed. In other words, we are simulating a solution and answering the question "What would the cost be if I executed the given sequence of decisions?"

We use the term *search space* or *search tree* to denote the part of the problem space that has been generated by a search algorithm. A node *generation* is the process of creating a node to represent a problem state adjacent to the parent node along with the edge that connects the child to its parent. In addition to creating the node structure, generating a node includes evaluating the cost of the new node either by determining the cost of the operation that the edge represents or evaluating a heuristic function in order to estimate the node's cost. Similarly, a node *expansion* is the process of generating all the immediate children of a node in the search tree.

We have adopted the term *exploration* to denote the process of expanding a node for the first time. This term is used to distinguish between adding a node from the problem-space tree to the search tree (exploration) and traversing a node that already exists in the search tree (node expansion).

We define *decision making* to be the process of choosing which one of the operators available at the current decision point to execute. This is equivalent to choosing one of the children of the search-tree root node to be the root node for the next incremental search.

To further distinguish between the search tree and problem-space tree, we define *leaf nodes* to be nodes at the bottom of the problem-space tree, and *frontier nodes* to be nodes at the bottom of the search tree. Alternatively, we can think of frontier nodes as search-tree nodes that have been generated but not yet further explored.

The real-time constraint for these decision problems is modeled as a finite number of node generations per decision. This directly affects the size of the search tree that is available to the decision maker. There are several options for limiting the number of available node generations. One is to allow the problem solver to have a constant number (k) of node generations for each decision. Another is to limit the depth of the search tree with respect to the current deci-

sion node. A third is to allow the number of available node generations to be a more general function of the problem description. For example, in the flowshop scheduling problem, the cost of the solution also determines to some extent the amount of time available to explore the problem space.

In the rest of the chapter, we will present and discuss a random-tree model that allows us to control the characteristics of the problem space (*e.g.,* edge-cost distribution, tree depth, average number of children), and also to efficiently generate random problem instances. This makes it easier to separate the solution from the problem, and also makes it possible to develop a decision-theory based algorithm, given that we know the parameters necessary to model the way that the problem space is generated.

2.2 General Random-Tree Problem Description

For this research, we have focussed on an abstract real-time decision-making problem based on an incremental random-tree model. The problem space is a uniform-depth search tree with branching factors drawn independently from identical distributions. Each node of the tree corresponds to a decision point, and the arcs emanating from a node correspond to the choices (operators or actions) available for the decision at that node. Each edge has an associated random value chosen independently from an identical distribution (*e.g.*, uniform over $[0, 1]$). This value corresponds to the cost or penalty that the problem solver will incur if it chooses to execute the action associated with that edge. The cost of a node x or *node_cost(x)* is the sum of the edge costs along the path from the root to node x. An example random tree is shown in Figure 2.2. This random-tree problem can be characterized as *solution rich* since every leaf node of the tree is a solution to the problem. The objective of the problem solver is to traverse a lowest-cost path, from the root to a leaf node, given the computation-limited information available for each decision.

One way to view the random tree is as a problem-space formulation of an optimization problem. The root of the tree is the initial problem, and the children of the root are the problems that result after the application of a single operator. The problem is thus to find a sequence of operators that leads to a minimum-cost solution. In this interpretation, the edge costs correspond to the costs of applying the operators.

Alternatively, we can view the random tree as a heuristic search tree. If the heuristic function is admissible (never overestimates the actual cost to a leaf node [Pea84]) and consistent (the estimated complete solution cost of a parent node is

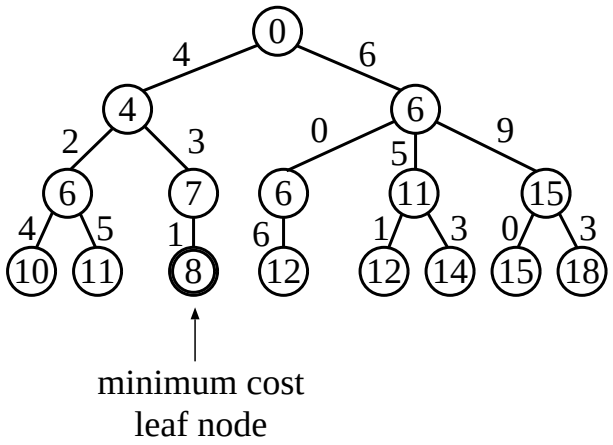


Figure 2.2: Example of a random tree problem.

never greater than the estimated complete solution cost of a child node [Pea84]), then the edge costs are equal to the difference between adjacent node costs in the random-tree model. From the perspective of heuristic search, edge costs can be viewed as the improvement in heuristic estimate that results when a node is expanded. Alternatively, the edge costs can be viewed as an error in the heuristic function's ability to accurately predict what the estimate of the solution cost will be at the next level of the search tree.

The main advantage of using this random-tree model is that it can be used to analyze and experimentally compare the average-case performance of search algorithms under many different problem conditions. Because of the way we have specified the general random-tree model, the conditions of an experiment to compare a pair of search algorithms, namely edge-cost distribution, branching factor distribution and search depth, are all under the control of the experimenter.

2.3 An Argument for Reproducible Random Trees

Given that we have decided to use random trees as a testbed for our search algorithms, the question is how to generate these random trees. The main task in building a random tree is generating the branching factors and edge costs. Perhaps the easiest way to generate the branching factors and edge costs is to just call a random number generator every time a random value is needed during the search process. For example, every time a node is expanded, we could first call a random number generator to determine the branching factor b, and then call the random number generator again b times, once for each edge cost. This

on-demand method [KPZ94] works fine if the tree will only be searched once and the same node is never revisited by the search process.

There are two potential problems with this approach, however, both of which stem from the fact that we want to be able to generate the same random tree or subtree more than once. The first problem is that a linear-space search algorithm (*e.g.*, iterative-deepening [Kor85] or recursive best-first search [Kor93]) usually regenerates part of the random tree, and expects a node and its subtree to look the same when it is revisited. Of course, we could solve this problem by just storing the problem-space tree in memory as it is generated, but this severely limits the size of the problems that can be considered.

Another argument for being able to reproduce the same random tree is to support head-to-head comparison of different search algorithms. In addition, we expect results comparing average-case performance of different search algorithms to be more reliable (*i.e.,* require less trials for reliable comparison) if the performance is measured on the same set of random trees. Since it is likely that different search algorithms will generate the search tree in different orders and will also search different parts of the search space, it makes sense to have a systematic way to generate the random values that is independent of the order in which the tree is generated.

2.4 How Should We Generate Random Trees?

We will now describe a breadth-first method of systematically generating random search tree values, and then describe an efficient method for generating these random tree values on-the-fly.

Assume for now that there is sufficient space to store the random tree. One commonly used method for storing a uniform branching factor tree in memory is to place the nodes in an array in breadth-first order (*i.e.*, the order in which the nodes would be generated by a breadth-first search). An example is shown in Figure 2.3. In breadth-first ordering, the breadth-first index (and array location) of the first child is equal to one greater than the parent's breadth-first index times the branching factor. The breadth-first index of each additional child is just one greater than the breadth-first index of the sibling immediately to its left. If the branching factor of the search tree is a fixed constant, then this is an efficient method for storing the tree and allows us to find a node easily given its node number. If there is sufficient memory to store the tree, we can generate the edge costs by calling a random number generator once for each node in the tree, store this value with the node, and then refer to the stored tree each time a

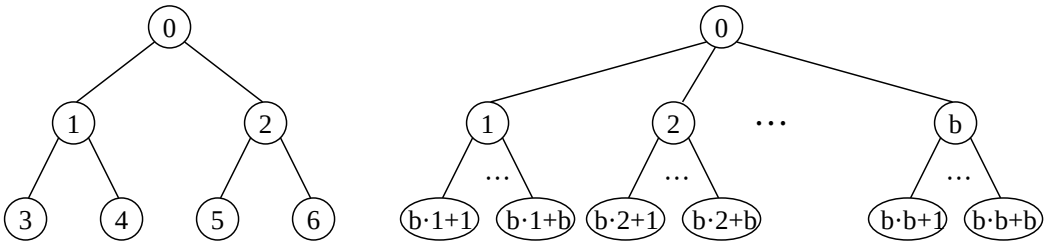


Figure 2.3: Example random trees: (a) b=2, and (b) branching factor = b.

node is (re)generated. If the branching factor is not constant, but the maximum branching factor is known, then we can store the tree much less efficiently in the same sort of array by assuming that the branching factor is a constant equal to the maximum branching factor plus one, and storing a random value for the branching factor with each node as well. This approach gives us a systematic and reproducible way to generate random trees. The remaining problem is the space needed to store the tree.

In order to reproducibly generate a random tree without storing it, we first make the following observation. If we know the seed used to initialize the random number generator before the random tree was first generated, then the random value associated with the k^{th} breadth-first index can be regenerated by first initializing the random number generator with the initial seed, and then calling the random number generator k times. As might be expected, this approach is very inefficient.

To address the problem of efficiently generating the random number associated with a given breadth-first index, we developed a method for efficiently finding the k^{th} random seed that would occur after k calls to the *rand()* function in the standard C library distribution [KR88]. The approach is based on analyzing what operations the *rand*() function performs and then solving for an equation that describes the k^{th} random seed as a function of the initial random seed and k. The result is a subroutine called $NEXTRAND$ that takes a random seed and the number of calls to the random number generator, and returns the random seed that would result if *rand*() were called k times. Our implementation of $NEXTRAND$ produces the k^{th} random seed in time that is proportional to the number of bits m used to represent an integer in the computer being used (typically 32 bits).

The general idea of this approach is to keep track of the random seed and breadth-first index of the frontier nodes in the tree. When a frontier node n is expanded, $NEXTRAND$ is called with the random seed of n and the difference

between the breadth-first index of n and its first child $(b \cdot n + 1)$. The random seed for each additional child can be easily computed using the random seed of the sibling to its left. Thus expanding a node (*i.e.*, generating all children) is almost as easy from a computational standpoint as generating only one child. The details of this approach are presented along with the C-code in Appendix A.

2.5 The Incremental Decision Problem

In this section, we present the general incremental decision problem through a series of definitions, starting with the optimal decision problem. The optimal decision problem is simply defined as the problem of finding the first step along a minimum-cost path from the current root node to a leaf node.

Definition 1 *Given a completely explored problem-space tree (i.e., all frontier nodes are leaf nodes), the* optimal decision problem *is to choose a child of the current root node that is on a minimum-cost path from the current root node to a leaf node.*

If each incremental decision is an optimal decision along a path from the root node to a leaf node, then the result is a *minimum-cost path.* Finding the optimal decision at any given point may require that we consider all leaf nodes in the problem-space tree. Note that this definition of an optimal decision is with respect to the complete problem-space tree, whereas the optimality of an incremental decision algorithm is with respect to the information available when the incremental decision is made.

Theorem 1 *Any algorithm that is guaranteed to make an* optimal decision *will in the worst case have to generate every node in the problem-space tree.*

Proof: This theorem is based on the observation that we can construct a problem-space tree where, no matter how it is explored, if we don't generate all of the leaf nodes, then we cannot guarantee that an incremental decision is optimal. Consider a binary decision tree such as the example tree in Figure 2.4, where all edge costs are zero except for the leaf-node edge costs, which all have a unique cost. In this case the minimum-cost path will consist of some number of zero-cost edges followed by the minimum-cost edge to a leaf node. Since all the interior-edge costs are zero, there is no way to prune the tree without possibly pruning the minimum-cost leaf node. Suppose that there exists an optimal decision algorithm, A_1, that makes optimal decisions without generating all of the frontier nodes. The

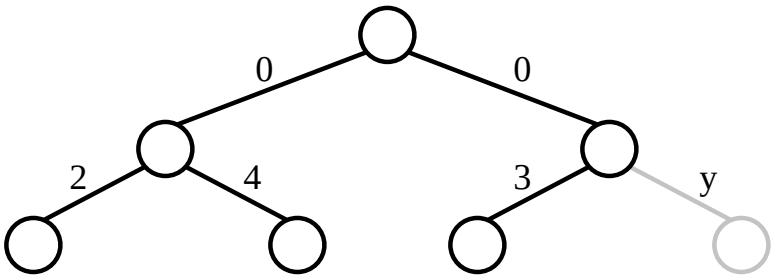


Figure 2.4: Example binary tree with all but one leaf node generated.

worst-case result comes from the fact that, given the set of leaf nodes generated by A_1, it is possible to assign edge costs to the leaf nodes so that the optimal decision depends on the leaf nodes that weren't generated. For example, suppose that algorithm A_1 generates all but one of the leaf nodes as shown in the figure. The gray edge to the unexplored leaf node has cost y. In this case, A_1 must choose between the left and right child of the root, without knowing the value of y. If A_1 chooses the left child, then we can set y equal to zero. If A_1 chooses the right child, then we can set y equal to three. In both cases, A_1 makes the wrong decision, thus finding the minimum-cost path and the optimal incremental decision can require us to generate every leaf node of the problem-space tree. □

One case where we won't have to find the shortest path to make an optimal decision is when the edge costs can be infinite. In this case, if there are two children of the root and one child's edge costs is infinite, then an optimal decision is to move to the other child. Also, if both children have infinite edge costs, then either child is an optimal decision. Additional exploration shortcuts can be taken when the edge costs are non-negative, in which case it is often possible to prune the search space to reduce the average number of nodes generated. Even with pruning, in the worst case we may still have to generate all of the leaf nodes in order to make an optimal incremental search decision.

The problem of finding a minimum-cost path in a random tree has been previously examined by [KP83, McD90b, MP91, ZK95, KPZ94]. Their results include the observation that there is no known closed form for the distribution of minimum-cost paths in a general random tree. In addition, this work has identified and characterized a phase transition in the average-case complexity of finding the minimum-cost path in random trees. One additional result is that, under certain conditions, any algorithm for finding a solution path with cost that is within a constant of optimal must take time that is exponential in the search depth on average.

David Mutchler [Mut86] has looked at a similar problem, where his task was to decide which nodes to expand (where expanding a node consists of learning the edge costs to its children) in order to minimize the cost of a traversed path. Since Mutchler was interested in allocating a limited number of node expansions, a solution path for his problem consisted of a specific path from the root node to a frontier node of the search tree, followed by an arbitrarily generated path from the frontier node to a leaf node of the problem-space tree. Once the node expansions have been performed, Mutchler's decision problem is simply to choose the path to traverse that has minimum expected cost.

Definition 2 *Given a partially explored problem-space tree,* Mutchler's decision problem *is to choose a root-to-leaf path through a frontier node such that the cost of the chosen path to that frontier node, plus the expected cost of the path from that frontier node to a leaf node using some predetermined path-completion method, is minimized on average.*

Mutchler assumed that the remainder of the path below a frontier node was generated arbitrarily (*i.e.,* always choose the left child). For this completion method, the expected cost of a complete path is simply the frontier-node cost, plus the expected cost of a single arbitrary decision times the number of such decisions from the frontier node to the leaves of the problem-space tree. The expected cost of a single arbitrary decision is calculated using the branching factor and the distribution of edge costs. For example, if the expected cost of a single arbitrary decision in Figure 2.5 is 5, then the expected cost of the complete solution path shown can be calculated as follows.

$$\begin{aligned} E(\text{solution path cost}) &= \text{node cost(frontier node)} \\ &\quad + \text{remaining depth} \cdot E(\text{arbitrary decision}) \\ &= 8 + 3 \cdot 5 = 23 \end{aligned}$$

Note that when a tree is expanded to a uniform depth, then a minimum-cost frontier node is an optimal Mutchler decision. In this case, Mutchler's decision algorithm makes the same decisions as MINIMIN, which we will discuss further in Chapter 5.

We next define the last incremental decision problem, which is one step removed from the optimal decision problem, and an extension of Mutchler's decision problem. Basically, the last incremental decision is a choice between a set of subtrees in which a series of optimal decisions will be made in the future.

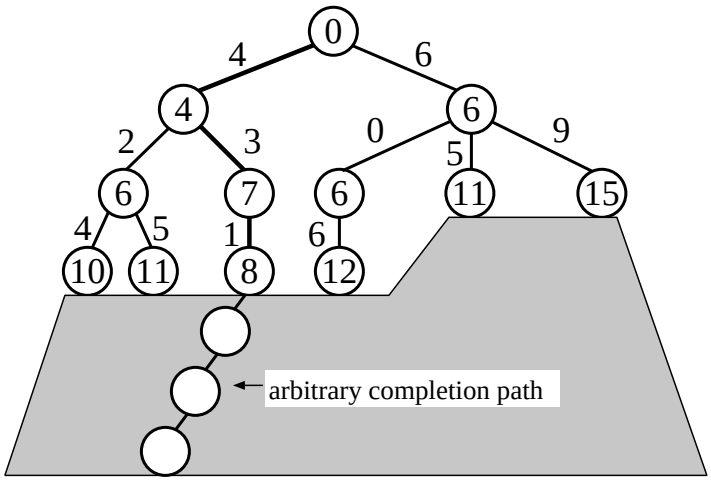


Figure 2.5: Example of Mutchler's decision problem.

Definition 3 *Given a partially explored problem-space tree that will be completely explored once a single decision choice is made, the* last incremental decision problem *is to choose a child of the current root node such that the edge cost to the child, plus the expected cost of the optimal path that will subsequently be traversed from that child to a leaf node, is minimized on average.*

Since we know that the remainder of the tree will be explored after the last incremental decision, the distribution of minimum-cost paths that will be traversed below a given child of the root node is the same as the distribution of minimum-cost paths in the tree below that child, independent of the decision and exploration methods. This means that we don't have to worry about the effect of the incremental decision and exploration methods on future decisions.

The last incremental decision problem can be extended to more than one incremental decision as follows.

Definition 4 *Given a partially explored problem-space tree that will be completely explored after n incremental decisions are made, the* n^{th}-to-last incremental decision problem *is to choose a child of the current root node such that the cost of the edge to that child, plus the expected cost of the path that will subsequently be traversed by a sequence of* $n-1$ *incremental decisions, followed by another sequence of optimal decisions, is minimized on average.*

The main difference between the n^{th} incremental decision problem and the general incremental decision problem is that for the general incremental decision problem we don't know when or if the bottom of the tree will be discovered.

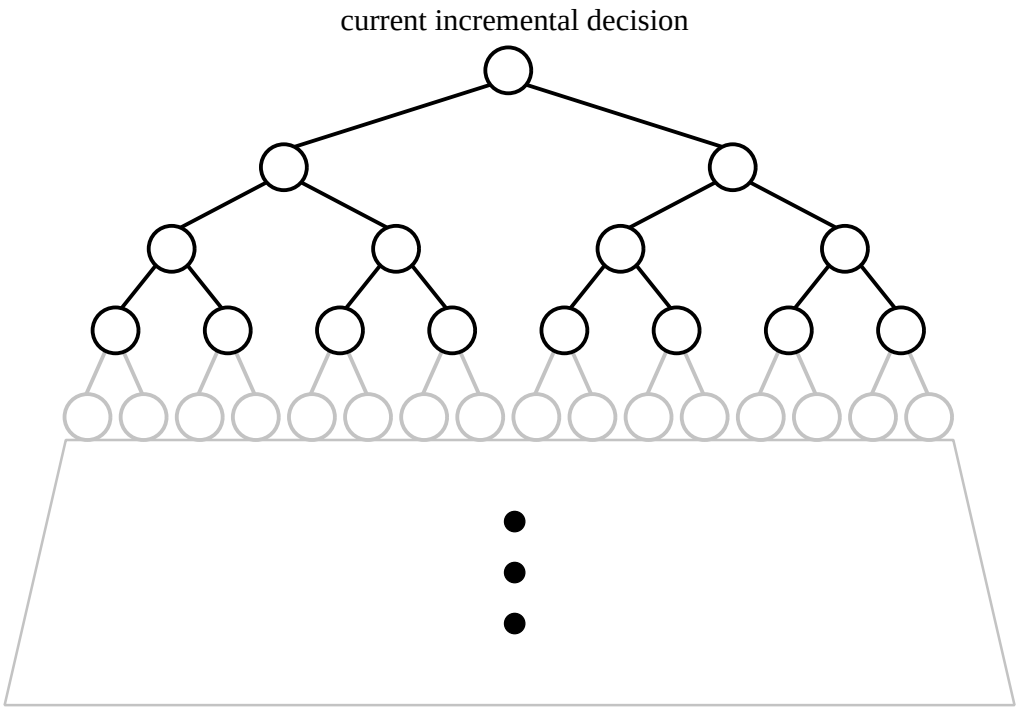


Figure 2.6: Example of the general incremental decision problem.

Definition 5 *Given a partially explored problem-space tree, the* general incremental decision problem *is to choose a child of the current root node such that the average cost of the root-to-leaf path through that child, which consists of the edge to that child plus a sequence of general incremental decisions starting from the child node, perhaps followed by another sequence of optimal decisions, is minimized.*

The search tree in Figure 2.6 illustrates a general incremental decision problem. Note that once a decision is made, a significant portion of the problem-space tree (of perhaps unknown size) is still unexplored. The general incremental decision problem has the following characteristics. Exploration has stopped, so a decision must be made based only on the current search tree. The current search tree does not contain any leaf nodes of the problem space tree, nor do we know when a leaf node will be discovered in the future, or if a leaf node will ever be discovered (*i.e.,* a dynamic problem where the number of decisions is potentially infinite). Once a decision choice is made, the next phase of exploration is performed, resulting in a new incremental decision problem.

We now have a definition for the general incremental decision problem. In Chapter 4, we will discuss how to optimally solve these incremental decision

problems. In the next section, we present the incremental search approach to real-time decision making.

2.6 The Incremental Search Approach

In this section, we present the incremental search approach to making real-time decisions. We first describe the general incremental search approach, and then discuss how incremental search maps to the three subtasks of real-time decision making.

2.6.1 What is Incremental Search?

Our incremental search approach to the single-agent case (*e.g.,* scheduling and planning problems) is to perform a lookahead search on a problem space representation of the planning task (*i.e.*, a search tree), and interrupt the lookahead search when time runs out. At this point the current best decision is executed, and the problem space is updated to reflect the decision. The search process is then repeated for the next decision. We call this approach incremental search because the sequence of decisions is generated incrementally over time as needed.

An example of the incremental search process is shown in Figure 2.7. At the top of the figure, the triangle represents the exploration for the first decision. After the first decision, the middle of the figure shows the exploration for the second decision, which overlaps the exploration from the first decision. The bottom of the figure shows the exploration for the third decision. This process of exploration followed by a decision is continued until the complete solution path is generated. The size of the exploration triangle represents the amount of time spent exploring the problem space tree.

Incremental heuristic search differs from traditional heuristic search in that traditional heuristic search normally generates the complete solution as its output (often in exponential time), whereas incremental heuristic search generates the solution one step at a time. Incremental heuristic search algorithms are well adapted to the problem of real-time decision making, especially when time constraints are on individual decisions, rather than on the whole sequence of decisions.

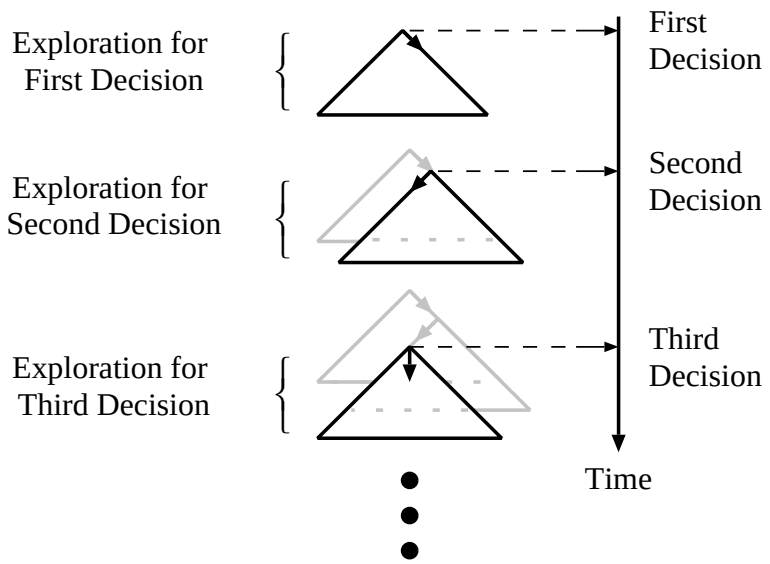


Figure 2.7: Example of the general incremental search process.

2.6.2 The Three Sub-Tasks of Incremental Search

In Chapter 1, we discussed the three subtasks of real-time decision making, namely what information to gather, when to stop gathering information to make a decision, and what decision to make given an incomplete set of information. For the incremental search approach, these three subtasks correspond to: what nodes should be explored, when should we stop exploring the problem space and make a decision, which decision is the best given a partially explored problem-space tree.

2.6.2.1 Exploration: What nodes to explore?

Given the fact that we can only explore a subset of the problem-space nodes necessary to make an optimal decision, the natural question is how should we choose the subset of node explorations in order to best support our objective of traversing a root-to-leaf path with lowest expected cost. Our answer to this question is the topic of Chapter 3.

2.6.2.2 Stopping: When to Stop Exploring?

The optimal solution to the stopping problem requires us to understand both the exploration and decision making problems. For this reason, we have avoided the stopping problem in our research by assuming that there is a strict deadline on

the exploration. We do, however, briefly discuss the general issues related to the stopping problem in the last part of Chapter 3.

2.6.2.3 Decision Making: What Action to Execute?

Decision making is the problem of choosing an action or operation to perform, given a partially explored problem space, and perhaps general background information, such as the depth of the tree, the distribution of edge costs, and the exploration method used. Our approach to decision making is discussed in detail in Chapter 5.

CHAPTER 3

Exploring the Problem Space and Deciding When to Stop

In this chapter, we discuss both the exploration problem and the stopping problem. We first define the exploration problem. Then, we describe Mutchler's work on a simpler version of the exploration problem, and use his results to argue that our more general exploration problem is even more difficult. Given the difficulty and complexity of optimally solving the general exploration problem, we have adopted simple heuristic exploration methods, which we present in the third section. In the fourth section, we briefly discuss the stopping problem in the context of incremental search. The last section contains a summary of this chapter.

3.1 The Exploration Problem

The exploration problem is the task of determining where to spend a limited amount of computational resources in support of a sequential decision process. The exploration problem is defined as follows.

Definition 6 *Given that only k nodes of the search tree can be expanded before a decision must be made, the* exploration problem *is to determine which set of nodes should be explored in order to produce the best decision quality on average.*

The quality of a decision is measured by the average or expected solution-path cost that results from making that decision, compared to some other decision. Lower expected solution-path costs correspond to better quality decisions.

Note that this definition of the exploration problem relies on our assumption that there is a strict deadline. Alternatively, we could define the exploration problem based on some different stopping criterion, but this particular definition was chosen to simplify the analysis. This formulation of the problem also ignores the cost of the metareasoning [Hor90], namely the cost of figuring out which nodes to explore. If we include the cost of the metareasoning computation, the problem becomes: *given k time units before a decision must be made, which computations*

should be performed in order to produce the best quality decision on average? For this discussion, we ignore the cost of the metareasoning, and use the issue of overall efficiency to argue against any algorithm that requires more than a small amount of computation for metareasoning.

3.2 Background on Exploration

Our definition of the exploration problem can be viewed as an extension of the problem investigated by Mutchler in [Mut86]. In his work, Mutchler looked at the following problem: *Given that we can learn the value of only k edge costs in a random tree, if the objective is to find a least-cost root-to-leaf path, then which edge costs should we learn?* Note that learning an edge cost can be viewed as equivalent to generating a node in the search tree. If k is large enough to generate the complete search tree, then the problem can be solved optimally using an existing traditional search algorithm. The difficulty occurs when k is much less than the number of node generations needed to make an optimal decision.

Mutchler focussed on a binary tree that is complete to depth N, with edge costs that are independently set to 1 with probability p, or 0 with probability $(1-p)$. This tree is a special case of the general random-tree model described in Chapter 2. The cost of a node is equal to the sum of the edge costs along the path to the root node. Once the k nodes are generated, a complete solution path is chosen based on the expected-total-path-cost as follows. The expected-total-path-cost of a frontier node is simply the actual cost of the path from the root to that node, plus the expected cost of an arbitrary path from the frontier node to a leaf node. Given that we can only generate k nodes before choosing a complete path to a leaf node, Mutchler explored the question of which nodes should be generated in order to minimize the expected cost of the leaf node selected.

To illustrate the difficulty of this problem, Mutchler presented the following simple example shown in Figure 3.1. In this example, the tree is binary (each node has two children) with a depth of 4 (only two levels of the tree are shown in Figure 3.1). A depth of 4 was chosen because it is the smallest tree depth for which one more node generation is guaranteed not to discover a leaf node of the tree. The node labeled A has already been generated, revealing an edge of cost 1. The question in this situation is: which node should be generated next assuming that only one more arc can be examined before a complete path from the root to a leaf node at depth 4 must be chosen? Since the remainder of the tree has not been explored, it is assumed that the edges at the bottom of the solution path are chosen arbitrarily (*e.g.*, at random). The problem solver knows

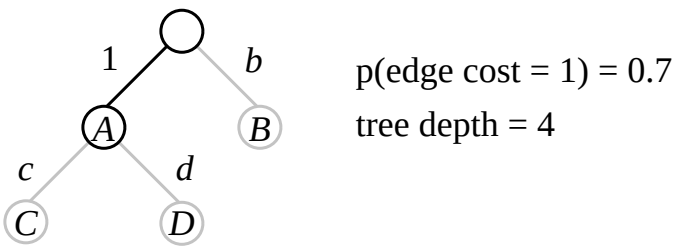


Figure 3.1: Mutchler's very limited resource search problem.

that the probability that an edge has cost 1 is 0.7 (and an edge has cost 0 with probability 0.3), and that the depth of the tree is 4. The reader is encouraged to stop and decide which node should be generated under these circumstances before continuing.

The intuitive answer is to generate node B to learn the cost of edge b. This is the greedy exploration choice based on the expected cost of a path to depth 4, because the expected cost of a path through node B is lower than the expected cost through either C or D. The correct answer is either node C or D, since the expected cost of a decision after generating node B is greater than the expected cost of a decision after generating either node C or D. The intuition behind this answer is that if we generate node B, then the expected value of any leaf node below node B is at least as good as the expected value of any leaf node below node A. If $b = 0$, then we will choose a leaf node below node B with an expected cost of $0 + 3p$. If instead $b = 1$, then an arbitrary leaf node below node B will have the same expected cost as any leaf node below node A, namely $1 + 3p$. In either case, choosing an arbitrary leaf node below node B is an optimal decision strategy, thus learning the cost of b can have no effect on the decision.

Suppose however that we generate node C instead, and $c = 0$. Because zero-cost edges are relatively rare, we will choose a leaf node below node C with an expected cost of $1 + 2p$. If instead $c = 1$, we can still choose a leaf node under node B with expected cost $4p$.

Let $E(gen\ B)$ and $E(gen\ C)$ denote the expected solution cost if we generate node B and node C respectively. The expected solution cost if we generate node B is the sum of the expected solution cost if $b = 1$ and the expected solution cost if $b = 0$, weighted by their probabilities.

$$E(gen\ B) \quad = \quad p * (1 + 3p) + (1 - p) * 3p = 4p = 2.8$$

Note that the expected solution cost after generating node B is equal to the expected solution cost of an arbitrary leaf node below B, implying that there is no informational value to knowing the cost of b. Similarly, the expected solution

cost if we generate node C is the sum of the expected solution cost if $c = 1$ and the expected solution cost if $c = 0$, again weighted by their probabilities.

$$\begin{aligned} E(gen\ C) &= p * (4p) + (1-p) * \min(1+2p, 4p) \\ &= (1.96 + .3 * \min(2.4, 2.8) \\ &= 2.68 \end{aligned}$$

Thus, for this value of p, the decision to generate node C will result in a lower expected solution cost.

This example relies on the fact that zero-cost edges occur less than half of the time. In his paper, Mutchler also showed that if nodes are generated in increasing order of the expected cost of a solution path below that node (*i.e.*, a greedy exploration method), then the set of nodes generated is optimal for all node generations when the probability that an edge has cost one is less than or equal to 0.5 ($p \leq 0.5$). This comes from observing the conditions under which a greedy exploration method can be non-optimal, namely when $p \geq (1-p)$. He further extended the result as follows. If $0.5 \leq p \leq \frac{\sqrt{5}-1}{2} \approx 0.618$ then the greedy exploration method is optimal up to the last exploration decision. Furthermore, if $0.618 \leq p \leq 0.682$, then the greedy exploration method is optimal except for the last two exploration choices, which are not guaranteed to be optimal. The values 0.618 and 0.682 come from solving the equations $p^2 = 1-p$ and $p^3 = 1-p$, respectively. He further conjectures that for any fixed value of p, the greedy exploration method will be optimal as long as the number of node generations is greater than k such that $p^k = 1-p$ or $k = log_p(1-p)$. Note that these results assume that there are not sufficient node generations to reach the bottom of the tree.

The general conclusion from Mutchler's work is that the exploration problem is difficult. Others have also argued that the general exploration problem is difficult (*e.g.,* [RW91]), and that it is necessary to make simplifying assumptions in order to address the problem. For example, a common assumption is that the value of a computation can be evaluated in isolation from other computations (a myopic assumption [Pea88]). This is clearly suboptimal, because one the reasons for expanding a node is that it makes other node expansions possible. Nonetheless, this assumption is reasonable, when the main value of a node expansion is being able to see and evaluate its children. This assumption is also necessary, when the computation to estimate the true value of expanding a node exceeds the cost of the node expansion itself.

The optimal solution to the exploration problem as we have defined it is more difficult than Mutchler's exploration problem for two reasons. One is that

Mutchler assumes that the output after the exploration is performed is a complete path to a leaf node. Our problem is sequential, so the output is one step along a path to a leaf node. This makes it more difficult to assess the expected value of a decision, as we will discuss further in the next chapter. The second reason is that since we are interested in a sequential decision process, some information from one decision is available for the next decision. In this case, the value of an exploration must include some measure of its value for future decisions as well as for the current decision. For these reasons, and also to allow us to focus our attention on the decision-making problem, we have limited ourselves to heuristic exploration methods.

3.3 Heuristic Exploration Methods

In this section, we describe the two basic heuristic methods that we have considered for choosing which nodes to expand: depth-first branch-and-bound and best-first search. These exploration strategies were chosen because they are general purpose and can be implemented efficiently.

3.3.1 Depth-First Branch-and-Bound Exploration

Depth-first branch-and-bound is a depth-first exploration method that uses a branch-and-bound technique to prune the search space. Depth-first search always expands next the deepest unexpanded node. Figure 3.2 shows an example of depth-first search, where the nodes are labeled with the order that they would be expanded by a depth-first exploration (*e.g.,* first, second, *etc.*), and the numbers in the nodes are the node costs. In general, depth-first search expands the children of a node before it expands any of its unexplored siblings. One big advantage of depth-first search is that it can be used to systematically enumerate every state in the problem space, while using memory that is linear in the depth of the search tree. This description of depth-first search assumes that all children of a node are generated (*i.e.,* a node is expanded) before any of a node's grandchildren are generated. The advantage of completely expanding a node before searching deeper is that the node costs of the children can be used to order the expansion of the children (*node ordering*), which can significantly increase the number of nodes that can be pruned.

When the node costs are monotonic non-decreasing along any path from the root node to a frontier node, then it is possible and often desirable to use a branch-and-bound technique to prune the depth-first search space. The general idea of

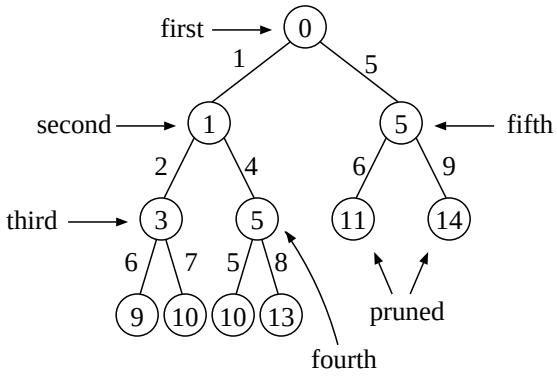


Figure 3.2: Example of depth-first exploration.

branch-and-bound is to prune the exploration process when a node cannot have an impact on the decision. One example of pruning occurs when we are trying to find a minimum-cost path from the root to a leaf node in a search tree. In this case, the first leaf node discovered is an upper bound on the minimum path cost. Pruning is based on the observation that any interior (non-leaf) node whose path cost is greater than or equal to the upper bound cannot have a node with lower path cost below it, and can thus be discarded (pruned) without affecting the end result of the search. The tree in Figure 3.2 is an example of shortest-path pruning.

Depth-first branch-and-bound (DFBnB) is a specific branch-and-bound search algorithm that operates by exploring the tree in a depth-first order, using the best leaf node cost found so far to prune the interior nodes. The two main advantages of DFBnB are that it is often able to find a minimum-cost frontier node using much less computation than a brute-force breadth-first or pure depth-first exploration, and that it only requires space that is linear in the search depth. One useful optimization for DFBnB, called node ordering, is to expand the children of a node in increasing order of cost. The idea is that we are likely to find a lower-cost leaf node under a lower-cost child node.

There are several different ways to adapt DFBnB for the real-time constraint. The first, and perhaps most obvious way is called truncated branch-and-bound (tBnB) [Iba76]. The idea is to perform DFBnB until the available time expires (or the available node generations are used up), and then return the best leaf node found so far. This works well for the complete solution problem when there is sufficient time to explore a reasonable number of leaf nodes, but performs poorly on the incremental decision problem. This is due to the fact that the initial set of node expansions is spent exclusively under only one child of the current decision. If the time expires before a second child's subtree can be explored, then the

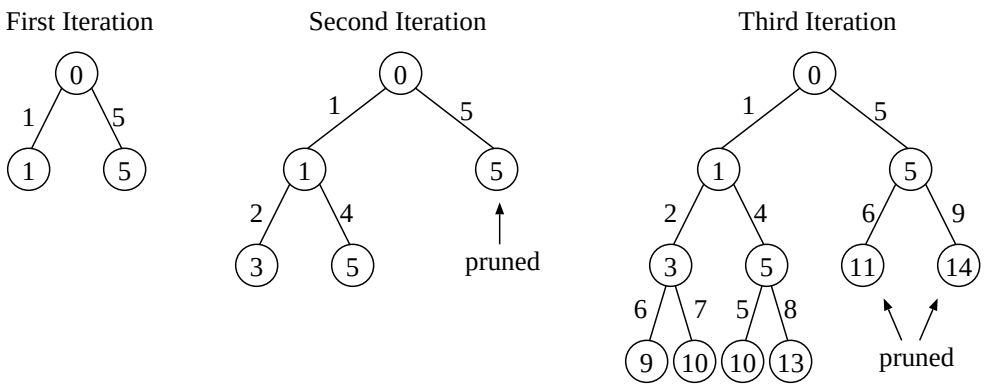


Figure 3.3: Example of iterative-deepening depth-first branch-and-bound exploration.

decision at the root level is be based on a very biased search tree. This problem will be discussed further in Chapter 7.

The approach that we have used in our implementations is to perform depth-first branch-and-bound with iterative deepening (IDBnB). The idea here is to perform a depth-bounded branch-and-bound search with increasingly larger depth bounds until the computation time expires. The result is a linear-space exploration algorithm that can find the minimum-cost frontier node at a depth that increases with the amount of time available. An example of iterative deepening BnB is shown in Figure 3.3, where the numbers in the nodes are the node costs.

One drawback of IDBnB is the overhead necessary to regenerate the previous levels of the tree in subsequent iterations. Under many circumstances, though, this regeneration overhead is small. A second drawback is that IDBnB cannot guarantee that it has found the minimum-cost frontier node at a given level of the tree until the exploration of that level is complete. This second problem is discussed further in Chapter 5, in the context of the decision making problem.

There are two ways to improve the performance of IDBnB. One is to initially calculate a lower-bound estimate of the search depth that could be explored in the available time, and start the depth bound at this value. The end result is a slightly reduced regeneration overhead in exchange for the cost of calculating the lower-bound search depth.

The other way to improve the performance of IDBnB is to use some memory to store the relevant part of the search tree from the previous iteration. This is the approach we have used in our experiments. At each interior node that is one level above the frontier nodes, we can store the minimum node cost of the frontier nodes below it, and, for each of the other interior nodes, we can store

BFS:

1. Add the root node to the OPEN list.
2. Remove the best node n from the OPEN list.
3. If n is a leaf node, then stop.
4. Else, expand n, and add its children to the OPEN list.
5. Loop to 2.

Figure 3.4: General best-first search algorithm.

the minimum of the stored values of their children. These backed-up minimum values can then be used to more efficiently order the next iteration. The idea is that the optimal frontier node at depth $d+1$ is more likely to be found below the frontier nodes at depth d that have a lower cost.

Of course, this approach requires memory that is exponential in the search depth. An alternative way to make use of the information from the previous iteration is to just keep track of the path from the root node to a minimum-cost frontier node, and explore this path first in the next iteration. The idea here is that a minimum-cost frontier node at depth $d+1$ is very likely to be found beneath a minimum-cost frontier node at depth d. This allows the next iteration to start with a good initial upper bound.

3.3.2 Best-First Exploration

Best-first search methods employ a preference relation between nodes to order the node expansions. One nice feature of best-first search methods is that we can easily interrupt the exploration when the available computations run out, and use the information gathered thus far to make a decision. Unlike depth-first branch-and-bound, there is no need to worry about completing the exploration of a level of the search tree.

The main difference between different best-first search (BFS) algorithms is the function that they use to order the search (*i.e.,* to define best). The general best-first search algorithm is shown in Figure 3.4.

3.3.2.1 BFS(node cost)

One obvious way to explore a problem-space tree is to expand nodes in increasing order of their node cost. With this approach, if there is sufficient time to choose

a leaf node of the problem-space tree for expansion, then, when the edge costs are non-negative, the path to that leaf node is a minimum-cost path. In fact, for the random-tree model that we have adopted, the node-cost search heuristic is the only admissible heuristic. A heuristic function h is admissible if $h(n) \leq h^*(n)$ for all n, where h^* is the actual value being estimated [Pea84]. In other words, an admissible heuristic function never overestimates the value being estimated. In the random-tree model, since it is possible for an unexplored path to exist that consists entirely of zero-cost edges, the node cost of a node is the best admissible estimate of the total path cost through that node. By a best admissible estimate, we mean an admissible estimate that is greater than or equal to any other admissible estimate.

3.3.2.2 BFS(estimated total cost)

An alternative way to explore a problem space tree is to expand nodes in increasing order of an estimate of their expected total path cost. Given that we know the depth of the problem space tree, the simplest way to estimate the expected total path cost of a frontier node is to first estimate the expected cost per decision of the remaining path, multiply this by the remaining number of decisions, and then add it to the node cost of the frontier node. One way to estimate the expected cost per decision on the remaining path is to use the expected cost of a single decision based on one level of the tree.

$$\hat{E}(total_cost(x)) = node_cost(x) + E(cost/decision) \cdot (tree_depth - depth(x))$$

where $\hat{E}$ is the estimated expected value. This is similar to the way that Mutchler evaluated frontier nodes in [Mut86].

3.4 The Stopping Problem

Although we have assumed strict decision deadlines in order to avoid the stopping problem, in this section we define the stopping problem and discuss the issues related to solving it. The stopping problem can be defined as follows.

Definition 7 *Given a finite number of node expansions, a set of decisions, the* stopping problem *is to determine home many nodes to expand before committing to a decision.*

From a decision-theoretic standpoint, the obvious answer to the question of when to stop exploring the problem space and commit to a decision is when

the marginal expected utility of performing the next exploration, in terms of its impact on the decision quality, is equal to its marginal cost. The *marginal utility* in this case is the added benefit to the decision maker of the information made available by one additional node expansion, and the *marginal cost* is the time needed to expand a node. This is essentially what Russell and Wefald suggested in [RW91]. There are two potential problems with this approach. One is that the time needed to calculate the marginal utility may be more than the time needed to perform the computation itself. The second is that the information needed to calculate or estimate the marginal utility may not be available. The fact that we are generating a sequence of decisions complicates the problem because we need to assess the marginal utility of a computation now, versus saving it for later decisions. This is especially true because the utility of a later computation will depend on both the decision making and exploration methods.

Thus it is necessary to make some assumptions about the marginal utility of a computation and its marginal cost in order to reasonably address the stopping question. One option is to assume that the marginal cost of a node exploration is constant, and that the marginal utility of a node exploration can be calculated independent of other exploration steps. In this case, the value of a node expansion can be approximated by the expected effect that it would have on the decision if it were the last computation performed before making a decision [RW91]. One problem with this approach is that it can easily underestimate the value of a node exploration by not considering the additive effect of several node explorations together. The net result is that it may stop exploring too soon, making move decisions based on less information than is it should.

In order to remove this issue from our analysis, we have adopted a more restrictive set of assumptions. We assume that the stopping condition is dictated by the problem description, namely that the amount of computation available is limited by the conditions of the problem being solved (*i.e.,* a strict job-scheduling deadline with no bonus for finishing the computation early).

Initially, we will adopt a more restrictive assumption of a depth limit on the exploration. This makes it easier to compare algorithms and to avoid the boundary effects at the bottom of the problem-space tree. We later relax this assumption, limiting the the number of node expansions per decision instead of the exploration depth.

In terms of a general cost function for the computation, the allowed computations are free, and any additional computations have infinite cost. Thus, this is a strict-deadline problem formulation [Hor90]. In our flowshop-scheduling example, there is no cost for the exploration performed during the time that the

current job is being processed on the first machine. Of course there is still the opportunity cost of not being able to spend the computation time evaluating some other partial schedule.

3.5 Summary

In this chapter, we have defined the exploration problem. We then summarized Mutchler's work on a simpler problem, and argued that our exploration problem is even more difficult. In general, our belief is that anything more complicated than a simple heuristic exploration method will either require information that in general is not available, or will require too much of the available computation time to be worthwhile. This is our justification for focussing our attention on depth-first branch-and-bound and best-first exploration methods.

There are two main differences between iterative-deepening depth-first branch-and-bound (IDBnB) and best-first exploration (BFS). One is that IDBnB only updates the location of the minimum-cost frontier node after a complete iteration, whereas BFS can update the location of the minimum-cost frontier node after each node expansion. Another difference is that IDBnB only requires space that is linear in the search depth, whereas BFS must store all frontier nodes. In addition, there is an overhead cost associated with maintaining the OPEN list. In spite of this overhead cost, BFS requires fewer node generations on average than IDBnB to find the minimum-cost frontier node at a given level of the search tree.

We have also defined the stopping problem in this chapter, and discussed issues related to solving it in general. In addition, we described our assumption that the deadlines on exploration are strict, which we made in order to avoid the stopping problem in our analysis and experiments. In the next chapter, we will define and present the incremental decision problem.

CHAPTER 4

Making an Optimal Incremental Decision

In this chapter, we discuss and analyze the problem of making optimal incremental decisions. In the first section, we discuss how to optimally solve the different incremental decision problems. In the second section, we develop the optimal decision method for making the last incremental decision, and argue that making an optimal last incremental decision is impractical, due the size and complexity of the equations needed to make an optimal decision.

4.1 Analysis of Incremental Decision Making

In this section, we discuss the problem of making an optimal decision for the incremental decision problems described in Chapter 2.

As shown in Chapter 2, the *optimal decision problem* can in the worst case require us to examine every leaf node in the problem-space tree. This is based on the observation that in order to find the first step on a minimum-cost path, it may be necessary to find a complete minimum-cost path as well. Since finding the minimum-cost path in a tree can require us to examine every frontier node, the worst-case complexity of the optimal decision problem is exponential in the depth of the problem-space tree. Of course, once a minimum-cost path is found, then each subsequent optimal decision can be made in constant time.

Mutchler's decision problem is to select a leaf node of the problem space given the information in the search tree, and assuming that an arbitrary completion rule will be used to generate a path from frontier nodes to leaf nodes. With this assumption, the expected cost of a root-to-leaf path through a given frontier node can be calculated as the sum of the path cost from the root to that frontier node, plus the product of the expected cost of each decision in the arbitrary completion, and the number of such decisions from the frontier node to a leaf node. Once we have calculated the expected cost of a root-to-leaf path for each frontier node, then the optimal Mutchler decision is to choose a path with minimum expected cost. In the worst case, finding a path with minimum expected cost may require us to examine every frontier node. Thus the worst-case complexity of the Mutchler

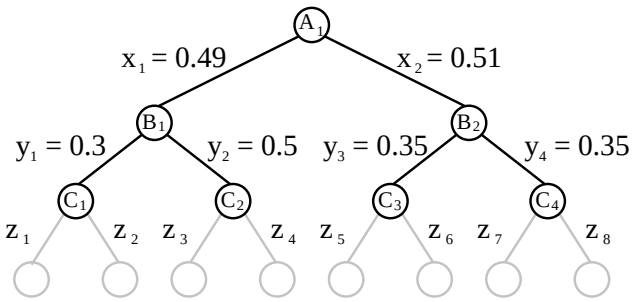


Figure 4.1: Example of the last incremental decision problem. Which node is a better decision, B_1 or B_2?

decision problem is exponential in the depth of the search tree.

The last incremental decision problem is one step removed from Mutchler's decision problem. An example of a last incremental decision problem is shown in Figure 4.1. In this figure, the black edge costs (x_1, x_2, y_1, y_2, y_3, y_4) are known. The costs of the gray edges (z_1,..., z_8) are independently chosen at random from a uniform distribution over the range $[0, 1]$. The decision task is to choose a child of the current root node so that the average cost of a complete root-to-leaf path is minimized. After this choice is made, the entire relevant subtree is explored and hence the two subsequent decisions will be optimal. The reader is encouraged to stop and decide which is the better decision under these conditions, node B_1 or node B_2? The correct answer to this question (node B_2) depends on the edge-cost distribution and known edge costs.

For the last incremental decision problem, we have to commit to one of the children of the root node, after which we will be able to see the complete subtree below the chosen child. Thus each of the remaining decisions will be optimal with respect to the chosen child's subtree. The optimal last incremental decision is a child of the root node that has the lowest expected root-to-leaf path cost. In order to calculate the expected root-to-leaf path cost of a child of the root node, we need to know the distribution of possible minimum-cost paths below each frontier node. This distribution is then combined with the frontier-node costs to determine the expected value of each child of the root. The details of this calculation are provided in the next section.

The difference between Mutchler's decision problem and the last incremental decision problem is illustrated by the tree in Figure 4.1. Since all of the frontier nodes are at the same depth, the optimal Mutchler decision is to move toward node C_1, which is also the minimum-cost frontier node. As mentioned above, the optimal last incremental decision for this example is to move to node B_2

because it has the lowest expected root-to-leaf path cost. The intuition behind this is that a decision to move to node B_1 only has two chances to find a low-cost extension of the minimum-cost frontier node (C_1), whereas a move to node B_2 has four chances to find a low-cost extension to a frontier node with cost that is near to the minimum. For this set of known edge costs and distribution of unknown edge costs, the expected cost of the four chances for extending the near-to-minimum-cost frontier nodes (z_5,z_6,z_7, and z_8) is less than the expected cost of choosing between the extension of either the minimum-cost frontier node (z_1 or z_2) or frontier node C_2 (z_3 or z_4), which has the highest frontier-node cost. Thus, in order to make an optimal last incremental decision, we need to know the distribution of root-to-leaf path costs below each child of the root node. In order to calculate this distribution, we need to know the distribution of minimum-cost paths below the frontier nodes. This distribution is needed to model the optimal decision choices that will occur after the current decision is made and the remainder of the tree is explored. The fact that the last incremental decision problem requires the distribution of minimum-cost paths below the frontier nodes, rather than just the expected cost of a path below the frontier nodes, makes the last incremental decision problem more difficult than Mutchler's decision problem. The difficulty of calculating the optimal last incremental decision will be discussed later in this chapter.

The second-to-last incremental decision problem is even more complicated than the last incremental decision problem, because the optimal second-to-last incremental decision depends on the frontier-node costs and the relative location of the frontier node costs in the search tree. For example, consider the incremental decision tree in Figure 4.2. For this tree, there are two remaining levels of the tree that have not been explored, and the unknown edge costs are either zero (with probability p) or infinite (with probability $(1-p)$). If this tree represents a last incremental decision, then the expected cost of both decisions is the same, since the two subtrees have the same set of frontier-node costs. If instead, the figure represents the second-to-last incremental decision, then there will be two incremental decisions before the leaf nodes are generated. After each decision, one additional level of the search tree will be explored. In this case, there is an obvious advantage to choosing node B_2. The intuition for B_2 being a better move is that a move to B_1 will be immediately followed by choice between C_1 and C_2, whereas a move to B_2 will be followed by a move to C_3, which is supported by two zero-cost choices (D_5 and D_6). More specifically, a move to B_1 will lead to a last incremental decision that has to choose between the zero-cost paths through D_1 and D_3 with only one additional level of information to help determine which is a better decision. Alternatively, a move to B_2 allows us to delay the choice

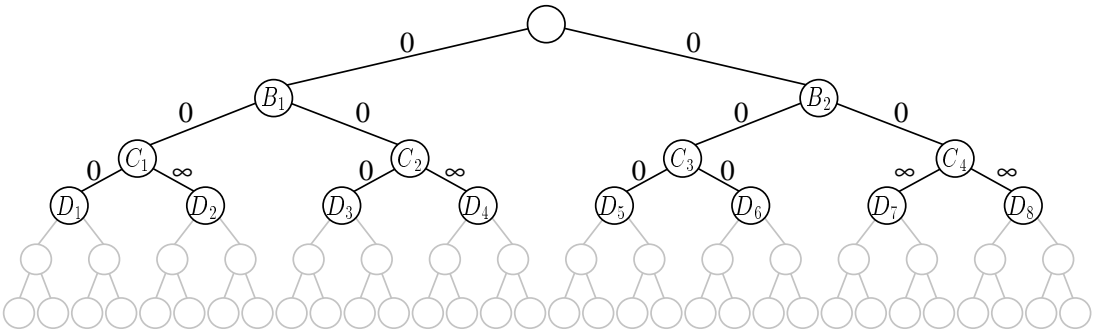


Figure 4.2: Example where the optimal last incremental decision and second-to-last incremental decision differ.

between the two zero-cost frontier nodes (D_5 and D_6) until the remainder of the problem-space tree is completely explored.

The probability of finding a zero-cost root-to-leaf path under B_1 can be calculated as follows. Let $P(abcd)$ be the probability that the edge costs directly below D_1 and D_3 have cost a, b, c, and d (*e.g.*, $P(0000)$ = probability that all the edges directly below D_1 and D_3 have cost zero). The following equations describe the probabilities of the different exploration outcomes after the first incremental decision.

$$\begin{aligned}
P(0000) &= p^4 \\
P(0001) &= P(0010) = P(0100) = P(1000) \\
&= p^3(1-p) \\
P(0011) &= P(0101) = P(0110) = P(1001) = P(1010) = P(1100) \\
&= p^2(1-p)^2 \\
P(0111) &= P(1011) = P(1101) = P(1110) \\
&= p(1-p)^3
\end{aligned}$$

After we have moved to B_1 and explored the next level of the tree, in order maximize the likelihood of complete path with zero cost, the next decision will be to move a child C_i of B_1 that has the greatest number of zero-cost frontier nodes below it. The two cases of interest are when C_i has either one or two zero-cost frontier nodes below it. If C_i has only one zero-cost frontier node below it, then the probability of a zero-cost complete path is one minus the probability that all paths below this frontier node have infinite cost.

$$P(\text{at least one zero-cost edge} \mid \text{one zero-cost frontier node}) = 1 - (1-p)^2$$

If C_i has two zero-cost frontiers nodes below it, the probability of a zero-cost

complete path can be similarly expressed.

$$P(\text{at least one zero-cost edge} \mid \text{two zero-cost frontier node}) = 1 - (1-p)^4$$

If C_i does not have any zero-cost frontier nodes below it, then there is not a zero-cost solution path.

We can now calculate the probability of a zero-cost path below B_1 by combining the equations for $P(abcd)$ with the probability of a zero-cost path in the C_i tree, assuming that C_i's subtree has the greatest number of zero-cost frontier nodes.

$$\begin{aligned} &P(\text{zero-cost root-to-leaf-path} \mid B_1) = \\ &\quad (p^4 + 4p^3(1-p) + 2p^2(1-p)^2))(1-(1-p)^4) \\ &\quad +(4p^2(1-p)^2 + 4p(1-p)^3)(1-(1-p)^2) \end{aligned}$$

where p is the probability of a zero-cost edge. The first product is for the case where there are two zero-cost frontier nodes under either D_1 or D_3 (or both), and the second product is for the case where there is at most one zero-cost frontier node under either D_1 and D_3.

Alternatively, we can calculate the probability of a zero-cost root-to-leaf path below B_2 as follows. Since both of the frontier nodes under C_4 have infinite cost (D_7 and D_8), if B_2 is chosen, then the next decision below B_2 will be C_3. In this case, the probability of a zero-cost complete path is just one minus the probability of no zero-cost paths below C_3.

$$\begin{aligned} &P(\text{zero-cost root-to-leaf-path} \mid B_2) \\ &\quad = 1 - P(\text{infinite-cost root-to-leaf-path} \mid B_2) \\ &\quad = 1 - ((1-p)^4 + 4p(1-p)^5 + 6p^2(1-p)^6 + 4p^3(1-p)^7 + p^4(1-p)^8) \end{aligned}$$

In the above equation, the five terms in the right half correspond to the five ways for the minimum-cost complete path to have infinite cost. The first term is for the case where all four edges directly below nodes D_5 and D_6 have infinite cost (probability equals $(1-p)^4$). The second term is when all but one of the four edges directly below D_5 and D_6 have infinite cost, but both of the edges below this single zero-cost edge have infinite cost. This occurs with probability $p(1-p)^5$, and there are four different ways for it to occur. The third term is for the case where only two of the edges directly below D_5 and D_6 have infinite cost, and all four of the edges below the two zero-cost edges have infinite cost. This case occurs with probability $p^2(1-p)^6$, and there are six ways to have only two zero-cost edges in the level below the D_i nodes. The fourth case is where all but

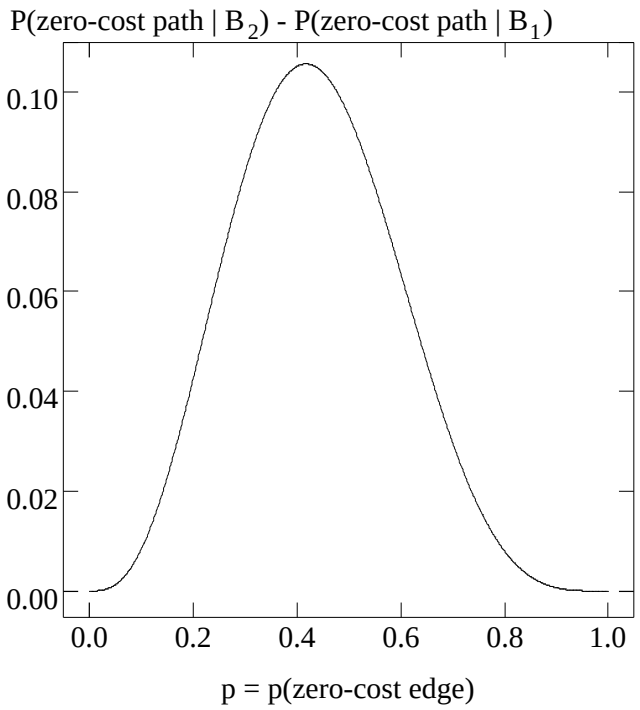


Figure 4.3: The difference between the probability of finding a zero-cost path below B_2, and the probability of finding a zero-cost path below B_1, given the probability, p, of a zero-cost edge in the unexplored part of the example tree.

one of the edges directly below D_5 and D_6 have zero cost, and all of the edges below these three zero cost edges have infinite cost. This event can occur four different ways, and the probability of each event is $p^3(1-p)^7$. Finally, the last term is for the case where all of the edges directly below D_5 and D_6 are also zero cost, but all of the edges in the bottom level of the tree have infinite cost. The probability of this event is $p^4(1-p)^8$.

A graph of the difference between these two probabilities as a function of p is shown in Figure 4.3. This graph shows that B_2 is the optimal second-to-last incremental decision for all values of p, even though the last incremental decision for this problem would have to be made by a tie-breaking rule. We could easily generate a variation of this problem where the optimal last incremental decision is to the left child, whereas the optimal second-to-last incremental decision is to the right child. The opportunity for this type of complication increases with the number of incremental decisions before the last incremental decision. Thus, any optimal n^{th} incremental decision must include these effects as part of the decision process.

For a problem-space tree with edge costs that are independently chosen from

the same distribution, one might reasonably conjecture that the optimal solution to the general or n^{th}-to-last incremental decision problem could be obtained by solving the last incremental decision based on the minimum-path-cost distribution for the remainder of the problem-space tree. We could easily adjust the example in Figure 4.2 so that the optimal last incremental decision was a move to B_1 by making one of the zero-cost frontier nodes below B_2 have a node cost equal to one instead of zero. In this case, the last incremental decision is not optimal given the minimum-path-cost distribution for even the second-to-last incremental decision, thus providing a counter example to our conjecture.

The optimal solution to the general incremental decision problem is at least as difficult as finding an optimal n^{th}-to-last incremental decision. To see why this is true, we can simply argue that if this weren't true, then we could just ignore the number of remaining incremental decisions, and make use of the solution to the general incremental decision problem as our decision. In order to optimally solve the n^{th}-to-last incremental decision problem, we need to know the expected cost of a root-to-leaf path that will be traversed below each child of the root node given the edge costs in the search tree, the general properties of the unexplored part of the problem space, the decision method, the exploration method, and the number of remaining incremental decisions. In other words, given the equation for following probability density function,

$$f(\text{minimum path cost through } B_i) = f(MPC(B_i)|T_i, U, D, E, n), \qquad (4.1)$$

we can calculate the expected root-to-leaf path cost that would be traversed for each child of the root by integrating the product of this distribution and the minimum path cost through node B_i ($MPC(B_i)$). We can then compare the expected minimum root-to-leaf path cost information to determine the optimal n^{th}-to-last incremental decision. In Equation 4.1, T_i is the search tree below node B_i, U is the general information about the unexplored part of the problem-space tree, D is the decision method, E is the exploration that will occur between decisions, and n is the number of remaining incremental decisions.

In general, we can't expect to have an oracle to provide us with the distribution of path costs that would be traversed by an optimal decision method below the children of the root. Instead, suppose that we have an oracle that tells us the distribution of actual traversed-path costs from the grandchildren of the root node to the bottom of the problem-space tree. In this case, we also need to know the probability that the optimal decision process will lead us to a particular grandchild given the edge costs in the search tree, and that we chose the child above it. Likewise, if the oracle provides the distribution of traversed path costs from the frontier nodes to the bottom of the problem-space tree, then to make an

optimal n^{th}-to-last incremental decision we will also need to know the probability that the optimal decision process will lead us to a particular frontier node given that we choose the child of the root that is above it. In general, this conditional probability is difficult to obtain because the optimal decision method is part of its own definition, namely that knowing how the optimal decision method will make future decisions is needed to make an optimal choice for the current decision.

Because of the difficulty in making use of the frontier node to leaf node traversed-path distribution information, and the fact that this information is not likely to be available, we have focussed on an approximation to the optimal incremental decision method. Our approach is to model the general incremental decision problem as if it were a last incremental decision problem. The last incremental decision problem is more tractable than the general incremental decision problem, and n^{th}-to-last incremental decision problem, because the rest of the decisions are optimal with respect to the remainder of the problem-space tree. When the remaining decisions are optimal, it is easier to describe the distribution for minimum-cost paths in a random tree model than it is to describe the distribution of paths that would be traversed by an incremental search algorithm. The last incremental decision approximation to general incremental decision making is discussed in greater detail in the next section.

4.2 Making Optimal Last Incremental Decisions

4.2.1 Calculating the Expected Minimum Path Cost ($E(MPC)$)

In order to make an optimal last incremental decision, we need to know the expected minimum cost of a root-to-leaf path that will be traversed after the incremental decision is made and the remaining edge costs are learned. In order to calculate the expected minimum root-to-leaf path cost (or simply the minimum path cost) in this case, we first need to know the distribution of possible minimum path costs. The problem of calculating the distribution for the minimum path cost from a child of the root node to a leaf node can be broken down into two steps. The first step is to determine the distribution for a minimum path cost from a frontier node to a leaf node in the tree, which we also refer to as the completion-cost distribution. The second step is to combine the frontier-node completion-cost distributions with the edge costs in the explored search tree, in order to generate the distribution of minimum root-to-leaf path costs under each child of the root.

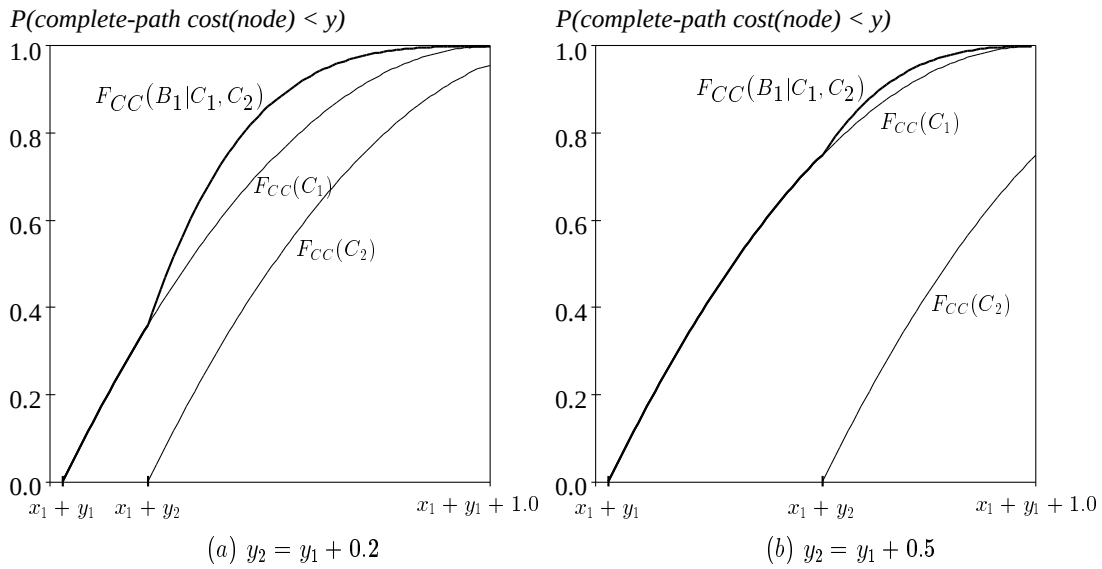
The problem of determining the expected minimum path cost in an unexplored

random tree has been previously studied [KP83, McD90b, MP91]. Unfortunately, there is no known closed form solution for the distribution or expected minimum path cost as a function of the tree depth (d) and branching factor (b). Although it is possible to express the steps of a process that can be used to generate the equations for the minimum-path-cost distribution, it becomes difficult to solve the equations for trees that are deeper than 2 levels. A detailed description of this process is presented in the next section. For our example in Figure 4.1, the cumulative completion-cost distribution for node C_1 ($F_{C_1}(z)$) can be calculated as follows. We assume that the edge costs are are independent and identically distributed.

$$\begin{aligned}
F_{C_1}(z) &= F(\min[z_1, z_2]) \\
&= p(\min[z_1, z_2] < z) \\
&= 1 - p(\min[z_1, z_2] > z) \\
&= 1 - p(z_1 > z) \cdot p(z_2 > z) \\
&= 1 - (1 - p(z_1 < z)) \cdot (1 - p(z_2 < z)) \\
&= 1 - (1 - F(z)) \cdot (1 - F(z)) \\
&= 2F(z) - F(z)^2
\end{aligned}$$

Given the distribution for a minimum cost path from a frontier node to a leaf node, the second step is to calculate the expected minimum cost path through a child of the root node. This involves performing a piece-wise combination of the frontier-node completion-cost cumulative distribution functions (CDF's). Figure 4.4 shows the completion-cost CDF's for frontier nodes C_1 and C_2 as well as the combined completion-cost CDF for node B_1 for two different relative values of y_1 and y_2 ($F_{CC}(C_1) = F_{C_1}(z - x_1 - y_1)$, $F_{CC}(C_2) = F_{C_2}(z - x_1 - y_2)$).

We observe that the completion-cost distribution for root-to-leaf paths through node B_1 in Figure 4.4 depends initially on the completion-cost distribution of node C_1. This is because the only way for the minimum path cost to be below $x_1 + y_2$ is when the path is through node C_1. When the minimum path cost exceeds the node cost of C_2 ($x_1 + y_2$), then the completion-cost distribution of node B_1 becomes a combination of the completion-cost distribution of nodes C_1 and C_2. The result is a sharp rise in the cumulative distribution as shown in Figure 4.4. By comparing Figures 4.4a and 4.4b, we observe that the effect of potentially having a minimum-cost path through node C_2 on the completion-cost distribution for node B_1 critically depends on the difference between y_2 and y_1 (*i.e.,* a smaller the difference between y_1 and y_2 means a greater effect on the minimum-path-cost distribution).



(a) $y_2 = y_1 + 0.2$ (b) $y_2 = y_1 + 0.5$

Figure 4.4: Completion-cost cumulative distribution functions for root-to-leaf paths through nodes B_1, C_1 and C_2 when (a) $y_2 = y_1 + 0.2$ and (b) $y_2 = y_1 + 0.5$.

Finally, the cumulative distribution functions for the children of the root node (*i.e.*, B_1 and B_2) are differentiated to yield the probability density functions (PDF's) for minimum path costs through these nodes. These PDF's are then used to calculate the expected minimum path costs (MPC's) through each child of the root (*e.g.*, $E(MPC(B_1))$ and $E(MPC(B_2))$) by integrating over the range of possible minimum path costs. The optimal decision is then to move to the child with the lowest $E(MPC)$ value.

We can now calculate the expected value of the two decisions in the example in Figure 4.1. If we assume without loss of generality that $y_1 \leq y_2$, then the equation for the expected cost of a complete path through node B_1 is:

$$\begin{aligned} E(MPC(B_1)) = \quad & x_1 + \textstyle\int_{y_1}^{y_2} 2(1+y_1-y) \cdot y \; dy \\ & + \textstyle\int_{y_2}^{y_1+1} 2((1+y_1-y) \cdot (1-2y+2y_2+(y-y_2)^2) \\ & \quad +(1+y_2-y) \cdot (1-2y+2y_1+(y-y_1)^2)) \cdot y \; dy \end{aligned} \tag{4.2}$$

An analogous equation applies for node B_2. Substituting the edge costs from the search tree in Figure 4.1 into equation 4.2, we get $E(MPC(B_1)) = 1.066$, whereas $E(MPC(B_2)) = 1.06$. Thus node B_2 is the best decision because it has the lowest expected minimum path cost. Intuitively, a move to node B_1 relies on

either z_1 or z_2 having a low cost, whereas a move to B_2 has four chances (z_5, z_6, z_7 and z_8) for a low final edge cost.

4.2.2 How Difficult is it to Calculate $E(MPC)$?

The short answer to this question is that we have only been able to calculate $E(MPC)$ for a very small set of search trees, because of the size and complexity of the equations. The long answer comprises the remainder of this section.

There are two factors that make it very difficult to calculate the $E(MPC)$ decision in general. The first is the fact that there is no easy way to solve the equations for the distribution of the minimum-cost path in random trees in general when the unexplored depth is greater than two. The second factor is that in the worst case there is a different term in the expected minimum-path-cost equation for each frontier node in the subtree of the decision being evaluated.

4.2.2.1 Calculating the completion-cost distribution

Although there is no known closed form solution for the distribution of minimum root-to-leaf path costs in a tree with a fixed branching factor b and depth d, there is a recursive formulation for calculating the distribution. The equations for this recursive formulation are shown in Figure 4.5, and are discussed in greater detail in Appendix B. At this point, we are mainly concerned with the complexity of solving these equations in the worst case. Note that we have assumed that the branching factor is fixed, although this problem becomes even more difficult for the case of random branching factors

For this discussion, we need some additional notation. Let $f_1(x)$ denote the probability density function for edge costs in a uniform random tree. Let $f_{\min_i}(x)$ denote the distribution for the minimum path cost in a tree of depth i. Let $f_i(x)$ denote the distribution of minimum root-to-leaf path cost through one child of the root node in a tree of depth i. We similarly define $F_{\min_i}(x)$ and $F_i(x)$ as the corresponding cumulative distribution functions. We finally use $E_{\min_i}(x)$ to denote the expected minimum path cost in a tree with depth i.

Consider the equations in Figure 4.5. The first equation is basically the definition of the expected cost. The second equation involves differentiating the cumulative distribution to find the density function for use in the expected-cost equation. The third equation describes how to combine b independent cumulative probability distributions to find the probability of the minimum over b choices.

Expected minimum-path-cost equations:

1. $E_{\min_i}(x) = \int f_{\min_i}(x) \cdot x \cdot dx$
2. $f_{\min_i}(x) = \frac{d}{dx} F_{\min_i}(x)$
3. $F_{\min_i}(x) = 1 - (1 - F_i(x))^b$
4. $F_i(x) = \int_0^x f_i(w) \cdot dw$
5. $f_i(x) = \int_0^x f_1(w) \cdot f_{\min_{(i-1)}}(x - w) \cdot dw$

Figure 4.5: Set of recursion equations for calculating the expected minimum path cost in a random tree.

This equation can be derived as follows.

$$\begin{aligned} F_{\min(z_1,...,z_b)}(z) &= P(\min(z_1, ..., z_b) \leq z) \\ &= 1 - P(\min(z_1, ..., z_b) > z) \\ &= 1 - P(z_1 > z) \cdot ... \cdot P(z_b > z) \\ &= 1 - (1 - P(z_1 \leq z)) \cdot ... \cdot (1 - P(z_b \leq z)) \\ &= 1 - (1 - F(z))^b \end{aligned}$$

The fourth equation describes how to calculate the cumulative distribution from the probability density function. The fifth equation defines the density of the minimum choice at level i as a convolution between the edge-cost density and the density of the minimum choice at level $i - 1$.

The completion-cost distribution is difficult to calculate because the distribution equations quickly become unmanageable. Consider a random tree with depth d and branching factor b whose edge-cost distribution is uniform over the range $[0, 1]$ (*i.e.*, $f_1(x) = 1,\ 0 \leq x \leq 1$). Assuming that we can perform the necessary integrals and derivatives in Equations 1, 2, and 4, it is Equations 3 and 5 that combine to make the resulting distribution difficult to express (*i.e.*, too large). For example, each time that Equation 3 is performed, the number of different terms in the expanded distribution equation can grow by a factor of b in the worst case. Thus the number of terms in the final distribution equation can be as large as $|F(z)|^{b^d}$, where $|F(z)|$ refers to the number of terms in the edge cost distribution equation. This means that the final equations for the completion cost distribution can have a number of terms that is exponential in the depth of the unexplored part of the problem-space tree.

In addition, each time that Equation 5 is executed, the number of distinct

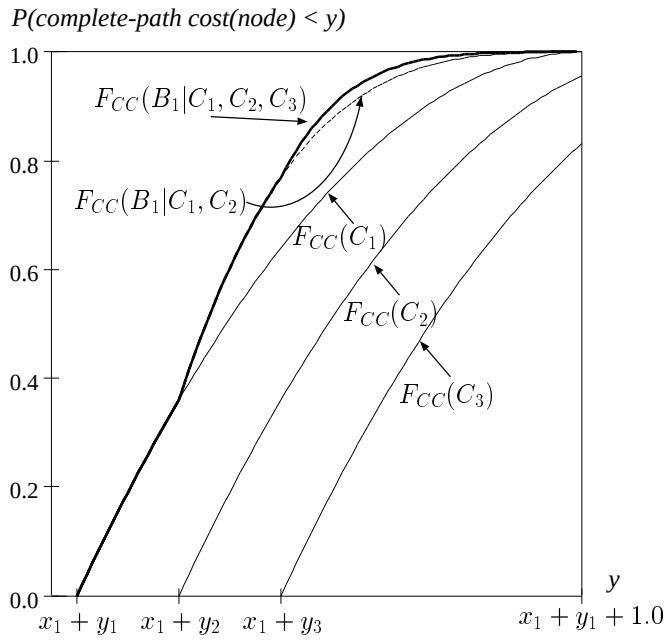
regions in the resulting distribution can, in the worst case, increase by the number of distinct regions in the edge-cost distribution. By a distinct region of the probability density function, we mean a part that can be described by a continuous function and that is not part of a larger distinct region. Since each region will require us to perform a different integral step, the number of integrals that could be necessary to calculate the completion-cost distribution in the worst case is $jd(d-1)/2$, where d is the depth of the unexplored tree and j is the number of distinct regions in the edge-cost distribution.

In practice, these two factors combine to make it difficult to calculate the minimum-path-cost distribution for random trees with $d \geq 3$ when the edge costs are uniformly distributed over the range $[0, 1]$.

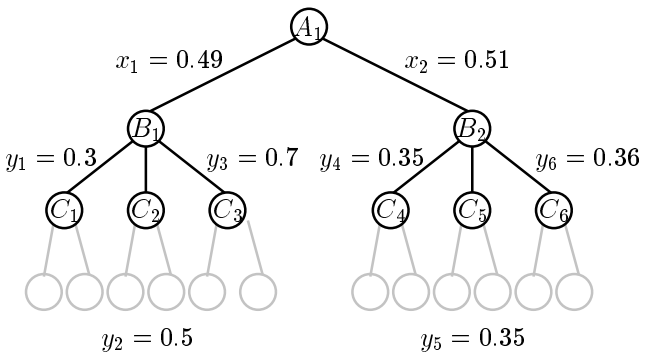
4.2.2.2 Combining the completion-cost distribution and edge costs

The complexity of combining frontier-node completion-cost distributions with the edge costs in the explored part of the problem space comes from the fact that the distribution of minimum path costs through a child of the root can have a distinct distribution function for each frontier node in the child node's subtree. For example, consider the graph and tree shown in Figure 4.6. This tree has one more frontier node under B_1 and B_2 than in our original example (Figure 4.1). The graph shows the cumulative distribution of minimum root-to-leaf path costs through nodes C_1, C_2, C_3, and B_1, versus the cost of a path to a leaf node. We observe that there are three frontier nodes under B_1, and that each frontier node contributes to the completion-cost distribution of B_1. For example, the change in the distribution at $x_1 + y_2$ is due to the fact that at first every minimum-cost path below B_1 must also pass through C_1. Once the minimum path cost is greater than or equal to $x_1 + y_2$, then the minimum-cost path below B_1 can be through either C_1 or C_2. This additional choice causes a discontinuity in the cumulative distribution function for B_1 at $x_1 + y_2$, as shown in the graph. There is a similar discontinuity at $x_1 + y_3$, which is due to the fact that the minimum-cost path below B_1 can be also be through C_3 for minimum path costs greater than or equal to $x_1 + y_3$. Since the number of frontier nodes in each subtree is exponential in the search depth, in the worst case the number of different functions needed to describe the minimum-path-cost distribution is also exponential in the search depth. This is why calculating the expected minimum root-to-leaf path cost and consequently making an optimal last incremental decision is impractical in general.

We have shown that it is difficult to solve for the completion-cost distribution, and that combining the completion-cost distribution and search-tree edge costs



(a)



(b)

Figure 4.6: Completion-cost cumulative-distribution functions for the minimum root-to-leaf path cost through nodes B_1, C_1, C_2 and C_3 when $y_2 = y_1 + 0.2$ and $y_3 = y_2 + 0.2$ and the associated search tree (b).

is impractical, due to the size and complexity of the equations needed to make an optimal decision. For example, with the help of Maple[1], we have only been able to solve the expected minimum-path-cost equations for random trees, with edge costs uniformly distributed over the range $[0, 1]$, for the case where each subtree of the root has seven or less frontier nodes. Since the $E(MPC)$ equations can have a separate term for each frontier node in a subtree, the space required to calculate $E(MPC)$ appears to be exponential in the search depth. In other words, we are not aware of any way to calculate the optimal decision that in the worst case doesn't require us to store every frontier-node cost, and thus we conjecture that the worst-case space complexity of $E(MPC)$ is exponential in the search depth. These results serve as justification for our new approximation method, which we will describe in Chapter 6.

[1]Maple is an interactive mathematical tool developed at the University of Waterloo.

CHAPTER 5

Making a Single Incremental Decision: A Comparison of MINIMIN and $E(MPC)$

In this chapter, we present two algorithms for making a single incremental decision in a real-time setting. First, we describe the MINIMIN decision method and an expected minimum-path-cost algorithm called $E(MPC)$, which is optimal for the last incremental decision problem. We then develop a bound on the worst-case expected error when MINIMIN is used to make a last incremental decision.

The second section presents experimental results that compare the performance of $E(MPC)$ and MINIMIN on a small set of search trees for which an implementation of $E(MPC)$ is possible. Our results show that, for these small trees, the performance of MINIMIN is very close to that of $E(MPC)$.

The third section describes how we used a Monte-Carlo simulation method to compare the relative performance of $E(MPC)$ and MINIMIN on larger search trees. Our results show that the performance of MINIMIN is still quite good relative to $E(MPC)$ on larger trees, with increased search depth, branching factor and unexplored depth.

The main result of this chapter is a comparison of the MINIMIN and $E(MPC)$ decision methods, both analytically and through experiments on a variety of random-tree problems. These results show that, although it is possible to improve upon MINIMIN's performance on the last incremental decision problem, the margin for improvement is bounded by the expected minimum path cost in the unexplored part of the problem space tree. Since the expected minimum path cost is an increasing function of the unexplored depth, the margin for improvement will increase with the number of unexplored levels in the problem-space tree.

5.1 Making a Last Incremental Decision

In this section, we present MINIMIN and $E(MPC)$, which are two methods for making incremental decisions.

5.1.1 The MINIMIN Decision Method

MINIMIN algorithm [Kor90] is the single-agent analog of the minimax algorithm for two-player games, such as chess. Minimax first generates a bounded-depth search tree, and then backs up the values of the frontier nodes by either minimizing the children's values for the opponent's moves or maximizing the children's values for the player's moves. Minimax then makes a move decision by choosing a child with the highest backed-up value. Minimax thus refers to both the exploration and the decision making.

The MINIMIN algorithm operates as follows. The input to MINIMIN is the explored search tree. We assume that the objective is a low-cost path to a goal. In this case, the heuristic values of the frontier nodes are passed up the tree, and at each level, the value of a node is the minimum of its children's backed-up values. This is the motivation behind the name MINIMIN. For example, in Figure 5.1, the MINIMIN value of the root node (5) is the minimum of the backed-up values of its children ($\min(5,6)$). Once a backed-up value for each child of the current root node is known, then MINIMIN commits to the move associated with a child that has the lowest backed-up value. This is equivalent to moving one step along the path to a minimum-cost frontier node. This least commitment strategy is based on the observation that subsequent limited-horizon explorations may provide additional information for the remaining decisions. If the problem-space tree has been completely explored, then the MINIMIN decision is optimal.

The heuristic values used for frontier nodes will depend on the specific problem instance. For the random-tree problems that we have considered, the objective is a minimum-cost path, so the obvious thing to backup is either the minimum cost of the frontier nodes or the minimum expected cost of a path to a leaf node. A MINIMIN decision based on the minimum-cost frontier node will be optimal when the tree is completely explored. Alternatively, the expected cost of a path to a leaf node is one useful way to compare the cost of nodes at different depths in the search tree.

One big advantage of MINIMIN is that, when the edge-costs are non-negative, MINIMIN will make the same decision on a full-width, fixed-depth search tree, as on a complete depth-first branch-and-bound exploration to the same search depth. This is because the only information that MINIMIN needs is the location of a minimum-cost frontier node, thus the MINIMIN decision can be made with many fewer node generations if a depth-first branch-and-bound exploration is used instead of a full-width, fixed-depth exploration. MINIMIN will also make the same decision on a tree generated by a best-first exploration provided that the exploration is stopped when a frontier node at the depth bound is chosen

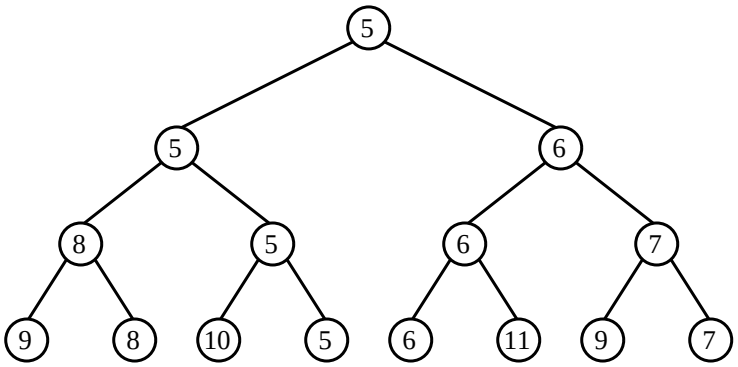


Figure 5.1: An example of MINIMIN backed-up values.

for expansion. If the best-first exploration is stopped at any other point, then there is no guarantee that a MINIMIN decision based on a best-first exploration will be the same as a MINIMIN decision based on depth-first branch-and-bound exploration.

5.1.2 The Decision-Theoretic Approach ($E(MPC)$)

The expected minimum-path-cost algorithm ($E(MPC)$) makes move or action decisions by choosing a child of the current root node that is on a root-to-leaf path that has the lowest expected minimum cost. This algorithm makes optimal decisions when there is only one incremental decision left (*i.e.,* the last incremental decision problem). The fact that we are focussing on the last incremental decision problem allows us to ignore the effect of the decision-making algorithm itself, and other secondary effects, on the expected cost of the path that will ultimately be traversed.

To summarize, the $E(MPC)$ decision algorithm calculates the expected minimum root-to-leaf path cost through each child of the current root node. This calculation requires us to know the distribution of minimum-cost paths below the frontier nodes, and also depends on the knowledge that the remaining decisions will be optimal given the current decision. The $E(MPC)$ decision is then to move to a child of the root that has a minimum expected root-to-leaf path cost.

5.1.3 How Much Can We Expect to Gain by Using $E(MPC)$?

An obvious question at this point is what is the most that we can hope to gain by making better use of the search-tree information, or, equivalently, what is the most that we can lose by using MINIMIN, which ignores all but the cost of the

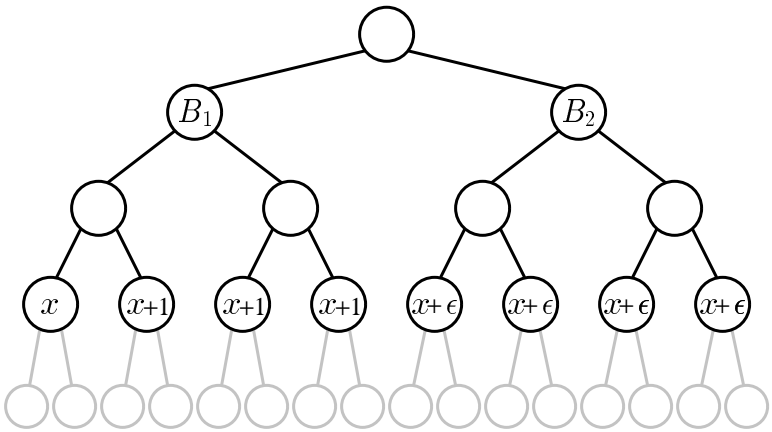


Figure 5.2: Example of a choice between one minimum-cost path and many paths with near minimum cost.

best frontier node? In particular, we are interested in the worst-case expected error of a MINIMIN decision. By worst-case expected error we mean the largest that the expected error can be, where the expectation is over possible exploration outcomes.

The worst-case expected error for a MINIMIN decision occurs when MINIMIN chooses a decision that only has one good frontier node, and all of the frontier nodes below the decision that wasn't chosen by MINIMIN are good, in the sense that their node costs are very close to (or perhaps equal to) the minimum frontier-node cost. This is the situation where $E(MPC)$ has the greatest potential advantage over MINIMIN. An example of this situation is shown in Figure 5.2, where ϵ is a small constant.

For this example, the leftmost frontier node is the minimum-cost frontier node with cost x, and the other frontier nodes below node B_1 have node costs equal to $x + 1$, which is equal to the minimum frontier-node cost plus the maximum edge cost. This means that only the leftmost frontier node can contribute to the expected minimum path cost below B_1. We assume that there are k frontier nodes below B_2, all of which have node cost equal to $x + \epsilon$. As before, we also assume that there is one remaining unexplored level of the tree and that the branching factor is b.

The expected error of a MINIMIN decision is simply the difference between the expected minimum path cost below the MINIMIN decision (B_1), and the expected minimum path cost below the $E(MPC)$ decision (B_2). If the unexplored edge costs are independently chosen from a uniform distribution over the range $[0, 1]$, then the cumulative distribution is $F(z) = z$ $(0 \leq z \leq 1)$. Since the minimum-

cost frontier node is the only frontier node that can appear on a minimum-cost path below B_1, the expected minimum path cost below B_1 is equal to x plus the expected completion cost from the minimum-cost frontier node to a leaf node one level below. In this case, the expected completion cost is simply the expected value of a minimum choice between b random edge costs.

From Chapter 4, we know that the cumulative distribution for the minimum of b random values, each with distribution $F(z) = z$, can be expressed as follows.

$$\begin{aligned} F_{\min(z_1,...,z_b)}(z) &= P(\min(z_1, ..., z_b) \leq z) \\ &= 1 - (1 - F(z))^b \\ &= 1 - (1 - z)^b \end{aligned}$$

Next, the probability density function is just the derivative of the cumulative distribution function.

$$f_{\min(z_1,...,z_b)}(z) = b(1 - z)^{b-1}$$

Finally, the expected value of b random choices can be calculated as follows.

$$\begin{aligned} E(\min(z_1, ..., z_b)(z)) &= \int_0^1 b(1 - z)^{b-1} z \quad dz \\ &= b(1 - x)^b (\frac{1 - x}{b + 1} - \frac{1}{b})|_0^1 \\ &= \frac{1}{b + 1} \end{aligned}$$

Thus the expected minimum path cost below B_1 is $x + 1/(b + 1)$.

In order to calculate the expected minimum path cost below B_2, we simply observe that the optimal decision after moving to B_2 will be a choice between kb edge costs added to one of the k frontier nodes that have cost $x + \epsilon$. In this case, the expected value of kb random choices can be calculated as follows.

$$\begin{aligned} E(\min(z_1, ..., z_{kb})(z)) &= \int_0^1 kb(1 - z)^{kb-1} z \quad dz \\ &= kb(1 - x)^{kb} (\frac{1 - x}{kb + 1} - \frac{1}{kb})|_0^1 \\ &= \frac{1}{kb + 1} \end{aligned}$$

Thus the expected minimum path cost below B_2 is $x + \epsilon + 1/(kb + 1)$.

The worst case expected error for a MINIMIN decision, when the search tree has k frontier nodes per child node, is simply the difference between these two

expected minimum path costs.

$$\begin{aligned} E(MPC(B_1)) - E(MPC(B_2)) &= x + \frac{1}{b+1} - x - \epsilon - \frac{1}{kb+1} \\ &= \frac{1}{b+1} - \frac{1}{kb+1} - \epsilon \\ &= \frac{(k-1)b}{(b+1)(kb+1)} - \epsilon \end{aligned}$$

In the limit as k goes to infinity and ϵ goes to zero, this difference approaches $1/(b+1)$, which is the expected completion cost for the path below the minimum-cost frontier node. This make sense, because as the number of frontier nodes below B_2 increases, it becomes more likely that a zero-cost edge will be generated.

For the case of uniformly distributed $[0, 1]$ edge costs, it is easy to see that the expected minimum path cost through the child chosen by $E(MPC)$ approaches x as the number of frontier nodes with cost x approaches infinity. Thus for this edge-cost distribution, the most that can be gained by using $E(MPC)$ instead of MINIMIN is the expected completion cost below the MINIMIN decision ($E_{CC}(z) = 1/(b+1)$).

Intuitively, as the number of frontier nodes k approaches infinity, the likelihood of finding a zero-cost path from a frontier node to a leaf node in the subtree under node B_2 approaches one. Thus the expected cost of the minimum root-to-leaf path through B_2 approaches x. Since the number of frontier nodes does not affect the quality of the MINIMIN decision in the example, its expected root-to-leaf path cost is still the cost of the minimum-cost frontier node (x) plus the expected cost of a path from the minimum-cost frontier node to a leaf node one level below it ($E_{CC}(z)$). Thus, the expected root-to-leaf path cost of the MINIMIN decision can be greater than the expected root-to-leaf path cost of the $E(MPC)$ decision by at most the expected cost of a path from the minimum-cost frontier node to a leaf node, namely $E_{CC}(z)$. In general, the size of $E_{CC}(Z)$ relative to the total path cost will depend on the search depth, unexplored depth, branching factor, and edge-cost distribution.

5.2 Experimental Comparison: $E(MPC)$ versus MINIMIN.

Admittedly, the worst-case situation for MINIMIN decision making when compared with $E(MPC)$ is rather contrived. What is more important than the worst-case expected error is the average-case expected error. For instance, how often does MINIMIN make the same decision as the optimal $E(MPC)$ algorithm?

Tree			$E(MPC)$ decisions			MINIMIN decisions			Same
b	s	u	optimal	error	wins	optimal	error	wins	Decision
2	2	1	84.5%	3.1%	1.7%	84.2%	3.2%	1.5%	96.8%
2	2	2	80.8%	4.0%	2.2%	80.4%	4.1%	1.9%	95.9%
2	3	1	87.2%	1.7%	2.0%	86.8%	1.9%	1.6%	96.4%
3	2	1	81.9%	4.0%	2.5%	81.6%	4.2%	2.2%	95.3%

Table 5.1: Results comparing $E(MPC)$ decisions with MINIMIN decisions for balanced random trees.

Another question is what is the average cost in terms of solution quality if we use the MINIMIN decision strategy instead of the $E(MPC)$ decision strategy?

To answer these questions, we implemented the $E(MPC)$ equations for search trees generated by either a balanced exploration strategy (full-width, fixed-depth), or a best-first exploration strategy, and performed a set of experiments to compare the average-case performance of $E(MPC)$ and MINIMIN.

5.2.1 Balanced Exploration

In order to compare the performance of MINIMIN and $E(MPC)$, we performed a set of experiments on balanced search trees. A balanced search tree has all frontier nodes at the same level of the tree. The motivation for looking at balanced search trees is that we can easily implement the equations for $E(MPC)$ on these trees.

For the balanced exploration experiments, we implemented the $E(MPC)$ equations for the search tree in Figure 4.1 and three other balanced trees created from $T(b = 2, s = 2, u = 1)$ by incrementing either b, s, or u by one (see Figure 5.3). We performed the following experiment on all four trees. For each trial, random values were assigned to each edge in the explored tree uniformly from the range $[0, 1]$, and then the MINIMIN and $E(MPC)$ decisions were calculated based on the edge costs in the explored part of the tree and the edge-cost distribution. Random values were then assigned to the remaining edges and the complete-solution path costs were calculated for both decision methods. In addition, the optimal path cost for the whole tree was recorded for comparison.

The results in Table 5.1 show the percentage of the trials that the MIN-

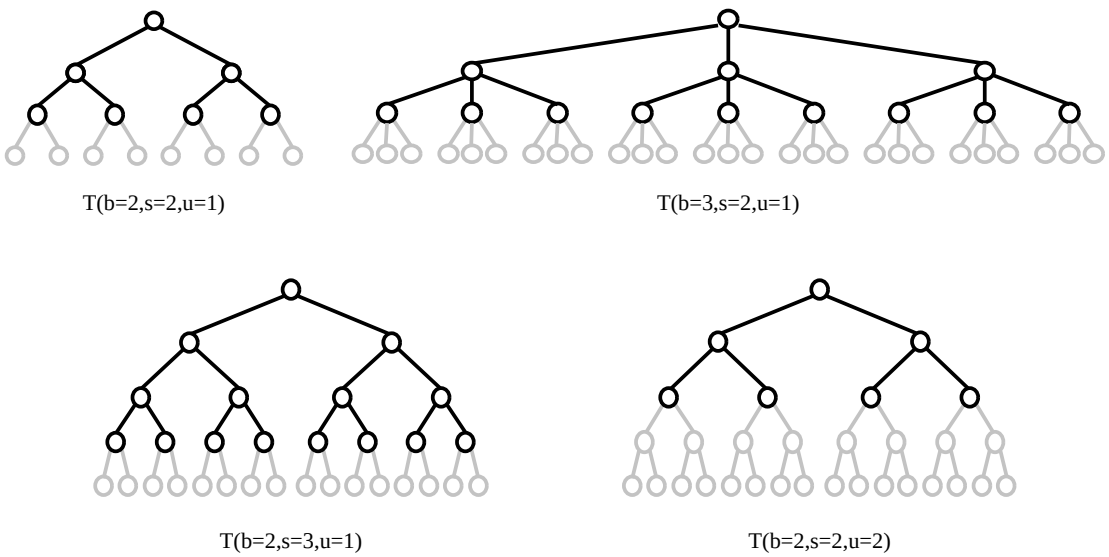


Figure 5.3: Four balanced search trees

IMIN and $E(MPC)$ decisions are equal to the first step on the optimal path, the solution-cost error with respect to the optimal path cost (*i.e.*, $100*(\textit{solution cost} - \textit{optimal cost})/\textit{optimal cost}$), and the percentage of the trials that each decision algorithm wins (*i.e.*, produces a lower-cost solution). In addition, the last column shows the percentage of the trials that the $E(MPC)$ method and the MINIMIN method produced the same decision. The results are averaged over one-million trials for each line of the table.

The results for randomly generated, uniform-depth search trees show that MINIMIN decisions are quite good. For instance, MINIMIN makes the same decision as the $E(MPC)$ method more than 95% of the time, and its average error rate is only slightly higher (*i.e.*, $\leq .2\%$). In addition, the $E(MPC)$ method only wins a small fraction more of the head-to-head competitions. As might be expected, the error rate decreases with increased search depth, and increases with increased unexplored depth or branching factor, although the amount of increase is relatively small (*i.e.*, less than 1%).

5.2.2 Best-First Exploration

To further analyze the difference between MINIMIN decision making and the $E(MPC)$ method, we also performed a set of experiments on random trees that correspond to a best-first exploration using node costs. We first looked at balanced best-first search trees, again because we can only implement the $E(MPC)$ equations for a small set of trees. Balanced best-first search trees were generated

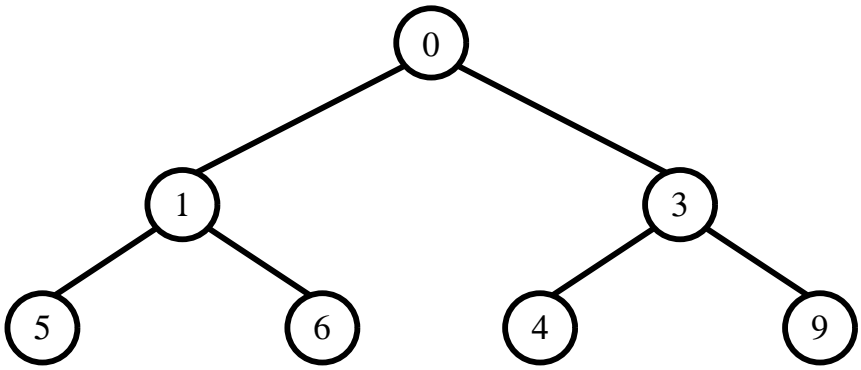


Figure 5.4: Example of a balanced best-first search tree.

as follows. For each of the trees in Figure 5.3, random edge costs were assigned to the black edges in the tree. The tree was then evaluated to determine if it could have been created by a best-first search. In a best-first search tree with non-negative edge costs, all frontier node costs must be greater than or equal to the node cost of all expanded nodes. This is because best-first exploration will only expand a node if it is a lowest-cost frontier node. If this property did not hold, then the edge costs were discarded, and a new set was generated. An example of a best-first search tree is shown in Figure 5.4, where the node costs are shown in the nodes. This procedure was adopted to easily generate best-first search trees of a given structure.

As before, we performed a set of experiments on the four trees in Figure 5.3, except this time the edge costs were chosen so that the tree satisfies the best-first property. For each trial, random values were assigned to each edge in the explored tree from the range $[0, 1]$, subject to the best-first search tree properties. The MINIMIN and $E(MPC)$ decisions were then calculated based on the edge costs in the explored part of the tree. Next, random values were assigned to the remaining edges, and the complete-solution path costs were calculated for both decision methods. The optimal path for the whole tree was also recorded for comparison. The results of these experiments averaged over one-million best-first search trees are shown in Table 5.2.

For this set of balanced best-first search trees, the results also show that the MINIMIN decisions are quite good on average. As with the balanced random edge-cost trees, the solution-cost error for MINIMIN decisions is only slightly higher than the $E(MPC)$ decisions, and the win/loss ratio is very close to 1. The overall performance of both MINIMIN and $E(MPC)$ relative to optimal is worse on the balanced best-first search trees because the choice in a balanced best-first search tree is between frontier nodes that are closer in value than in a

Tree			$E(MPC)$ decisions			MINIMIN decisions			Same
b	s	u	optimal	error	wins	optimal	error	wins	Decision
2	2	1	76.4%	4.8%	3.3%	76.0%	4.9%	2.9%	93.9%
2	2	2	72.1%	5.9%	4.3%	71.3%	6.1%	3.5%	92.2%
2	3	1	82.5%	2.4%	3.2%	81.9%	2.5%	2.6%	94.2%
3	2	1	65.5%	6.4%	7.4%	64.8%	6.9%	6.4%	86.2%

Table 5.2: Best-first results comparing distribution-based decisions with MINIMIN decisions for balanced search trees.

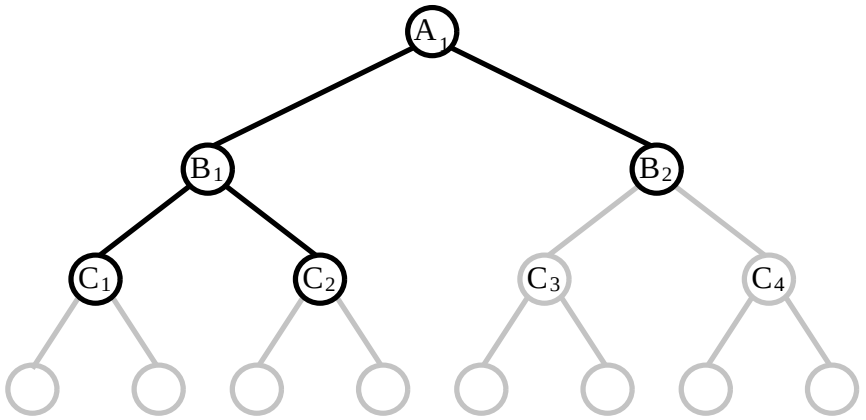


Figure 5.5: Example of an unbalanced search tree.

balanced tree with random edge costs. Intuitively, the random trees that meet the best-first search property have frontier nodes that are closer in cost, and thus the optimal decision depends more on the information in the unexplored level than in the random balanced search trees.

Since it is likely that a best-first exploration will not generate a balanced search tree, we also implemented the $E(MPC)$ equations for a small set of unbalanced search trees. Consider the unbalanced search tree in Figure 5.5, where the black edges and nodes have been explored, and the gray edges have not been explored. The traditional interpretation of MINIMIN, which we call node-cost MINIMIN, moves one step toward the frontier node with the lowest node cost (*e.g.*, $nc_MINIMIN = \min(C_1, C_2, B_2)$). Since the MINIMIN decision strategy is designed for a full-width fixed-depth search tree (or its associated

DFBnB tree), we have developed two alternate interpretations of MINIMIN in order to make decisions on an unbalanced search tree. Deepest-complete MINIMIN moves one step toward the frontier node with the lowest node cost at the deepest completed search level (*e.g.*, $dc_MINIMIN = \min(B_1, B_2)$). Lowest-deepest MINIMIN moves toward the deepest node with the lowest node cost (*e.g.*, $ld_MINIMIN = \min(C_1, C_2)$). Traditional MINIMIN will bias the choice toward the least explored part of the search tree, whereas deepest-complete MINIMIN will ignore potentially valuable information. Lowest-deepest MINIMIN can also ignore valuable information, particularly when there are only a few nodes at the deepest level. For these reasons, we also considered a further extension to MINIMIN decision making that uses the information about the edge-cost distribution and search tree depth to estimate the expected path cost of frontier nodes ($ec_MINIMIN$). This is accomplished by summing the node cost of a frontier node and an estimate of the expected completion cost from a frontier node to a leaf node. An estimate of the expected path cost of node C_1 in Figure 5.5 is calculated as follows.

$$ec_MINIMIN(C_1) = node_cost(C_1) + \hat{E}(completion_cost(C_1)) \qquad (5.1)$$

For a binary tree with edge costs chosen uniformly from $[0, 1]$, the expected completion cost from a frontier node to a leaf node one level below it is $1/3$, which is the expected cost of the minimum of two random edge costs that are independently chosen from a uniform distribution over $[0, 1]$. The expected completion cost from a frontier node to a leaf node two levels below it is $391/630 \approx .620635$. This value comes from solving the expected minimum-path-cost equation given the distribution of minimum path costs in a binary tree of depth 2. For larger unexplored depths, this is difficult to calculate analytically, but it can easily be estimated from experimental data. $E(MPC)$ differs from $ec_MINIMIN$ in the following way. $E(MPC)$ moves to the *child of the root node* that has the lowest *expected* minimum root-to-leaf path cost, whereas $ec_MINIMIN$ moves one step *toward the frontier node* that has the lowest *estimated* root-to-leaf path cost. The fact that $E(MPC)$ calculates the expected minimum path cost for each child node, and then chooses between the children of the root node, allows it take into account the additional choices that will be made in the future. On the other hand, $ec_MINIMIN$ calculates the estimated root-to-leaf path cost for each frontier node, and then makes a move by choosing between the frontier nodes. Thus $ec_MINIMIN$ essentially ignores all of the decisions that will be made between the current root node and the frontier of the search tree.

To investigate the effect of unbalanced search trees on the decision quality, we performed a new set of experiments on the unbalanced random trees shown in

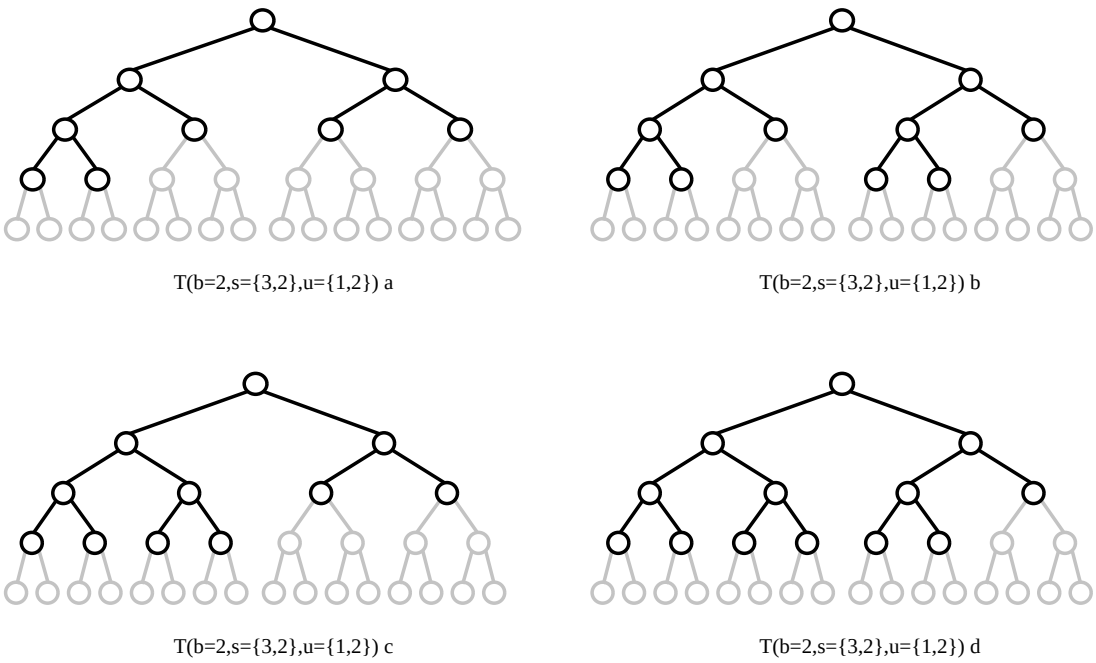


Figure 5.6: Four unbalanced search trees

Figure 5.6. For each trial, random values were assigned to each black edge in the explored tree uniformly from the range $[0, 1]$, and subject to the best-first search tree property. The $E(MPC)$ and four alternative MINIMIN decisions were then calculated. Random values were then assigned to the rest of the tree, and the complete solution path costs were calculated for all four decision methods. The results for this set of experiments are shown in Table 5.3. These results are averaged over 10,000 trials each.

The solution-cost error of $nc_MINIMIN$ and $dc_MINIMIN$ increases relative to optimal and to $E(MPC)$ as the number of nodes in the bottom level of the search increases. $ld_MINIMIN$ performs better as the next level of the tree is explored, performing nearly as well as $ec_MINIMIN$ on tree *d*. $ec_MINIMIN$ outperforms the three other MINIMIN algorithms and performs nearly as well as the $E(MPC)$ algorithm on all four trees. One important point to note here is that non-uniform exploration hurts the quality of pure MINIMIN decisions.

5.3 Evaluation of Larger Search Trees

The results in the previous section show that MINIMIN is a reasonable decision strategy for the last incremental decision problem on small search trees. As

Tree	$E(MPC)$ decisions			MINIMIN decisions				Same
	optimal	error	wins	alg	optimal	error	wins	Decision
a	84.1%	2.7%	6.5%	nc	81.5%	3.8%	3.9%	89.6%
			7.3%	dc	81.0%	4.0%	4.2%	88.6%
			7.3%	ld	81.0%	4.0%	4.2%	88.6%
			2.3%	ec	83.4%	2.9%	1.6%	96.2%
b	73.2%	4.1%	13.0%	nc	67.3%	5.8%	7.2%	79.9%
			30.1%	dc	55.4%	11.1%	12.3%	57.6%
			5.6%	ld	72.0%	4.5%	4.4%	90.0%
			4.5%	ec	72.5%	4.3%	3.9%	91.6%
c	87.4%	1.8%	19.4%	nc	75.5%	6.2%	7.5%	73.1%
			2.2%	dc	86.7%	2.0%	1.5%	96.3%
			2.2%	ld	86.7%	2.0%	1.5%	96.3%
			1.3%	ec	87.2%	1.9%	1.1%	97.7%
d	74.1%	3.5%	20.4%	nc	63.2%	6.5%	9.5%	70.1%
			23.9%	dc	60.6%	7.5%	10.4%	65.8%
			5.7%	ld	73.1%	3.8%	4.6%	89.7%
			5.5%	ec	73.0%	3.8%	4.4%	90.1%

Table 5.3: Results comparing distribution-based decisions with four different MINIMIN decision strategies on unbalanced best-first search trees.

mentioned above, these search trees were chosen because they were small enough to make it possible to solve (and also to implement) the equations for the expected minimum path cost as a function of the edge costs and the edge-cost distribution. In order to determine if these results extend to larger search trees, we performed a set of Monte-Carlo simulations on larger search trees.

5.3.1 Using Monte-Carlo Simulation to Evaluate Larger Search Trees

The idea behind the our Monte-Carlo simulations[1] is to use an average of the minimum root-to-leaf path cost in a set of simulated completions for each child of the current root as an estimate of the expected cost of that decision. The estimated expected costs can then be used to both make an estimated $E(MPC)$ decision and to measure the expected improvement that could be gained by using $E(MPC)$ instead of MINIMIN.

In order to estimate the expected cost of a decision for search trees larger than we could analyze in the previous section, we performed the following Monte-Carlo simulation steps. First, we perform depth-first branch-and-bound search with a fixed search depth, assigning the edge costs independently from a uniform distribution over the range $[0, 1]$ using the method described in Chapter 2. The result is a partially explored (unbalanced) search tree. Given the search tree, we calculated an estimate of the expected root-to-leaf path cost for each child of the root by randomly generating a set of edge costs for the unexplored edges in the problem space tree, and then calculated the minimum path cost for each child of the root. We performed 1000 different samples, and calculated the average minimum path cost to a leaf node for each child of the root. This average minimum completion cost was then used as an estimate of the expected minimum path cost. The $\hat{E}(MPC)$ decision is to move to the child of the root with the lowest estimated minimum root-to-leaf path cost. Note that a MINIMIN decision on a depth-first branch-and-bound search tree is the same as a MINIMIN decision on a full-width exploration to the same search depth. Conversely, an $E(MPC)$ decision on a DFBnB search tree may differ from an $E(MPC)$ decision on the complete search tree to the same depth because it is based on a different set of frontier-node costs.

[1] Matt Ginsberg suggested the idea of using Monte-Carlo simulations to examine larger search trees.

5.3.2 Monte-Carlo Simulation Results

We performed a set of experiments on trees with different search depths, unexplored depths and branching factors. In all experiments, the tree was explored using depth-first branch-and-bound to a fixed depth (s), and then an estimate of the expected minimum path cost for each child of the root node was calculated by randomly generating the remainder of the problem space tree, and averaging the minimum path cost for each root child over 1000 different samples. The average minimum path costs were then used to approximate the $E(MPC)$ decision. The remainder of the tree was then generated one additional time, and the minimum path cost below the chosen child of the root was recorded, along with the optimal path cost in the final problem space tree. Results were collected this way to separate the sampling phase from the decision making.

The results are presented in Appendix C, and can be summarized as follows. Both $\hat{E}(MPC)$ and MINIMIN make optimal decisions for a large fraction of the trials (*i.e.,* $> 80\%$). In addition, the percent solution-cost error for $\hat{E}(MPC)$ was slightly although consistently lower than for MINIMIN (*e.g.,* 0.04% lower for random $T(b = 2, s = 10, u = 1)$ trees). Finally, for most of the experiments, $\hat{E}(MPC)$ won a slightly larger fraction of the head-to-head competitions than MINIMIN (*e.g.,* $1.82\% \pm 0.08$ wins for $\hat{E}(MPC)$ versus $1.42\% \pm 0.07$ for MINIMIN on random $T(b = 2, s = 10, u = 1)$ trees). $\hat{E}(MPC)$ and MINIMIN made the same decision in over 94% of the trials performed.

In general, these Monte-Carlo simulation results confirm the results from the previous section, namely that MINIMIN often makes the same decision as the $E(MPC)$ algorithm and has only a slightly higher average percent error over a large sampling of search trees. In addition, the difference between the estimated expected cost of the MINIMIN decision and the $E(MPC)$ decision is consistently a small positive value, indicating that $E(MPC)$ decisions are only a slight improvement over MINIMIN decisions for the last incremental decision problem on this class of random trees.

Thus far, we have presented two disparate incremental decision methods: MINIMIN, which is easy to calculate but makes suboptimal decisions, and the optimal decision strategy ($E(MPC)$), which is impractical in general. In the next chapter, we present a new decision method that bridges the gap between MINIMIN and the optimal decision method.

CHAPTER 6

Approximating the Optimal Decision Method

In this chapter, we present a new method for approximating $E(MPC)$. The first section describes our approximation to $E(MPC)$ called k-best. The idea is to approximate the $E(MPC)$ decision method by only considering the effect of the k-best frontier nodes below a child of the root node. We discuss how to explore the problem space while maintaining the k-best frontier nodes, and observe that MINIMIN makes the same decisions as k-best when $k = 1$. We then present experimental results that compare the performance of k-best with MINIMIN on a set of random-tree decision problems. We then show analytically that the maximum expected error that can result from only considering the k-best frontier nodes is a decreasing function of k. The conclusion of this section is that k-best is a reasonable approximation for $E(MPC)$ in terms of decision quality, but that the computation cost is very high. This means that a fair real-time comparison between MINIMIN and k-best would have to allow MINIMIN to search deeper on average. In practice, the average value of one additional search-ply far exceeds the improvement in average solution quality produced by k-best. Thus, MINIMIN is still the algorithm of choice.

To answer the efficiency concerns brought up in Section 6.1, the second section contains a description of a further approximation called αk-best, that estimates the k-best frontier nodes by the k-best frontier nodes *seen* during a single-bound branch-and-bound exploration of the problem space. We present experimental results that compare αk-best with MINIMIN. The results show that αk-best is a slight improvement over MINIMIN, and uses on average the same number of node generations for a given search-depth bound.

6.1 k-best: An Approximate $E(MPC)$ Algorithm

Although the $E(MPC)$ algorithm makes optimal decisions for the last incremental decision problem, we showed in Chapter 2 that solving the optimal decision equations is impractical, due to the size and complexity of the expected minimum-path-cost equations. For the case where the edge costs are independently chosen from a uniform distribution over the range $[0, 1]$, we have only been able to solve

the expected minimum-path-cost equations for search trees that have at most seven frontier nodes in each subtree of the root node (*i.e.,* under each child of the root node). In this section, we present an approximation of the expected minimum-path-cost algorithm on larger search trees.

6.1.1 Description of the k-best Algorithm

The idea behind our approximation of $E(MPC)$ is based on the observation that the lowest-cost frontier node has the greatest effect on the distribution of minimum path costs at a child of the root node. This is because the lowest-cost frontier node determines the starting point and initial segment of the minimum-path-cost distribution (*i.e.,* the smallest possible path cost). The next largest effect is due to the cost of the second-best frontier node, and the size of the effect on the minimum-path-cost distribution diminishes as we approach the highest-cost frontier node. In fact, it is often the case that the higher cost frontier nodes do not contribute at all to the distribution of minimum path costs. The effect of an additional frontier-node cost on the distribution was previously discussed in Chapter 4, in particular, see Figures 4.4 and 4.6.

The k-best approximation to $E(MPC)$ simply ignores all but the k-best frontier node costs below each child of the root node. When $k = 1$, the decisions made by k-best are the same as the decisions made by MINIMIN. When the number of frontier nodes below each child is less than or equal to k, then k-best is optimal. Thus k-best defines a spectrum of approximation algorithms between MINIMIN and $E(MPC)$. The value for k in k-best is chosen to be small enough to make it possible to calculate the expected minimum path cost given k frontier-node costs. k-best operates by maintaining a list of the k-best frontier nodes under each child of the root node as the subtrees are explored. When the exploration phase is completed, it then calculates an estimate of the expected root-to-leaf path cost for each child of the root, based on the expected minimum path cost of a subtree that only contained the k-best frontier nodes discovered in the child's subtree. These k-best estimates of the expected minimum path costs are then used to make the decision choice (*i.e.,* move to a child with the lowest k-best estimate of the expected minimum root-to-leaf path cost).

6.1.2 How Good is the k-best Approximation?

The obvious question at this point is what is the cost in terms of solution quality of this approximation? For example, what is the worst-case expected difference in the solution quality if we make decisions based on only two frontier nodes.

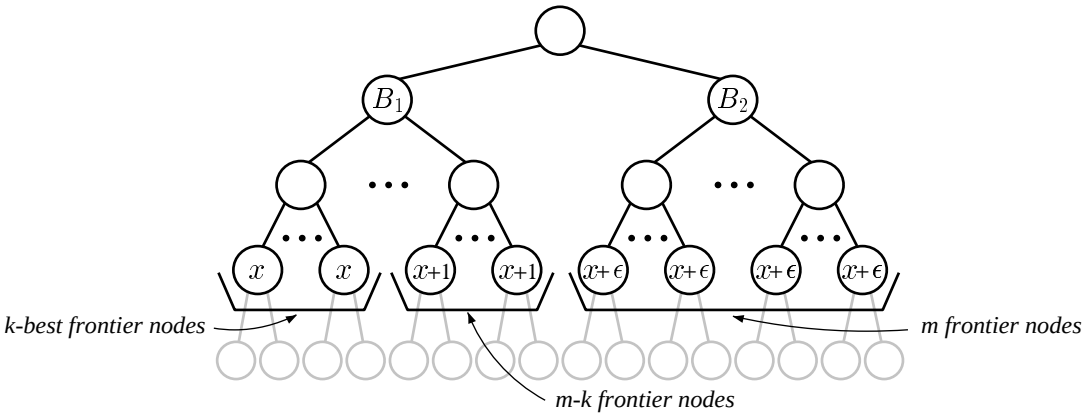


Figure 6.1: Example of the cost of using the k-best approximation to make incremental decisions.

This is analogous to the question raised in Chapter 5 of how much expected error can occur if we use MINIMIN instead of $E(MPC)$. To answer the question, we consider a similar example.

The worst-case expected error for a k-best decision occurs when k-best chooses a decision that only has k good frontier nodes, and all the frontier nodes below the decision that wasn't chosen by k-best are good, in the sense that their node costs are very close to (or perhaps equal to) the minimum frontier-node cost. This is the situation where $E(MPC)$ has the greatest potential advantage over k-best. An example of this situation is shown in Figure 6.1, where ϵ is a small constant.

For this example, the k-leftmost frontier nodes all have cost x, which is the minimum frontier-node cost, and the other $m - k$ frontier nodes below node B_1 have node costs equal to $x + 1$, which is equal to the minimum frontier-node cost plus the maximum edge cost. This means that only these k minimum-cost frontier nodes can contribute to the expected minimum path cost below B_1. We assume that there are m frontier nodes below B_2, all of which have a node cost equal to $x + \epsilon$. We also assume that there is one remaining unexplored level of the tree and that the branching factor is b.

The expected error of a k-best decision is simply the difference between the expected minimum path cost below the k-best decision (B_1), and the expected minimum path cost below the $E(MPC)$ decision (B_2). For simplicity, we assume that the unexplored edge costs are independently chosen from a uniform distribution over the range $[0, 1]$ (*i.e.*, $F(z) = z$ $(0 \leq z \leq 1)$). Since the k minimum-cost frontier nodes are the only frontier nodes that can appear on a minimum-cost path below B_1, the expected minimum path cost below B_1 is equal to x plus the

expected cost of a minimum choice between kb random edge costs. From Chapter 5, we know that the expected cost of a minimum choice between kb random edge costs is $1/(kb+1)$.

In order to calculate the expected minimum path cost below B_2, we simply observe that the optimal decision after moving to B_2 will be a choice between mb edge costs added to one of the m frontier nodes that have cost $x+\epsilon$. In this case, the expected value of mb random choices which is simply $1/(mb+1)$. Thus the expected minimum path cost below B_2 is $x+\epsilon+1/(mb+1)$.

The worst case expected error for a k-best decision, on this example problem, is simply the difference between these two expected minimum path costs.

$$\begin{aligned} E(MPC(B_1)) - E(MPC(B_2)) &= x + \frac{1}{kb+1} - x - \epsilon - \frac{1}{mb+1} \\ &= \frac{1}{kb+1} - \frac{1}{mb+1} - \epsilon \\ &= \frac{(m-k)b}{(kb+1)(mb+1)} - \epsilon \end{aligned}$$

In the limit as m goes to infinity and ϵ goes to zero, this difference approaches $1/(kb+1)$, which is the expected completion cost for the path below a set of k minimum-cost frontier node. This make sense, because as the number of frontier nodes below B_2 increases, it becomes more likely that a zero-cost edge will be generated.

Thus the worst-case possible error for k-best on a last incremental decision problem with an edge-cost distribution of $F(z)=z$ is a decreasing function of k. In addition, the amount of information that is gained by each additional frontier node being considered in the k-best estimate of the expected minimum root-to-leaf path cost, is a decreasing function of the number of nodes already being considered. Intuitively, it makes sense that the most important piece of information about a root-child decision subtree is the minimum frontier-node cost, and the second most important piece of information is the second smallest frontier-node cost, etc.

6.1.3 Exploring to Find the k-best Frontier Nodes

When the objective is to maintain the k-best frontier-node costs, pruning becomes more difficult than in pure (alpha-pruning) DFBnB. For example, we can no longer prune an interior node i simply because its node cost is greater than the current best frontier-node cost in the subtree being explored, because one of its descendants may influence one of the other $k-1$ values in the subtree.

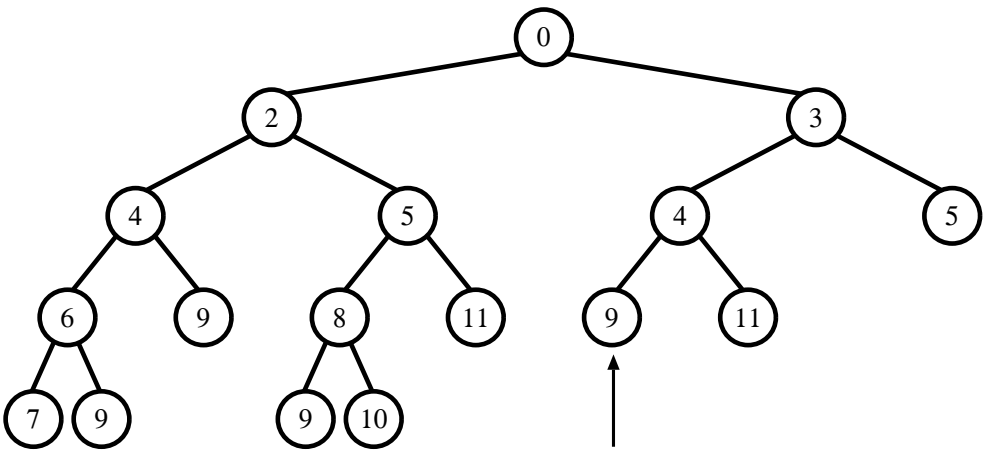
Thus, within a given subtree, we can prune an interior node only if its value exceeds the k^{th} best frontier node value discovered so far. A second problem is that we can no longer use an alpha bound from a previously explored subtree to prune the exploration of the remaining subtrees as in DFBnB. This is also because even if an interior node's cost is greater than the best frontier node cost discovered in another subtree, it could still affect the other $k-1$ nodes in the current subtree. In this case, we can only safely prune using the alpha-bound from a different subtree plus the maximum possible completion cost. For a single unexplored level, this means that we must use the minimum frontier-node cost in the previously explored subtrees, plus the maximum edge cost as the initial bound for pruning the next subtree to be explored.

For example, consider the 2-best exploration shown in Figure 6.2a, that has been interrupted at the point indicated by the arrow. The numbers in the nodes are the node costs, and the children of a node are explored from left to right. First, we observe that the node that has cost 8 was not pruned even though its node cost is less than the minimum node cost at that point (*i.e.,* 7). This is because its node cost was not less than the second-best frontier node cost at that point (*i.e.,* 9). We also observe that, even though the node indicated by the arrow has a node cost that is greater than the 2-best frontier node costs below the first child of the root node, we still need to expand this node because it is possible that a frontier node below it will be the second-best or even the best frontier node. One example of this case is shown in 6.2b.

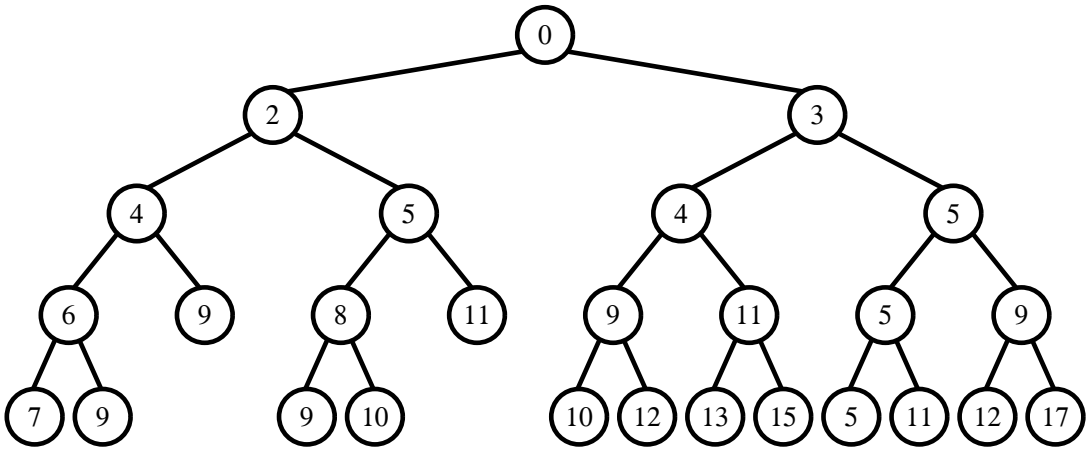
6.1.4 Experimental Comparison of k-best and MINIMIN

Although the worst-case results are important, we are more interested in the average-case performance of k-best when compared with $E(MPC)$ and MINIMIN. In order to compare the performance of k-best and MINIMIN, we performed a new set of experiments on random trees with fixed branching factor, and with edge costs that are independently chosen from the set $\{0, 1/1024, 2/1024, ..., 1023/1024\}$. In order to greatly simplify our implementation of the k-best algorithm, decisions were made based on the k-best frontier nodes at the search-bound depth.

For each trial, random values were assigned to each edge of the problem-space tree except for one unexplored level at the bottom of the tree. Both MINIMIN and k-best were then allowed to explore the problem space up to this unexplored level. Random values were then assigned to the remaining edge costs, and the remainder of the tree was explored, so that the rest of the decisions were optimal. In addition to the k-best and MINIMIN path costs, we also calculated the optimal



(a)



(b)

Figure 6.2: Example of an exploration to find the 2-best frontier nodes, (a) part of the way through the exploration, and (b) an example of the complete exploration.

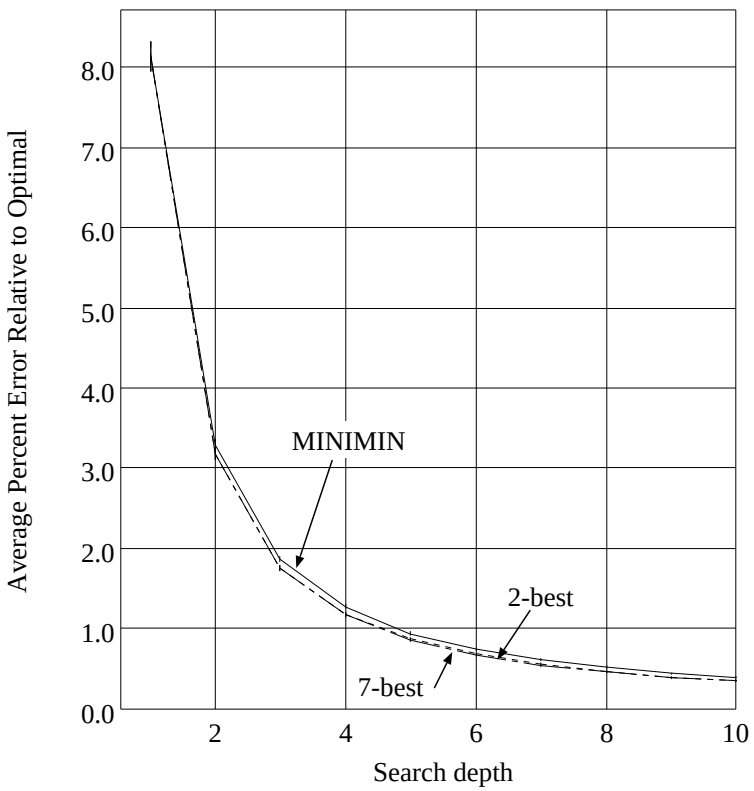


Figure 6.3: Average percent solution cost error relative to optimal versus search depth for MINIMIN and k-best ($k\epsilon\{2,7\}$).

path cost in the problem-space tree. We then recorded the percent solution-cost error, calculated as follows.

$$\%error = 100 * \frac{(algorithm\ solution\ cost\ -\ optimal\ cost)}{optimal\ cost}$$

We also compared the solution cost of the two algorithms, and recorded the number of times that each algorithm produced a lower-cost solution.

The graph in Figure 6.3 shows the average percent error in the solution cost, relative to optimal versus the depth of the search tree (which is one less than the depth of the problem-space tree), for MINIMIN, 2-best and 7-best. The results are averaged over 100,000 trials on binary random trees, and show the 95% confidence intervals. The results show that the *k-best* algorithms produce better quality solutions than MINIMIN when both explore to the same depth.

We have also performed experiments on 3-best, 4-best, 5-best, and 6-best. An example of these results are presented in Figure 6.4 which shows the solution cost error as a function of k for a binary tree of depth 10 (*i.e.*, a search depth of 9).

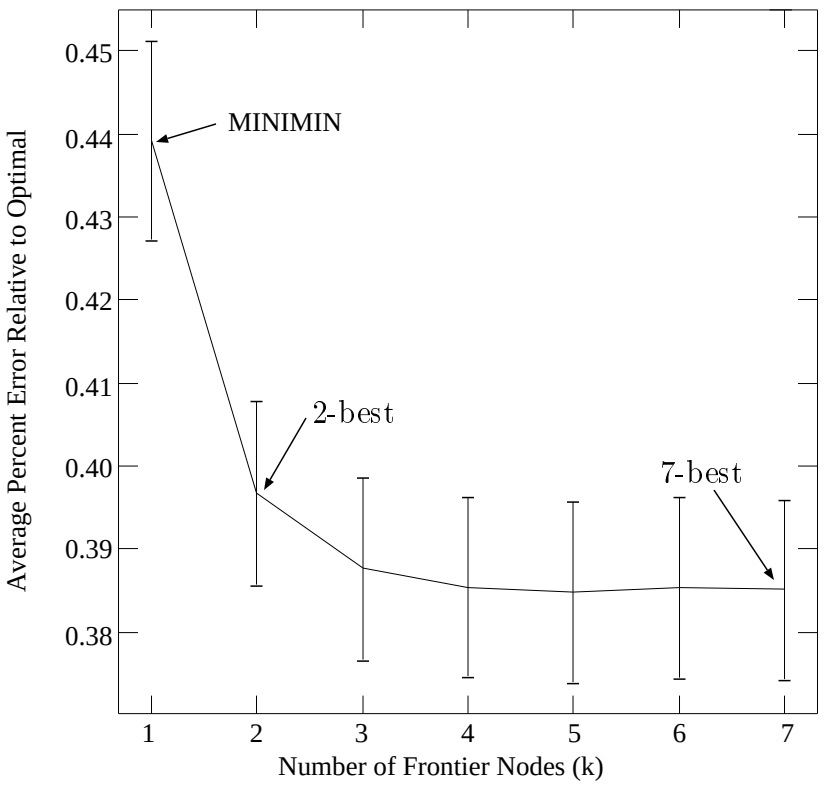


Figure 6.4: Average percent solution cost error relative to optimal versus k for k-best ($k\epsilon\{1,2,3,4,5,6,7\}$).

The 95% confidence limits are as shown in the figure. Initially, the performance of the k-best algorithms improves with k, although for k greater than four, the average solution cost error relative to optimal stays fairly constant as k increases. The results clearly show that the biggest improvement in the solution cost error occurs when we move from MINIMIN to 2-best. This is in line with our result that the second-best frontier-node cost provides the most information after the best frontier-node cost.

Since the k-best average solution-cost error is only a slight improvement over the MINIMIN average solution-cost error, we also calculated the percentage of trials that each algorithm produced a lower-cost solution. When both algorithms make the same decision, neither algorithm wins. The results in Tables 6.1 and 6.2 show that the k-best algorithm consistently outperforms MINIMIN when both are allowed to search to the same depth. For example, 2-best won about 1.4444 times as many head-to-head trials as MINIMIN for a binary tree with depth 10.

Unfortunately, the k-best algorithms typically require many more node generations than MINIMIN to search to the same depth (see Figure 6.5). The ad-

Search depth	2-best % wins	*MINIMIN* % wins	ratio
1	0.0 ± 0.00	0.0 ± 0.00	——
2	1.7 ± 0.08	1.5 ± 0.08	1.1333
3	1.7 ± 0.08	1.4 ± 0.07	1.2142
4	1.6 ± 0.08	1.2 ± 0.07	1.3333
5	1.5 ± 0.08	1.1 ± 0.06	1.3636
6	1.4 ± 0.07	1.0 ± 0.06	1.4000
7	1.4 ± 0.07	1.0 ± 0.06	1.4000
8	1.4 ± 0.07	0.9 ± 0.06	1.5555
9	1.3 ± 0.07	0.9 ± 0.06	1.4444
10	1.3 ± 0.07	0.9 ± 0.06	1.4444

Table 6.1: Percent wins for 2-best versus MINIMIN.

Search depth	7-best % wins	*MINIMIN* % wins	ratio
1	0.0 ± 0.00	0.0 ± 0.00	——
2	1.7 ± 0.08	1.5 ± 0.08	1.1333
3	2.0 ± 0.09	1.7 ± 0.08	1.1764
4	2.0 ± 0.09	1.6 ± 0.08	1.2500
5	1.9 ± 0.09	1.5 ± 0.08	1.2666
6	2.0 ± 0.09	1.4 ± 0.07	1.4285
7	1.9 ± 0.08	1.4 ± 0.07	1.3571
8	1.9 ± 0.08	1.4 ± 0.07	1.3571
9	1.8 ± 0.08	1.4 ± 0.07	1.2857
10	1.8 ± 0.08	1.3 ± 0.07	1.3846

Table 6.2: Percent wins for 7-best versus MINIMIN.

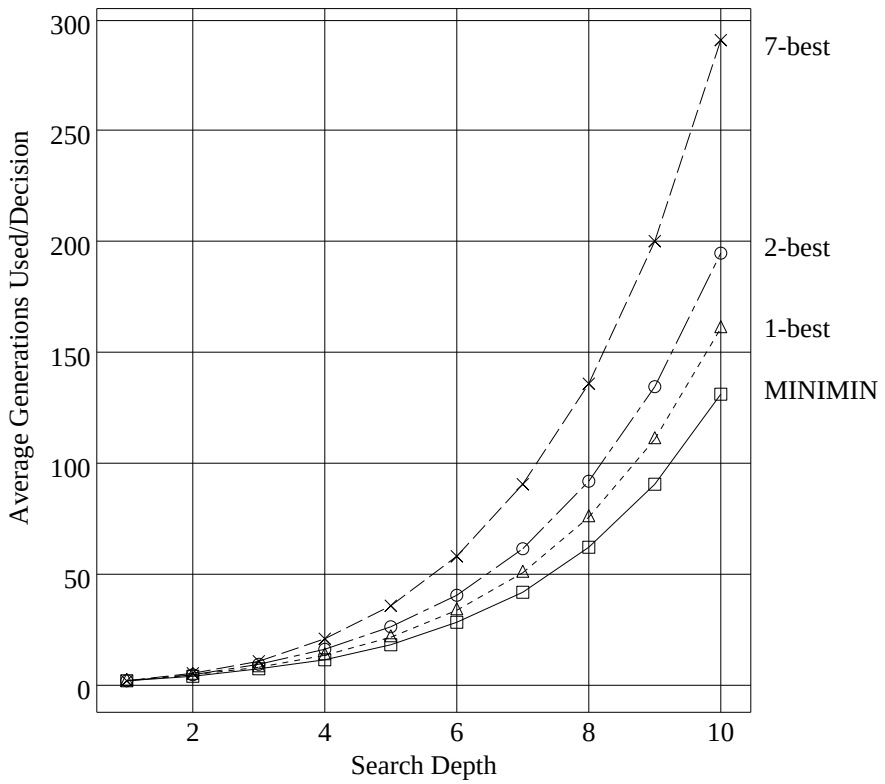


Figure 6.5: Average generations per decision versus search depth for MINIMIN and $E(MPC, k\epsilon\{1,2,7\})$.

ditional node generations required by the *k-best* algorithms result from the need to know the k-best frontier nodes under each child of the root. Recall that MINIMIN only needs to know the best frontier node under the root. In fact, just finding the best frontier node under each child (labelled 1-best in Figure 6.5) requires more node generations than MINIMIN. 1-best is shown for comparison, although there is no advantage to performing 1-best instead of MINIMIN.

To further demonstrate this problem, we graphed the average percent error versus the number of node generations used by each algorithm averaged over the set of trials for a given search depth. The results in Figure 6.6 show that, for a given average number of node generations, MINIMIN produces a lower expected error than either 2-best or 7-best. From these results we can conclude that MINIMIN makes better quality real-time decisions on average when compared with k-best because of its exploration efficiency. We will report on experiments where the computation bound is the number of available node generations in Chapter 7. The difference is that in this chapter we allow the depth-first branch-and-bound exploration to finish it exploration to a given search depth, whereas, when the

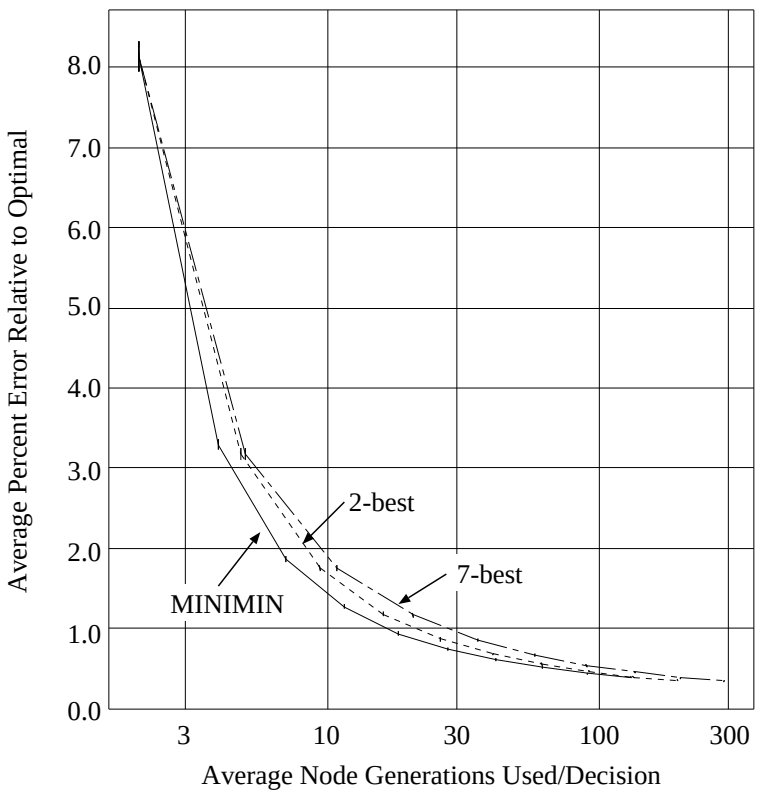


Figure 6.6: Average percent error versus average node generations ($b = 2$).

node generations are bounded instead of the search depth, it is very likely that a depth-first branch-and-bound exploration will run out of node generations in the middle of exploring the tree to a particular search depth.

6.1.5 Discussion

The results presented in this section show that it is possible to improve upon the MINIMIN decision method, when the exploration is only limited in depth, by approximating the expected minimum-cost path algorithm on larger search trees. Although the improvement is small, the hope is that this improvement will accumulate over a sequence of decisions. The drawback of k-best is that the number of node generations required to maintain the k-best frontier node-costs far exceeds the node generations required by MINIMIN. In fact, the improvement gained by spending the time associated with the additional node generations used to support k-best decision, over a MINIMIN decision with the same search tree, is sometimes much smaller than the average improvement that could be gained by MINIMIN if it just spent the time associated with these additional node

generations to look one level deeper. For example, in Figure 6.5 we observe that MINIMIN requires about 90 node generations on average to search to depth 9, whereas 7-best requires about the same number of node generations on average to search to depth 7. From Figure 6.3, we can further observe that MINIMIN's average solution cost error for a depth 9 search is lower than 7-best's average error for a depth 7 search. Thus, in order for k-best to be competitive with MINIMIN, we must find a way to reduce the number of generations that it uses to explore to a given search depth. In the next section, we present alpha-bound k-best (αk-best) which is a more efficient approximation of $E(MPC)$.

6.2 Alpha-bound k-best

In this section, we present alpha-bound k-best, which is a more efficient approximation to the $E(MPC)$ approach to real-time decision making.

6.2.1 Description of the Alpha-Bound k-best Algorithm

The idea behind alpha-bound k-best, or simply αk-best, is to explore the problem space using a single-bound depth-first branch-and-bound search. During the exploration process, the k-best frontier nodes that are *generated* or *seen* are maintained as before. These k frontier-node costs are then used to estimate the actual k-best frontier-node costs in the equation for $E(MPC|k)$. Since αk-best uses a single-bound depth-first branch-and-bound search, on average it will use the same number of node generations as MINIMIN for the same search depth bound. A small difference in the actual number of node generations used will only occur when αk-best produces a different sequence of decisions than MINIMIN.

The basic difference between αk-best and plain k-best is in the way that the exploration is performed. Recall that k-best assumes that the exploration will find the k-best frontier-node costs at a given depth bound in each subtree of the root node. We earlier observed that pruning is significantly reduced when we need to find more than just the best frontier node. αk-best explores the problem space using a single alpha bound to prune the search tree in the same manner as MINIMIN. As the search is performed, the k-best frontier-node costs *seen* under each subtree are recorded. These k-best *seen* frontier-node costs are then used to approximate the actual k-best frontier-node costs in the equations used to calculate $E(MPC|k)$.

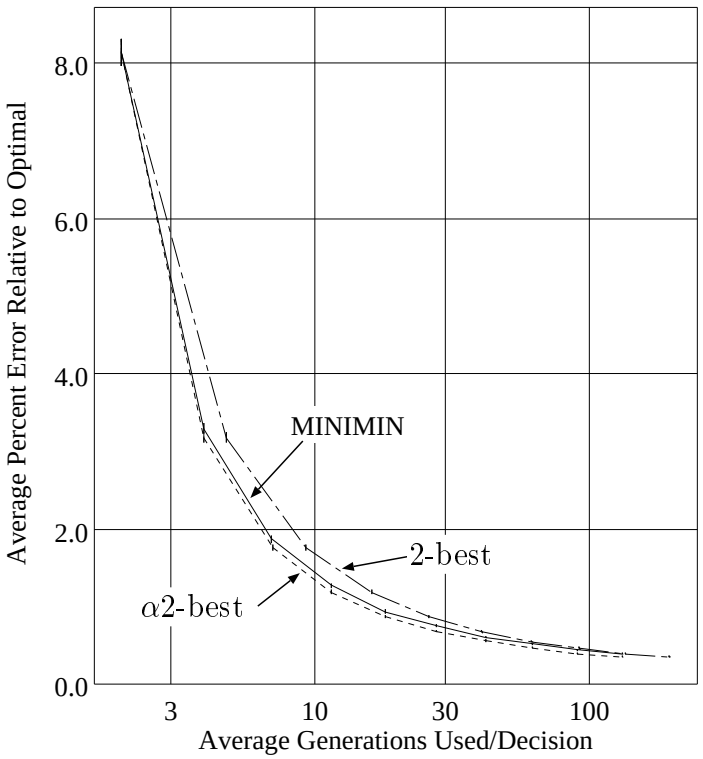


Figure 6.7: Average percent error versus average node generations ($b = 2$).

6.2.2 Results Comparing αk-best with MINIMIN

In order to compare the average-case performance of αk-best with MINIMIN, we repeated the random-tree experiments of the previous section. As before, in order to simplify our implementation of the k-best algorithm, decisions were made based on the k-best frontier nodes seen at the search-bound depth. The results are averaged over 100,000 random trees with fixed branching factor ($b = 2$), and with edge costs that are independently chosen from the set $\{0, 1/r, 2/r, ..., (r-1)/r; r = 1024\}$.

6.2.3 Discussion

Figure 6.7 shows the average percent solution-cost error versus average generations used per decision for α2-best, MINIMIN and 2-best. The generations used per decision are averaged over the set of trials for a given exploration search depth. The results show that although 2-best is less efficient than MINIMIN, α2-best is more efficient, and thus produces solutions with lower average percent error, using roughly the same average number of node generations per decision

Search depth	α2-best % wins	*MINIMIN* % wins	ratio
1	0.0 $\pm$ 0.000	0.0 $\pm$ 0.000	—
2	1.7 $\pm$ 0.079	1.5 $\pm$ 0.074	1.133
3	1.7 $\pm$ 0.080	1.4 $\pm$ 0.072	1.214
4	1.6 $\pm$ 0.079	1.2 $\pm$ 0.068	1.333
5	1.5 $\pm$ 0.076	1.2 $\pm$ 0.067	1.250
6	1.5 $\pm$ 0.075	1.1 $\pm$ 0.066	1.363
7	1.4 $\pm$ 0.073	1.1 $\pm$ 0.064	1.272
8	1.4 $\pm$ 0.073	1.1 $\pm$ 0.064	1.272
9	1.4 $\pm$ 0.072	1.0 $\pm$ 0.062	1.400
10	1.4 $\pm$ 0.072	0.9 $\pm$ 0.060	1.555

Table 6.3: Percent wins for α2-best versus MINIMIN.

as MINIMIN.

In addition, the results in Tables 6.3 and 6.4 show that both α2-best and α7-best win more of the head-to-head competitions than MINIMIN for a given search depth. Although the percentage wins by αk-best is only slightly higher than the percentage wins by MINIMIN, this difference is consistent.

Thus, with this series of approximations, we finally have a decision algorithm that can use additional frontier node-cost information to make slightly better quality decisions on average than MINIMIN, when computation time is measured in the number of node generations per decision.

At this point, we observe that there is some additional overhead in the αk-best exploration phase needed to maintain the k-best frontier-node costs seen. Since k is currently restricted to being less than or equal to 7, the amount of overhead is bounded by a constant amount of computation per frontier node generation. There is also some overhead in the function used to calculate the αk-best decision given the αk-best frontier node costs. Again, this overhead is bounded by a small constant.

Search depth	α7-best % wins	*MINIMIN* % wins	ratio
1	0.0 $\pm$ 0.000	0.0 $\pm$ 0.000	—
2	1.7 $\pm$ 0.079	1.5 $\pm$ 0.074	1.133
3	1.9 $\pm$ 0.085	1.7 $\pm$ 0.079	1.117
4	2.0 $\pm$ 0.086	1.6 $\pm$ 0.077	1.250
5	1.9 $\pm$ 0.084	1.5 $\pm$ 0.076	1.266
6	1.9 $\pm$ 0.084	1.5 $\pm$ 0.076	1.266
7	1.8 $\pm$ 0.083	1.5 $\pm$ 0.075	1.200
8	1.8 $\pm$ 0.083	1.5 $\pm$ 0.074	1.200
9	1.8 $\pm$ 0.083	1.4 $\pm$ 0.074	1.285
10	1.8 $\pm$ 0.082	1.4 $\pm$ 0.073	1.285

Table 6.4: Percent wins for α7-best versus MINIMIN.

Up to this point, we have only focussed on the last incremental decision problem. Now that we have a new decision method that consistently produces lower-cost solutions on average than MINIMIN, while using the same amount of computation, we can move on to the problem of making a sequence of incremental decisions, which is the subject of the next chapter. Our hope is that k-best's slight improvement over MINIMIN will accumulate over a sequence of decisions.

CHAPTER 7

Real-Time Search: Making a Sequence of Incremental Decisions

In the first part of this chapter, we present our approach to the problem of making a sequence of incremental decisions. Our basic idea is to treat the general incremental decision as if it were a last incremental decision. We assume for this discussion that the exploration is a fixed-depth branch-and-bound search. Our experimental results show that on average αk-best slightly outperforms MINIMIN on problems that involve a sequence of incremental decisions, and that this average performance improvement increases with the number of decisions in the sequence.

In the second part of this chapter, we relax the fixed-depth exploration assumption, and consider the case where the computation per decision is limited by the number of node generations allowed. In this case, we address the problem of combining exploration and decision-making algorithms to create real-time search algorithms. We first review the depth-first branch-and-bound MINIMIN algorithm, which we use as a benchmark. We then present αk-best BnB, which explores the problem space using a single-bound branch-and-bound search, and makes decisions using the αk-best approximation to $E(MPC)$. Next, we present algorithms based on obvious yet naive best-first search methods that use the same heuristic for both exploration and decision making. We show that using the same heuristic for both exploration and decision making can lead to either a *swap-pathology* or uneven exploration or both. Next, we present two new best-first search algorithms: *hybrid* best-first search and αk-best best-first search. Both of these best-first algorithms use an evaluation function to make the incremental decisions that is different from the evaluation function used to explore the problem space. The last two sections are concerned with evaluating the relative performance of these real-time search algorithms on random-tree problems.

7.1 Approximating Each Decision as if it Were the Last

For a sequence of incremental decisions, in general we won't be able to see the bottom of the problem-space tree before the next decision. In this case, the completion-cost distribution that we developed in Chapter 5 no longer accurately reflects the situation that will exist for the next decision. Instead, the distribution of minimum path costs that might be traversed below a child of the root will depend on the decision-making algorithm, the exploration algorithm, and the structure of the search tree, as we previously discussed in Section 4.1.

For this research, we have assumed that it is reasonable to model the general incremental decision problem as each decision were a last incremental decision problem. The justification for this is two-fold. First, we have already argued that the last incremental decision problem is sufficiently difficult that we have little hope of adding more information to the decision process (*e.g.,* such as the effect of the decision rule on the expected decision quality of future decisions), without further complicating the task. The second argument is that, for many real-time problems, the average depth of the search tree that can be explored per decision will be fairly constant. Thus, each new decision is based on an exploration whose frontier nodes are one level deeper than the frontier nodes used to make the previous decision.

A MINIMIN decision is to move one step toward a minimum-cost frontier node in the search tree. This decision rule is optimal when the minimum-cost frontier node is a leaf node, or equivalently MINIMIN decisions are based on the assumption that a frontier node's cost is an accurate evaluation of the minimum leaf-node cost below that frontier node. For k-best, we take this one step further by assuming that the distribution of minimum path costs to one level below the frontier nodes is an accurate representation of the distribution of minimum path costs in the unexplored part of the problem space.

7.2 Comparison of MINIMIN and αk-best

In this section, we present experimental results that compare the performance of MINIMIN and αk-best on random trees over a sequence of decisions. These results are for the case where the exploration is limited by a search depth bound (*i.e.*, a fixed lookahead depth for each decision). Later in this chapter, we present real-time search algorithms where the exploration is instead limited to a fixed number of node generations per decision. Note that MINIMIN and αk-best generate the same number of nodes per decision on average, since both decisions are

based on a single-bound branch-and-bound exploration.

In order to compare the performance of αk-best to MINIMIN for a sequence of decisions, we performed the following set of experiments. Each trial consisted of calculating the optimal solution cost, and the MINIMIN and αk-best solution cost, over the sequence of decisions necessary to generate a complete solution, given a specific search-depth bound. The percent error relative to optimal $(100 * (solution\ cost - optimal)/optimal)$ was then calculated for both MINIMIN and αk-best. If the solution-path cost produced by MINIMIN was lower than the solution-path cost produced by αk-best, then MINIMIN won that trial. If instead αk-best produced a lower-cost solution than MINIMIN, then αk-best won. Otherwise the result was a tie. The results were averaged over 10,000 trials unless otherwise noted.

The results in Figure 7.1 show the percent error relative to optimal versus search depth for depth 15 binary search trees for MINIMIN, and for both $\alpha 2$-best and $\alpha 7$-best. From these results, we observe that the αk-best algorithms produce solutions whose percent error is on average slightly lower than the percent error of the solutions produced by MINIMIN (the 95% confidence limits are shown in the figure). In addition, we observe that there is almost no difference between the average percent error from using the 2-best frontier nodes seen, versus using the 7-best frontier nodes seen. This suggests that a very large fraction of the improvement of αk-best over MINIMIN is due to the information from the second-best frontier node. We also note that the relative difference between MINIMIN and the αk-best algorithms increases slightly with the search depth.

The results from experiments on trees with larger branching factors are qualitatively similar to the results for binary trees. One notable difference is that the percent error relative to optimal for a given search depth increases with the branching factor. One reason for this may be that an increase in the number of choices means that any myopic decision strategy (*e.g.,* MINIMIN or αk-best) has more opportunity to make a suboptimal choice.

In order to further compare the performance of MINIMIN and αk-best, we also recorded the percentage of the time that each algorithm produced a lower-cost solution path (*i.e.*, it won the head-to-head competition). Tables 7.1 and 7.2 contain the percentage of the head-to-head trials that $\alpha 2$-best and $\alpha 7$-best produced a lower-cost solution than MINIMIN, and the percentage of the trials that MINIMIN produced a lower-cost solution, for depth 15 binary trees (ties are counted as trials, but not counted as wins for either algorithm). The results clearly show that both αk-best algorithms produces a lower-cost solution than MINIMIN a larger fraction of the time than MINIMIN produces a lower-cost

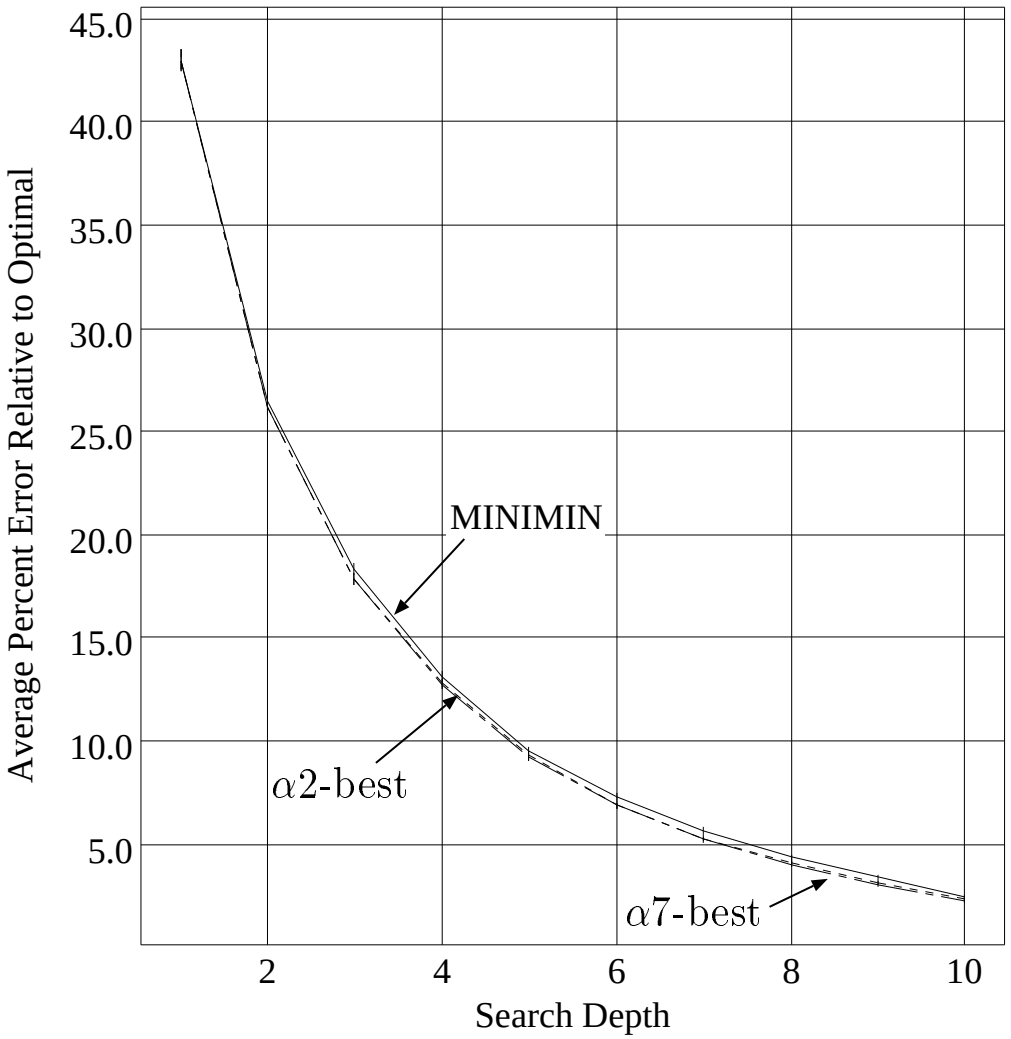


Figure 7.1: Average percent error relative to optimal versus search depth for MINIMIN, α2-best and α7-best.

Search depth	α2-best % wins	*MINIMIN* % wins	ratio
1	0.0 $\pm$ 0.00	0.0 $\pm$ 0.00	—
2	16.2 $\pm$ 0.72	15.4 $\pm$ 0.71	1.051
3	14.5 $\pm$ 0.69	12.2 $\pm$ 0.64	1.188
4	11.9 $\pm$ 0.63	9.8 $\pm$ 0.58	1.214
5	9.8 $\pm$ 0.58	8.9 $\pm$ 0.56	1.101
6	9.1 $\pm$ 0.56	6.8 $\pm$ 0.49	1.338
7	7.8 $\pm$ 0.53	5.3 $\pm$ 0.44	1.471
8	6.8 $\pm$ 0.49	4.8 $\pm$ 0.42	1.416
9	5.8 $\pm$ 0.46	3.8 $\pm$ 0.37	1.526
10	4.8 $\pm$ 0.42	3.6 $\pm$ 0.37	1.333

Table 7.1: Percent wins over 15 decisions for α2-best versus MINIMIN.

solution. Our results for trees with larger branching factors were qualitatively the same as the results in Figure 7.1 and Tables 7.1 and 7.2.

The last set of data in Figure 7.2 shows the percent wins (MINIMIN against α7-best) versus the log of the tree depth for a fixed search depth of 5 on a binary tree. The results show that as the number of decisions increases (*i.e.*, the tree depth), the percentage of the trials that each algorithm wins initially grows. This makes sense because the number of opportunities for the algorithms to make a different choice increases with the number of decisions, thus we would expect the number of ties to be a decreasing function of the number of decisions. What we didn't expect is that beyond a depth of about 300 decisions, the percentage of the trials in which αk-best produces a lower-cost solution than MINIMIN continues to grow, at the expense of the percentage of the trials in which MINIMIN produces a lower-cost solution. This clearly shows that αk-best makes better quality decisions on average than MINIMIN over a long sequence of decisions.

Search depth	α7-best % wins	*MINIMIN* % wins	ratio
1	0.0 $\pm$ 0.00	0.0 $\pm$ 0.00	—
2	16.2 $\pm$ 0.72	15.4 $\pm$ 0.71	1.051
3	16.3 $\pm$ 0.72	13.9 $\pm$ 0.68	1.172
4	14.0 $\pm$ 0.68	12.1 $\pm$ 0.64	1.157
5	12.4 $\pm$ 0.65	10.9 $\pm$ 0.61	1.137
6	11.3 $\pm$ 0.62	8.8 $\pm$ 0.56	1.284
7	9.9 $\pm$ 0.58	7.0 $\pm$ 0.50	1.414
8	8.7 $\pm$ 0.55	6.3 $\pm$ 0.48	1.380
9	7.5 $\pm$ 0.52	5.1 $\pm$ 0.43	1.470
10	6.6 $\pm$ 0.48	4.8 $\pm$ 0.42	1.375

Table 7.2: Percent wins over 15 decisions for α7-best versus MINIMIN.

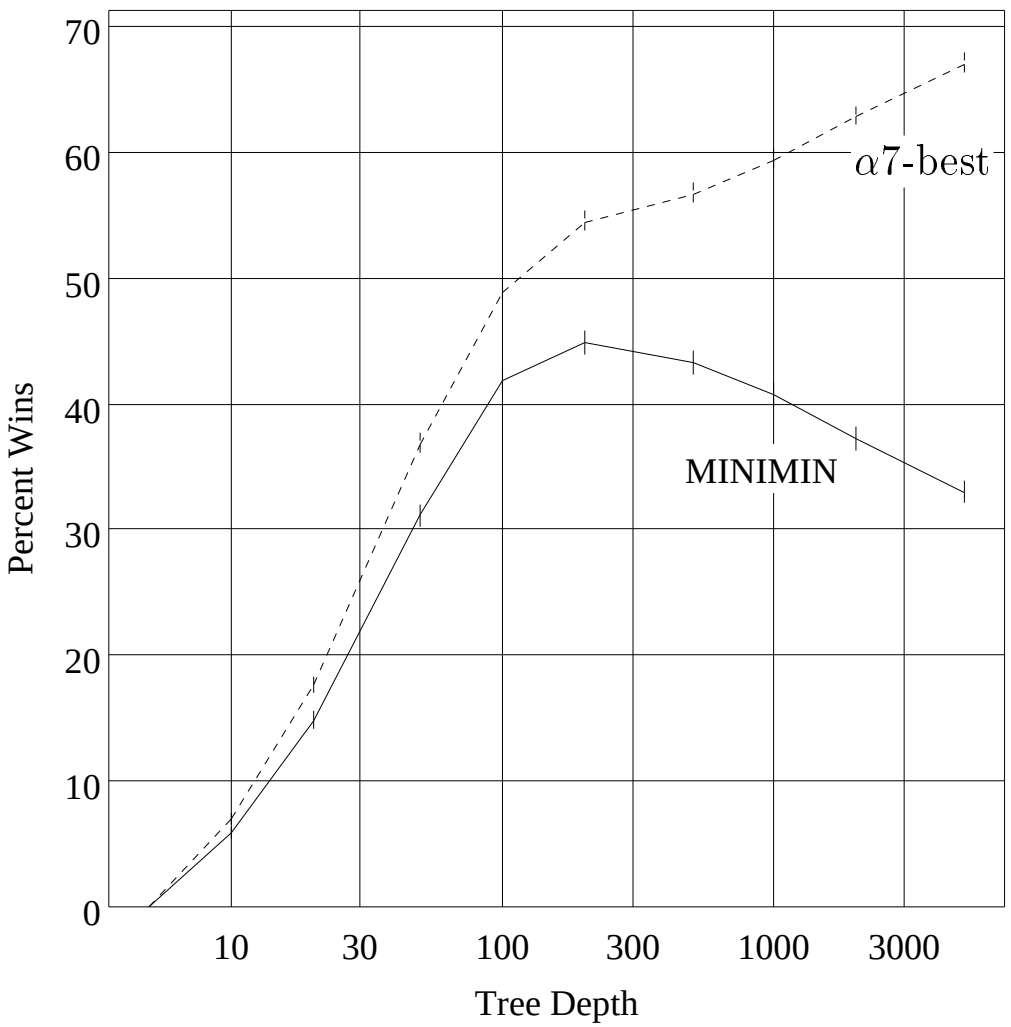


Figure 7.2: Percent wins by α7-best and MINIMIN versus tree depth.

7.3 Real-Time Search Algorithms

In this section, we relax the assumption that the exploration is bounded by the lookahead depth, and instead allow the real-time search algorithms to generate a finite number of nodes per decision. We first discuss algorithms that use a depth-first branch-and-bound exploration. Next, we discuss a set of algorithms that use a best-first exploration. Finally, we present experimental results that compare the performance of these real-time search algorithms on random-tree problems.

For the real-time search algorithms that we present, we assume that there is sufficient memory to store the explored search tree. When memory is a concern, there are several existing methods for searching with limited memory that are briefly discussed in Chapter 11. We also assume that the cost of initially generating a node far exceeds the cost of revisiting a node in a subsequent iteration, thus we ignore the node regeneration overhead cost. The initial search depth bound is calculated based on the worst-case node generation requirement (*i.e.,* the deepest that depth-first branch-and-bound can search when no pruning occurs), and the stored values from the previous iteration are then used to order the exploration in the current iteration. The stored values make it possible to find the minimum-cost frontier node in the next level of the tree very efficiently, which often means that very few additional node generations are needed to extend the search depth by one level.

7.3.1 Real-Time Branch-and-Bound Algorithms

In this section, we describe two real-time search algorithms (iterative-deepening MINIMIN and αk-best branch-and-bound) that use iterative-deepening depth-first branch-and-bound to explore the problem space.

7.3.1.1 Iterative-deepening MINIMIN

Perhaps the simplest real-time search algorithm results when we combine a depth-first branch-and-bound (with a depth bound) exploration, with MINIMIN decision making. We call this real-time algorithm MINIMIN depth-first branch-and-bound (MINIMIN-BnB). In a real problem domain, we typically won't know in advance what depth bound to use, so we have adopted an iterative-deepening approach to exploration. Iterative-deepening depth-first branch-and-bound performs a depth-limited branch-and-bound exploration where the depth bound is initially a small value. The depth bound is increased by one after each iteration

is completed, and a new iteration is performed until the available computation time is used up. By iteratively increasing the depth bound, it gradually explores more and more of the problem space until the available computation for a given decision is exhausted. When time runs out, the MINIMIN decision is based on the expected cost of the frontier nodes as discussed in Chapter 5. The basic idea is to take advantage of the information in the partially explored level of the tree by estimating how much the cost of the frontier nodes from the last iteration, that have not been processed in the current iteration, would be increased if this iteration of the exploration were allowed to complete. This is accomplished by adding the expected cost of a single greedy decision, times the number of steps from the frontier node to the depth of the deepest frontier node, to the node cost of the frontier node. This allows the MINIMIN decision to include information from the last exploration level, even though it has not been completed. Thus, iterative-deepening MINIMIN (*idMINIMIN*) combines iterative-deepening exploration, and expected-cost frontier-node evaluation, with MINIMIN decision making.

7.3.1.2 αk-best Branch-and-Bound

We have also considered a real-time search algorithm based on the αk-best decision algorithm described in Chapter 6. αk-best BnB uses iterative-deepening depth-first branch-and-bound exploration with one simple modification. Instead of just keeping track of the minimum-cost frontier node, the exploration maintains the k-best frontier-node costs seen under each child of the root during the exploration.

Our previous description of the αk-best decision method relied on the problem space being explored with a depth bound on the computation. With iterative-deepening exploration, it will usually be the case that the last level of the search tree is not completely explored. We have considered three ways to address the problem of what to do with this additional but incomplete information. One way is to simply ignore it. In this case, the move decision is based on the information from the last completed level of exploration. The problem with this approach is that the information contained in the partially explored level is often very useful, as shown in our experiments with *ec-MINIMIN* and *lc-MINIMIN* in Chapter 5.

A second way to use this information is to calculate the $E(MPC|k)$ equations for frontier nodes at different levels of the search tree, and then estimate the expected value of the current decisions using a combination of these expected value equations. The problem with this approach is two-fold. First, the equations for the effect of frontier nodes at different levels of the search tree on the

distribution of complete solution costs at a child of the root node are more difficult to calculate, the more unexplored levels we need to consider (each additional unexplored level requires us to perform an additional convolution integral). The second problem is that if frontier nodes that affect the decision are allowed to be at different levels of the search tree, then the exploration will be complicated because we cannot immediately determine which frontier nodes are the k best. For these reasons, we have not considered this approach any further.

The last approach, which we have adopted for our experiments, employs the same method we used to adapt MINIMIN. Namely, we maintain an estimate of the expected cost of a path through each frontier node rather than the node cost of each frontier node. This estimated cost is calculated as the node cost plus a constant times the remaining depth in the problem space tree. If the remaining depth is not known, then a sufficiently large depth is chosen arbitrarily (*i.e.,* a depth that is greater than the deepest frontier node). The constant is an estimate of the average cost per decision. For our experiments, we estimated the average cost per decision as the expected cost of a single greedy decision. For trees with fixed branching factor b and independent edge costs uniformly distributed over $[0,1]$, the expected cost of a greedy decision is $1/(b+1)$. Alternatively, we could calculate the expected minimum root-to-leaf path cost equations for both one and two unexplored levels, and keep track of the depth of the frontier nodes seen so that we could combine the frontier nodes from different iterations depending on their depth. Although this might work better, this approach is rather cumbersome, and thus we have not considered it further. In summary, what we have done is to apply the same iterative-deepening patch to αk-best BnB that we applied to idMINIMIN.

Decisions are made by combining the k-best estimated frontier-node costs from the iterative-deepening exploration using the equations for $E(MPC|k)$. The decision is to move to the child that has the lowest k-best estimate of the expected minimum root-to-leaf-path cost. An exception to this decision method is when the minimum-cost frontier node is also a leaf node, in which case an optimal completion path has been found. In this case the remaining decisions are made along this path. The basic idea behind this algorithm is to extract additional information out of the same search tree generated by a traditional depth-first branch-and-bound. The main problem with this approach is that it requires us to have a model of how the edge costs of the tree are generated.

7.3.2 Real-Time Traditional Best-First Algorithms

In this section, we describe the two best-first algorithms that result when best-first exploration is combined with a traditional interpretation of MINIMIN decision making.

Traditional best-first search (BFS) expands nodes in increasing order of cost, always expanding next a frontier node on a lowest-cost path. This typically involves maintaining a heap of nodes to be expanded. The obvious way to apply BFS to a real-time search problem is to explore the problem space using a cost function to order the exploration, and when the available computation time is spent, use the same cost function to make the move decision. This single-heuristic cost function approach has also been suggested by Russell and Wefald [RW91] for use in real-time search algorithms (*e.g., DTA**). The motivation for using a single heuristic for exploration and decision making comes from a combination of their *meta-greedy assumption* and their *single-step assumption*. Their *meta-greedy assumption* is that a good meta-level policy is to choose the node exploration that has the highest immediate benefit with respect to the current decision. Their *single-step assumption* is that the value of a node expansion can be estimated by assuming that it is the last computation before a move decision. When the node costs are monotonic nondecreasing, then expanding a node can only increase its value, so the only node expansion that could have any effect on the current decision is that of the frontier node at the end of the current best decision path. This is because the MINIMIN decision is to move one step toward the minimum-cost frontier node, and the only way to change this with monotonic and nondecreasing node costs is to expand the frontier nodes under the current MINIMIN decision. This implies that the exploration strategy and decision maker should use the same heuristic function, and that the purpose of the exploration strategy is to try to *change* the decision.

7.3.2.1 Node-Cost Best-First Search

Node-cost best-first search (*node-cost BFS*) is perhaps the simplest best-first real-time search algorithm. It uses the node cost of frontier nodes to order the exploration and also to make decisions. Recall that the node cost is simply the sum of the edge costs along the path to the root node. *Node-cost BFS* explores the problem space by expanding nodes in order of the cost of the path from the root to the node. When time runs out, a decision is made to move toward a minimum-cost frontier node. In the event that there is more than one minimum-cost frontier node, a tie-breaking rule (*e.g.,* deepest minimum-cost frontier node)

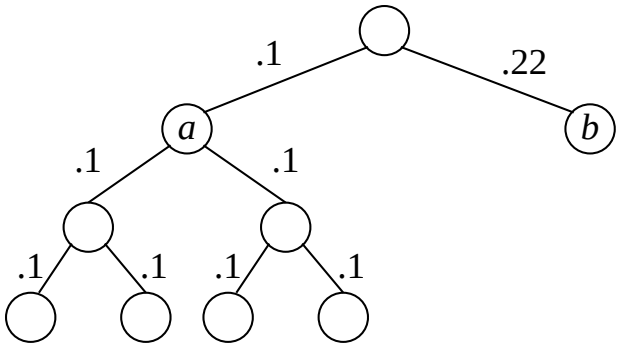


Figure 7.3: Example of best-first *swap-pathology*.

is used to choose between the minimum-cost frontier nodes. After the decision is made, the search tree is pruned to reflect the decision in preparation for the next phase of exploration.

One problem with *node-cost BFS* is that, when the node costs are monotonic and non-decreasing, the algorithm will eventually swap away from a good decision, simply because it has been explored more deeply. Consider the tree in Figure 7.3. *Node-cost BFS* will first explore the paths below node a until the path cost equals 0.3. At this point, the best-cost path (and decision) swaps to node b. If the computation time runs out before the nodes labeled x and y are generated, then *node-cost BFS* will move to node b instead of node a, even though the expected cost of a path through node a is the lowest (*i.e.,* for a uniform $[0, 1]$ edge cost distribution and binary tree). Over the course of a long exploration of a binary problem-space tree, *node-cost BFS* may swap back and forth between the two children of the root many times. This behavior, which we refer to as the best-first *swap-pathology*, is a result of directly comparing node costs of frontier nodes at different depths in the search tree.

7.3.2.2 Estimated-Cost Best-First Search

In retrospect, comparing the node costs of frontier nodes at different depths in the search tree is obviously not the best way to make real-time decisions. As an alternative to *node-cost BFS*, we considered *estimated-cost BFS* which uses an estimate of the expected total root-to-leaf path cost both to order the exploration and to make decisions, in order to better compare the value of frontier nodes at different depths. The estimated total cost of a root-to-leaf path through a frontier node x (or simply the estimated cost of x) can be expressed as the sum of the node cost of x, plus a constant c times the remaining path length,

$f(x) = node_cost(x) + c \cdot (tree_depth - depth(x))$, where c is the expected cost per decision of the path from a frontier node to a leaf node. This heuristic is only admissible when $c = 0$. In general, we don't know or can't calculate an exact value for c, so it must somehow be estimated (*e.g.,* c is the expected cost of a one-level greedy decision).

This *estimated-cost* heuristic function, which is used to estimate the cost of a path from a frontier to a leaf node, should not be confused with the $E(MPC)$-based decision methods, which combine the distributions of path costs to find the expected cost of a root-to-leaf path through a child of the current decision node. The main difference is that the $E(MPC)$ decision method includes the fact that future decisions will be made that will affect which frontier node will eventually be traversed. In some sense, though, MINIMIN decisions based on the *estimated-cost* heuristic function can be viewed as an approximation of the $E(MPC)$ decision method. In particular, when the minimum-cost frontier node is very likely to be chosen by the problem solver when it gets to that level of the tree, then the MINIMIN decision is a reasonable approximation of $E(MPC)$.

The problem with *estimated-cost BFS* is that when the exploration heuristic is non-monotonic, the exploration will stay focussed on any path it discovers with a non-increasing estimated-cost value. The result is often a very unbalanced search tree with some paths explored very deeply and others not explored at all.

For example, consider the tree in Figure 7.4. If $c > .3$, then *estimated-cost BFS* will first expand the root node, generating nodes a and b. If we assume that the tree has a depth of 4, then the estimated cost of node a and b can be calculated as follows.

$$\begin{aligned}
\hat{E}(total_cost(a)) &= node_cost(a) + c \cdot (tree_depth - depth(a)) \\
&= (.1) + .3 \cdot (4 - 1) \\
&= 1.0
\end{aligned}$$

$$\begin{aligned}
\hat{E}(total_cost(b)) &= node_cost(b) + c \cdot (tree_depth - depth(b)) \\
&= (.2) + .3 \cdot (4 - 1) \\
&= 1.1
\end{aligned}$$

Since node a has the lowest estimated cost, it will be expanded next. We can similarly calculate the estimated cost of frontier nodes w and x as follows.

$$\begin{aligned}
\hat{E}(total_cost(w)) &= \hat{E}(total_cost(x)) \\
&= node_cost(w) + c \cdot (tree_depth - depth(w)) \\
&= (.1 + .3) + .3 \cdot (4 - 2) \\
&= 1.0
\end{aligned}$$

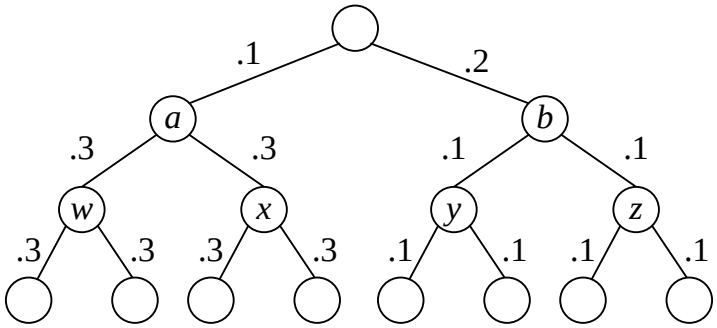


Figure 7.4: Example of uneven exploration problem.

Since both w and x have a lower estimated cost than b, they will both be expanded before b. At this point, we note that as along as there is a path below a whose estimated total cost is less than the estimated total cost of node b, we will fail to generate the nodes labeled y and z. In practice, we observe that exploration based on the estimated total cost often reaches the leaf nodes of one subtree before it has performed much exploration on the other subtrees of the root node. This results in a decision at the root node that is based on a very unbalanced search tree.

7.3.3 Alternative Real-Time Best-First Algorithms

In this section, we present two new best-first algorithms (*hybrid BFS* and k-best BFS) that are improvements over the more traditional best-first exploration algorithms. They are both based on the idea of using a separate function to control the exploration and decision making.

7.3.3.1 Hybrid Best-First Search

One approach to the real-time decision-making problem is to adapt the best-first method so that it avoids the pathological behaviors described above. The main idea behind our approach is to use a different heuristic function for the exploration and decision-making tasks.

Our first new real-time search algorithm, called hybrid best-first search (*hybrid BFS*), avoids the pathological behaviors of a single-heuristic best-first search by combining the exploration heuristic of *node-cost BFS* with the decision heuristic of *estimated-cost BFS*. The intuition behind *hybrid BFS* is that the node-cost exploration will be more balanced than the estimated-cost exploration, while the estimated-cost decision heuristic will avoid the swap pathology by comparing the

estimated total costs of frontier nodes at different depths.

As with all best-first search algorithms, the main problems with this approach are that the space requirement is exponential in the search depth, and there is an overhead cost associated with maintaining the OPEN list. Our solution to the space requirement is to monitor the size of the OPEN list, and when it becomes full, we stop the exploration and make a decision. After a decision is made, the irrelevant part of the search tree is pruned, freeing up space for the next exploration phase. To handle the additional overhead concern, we can just limit the size of the OPEN list, in which case the additional overhead per node generation is bounded by a constant. We can then choose the size of the OPEN list so that this constant is small enough to ignore, or so that the algorithm performance in a real-time setting is optimized. Increasing the size of the OPEN list will increase the overhead, but reduce the number of decisions made because the OPEN list is full.

Alternatively, we could have chosen to implement a linear-space best-first search algorithm (*e.g.,* [Kor93]). We decided against this because of the overhead cost resulting from regenerating nodes. In addition, for the problems that we considered, computation time was a more important consideration than memory.

7.3.3.2 k-Best Best-First Search

k-best BFS is essentially αk-best BnB using best-first exploration instead of depth-first branch-and-bound exploration. k-best BFS performs a best-first exploration of the problem space by expanding nodes in increasing order of their node cost. When the available time runs out, decisions are made based on the k-lowest estimated-cost frontier nodes in the OPEN list under each child of the root node. As with αk-best BnB, we calculate the estimated cost as the sum of the node cost, and an estimate of the average cost per decision times the remaining depth in the problem space tree. If the remaining depth is not known, then a sufficiently large depth (*i.e.,* greater than the deepest frontier node) is chosen arbitrarily. For our experiments we estimated the average cost per decision as the estimated cost of a single greedy decision (*i.e.,* $1/(b+1)$). If the minimum-cost frontier node is also a leaf node, then the exploration stops, and all remaining decisions follow this optimal path.

Algorithm	Exploration Rule: f_{exp}	Decision Rule: f_{dec}
idMINIMIN	$idBnB(node_cost(x))$	$estimated_total_cost(x)$
αk-best BnB	$idBnB(node_cost(x))$	$E(MPC\|k)$
node-cost BFS	$node_cost(x)$	$node_cost(x)$
estimated-cost BFS	$estimated_total_cost(x)$	$estimated_total_cost(x)$
hybrid BFS	$node_cost(x)$	$estimated_total_cost(x)$
k-best BFS	$node_cost(x)$	$E(MPC\|k)$

Table 7.3: Table of algorithms considered.

7.4 Results for Random-Tree Experiments

In order to compare the real-time search algorithms presented in the previous section, we performed the following set of experiments. For each trial, we constructed a random tree using the method described in Chapter 2, and choose a fixed number of node generations that would be allowed for each decision. Each algorithm was then used to generate a path from the root node to a leaf node. The percent error of this path cost relative to the optimal path cost was recorded, and then averaged over 1000 different trials.

For these experiments, the expected or average cost of a single decision was estimated as the cost of a greedy decision ($c = 1/(b+1)$ for a tree with branching factor b), and edge costs were chosen uniformly from the set $\{0, 1/2^{10}, ..., (2^{10} - 1)/2^{10}\}$. We tested the six algorithms listed in Table 7.3 over a range of constraints on the exploration phase (*i.e.,* available generations per decision). The results in Figure 7.5 show the average over 1000 trials of the solution-cost error, as a percentage of optimal solution path cost, versus the number of generations allowed per decision for a binary tree of depth 20. Note that the results are presented with a log-scale on the horizontal axis. All algorithms had sufficient space to save the relevant explored subtree from one decision to the next. The leftmost data points correspond to a greedy decision rule based on a 1-level lookahead (*i.e.*, 2 generations per decisions).

The results indicate that k-best BFS performs better than *node-cost BFS*, *estimated-cost BFS*, and only slightly better than idMINIMIN, αk-best BnB and *hybrid BFS*. *Node-cost BFS* produces average solution costs that are initially

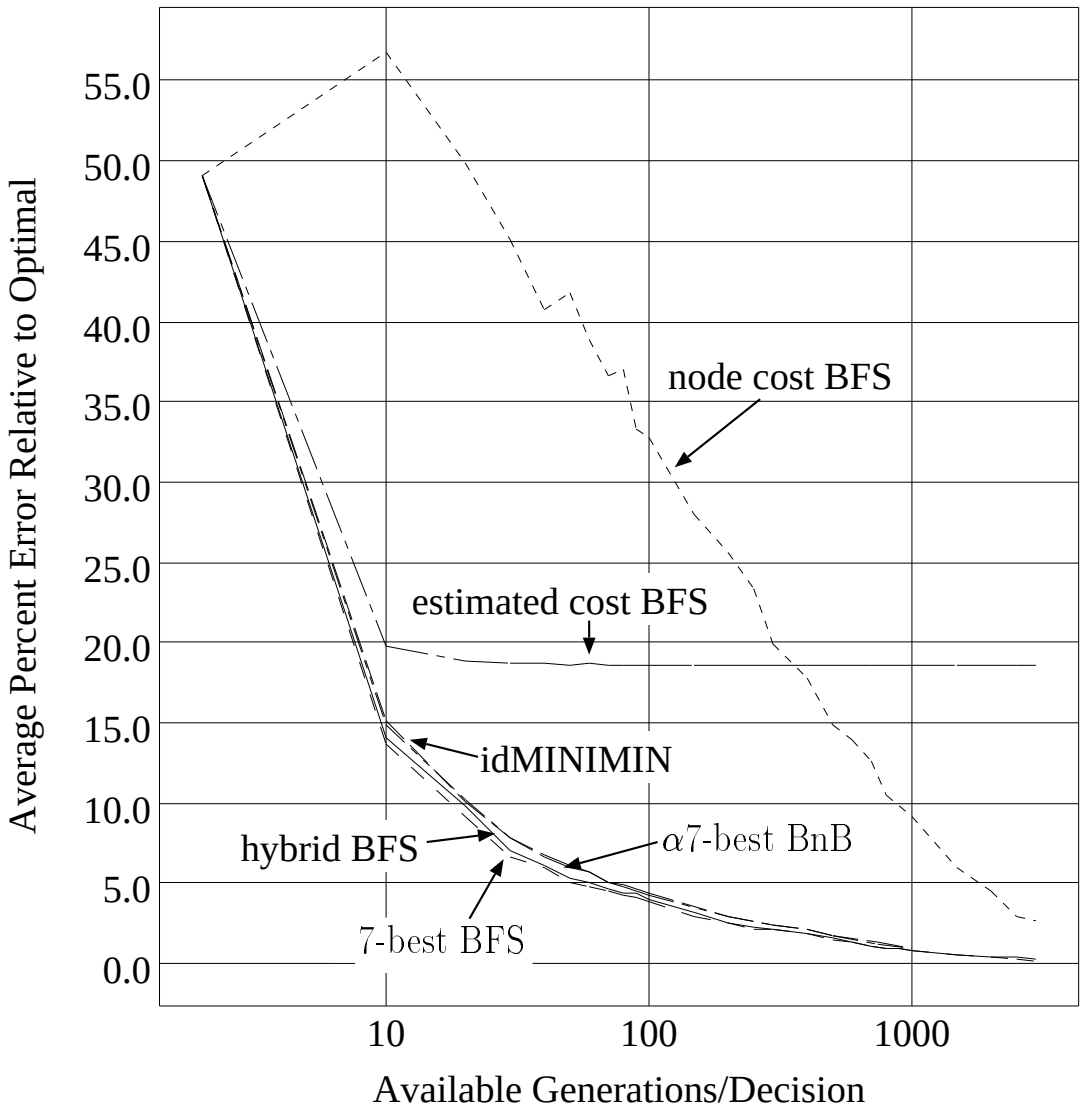


Figure 7.5: Average percent error versus node generations available per decision for depth 20 binary trees with real-time node generation constraint.

higher than a greedy decision maker. This is due to the best-first "swap pathology", since the initial computation is spent exploring the subtree under the greedy root child, eventually making it look worse than the other root child. *Estimated-cost BFS* does perform better than greedy, but its performance quickly levels off well above the average performance of the other algorithms. This is because the estimated-cost exploration heuristic is not admissible and does not generate a balanced tree to support the current decision. Thus *estimated-cost BFS* often finds a sub-optimal leaf node before using all the available computations and then commits to decisions along the path to that node without further exploration.

The difference in performance between idMINIMIN and αk-best BnB is very small. αk-best produces solutions whose average error is only very slightly lower than the average error of the solutions produced by idMINIMIN, over the range of available generations per decision. Both k-best BFS and *Hybrid BFS* performs slightly better than idMINIMIN and αk-best BnB. This is probably due to the fact that a best-first exploration will find the minimum-cost frontier node at a given depth with less node generations than a depth-first branch-and-bound exploration method. k-best BFS produces solutions whose average error is consistently lower than *hybrid BFS*.

Since the difference in performance between idMINIMIN and $\alpha 7$-best BnB is very small, we also recorded the number of times that each algorithm produced a lower cost solution in a head-to-head competition. These results shown in Figures 7.6 and 7.7 do *not* indicate that αk-best BnB is a clear winner over idMINIMIN. idMINIMIN and $\alpha 2$-best BnB trade off winning a larger percentage of the head-to-head trials. This may be due to the *ad hoc* way that we extended the fixed lookahead bound version of $\alpha 2$-best to handle the partial information in the next exploration level. $\alpha 7$-best does perform better on average than idMINIMIN over a wide range of available node generations per decision.

We also recorded the percentage of time that k-best BFS and *hybrid BFS* produced lower cost solutions in a head-to-head competition. These results, which are presented in Figure 7.8, are very similar to the results for idMINIMIN and $\alpha 7$-best BnB.

The results for the traditional best-first search algorithms are not surprising since *node-cost* and *estimated-cost BFS* were not expected to perform well. What is interesting is that a previous decision-theoretic analysis of the exploration problem [RW91] suggested that, for a given decision heuristic evaluation function and the *single-step* assumption (*i.e.,* that the value of a node expansion can be determined by assuming that it is the last computation before a move decision), the best node to explore should be determined by the same heuristic function.

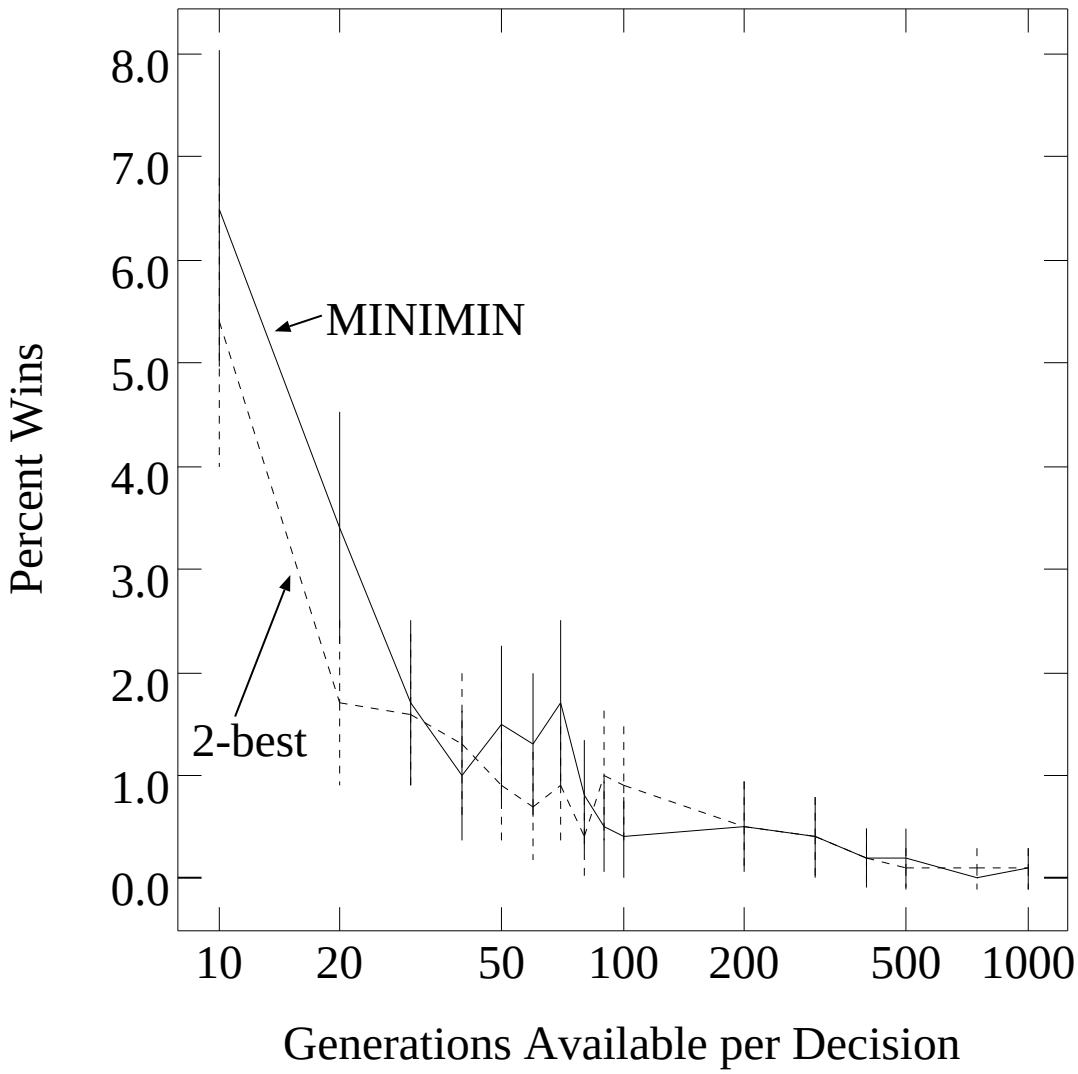


Figure 7.6: Percent wins over 20 decisions for α2-best BnB and idMINIMIN versus available generations per decision.

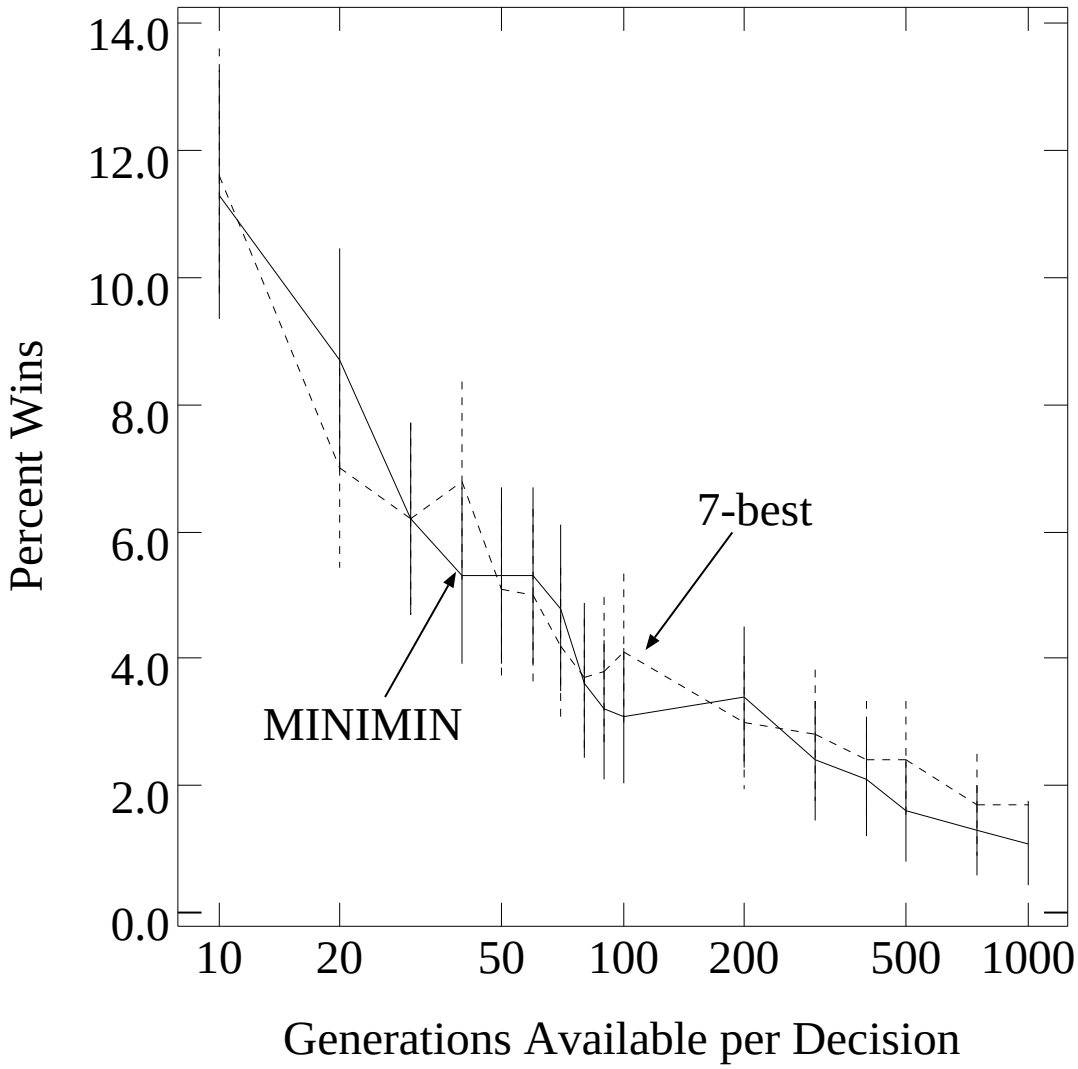


Figure 7.7: Percent wins over 20 decisions for α7-best BnB and idMINIMIN versus available generations per decision.

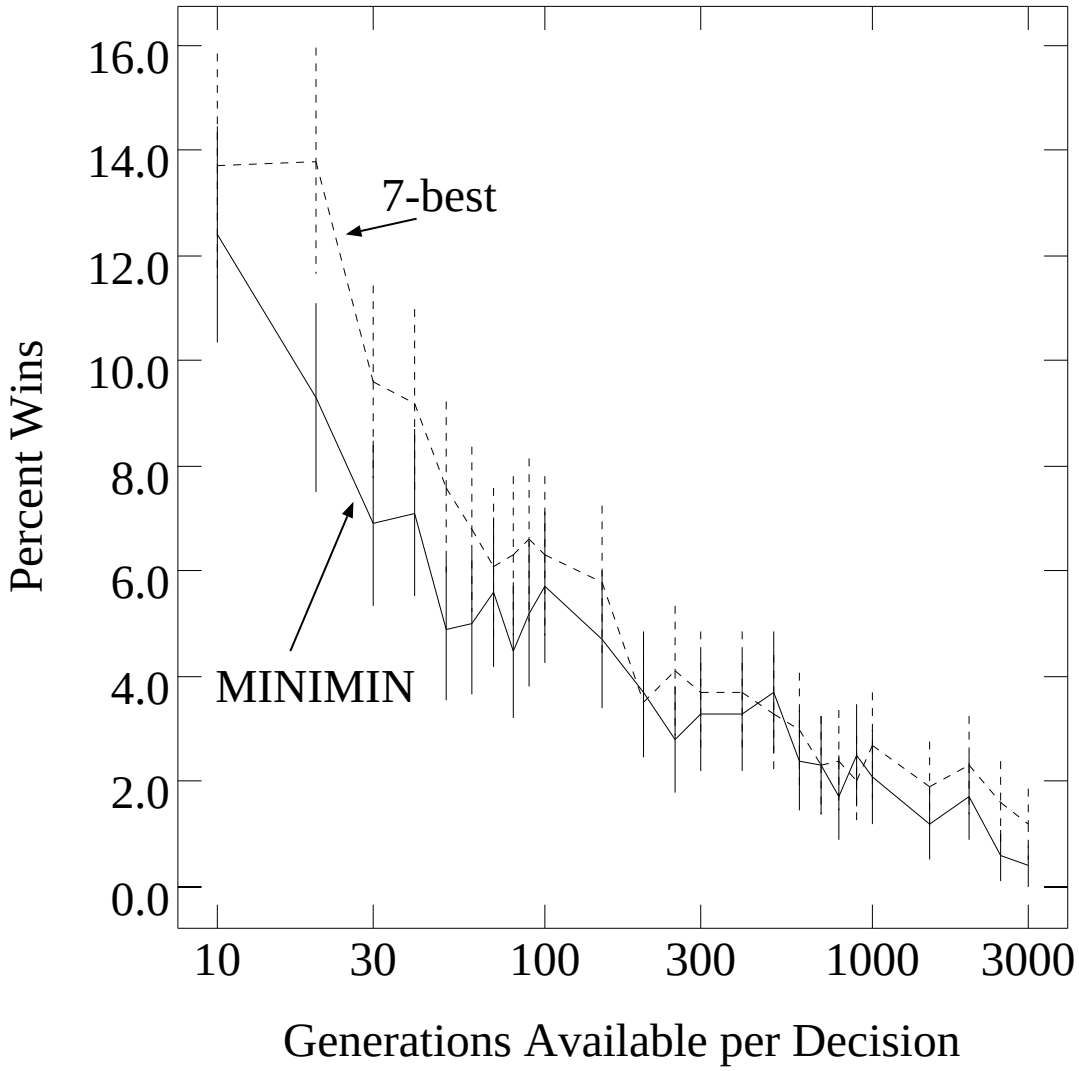


Figure 7.8: Percent wins over 20 decisions for 7-best BFS versus *hybrid BFS*.

Our experimental results and pathological examples contradict this suggestion.

The fact that both αk-best BnB and k-best BFS outperform idMINIMIN and *hybrid BFS* by a small margin is in line with our expectation. The fact that the best-first algorithms outperform the depth-first branch-and-bound algorithms is probably due to the fact that a best-first exploration typically finds the minimum-cost frontier node at a given level sooner (*i.e.,* with less node generations) than a depth-first branch-and-bound exploration.

7.5 Discussion of Other Experiments

Thus far we have only reported results for binary trees. The main reason for this is that in the worst case it takes less time to search a binary tree to the same depth that a tree with a larger branching factor. Thus for experiments on binary trees, we are able to report results for deeper trees and for more trials.

We have also performed experiments on trees with larger branching factors (*i.e.,* $b\epsilon\{3,4,5,6\}$). The results we obtained are qualitatively the same as the results that we have reported for binary random trees, thus we will only make some general observations about the results. Our initial intuition was that the performance of k-best decision making relative to MINIMIN would improve with increasing branching factor. The reasoning behind this intuition is that MINIMIN decisions are based on a single frontier node, and thus MINIMIN decisions ignore all the other information in the search tree. When the search tree has a larger branching factor, then MINIMIN ignores a larger amount of information. Our results do not support this intuition. One reason that MINIMIN's decision making performance relative to k-best is apparently not affected by an increase in the branching factor is that k-best is also a myopic decision method, so it also must ignore more information in trees with a larger branching factor. Another possible explanation is that MINIMIN's commitment to a single step along the minimum-cost path to a frontier node is most likely to be a bad decision when there is a good chance that all paths below a minimum-cost frontier node turn out to have a high cost. When the branching factor is higher, there are more opportunities for a good frontier node to lead to a good leaf node in the random tree model that we have adopted. Thus, when MINIMIN is ignoring more of the information available, it has a lower risk of making a bad decision. Although the average percent error relative to optimal in our experiments increased with the branching factor for both idMINIMIN and αk-best BnB, the difference between their average percent error did not appreciably increase.

We should note at this point that the αk-best decision equations depend on

Branching Factor (b)	Frontier Nodes (k)
2	7
3	5
4	4
5	4
6	4

Table 7.4: Maximum number of frontier nodes (k) in k-best equations for different branching factors.

the branching factor of the next level of the search tree. Recall that part of the difficulty in solving the expected minimum-path-cost equations comes from the number of terms in the completion-cost distribution. In general, the number of terms in the completion-cost distribution is an increasing function of the branching factor in the unexplored part of the problem-space tree. For this reason, we have only been able to solve the expected minimum-path-cost equations for trees with a small number of frontier nodes as shown in Table 7.4. For larger numbers of frontier nodes, and when the edge costs are independently chosen at random from a uniform distribution over the range $[0, 1]$, MAPLE[1] is not able to solve the expected minimum-path-cost equations without running out of memory.

We have also performed experiments on random trees where the branching factor of each node is chosen independently from a uniform distribution over a set of values (*e.g.,* $b \epsilon \{2, 3, 4\}$ or $b \epsilon \{2, ..., 10\}$). Again, our results are qualitatively the same as for the binary random-tree experiments. One general observation that we can make is that our k-best algorithms were developed under the assumption that the general structure of the problem space is known (*i.e.,* branching factor and edge-cost distribution). When the nodes have random branching factors, then one option is to recalculate the completion-cost distribution based on the range and distribution of possible branching factors. Of course, this also depends on knowing the range and distribution of branching factors *a priori*. Instead of this approach, we simply allowed the k-best algorithms to use a completion-cost

[1]MAPLE is an interactive computer algebra package developed at the University of Waterloo.

distribution based on the minimum branching factor possible. This assumption did not appear to significantly affect the experimental results.

This set of experiments does point out a weakness in the k-best approach, however, namely that it requires more information that MINIMIN. We will address this issue in Chapter 8.

We also investigated the question of how the range of edge costs in the random tree might affect the relative performance of k-best and MINIMIN decision making. We performed a new set of experiments on depth 20 binary trees for three different numbers of node generations per decision (g/d) and a wide selection of edge costs ranges. Each data point in Figure 7.9 is an average of 1000 trials. In each trial, a random tree was constructed with edge costs chosen independently from the set $\{0, 1/r, ..., (r-1)/r\}$ where r is the range value shown on the horizontal axis of the graph (*i.e.,* the maximum edge cost).

The results show that α7-best BnB produces solutions with lower average error than idMINIMIN for all edge cost ranges tested. In addition, we observe that the relative difference between α7-best BnB and idMINIMIN is larger when the range of edge costs is smaller. This is likely due to the fact that MINIMIN only takes into account the first minimum-cost frontier node. In random trees with a small range of edge costs, it is more likely that there will be more than one minimum-cost frontier node, in which case α7-best is much more likely to choose the child of the root that has more minimum-cost frontier nodes below it.

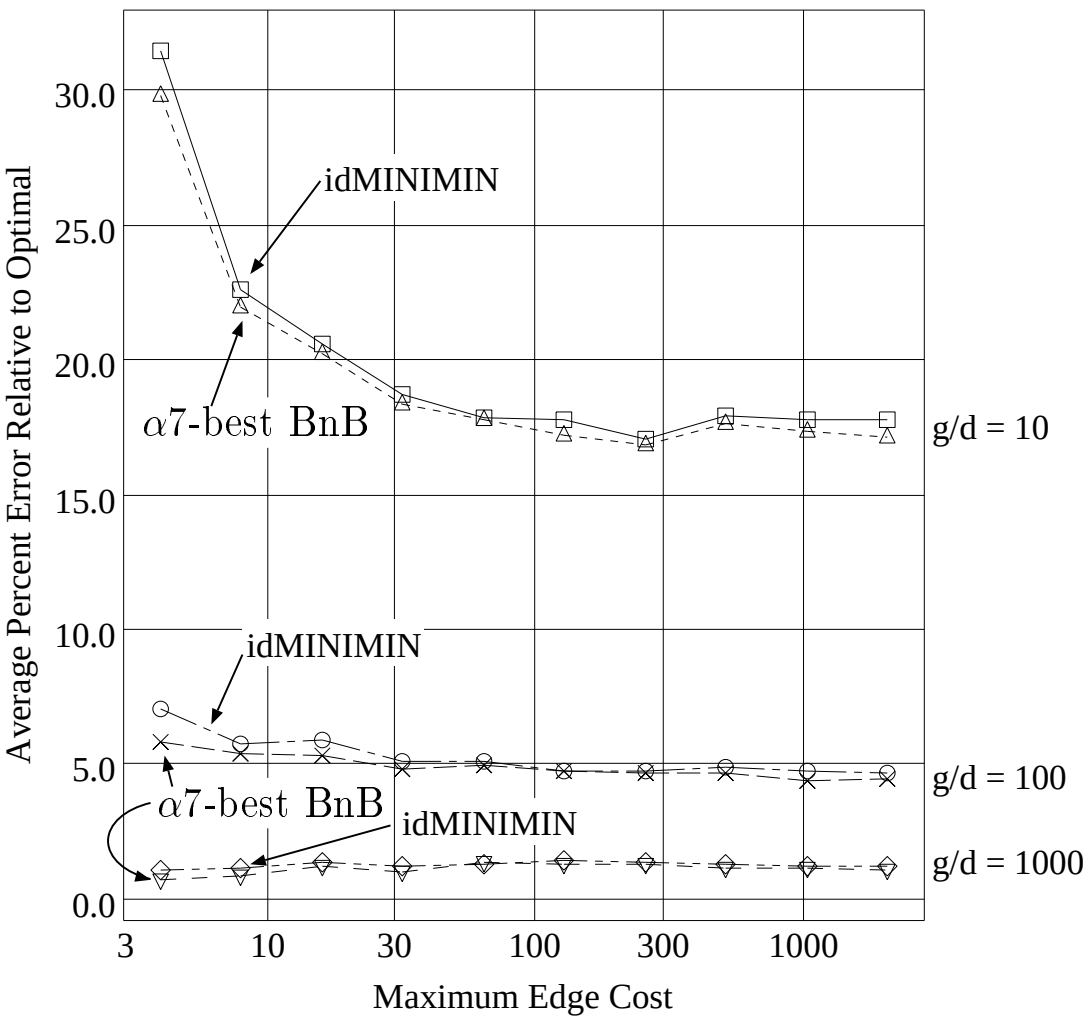


Figure 7.9: Average percent error versus maximum edge cost (generations per decision $\epsilon\{10, 100, 1000\}$).

CHAPTER 8

Learning the Completion-Cost Distribution

In this chapter we present an extension to the k-best decision algorithm for problems where the edge-cost distribution is not known *a priori*. After motivating the need for sampling the edge-cost distribution, we present our method, and suggest that in some instances this might provide better information than the actual edge-cost distribution because of the assumptions that we made to justify k-best. We then present results comparing αk-best BnB using our sampling method with αk-best BnB using the expected-value equations derived from the edge-cost distribution for small search trees.

8.1 How to Sample the Completion-Cost Distribution

All the variations on the k-best decision algorithm that we have presented thus far have relied on the fact that the distribution of edge costs was known and available to the algorithm. When the edge-cost distribution is not known *a priori*, we can extend our approach by building up an estimate of the distribution during the decision-making process.

In order to evaluate the performance of k-best decision making on real-world problems, we have implemented a new version of the αk-best decision algorithm called $s\alpha k$-best (s for sampled) that uses the following sampling method for learning the distribution. When the edge-cost distribution is known, it is used to calculate the completion-cost distribution for frontier nodes (see Chapter 5). Otherwise, since we are ultimately interested in the completion-cost distribution, it makes more sense to sample it rather than calculating it from a sample of the edge-cost distribution.

Our approach to sampling or learning the completion-cost distribution is very simple. We first divide up the range of possible completion-costs into discrete segments, and associate a bucket with each segment. When a node is expanded, we calculate the minimum edge cost over all the child nodes, and then increment the bucket counter associated with this minimum edge cost. Once a sufficiently large set of samples has been taken (*e.g.,* 10 times the number of buckets), then

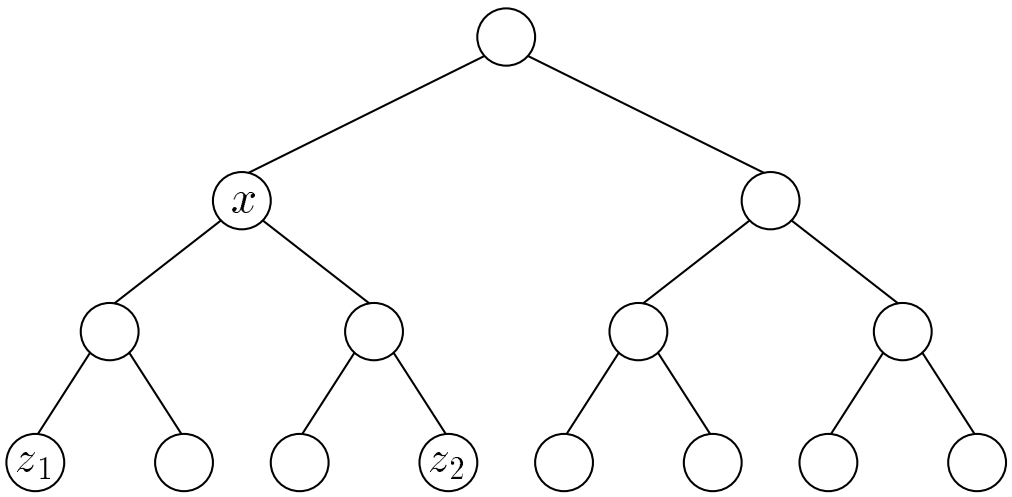


Figure 8.1: An example search tree.

the numbers in the buckets are used to estimate the completion-cost distribution using the following steps. First, the cumulative distribution function is calculated as follows.

$$F_{CC}(i) \approx p(\text{min edge cost} \leq x_i) = F_S(i) = \sum_{j=0}^{i} \frac{(\text{samples in bucket } i)}{(\text{number of samples})}$$

where x_i is the maximum edge cost in the range associated with bucket i and F_S is the sample-based estimate of the cumulative completion-cost distribution. This estimate of the cumulative distribution for the completion cost ($F_{CC}(x)$) is just the sum of the samples in the buckets associated with each cost that is less than or equal to x_i, divided by the total number of samples taken. The initial values for this sampled distribution are learned on-line by making MINIMIN decisions until enough samples have been taken, after which we continue to take samples while making decisions based on the sampled distribution.

Next, the completion-cost density function is estimated as the ratio of the number of samples in bucket i to the total number of samples, multiplied by the ratio of the range of edge costs to the width of a bucket.

$$f_{CC}(i) \approx f_S(i) = \frac{(\text{samples in bucket } i)}{(\text{number of samples})} \cdot \frac{(\text{edge cost range})}{(\text{bucket width})}$$

Here, $f_S(i)$ is the sample-based estimate of the completion-cost density function.

Consider the example tree in Figure 8.1. Let z_1 and z_2 be the two minimum frontier-node costs below the node labeled x. An example set of buckets are

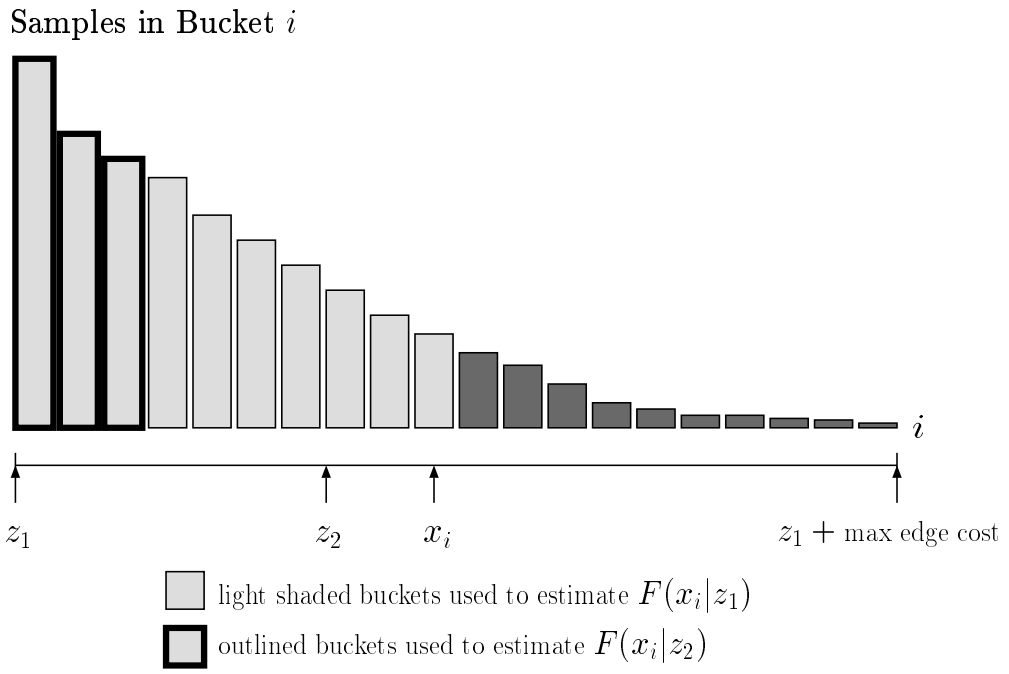


Figure 8.2: Example of buckets for a sampled completion-cost distribution.

shown in Figure 8.2a. If we assume that the completion-cost distributions below the two minimum-cost frontier nodes are independent, then we can calculate an estimate of the combined completion-cost distribution of a common ancestor x of these two frontier nodes using the following set of equations.

Let $B(z)$ be the number of the bucket corresponding to the completion cost z.

$$B(z) = \begin{cases} int(\frac{z}{c_{max}} \cdot n) & 0 \leq z \leq c_{max} \\ \text{undefined} & \text{otherwise)} \end{cases}$$

where c_{max} is the maximum edge cost, n is the number of buckets, and $int(x)$ returns the truncated integer value of x. Without loss of generality, we assume that $z_1 \leq z_2$ and that the minimum edge cost is zero. Let $j = B(z_2 - z_1)$ be the number of buckets corresponding to the difference between the two minimum frontier-node costs (z_1, z_2). First we can estimate the cumulative distribution for the minimum path cost through a child, x_i, given the two minimum frontier-node costs as follows.

$$F_{CC}(x_i|z_1, z_2) \approx 1 - (1 - F_S(B(x_i - z_1))) \cdot (1 - F_S(B(x_i - z_2)))$$

Note that when $B(z)$ is undefined, then $F_S(B(z))$ and $f_S(B(z))$ equal zero. An example of how the buckets are used to calculate the cumulative distribution is shown in Figure 8.2. The density function for the minimum path cost through a child node, x_i, is then just the derivative of the cumulative distribution function.

$$\begin{aligned} f_{CC}(x_i|z_1,z_2) &\approx \frac{d}{dx}(F_S(B(x_i-z_1)) + F_S B(x_i-z_2)) \\ &\quad -F_S(B(x_i-z_1)\cdot F_S(B(x_i-z_2)) \\ &= f_S(B(x_i-z_1)) + f_S(B(x_i-z_2)) \\ &\quad -f_S(B(x_i-z_1))\cdot F_S(B(x_i-z_2)) \\ &\quad -f_S(B(x_i-z_2))\cdot F_S(B(x_i-z_1)) \end{aligned}$$

With these equations, we can now calculate an estimate of the expected completion cost for a path below a child of the root node based on the two minimum frontier-node costs ($z_1 \leq z_2$) as follows.

$$\begin{aligned} E(x|z_1,z_2) &\approx \sum_{i=1}^{n} x_i \cdot (f_S(B(x_i-z_1)) + f_S(B(x_i-z_2)) \\ &\quad -f_S(B(x_i-z_1))\cdot F_S(B(x_i-z_2)) \\ &\quad -f_S(B(x_i-z_2))\cdot F_S(B(x_i-z_1))) \end{aligned} \tag{8.1}$$

This equation can then be used to make sαk-best move decisions in the same manner that the $E(MPC|k)$ equation is used by αk-best. The equation for the expected minimum path cost based on the sampled completion-cost distribution can be extended to an arbitrary number of frontier nodes, although we have only implemented the sampled expected minimum-path-cost equation for the case of two frontier nodes. Our code for sampling the completion-cost distribution and then using the sampled distribution to calculate the expected completion cost is contained in Appendix D.

The main advantage of this approach is that it allows us to sample the completion cost distribution directly, rather than basing it on the edge-cost distribution. One problem with using the edge-cost distribution information to calculate the completion-cost distribution comes from our assumption that the last incremental decision problem is a reasonable approximation of the general incremental decision problem. When we sample the completion cost distribution, we are not forced to make this same assumption. This means that if the error that results from the last incremental decision assumption is greater than the error due to sampling the completion-cost distribution, then the sampling approach will make better quality decisions on average. Another advantage of this approach is that we don't need to know the branching factor of the tree in advance, because the

effect of the branching factor on the completion-cost distribution will also be captured by the information in the sampled distribution.

8.2 Results for Sampling Random Trees

In order to evaluate sαk-best decision making, we implemented a search algorithm (sαk-best BnB) that uses depth-first branch-and-bound exploration and makes decisions using the sαk-best decision algorithm. We then repeated some of the experiments on random trees from Chapter 6. sαk-best BnB was configured to use 100 buckets, and made MINIMIN decisions until 1000 samples had been taken. Although it is possible that making MINIMIN decisions during the initial sampling phase might bias the resulting samples, we did not observe any such bias in our results for $s\alpha k$-best. After 1000 trials were performed, each subsequent trial consisted of calculating the optimal solution cost, and the solution cost produced by sαk-best BnB and MINIMIN BnB over a sequence of 20 decisions. The percent error relative to optimal ($100*(solution\ cost - optimal)/optimal$) was then calculated for both sαk-best BnB and MINIMIN BnB. If the solution-path cost produced by sαk-best BnB was lower than the solution-path cost produced by MINIMIN BnB, then sαk-best BnB won that trial. If instead MINIMIN BnB produced a lower cost solution path than sαk-best BnB, then MINIMIN BnB won. Otherwise the result was a tie. The results were averaged over 10,000 trials that included the initial 1,000 trials during which sαk-best BnB made MINIMIN decisions.

The results in Figure 8.3 show the percent error relative to optimal versus search depth on depth 20 binary random trees for sα2-best BnB and MINIMIN BnB (the 95% confidence limits are shown in the figure). From these results, we observe that the sα2-best BnB makes better decisions on average than MINIMIN BnB, although, as with αk-best, the amount of improvement is slight. To further compare the performance of sα2-best BnB and MINIMIN BnB, we also recorded the percentage of trials that each algorithm produced a lower-cost solution than the other. These results are summarized in Table 8.1, and show that sα2-best BnB produces a lower-cost solution than MINIMIN BnB a greater fraction of the time than MINIMIN BnB produces the lower-cost solution.

The results in Figure 8.3 and Table 8.1 demonstrate that our approach to sampling the completion-cost distribution is able to support k-best decision making, and still improve slightly over MINIMIN decision making. As with αk-best when the distribution information is known, sαk-best requires some overhead to compute the completion-cost distribution that results from taking samples and

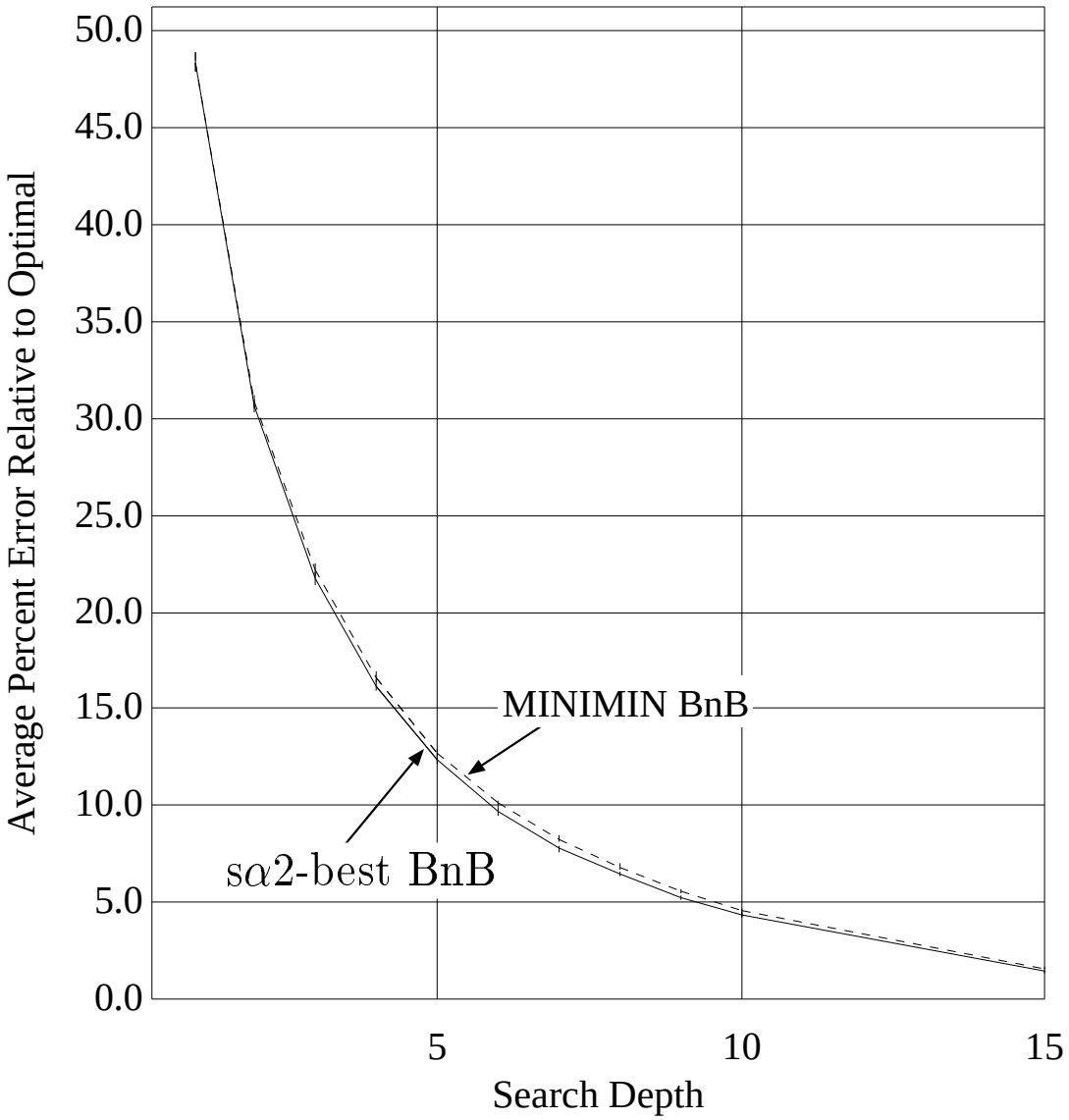


Figure 8.3: Average percent error relative to optimal versus search depth for MINIMIN BnB and sα2-best BnB.

Search depth	sα2-best BnB % wins	MINIMIN BnB % wins	ratio
1	0.0 $\pm$ 0.000	0.0 $\pm$ 0.000	0
2	21.2 $\pm$ 0.802	20.6 $\pm$ 0.793	1.029
3	19.2 $\pm$ 0.772	16.4 $\pm$ 0.726	1.170
4	16.3 $\pm$ 0.724	13.7 $\pm$ 0.674	1.189
5	14.4 $\pm$ 0.689	12.1 $\pm$ 0.639	1.190
6	12.9 $\pm$ 0.658	10.0 $\pm$ 0.587	1.290
7	11.3 $\pm$ 0.621	8.3 $\pm$ 0.541	1.361
8	10.5 $\pm$ 0.600	7.6 $\pm$ 0.519	1.381
9	9.2 $\pm$ 0.566	6.6 $\pm$ 0.488	1.393
10	8.1 $\pm$ 0.535	5.8 $\pm$ 0.459	1.396
15	4.5 $\pm$ 0.404	2.9 $\pm$ 0.326	1.551

Table 8.1: Percent wins over 20 decisions for sα2-best BnB and MINIMIN BnB versus search depth.

processing the sampled distribution information. The cost of this overhead computation will be discussed in Chapter 9.

The sαk-best decision algorithm raises the question of how many buckets should we use. The tradeoff is that increasing the number of buckets results in a more accurate representation of the underlying completion-cost distribution, but also increases the overhead since the number of steps required to calculate the distribution in Equation 8.2 is linear in the number of buckets used. We chose 100 buckets with this tradeoff in mind, and the results presented above support this as a reasonable choice. For other problem domains with different completion-cost distributions, it will be necessary to tune the number of buckets to optimize the performance of sαk-best BnB.

CHAPTER 9

Incremental Search and Flowshop Scheduling

In this chapter, we present and evaluate our incremental search approach to the problem of flowshop scheduling. We first describe the general flowshop scheduling problem, and then present our real-time search approach to this problem. We next present experimental results comparing the performance of two real-time extensions of traditional branch-and-bound search to our incremental search algorithm (idMINIMIN). The experimental results support our belief that incremental search is a reasonable approach to solving real-time decision problems such as flowshop scheduling.

9.1 What is Flowshop Scheduling?

One example of flowshop scheduling is the problem of typesetting and printing documents. There are two machines (the typesetter and the printer), and n documents to be printed. All documents must be typeset before being printed. The objective is to find an order to process the documents that minimizes some cost function. This document processing problem is a two-machine flowshop scheduling problem.

We define the flowshop scheduling problem as follows. There are m machines and n jobs. Each job has m tasks, one for each machine, and the tasks for each job must be performed in the same prespecified order. This task order is the same for all jobs. Each task has a duration or time that is required to process a job, and the machines can only process one job task at a time. A flowshop schedule consists of an ordered list of tasks for each machine to process, subject to the constraint that the tasks for every job are performed in their prespecified order.

Recall the two-machine flowshop scheduling example that we first presented in the introduction. For this example, there are two machines and four jobs (see Figure 9.1a). The tasks times for four jobs are shown in Figure 9.1b. For each job, its task on machine M_1 must be completed before it can start executing its task on machine M_2. A sample flowshop schedule is shown in Figure 9.1c.

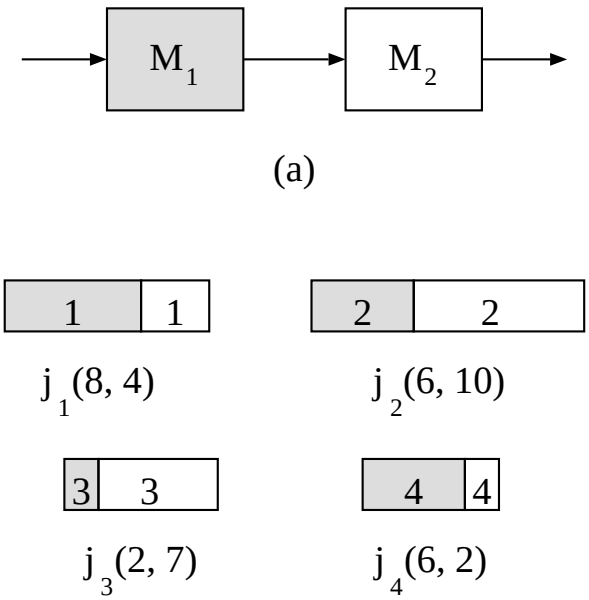


Figure 9.1: Simple flowshop scheduling problem, (a) two machine flowshop, (b) jobs and their processing times, and (c) a sample schedule.

The cost of a flowshop schedule depends on the cost function employed. For the two-machine flowshop scheduling problem, there are two commonly used cost functions: *makespan* and *sum-finish-time* [PS82]. The *makespan*, which is perhaps the most obvious cost function, is simply the time when the last job finishes being processed by the last machine. For the example, the *makespan* of the sample schedule in Figure 9.1 is 32 time units. One situation where we might be interested in finding a flowshop schedule that minimizes the makespan is when the production facility is rented by the hour, and this rent is a dominating cost of production. In this case, we want to finish processing all the jobs in the minimum amount of time (*i.e.,* find a schedule that minimizes the *makespan*).

The *sum-finishing-time* (SFT) (also called the *shortest-processing-time* and *mean-completion-time*) is the sum of the times that each job finishes processing on the last machine. For the sample schedule in Figure 9.1, the sum-finishing-time is calculated as follows.

$$sum\ finishing\ time = 8 + 15 + 20 + 32 = 75$$

This sample schedule happens to be optimal with respect to the *sum-finishing-time* cost function. One motivation for the *sum-finishing-time* cost function is the case where the dominating cost of production is the inventory cost of the raw materials that need to be processed (*i.e.,* the interest that is either paid on borrowed money or lost on capital that could be invested elsewhere). Note that we can sell the complete product one we have finished processing it on the last machine.

To further illustrate the difference between the *makespan* cost function and the *sum-finishing-time* cost function, consider again the document processing example. If one person submits ten documents for printing, then a reasonable objective is to find a schedule that minimizes the total time before all documents are printed (*i.e.,* a schedule that minimizes the *makespan*). If instead ten people each submit a single document for printing, then a better objective is to find a schedule that minimizes the total time that people have to wait for their printouts (*i.e.,* a schedule that minimizes the *sum-finishing-time*).

For either cost function, we can find the optimal schedule by enumerating the set of all possible schedules using a brute-force search. One useful way to enumerate this set is by exploring a problem-space representation of the schedules. The problem-space representation of our four-job scheduling problem is shown in Figure 9.2. In the figure, the root node represents the initial problem, and each interior node represents a partial schedule. Each node is labeled with the name of the job that is added to its parent node's partial schedule, and the node represents the subproblem of scheduling the remaining jobs given the partial schedule up

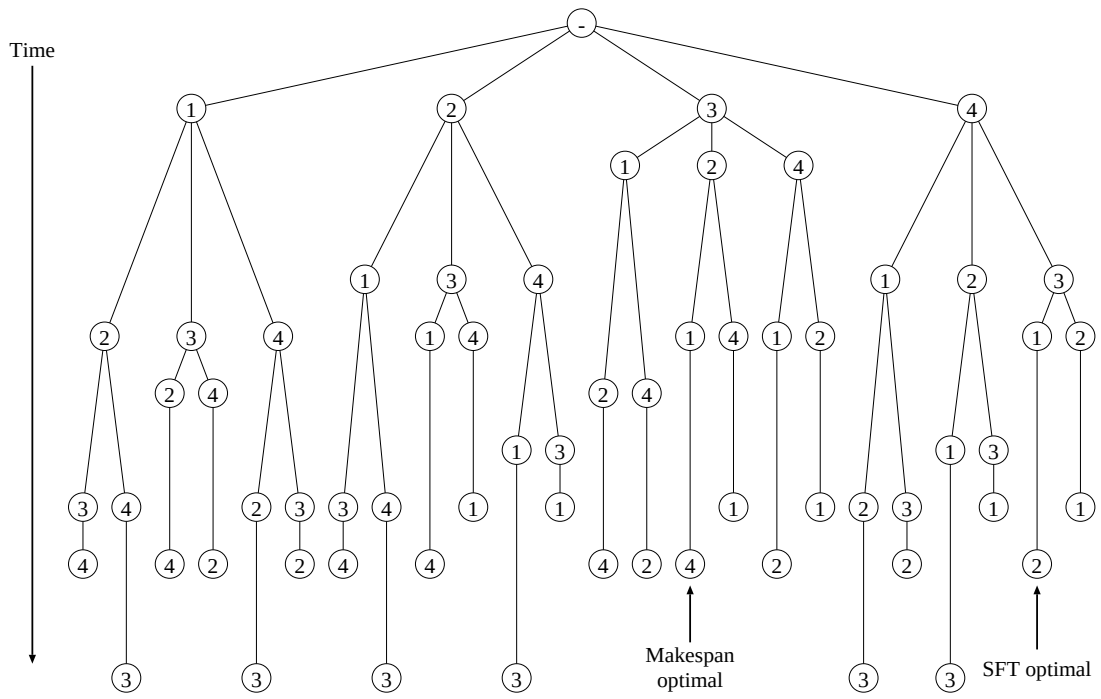


Figure 9.2: Problem-space representation for a four-job, two-machine flowshop scheduling problem.

to that node. Associated with each leaf node is a complete schedule, which is simply the order of the jobs along the path from the root to that leaf node. We have assumed that the complete schedule is simply the order in which jobs will be processed, which will be the same for both machines (called a permutation schedule). This is a reasonable assumption because, for any schedule with a cost based on any regular measure of performance (*e.g., makespan* or *sum-finishing-time*), there exists a schedule with the same or lower cost for which the order of jobs on each machine is the same [CMM67]. The optimal schedule with respect to the *makespan* and *sum-finishing-time* cost functions are indicated by the arrows in Figure 9.2. The vertical position of a node corresponds to the time at which machine M_1 finishes processing the partial schedule up to that point (*i.e.,* the time when the next decision must be made to keep the first machine from being idle). Note that the minimum-makespan schedule is not the same as the minimum-cost schedule with respect to the sum-finishing-time cost function.

It has been observed that for the case of a two-machine flowshop, the makespan decision problem (namely does there exist a flowshop schedule with a makespan of at most T?) can be solved in time that is polynomial in the number of jobs [Joh54]. It has also been observed that the minimum sum-finishing-time decision

problem (namely does there exist a flowshop schedule with a sum-finishing-time of at most T?) is NP-complete for flowshop problems with two or more machines [GJS76, GS79]. The sum-finishing-time optimization problem (namely what is the minimum sum-finishing-time schedule?) is at least as hard as the sum-finishing-time decision problem. For these reasons, we have focussed our attention on sum-finishing-time flowshop scheduling problems.

9.2 Real-Time Flowshop Scheduling

Given sufficient time, the flowshop-scheduling problem-space tree can simply be searched using a branch-and-bound exploration, while keeping track of the schedule associated with the minimum-cost frontier node. After the complete tree has been searched, the path from the root to a minimum-cost leaf node specifies an optimal flowshop schedule. There are several factors that contribute to make a flowshop-scheduling problem a real-time decision problem. One is when there is not sufficient time to find an optimal solution, we have a real-time flowshop-scheduling problem. Another is that we cannot solve all possible problem instances in advance. One important characteristic of real-time flowshop scheduling is that computation time spent while the machines are busy does not adversely impact the cost of the schedule, whereas computation time spent while the machines are idle directly increases the cost of the schedule produced. For example, any flowshop scheduling problem can be made into a real-time decision problem by starting the cost clock as soon as the scheduler is given the set of jobs and their processing times.

For example, consider another real-time flowshop scheduling problem. In the winter, airplanes must be refueled and de-iced before takeoff. If we assume that planes must be de-iced immediately before takeoff, then planes must first be refueled and then de-iced. We further assume that there is only one refueling truck and one de-icing truck. The time needed to refuel and de-ice varies from one airplane to another due to different fuel levels and capacities and to different wing sizes. The airplane refuel and de-ice scheduling problem is a flowshop scheduling problem. If we only have to refuel, then the obvious schedule that minimizes the total time spent by airplanes waiting on the ground is to refuel the planes in increasing order of their refueling times. When the weather conditions change, then we have a real-time two-machine flowshop scheduling problem where our objective is to minimize the average (or total) time that airplanes spend on the ground. In this case, the scheduler's goal is find a schedule that minimizes the sum-finishing-time. Note that minimizing the average-finish-time is the same

problem as the sum-finishing-time problem except that all the times are divided by the number of jobs. Since we don't know in advance the refueling and de-icing requirements of all planes in the future, we must adopt a real-time approach to this problem, and schedule the refueling and de-icing as the planes arrive. This makes the airplane refueling and de-icing problem a real-time flowshop scheduling problem. While planes are being de-iced and refueled, we can be figuring out which planes to refuel and de-ice next.

In summary, the sum-finishing-time cost of a real-time flowshop scheduling problem must include the cost of all time spent figuring out which schedule to execute that is not performed concurrently with the execution. The nice part about flowshop scheduling is that while the current job is being processed, we can perform additional computation to try to improve the schedule of the remaining jobs, without adding to the cost of the schedule. This is our motivation for applying incremental search to this problem.

9.3 Three Approaches to Real-Time Scheduling

We have considered three basic approaches to real-time scheduling. The first two are somewhat naive extensions to the optimal off-line branch-and-bound approach. The third is our incremental search approach. All of the search algorithms presented make use of a simple but accurate heuristic function for estimating the sum-finishing-time cost of completing a partial schedule that was first suggested by Ignall and Schrage [IS65]. Their basic idea is to calculate two simple estimates of the complete-schedule cost, one by ignoring the one-job-at-a-time constraint for jobs on the first machine and the other by ignoring the one-job-at-a-time constraint for jobs on the second machine. The maximum of these two completion-cost estimates is a very good lower bound on the actual completion cost of a partial schedule. We refer to this as the InS heuristic evaluation function, which we discuss in greater detail in Appendix E.

Perhaps the most obvious approach to real-time scheduling is to first solve for the optimal schedule and then execute it. We call this approach solve-then-execute branch-and-bound (steBnB). This is the best thing to do when the cost of computation is very small relative to the cost of execution. By computation, we mean the time spent finding a schedule, whereas by execution we mean the actual time required to process the jobs on the machines. For the sum-finishing-time cost function, this is usually not a very useful approach because the initial time needed to find the optimal solution is added to the finish time of all the jobs being scheduled. The multiplier of this penalty increases linearly with the number of

jobs, and the average time to find the optimal schedule grows exponentially with the number of jobs, eventually making this approach very costly.

Our implementation of the solve-then-execute (steBnB) algorithm explores the space of possible schedules using the InS heuristic evaluation function to both order the node expansions under a node, and to facilitate pruning of partial schedules. Once the branch-and-bound exploration returns an optimal solution, the sum-finishing-time cost of this solution is calculated. This cost includes the processing time used to find the optimal solution minus an initial processing time allotment. Although this algorithm will obviously perform poorly as the number of jobs increases, we present it as a baseline for comparison.

One drawback of the solve-then-execute approach is that the time spent finding the optimal solution is wasted, because no execution is performed during this computation time. At the very least, we could process some of the jobs while we are figuring out the optimal schedule, and then execute the remaining jobs according to the optimal schedule when it becomes available. One such algorithm is called truncated branch-and-bound (tBnB) [Iba76]. TBnB is a branch-and-bound algorithm for finding suboptimal solutions to combinatorial optimization problems, such as flowshop scheduling. It is most useful when there is only a limited amount of computation available to produce a complete solution. The idea is to perform a standard branch-and-bound search, returning the best leaf node found when the computation time runs out.

In order to adapt tBnB to real-time flowshop scheduling, we propose that the tBnB algorithm be performed, and when the time for the current decision runs out, the information available on the children of the current root node is used to schedule the next job. Our implementation of the truncated branch-and-bound (tBnB) algorithm explores the space of partial schedules using the InS heuristic for node-ordering and pruning. The exploration is stopped and a decision is made as soon as M_1 becomes idle, or until the space of possible schedules for the remaining jobs is completely explored. At this point, if only one child of the root node has been explored, then the job associated with the child node that has the lowest InS heuristic cost is scheduled for processing (a *greedy* decision). Since the exploration is ordered using the InS heuristic, the greedy scheduling decision will be towards the part of the problem-space tree that is currently being explored. If more than one child of the current root node has been completely explored when M_1 becomes idle, then the decision is to schedule the first job along the path to a minimum-cost leaf node. Once this occurs, the remainder of the scheduling decisions are optimal with respect to the set of as yet unscheduled jobs. This is because the problem space below the last greedy decision has already been

completely explored.

Note that tBnB focusses all of its exploration effort on the bottom of the problem space tree, which corresponds to the last decisions, rather than focussing on the current decision. The solution generated by tBnB can be characterized as a sequence of greedy decisions (*i.e.,* a minimum-cost decision based on the estimated cost of all partial schedules that include one more job than the current partial schedule) followed by a sequence of scheduling decisions that are optimal with respect to the set of unscheduled jobs. For the sum-finishing-time cost function, the cost of the greedy decisions is weighted more heavily than the optimal decisions because the finishing times of earlier jobs affect the cost of all jobs that finish later. This has a direct impact on the performance of tBnB on this problem.

We have also implemented two different incremental search algorithms. The first incremental search algorithm is a version of idMINIMIN. For the real-time flowshop-scheduling problem, we assume that the iterative-deepening branch-and-bound exploration stops when the first machine becomes idle, or when the remaining problem space is completely explored. This is a reasonable assumption since, for any schedule where the first machine is idle, it is possible to construct a new schedule where the first machine is not idle that has the same or lower cost. At this point, the problem solver makes a MINIMIN decision by moving to a child of the root that is on a minimum-cost path in the explored part of the problem space. Our implementation of idMINIMIN stores the explored problem-space tree from one iteration to the next and uses the backed-up InS heuristic values to order the search in the next iteration. To simplify our implementation and analysis, we did not implement the estimated-cost MINIMIN decision method described in Chapter 7 for the flowshop scheduling problem. When M_1 becomes idle, the MINIMIN decision is to schedule the job associated with the first step toward the minimum-cost frontier node in the last completed branch-and-bound iteration. The frontier-node costs for the MINIMIN decision are calculated using the InS heuristic evaluation function.

Our incremental search approach to real-time flowshop scheduling is based on the following premise. We believe that computation is better spent trying to improve the quality of the current decision rather than trying to improve the quality of some possible future decision. For this reason, we propose an incremental search approach to real-time scheduling. The basic idea is spend the computation trying to improve the overall solution quality by improving the quality of the next decision. Our incremental search algorithms explore the problem space of possible partial schedules until the first machine finishes processing its current

job. At this point, the best choice for the next job is scheduled for processing on the first machine, and the job that was just finished is scheduled for processing on the second machine. Note that we have adopted the heuristic policy of always keeping the first machine busy, because we typically do not have the information needed to determine when it would be worthwhile to continue computation while the first machine is idle.

The second incremental search algorithm that we have implemented is an adaptation of our sampled-α-bound k-best search algorithm (sαk-best BnB). Our implementation of sαk-best BnB explores the problem space using the same iterative deepening approach that idMINIMIN uses, except that the BnB exploration keeps track of the $k = 2$ best frontier nodes seen under each child of current problem-space root node. Scheduling decisions are then made by calculating an estimate of the expected minimum root-to-leaf path cost below each child based on the two frontier-node heuristic costs. The estimate of the expected minimum root-to-leaf path cost is calculated using the equations derived in Chapter 8.

The main differences between the flowshop-scheduling problem space and the random-tree problem space come from the fact that the branching factor of the scheduling problem decreases with the depth of the tree, and the solution-cost function is also sensitive to the depth of the tree. For these reasons and because of preliminary experimental results, we decided to take a different sample of the completion-cost distribution for a single additional level of exploration at each level of the search tree. The result is a two-dimensional array of buckets that is indexed by the branching factor of the parent node, and also by the bucket number of the minimum parent-to-child edge cost. For this problem space, the edge cost is the difference between the parent and child heuristic estimates of the complete-schedule cost. The completion-cost distributions were sampled by calculating the minimum edge cost over all children of the node being sampled. For our implementation we used 200 buckets, and we initially loaded the buckets by sampling the minimum edge costs in 10,000 schedules.

When using the sum-finishing-time cost function and the InS heuristic, the maximum edge cost that could occur depends on the number of remaining jobs to be scheduled. Let t_{max} be the maximum processing time of a job task, and let r be the number jobs that remain to be scheduled. We can bound the maximum possible edge cost (C_{max}) as follows.

$$C_{max} \leq 2t_{max}r$$

We used this bound as the maximum possible edge cost for the sampling buckets in our implementation of sαk-best BnB.

9.4 Experiment and Results

In order to evaluate the relative performance of the four flowshop scheduling algorithms (steBnB, tBnB, idMINIMIN, and sαk-best BnB), we performed a set of experiments on randomly-generated flowshop scheduling problems. Problem instances were constructed by randomly assigning processing times to both tasks of each job in a set of jobs. All processing times were chosen independently from a set integers (*e.g.,* $\{0, ..., 100\}$). Computation time was measured in node generations, where a node generation is the process of creating and evaluating a node in the problem-space tree. For a given problem instance, a set of trials was performed by varying the number of node generations that could be performed in a single processing time unit. For example, a small number of node generations per processing-time unit corresponds to a relatively slow computer, whereas a larger number of node generations per processing-time unit corresponds to a computer that is fast relative to the flowshop machines. In addition, all four algorithms were given one-half of the maximum processing time for a single job to schedule the first job. The rationale for this decision was to reduce the boundary effect of processing for the initial decision, and in some sense to simulate the situation of a dynamic scheduling problem where n new jobs arrive and need to be added to the end of the current schedule.

We ran steBnB, tBnB, idMINIMIN, and sαk-best BnB on a set of 1000 randomly generated 16-job flowshop scheduling problems. We chose 16 jobs because steBnB takes too long to run on larger problems. The results in Figure 9.3 show that steBnB produces solutions whose average cost per job is much larger than the other three algorithms, as expected. We have re-plotted the results for tBnB and idMINIMIN in Figure 9.4. These results show that idMINIMIN and sαk-best are able to outperform tBnB on the 16-job random flowshop problem for a wide range of node generations per job. In particular, their improvement over tBnB increases as the relative of the speed of the computation to the flowshop processing increases (*i.e.,* higher average node generations per job).

From the graph in Figure 9.4, it is hard to distinguish between the performance of sαk-best BnB and idMINIMIN, although sαk-best BnB produces solutions whose average cost is slightly lower than the average cost of solutions produced by idMINIMIN. The results of a head-to-head competition are shown in Table 9.1. We observe that neither algorithm consistently wins a larger percentage of the head-to-head trials. MINIMIN wins a larger percentage of the head-to-head competitions when the average number of available node generations per job is between 250 and 5,000. sαk-best wins a larger percentage of the head-to-head trials when the average generations available per job is less than

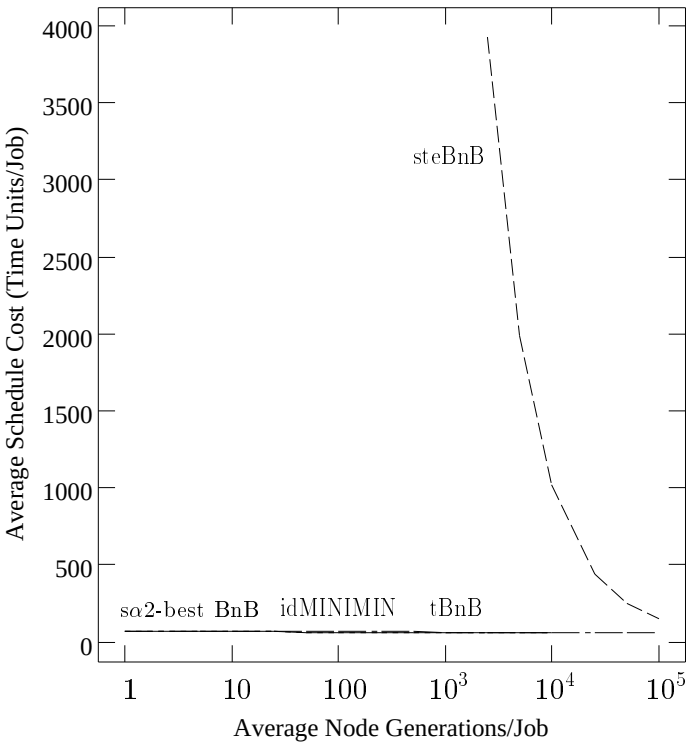


Figure 9.3: Average schedule cost versus average node generations per job.

250 and greater than 5, except for the case where the average number of available node generations is 10,000.

To further compare tBnB, idMINIMIN, and sαk-best BnB, we fixed the average number of generations per processing-time unit at 10 (*i.e.,* 500 generations per job on average) and repeated the random flowshop scheduling experiments for different numbers of jobs. The results in Figure 9.5 show that idMINIMIN and sαk-best BnB continue to outperform tBnB over many different problem sizes. For a constant number of generations per processing-time unit, the difference between the average tBnB schedule cost and the average schedule cost for idMINIMIN increases with the problem size. This is probably due to the fact that as the number of jobs increases, the initial greedy decisions made by tBnB affect the finish times of a greater number of jobs.

Up to this point, we have ignored the overhead cost of our k-best algorithms. Since tBnB and MINIMIN have very similar implementations, we expect their overhead to be very similar as well. Thus, to determine the effect of k-best's overhead on its solution quality, we performed an additional set of experiments to compare the performance of MINIMIN and sαk-best when both algorithms use approximately the same amount of time rather than node generations.

Average Generations Available	sαk-best BnB %wins	idMINIMIN %wins	ratio
0	0.00	0.00	—
5	1.40	2.10	0.666
10	2.90	2.80	1.035
25	5.60	3.40	1.647
50	6.30	5.40	1.166
100	8.70	8.10	1.074
250	7.50	8.10	0.925
500	6.00	7.20	0.833
1000	2.70	4.50	0.600
2500	2.60	3.60	0.722
5000	1.90	3.10	0.612
10000	2.10	1.50	1.400

Table 9.1: Percent wins for 16-job flowshop schedule for sα2-best versus. MINIMIN.

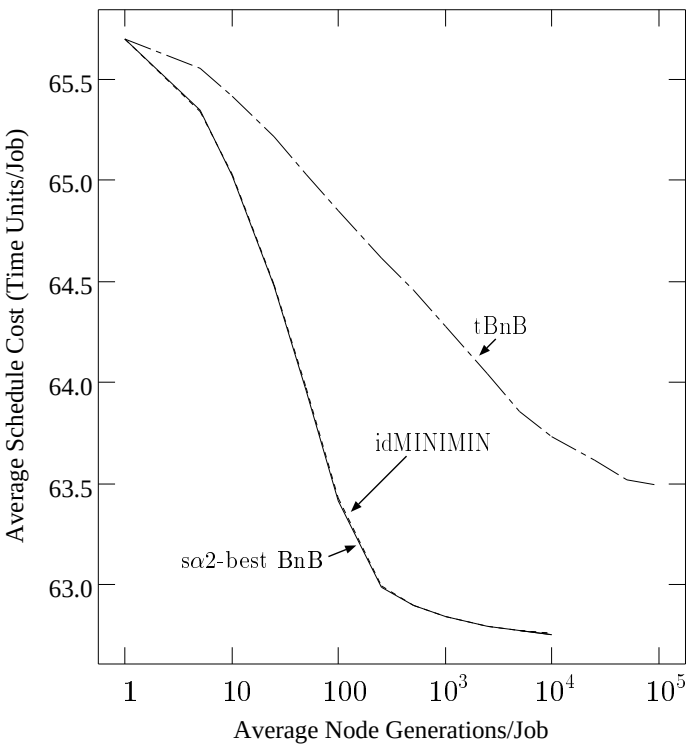


Figure 9.4: Average schedule cost versus average node generations per job (tBnB, idMINIMIN, and sαk-best BnB).

In an effort to reduce the overhead, we re-implemented sαk-best for the flowshop scheduling problem as follows. First, we used 10,000 trials to initialize the sample distribution. We then shut off the sampling code to reduce the overhead by not having to sample, and also so that we could precalculate the sample-based cumulative distribution. With these changes, the two remaining sources of overhead are solving the equation for the expected minimum path cost and maintaining the 2-best frontier-node costs for each child of the root node.

In order to balance the time used by idMINIMIN and sα2-best, we first calculated the ratio of sα2-best BnB's average time per node generation to idMINIMIN's average time per node generation using the streamlined implementation of sα2-best. We then adjusted the number of node generations available per time unit for idMINIMIN by this ratio (*i.e.*. idMINIMIN got to use 1.07946 times the number of node generations per time unit used by sα2-best BnB). We then ran both sαk-best BnB and the adjusted idMINIMIN on the previous set of 1000 randomly generated 16-job flowshop scheduling problems for different numbers of generations per schedule time unit. Figure 9.6 shows the average schedule cost versus the average run time used on a Sun Sparc 20. Points to the left model the

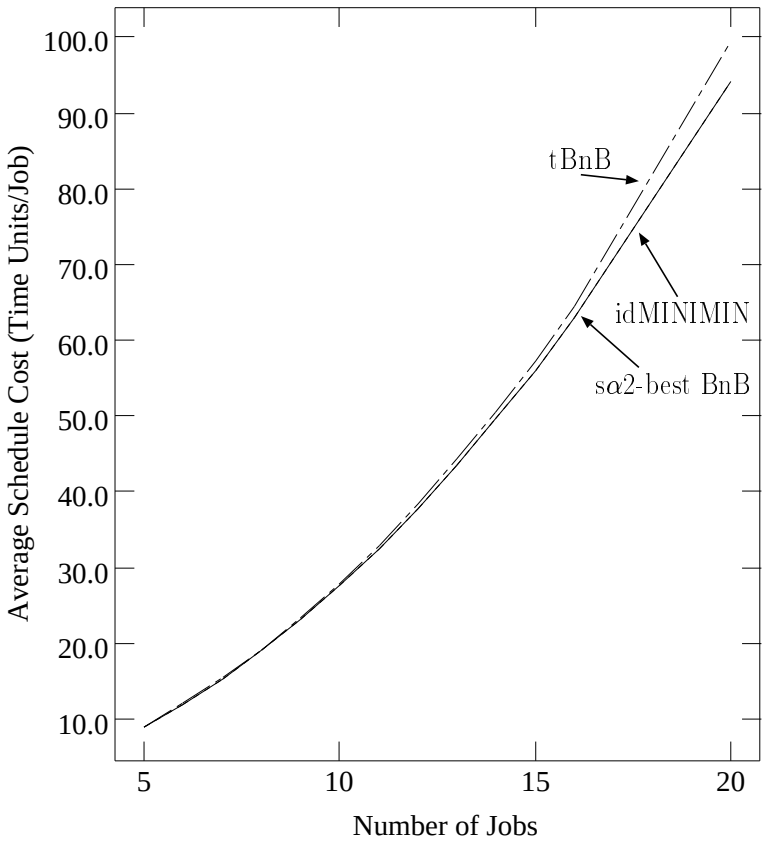


Figure 9.5: Average schedule cost per job versus number of jobs in schedule.

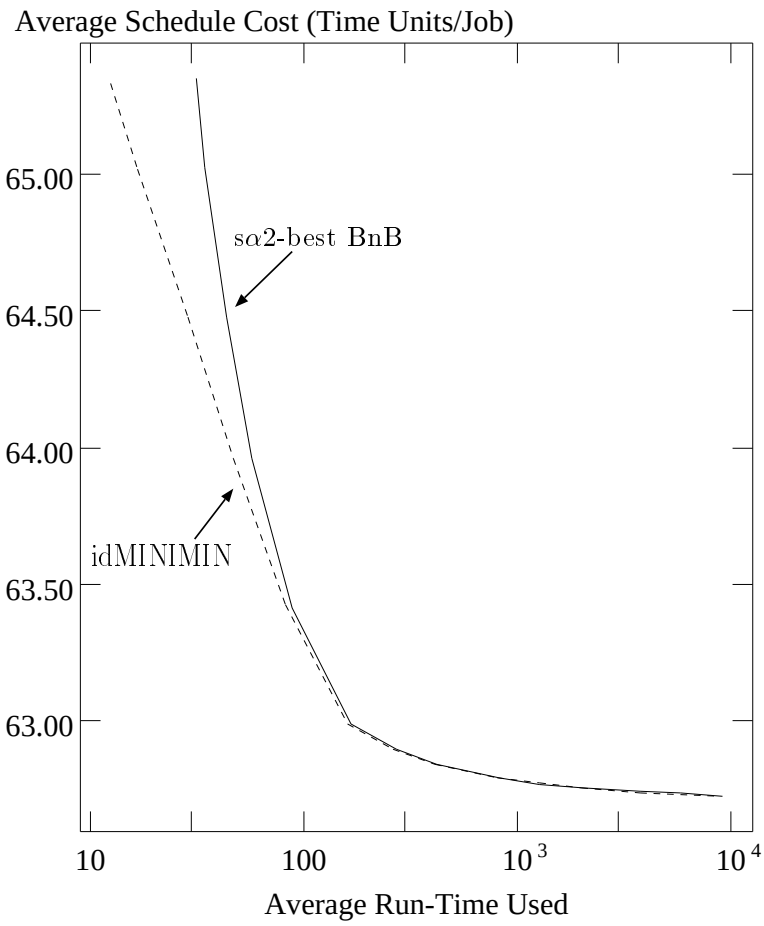


Figure 9.6: Average schedule cost per job versus average run-time used.

case where the computer is slow relative to the processing time, and points to the right correspond to the case where the computer is fast relative to the processing time, and thus it can explore more of the problem space and consequently make better decisions in the time available.

The results show that on the average idMINIMIN is able to make better quality decisions than sα2-best when the cost of making a decision is included in the computation bound. This difference is particularly evident when the computing resources are particularly constrained (*i.e.,* the left most points), in which case a few additional node generations can provide a large amount of additional information to the decision maker.

CHAPTER 10

Related Work

This chapter consists of two sections. The first section focusses on the related work that provided the background for our research. The second section contains a discussion of other work that is relevant to our research.

10.1 Background

In this chapter, we summarize background research that has directly impacted the direction of our research. We first summarize Mutchler's work on searching with limited resources. Next we discuss Korf's approach to real-time search. Lastly we present a brief summary of Russell and Wefald's approach to real-time decision making.

10.1.1 Searching Random Trees with Limited Resources

Much of our work on using random trees to analyze real-time decision making was motivated by David Mutchler's work on how to optimally allocate scarce search resources [Mut86]. In his work, he used a random-tree to model the *complete solution problem* under real-time constraints. Mutchler looked at the following problem: *Given that we can learn the value of only k edge costs in a random tree, if the objective is to find a least-cost root-to-leaf path, then which edge costs should we learn?* Note that learning an edge cost can be viewed as equivalent to generating a node in the search tree. If k is large enough to generate the complete search tree, then the problem can be solved optimally using an existing traditional search algorithm. The difficulty occurs when k is much less than the number of node generations needed to make an optimal decision. We previously discussed Mutchler's work in Chapters 2 and 3.

Our work differs from Mutchler's analysis of searching with limited resources in two ways. First, Mutchler focussed on the problem of how to explore the search tree given a limited number of node generations, for the case where a complete solution must be produced after the node generations have been ex-

hausted. We focus instead on the sequential decision problem where the solution path is produced by a sequence of decisions interleaved with a limited amount of exploration. The complete solution problem with k node generations is a special case of the sequential decision problem where the first decision has k node generations available, and each subsequent decision has zero additional node generations available.

There is a subtle yet important difference between Mutchler's complete solution problem with limited search resources, and our incremental real-time decision problem. For the Mutchler's complete-solution problem, the optimal strategy is to choose an arbitrary leaf node below the frontier node with the lowest expected path cost. The first step on this path is equivalent to the expected-cost MINIMIN decision. In contrast, the optimal decision for the incremental search problem is to the *child* of the root that has the lowest expected complete path cost. Thus Mutchler's decision problem can be solved by looking at the frontier nodes, whereas the optimal incremental decision problem must consider the effect of the frontier-node costs, and the distribution of paths in the unexplored part of the problem space, on the expected path cost of the children of the root node.

The second way that our problem differs from the problem investigated by Mutchler is that he assumed that edge costs are either zero or one with known probability, whereas we assume that edge costs are drawn from a larger set or even a continuous range of values with a known probability distribution. This extension makes the results more generally applicable, although the analysis is more difficult.

10.1.2 Real-Time Search

The idea of exploring to a limited search depth and then committing to moves in constant bounded time has existed in two-player game research since the initial work of Shannon [Sha50]. To our knowledge, Korf [Kor90] was the first to apply these ideas to single-agent heuristic search problems. Real-time A* (RTA*) is the first example of a single-agent incremental search algorithm, and has served as a strong motivation for our approach. The MINIMIN algorithm also serves as a benchmark algorithm for our experiments.

The idea behind the development of RTA* was to apply the assumptions of two-player games, namely exploring with a limited search horizon and committing to moves in constant bounded time, to single-agent problems. In some cases, the limited search horizon may be due to a constraint of the problem. For example, a robot in an office building cannot feasibly consider actions involving objects

in an adjacent room until it enters that room and determines what objects are there. Alternatively, the constant time bound on choosing an action could result from trying to solve real-world problems. For the robot navigation example, even if the robot can see into all the adjacent rooms, it will make sense to commit to intermediate moves when the cost of waiting to execute in order to find an optimal solution is large relative to the cost of executing a suboptimal solution.

Korf also proposed a best-first real-time search algorithm called time-limited-A*. The idea is to perform A* until time runs out, and then commit to the first move along the path toward the best node on the OPEN list. The OPEN list is the set of nodes that have been generated but not expanded by a best-first exploration. The main problem with this approach is the exponential memory requirement.

To avoid the memory problem, Korf suggested a variation of IDA* [Kor85] called threshold-limited-IDA*. The idea is to run a single iteration of IDA* with a threshold that is a constant amount greater than the heuristic estimate of the current node. The end result of the lookahead phase will be a set of frontier nodes with heuristic values that are either equal to or greater than the threshold. Decisions are then made toward the frontier node with minimum estimated total cost that has the lowest node cost. Note that threshold-limited A* on the random-tree problems is similar to node-cost BFS, and thus it will also suffer from the swap pathology.

In general, Korf's work on real-time search algorithms has provided us with a good starting point, and with a better than expected benchmark (MINIMIN).

10.1.3 Decision-Theoretic Search (DTA*)

Russell and Wefald [RW91] present a general decision-theoretic framework for making real-time search decisions. In particular, they present a real-time search algorithm called DTA* that is based on decision-theoretic principles. DTA* consists of three parts. It uses the MINIMIN decision rule, committing to the first move along the MINIMIN path when moves are made. It also employs a *stopping criterion* to decide when to end the search phase and make a decision. Lastly, a *node-selection criterion* is used to decide which frontier node to expand next. The idea behind DTA* is that the best *node-selection criterion* is to expand the frontier node whose expansion has the greatest expected utility, and the best *stopping criterion* is to stop searching when no node expansions have an expected utility that is greater than some constant (*i.e.*, the *time-cost* of a node expansion).

Note that the DTA* approach is not optimal from a decision-theoretic stand-

point, but is based on four assumptions. The *constant time-cost assumption* is that the time cost of a single node expansion is equal to a constant. The *meta-greedy assumption* is that it is reasonable to evaluate the ultimate effect of a single computation step in isolation, and then choose to perform the computation with the highest immediate effect. The *single-step assumption* is that it is reasonable to determine the ultimate effect of a single computation as if it were the last computation performed before a decision. The fourth assumption involves the issue of *subtree independence*, namely that the value of a node expansion in one subtree will not affect the backed-up value of a node in some other subtree. In addition, DTA* assumes that MINIMIN is the optimal decision-theoretic decision rule. MINIMIN is only guaranteed to be optimal under certain conditions, such as when a single lookahead search is followed by a choice of a complete solution path, as in the problem considered by Mutchler.

For example, consider the tree in Figure 10.1. Here our notation follows [RW91]. At this point in the search, the child node labeled α has the lowest f-value of children of the root. The f-value of a frontier node is the heuristic estimate of the complete path cost through that node, and the f-value of an interior node is simply the minimum of its childrens' f-values. The second-lowest f-value root child is labeled β_1, and so forth up to the largest f-value root child which is labeled β_k. If search were to stop with this search tree, then α would be the decision, and the action associated with α would be executed. Let $\mathcal{L}_\alpha$ be the set of frontier nodes below α. As shown in the figure, M_0 is the set of frontier nodes n below α that have an f-value that is less than β_1. The observation illustrated by this figure is that, when the cost function is monotonic non-decreasing, the only node expansions that could possibly have a positive expected utility (by changing the root decision) are the expansion of nodes in M_0. If there are m nodes in M_0, then unless there are at least m node expansions left, there is no way for β_1 to become the best decision, thus we can safely choose α and save the remaining node expansions for the next decision.

As Russell and Wefald observed, the problem with accurately estimating the expected value of a node expansion is the interaction effect, namely the expected value of two or more node expansions may not be equal to the sum of the expected values of the individual node expansions considered in isolation. In spite of this observation, they make an additional set of assumptions for DTA* to make the algorithm tractable. First, they only consider expanding nodes in $\mathcal{L}_\alpha$. Next they only consider possible expansions of a frontier node by performing a full-width search to a prespecified depth. Lastly, they only consider schemes that spend an equal amount of computation on each frontier node to be expanded.

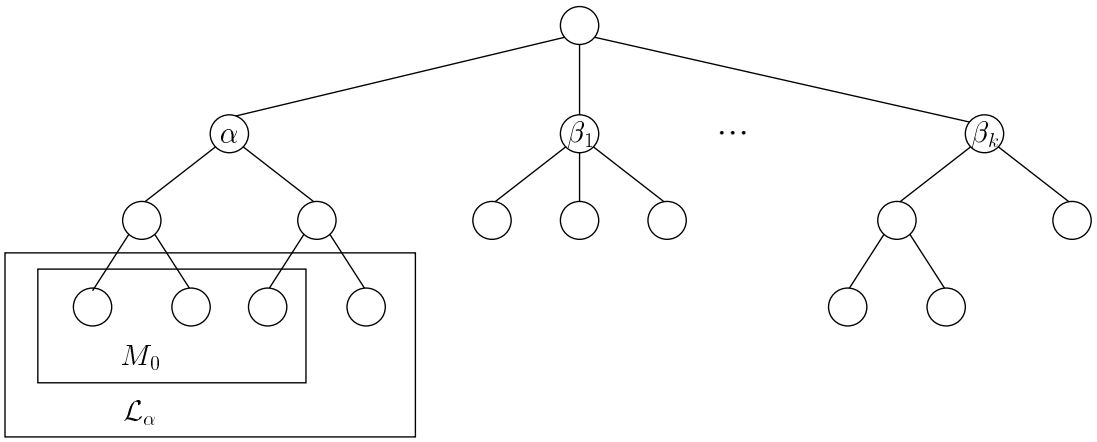


Figure 10.1: Example of DTA* node selection.

In summary, DTA* makes decisions using the MINIMIN approach of moving one step toward the best frontier node. Exploration is ordered using a decision-theoretic method with a number of simplifying assumptions described above. DTA*'s stopping criterion is to continue exploration until the point where the expected utility of continued exploration is less than the cost of performing the exploration. We discussed the problems with this approach in Chapter 7, where our analysis and experimental results showed that this collection of assumptions can lead to pathological search behavior.

Our research differs from Russell and Wefald's work in two important ways. First, we have observed that incremental search methods that use the same heuristic for choosing explorations and making move decisions can lead to pathological decisions. Second, we do not assume that MINIMIN makes optimal decisions, and thus we have considered a new approach to making search decisions. Thus our work can be seen to complement Russell and Wefald's work. Although they have presented a very general decision-theoretic framework for real-time search, they focused on the exploration problem, whereas we have focused on the decision-making problem.

If both approaches are applied to the same problem, we expect that Russell and Wefald's approach will perform better when the quality of the exploration is more important than the quality of the decision (*i.e.,* when the cost of exploring nodes is very high, or when the effect of the exploration on the decision critically depends on which exploration is performed). This is because the overhead cost of deciding where to explore will be small relative to the cost (or benefit) of actually performing the exploration. Alternatively, when the quality of the decision is more important than the quality of the exploration, we expect that our approach

will perform better because our decision method makes slightly better use of the available information than MINIMIN. Which is better will depend on the specific application, although our experience suggests that the overhead of a decision-theoretic exploration will outweigh any potential benefit.

10.2 Other Related Work

The remainder of the related work can be divided into four areas: random trees, resource-bounded decision making, learning the heuristic function, and scheduling.

10.2.1 Random Trees

A number of people have used random tree models that are similar to ours for analyzing and testing search algorithms. In addition, these models have been used to better understand the general search process.

Fuller *et al.* [FGG73] first used a random tree model to evaluate alpha-beta pruning in two player games. Karp and Pearl [KP83] considered the problem of finding an optimal path from the root to a leaf node in a uniform binary tree of known depth. Their random trees had edge costs of 1 or 0 with probability p and $(1 - p)$. Their analytical results concerned the expected complexity of algorithms for finding an optimal or near optimal solution as a function of p. Others [McD90b, ZK95, ZP94a] have extended the single-agent search model to real-valued edge costs and have further analyzed both the model and new algorithms based on this analysis. Our development of real-time search on random trees and our choice of edge-cost ranges has benefited from this prior work on random trees. For example, we chose a range of edge costs for our random trees so that the average-case search complexity of finding a minimum-cost root-to-leaf path is exponential in the search depth.

10.2.2 Resource-Bounded Decision Making

Dean and Boddy first coined the phrase *anytime algorithms* to refer to one type of time-dependent decision making in [DB88], and later developed this idea further in [Bod91, BD94]. The idea behind *anytime algorithms* is that they can be interrupted at any time during the computation and return a result whose quality is a non-decreasing function of the computation time. The main difference between anytime algorithms and our real-time incremental search algorithms is that

anytime algorithms address what we have referred to as the complete solution problem, whereas we have focussed on the incremental decision-making problem. For example, in the random tree model, an anytime algorithm would generate a complete root-to-leaf path. Real-time incremental search, on the other hand, focuses its attention on improving the quality of the next decision, generating the complete root-to-leaf path incrementally, while interleaving computation and execution.

In some sense, we can view the computation for each incremental decision as an anytime decision problem. Thus the difference between our incremental search algorithms and anytime algorithms is in the way that the real-time search problem is formulated. Anytime algorithms try to find the best complete solution under a time constraint, whereas our real-time incremental search algorithms try to find the best next decision under a time constraint.

Eric Horvitz [Hor90] has also investigated the problem of reasoning under resource constraints, which he called *flexible computation*. He has looked at many different aspects of the resource-bounded decision problem. In particular, he used decision theory to develop a normative approach to metareasoning. Metareasoning, also known as deliberation scheduling in [Bod91, DKK93b, DKK93a], and metal-level reasoning in [RW91], describes the process of deciding what computations to perform in support of a reasoning process. A general decision-theoretic approach to metareasoning is useful for analyzing problems and algorithms, but, as Horvitz notes, the improvement in resource allocation may not make up for the computation time spent performing the metareasoning. In other words, the computation needed to make a rational decision about where to perform computation may require nearly as much if not more time than if we just performed the computation in question. In addition, it is often the case that the information needed to make a more intelligent computational-resource-allocation decision is often not available. For our experiments, we made part of the metareasoning decision in advance (*i.e.,* the stopping criteria) when we chose to investigate problems that have strict decision deadlines measured in terms of node generations.

A large number of people have investigated similar problems (*e.g.,* [HM90] and others in the 1989 AAAI Spring Symposium on AI and Limited Rationality). One common thread in this work is a desire to understand the problem of how to operate under conditions where computational resources are limited. Three surveys of real-time problem solving techniques can be found in [DW91, SP94, Dea94].

Hansson and Mayer [HM90] have also proposed a decision-theoretic approach to search control. Their basic idea is to treat search as a decision problem. They

use heuristic values in the search tree to update a Bayesian network [Pea88] that is used to maintain the set of beliefs about the current search problem. Their BPS search algorithm uses the Bayesian network to calculate the expected effect of an exploration, and then choose the exploration with the best expected influence on the current decision. Their work is complementary to ours, in the sense that they have focussed on more effective exploration, and we have focussed on better decision making.

Sutton, Barto and others [Sut91, Sut90b, Sut90a, BBS93], have proposed a dynamic programming approach to incrementally generating plans in situations where the same or similar problems recur. This is a very interesting approach that can be easily extended to stochastic problem domains [BBS93]. Their work focusses on learning to solve a problem over many instances. Our work has focussed on solving a problem once, although our work on learning the problem-space distribution in Chapter 8 can be viewed as related to their work on learning. We have not yet considered stochastic problem domains in any detail (see our discussion of future work in Chapter 11).

10.2.3 Scheduling

In this section, we first discuss a previous application of MINIMIN to scheduling problems. We then briefly comment on the scheduling literature that we used in our research.

10.2.3.1 Real-Time Scheduling

Tadepalli and Joshi [TJ92] have also proposed using real-time incremental search methods for scheduling problems. In their paper, they first present a state-space formulation of job shop scheduling and then apply MINIMIN with bounded search-depth exploration as a method for finding suboptimal solutions quickly. Their real-time constraint was imposed by the programmer in order to measure the average solution quality as a function of the exploration search depth. The main difference between their work and ours is that they view real-time search as a method for generating suboptimal solutions quickly, whereas we also view real-time search as a method for making decisions in real time, by interleaving the scheduling process with the execution of the schedule.

10.2.3.2 General Scheduling References

In our experiments presented in Chapter 9, we used a flowshop scheduling heuristic function (InS) that was first presented by Ignall and Schrage [IS65]. Their objective was to find a way to better prune a branch-and-bound search in order to solve large scheduling problems optimally. Papadimitriou and Steiglitz [PS82] further discuss this problem in their book on combinatorial optimization. We found their treatment of the InS heuristic function easier to follow than the original treatment. Other good general references on scheduling problems include [CMM67], which contains detailed examples of different schedule cost functions, and [BCG76], which is a collection of papers that investigate the use of computers in scheduling domains.

Garey and Johnson [GS79] have compiled a number of complexity results on scheduling problems. In particular, they cite the results of a different Johnson [Joh54] who first showed that two-machine flowshop scheduling can be solved in polynomial time when the objective is to minimize the time that the last job finishes (*i.e.,* minimizing the makespan). Garey, Johnson, and Sethi [GJS76] later showed that the sum-finishing-time problem (*i.e.,* is there a schedule with sum-finishing-time less than a certain value) is NP-complete for the two-machine flowshop scheduling problem.

10.2.4 Related Search Methods

McAllester has also proposed a different search control method called conspiracy search [McA88]. His basic idea is to focus the search resources in locations where they are most likely to have an effect on the current decision. He introduced the idea of a *conspiracy number*, which is the number of frontier nodes whose value would have to change the current decision.

In some sense, the conspiracy number of the minimum-cost child can be viewed as a measure of our confidence in the statement that the node is the best decision. We see a strong similarity between the conspiracy idea for controlling exploration and our k-best algorithm for making decisions. In particular, when the number of distinct edge costs is small, it is likely that there will be more than one minimum-cost frontier node, in which case the k-best decision will be to the child node that is supported by the most (up to k) minimum-cost frontier nodes.

CHAPTER 11

Open Problems and Future Work

In this chapter, we present problems and issues that have come up during the course of this dissertation, but that we have not yet been able to address.

11.1 Extensions

Here we present several natural extensions of our research. In general, the extensions come from considering ways to relax the assumptions that we have made in the problem description and the random-tree model.

The question of when to stop performing computation and commit to a decision is still open. We avoided this question by assuming that the problem would dictate the appropriate time to stop exploring and make a decision through strict decision deadlines. The obvious relaxation of this assumption is to allow for a general computation-cost function and then require the decision maker to determine when it should to stop and make decisions over the course of generating a solution. The work of Goodwin [Goo94] is one example of what has been done on this problem, although there is still ample room for research on this topic.

In Chapter 9, we assumed that a good strategy for solving the real-time two-machine flowshop-scheduling problem is to keep the first machine busy. The justification is that an optimal schedule will always keep the first machine busy until all jobs are processed. This makes sense because the idle time on the first machine directly impacts the finish time of all unscheduled jobs. We have not addressed the question of how to decide whether to let the first machine be idle to allow a particular computation to finish, in order to improve upon the decision quality. For example, when the decision rule requires the exploration to be complete up to a fixed depth, then if we know that we are only a few nodes from completing the next level, we may be better off waiting for the new decision based on the few additional node expansions. Our intuition is that the amount of time necessary to decide whether or not to let the first machine be idle would be better spent trying to make a better quality decision in the first place. In general, the problem of when to stop computing and commit to a suboptimal decision is

still open.

We have also not addressed the issue of limited memory. Our real-time search algorithms in Chapter 7 assumed that there was sufficient space to store the whole search tree. This assumption is not without merit, since it is very likely to be the case that time will be a more serious constraint that space in a real-time search problem. Still, it is not inconceivable that space could be a constraint as well, so at some point it may be necessary to consider this problem in more detail. For example, if a computer can generate 10^5 nodes per second (assuming 100 instructions per node generation and 10^7 instructions per second) and has 10^7 node storage locations (assuming 100 megabytes and 10 bytes per node), then it will take 100 seconds before the memory is full. In this case, when the time available to make a decision is more than 100 seconds and the problem space contains more than 10^7 nodes, we will need to address the problem of limited memory.

When memory is a problem, then we can easily adapt depth-first branch-and-bound exploration to reduce its memory requirement. Since depth-first branch-and-bound is ordinarily a linear-space algorithm, the limited memory will only affect the use of stored values to order the exploration from one iteration of iterative deepening to the next. One alternative is to store as many of the backed up values as memory will allow, and then just use the static heuristic values to order the search in the parts of the problem space that we can't store. Another alternative that only requires space that is linear in the search depth is to just keep track of the path to a minimum-cost frontier node in each iteration. When the search depth is incremented, then the exploration to the new search depth first explores the frontier nodes at the end of the stored path from the previous iteration. The idea is that a minimum-cost frontier node in the current iteration is likely to be found below a node that was a minimum-cost frontier node in the previous iteration. Even if we don't find the minimum-cost frontier node at the end of this stored path, the frontier-node cost that we do find will typically provide us with a good bound for pruning the rest of the search tree. How much either of these ideas affect the pruning efficiency of depth-first branch-and-bound has not been explored.

The best-first exploration can also be adapted when memory is a problem, although the adjustment is not as clean as with depth-first branch-and-bound. One approach that we have used to make real-time decisions using best-first exploration when memory is limited is to use the fact that we have filled up the memory as an additional stopping condition. The basic idea is to explore the problem space as time permits until the OPEN list is full. At this point a decision

is made. This allows us to prune the irrelevant nodes from the OPEN list, freeing up space to explore for the next decision. Thus decisions are made when the time runs out or the memory becomes full. Other approaches have been considered in [CGA89, Rus92, SB89, MGP92, Kor93], for the general search problem.

Through the design of our random-tree model, and our choice of the flowshop-scheduling problem, we have focussed on problems where every leaf node is a solution, and the objective is simply to find a low-cost solution. Such problem spaces are sometimes referred to as *solution rich* [Smi93]. Many planning and scheduling problems do not have this characteristic, in which case much of the computation for these problems is spent finding a feasible solution, if one exists. Since we are interested in real-time decision making in general, we have thought about ways to overcome this limitation of our model. One way to address this problem might be to augment the problem space tree with edges from the dead-end (non-solution) leaf nodes to some restart state, with some associated cost. In this way, we can model recovery from failure. This is an appropriate change to the problem description when failure recovery is possible (*e.g.,* if a plane is not de-iced before its scheduled departure time, then it can be rescheduled for a later departure). Incremental search methods will still be appropriate for this modified problem space when the expected cost of restarting is much less than the cost of trying to find a feasible solution before executing.

This approach to handling recovery from failure raises another limitation, namely that we have focussed on problem-space representations that are trees. In many problem domains, including ones that allow recovery from failure, the problem space is a graph rather than a tree. We have not yet considered how to extend our incremental search methods to handle problem-space graphs any differently from trees.

If we are faced with a problem where failure recovery is not possible or practical, then we need know that a feasible solution is still reachable before executing any steps of the plan. The incremental search approach to this problem would be to first find a feasible solution, and then, while executing steps along the solution path, spend the computation available in ways that improve the quality of the next decision to be made. This is essentially an incremental version of local search (*e.g.,* [Joh90]). Whether failure recovery is possible or not, the general problem of making real-time decisions when all paths don't lead to a solution is still open.

One other limitation of the random-tree model is that there is a strong correlation between the node costs of siblings and between parents and children, and that this correlation is greater for nodes that are deeper in the tree. The intu-

ition is that the node costs of two siblings differ only in their edge costs to their shared parent, and as the tree depth increases, the relative importance of one edge cost on a longer path to the root decreases. In order to address the issue of sibling-node cost correlation, it has been suggested [Cra94] that the random-tree model can be updated to allow the range of edge costs to change with the depth of the tree. If the edge-cost range increases with the depth of a node, then the correlation between sibling node costs will be reduced. In fact, if the edge-cost range increases exponentially with the node depth, then the node costs of siblings will be uncorrelated because the last edge cost on the path from the root node will dominate the node cost. Likewise, if the edge-cost range decreases with the depth, then the sibling node costs will be even more correlated as the depth increases than in the original random-tree model. We have not yet experimented with this variation to our model, but we expect that it will help us to develop more general real-time search algorithms, because it will allow us to relax one of the assumptions in our random-tree model.

One additional extension of the random-tree model would be to allow for stochastic events rather than just deterministic execution of the schedule as specified (*i.e.,* a Markov Decision Problem [Dea91a, BBS93]). One example is if the time required to process a job was not a fixed constant, but rather was described by a continuous probability distribution over a range of processing times. Alternatively, we could have a stochastic element in the scheduler itself, where there is a probability that the job scheduled is actually the job that is executed. These extensions would help make our model more applicable to real-world problems. We conjecture that k-best will outperform MINIMIN in cases where there is stochastic noise in the heuristic function, since MINIMIN depends on the accuracy of a single heuristic value. k-best, on the other hand, will be less susceptible to noise in the heuristic function because it bases its decision on the k-best frontier-node costs under each decision.

One other issue that we have not addressed is the question of making real-time decisions in a dynamic environment. In the random tree model, a dynamic environment might be modeled as the value of edge costs changing over time. In the dynamic flowshop-scheduling problem, new jobs are added and old jobs are removed as the schedule is being executed. Since the real world is dynamic, we eventually need to extend our analysis and experimental investigation to problems that can change or evolve as they are being solved.

11.2 Open Issues

Our heuristic-search formulation of the flowshop-scheduling problem contained a single source of heuristic information, namely the InS heuristic function. In other problem domains, there may be more than one heuristic function with different accuracies and computation costs. The question at this point is given a fixed amount of computation, which heuristic function should be used when. Thus far, we have only considered the single heuristic question, namely where to apply the heuristic function. When there is more than one heuristic function available, the exploration question becomes which function to use as well as where to apply it. The decision-theoretic answer is to perform the exploration with the highest expected return, but this begs the questions of how to determine the expected return, and still does not directly answer the question of what is the best thing to do under real-time constraints.

11.3 Future Plans

Our k-best approach to decision making is constructed through a series of assumptions that follow from our analysis of the last incremental decision problem. We have not yet applied the same sort of analysis to the second-to-last incremental decision problem, or the third-to-last, *etc.* Perhaps a more reasonable way to think of the general incremental decision problem is the limiting case of the n^{th}-to-last decision problem as n approaches infinity. Alternatively, we could model how many more incremental decisions are left, and adjust the decision making accordingly. Clearly this makes sense, because the likelihood of making up for bad choices with more opportunities for a low-cost completion is greater when there are more decisions left in which to make up the difference.

MINIMIN decisions are optimal if the frontier nodes are leaf nodes. $E(MPC)$ decisions are optimal if the next exploration phase will explore the remainder of the search tree, and edge costs are chosen independently from a known distribution. The next step is to extend this analysis one more level by trying to come up with the optimal decision strategy for the second-to-last incremental decision, which is the situation where there are two incremental decisions left before the problem space is completely explored. This is more difficult than the last incremental decision problem for a number of reasons. One is that for the last incremental decision problem, we can safely assume that the remainder of the decisions would be optimal. In the second-to-last incremental decision problem, the optimal second-to-last decision depends on how the last incremental decision

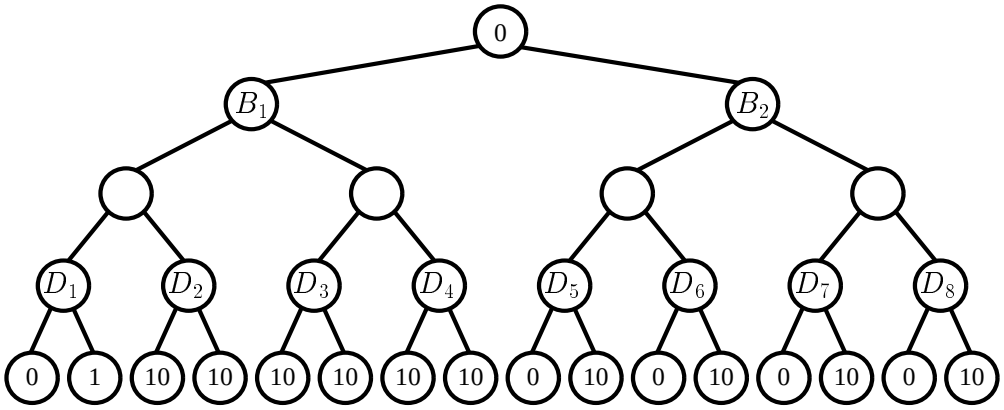


Figure 11.1: Example tree where the proximity of the second best frontier node cost affects the incremental decision.

will be made. Another reason is that the structure of the search tree can affect the optimal decision as we demonstrated in Chapter 2.

These are just two of the possible factors that could affect the second-to-last decision problem. Our point is that there are a number of additional factors that affect the second-to-last incremental decision, and still more that affect the n^{th}-to-last incremental or the general incremental decision. How these factors should be combined to make an optimal incremental decision in general is still an open problem that we plan to investigate in the future.

We also have not investigated the question of how the exploration and decision making interact. For example, if we know that we will probably end up at D_1 in Figure 11.1 (after first choosing B_1), then one advantage to choosing B_1 is that all the subsequent exploration can be focussed on the subtree below D_1. This will allow our decision at D_1 to be more informed than average, thus making D_1 a better choice because the decision quality at that level of the tree is likely to be better. This must be balanced with the fact that a move to B_2 will have more chances to avoid a set of high-cost edges in the unexplored part of the problem space. Making use of this sort of information requires interaction between the exploration and decision methods. The optimal combination of exploration and decision making is still an open question.

In addition to flowshop scheduling, we are interested in applying our incremental search to other real-time planning and scheduling problems. We have considered applying our incremental search methods to a real-time formulation of the traveling salesperson problem (*e.g.,* the robot courier problem in [Bod91]). We have also investigated the possibility of applying our methods to more general scheduling problems such as airline gate scheduling [Gre94], diverting airplanes

when an airport is closed [Cra94], and allocating medical resources during trauma care [Hor93]. Our plan is to investigate these and other applications as the next step of our research.

Another area that deserves more attention is the connection between our k-best algorithms and McAllester's work on conspiracy theory [McA88]. His basic idea is to focus the search resources in locations where they are most likely to have an effect on the current decision. He introduced the idea of a *conspiracy number*, which is the number of frontier nodes whose value would have to change the current decision. The idea of conspiracy search is to expand nodes in small conspiracy groups, because they are more likely to provide useful information.

McAllester focussed on the exploration problem for two-player games, whereas we have focussed on the decision problem for single-agent search. One possible combination of conspiracy theory and our work could be called conspiracy decision making. If frontier nodes are labeled as either good or bad, then the obvious first step is to choose the child node with the most good frontier nodes below it. Of course, this is not always the optimal decision strategy, as we demonstrated in Figure 11.1. In this example, B_1 has two good frontier nodes below it (labeled 0 and 1), whereas B_2 has four good frontier nodes below it. In this case, B_1 may be a better decision by virtue of the fact that node D_1 has more opportunities for a good path below it than any of the nodes in the D-level below B_2. One way to fix this problem might be to count the number of good frontier nodes in each subtree at some depth in the search tree, say j levels below the current decision, and then move one step toward the node at level j that had the most good descendants. In the example, if j is 3, then D_1 has two good frontier nodes in its subtree, and all other nodes in the D-level have fewer good frontier nodes in their subtrees. When the frontier nodes have more than just two possible values, then we could possibly calculate a weighted sum of quality of the frontier-node costs in a subtree, where the minimum-cost frontier nodes are counted more than frontier nodes with costs that are near-to-minimum. Although we believe that conspiracy-based decision making has potential, there is still work necessary to determine whether or not this is a reasonable approach.

Perhaps the most important possibility that we have not yet investigated is the idea of applying our analysis of $E(MPC)$ and its approximation algorithms to two-player games. Minimax is the two-player equivalent of MINIMIN, and the idea behind minimax is to model the player as a maximizer and the opponent as a minimizer, and then evaluate frontier nodes with respect to their value to the player at the root node. In a zero-sum game (where one player's gain is the other player's loss) such as chess or checkers, the opponent is trying to

minimize the player's gain and likewise the player is trying to maximize its own gain. Minimax exploration is typically performed using the two-player version of depth-first branch-and-bound, where the single-bound pruning is replaced by alpha-beta pruning. A natural extension of our work would be to analyze the last incremental decision in a two-player game, and then use this analysis to generate an α-βk-best extension of minimax with alpha-beta pruning. Whether a k-best minimax algorithm is practical or not remains to be seen. At the very least, an analysis of optimal decision making in this domain would provide some indication of how much opportunity there is for improving upon minimax with alpha-beta pruning. This would particularly important in light of the many less than entirely successful attempts to improve upon minimax with alpha-beta pruning (*e.g.,* [ACH90, Bau93, Ber79, KC94, Nau83, Nil69, McA88, Riv87, RW89]).

CHAPTER 12

Contributions and Conclusions

There are four main research contributions reported in this dissertation.

The first contribution is our formulation of the real-time incremental decision-making problem. The main characteristic of real-time incremental decision-making is that the solution can be generated one step at a time. This makes is possible to interleave planning and execution. We have broken up the problem of generating the next step of the solution into three subproblems: exploration, stopping, and decision making. We only briefly discussed exploration and stopping. Instead, we focussed our attention on the decision-making problem and developed an optimal algorithm for making the last incremental decision.

Our second contribution is our analysis of the optimal last incremental decision algorithm. We showed that making optimal last incremental decisions is impractical in general, due to the size and complexity of the expected minimum-path-cost equations.

Our third result concerns the performance of MINIMIN. We implemented the optimal last incremental decision algorithm ($E(MPC)$) on a set of small search trees, and observed that both $E(MPC)$ and MINIMIN choose the first edge along an optimal path greater than 90 percent of time. In addition, we observed that MINIMIN makes the same decision as $E(MPC)$ more than 95 percent of the time, and produces solutions whose average percent error (relative to optimal) is within a few tenths of a percent of the average percent error of $E(MPC)$.

Our fourth contribution is the k-best decision algorithm, which is an approximation of the $E(MPC)$ decision algorithm. The k-best algorithm is based on the observation that the minimum cost of a frontier node below a child node provides the most information about the expected minimum path cost below that child node, and that the next most important piece of information is the value of the second smallest frontier-node cost below that child, *etc.* We observed that MINIMIN makes the same decisions as k-best when $k = 1$, and that k-best is optimal when the number of frontier nodes in each subtree is less than or equal to k. Thus, k-best defines a spectrum of real-time decision algorithms between MINIMIN and $E(MPC)$. When the edge-cost distribution information is not

available, we demonstrated that it is possible to learn a discrete sample of the completion-cost distribution and then use this sample to calculate an estimate of the k-best decision. Our k-best algorithm performs slightly but consistently better than MINIMIN on a number of different decision problems.

In addition to these contributions, we have also made some general observations about MINIMIN and k-best that we will summarize below. These conclusions are meant to characterize the relative performance of MINIMIN and k-best under different conditions.

When the number of distinct edge costs is small (i.e., ≤ 10) the average relative improvement of αk-best over MINIMIN is greater than when the number of distinct edge costs is larger. The intuition is that, when the number of distinct edge costs is small, there is a greater chance of having more than one minimum-cost frontier node. In this case, MINIMIN will clearly make worse decisions than αk-best when there is more than one minimum-cost frontier node, because MINIMIN will choose between the minimum cost frontier nodes arbitrarily, while αk-best will choose the child that has the most minimum-cost frontier nodes below it up to k, and if two or more children have k or more frontier nodes, then it will choose between these children arbitrarily. Of course, it would be easy to modify MINIMIN to handle this problem, but the fact that we noticed this problem in the first place is due to our analysis of k-best.

We observed no appreciable variation in experimental results for trees with different constant branching factors and trees with randomly generated branching factors. As we discussed before, the reason for this may be that although MINIMIN ignores a greater number of frontier-node costs when the branching factor is larger, this may be negated by the fact that the number of possible ways to make up for a poor decision choice is also much greater. In addition, k-best is also a myopic decision strategy so it ignores more frontier-node costs in a fixed depth tree with larger branching factor as well.

MINIMIN and αk-best are equivalent (*i.e.,* make the same decisions) when the search depth is 1 (*i.e.,* a greedy decision). The difference in average solution-cost error between MINIMIN and αk-best grows with the search depth and with the number of decisions that both algorithms make (exclusive of the end game where both algorithms make optimal decisions once a minimum-cost frontier node is also a leaf node). Thus for a fixed tree depth, the relative improvement in solution quality of αk-best over MINIMIN initially increases with the search depth, and then decreases as the search depth approaches the depth of the tree.

As the tree depth increases relative to the search depth, αk-best's marginal improvement in decision quality can accumulate. Although the difference in

average solution cost of MINIMIN and αk-best is very small, αk-best wins a larger percentage of the head-to-head competitions, and this percentage grows with the tree depth.

Our real-time comparison of sα2-best BnB and idMINIMIN in Chapter 9 shows that, when the overhead costs are included, idMINIMIN's efficiency makes is a very strong competitor. The main advantages of idMINIMIN are that it has a very low overhead for the decision making relative to the exploration costs, and that it doesn't require additional information to make its decisions. The main disadvantage of idMINIMIN is that it ignores potentially useful information. At least for the flowshop-scheduling problem, and we suspect for many other problems, the advantage that idMINIMIN gains from being able to search deeper in the same time as more sophisticated algorithms, more than makes up for the loss in decision quality that results from ignoring some of the available information. This is why we would recommend idMINIMIN as the first choice for real-time search problems. We can then use the procedure in Chapter 5 to bound the expected error of using MINIMIN, and use this bound to decide if a more sophisticated algorithm should be considered.

The main advantage of k-best BnB is that we can easily incorporate additional information from the search tree into the incremental decision at the root node. The main disadvantages of k-best BnB are the overhead of maintaining the additional information needed and the overhead of calculating the k-best decision based on this additional information. When the amount of exploration is limited by time, our results have shown that these overhead costs are too high, because they reduce the amount of exploration enough to lower the average decision quality of k-best BnB to below idMINIMIN's average decision quality. If instead the exploration is limited by physical bounds of the problem, then k-best decision making is a good way to spend any additional computation time in order to improve the expected decision quality.

We initially approached the real-time decision-making problem with the intention of finding an algorithm that makes better quality decisions than MINIMIN. The random-tree model was chosen in part because the information needed to make an optimal decision is readily available. Even under these conditions, which are in some sense ideal for the decision-theoretic approach, MINIMIN performed nearly as well as the optimal decision method. When we compared MINIMIN to our k-best approximation method, using time rather than node generations as the measure of computation, the small improvement in decision quality that k-best had obtained was more than overshadowed by the overhead it requires to calculate the more complicated decision. The bad news is that we did not find a

clear replacement for MINIMIN. The good news is that MINIMIN makes pretty good incremental decisions. Thus, perhaps the most general conclusion that we can draw from our results is that, for problem instances where time is the main constraint on the decision process, a fast algorithm that gathers more information with minimal processing (*e.g.,* MINIMIN) is more likely to make better quality decisions on average than a slower algorithm that gathers less information, even if the smaller amount of information is analyzed more carefully (*e.g.,* $E(MPC)$ or k-best). This is in-line with the experience in two-player games as well, where α-β minimax outperforms most slower but more sophisticated algorithms.

APPENDIX A

Generating Reproducible Incremental Random Search Trees

This appendix describes in detail the method we used to efficiently and reproducibly generate random trees. The first section is from [KPZ94]. The second section contains annotated code that implements the random tree generator.

A.1 The Method

In order to reproducibly generate a random tree without storing it, we first make the following observation. If we know the seed used to initialize the random number generator before the random tree was first generated, then the random value associated with the k^{th} random number can be regenerated by first initializing the random number generator with the initial seed, and then calling the random number generator k times. As might be expected, this approach is very inefficient.

In order to efficiently generate the random number associated with a given breadth-first index, we examined the details of the *rand()* function in the standard C library distribution [KR88]. Recall that the breadth-first index of a node is determined by the order in which nodes would be generated by a breadth-first search.

Given a seed s_i, the next seed is computed as $s_{i+1} = a \cdot s_i + c$, where $a = 1103515245$ and $c = 12345$ in this case, and all arithmetic is modulo 2^m, where m is the word size of the machine in bits. The random value returned is a subset of the bits of the seed. All linear congruential pseudo-random number generators are of this form, with different values for the constants a and c.

Expanding the recurrence further, we see that

$$s_{i+2} = a \cdot s_{i+1} + c = a^2 \cdot s_i + a \cdot c + c,$$

$$s_{i+3} = a^3 \cdot s_i + a^2 \cdot c + a \cdot c + c,$$

and in general,

$$s_{i+n} = a^n \cdot s_i + c \cdot \sum_{j=o}^{n-1} a^j$$

By successive squaring, a^n can be computed in $O(logn)$ time, using the recurrences $a^{2n} = a^n \cdot a^n$ and $a^{2n+1} = a \cdot a^{2n}$. Unfortunately, the familiar closed form solution for the sum of the powers of a cannot be used, since it involves division, and division is not valid in modular arithmetic, since only the least significant bits are retained. However, we can also compute $\sum_{j=0}^{n-1} a^j$ in $O(logn)$ time, using the recurrences

$$\sum_{j=0}^{2n-1} a^j = \sum_{j=0}^{n-1} a^j + a^n \cdot \sum_{j=0}^{n-1} a^j$$

and

$$\sum_{j=0}^{2n} a^j = \sum_{j=0}^{2n-1} a^j + a^{2n}$$

This allows us to move from one random seed to another that is n steps away in the sequence in $O(logn)$ time rather than $O(n)$ time.

Since a is a constant, this can be sped up further by about a factor of 2 as follows: Precompute and store the values for a^{2^i} for i from 1 to m, the word length of the machine. Then, to compute a^n, multiply together the powers a^{2^i} for those values of i where the i^{th} bit of n is equal to one. Similarly, precompute and store the sums of the powers a from a^0 to a^{2^i-1}, for i from 1 to m. Then, accumulate the sums of the powers of a from a^0 to a^{n-1} as follows. Initialize the accumulator to zero. Starting from the rightmost bit of n, bit 0, for each i, where bit i of n is one, multiply the accumulator by a^{2^i} and add the sum of the powers of a from a^0 to a^{2^i-1}. This scheme reduces the number of multiplications and additions from order $logn$ to the order of the number of one bits in the binary representation of n.

Note that all arithmetic above is unsigned integer arithmetic, and only the residues mod m are preserved. In a large tree, the node indices will eventually exceed 2^m and wrap around. No special action is taken when this happens, and the sequence of random values similarly wraps around when its length exceeds 2^m. Thus, the asymptotic complexity of jumping ahead in the random sequence as described above is only a constant, which is proportional to the number of bits m used to represent an integer in the machine, typically 32 bits.

Once the random value for one child is computed, its sibling's values are computed by successive applications of the random number generator. This makes node expansion almost as efficient as node generation. Nevertheless, evaluation is still the dominant cost of most search algorithms on a random tree, as it is in real problems as well.

A.2 Random Seed Code

```
/* This version of the code was written by Rich Korf.  The ideas came
 * out of several discussions between Rich and myself and make use of
 * ideas that came from discussions with Milos Ercegovac and Peter
 * Montgomery.
 */

/* This is an improved version of the NEXTRAND subroutine. It cuts the
 * number of multiplications roughly in half. INIT must be run once
 * before NEXTRAND can be called. */

#include <stdio.h>                      /* standard I/O library */

#define A 1103515245                            /* random number multiplier */
#define C 12345                           /* random number additive constant */

/* global variables: */
unsigned long int a2n[32];                         /* a2n[i] contains A^(2^i) */
unsigned long int s2n[32];    /* s2n[i] is the sum from n=0 to 2^i-1 of A^n */

/* INIT initializes the fixed tables */
void init (a2n, s2n)
  unsigned long int a2n[32];                       /* a2n[i] contains A^(2^i) */
  unsigned long int s2n[32];  /* s2n[i] is the sum from n=0 to 2^i-1 of A^n */
{
  int index;                                              /* index into array */

  a2n[0] = A;                                                    /* A^(2^0) */
  s2n[0] = 1;
  for (index = 1; index < 32; index++) {       /* for each element in array */
    a2n[index] = a2n[index-1] * a2n[index-1];             /* A^2n = A^n*A^n */
    s2n[index] = s2n[index-1] * a2n[index-1] + s2n[index-1];
  }
}

/* NEXTRAND takes a random number seed, and an integer N, and returns
   the Nth random seed after the given one. It computes it in time
   logarithmic in N. */
```

```
unsigned long int nextrand (seed, N)
  unsigned long int seed;                    /* initial random number seed */
  unsigned long int N;             /* number of intervening random numbers */
{
  unsigned long int an;                                          /* A^N */
  unsigned long int sn;       /* sum of the powers of A from A^0 to A^(N-1) */

  int i;                                              /* current bit of N */

  an = 1;                                                        /* A^0 */
  sn = 0;                                      /* sum of no powers of A yet */
  i = 0;                             /* start with least significant bit of N */
  while (N > 0) {                                    /* until N is exhausted */
    if (N & 1 == 1) {                                     /* LSB of N is one */
      an = an * a2n[i];                  /* multiply by A to the 2 to the i */
      sn = sn * a2n[i] + s2n[i];                /* see text for explanation */
    }
    N = N >> 1;                                    /* shift N right by one bit */
    i++;                                              /* increment bit counter */
  }
  return (an * seed + C * sn);                     /* return new random seed */
}
```

APPENDIX B

Expected Minimum-Path-Cost Equations

This appendix presents a detailed derivation of the equations used to implement the expected minimum-path-cost algorithm ($E(MPC)$). The first section describes how to calculate the expected completion-cost distribution for frontier nodes of the search tree. The second section then describes how to combine the frontier-node completion-cost distributions with the edge costs of the tree to find the expected minimum root-to-leaf path cost through a child of the root node. The next two sections demonstrate the difficulty in expressing the expected minimum root-to-leaf path cost equations. One contains the algebraic equations for the expected minimum path cost through a child of the root in a binary tree with a search depth of 3 and an unexplored depth of 1, and the other contains the Maple[1] session used to generate these equations for the case where edge costs are chosen independently from a uniform distribution over $[0, 1]$. At the end of this section, we develop the expected minimum root-to-leaf path cost equations for the case where the completion-cost cumulative distribution is of the form $F_{CC}(x) = 1 - e^{-\lambda x}$.

B.1 Calculating the Frontier-Node Completion-Cost Distribution

In this section, we describe how to calculate the cumulative distribution for the minimum-cost frontier-to-leaf path for the case where the unexplored depth is 2.

Consider the problem of finding the minimum-cost path in a tree with the following characteristics. The depth (d) is fixed, the branching factor (b) is fixed, and the edge costs are uniformly distributed over the closed interval $[0, 1]$ (*e.g.*, $f(z) = 1; 0 \leq z \leq 1$). As in the main body of this dissertation, we use $T(b, s, u)$ to denote a random tree with branching factor b, search depth s, and unexplored depth $u = d - s$. We are interested in the expected minimum path cost in such a tree.

[1] Maple is an interactive mathematical tool developed at the University of Waterloo.

B.1.1 E(Single Level Greedy Decision)

Before deriving the general equations for the expected minimum root-to-leaf path cost, we will first present a derivation of the expected minimum edge cost below a node, or equivalently the expected value of the minimum of b independent random values. We will first calculate the cumulative distribution for the minimum of b independent random values, and then use this to calculate the density of the minimum, and finally the expected value of the minimum of b independent random values. In each case, the final equation is for the case where edge costs are independently chosen from a uniform distribution over the range $[0,1]$ (*i.e.*, $F(x) = x,\ 0 \leq x \leq 1$).

The cumulative distribution for the minimum of b independent and identically distributed random variables can be expressed as follows.

$$\begin{aligned}
F(\min(x_1,...,x_b) \leq x) &= P(x_1,...,x_b \leq x) \\
&= 1 - P(x_1,...,x_b > x) \\
&= 1 - P(x_1 > x \text{and} x_2 > x \text{and} ... x_b > x) \\
&= 1 - P(x_1 > x) \cdot ... \cdot P(x_b > x) \\
&= 1 - (1 - P(x_1 \leq x)) \cdot ... \cdot (1 - P(x_b \leq x)) \\
&= 1 - (1 - F(x_1 \leq x)) \cdot ... \cdot (1 - F(x_b \leq x)) \\
&= 1 - (1 - F(x))^b \\
&= 1 - (1 - x)^b
\end{aligned}$$

The density function is then just the derivative of the cumulative distribution:

$$\begin{aligned}
f(\min(x_1,...,x_b) = x) &= \frac{d}{dx} F(\min(x_1,...,x_b) \leq x) \\
&= \frac{d}{dx} 1 - (1 - F(x))^b \\
&= \frac{d}{dx} 1 - (1 - x)^b \\
&= b(1 - x)^{b-1}
\end{aligned}$$

The expected value of the minimum of b independent and uniformly distributed random variables can now be calculated as follows.

$$\begin{aligned}
E(\min(x_1,...,x_b) = x) &= \int_0^1 f(\min(x_1,...,x_b) = x) \cdot x \cdot dx \\
&= \int_0^1 b(1 - x)^{b-1} \cdot x \cdot dx
\end{aligned}$$

$$
\begin{aligned}
&= b(1-x)^b\left(\frac{1-x}{b+1} - \frac{1}{b}\right)\Big|_0^1 \\
&= \frac{1}{b+1}
\end{aligned}
$$

B.1.2 General Recursive Equations for Expected Minimum Path Cost

We begin by defining some notation. Let $f_1(x)$ denote the probability density function for edge costs in a uniform random tree. Let $f_{\min_i}(x)$ denote the distribution for the minimum path cost in a tree of depth i. Let $f_i(x)$ denote the distribution for a minimum root-to-leaf path cost through one child of the root node in a tree of depth i. We similarly define $F_{\min_i}(x)$ and $F_i(x)$ as the corresponding cumulative distribution functions. We finally use $E_{\min_i}(x)$ to denote the expected minimum path cost in a tree with depth i.

For now, we will consider edge costs that are chosen independently from a uniform distribution over the range $[0,1]$, although the derivation applies to any edge-cost density function. For example, we can calculate the expected cost of a single edge as follows.

$$f_1(x) = 1; \quad 0 \leq x \leq 1$$

$$F_1(x) = x; \quad 0 \leq x \leq 1$$

$$
\begin{aligned}
E_1(x) &= \int_0^1 f_1(x) \cdot x \cdot dx \\
&= \frac{x^2}{2}\Big|_0^1 \\
&= \frac{1}{2}
\end{aligned}
$$

We now present the recursive formulation of the expected optimal path cost as a function of the depth in the search tree. The probability density function for the cost of a path through one child of the root in a tree of depth i is:

$$f_i(x) = \int_0^x f_1(w) f_{\min_{(i-1)}}(x-w) \cdot dw$$

Note that this equation is the convolution of the edge-cost density function ($f_1(w)$) with the density function for a minimum-cost path of depth $i-1$.

From f_i, we can calculate the cumulative distribution function (CDF) for the cost of a path through one child of the root:

$$F_i(x) = \int_0^x f_i(w) \cdot dw$$

Expected minimum-path-cost equations:

1. $E_{\min_i}(x) = \int f_{\min_i}(x) \cdot x \cdot dx$
2. $f_{\min_i}(x) = \frac{d}{dx} F_{\min_i}(x)$
3. $F_{\min_i}(x) = 1 - (1 - F_i(x))^b$
4. $F_i(x) = \int_0^x f_i(w) \cdot dw$
5. $f_i(x) = \int_0^x f_1(w) \cdot f_{\min_{(i-1)}}(x - w) \cdot dw$

Figure B.1: Summary of recursion equations for calculating the expected minimum path cost in a random tree.

We can then use this to calculate the CDF for the minimum root-to-leaf path cost over *all* children of the root:

$$F_{\min_i}(x) = 1 - (1 - F_i(x))^b$$

Finally, we can calculate the probability density function for the optimal path cost of a depth i tree:

$$f_{\min_i}(x) = \frac{d}{dx} F_{\min i}(x)$$

Given $f_i(x)$, we can calculate the expected value for the minimum path cost as follows:

$$E_{\min_i}(x) = \int f_{\min_i}(x) \cdot x \cdot dx$$

The general steps necessary to calculate the expected minimum path cost in a tree with depth d and branching factor b are summarized in Figure B.1

B.1.3 Detailed Example: $E(MPC)$ for a Depth-Two Binary Tree.

In this section, we develop the equations for the expected minimum path cost in a binary tree of depth two. What we want to know is the expected cost of the minimum of the four paths shown in Figure B.2.

The expected minimum path cost for the tree in Figure B.2 with $d = 2$ and $b = 2$ can be expressed as follows.

$$E(\min(x_1 + \min(y_1, y_2), x_2 + \min(y_3, y_4)))$$

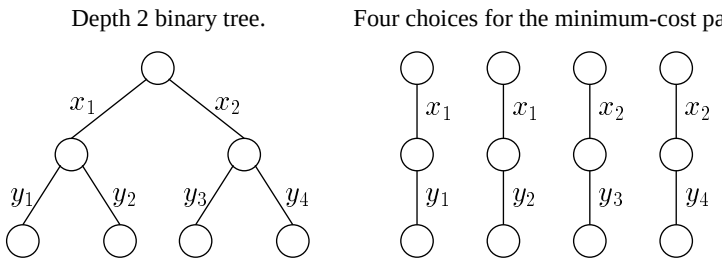


Figure B.2: Four choices for the minimum-cost path in a depth 2 binary tree.

$$
\begin{aligned}
&= E(\min(x_1 + y_1, x_1 + y_2, x_2 + y_3, x_2 + y_4)) \\
&= \int_{-\infty}^{+\infty} x \cdot f(\min(x_1 + y_1, x_1 + y_2, x_2 + y_3, x_2 + y_4) = x) \cdot dx
\end{aligned}
$$

Let $w_1 = x_1 + \min(y_1, y_2)$, and let $w_2 = x_2 + \min(y_3 + y_4)$. The cumulative density function can be expressed as follows.

$$
\begin{aligned}
F_{\min_2}(w) = F(\min(w_1, w_2)) &= p(\min(w_1, w_2) < w) \\
&= 1 - p(\min(w_1, w_2) > w) \\
&= 1 - P(w_1 \text{and} w_2 > w) \\
&= 1 - p(w_1 > w) \cdot p(w_2 > w) \\
&= 1 - (1 - p(w_1 < w))(1 - p(w_2 < w)) \\
&= 1 - (1 - F_2(w_1))(1 - F_2(w_2)) \\
&= 1 - (1 - F_2(w))^2 \\
&= 2F_2(w) - (F_2(w))^2
\end{aligned}
$$

where

$$
\begin{aligned}
F_2(w) &= p((x_1 + \min(y_1, y_2)) < w) \\
&= p((x_2 + \min(y_3, y_4)) < w)
\end{aligned}
$$

In order to calculate $F_2(w)$, we first need to know the density function for w. From Equation 5 in Figure B.1, we know that:

$$
f_2(w) = \int_0^w f_1(x) \cdot f_{\min_1}(w - x) \cdot dx \tag{B.1}
$$

When the edge-cost distribution is uniform over $[0, 1]$,

$$
f_1(x) = 1 \quad \text{and} \quad F_1(x) = x.
$$

The cumulative distribution for a minimum choice of two edges is calculated as follows.

$$\begin{aligned} F_{\min_1}(y) &= 1-(1-F_1(y))^2 \\ &= 2F_1(y)-(F_1(y))^2 \\ &= 2y-y^2 \end{aligned}$$

From this we can calculate the density function as follows.

$$\begin{aligned} f_{\min_1}(y) = f(\min(y_1,y_2)=y) &= \frac{d}{dy}F_1(y) \\ &= \frac{d}{dy}2y-y^2 \\ &= \begin{cases} 2-2y & 0 \le y \le 1 \\ 0 & otherwise \end{cases} \end{aligned}$$

We can now plug these into the convolution integral in Equation B.1:

$$\begin{aligned} f_2(w) &= \int_0^w f_1(x)\cdot f_{\min_1}(w-x)\cdot dx \\ &= \int_0^w 1\cdot(2-2(w-x))\cdot dx \\ &= \begin{cases} 2w-w^2 & 0 \le w \le 1 \\ 4-4w+w^2 & 1 \le w \le 2 \\ 0 & w \le 0; w \ge 2 \end{cases} \end{aligned}$$

At this point, we will adopt a superscript a or b to distinguish between the two different density functions that are produced by the convolution integral for different ranges of the path cost. The two different regions in the combined density function come from the fact that the two density functions in the convolution integral are discrete. The top line of the above equation describes the region where the overlap between the ranges two density functions is increasing, and the second line describes the region where the overlap between the ranges of two density functions is decreasing. The last line describes the regions where there is no overlap between the two functions. From $f_2^a(w)$ and $f_2^b(w)$, we can calculate the cumulative distribution for the sum of a single edge cost and the minimum choice over two edge costs below it (*e.g.*, $x_1+\min(y_1,y_2)$) as follows.

$$F_2(w_0) = p(w < w_0)$$

$$= \begin{cases} \int_0^{w_0} f_2^a(w)dw & 0 \le w_0 \le 1 \\ \int_0^1 f_2^a(w)dw + \int_1^{w_0} f_2^b(w)dw & 1 \le w_0 \le 2 \end{cases}$$

For case a $(0 \le w_0 \le 1)$,

$$\begin{aligned} F_2^a(w_0) &= \int_0^{w_0} 2w - w^2 dw \\ &= w_0^2 - \frac{w_0^3}{3} \end{aligned}$$

For case b $(1 \le w_0 \le 2)$,

$$\begin{aligned} F_2(w_0) &= \int_0^1 2w - w^2 dw + \int_1^{w_0} 4 - 4w + w^2 dw \\ &= (w_0^2 - \frac{w_0^3}{3})|_0^1 + (4w - 2w^2 + \frac{w^3}{3})|_1^{w_0} \\ &= \frac{w_0^3}{3} - 2w_0^2 + 4w_0 - \frac{5}{3} \end{aligned}$$

To summarize:

$$F(w_1) = F(w_2) = F_2(w) = \begin{cases} 0 & w \le 0 \\ w^2 - \frac{w^3}{3} & 0 \le w \le 1 \\ \frac{w^3}{3} - 2w^2 + 4w - \frac{5}{3} & 1 \le w \le 2 \\ 1 & w \ge 2 \end{cases}$$

We can now derive the cumulative distribution function for the minimum path cost in a depth-two random tree.

$$F_{\min_2}(w) = F(\min(w_1, w_2)) = 2F_2(w) - (F_2(w))^2$$

We must consider two cases ($0 \le w \le 1$ and $1 \le w \le 2$) because the equation for $F_2(w)$ depends on the range of w.

Case a: $(0 \le w \le 1)$.

$$\begin{aligned} F_{\min_2}^a(w) &= 2(w^2 - \frac{w^3}{3}) - (w^2 - \frac{w^3}{3})^2 \\ &= 2w^2 - \frac{2w^3}{3} - w^4 + \frac{2w^5}{3} - \frac{w^6}{9} \end{aligned}$$

Case b: $(1 \leq w \leq 2)$.

$$\begin{aligned} F^b_{\min_2}(w) &= 2(\frac{w^3}{3} - 2w^2 + 4w - \frac{5}{3}) - (\frac{w^3}{3} - 2w^2 + 4w - \frac{5}{3})^2 \\ &= \frac{1}{3}(\frac{-55}{3} + 64w - 80w^2 + 4w^3 - 20w^4 + 4w^5 - \frac{w^6}{3}) \\ &= 1 - \frac{(2-w)^6}{9} \end{aligned}$$

Given the cumulative distributions, we can next calculate the density function for the minimum path cost in a depth-two tree.

Case a: $(0 \leq w \leq 1)$.

$$\begin{aligned} f^a_{\min_2}(\min(w_1, w_2)) &= \frac{d}{dw}(F^a_{\min_2}(w)) \\ &= \frac{d}{dw}(2w^2 - \frac{2w^3}{3} - w^4 + \frac{2w^5}{3} - \frac{w^6}{9}) \\ &= 4w - 2w^2 - 4w^3 + \frac{10w^4}{3} - \frac{2w^5}{3} \end{aligned}$$

Case b: $(1 \leq w \leq 2)$

$$\begin{aligned} f^b_{\min_2}(w) &= \frac{d}{dw}(F^b_{\min_2}(w)) \\ &= \frac{d}{dw}(1 - \frac{(2-w)^6}{9}) \\ &= \frac{2}{3}(2-w)^5 \end{aligned}$$

Using these two equations, we can finally solve for the expected value of the minimum-cost path in a depth two random tree.

$$\begin{aligned} E_{\min_2}(w) &= \int_0^1 w \cdot f^a_{\min_2}(w) \cdot dw + \int_1^2 w \cdot f^b_{\min_2}(w) \cdot dw \\ &= \int_0^1 w \cdot (4w - 2w^2 - 4w^3 + \frac{10w^4}{3} - \frac{2w^5}{3}) \cdot dw \\ &\quad + \int_1^2 w \cdot \frac{2}{3}(2-w)^5 \cdot dw \\ &= \frac{391}{630} \\ &\approx 0.620635 \end{aligned}$$

This yields an average cost per edge along the path of $\frac{391}{1260} \approx 0.310317$. Note that this is less than the average cost of a single-level greedy decision ($0.\overline{33}$). This is reasonable because we expect the average cost of two greedy choices to be greater than the average cost of an optimal choice over two levels.

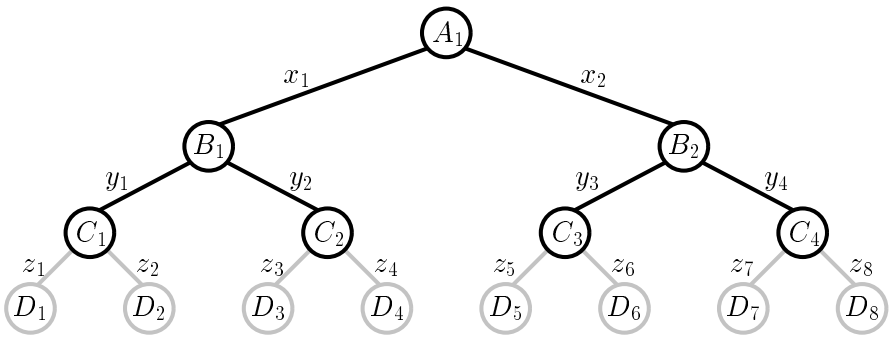


Figure B.3: Example tree for last incremental decision problem.

B.2 Calculating the Expected Minimum Path Cost

This section contains a detailed example of how to calculate the expected minimum path cost given the distribution of minimum path costs below the frontier nodes, and the edge costs in the explored search tree.

Consider the tree in Figure B.3. At this point, the costs of the black edges (x_1, x_2, y_1, y_2, y_3, y_4) are known, and the task is to choose between the left child (B_1) and the right child (B_2). Once a choice is made, then the remaining edge costs below the chosen node (either $z_1, ..., z_4$ or $z_5, ..., z_8$) will become known. The question is how does the expected minimum path cost through node B_1 depend on x_1, y_1, y_2, and the distribution of minimum path costs below the frontier nodes (C_1 and C_2)?

For the following calculations, we assume that the branching factor is two and that the edge-cost distribution is uniform over the range $[0, 1]$. Where practical, we will extend the derivation to the case of a random branching factor and arbitrary edge-cost distribution.

We denote the cumulative distribution for the minimum path cost through a frontier node C_1 as $F_{C_1}(z)$. From the previous section, we can calculate the distribution as follows.

$$\begin{aligned} F_{C_i}(z) &= F(\min[z_{2i-1}, z_{2i}]) \\ &= 2F_1(z) - F_1(z)^2 \end{aligned}$$

In this example, $F_1(z)$ is the edge-cost distribution because of our assumption that the depth of the tree is only one greater that the current search horizon. In general, $F_1(z)$ will need to be replaced by a more complex function or an approximation of this function for the distribution of completion costs.

For a constant branching factor b, the cumulative distribution for the completion cost of a path from node C_1 can be expressed as:

$$F_{C_1}(z) = F(\min[z_1, ..., z_b] = z) \quad = \quad 1 - (1 - F(z_i))^b$$

Note that we have assumed that the edge costs are drawn independently from identical distributions (*i.i.d.*). We have also assumed that all frontier nodes are at the same depth in the tree, thus the completion-cost distributions for all the frontier nodes are identical (*i.e.*, $F_{C_1}(z) = F_{C_2}(z) = F_{C_3}(z) = F_{C_4}(z)$). In this case, we can replace F_{C_i} with F_C.

Next, given the distributions for the completion costs of C_1, and C_2 and the edge costs y_1 and y_2, we can calculate the distribution of completion costs for a path through node B_1 as follows.

$$\begin{aligned}
F_{B_1}(y) &= p(\min[y_1 + \min[z_1, z_2], y_2 + \min[z_3, z_4]] < y) \\
&= 1 - p(\min[y_1 + \min[z_1, z_2], y_2 + \min[z_3, z_4]] > y) \\
&= 1 - p(y_1 + \min[z_1, z_2] > y) \cdot p(y_2 + \min[z_3, z_4] > y) \\
&= 1 - (1 - p(y_1 + \min[z_1, z_2] < y)) \cdot (1 - p(y_2 + \min[z_3, z_4] < y)) \\
&= 1 - (1 - p(\min[z_1, z_2] < y - y_1)) \cdot (1 - p(\min[z_3, z_4] < y - y_2)) \\
&= 1 - (1 - F_{C_1}(y - y_1)) \cdot (1 - F_{C_2}(y - y_2)) \\
&= F_{C_1}(y - y_1) + F_{C_2}(y - y_2) - F_{C_1}(y - y_1) \cdot F_{C_2}(y - y_2) \\
&= F_C(y - y_1) + F_C(y - y_2) - F_C(y - y_1) \cdot F_C(y - y_2)
\end{aligned}$$

If we assume that $y_1 < y_2$, then initially the distribution for the minimum path cost through node B_1 will depend entirely on the path through node C_1. More specifically, we can express the distribution of minimum path costs from node B_1 to a leaf node as a set of distributions for different ranges of values as follows.

$$F_{B_1}(y) = \begin{cases}
0 & y \leq y_1 \\
F_C(y) & y_1 \leq y \leq y_2 \\
F_C(y - y_1) + F_C(y - y_2) - F_C(y - y_1)F_C(y - y_2) & y_2 \leq y \leq y_1 + 1 \\
1 & y > y_1 + 1
\end{cases}$$

Similarly, we can express the distribution of the minimum path cost through

node B_2, assuming that $y_3 < y_4$.

$$F_{B_2}(y) = \begin{cases} 0 & y \leq y_3 \\ F_C(y) & y_3 \leq y \leq y_4 \\ F_C(y-y_3) + F_C(y-y_4) - F_C(y-y_3)F_C(y-y_4) & y_4 \leq y \leq y_3 + 1 \\ 1 & y > y_3 + 1 \end{cases}$$

In general, the minimum path cost through node B_1, when the tree has a constant branching factor b, can be expressed as follows. We assume without loss of generality that $y_1 < y_2 < ... < y_b$.

$$F_{B_1}(y) = \begin{cases} 0 & y \leq y_1 \\ F_C(y) & y_1 \leq y \leq y_2 \\ 1 - (1 - F_C(y-y_1)) \cdot (1 - F_C(y-y_2)) & y_2 \leq y \leq y_3 \\ 1 - (1 - F_C(y-y_1)) \cdot (1 - F_C(y-y_2)) & y_3 \leq y \leq y_4 \\ \qquad \cdot (1 - F_C(y-y_3)) & \\ \quad . & \\ \quad . & \\ \quad . & \\ 1 - (1 - F_C(y-y_1)) \cdot ... \cdot (1 - F_C(y-y_b)) & y_b \leq y \leq y_1 + 1 \\ 1 & y > y_1 + 1 \end{cases}$$

The next step in the process of deciding which child of the root to choose is to calculate the probability density function from the cumulative distribution function for minimum path costs. This requires that we know the edge-cost distribution. If the edge costs are independent and uniformly distributed over the range $[0, 1]$, then the edge-cost cumulative distribution is $F(z) = z$, and we can rewrite the minimum-path-cost distribution as follows.

$$F_C(y) = 2F(y) - F(y)^2 = 2y - y^2$$

$$F_{B_1}(y) = \begin{cases} 0 & y \leq y_1 \\ 2(y-y_1) - (y-y_1)^2 & y_1 \leq y \leq y_2 \\ 1 - (1 - (2(y-y_1) - (y-y_1)^2)) & \\ \qquad \cdot(1 - (2(y-y_2) - (y-y_2)^2)) & y_2 \leq y \leq y_1 + 1 \\ 1 & y > y_1 + 1 \end{cases}$$

We can now calculate the probability density function for the minimum path cost from a child of the root node as follows.

$$f_{B_1}(y) = \begin{cases} 0 & y \leq y_1,\ y > y_1 + 1 \\ 2 + 2y_1 - 2y & y_1 \leq y \leq y_2 \\ 2(1 + y_1 - y) \cdot (1 - 2y + 2y_2 + (y-y_2)^2) & y_2 \leq y \leq y_1 + 1 \\ \qquad +2(1 + y_2 - y) \cdot (1 - 2y + 2y_1 + (y-y_1)^2) & \end{cases}$$

With this probability density function, we can finally calculate the expected minimum path cost from a child of the root node. First, we can express the completion cost of a path from B_1 to a leaf node ($CC(B_1)$) as follows.

$$\begin{aligned} E(CC(B_1)) = \ & \int_{y_1}^{y_2} 2 \cdot (1 + y_1 - y) * y \ dy \\ & + \int_{y_2}^{1+y_1} 2((1 + y_1 - y) \cdot (1 - 2y + 2y_2 + (y-y_2)^2) \\ & \qquad + (1 + y_2 - y) \cdot (1 - 2y + 2y_1 + (y-y_1)^2)) \cdot y \ dy \end{aligned} \tag{B.2}$$

Note that when $y_1 = y_2$, $E(CC(B_1)) = y_1 + 0.2$, because in this case the choice is between the paths associated with the four leaf nodes (D_1, D_2, D_3, D_4). This is equivalent to the case where there is a single edge with cost y_1 followed by a node with branching factor 4. A greedy choice of 4 random variables that are uniformly distributed over [0,1] has an expected value of $1/(4+1) = 1/5$.

We can now use Equation B.2 and the analogous equation for $E(CC(B_2))$ to calculate the total expected minimum path costs ($MPC()$) for each root child.

$$E(MPC(B_1)) = x_1 + E(CC(B_1)) \tag{B.3}$$

$$E(MPC(B_2)) = x_2 + E(CC(B_2)) \tag{B.4}$$

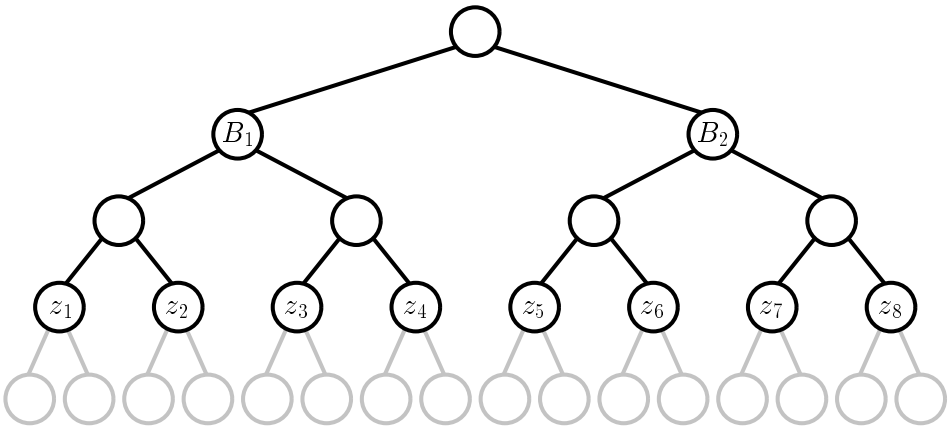


Figure B.4: Example tree $T(b = 2, s = 3, u = 1)$.

We can expand this equation for $E(MPC(B_1))$ to yield the algebraic equation for the expected minimum path cost from the root through node B_1.

$$\begin{aligned} E(MPC(B_1)) &= x_1 - y_1 y_2^2 - \frac{2}{3}y_2^2 + \frac{1}{3}y_2^3 - \frac{2}{3}y_1^2 - \frac{1}{3}y_1^3 + \frac{1}{2}y_1 + \frac{1}{1}y_2 + \frac{1}{3}y_1^3 y_2^2 \\ &\quad + y_1^2 y_2 - \frac{1}{6}y_1^4 y_2 + \frac{1}{5} + \frac{1}{30}y_1^5 + \frac{4}{3}y_1 y_2 + \frac{1}{6}y_1 y_2^4 - \frac{1}{3}y_1^2 y_2^3 - \frac{1}{30}y_2^5 \end{aligned}$$

B.3 $E(MPC(T(b = 2, s = 3, u = 1)))$

The first part of this section contains the algebraic equation for the expected minimum path cost below a child of the root in a tree with search depth of 3 and an unexplored depth of 1. For completeness, the second part of this section contains a trace of the Maple session that generated the algebraic equation in the first part.

B.3.1 The Algebraic Equation

Since we have assumed that this is the last incremental decision, we can ignore the individual edge costs, and just consider the node costs of the frontier nodes in the subtree of the root child being considered. The frontier-node costs are labeled z_i in the search tree shown in Figure B.4. We will present the equations for $E(MPC(B_1))$.

For the following equation, the frontier-node costs are ordered as follows: $z_a \leq z_b \leq z_c \leq z_d$ In order to use this equation, we will need to map the set

$\{z_1, z_2, z_3, z_4\}$ to $\{z_a, z_b, z_c, z_d\}$.

$$
\begin{aligned}
E(MPC(B_1)) = \\
& z_a - \frac{3}{2} z_b z_c^2 z_d^2 + \frac{1}{4} z_b + \frac{1}{4} z_c + \frac{1}{4} z_d - \frac{4}{105} z_b z_c z_d^7 + \frac{4}{3} z_b z_c z_d + \frac{4}{5} z_b z_c^2 z_d \\
& - \frac{16}{5} z_b z_c z_d^2 + \frac{4}{5} z_b^2 z_c z_d - \frac{6}{7} z_b^2 + \frac{1}{3} z_b^3 + \frac{4}{5} z_b z_c z_d^5 - \frac{1}{30} z_b^5 \\
& - \frac{6}{7} z_c^2 - \frac{1}{2} z_b z_c^4 + \frac{4}{3} z_b z_c^3 + \frac{1}{3} z_b^2 z_c^3 + z_c^3 - \frac{2}{3} z_c^4 + \frac{1}{5} z_c^5 \\
& - \frac{5}{3} z_b z_c^2 - \frac{4}{5} z_b^2 z_c^2 + \frac{1}{2} z_b^2 z_c^2 z_d - \frac{1}{105} z_c^7 + \frac{1}{30} z_b z_c^6 - \frac{1}{30} z_b^2 z_c^5 \\
& - \frac{6}{7} z_d^2 - \frac{1}{2} z_b z_c^2 z_d^4 - \frac{8}{3} z_b z_c z_d^4 - \frac{1}{2} z_b^2 z_c z_d^4 + \frac{4}{3} z_b z_c^2 z_d^3 \\
& + \frac{1}{3} z_b^2 z_c^2 z_d^3 + 4 z_b z_c z_d^3 + \frac{4}{3} z_b^2 z_c z_d^3 - \frac{3}{2} z_b^2 z_c z_d^2 - \frac{2}{3} z_b^2 z_c^2 z_d^2 \\
& + \frac{1}{30} z_b z_c^2 z_d^6 + \frac{1}{30} z_b^2 z_c z_d^6 + \frac{5}{3} z_d^3 - 2 z_d^4 + \frac{3}{2} z_d^5 - \frac{2}{3} z_d^6 + \frac{1}{7} z_d^7 \\
& - \frac{2}{3} z_c^2 z_d^4 - \frac{1}{3} z_b z_d^6 - \frac{1}{3} z_c z_d^6 + \frac{4}{3} z_b z_d^5 + \frac{1}{5} z_b^2 z_d^5 + \frac{4}{3} z_c z_d^5 \\
& + \frac{1}{5} z_c^2 z_d^5 - \frac{5}{2} z_b z_d^4 - \frac{5}{2} z_c z_d^4 - \frac{2}{3} z_b^2 z_d^4 + \frac{8}{3} z_b z_d^3 + \frac{8}{3} z_c z_d^3 + z_b^2 z_d^3 \\
& + z_c^2 z_d^3 - \frac{5}{3} z_c z_d^2 - \frac{4}{5} z_c^2 z_d^2 - \frac{5}{3} z_b z_d^2 - \frac{4}{5} z_b^2 z_d^2 - \frac{1}{252} z_d^9 + \frac{1}{84} z_b z_d^8 \\
& + \frac{1}{84} z_c z_d^8 - \frac{1}{105} z_b^2 z_d^7 - \frac{1}{105} z_c^2 z_d^7 - \frac{1}{30} z_b^2 z_c^2 z_d^5 + \frac{4}{7} z_b z_c + \frac{4}{7} z_c z_d \\
& + \frac{4}{7} z_b z_d + \frac{1}{3} z_b^2 z_c + \frac{1}{3} z_b^2 z_d + \frac{1}{3} z_c^2 z_d + \frac{1}{9}
\end{aligned}
$$

B.3.2 Maple Session to Produce $E(MPC(B_1))$ for $[0,1]$ Edge Costs

The following is taken from the interactive Maple session that generated the algebraic equations for $E(MPC(B_1))$ for the case where the edge costs are chosen independently from a uniform distribution over the range $[0,1]$. Some decimal constants were truncated in order to keep the output within the margins.

```
> maple
------------------------------------------------------------------------
    |\^/|      MAPLE V
._|\|   |/|_.  Copyright (c) 1981-1990 by the University of Waterloo.
 \  MAPLE  /   All rights reserved.  MAPLE is a registered trademark
 <____ ____>   of Waterloo Maple Software.
      |        Type ? for help.
------------------------------------------------------------------------
```

```
> FA1a := proc(x) 2*(x)-(x)^2 end;

FA1a := proc(x) 2*x-x^2 end

------------------------------------------------------------------------
> FA1b := proc(x) 1-(1-2*(x)+(x)^2)*(1-2*(x-zb)+(x-zb)^2) end;

FA1b := proc(x) 1-(1-2*x+x^2)*(1-2*x+2*zb+(x-zb)^2) end

------------------------------------------------------------------------
> FA1c := proc(x) 1-(1-2*(x)+(x)^2)*(1-2*(x-zb)+(x-zb)^2)
                   *(1-2*(x-zc)+(x-zc)^2) end;

FA1c := proc(x) 1-(1-2*x+x^2)*(1-2*x+2*zb+(x-zb)^2)
                 *(1-2*x+2*zc+(x-zc)^2) end

------------------------------------------------------------------------
> FA1d := proc(x) 1-(1-2*(x)+(x)^2)*(1-2*(x-zb)+(x-zb)^2)
                   *(1-2*(x-zc)+(x-zc)^2)*(1-2*(x-zd)+(x-zd)^2) end;

FA1d := proc(x) 1-(1-2*x+x^2)*(1-2*x+2*zb+(x-zb)^2)
                 *(1-2*x+2*zc+(x-zc)^2)*(1-2*x+2*zd+(x-zd)^2)
        end

------------------------------------------------------------------------
> fa1a := proc(x) diff(FA1a(x),x) end;

fa1a := proc(x) diff(FA1a(x),x) end

------------------------------------------------------------------------
> fa1b := proc(x) diff(FA1b(x),x) end;

fa1b := proc(x) diff(FA1b(x),x) end

------------------------------------------------------------------------
> fa1c := proc(x) diff(FA1c(x),x) end;

fa1c := proc(x) diff(FA1c(x),x) end

------------------------------------------------------------------------
> fa1d := proc(x) diff(FA1d(x),x) end;

fa1d := proc(x) diff(FA1d(x),x) end
```

```
------------------------------------------------------------------------
> empc4 := proc(x) int(fa1a(x)*x,x=0.0..zb) + int(fa1b(x)*x,x=zb..zc)
                 + int(fa1c(x)*x,x=zc..zd) + int(fa1d(x)*x,x=zd..1.0)
           end;

empc4 := proc(x)
             int(fa1a(x)*x,x = 0 .. zb)+int(fa1b(x)*x,x = zb .. zc)+
             int(fa1c(x)*x,x = zc .. zd)+int(fa1d(x)*x,x = zd .. 1.0)
         end

------------------------------------------------------------------------
> simplify(empc4(x));

             2   2
  - 1.5 zb zc  zd  + .25 zb + .25 zc + .25 zd

                           7                                    2
      - .0380952381 zb zc zd  + 1.33333334 zb zc zd + .8 zb zc  zd

                    2        2                     2
      - 3.2 zb zc zd  + .8 zb  zc zd - .85714286 zb

                     3              5                   5
      + .3333333333 zb  + .8 zb zc zd  - .03333333333 zb

                    2           4                    3
      - .85714286 zc  - .5 zb zc  + 1.333333333 zb zc

                     2   3     3                 4        5
      + .3333333333 zb  zc  + zc  - .6666666667 zc  + .2 zc

                      2        2   2        2   2
      - 1.66666666 zb zc  - .8 zb  zc  + .5 zb  zc  zd

                      7                   6                2   5
      - .009523809524 zc  + .033333333 zb zc  - .033333333 zb  zc

                    2           2   4                      4
      - .85714286 zd  - .5 zb zc  zd  - 2.666666667 zb zc zd

             2      4                    2   3
      - .5 zb  zc zd  + 1.333333333 zb zc  zd
```

```
              2   2   3              3               2      3
+ .33333333 zb  zc  zd  + 4. zb zc zd  + 1.3333333 zb  zc zd

        2      2               2   2   2
- 1.5 zb  zc zd  - .666666667 zb  zc  zd

                  2   6               2      6              3
+ .033333333 zb zc  zd  + .033333333 zb  zc zd  + 1.6666667 zd

       4         5                6                7
- 2. zd  + 1.5 zd  - .6666666667 zd  + .1428571429 zd

                2   4                    6                    6
- .6666666667 zc  zd  - .3333333333 zb zd  - .3333333333 zc zd

                   5        2   5                    5
+ 1.333333333 zb zd  + .2 zb  zd  + 1.333333333 zc zd

       2   5            4            4
+ .2 zc  zd  - 2.5 zb zd  - 2.5 zc zd

             2   4               3                3     2   3
- .6666667 zb  zd  + 2.66667 zb zd  + 2.666667 zc zd  + zb  zd

    2   3                2        2   2                2
+ zc  zd  - 1.666666 zc zd  - .8 zc  zd  - 1.666666 zb zd

       2   2                   9                    8
- .8 zb  zd  - .003968253968 zd  + .0119047619 zb zd

                  8                2   7                2   7
+ .011904762 zc zd  - .0095238095 zb  zd  - .0095238095 zc  zd

               2   2   5
- .033333333 zb  zc  zd  + .571428571 zb zc + .57142857 zc zd

                                   2                 2
+ .57142857 zb zd + .333333337 zb  zc + .33333334 zb  zd

               2
+ .333333337 zc  zd + .111111111
```

B.3.3 $E(MPC)$ When the Completion-Cost Cumulative Distribution is of the Form $(1 - e^{-\lambda x})$

In this section, we develop the expected minimum root-to-leaf path cost equations for the case where the completion-cost cumulative distribution can be written as follows.

$$F_{CC}(x) = \begin{cases} 1 - e^{-\lambda x}; & x \geq 0 \\ 0; & otherwise \end{cases}$$

Given this distribution for the completion-cost, we know that, for the last incremental decision problem, the cumulative distribution for the minimum root-to-leaf path cost (x) through a child (B_i) with frontier-node costs $z_1, ..., z_k$ can be expressed as follows.

$$\begin{aligned} F_{\min_k}(x) &= 1 - (1 - F_{CC}(x - z_1)) \cdot ... \cdot (1 - F_{CC}(x - z_k)) \\ &= 1 - (1 - (1 - e^{-\lambda(x - z_1)})) \cdot ... \cdot (1 - (1 - e^{-\lambda(x - z_k)})) \\ &= 1 - e^{-\lambda(kx - z_1 - z_2 - ... - z_k)} \\ &= 1 - C_1 e^{-\lambda k x} \end{aligned}$$

where x must be greater than or equal to all the z_i, and C_1 is a constant. At any given point where x is less than some x_i, then the above equation for $F_{\min_k}(x)$ will simply have fewer terms in it. Note that, although these equations are easier to express than the equations for the $[0, 1]$ edge-cost case, there is still a different minimum-path-cost distribution equation for each of the frontier nodes in the subtree of a child node.

Thus, if we are fortunate enough to have a completion-cost cumulative distribution that is of the form $F_{CC} = 1 - e^{-\lambda x}$, we can easily express the complete root-to-leaf path cumulative distribution using the above equation. If in fact we have sufficient resources to perform a set of integrals that is exponential in the depth of the tree, then we can use this distribution to calculate the expected minimum path cost for each child of the root node.

APPENDIX C

Monte-Carlo Simulation Results

In this appendix, we report the detailed results for the Monte-Carlo simulation experiments that are described in Chapter 5.

We performed a set of experiments on trees with different search depths (s), unexplored depths (u), and fixed branching factors (b). In all experiments, the tree was explored using depth-first branch-and-bound to a fixed depth (s), and then an estimate of the expected minimum path cost for each child of the root node was calculated by randomly generating the remainder of the problem-space tree, and then calculating an average of the minimum path cost below each root child over 1000 different samples. The average minimum path costs were then used to approximate the $E(MPC)$ decision. The remainder of the tree was then generated one additional time, and the minimum path cost below the chosen child of the root was recorded, along with the optimal path cost in the final problem space tree. Results were collected this way to separate the sampling phase from the decision making.

Tables C.1, C.2, C.3, and C.4 contain the percentage of the trials where $E(MPC)$ and MINIMIN make optimal decisions (% optimal), the average percent solution-cost error of the $E(MPC)$ and MINIMIN decisions relative to the optimal path cost (% error), and the percentage of the trials where $E(MPC)$ produced a lower cost solution than MINIMIN, and vice versa (% wins). Table C.1 is for binary trees with one unexplored level ($b = 2, u = 1$). Table C.2 is for trees with one unexplored level, and where each interior node has three children ($b = 3, u = 1$). Table C.3 is for trees with one unexplored level, and where each interior node has four children ($b = 4, u = 1$). Finally, Table C.4 is for binary trees that have two unexplored levels ($b = 2, u = 2$). The results include the 95% confidence intervals.

The percent solution-cost error is calculated using the following formula, and then averaged over all trials.

$$\%\ error \quad = \quad 100 * \frac{path\ cost(E(MPC)) - optimal\ path\ cost}{optimal\ path\ cost} \qquad \text{(C.1)}$$

In addition, Table C.5 shows the percentage of the trials that MINIMIN makes

Search Depth	$E(MPC)$ decisions			MINIMIN decisions		
	% optimal	% error	% wins $\pm$ 0.08	% optimal	% error	%wins $\pm$ 0.07
2	84.6 $\pm$ 0.8	3.12 $\pm$ 0.07	1.83	84.2 $\pm$ 0.9	3.25 $\pm$ 0.07	1.44
3	87.1 $\pm$ 0.8	1.78 $\pm$ 0.04	1.99	86.7 $\pm$ 0.8	1.88 $\pm$ 0.04	1.65
4	88.7 $\pm$ 0.7	1.15 $\pm$ 0.03	1.97	88.3 $\pm$ 0.7	1.26 $\pm$ 0.03	1.54
5	89.4 $\pm$ 0.7	0.86 $\pm$ 0.02	1.88	89.2 $\pm$ 0.7	0.93 $\pm$ 0.02	1.63
6	90.1 $\pm$ 0.7	0.69 $\pm$ 0.02	1.95	89.7 $\pm$ 0.7	0.76 $\pm$ 0.02	1.55
7	90.6 $\pm$ 0.7	0.56 $\pm$ 0.01	1.97	90.1 $\pm$ 0.7	0.62 $\pm$ 0.02	1.48
8	91.3 $\pm$ 0.6	0.45 $\pm$ 0.01	1.88	90.8 $\pm$ 0.7	0.51 $\pm$ 0.01	1.42
9	91.4 $\pm$ 0.6	0.40 $\pm$ 0.01	1.84	90.9 $\pm$ 0.7	0.45 $\pm$ 0.01	1.35
10	91.8 $\pm$ 0.6	0.35 $\pm$ 0.01	1.82	91.4 $\pm$ 0.6	0.39 $\pm$ 0.01	1.43

Table C.1: Results from Monte-Carlo simulation of last incremental decision problem ($b = 2$, $u = 1$).

the same decision as $E(MPC)$ (% same decision), and the difference between the estimated expected cost of the MINIMIN solution path, and the $E(MPC)$ solution path averaged over the set of trials. The estimate of the expected cost is based on the average minimum path cost over the 1000 samples collected in order to make the $E(MPC)$ decision. $\hat{E}(MPC)$ and $\hat{E}(MINIMIN)$ are the average over either 100,000 or 10,000 trials of the minimum path cost that is averaged over 1000 samples. Although it might seem unusual to present a result that is the difference between two numbers that are themselves averages of averages, most people who have heard about these experiments have inquired about this particular result.

The results for $T(b = 2, u = 1)$ are averaged over 100,000 trials, whereas all other results are averaged over 10,000 trials.

In general, these Monte-Carlo simulation results confirm the results from the previous section, namely that MINIMIN often makes the same decision as the $E(MPC)$ algorithm and has only a slightly higher average percent error as shown in Tables C.1, C.2, C.3, and C.4. In addition, the difference between the estimated expected cost of the MINIMIN decision and the $E(MPC)$ decision shown in Table C.5 is consistently a small positive value, indicating that $E(MPC)$ decisions are

Search	$E(MPC)$ decisions			MINIMIN decisions		
Depth	% optimal	% error	% wins	% optimal	% error	% wins
2	82.3 ± 0.7	3.89 ± 0.27	2.26 ± 0.29	82.3 ± 0.7	3.99 ± 0.27	2.17 ± 0.29
3	85.0 ± 0.7	2.11 ± 0.15	2.75 ± 0.32	84.4 ± 0.7	2.33 ± 0.16	2.05 ± 0.28
4	86.2 ± 0.7	1.42 ± 0.10	2.70 ± 0.32	85.6 ± 0.7	1.60 ± 0.11	2.03 ± 0.28
5	88.0 ± 0.6	1.00 ± 0.07	2.73 ± 0.32	87.3 ± 0.7	1.17 ± 0.08	1.86 ± 0.26

Table C.2: Results from Monte-Carlo simulation of last incremental decision problem ($b = 3$, $u = 1$).

Search	$E(MPC)$ decisions			MINIMIN decisions		
Depth	% optimal	% error	% wins	% optimal	% error	% wins
2	80.8 ± 0.8	4.49 ± 0.29	2.63 ± 0.31	81.1 ± 0.8	4.54 ± 0.29	2.91 ± 0.33
3	84.6 ± 0.7	2.30 ± 0.16	2.63 ± 0.31	84.6 ± 0.7	2.43 ± 0.17	2.52 ± 0.31
4	86.2 ± 0.7	1.43 ± 0.10	3.02 ± 0.34	85.6 ± 0.7	1.60 ± 0.11	2.32 ± 0.30
5	88.0 ± 0.6	0.99 ± 0.08	2.65 ± 0.31	87.6 ± 0.7	1.07 ± 0.08	2.24 ± 0.29

Table C.3: Results from Monte-Carlo simulation of last incremental decision problem ($b = 4$, $u = 1$).

Search	$E(MPC)$ decisions			MINIMIN decisions		
depth	% optimal	% error	% wins	% optimal	% error	% wins
2	80.7 ± 0.8	3.96 ± 0.24	2.12 ± 0.28	80.2 ± 0.8	4.11 ± 0.24	1.67 ± 0.25
3	84.6 ± 0.7	2.23 ± 0.15	2.73 ± 0.32	83.7 ± 0.7	2.44 ± 0.16	1.87 ± 0.27
4	86.3 ± 0.7	1.57 ± 0.11	2.74 ± 0.32	85.5 ± 0.7	1.77 ± 0.12	1.90 ± 0.27
5	87.1 ± 0.7	0.99 ± 0.08	2.65 ± 0.31	86.2 ± 0.7	1.07 ± 0.08	2.24 ± 0.29
10	89.4 ± 0.6	0.54 ± 0.04	2.65 ± 0.31	88.8 ± 0.6	0.62 ± 0.05	2.00 ± 0.27

Table C.4: Results from Monte-Carlo simulation of last incremental decision problem ($b = 2$, $u = 2$).

Search Tree			% Same	$\hat{E}(MINIMIN) - \hat{E}$(MPC)
b	s	u	Decision	(estimate of expected path cost difference)
2	2	1	96.7 ± 0.1	0.035 ± 0.017
2	3	1	96.4 ± 0.1	0.041 ± 0.016
2	4	1	96.5 ± 0.1	0.041 ± 0.015
2	5	1	96.5 ± 0.1	0.043 ± 0.018
2	6	1	96.5 ± 0.1	0.044 ± 0.013
2	7	1	96.6 ± 0.1	0.043 ± 0.013
2	8	1	96.7 ± 0.1	0.044 ± 0.012
2	9	1	96.8 ± 0.1	0.043 ± 0.012
2	10	1	96.8 ± 0.1	0.043 ± 0.012
3	2	1	95.6 ± 0.4	0.032 ± 0.013
3	3	1	95.2 ± 0.4	0.033 ± 0.011
3	4	1	95.3 ± 0.4	0.035 ± 0.010
3	5	1	95.4 ± 0.4	0.035 ± 0.010
4	2	1	94.5 ± 0.4	0.027 ± 0.011
4	3	1	94.9 ± 0.4	0.026 ± 0.009
4	4	1	94.7 ± 0.4	0.027 ± 0.008
4	5	1	95.1 ± 0.4	0.026 ± 0.008
2	2	2	96.2 ± 0.4	0.043 ± 0.022
2	3	2	95.4 ± 0.4	0.056 ± 0.020
2	4	2	95.4 ± 0.4	0.056 ± 0.019
2	5	2	95.2 ± 0.4	0.055 ± 0.018
2	10	2	95.4 ± 0.4	0.059 ± 0.015

Table C.5: Percent same decision and average estimated expected difference between MINIMIN and $E(MPC)$ from Monte-Carlo simulations of the last incremental decision problem.

only a slight improvement over MINIMIN decisions, relative to the cost of a greedy decision (*i.e.*, $1/(b+1)$), for the last incremental decision problem on this class of random trees.

APPENDIX D

Support Code for Sampling the Completion-Cost Distribution

This appendix contains the code used to sample the completion-cost distribution, and to calculate an estimate of the expected minimum path cost given two frontier-node costs.

```
/* sample.c - subroutines to support s alpha k-best decision making.
 */
int numSamples;                 /* the number of sample so far */
int numBuckets;                 /* the number of buckets */
int fcc[MAXBUCKETS];            /* the array for the sample buckets */
int minRequiredSamples;         /* minimum samples before sk-best decisions */

/* take_sample(nptr) is called with an newly expanded node pointer and
 * it updates the global data structures to reflect the minimum edge-cost
 * information. */
void take_sample(nptr)
  struct node *nptr;            /* the node that was just expanded */
{
  extern int numSamples;
  extern int fcc[MAXBUCKETS];
  extern int maxCost;           /* the maximum edge cost */
  extern int numBuckets;
  int minEdgeCost, i;
  struct node *nextptr;
  int j;

  if (numSamples == 0) {  /* if no samples yet, initialize the buckets */
    for (i = 0; i < numBuckets; ++ i)
      fcc[i] = 0;         /* initially no samples in any bucket */
  }
  if (numSamples < MAXINT) {   /* only sample if there is room in buckets */
    /* find the minimum edge cost below nptr */
    minEdgeCost = nptr->childptr->edgeCost;
    nextptr = nptr->childptr;
    for (i = 0; i < nptr->numChildren; ++ i) {
      if (nextptr->edgeCost < minEdgeCost)
        minEdgeCost = nextptr->edgeCost;
```

```
      nextptr = nextptr->sibptr;
    }
    /* increment the count in the bucket and the number of samples */
    fcc[(int)((double)minEdgeCost/(double)(maxCost-1)*numBuckets)] += 1.0;
    ++ numSamples;
  }
}


/* calc_empc takes two frontier-node costs, z1 <= z2, and returns the expected
 * minimum path cost for a last incremental decision choice between these two
 * frontier nodes.  The frontier node costs are integers in the range:
 *                   0 <= z1 <= z2 <= maxCost
 * The value returned is the expected minimum path cost, normalized so that
 * the edge costs are distributed from 0 to 1.
 */
double calc_empc(z1, z2)
  int z1, z2;                   /* two frontier node costs */
{
  extern int maxCost, numBuckets, numSamples;
  double cumF1, cumF2;          /* the cumulative distributions */
  int i, j;
  double expMinPathCost;        /* the expected minimum path cost */

  cumF1 = cumF2 = 0.0;
  expMinPathCost = 0.0;

  /* where update the code to handle more than two frontier nodes */
  /* j is the number of buckets corresponding to the difference between
   * the frontier node costs */
  j = round_int((double)(z2-z1)/(double) (maxCost-1)*numBuckets);
  /* for each bucket */
  for (i = 0; i < numBuckets; ++ i) {
    /* calculate the cumulative distribution value for z1 so far */
    cumF1 += (double)fcc[i] / (double)numSamples;
    if (i >= j) {       /* is it time to consider z2 as well? */
      /* calculate the cumulative distribution value for z2 so far */
      cumF2 += (double)fcc[i-j] / (double)numSamples;
      /* calculate the expected minimum path cost based on two frontier-node
       * costs (z1 and z2) */
      expMinPathCost += ((double)(i+0.5)/(double)numBuckets)
        * ((double) fcc[i] / (double)numSamples
           +(double)fcc[i-j] / (double)numSamples
           - ((double)(cumF1 * fcc[i-j]) / (double)numSamples)
           - ((double)(cumF2 * fcc[i]) / (double)numSamples));
    }
    else {
```

```
      /* otherwise, calculate the expected minimum path cost based on a
       * single frontier-node cost */
      expMinPathCost += ((double)(i+0.5)/(double)numBuckets)
        * (double) fcc[i] / (double)numSamples;
    }
  }
  /* so far the expMinPathCost is only for the path from the frontier
   * node to a leaf node, so add z1 to the completion cost before returning */
  return(expMinPathCost + (double)z1/(double)(maxCost-1));
}
```

APPENDIX E

Estimating the Cost of a Flowshop Schedule

The first half of this appendix contains a detailed description of the Ignall and Schrage's lower-bound heuristic evaluation function for the two-machine flowshop scheduling problem when using the sum-finishing-time cost function [IS65]. The second half contains the code we used to implement the InS heuristic evaulation function.

E.1 Estimating the Complete Schedule Cost

Ignall and Schrage [IS65] have proposed a simple but accurate heuristic function (InS heuristic) for estimating the cost of completing a partial schedule when using the sum-finishing-time cost function. The basic idea is to calculate two simple estimates of the complete-schedule cost, one by ignoring the one-job-at-a-time constraint for jobs on the first machine, and the other by ignoring the one-job-at-a-time constraint for jobs on the second machine. The maximum of these two completion-cost estimates is a very good lower bound on the actual completion cost of a partial schedule. In addition, this heuristic function only requires computation that is linear in the number of remaining jobs, plus the initial cost of sorting the list of jobs twice, once by the task times on the first machine and once by the task times on the second machine. This is because we can calculate the two lower bound estimates by simply sorting the jobs based on their processing time on the machine that still has the one-job-at-a-time constraint. Our presentation of Ignall and Schrage's heuristic follows Papadimitriou and Steiglitz's treatment in [PS82].

Again we consider the scheduling problem in Figure 9.1, except that now we assume that these are the last four jobs to be scheduled. The other jobs (not shown) have already been scheduled. What we want to know is an estimate of the complete-schedule cost given the cost of the partial schedule, the finish times on machines M_1 and M_2 for the last job in the partial schedule, and the processing times that the four remaining jobs require on each machine. The processing times for the remaining jobs are summarized in Table E.1.

Machine	j_1	j_2	j_3	j_4
M_1	8	6	2	6
M_2	4	10	7	2

Table E.1: Processing times for four unscheduled jobs.

Let s be the number of jobs already scheduled, and let r be the number of jobs remaining to be scheduled (*i.e.*, $r = 4$). In addition, let FT_{ji} denote the time at which job i finishes on machine j. With this definition, the cost of a schedule is just the sum of FT_{2i} over all jobs i. If J is the set of jobs that have already been scheduled, then we can express the complete-schedule cost as follows.

$$cost(schedule) = \sum_{j \epsilon J} FT_{2j} + \sum_{j \notin J} FT_{2j} \tag{E.1}$$

The first term in Equation E.1 is just the cost of the partial schedule so far. What we want is an easy way to calculate a lower bound on the second term in Equation E.1, which we call the completion cost and denote CC.

The first lower bound on the completion cost is calculated by assuming that every job can start processing on machine M_2 as soon as it is finished being processed by machine M_1. With this assumption, the minimum-cost schedule occurs when the remaining jobs are scheduled in increasing order of their processing time on machine M_1. Let k be the index of the k^{th} job in a list of the unscheduled jobs that is sorted in increasing order of their processing time on machine M_1, and let S be the index of the last job in the partial schedule. Finally, let τ_{ikj} be the processing time on machine i of the job that has the k^{th} smallest processing time on machine j. In this case, the completion-cost estimate (CC) can be expressed as follows.

$$\begin{aligned} CC_1 &= \sum_{k=1}^{r} [FT_{1S} + (s+k)\tau_{1k1} + \tau_{2k1}] \\ &= rFT_{1S} + \sum_{k=1}^{r} [(s+k)\tau_{1k1} + \tau_{2k1}] \end{aligned}$$

The first term (rFT_{1S}) is the cost of completing the last job that is already scheduled that must be added to all of the remaining jobs. The second term is the sum over all remaining jobs of the cost that a job will incur on the first machine, plus its time to finish processing on the second machine. The $(s+k)$ factor takes into account the fact that the remaining jobs will all be delayed by the

time it takes to process the current job. This sum is performed in increasing order of the task times on machine M_1 because we are ignoring the interaction between jobs on machine M_2. If we include the constraint that jobs can only be processed one at a time on machine M_2, then CC_1 can only increase ($\sum_{j \notin J} FT_{2j} \geq CC_1$), thus CC_1 is a lower bound on the completion cost.

For the sample processing times on machine M_1 in Table E.1, we can calculate a lower bound on the completion cost as follows.

$$\begin{aligned} CC_1 &= 4FT_{1S} + 4(2) + 7 + 3(6) + 2 + 2(6) + 10 + 1(8) + 4 \\ &= 4FT_{1S} + 69 \end{aligned}$$

The second lower bound on the completion cost is calculated by assuming that the next job is always ready to be processed by machine M_2, or equivalently that machine M_1's constraint on the remainder of the schedule is ignored. For this assumption, the minimum-cost schedule occurs when the remaining jobs are scheduled in increasing order of their processing time on machine M_2. We now let k be the index of the k^{th} job in a list of the unscheduled jobs that is sorted in increasing order of their processing time on machine M_2. In addition, let τ_{1j} be the processing time of job j on machine M_1. In this case, the second completion-cost estimate can be expressed as follows.

$$\begin{aligned} CC_2 &= \sum_{k=1}^{r} [\max(FT_{2S}, FT_{1S} + \min_{j \notin J} \tau_{1j}) + (s+k)\tau_{2k2}] \\ &= r \max(FT_{2S}, FT_{1S} + \min_{j \notin J} \tau_{1j}) + \sum_{k=1}^{r} (s+k)\tau_{2k2} \end{aligned}$$

The first term ($\max(FT_{2S}, FT_{1S} + \min_{j \notin J} \tau_{1j})$) is the earliest that the remaining jobs could start processing on machine M_2, and it is counted r times because this time is added to the finish time of all remaining jobs. The second term is the sum of the finish times on the second machine. Since we have ignored the constraint on the first machine, this sum is minimized when the remaining jobs are processed in increasing order of their times on machine M_2. CC_2 is also a lower bound on the actual completion cost of the schedule because any conflict between jobs on machine M_1 that would keep the jobs from being immediately available for processing on machine M_2 could only increase the cost of the schedule.

Using the processing times on machine M_2 in Table E.1, we can solve for a second lower bound on the completion cost as follows.

$$\begin{aligned} CC_2 &= 4 \max(FT_{2S}, FT_{1S} + min_{j \notin J} \tau_{1j}) + 4(2) + 3(6) + 2(6) + 8 \\ &= 4 \max(FT_{2S}, FT_{1S} + 2) + 44 \end{aligned}$$

The final lower bound on the completion cost is simply calculated as the maximum of the two lower bounds.

$$cost(schedule) \geq cost(partial\ schedule) + \max(CC_1, CC_2) \tag{E.2}$$

If we initially sort the jobs by the M_1 and M_2 processing times, then the worst-case complexity of calculating the lower bound on the completion cost for a partial schedule is linear in the number of jobs remaining to be scheduled. In our experience with randomly-generated flowshop scheduling problems, the InS heuristic function provides a very accurate heuristic estimate of the minimum complete-schedule cost for a partial schedule.

E.2 Code for Ignall and Schrage's Flowshop Scheduling Heuristic Evaluation Function

This section contains the code that we used to implements Ignall and Schrage's heuristic evaluation function. Note that the machines are named $M0$ and $M1$ in the code instead of M_1 and M_2 as in the text.

```
#def COSTTYPE double

/* h_value takes a pointer to a node in the flowshop scheduling problem
 * space, and returns the InS heuristic estimate of the cost of scheduling
 * the remaining jobs given the partial schedule associated with the node.
 */
COSTTYPE h_value(nodeptr)
  struct node *nodeptr;
{
  extern int numJobs;           /* the no. of jobs in a complete schedule */
  int i, j, k, mzeromin;
  COSTTYPE Fsub0;               /* FT_{1S} in previous section */
  COSTTYPE Fsub1;               /* FT_{2S} in previous section */
  COSTTYPE suma;                /* sum in equation for CC_{1} */
  COSTTYPE sumb;                /* sum in equation for CC_{2} */
  int used[MAXJOBS];            /* 0 means used, 1 means not used */
                                /* used keeps track of the scheduled jobs*/
  struct node *nextptr;
  int MSort[NUMMACHINES][MAXJOBS]; /* a list of job lists sorted by their
                                    * processing time on each machine */

  if (nodeptr->numChildren == 0) /* no jobs left to schedule */
    return((COSTTYPE) 0);
  /* first compute the sum time estimate by removing the 1 job at a time
```

```
 * constraint on machine 2. */
/* setup the array of used = 0, not used = 1 jobs */
for (i = 0; i < numJobs; ++ i) {
  used[i] = 1;                /* 0 means used; 1 means not used */
}
/* walk up the parent pointers to initialize the schedule/used job array */
nextptr = nodeptr->parentptr;
while (nextptr != currentNodeptr) {
  used[nextptr->name] = 0;
  nextptr = nextptr->parentptr;
}

Fsub0 = nodeptr->MfinishTime[0];  /* F_{1t_{r}} in Papa&Steiglitz p.444 */
suma = (COSTTYPE) 0;
j = 0;
for (i = 0; i < nodeptr->numChildren; ++ i) {
  /* i is the number of jobs that have not been scheduled yet */
  /* find the next unused job in increasing order of task time on M0 */
  while ((used[MSort[0][j]] == 0) || (MSort[0][j] == nodeptr->name))
    ++ j;
  k = MSort[0][j];            /* k is the number of the next unused job */
  if (i == 0)
    mzeromin = k;             /* update the minimum task time on M0 */
  ++j;

  /* add to suma the cost of scheduling job k */
  suma = suma + Fsub0
    + (COSTTYPE) (nodeptr->numChildren - i) * Mtime[0][k]
      + (COSTTYPE) Mtime[1][k];
}
/* next compute the sum time estimate by removing the 1 job
 * constraint on machine 1. */
/* first calculate the impact of the partial schedule on the cost of
 * scheduling the remaining decisions */
Fsub1 = max(nodeptr->MfinishTime[1],
            nodeptr->MfinishTime[0] + Mtime[0][mzeromin]);
sumb = (COSTTYPE) 0;
j = 0;
for (i = 0; i < nodeptr->numChildren; ++ i) {
  /* find the next unused job in order of the task time on M1 */
  while ((used[MSort[1][j]] == 0) || (MSort[1][j] == nodeptr->name))
    ++ j;
  k = MSort[1][j];            /* k is the number of the next unused job */
  ++j;
  /* add to sumb the cost of scheduling job k */
  sumb = sumb + Fsub1
    + (COSTTYPE)(nodeptr->numChildren - i)
```

```
        * Mtime[1][k];
  }
  return(max(suma, sumb));
}
```

References

[ACH90] T. Anantharaman, M.S. Campbell, and F.H. Hsu. "Singular extensions: Adding selectivity to brute-force searching." *Artificial Intelligence*, **43**(1):99–109, 1990.

[Bau93] Eric B. Baum. "How a Bayesian approaches games like chess." In *Proceedings of the AAAI Fall Symposium on Games: Planning and Learning*, pp. 48–50, Raleigh, NC, October 1993.

[BBS93] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. "Learning to Act Using Real-Time Dynamic Programming." Technical Report UM-CS-1993-002, University of Massachusetts, Amherst, 1993.

[URL: ftp://ftp.cs.umass.edu/pub/techrept/techreport/1993/UM-CS-1993-002.ps.]

[BCG76] J. L. Bruno, E. G. Coffman, Jr., R. L. Graham, W. H. Kohler, R. Sethi, K. Steiglitz, and J. D. Ullman. "Computer and Job-Shop Scheduling Theory." John Wiley and Sons, 1976.

[BD94] Mark Boddy and Thomas L. Dean. "Deliberation scheduling for problem solving in time-constrained environments." *Artificial Intelligence*, **67**:245–285, 1994.

[Ber79] H. J. Berliner. "The B* tree search algorithm: A best-first proof procedure." *Artificial Intelligence*, **12**:23–40, 1979.

[Bod91] Mark Boddy. *Solving Time-Dependent Problems: A Decision-Theoretic Approach to Planning in Dynamic Environments.* PhD thesis, Brown University, May 1991.

[CGA89] P. P. Chakrabarti, S. Ghose, A. Acharya, and S. C. de Sarkar. "Heuristic Search in Restricted Memory." *Artificial Intelligence*, **41**:197–221, 1989.

[CMM67] Richard W. Conway, William L. Maxwell, and Louis W. Miller. *Theory of Scheduling.* Addison-Wesley, 1967.

[Cra94] James Crawford. "Personal Communication.", September 1994.

[DB88] Thomas Dean and Mark Boddy. "An Analysis of Time-Dependent Planning." In *Proceedings, 7th National Conference on Artificial Intelligence (AAAI-88), St. Paul, MN*, pp. 49–54, Palo Alto, CA, 1988.

[Dea86] Thomas L. Dean. "Intractability and Time-Dependent Planning." In Michael P. Georgeff and Amy L. Lansky, editors, *Proceedings of the 1986 Workshop on Reasoning About Actions and Plans*, pp. 245–266. Morgan Kaufmann, June 1986.

[Dea91a] Thomas Dean. "Decision-Theoretic Control of Inference for Time-Critical Applications." *International Journal of Intelligent Systems*, **6**:417–441, 1991.

[Dea91b] Thomas Dean. "Planning Under Uncertainty and Time Pressure." In *Proceedings of the 1990 Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 390–395, 1991.

[Dea94] Thomas Dean. "Decision-Theoretic Planning and Markov Decision Processes." Unpublished manuscript, 1994.

[DKK93a] Thomas Dean, Leslie Kaelbling, Jak Kirman, and Ann Nicholson. "Deliberation Scheduling for Time-Critical Sequential Decision Making." In David Heckerman and Abe Mamdani, editors, *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence*, pp. 309–316, Washington, D. C., July 1993.

[DKK93b] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. "Planning Under Time Constraints in Stochastic Domains." Technical Report CS-93-55, Brown University Computer Science Department, 1993.

[URL: http://www.cs.brown.edu/publications/techreports/reports/CS-93-55.]

[DS90] Thomas Dean and Greg Siegle. "An Approach to Reasoning About Continuous Change for Applications in Planning." In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI90)*, pp. 132–137. American Association for Artificial Intelligence, 1990.

[DSB93] Mark Drummond, Keith Swanson, John Bresina, and Rich Levinson. "Reaction-First Search." In *Proceedings of the* 13^{th} *International Joint Conference on Artificial Intelligence (IJCAI-93)*, 1993.

[DW91] Thomas L. Dean and Michael P. Wellman. *Planning and Control.* Morgan Kaufmann, San Mateo, California, 1991.

[Elk90] Charles Elkan. "Incremental, Approximate Planning." In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI90)*, pp. 145–150. American Association for Artificial Intelligence, 1990.

[Etz89] Oren Etzioni. "Tractable Decision-Analytic Control." In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pp. 114–125, May 1989.

[FGG73] S. H. Fuller, J. G. Gaschnig, and J. J. Gillogly. "An analysis of the alpha-beta pruning algorithm." Technical report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1973.

[Gin90] Matthew L. Ginsberg. "Computational Considerations in Reasoning about Action." In Katia Sycara, editor, *Proceedings of the 1990 Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 37–47, 1990.

[GJS76] M. R. Garey, D. S. Johnson, and R. Sethi. "The complexity of flowshop and jobshop scheduling." *Math. Oper. Res.*, **1**:117–129, 1976.

[Goo94] Richard Goodwin. "Reasoning About When to Start Acting." In K. Hammond, editor, *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pp. 86–91, 1994.

[Gre94] Lloyd Greenwald. "Personal Communication.", July 1994.

[GS79] Michael R. Garey and David S.Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[HA90] James Hendler and Ashok Agrawala. "Mission Critical Planning: AI on the MARUTI Real-time Operating System." In Katia Sycara, editor, *Proceedings of the 1990 Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 231–239, 1990.

[Har90] Leo Hartman. "Decision Theory and the Cost of Planning." Technical Report TR-CS-355, University of Rochester, 1990.

[HF90] Steve Hanks and R. James Firby. "Issues and Architectures for Planning and Execution." In Katia Sycara, editor, *Proceedings of the 1990 Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 231–239, 1990.

[HHR90] Adele E. Howe, David M. Hart, and Paul R.Cohen. "Addressing Real-Time Constraints in the Design of Autonomous Agents." *The Journal of Real-Time Systems*, **1**:81–97, 1990.

[HM90] Othar Hansson and Andrew Mayer. "Decision-Theoretic Control of Search in BPS." In *Working Notes of the 1989 AAAI Spring Symposium on AI and Limited Rationality*, pp. 59–63, 1990.

[HMR90] Othar Hansson, Andrew Mayer, and Stuart Russell. "Decision-Theoretic Planning in BPS." In *Working Notes of the AAAI Spring Symposium on Planning in Uncertain, Unpredictable or Changing Environments*, pp. 44–48. American Association for Artificial Intelligence, 1990.

[Hor87] Eric J. Horvitz. "Reasoning about Beliefs and Actions under Computational Resource Constraints." In *Proceedings, 3^{rd} Workshop on Uncertainty in AI, Seattle, WA*, pp. 301–324, July 1987.

[Hor90] Eric J. Horvitz. *Computation and Action Under Bounded Resources.* PhD thesis, Stanford University, December 1990.

[Hor93] Eric J. Horvitz. "Personal Communication.", August 1993.

[Iba76] T. Ibaraki. "Computational efficiency of approximate branch-and-bound algorithms." *Mathematics of Operations Research*, **1**:287–298, 1976.

[IS65] Edward Ignall and Linus Schrage. "Application of the Branch and Bound Technique to Some Flow-Shop Scheduling Problems." *Operations Research*, **13**:400–412, 1965.

[Joh54] S. M. Johnson. "Optimal Two– and Three-Stage Production Schedules with Setup Times Included." *Nav. Res. Log. Quart.*, **1**(1), March 1954. also in *Industrial Scheduling*, J. F. Muth and G. L. Thompson, eds., Prentice-Hall, 1963, Chapter 2.

[Joh90] David S. Johnson. "Local Optimization and the Traveling Salesman Problem." In M. S. Paterson, editor, *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*, pp. 446–461. Springer-Verlag, 1990.

[KC94] R. E. Korf and D. M. Chickering. "Best-First Minimax: Othello Results." In *Proceedings of the 12^{th} National Conference on Artificial Intelligence (AAAI-94)*, pp. 1365–1370, Seattle, WA, August 1994.

[KNP90] Sarit Kraus, Madhura Nirkhe, and Donald Perlis. "Deadline-Coupled Real-Time Planning." In Katia Sycara, editor, *Proceedings of the 1990 Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 100–108, 1990.

[Koe92] Sven Koenig. "The Complexity of Real-time Search." Technical Report CMU-CS-92-145, Carnegie Mellon University, 1992.

[Kor85] Richard E. Korf. "Depth-first iterative-deepening: an optimal admissible tree search." *Artificial Intelligence*, **27**:97–109, 1985.

[Kor90] Richard E. Korf. "Real-time heuristic search." *Artificial Intelligence*, **42**(2–3):189–211, March 1990.

[Kor93] Richard E. Korf. "Linear-space best-first search." *Artificial Intelligence*, **62**(1):41–78, July 1993.

[KP83] R.M. Karp and J. Pearl. "Searching for an optimal path in a tree with random costs." *Artificial Intelligence*, **21**:99–117, 1983.

[KPZ94] Richard E. Korf, Joseph C. Pemberton, and Weixiong Zhang. "Incremental Random Search Trees." In James M. Crawford and Bart Selman, editors, *Working Notes of the AAAI-94 Workshop on Experimental Evaluation of Reasoning and Search Methods*. American Association for Artificial Intelligence, August 1994.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[KS76] W. H. Kohler and K. Steiglitz. *Computer and Job-Shop Scheduling Theory*, chapter 6: Enumerative and Iterative Computational Approaches, pp. 229–288. John Wiley and Sons, 1976.

[LPD88] Victor R. Lesser, Jasmina Pavlin, and Edmund Durfee. "Approximate Processing in Real-Time Problem Solving." *AI Magazine*, pp. 49–61, 1988.

[Man79] Edwin Mansfield. *Micro-Economics: Theory and Applications*. W.W. Norton and Company, third edition, 1979.

[McA88] David Allen McAllester. "Conspiracy Numbers for Min-Max Search." *Artificial Intelligence*, **35**:287–310, 1988.

[McD90a] Drew McDermott. "Planning Reactive Behavior: A Progress Report." In Katia Sycara, editor, *Proceedings of the 1990 Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 231–239, 1990.

[McD90b] C. J. H. McDiarmid. "Probabilistic analysis of tree search." In G. R. Gummett and D. J. A. Welsh, editors, *Disorder in Physical Systems*, pp. 249–260. Oxford Science Publishing, 1990.

[MG90] A. Mahanti and S. Ghosh. "Best First Search with Controlled Time and Storage." Technical Report UMIACS-TR-90-108, Institute for Advanced Computer Studies, University of Maryland, 1990.

[MGP92] A. Mahanti, S. Ghosh, and A. K. Pal. "A High-Performance Limited-Memory Admissible and Real Time Search Algorithm for Networks." Technical Report UM-CS-TR-2858, University of Maryland, 1992.

[MP91] C. J. H. McDiarmid and G. M. A. Provan. "An expected-cost analysis of backtracking and non-backtracking algorithms." In *Proceedings, 12^{th} International Joint Conference on Artificial Intelligence, (IJCAI-91)*, pp. 172–177, Sydney, Australia, August 1991.

[MS90] Nicola Muscettola and Stephen F. Smith. "Integrating Planning and Scheduling to Solve Space Mission Scheduling Problems." In Katia Sycara, editor, *Proceedings of the 1990 Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 220–230, 1990.

[Mut86] David Mutchler. "Optimal Allocation of Very Limited Search Resources." In *Proceedings, 5^{th} National Conference on Artificial Intelligence (AAAI-86), Philadelphia, PA*, pp. 467–471, Palo Alto, CA, 1986.

[Mut92] David Mutchler. "Heuristic search with limited resources, Part I." Unpublished manuscript, November 1992.

[Nau83] D. S. Nau. "Pathology on Game Trees Revisited, and an Alternative to Minimaxing." *Artificial Intelligence*, **21**(1, 2):221–244, March 1983.

[Nil69] N. J. Nilsson. "Searching problem-solving and game-playing trees for minimal cost solutions." In A. J. H. Morrell, editor, *Information Processing 68, Proceedings of the IFIP Congress 1968*, pp. 1556–1562, Amsterdam, 1969. North-Holland.

[Pea83] Judea Pearl. "Knowledge versus Search: A Quantitative Analysis Using A*." *Artificial Intelligence*, **20**:1–13, 1983.

[Pea84] Judea Pearl. *Heuristics*. Addison-Wesley, Reading, MA, 1984.

[Pea88] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.

[Per90] E. L. Perry. "Solution of Time Constrained Scheduling Problems with Parallel Tabu Search." In Katia Sycara, editor, *Proceedings of the 1990 Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 231–239, 1990.

[PK93] Joseph C. Pemberton and Richard E. Korf. "An Incremental Search Approach to Real-Time Planning and Scheduling." In *Foundations of Automatic Planning: The Classical Approach and Beyond*, pp. 107–111. AAAI Press Technical Report #SS-93-03, 1993.

[PK94a] Joseph C. Pemberton and Richard E. Korf. "Incremental Search Algorithms for Real-Time Decision Making." In *Proceedings, Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, pp. 140–145, 1994.

[PK94b] Joseph C. Pemberton and Richard E. Korf. "An Incremental Search Approach to Real-Time Decision Making." In *Proceedings, AAAI Spring Symposium on Planning, Stanford, CA*, 1994.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[Riv87] R. L. Rivest. "Game tree searching by min/max approximation." *Artificial Intelligence*, **34**(1):77–96, 1987.

[Rus92] Stuart Russell. "Efficient memory-bounded search methods." In *Proceedings, European Conference on Artificial Intelligence (ECAI-92)*, 1992.

[RW89] S. Russell and E. Wefald. "On optimal game-tree search using rational meta-reasoning." In *Proceedings of the 11^{th} International Joint Conference on Artificial Intelligence (IJCAI-89)*, pp. 334–340, Detroit, MI, 1989.

[RW91] Stuart Russell and Eric Wefald. *Do the Right Thing*. MIT Press, Cambridge, MA, 1991.

[SB89] A.K. Sen and A. Bagchi. "Fast recursive formulations for best-first search that allow controlled use of memory." In *Proceedings*, 11^{th} *International Joint Conference on Artificial Intelligence, (IJCAI-89), Detroit, MI*, pp. 297–302, August 1989.

[Sha50] C. E. Shannon. "Programming a computer for playing chess." *Philosophical Magazine*, **41**(7):256–275, 1950.

[Smi93] David E. Smith. "Personal Communication.", August 1993.

[SOM90] Stephen F. Smith, Peng Si Ow, Nicola Muscettola, Jean-Yves Potvin, and Dirk C. Matthys. "OPIS: An Integrated Framework for Generating and Revising Factory Schedules." In Katia Sycara, editor, *Proceedings of the 1990 Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 231–239, 1990.

[SP94] Jay K. Strosnider and C. J. Paul. "A Structured View of Real-Time Problem Solving." *AI Magazine*, **15**(2):45–66, 1994.

[SRS90] Katia Sycara, Steve Roth, Norman Sadeh, and Mark Fox. "Managing Resource Allocation in Multi-Agent Time-Constrained Domains." In Katia Sycara, editor, *Proceedings of the 1990 Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 240–250, 1990.

[Sut90a] Richard S. Sutton. "First Results with DYNA, an Integrated Architecture for Learning, Planning and Reacting." In *Working Note of the 1990 AAAI Spring Symposium on Planning in Uncertain, Unpredictable or Changing Environments*, pp. 136–140, 1990.

[Sut90b] Richard S. Sutton. "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming." In *Proceedings of the Seventh International Workshop on Machine Learning (ML90)*, pp. 216–224, 1990.

[Sut91] Richard S. Sutton. "Planning by Incremental Dynamic Programming." In Lawrence A. Birnbaum and Gregg C. Collins, editors, *Proceedings of the Eighth International Workshop on Machine Learning (ML91)*, pp. 353–357. Morgan Kaufmann, 1991.

[TJ92] Prasad Tadepalli and Varad Joshi. "Real-time Scheduling Using Minimin Search." In *Working Notes of the 1992 AAAI Spring Symposium on Planning*, pp. 77–81. American Association for Artificial Intelligence, 1992.

[ZDG90] Monte Zweben, Michael Deale, and Robert Gargan. "Anytime Rescheduling." In Katia Sycara, editor, *Proceedings of the 1990 Workshop on Innovative Approaches to Planning, Scheduling and Control*, pp. 231–239, 1990.

[ZK95] Weixiong Zhang and Richard E. Korf. "Performance of linear-space search algorithms." *to appear in Artificial Intelligence*, 1995.

[ZP94a] Weixiong Zhang and Joseph C. Pemberton. "Epsilon-transformation: Exploiting phase transitions to solve combinatorial optimization problems - initial results." In *Proceedings of the* 12^{th} *National Conference on Artificial Intelligence (AAAI-94)*. American Association for Artificial Intelligence, July 1994.

[ZP94b] Weixiong Zhang and Joseph C. Pemberton. "Epsilon-transformation: Exploiting phase transitions to solve combinatorial optimization problems." Technical Report UCLA-CSD-940003, Computer Science Department, University of California, Los Angeles, January 1994. *to appear in Artificial Intelligence.*