MULTICRITERIA PATHFINDING IN UNCERTAIN SIMULATED ENVIRONMENTS

A Dissertation

Presented to

The Faculty of the Graduate School

At the University of Missouri

In Partial Fulfillment

Of the Requirements for the Degree

Doctor of Philosophy

By

ANDREW R. BUCK

Dr. James Keller, Dissertation Supervisor

MAY 2018

The undersigned, appointed by the dean of the Graduate School,
have examined the Dissertation entitled

MULTICRITERIA PATHFINDING IN UNCERTAIN
SIMULATED ENVIRONMENTS

presented by Andrew R. Buck,

a candidate for the degree of Doctor of Philosophy, and hereby certify that, in their opinion, it is worthy of acceptance.

Professor James Keller

Professor Alina Zare

Professor Marjorie Skubic

Professor Mihail Popescu

Dedicated to my parents, family, friends, teachers, mentors,
and everyone who believed I could.

Thank you.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. James Keller for providing the opportunity to pursue what has become one of the most challenging, enlightening, and rewarding endeavors I have ever undertaken. Your support, guidance, and encouragement over the years has pushed me to accomplish more than I ever thought possible. You have helped me navigate through the uncertainty of academic research, and I am forever grateful to have learned from your wisdom.

Thanks also to Dr. Popescu, Dr. Skubic, and Dr. Zare for their help during my graduate career. Our conversations have been insightful and full of valuable advice.

I would also like to thank the National Geospatial-Intelligence Agency and the Army Research Office with the RDECOM CERDEC Night Vision Electronic Sensors Directorate for support during my dissertation research.

Finally, thank you to all the students, friends, and teachers who have shared their thoughts and ideas along the way. I'm glad to have spent time together here at Mizzou.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

LIST OF FIGURES viii

LIST OF TABLES xiv

LIST OF ALGORITHMS xv

ABSTRACT xviii

1 INTRODUCTION 1

1.1 Problem Statement 1

1.2 The Computational Mental Map Framework 5

1.3 Contributions and Potential Applications 9

2 BACKGROUND 12

2.1 Wayfinding and Cognitive Mapping 12

2.2 Procedural Content Generation 15

2.2.1 Cellular Automata 16

2.2.2 Fashion-based Cellular Automata 18

2.2.3 Fractal Terrain 19

2.3 Viewshed Analysis 22

2.4 Least-Cost Paths in Fuzzy Weighted Graphs 28

2.4.1 Least-Cost Path Problems 30

2.4.2 Fuzzy Numbers 32

2.4.3 The Multiobjective Fuzzy Least-Cost Path Problem 36

2.5 Multiobjective Optimization 38

2.5.1 Multiobjective Optimization Problem Definition 39

2.5.2 Pareto Optimality 40
2.5.3 Scalarization 42
2.5.4 Method of the Global Criterion 44
2.5.5 Method of Weighted Metrics 45
2.5.6 Ordered Weighted Average Approach 47
2.6 Multiobjective Evolutionary Algorithms 49
2.7 Intelligent Agents 54
2.8 The Traveling Salesman Problem 55
3 CREATING GRID WORLD ENVIRONMENTS 57
3.1 Grid Worlds 57
3.2 Generating Caverns 60
3.3 Region Partitioning 65
3.4 Creating a Heightmap 76
3.5 Defining Terrain Types 81
3.5.1 Binary Terrain Environments 81
3.5.2 Trinary Terrain Environments 83
3.5.3 Full World Environments 86
3.6 Resource Placement 89
3.7 Summary 93
4 THE MENTAL MAP GRID 95
4.1 The Mental Map 95
4.1.1 Creating Observations 98
4.1.2 Viewshed Computation 100
4.1.3 Finalizing the Observation 104

4.1.4 Updating the Mental Map 105

4.2 The Action Graph 109

4.3 Crisp Feature Functions 111

4.3.1 Distance Feature 112

4.3.2 Terrain Type Features 112

4.3.3 Terrain Transition Features 113

4.3.4 Elevation Features 114

4.3.5 Other Features 116

4.3.6 Example 116

4.4 Fuzzy Feature Functions 118

4.4.1 Distance Feature 119

4.4.2 Terrain Type Features 119

4.4.3 Terrain Transition Features 124

4.4.4 Elevation Features 129

4.5 Summary 138

5 THE REGION GRAPH 140

5.1 The Region Graph 140

5.1.1 Defining the Local Region 142

5.1.2 Creating the Region Boundaries 144

5.1.3 Constructing the Region Graph 147

5.2 Fuzzy Region Distance 151

5.2.1 Computing the Distance Cost Matrix 152

5.2.2 Region Distance Feature 159

5.2.3 Region Terrain Type Features 160

5.2.4 Region Terrain Transition Features 164

5.3 General Fuzzy Region Features 169

5.3.1 General Framework for Computing Region Features 169

5.3.2 Region Elevation Features 177

5.3.3 Unobserved Elevation Costs 183

5.3.4 Combining Region Elevation Costs 193

5.4 Approximate Fuzzy Region Features 200

5.5 Updating the Region Graph 208

5.6 Summary 219

6 LEAST-COST PATH PROBLEMS 221

6.1 Shortest Paths in Grid Worlds 221

6.2 The Multiobjective Fuzzy Least-Cost Path Problem 228

6.2.1 Multiobjective Optimization for the MO-FLCPP 229

6.2.2 Scalarization 230

6.2.3 Example 233

6.3 Decomposition of the MO-FLCPP 238

6.3.1 Edge Normalization 239

6.3.2 Exponential Scaling 240

6.3.3 Pre-scalarized Decomposition 243

6.4 MOEA/D for the MO-FLCPP 247

6.5 Experiments 252

6.5.1 Two Objective Shortest Paths in Binary Terrain Environments ... 253

6.5.2 Two Objective Least-Cost Paths Using Elevation 258

6.5.3 Shortest Paths Using Terrain Transition Features 262

6.5.4 Many-Objective Least-Cost Paths 265

6.5.5 Comparing MOEA/D to Pre-scalarized Decomposition 267

6.6 A Greedy Algorithm for the CMM Framework 274

6.7 Summary 285

7 CONCLUSION 287

7.1 Summary of the CMM Framework 287

7.2 Future Work 289

REFERENCES 294

VITA 302

LIST OF FIGURES

Figure 1.1 Example environment with three different path options to reach a goal location ... 2

Figure 1.2 Block diagram of the server/client architecture used in the CMM framework ... 6

Figure 1.3 Examples of grid world environments from the CMM framework ... 7

Figure 1.4 An agent's mental map for an example scenario ... 8

Figure 2.1 Generative model of cognitive mapping ... 14

Figure 2.2 Visualization of the diamond-square algorithm on a 5×5 grid ... 20

Figure 2.3 A progression of the diamond-square algorithm generating a fractal terrain ... 21

Figure 2.4 An example of the successive random additions method for generating fractal terrain ... 22

Figure 2.5 Example of computing the viewshed of a grid cell ... 25

Figure 2.6 Summation and maximization of two triangular fuzzy numbers ... 34

Figure 2.7 Mapping from decision space to objective space in a multiobjective optimization problem ... 40

Figure 2.8 The mapping of solution vectors from decision space to objective space shows which solutions belong to the Pareto optimal set in decision space and the Pareto front in objective space ... 41

Figure 2.9 Examples of the range of the Pareto front ... 43

Figure 2.10 Different metrics applied in the global criterion method ... 45

Figure 2.11 Comparison of the weighted sum and Tchebycheff scalarization approaches ... 47

Figure 3.1 Examples of grid world problem domains generated in the CMM framework ... 59

Figure 3.2 Examples of cavern maps generated using Algorithm 3.1 ... 65

Figure 3.3 Tabu sampling on a 50×50 grid with different values for the separation radius 68

Figure 3.4 Results of the region partitioning algorithm on a 50×50 grid with different values for the separation radius 75

Figure 3.5 Random noise images at different scales on a 50×50 grid with no cave walls 78

Figure 3.6 Random noise images at different scales on a 50×50 grid with a provided cave wall map 78

Figure 3.7 Heightmaps generated on a 50×50 grid with different values of p and q using the same random seed 80

Figure 3.8 Examples of binary environments containing forest and meadow terrain types 83

Figure 3.9 Examples of the fashion-based cellular automata algorithm for creating trinary terrain environments 86

Figure 3.10 Examples of full world environments generated using Algorithm 3.15 89

Figure 3.11 Examples of shortest path problems in a cavern environment using the tabu sampling approach and the longest path approach 91

Figure 3.12 Examples of traveling salesman problems initialized using the tabu sampling method in meadow terrain only and using extrema locations in the elevation 92

Figure 3.13 Examples of traveling purchaser problems in full world environments 93

Figure 4.1 Examples of observations in various environments computed using Algorithm 4.2 and Algorithm 4.3 100

Figure 4.2 Updating the mental map from an observation 106

Figure 4.3 Filling in unreachable areas with walls 108

Figure 4.4 Fixing diagonal boundaries 109

Figure 4.5 Examples of the action graph for two mental maps 110

Figure 4.6 Plots of the elevation difference features 115

Figure 4.7 Four examples demonstrating the computation of the feature functions considered in this work for a single transition between two grid cells 117

Figure 4.8 Four examples demonstrating the computation of the fuzzy terrain type features for a single transition between two adjacent grid cells 123

Figure 4.9 Four examples demonstrating the computation of the fuzzy terrain transition features for a single transition between two adjacent grid cells 129

Figure 4.10 Plots of the elevation difference features when one cell is unobserved 132

Figure 4.11 Plots of the expected elevation difference features when only the first cell or the second cell is observed 134

Figure 4.12 Four examples demonstrating the computation of the fuzzy elevation difference features for a single transition between two adjacent grid cells 138

Figure 5.1 An example of determining the local region........ 144

Figure 5.2 Region boundaries computed from the example in Figure 5.1 using Algorithm 5.3, and the region graph defined from the region labels...... 147

Figure 5.3 An example of two regions used to demonstrate the computation of fuzzy region features........ 152

Figure 5.4 Composite distance grids for each of the three boundary edges for the example in Figure 5.3........ 158

Figure 5.5 Individual region and overall distance cost matrices for the example in Figure 5.4, given as the output of Algorithm 5.6. 158

Figure 5.6 Elevation edge costs computed for the example in Figure 5.3 176

Figure 5.7 Composite distance grids computed using Algorithm 5.13 for the example in Figure 5.3 using the maximum aggregation method........ 182

Figure 5.8 Composite distance grids computed using Algorithm 5.13 for the example in Figure 5.3 using the summation aggregation method 182

Figure 5.9 Plots of the elevation difference features over the unit square, with a shaded region showing the area where the function is less than a value x........ 186

Figure 5.10 Plots of the cumulative distribution functions of the elevation difference features 187

Figure 5.11 CDFs of the maximum of n elevation difference feature values 188

Figure 5.12 PDFs of the maximum of n elevation difference feature values 189

Figure 5.13 Expected values of $Y_n^{\text{abs}}$and Y_n^{dir} for n in $1, \ldots, 100$ 192

Figure 5.14 Approximation of the region distances using the region centroids for the example in Figure 5.3 203

Figure 5.15 Approximation of the region distance cost matrices for the example in Figure 5.14. 203

Figure 5.16 Elevation feature edge sets used to approximate the elevation difference features for the example in Figure 5.14 206

Figure 5.17 Step-by-step example of determining new regions 211

Figure 6.1 Example of the selection bias problem for choosing paths in grid-world domains 223

Figure 6.2 Examples of shortest paths chosen between opposite corners of an open grid world 226

Figure 6.3 An example fuzzy weighted graph with two features per edge, distance and slope, represented as triangular fuzzy numbers given in Figure 6.4 234

Figure 6.4 Triangular fuzzy numbers used to represent the distance and slope features for the example graph in Figure 6.3 235

Figure 6.5 Plots of the two-dimensional aggregated fuzzy cost vectors for each path in the example graph from Figure 6.3 236

Figure 6.6 The aggregated fuzzy cost vectors from Figure 6.5 are normalized using the nadir vector and defuzzified using weighted centroid defuzzification 237

Figure 6.7 Examples of different scalarization methods applied to the aggregated fuzzy cost vectors given in Table 6.1 238

Figure 6.8 Exponential scaling of a normalized edge cost x 242

Figure 6.9 Example of crossover and mutation on paths 251

Figure 6.10 Shortest paths found by the MOEA/D algorithm for the MO-FLCPP in a binary terrain environment using the weighted sum scalarization method to minimize $\tilde{f}_{t(1)}$ and $\tilde{f}_{t(2)}$ 254

Figure 6.11 Shortest paths found by the MOEA/D algorithm for the MO-FLCPP in a binary terrain environment using the Tchebycheff scalarization method to minimize $\tilde{f}_{t(1)}$ and $\tilde{f}_{t(2)}$ 256

Figure 6.12 Shortest paths found by the MOEA/D algorithm for the MO-FLCPP in a binary terrain environment using the ordered weighted average (OWA) scalarization method with weight vector $\boldsymbol{\theta} = 0.67, 0.33$ to minimize $\tilde{f}_{t(1)}$ and $\tilde{f}_{t(2)}$ 258

Figure 6.13 Least-cost paths found by the MOEA/D algorithm for the MO-FLCPP in a hilly environment using the ordered weighted average (OWA) scalarization method with weight vector $\boldsymbol{\theta} = 0.67, 0.33$ to minimize $\tilde{f}_d$ and $\tilde{f}_{h_\max}$ 259

Figure 6.14 Least-cost paths to the nearest goal found by the MOEA/D algorithm for the MO-FLCPP in a hilly environment using the ordered weighted average (OWA) scalarization method with weight vector $\boldsymbol{\theta} = 0.67, 0.33$ to minimize $\tilde{f}_d$ and $\tilde{f}_{h_\max}$ 260

Figure 6.15 Least-cost paths to the nearest goal found by the MOEA/D algorithm for the MO-FLCPP in a hilly environment with no region clustering using the ordered weighted average (OWA) scalarization method with weight vector $\boldsymbol{\theta} = 0.67, 0.33$ 262

Figure 6.16 Shortest paths found using terrain transition features 263

Figure 6.17 Pareto optimal least-cost paths found by the MOEA/D algorithm for the MO-FLCPP in many-objective environments 265

Figure 6.18 Environment types used to evaluate the MOEA/D algorithm for the MO-FLCPP 268

Figure 6.19 An agent solving a TPP in the CMM framework with no region clustering 280

Figure 6.20 An agent solving a TPP in the CMM framework using a small local region with no memory and region clustering in all areas 281

Figure 6.21 An agent solving a TPP in the CMM framework using a small local region with memory and region clustering in all areas 282

Figure 6.22 An agent solving a TPP in the CMM framework using region clustering only for unobserved regions and no region clustering elsewhere 283

Figure 7.1 Some selected moments from the greedy policy's solution for the PO-TSP 291

Figure 7.2 Some selected moments from the MMC policy's solution to the same PO-TSP environment used in Figure 7.1 292

LIST OF TABLES

Table 4.1 Crisp terrain type and terrain transition features 114

Table 4.2 Example of the fuzzy terrain type feature 122

Table 4.3 Example of the fuzzy symmetric terrain transition feature 127

Table 4.4 Example of the fuzzy directional terrain transition feature 127

Table 5.1 Expected values of Y_n^{abs} and Y_n^{dir} for various values of n 190

Table 5.2 Original and approximate region features with both regions either observed or unobserved 207

Table 5.3 Original and approximate region features with only one region observed 208

Table 6.1 Aggregated feature values of the example graph in Figure 6.3 235

Table 6.2 Best paths found in the example graph in Figure 6.3 246

Table 6.3 Summary of problem types used to compare MOEA/D to pre-scalarized decomposition 269

Table 6.4 Average percent improvement of MOEA/D over pre-scalarization (region cluster size = 3) 272

Table 6.5 Average percent improvement of MOEA/D over pre-scalarization (no region clustering) 274

Table 6.6 Average percent improvement of MOEA/D over pre-scalarization (region cluster size = 10) 274

Table 6.7 Feature weights for the example greedy agent 277

Table 6.8 Solution costs of the example greedy agent 279

LIST OF ALGORITHMS

Algorithm 2.1 Viewshed Analysis ... 23

Algorithm 2.2 Amanatides and Woo Line Traversal for Visibility ... 28

Algorithm 2.3 MOEA/D ... 53

Algorithm 3.1 Cave Environment Generation ... 62

Algorithm 3.2 Cellular Automata ... 63

Algorithm 3.3 Remove Diagonal Passages ... 64

Algorithm 3.4 Region Partitioning ... 66

Algorithm 3.5 Tabu Sampling ... 67

Algorithm 3.6 Grid Distance ... 69

Algorithm 3.7 Adjust Cluster Centers ... 70

Algorithm 3.8 Assign Cells to Clusters ... 72

Algorithm 3.9 Get Region Centers ... 74

Algorithm 3.10 Fix Orphans ... 76

Algorithm 3.11 Heightmap Generation ... 79

Algorithm 3.12 Generate Binary Terrain ... 82

Algorithm 3.13 Generate Trinary Terrain ... 84

Algorithm 3.14 Fashion-Based Cellular Automata ... 85

Algorithm 3.15 Generate Full World Environment ... 88

Algorithm 4.1 Initialize the Mental Map ... 97

Algorithm 4.2 Get Observation ... 99

Algorithm 4.3 Get Viewshed ... 103

Algorithm 4.4 Update Mental Map ... 106

Algorithm 4.5 Cave Wall Heuristics 107
Algorithm 5.1 Create the Initial Region Graph 142
Algorithm 5.2 Get the Local Region 143
Algorithm 5.3 Create the Initial Mental Map Regions 146
Algorithm 5.4 Update Region Map 147
Algorithm 5.5 Create Region Graph 150
Algorithm 5.6 Get Fuzzy Distance Cost Matrices for Two Regions 156
Algorithm 5.7 Get Region Indices 157
Algorithm 5.8 Get Boundary Edges 157
Algorithm 5.9 Create Region Edge Sets 170
Algorithm 5.10 Compute Region Features 172
Algorithm 5.11 Get Elevation Edge Costs 174
Algorithm 5.12 Elevation Feature 178
Algorithm 5.13 Bellman-Ford Grid Distance 180
Algorithm 5.14 Unobserved Elevation Costs 193
Algorithm 5.15 Combine Elevation Costs 197
Algorithm 5.16 Update Mental Map Regions 210
Algorithm 5.17 Get the Region Clustering Mask 213
Algorithm 5.18 Merge Region Labels 214
Algorithm 5.19 Update Region Graph 216
Algorithm 5.20 Update Region Graph Vertices 217
Algorithm 5.21 Update Region Graph Edges 219
Algorithm 6.1 Normalized Grid Distance 225
Algorithm 6.2 Pre-scalarized Decomposition of the MO-FLCPP 244

Algorithm 6.3 MOEA/D for the MO-FLCPP 248

Algorithm 6.4 A Greedy Algorithm for the CMM Framework 276

ABSTRACT

Multicriteria decision-making problems arise in all aspects of daily life and form the basis upon which high-level models of thought and behavior are built. These problems present various alternatives to a decision-maker, who must evaluate the trade-offs between each one and choose a course of action. In a sequential decision-making problem, each choice can influence which alternatives are available for subsequent actions, requiring the decision-maker to plan ahead in order to satisfy a set of objectives. These problems become more difficult, but more realistic, when information is restricted, either through partial observability or by approximate representations.

Pathfinding in partially observable environments is one significant context in which a decision-making agent must develop a plan of action that satisfies multiple criteria. In general, the partially observable multiobjective pathfinding problem requires an agent to navigate to certain goal locations in an environment with various attributes that may be partially hidden, while minimizing a set of objective functions. To solve these types of problems, we create agent models based on the concept of a mental map that represents the agent's most recent spatial knowledge of the environment, using fuzzy numbers to represent uncertainty. We develop a simulation framework that facilitates the creation and deployment of a wide variety of environment types, problem definitions, and agent models. This computational mental map (CMM) framework is shown to be suitable for studying various types of sequential multicriteria decision-making problems, such as the shortest path problem, the traveling salesman problem, and the traveling purchaser problem in multiobjective and partially observable configurations.

1 INTRODUCTION

The partially observable multicriteria pathfinding problem is well-suited for studying models of agent behavior. In this introductory chapter, we present the motivation for investigating these types of problems and give an overview of the simulation framework developed for this work. We list some of the major contributions of this work and provide some potential applications.

1.1 Problem Statement

Imagine a scenario in which you are tasked with finding the best route through an environment to some goal location. Perhaps there are multiple paths to consider, each with different attributes that make them more or less desirable according to your particular preferences. Figure 1.1 shows an example scene with three different routes to choose from. The shortest route goes directly over a hill, but it is steep and unpaved. The next shortest route goes through a forest that provides shade and has only a mild elevation change, but the route is still unpaved and has a stream crossing with no bridge. The last route is the longest, but it is completely paved and has almost no elevation change. Depending on how you value factors such as the path length, steepness, and path quality, any one of these paths could be considered the best choice. Once you begin down one of the paths, you may discover some new information that causes you to reevaluate your situation and develop a new plan. For example, if you started down the forested path and found that the stream crossing was flooded, you might choose to turn around and go a different way.

Figure 1.1 Example environment with three different path options to reach a goal location.

Now consider an autonomous agent faced with a similar scenario. This could be a robot or drone that needs to navigate through an unknown environment to a goal location while minimizing some set of objective functions such as distance, travel time, and risk. The agent uses various sensors to observe the world around it and constructs an internal representation of the environment in the form of a map. It uses this map to plan a course of action that best satisfies the predetermined criteria and begins to execute the plan. After each movement action, the agent receives a new observation and updates its internal map. If the original plan becomes invalid or a better route is discovered, the agent develops a new plan and responds accordingly.

Although the problem domain in this example is navigating through a physical environment, these types of partially observable sequential multicriteria decision-making problems occur in many additional real-world contexts. These include optimal packet routing through a computer network with uncertain loads, making long-term business decisions based on variable market factors, and designing optimal strategies for games with hidden information. These are all problems that are addressed by a decision-making agent (or agents) with a given set of goals and criteria. When the problem needs to be solved autonomously, such as with a self-guided robot or a recommendation system, the agent behavior should be defined in a structured and explainable way that responds appropriately for a wide variety of possible inputs.

Designing the desired agent behaviors can be a challenging problem. Good training data may not be available and what is available may be limited or incomplete. For many applications, it is often preferable to simulate the problem domain to give the designer complete control over the model. These results can then be applied in real-world contexts for final evaluation. Using a simulated environment allows for the creation of a virtually unlimited number of problem scenarios, each fine-tuned to study only the relevant aspects of the problem. It also allows the agent to easily internalize a representation of the problem domain, which can then be used to plan future actions.

The navigation problem is an ideal domain to study partially observable sequential multicriteria decision-making strategies. It is easy to visualize and understand the agent objectives and to develop interpretable problem scenarios. These can be thought of as proxy problems for other domains that may not be as straightforward to study. The agent's internal model of the environment is represented intuitively as a mental map, providing a

sense of spatial awareness that can help with planning. Spatial problem solving has also been studied extensively within the fields of mobile robotics and environmental psychology. We can build upon these existing models of wayfinding behavior to create simulations of autonomous agents for the navigation problem domain.

The primary focus of this work is the definition and development of the computational mental map (CMM) simulation framework. This framework allows for the creation of pathfinding scenarios that test different agent strategies in multiobjective and partially observable problems. We design these problems as a type of resource collecting game, where the agent moves within a grid world environment seeking out resources that may not be initially visible, all while working to minimize a set of objective functions. We show how the CMM framework can be used to study shortest path problems, the traveling salesman problem, and the traveling purchaser problem with various agent profiles. The result of each problem simulation is the path chosen by the agent for that scenario. Just as there may not be a "correct" answer for the three-route problem in Figure 1.1, solutions to problems in the CMM framework can only be evaluated using some established scoring metric. For some applications, the solution paths themselves are a useful dataset that can be used to anticipate how a given agent might act in a new situation.

The rest of this chapter provides an overview of the CMM framework and details some of the major contributions and potential applications of this work. Chapter 2 provides a literature review of the background material that this work builds upon. Chapter 3 describes the process for creating the grid world environments used to define the pathfinding problems. Chapter 4 introduces the concept of the mental map, used by the agent to represent the observed environment. Chapter 5 defines the region graph, which is

used to summarize the spatial properties of the mental map as a fuzzy weighted graph. Chapter 6 shows how this graph can be used to solve least-cost path problems in gridded domains and presents a greedy agent algorithm. Finally, Chapter 7 concludes this work by summarizing the capabilities of the CMM framework and proposing extensions of the greedy algorithm for improved agent strategies on more complex problems.

1.2 The Computational Mental Map Framework

The computational mental map (CMM) simulation architecture consists of two main components: an environment problem server and an agent program that interacts with the server to solve a specified problem. An overview of the server/client model is shown in Figure 1.2. The server component is responsible for defining the environment model $\mathcal{E}$ and implementing the physics of the world by waiting for and implementing the client's actions. The client acts as the decision-making agent $\mathcal{A}$ and receives information about the environment in the form of observations $\mathcal{O}$ from the server. The agent uses these observations to construct and update a mental map representation of the environment $\mathcal{M}$, which may be incomplete or contain other types of uncertainty or imprecision. The agent's goal is to move through the environment and collect a certain number of predefined resources while minimizing a set of objective functions. Using the information in the mental map, the agent develops a plan that brings it closer to satisfying the goal conditions and sends the appropriate sequence of actions to the server.

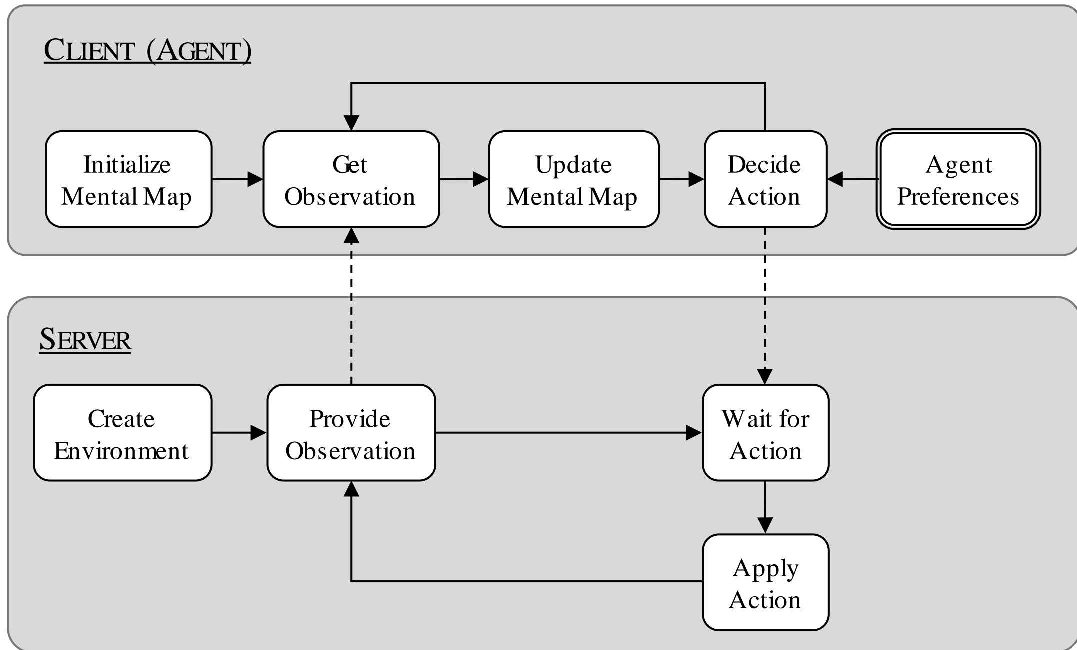


Figure 1.2 Block diagram of the server/client architecture used in the CMM framework.

The environments are procedurally generated grid worlds with various terrain types and elevation. In some environments, a maze-like cavern map is generated to create walls and passageways that reduce visibility. Some example environments are shown in Figure 1.3. The CMM server maintains the location of the agent within the environment and defines the locations of the resources. In shortest path problems, there may only be a single resource (goal) location, whereas multiple resource locations are defined for the traveling salesman and traveling purchaser problems. In the traveling salesman problem, each resource is the same type, whereas in the traveling purchaser problem, there are different types of resources that the agent can choose from. Details regarding the creation of the grid world environments are given in Chapter 3.

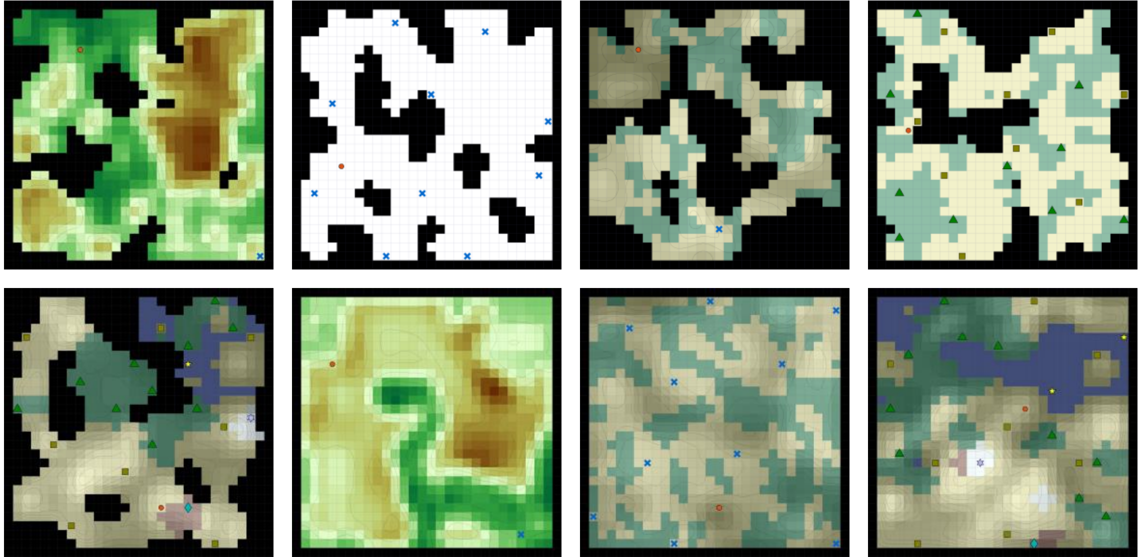
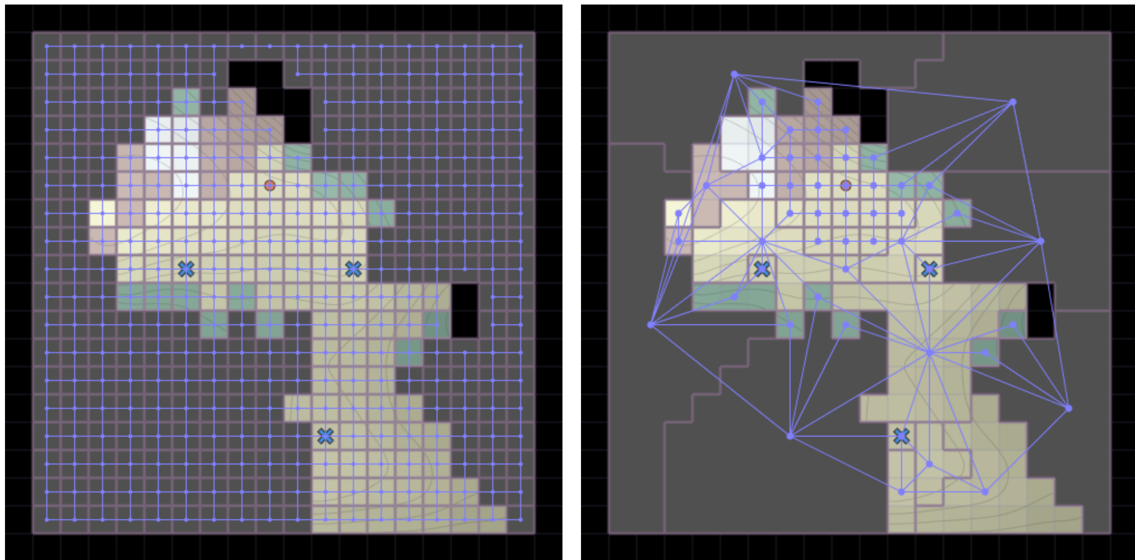


Figure 1.3 Examples of grid world environments from the CMM framework.

The agent can move through the environment in discrete steps to adjacent grid cells. Each step has multiple costs associated with it, defined by the environment attributes: terrain type, elevation, and observability. The agent specifies to the server a direction to move in and the server responds with an observation of the visible part of the environment from the agent's new location. The agent uses these observations to form a mental map image of the grid world, showing where resources and environmental features are located. Chapter 4 describes how observations are computed in the grid world domain and introduces the action graph, which is a fuzzy weighted graph that represents the movement actions available to the agent and their costs. Figure 1.4 (a) shows an example of a mental map and action graph where only part of the environment is observable.



(a) (b)

Figure 1.4 An agent’s mental map for an example scenario showing the action graph (a) and the region graph (b). The action graph shows where the agent can move and the region graph summarizes this information for high-level planning.

For some planning algorithms in large environments the action graph contains too much information to process quickly. To simplify the action graph and to provide a model that more closely resembles human models of spatial reasoning, we develop an approach to group adjacent grid cells into regions and define a region graph that can be used for high-level planning. Figure 1.4 (b) shows an example of a region graph, where each node represents multiple adjacent grid cells. As the agent moves, new parts of the environment are observed and the region graph is recomputed. Chapter 5 discusses the process for initializing and updating the region graph in the CMM framework.

We generalize the various pathfinding problem types as a resource collecting game. The agent is initialized with a list of demanded resources and the agent's goal is to plan a route through the environment that will collect enough of each resource type. When the

full environment is observable, the agent can develop a complete plan before actually moving. However, if there are parts of the environment that cannot be seen initially, then the agent may only be able to produce a partial plan that is updated as new information is discovered. Finding the least-cost path between two grid cells in an environment is the fundamental component of nearly all agent strategies. Chapter 6 describes how an agent can find a least-cost path in a fuzzy weighted graph with multiple objectives. We consider both gridded environments and general graphs and present a greedy agent strategy for solving generic resource collecting problems in the CMM framework.

There are many variations that can be applied within the CMM framework to focus on different aspects of the overall planning and optimization process. In general, these can be divided into the parameters that control how the environment is created and those that govern the behavior of the agent. While this work focuses mainly on environment generation parameters and a generic greedy approach for solving least-cost path problems, there are many additional potential applications. Chapter 7 concludes with a summary of the capabilities of the CMM framework and discusses some possible directions for future work with more advanced agent behaviors. These include strategies based on ant colony optimization and Markov decision processes.

1.3 Contributions and Potential Applications

Two of the major themes of this work are multiobjective optimization and planning under uncertainty. Both areas have seen significant research interest for a variety of applications. Multiobjective optimization is used for designing products and systems, making strategic decisions in economic and business settings, and managing limited

resources and conflicting objectives. Likewise, planning in partially observable environments is a critical capability for mobile robots and decision-making agents. Multiple methods and techniques have been developed to assess multiobjective problems and to help a decision-maker choose an appropriate solution. The vast majority of multiobjective optimization methods are used to address problems with no uncertainty. Also, most planning techniques are designed with a single objective or reward in mind, reducing multiple costs to a single value before optimizing. A major goal of this work is to develop tools and methods that can be used to study multiobjective decision-making and planning under uncertainty.

The CMM framework is designed to be a benchmark and simulation tool that can model the behavior of a decision-making agent in various configurable scenarios. The problem domain uses pathfinding in partially observable grid world environments to provide a goal for each agent to pursue. While this allows for an interpretable explanation of each example, these problems can also be used as generic templates for other applications. For instance, the environment creation process could be modified to produce a specific type of fuzzy weighted graph that matches a real-world problem requiring a least-cost path solution. This could occur in the development of personalized navigation systems or robot navigation. Many of the methods presented in this work can likewise be extended to other problem domains. For example, the process of computing a region graph of a grid world environment could be used to develop image features for classification or analysis.

One potential application of this work is to provide an unlimited number of training and testing examples of simulated agent behavior. This can be used to develop techniques for performing anticipatory analysis, which is a desirable capability in the intelligence

community. Human behavior is difficult to predict, but an approximate model of behavior can be learned by observing the responses of a decision-maker in many different situations. A richly attributed environment with many feature values can help to make sure that a given agent's choices are unique and distinguishable from other agents. In the context of a game environment, player modeling allows an opponent to anticipate the player's next actions and develop a more effective plan.

The CMM framework presented in this work is a starting point for developing more complex models of agent behavior. We build upon existing models of cognitive mapping and wayfinding, and begin with a foundation in procedurally generated environments, fuzzy methods, and multiobjective optimization. These are introduced along with other background concepts in the next chapter.

2 BACKGROUND

The CMM framework developed in this work draws upon many diverse backgrounds. We begin this chapter with a review of cognitive mapping and its origins for creating models of human wayfinding behavior. We next introduce fuzzy numbers and the fuzzy weighted graph representation. Then, we discuss methods of procedural content generation used to create the grid world environments. We continue with a comprehensive background on multiobjective optimization and end with a discussion on agent-based models.

2.1 Wayfinding and Cognitive Mapping

Wayfinding can be described as the process of spatial problem solving. In the wayfinding problem, a decision-making agent orients itself in an environment and navigates to some destination. It uses landmarks and cues to determine its position and to determine the best route to take. As the agent moves, it continues to update its plan using any new information that is acquired. Humans and animals routinely solve wayfinding problems in their everyday lives as they move about their environments, working to satisfy their goals and objectives. Autonomous agents are also being used increasingly to assist people in making navigation decisions and to carry out actions without human input. Self-driving cars, unmanned aerial vehicles, and other mobile robots are just a few examples of machines that must think on their own about how to solve problems of spatial navigation.

In general, wayfinding is a challenging computational problem that can be compounded by a lack of knowledge or perfect information. Furthermore, a decision-maker

may have multiple conflicting objectives that cannot all be optimized simultaneously. Within these partially observable multi-objective environments, an agent must decide how best to navigate using only the information that is available. For humans, this process is called cognitive mapping and it results in an information structure known as a mental map. Although humans may not be optimal problem solvers from a computational point of view, the concept of the mental map has proven to be a useful way to represent imprecise spatial knowledge and develop navigation plans.

The CMM framework is inspired by the study of how a person might make navigation decisions in an unfamiliar environment. The notion of using a cognitive map to represent spatial information dates as far back as the seminal work of Tolman (Tolman 1948), who established the now famous paradigm of studying decision-making behavior by observing how rats move through mazes. His work helped establish the fields of cognitive psychology and decision theory. Since these early studies, dozens of researchers have proposed behavioral models to explain the way humans make decisions in physical environments (Kitchin and Blades 2002). **Error! Reference source not found.** shows an example diagram of the general cognitive mapping process. An individual acting in an unknown environment maintains a set of beliefs about the world that influence the values or goals he or she wishes to achieve. These are combined with the most recent observation of the world to form a spatial image of the environment in working memory. This image is stored for later use and also updates the individual's beliefs about the world. A set of physical constraints are evaluated with the image to form a decision of the next immediate action to take. When applied in the environment, this action produces a behavior that can be observed in the real world and results in some new information presented to the

individual. The cycle repeats indefinitely, with the individual's desires and beliefs changing over time. Many variations of this general framework have been developed for use in various problem domains.

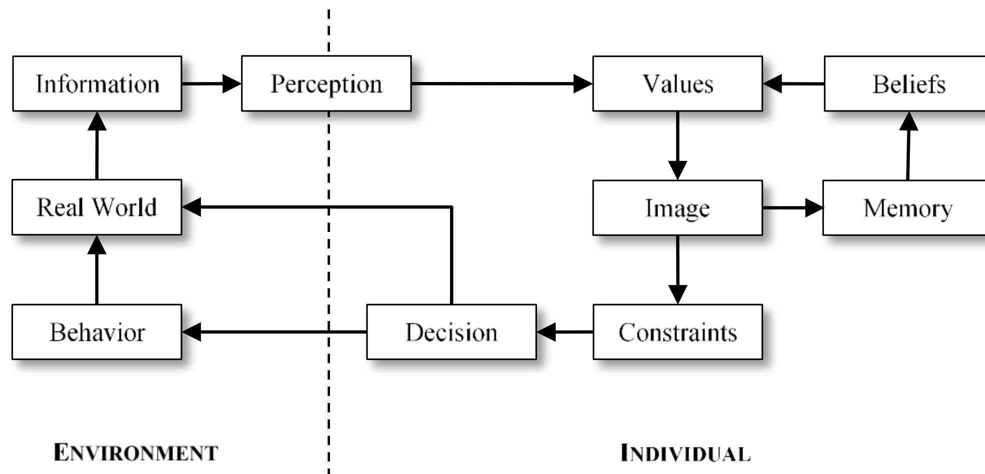


Figure 2.1 Generative model of cognitive mapping adapted from (Kitchin and Blades 2002). An individual observes the environment and constructs a mental image. This image is combined with other knowledge to produce a decision in the environment.

A mental map is the manifestation of an individual's spatial knowledge and beliefs into a geospatial context. Mental maps are often studied by asking a person to draw a map of their environment, or to relate spatial quantities such as the distance between two landmarks (Gould and White 1992). For instance, a mental map of an urban environment could be represented as a hierarchical structure consisting of paths, edges, districts, landmarks, and nodes (Lynch 1960). The cognitive distances within a mental map are unique to each individual and can be based on a variety of factors, including the amount of expected energy required to move along a route, patterns in the environment, and symbolic representations such as maps and road signs (Briggs 1973). These individual differences

lead to distortions in the map that may not necessarily align with ground truth data (Coucleis et al. 1987).

In general, mental maps may contain additional types of knowledge and reasoning processes besides just spatial information. Rules that govern an individual's behavior dictate important spatial decisions, such as the decision to move or not move, where to go, which route to take, and the method of transportation (Cadwallader 1976; Gärling, Book, and Lindberg 1985). In order to use a mental map for navigation, a person must first orient and conflate his or her mental map with the real world, identify an objective, and choose a route to follow (Downs and Stea 1977). The PLAN model (Prototypes, Location, and Associative Networks) (Chown, Kaplan, and Kortenkamp 1995) is an example method that implements this wayfinding process. In this model, the visual recognition of "what" is in the scene is combined with the spatial knowledge of "where" the visual landmarks appear. The landmarks are related to each other with a spatial relational graph that describes their relative positions. This attributed graph can then be used to represent the individual's mental map.

2.2 Procedural Content Generation

The study of wayfinding problems is often hampered by the difficulty of conducting controlled research experiments. Studies involving human subjects are limited by available time and resources, and usually consist of a relatively small number of data samples. Designing appropriate problems to solve can be a challenging and time-consuming task for a researcher, who may seek to use some automated methods for assistance. Using a game

engine as a synthetic problem domain to study wayfinding problems allows for the creation of a nearly infinite number of environments and scenarios.

Procedural Content Generation (PCG) for games can be defined as “the algorithmic creation of game content with limited or indirect user input” (Togelius et al. 2011). PCG is often used to produce content for games such as levels, maps, items, game rules, etc. A “game” in this context may refer to videogames, board games, puzzles, or any sort of interactive experience that is in some way “playable.” The value of using PCG over manual content creation is that PCG allows a computer algorithm to perform a task that might take a long time for a human designer. Furthermore, PCG can be parameterized in such a way that the generated content exhibits a desired set of properties. A designer can use PCG to enhance their own creativity by creating novel and unexpected solutions to content generation problems (Shaker, Togelius, and Nelson 2016).

There are many different approaches to PCG that can be used to create specific types of content. In this work, we use PCG to create environment maps that exhibit desirable characteristics for the problems we wish to study. We focus mainly on two common approaches: cellular automata and fractal terrain. For cellular automata, we consider both traditional and fashion-based update rules. These are described in the following sections.

2.2.1 Cellular Automata

A cellular automation is an iterative computational model that operates over a discrete domain. Perhaps the most famous example is Conway’s Game of Life (Gardner 1970) that simulates a grid of cells that can evolve into complex patterns demonstrating

emergent behavior and self-organization. In the Game of Life, the domain is a flat grid of cells that each can be in one of two states: alive or dead. We define a neighborhood for each cell, consisting of the eight neighboring cells, including those diagonally adjacent. This is called the Moore neighborhood and is just one of many possible neighborhood definitions. (Another possibility is the von Neumann neighborhood, which consists of only the four orthogonally adjacent cells.)

A simulation of a cellular automation iterates through a sequence of states s_t, where $t \geq 0$ indicates the time step. The initial state s_0 defines the starting state of each grid cell as either alive or dead. For each subsequent time step, the new state of a cell $x_{(i,j)}$ is defined by the current states of the cells in its neighborhood. A transition rule can be defined as a lookup table over all possible neighborhood configurations, or more commonly as a function of the proportion of neighboring cells that are in each state. Each rule gives rise to a unique behavior that can be classified based on whether it converges to a stable or periodic state, or if it exhibits chaotic non-repeating behavior (Packard and Wolfram 1985). In the Game of Life, the transition rule is specified using the following conditions:

1. A living cell that has two or three living neighbors survives to the next generation.
2. A living cell with more than three living neighbors dies from overpopulation.
3. A living cell that has fewer than two living neighbors dies from isolation.
4. A dead cell that has exactly three living neighbors becomes alive as through reproduction.
5. A dead cell with any number other than three living neighbors remains dead.

These rules are applied to each cell simultaneously to produce the next generation. The initial configuration of the cell states defines how the simulation will evolve. Various patterns have been discovered that result in fascinatingly complex configurations, such as blinkers, gliders, spaceships, and pulsars (Berlekamp, Conway, and Guy 1982). The Game of Life can even be configured as a universal Turing machine that (given enough time and space) is theoretically as powerful as any computer (Chapman 2002)!

Various types of cellular automata have been shown to be useful for procedural content generation. One example by Johnson et al. uses a cellular automation to generate two-dimensional cave-like mazes in real-time for an infinite game map (Johnson, Yannakakis, and Togelius 2010). In this approach, the two cell states represent floor and rock. The grid is initialized to some random state where each cell has an equal likelihood of being either floor or rock. The eight cells in the Moore neighborhood are evaluated for each cell and if there are five or more neighbor cells that are rock, the cell is set to rock. Otherwise the cell is set to floor. This single rule is applied simultaneously n times to generate the cave map. For aesthetic reasons, rock cells that border a floor cell are labeled as walls and contiguous rock regions are assigned unique labels. By varying the size of the Moore neighborhood, the rock threshold value, and the number of iterations, various types of maps can be generated.

2.2.2 Fashion-based Cellular Automata

A cellular automation can be defined with more than just two states for each cell. One way of modeling this is with the use of fashion-based cellular automata (Ashlock 2015). In this approach, we use the von Neumann neighborhood containing the four

orthogonally adjacent cells and a $k \times k$ real valued rule matrix R, where k is the number of states. The entry $R_{i,j}$ in the rule matrix specifies the score that a cell of type i receives if it has a neighbor in state j. Each generation the cells are all updated simultaneously and the total score of each cell is evaluated using the rule matrix. If the score of a cell is at least as high as its neighbors, it remains in the same state, otherwise it adopts the state of the neighboring cell with the highest score. This causes cells to "follow the fashion" of the neighborhood and results in large homogeneous regions that are well-suited for representing environment maps.

2.2.3 Fractal Terrain

While cellular automata are well suited for generating discrete environments consisting of a finite number of states, we often require the terrain to consist of real values to represent features such as elevation. A real-valued grid used to represent elevation is called a heightmap and is commonly used as a basis for artificial terrain. Random heightmaps can be generated via several different methods including value- or gradient-based interpolation such as Perlin noise (Perlin 1985), or using ideas from fractal mathematics to mimic the multiple scales of repeating patterns found in nature (Mandelbrot 1983). Fractional Brownian noise (Mandelbrot and Van Ness 1968) provides the basis for a random function that is useful for modeling naturally occurring time series and surfaces. The diamond-square algorithm (Fournier, Fussell, and Carpenter 1982) is a computationally efficient method for approximating fractional Brownian motion to produce a two-dimensional heightmap. The resulting fractal terrain exhibits random

variations at multiple scales with large hills and valleys as well as small undulations on the surface.

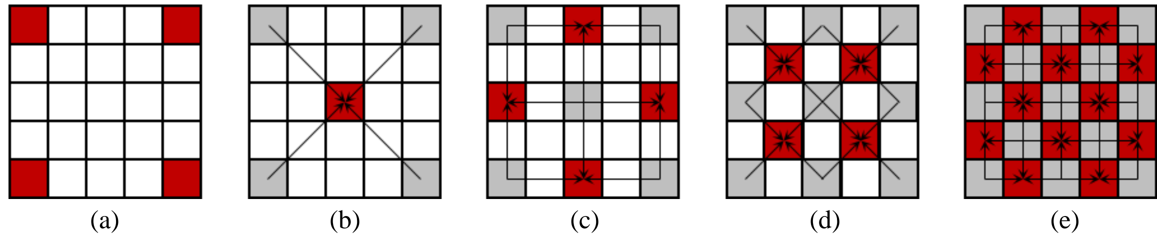


Figure 2.2 Visualization of the diamond-square algorithm on a 5×5 grid.

Figure 2.2 shows an overview of the diamond-square algorithm on a 5×5 grid. The algorithm begins by sampling random values for the four corner cells (a). The diamond step then sets the center point of these cells to the average of the four corners plus an additional random value (b). The magnitude of the random value is called the roughness and determines the texture of the terrain. Next, the square step interpolates the midpoints of the cells from the previous two steps and adds a random value proportional to the roughness (c). The original four cells defining a square have now been subdivided into four smaller squares. The diamond (d) and square steps (e) are then applied to each of the newly formed squares recursively using a smaller roughness value (typically half of the previous amount). These two steps repeat until the entire grid has been set. Note that the original grid should be square with $2^n + 1$ pixels on each side. Figure 2.3 shows an example of the diamond-square algorithm progression on a 257×257 grid. Note that basic terrain features are defined in the first few steps of the algorithm, with later steps serving to refine the terrain and add details.

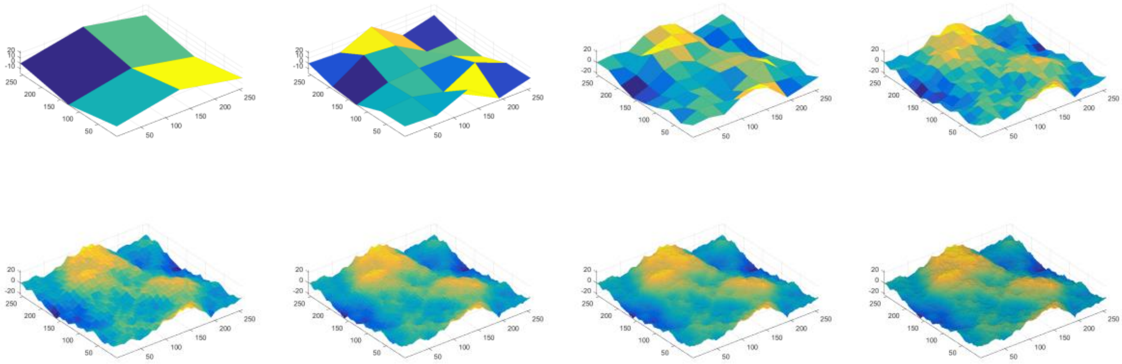


Figure 2.3 A progression of the diamond-square algorithm generating a fractal terrain.

As an alternative to midpoint displacement algorithms such as the diamond-square algorithm, successive random additions can also be used to generate fractal terrain with similar characteristics (Musgrave, Kolb, and Mace 1989). In this approach, several noise functions are generated at multiple levels of detail. These are then summed together using a weight that is inversely proportional to the level of detail. Figure 2.4 shows an example using multiple octaves of white noise, scaled to match the output size of the terrain. Each octave n is generated by creating an image with 2^{n+1} pixels in each dimension containing random values. These images are scaled to the size of the final output using bilinear interpolation and summed together using a weight of $\frac{1}{2^{n-1}}$ for each octave to produce the combined noise function. The resulting image can represent a heightmap that is qualitatively similar to the terrain generated using the diamond-square algorithm. The method of successive random additions is simple to implement and allows for additional control over the characteristics of the noise function at each scale.

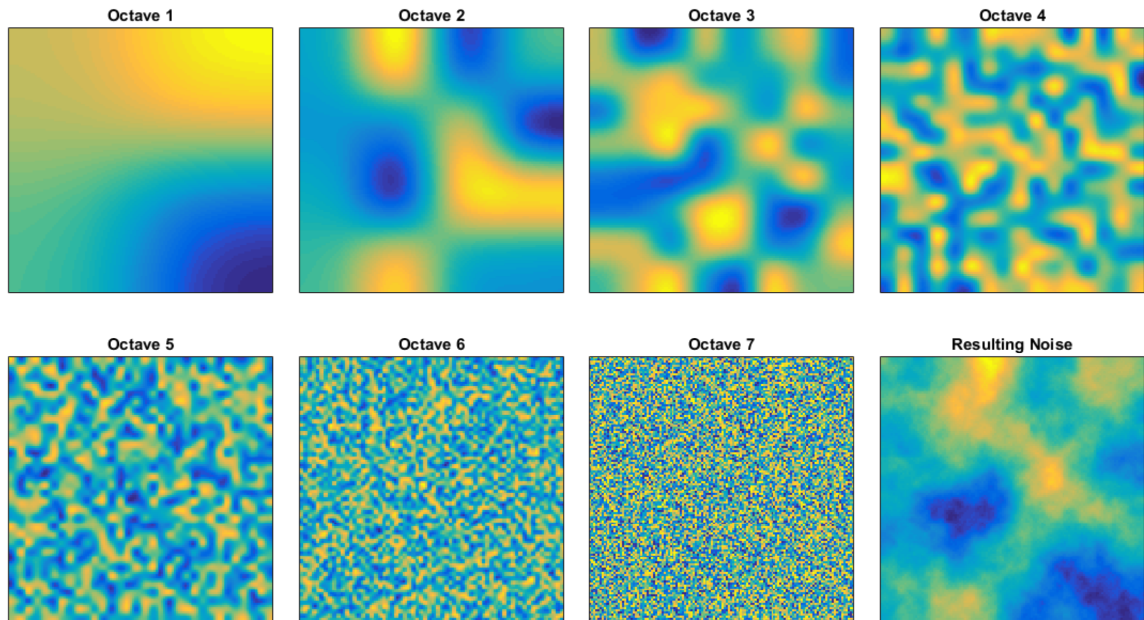


Figure 2.4 An example of the successive random additions method for generating fractal terrain.

2.3 Viewshed Analysis

Problems of visibility arise in many application domains, including computer graphics, robotics, and computational geometry (Durand 2000). In geographic information systems (GIS), the visibility problem is expressed as determining the viewshed of a region from a given location. For an elevation model represented as a regular square grid, viewshed analysis is used to find the grid cells that have a direct line of sight (LOS) from a specified observation point. These cells comprise the viewshed region and can be used for many applications including planning the placement of communication towers or watchtowers, path planning, and strategic defense (Franklin and Ray 1994; Floriani and Magillo 1994).

The viewshed region for a given point p in a raster grid elevation model E consists of all points v where a straight line can be drawn from p to v that is entirely above the terrain in E. Many algorithms have been proposed to compute the viewshed region, most based on sweeping rays (Kreveld 1996; Fishman, Haverkort, and Toma 2009; Haverkort, Toma, and Zhuang 2009) or parallel processing approaches (Zhao, Padmanabhan, and Wang 2013). Algorithm 2.1 gives an overview of an unoptimized approach for computing the viewshed called R3 that evaluates all grid cells independently. If r is the radius of the viewshed, then this method takes $O(r^3)$ time to evaluate each grid cell sequentially, but it can be implemented in parallel to reduce the computation time. Our own algorithm is detailed in Section 4.1 and builds upon the method presented here.

Algorithm 2.1 Viewshed Analysis

GET_VIEWSHED_R3(E, x_1, y_1, h)

/ Precompute the elevation angle to each grid cell */*

1: $(n, m) \leftarrow$ size of E

2: $A \leftarrow n \times m$ grid initalized to 0

3: **for** each $(x_2, y_2) \in \{(x_2, y_2) \mid 1 \leq y_2 \leq n \wedge 1 \leq x_2 \leq m \wedge (x_1, y_1) \neq (x_2, y_2)\}$

4: $\quad A[y_2, x_2] \leftarrow \tan^{-1}\left(\frac{E[y_2,x_2]-E[y_1,x_1]-h}{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}}\right)$

/ Evaluate the visibility of each grid cell */*

5: $V \leftarrow n \times m$ grid initalized to 0

6: **for** each $(x_2, y_2) \in \{(x_2, y_2) \mid 1 \leq y_2 \leq n \wedge 1 \leq x_2 \leq m\}$

7: $\quad v \leftarrow$ CHECK_VISIBILITY(A, x_1, y_1, x_2, y_2) *// Algorithm 2.2*

8: $\quad V[y_2, x_2] \leftarrow [v > 0]$

9: **return** V

The algorithm begins by precomputing the elevation angle from the observation point to each grid cell in the terrain (lines 1-4). We assume that the elevation of a grid cell is represented by its center point and that the observer is standing at height h above the observation point $p = (x_1, y_1)$. The horizontal distance to a grid cell $v = (x_2, y_2)$ is computed as

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}, \tag{2.1}$$

and the vertical elevation difference is

$$e = E[y_2, x_2] - E[y_1, x_1] - h. \tag{2.2}$$

From this we compute the elevation angle as

$$a = \tan^{-1} \frac{e}{d}. \tag{2.3}$$

For a grid cell v to be visible from the observation point p, the elevation angle from p to v must be greater than the elevation angle from p to any grid cell on a line from p to v (see Figure 2.5). Most viewshed algorithms that operate on discrete grid elevation models, including R3, are approximate in the sense that there is no notion of partial visibility for a grid cell. (A cell is either entirely visible or entirely hidden.) We use a raytracing algorithm to identify the grid cells that intersect the line from p to v and consider only the elevation angles to these cells when determining visibility. The commonly used Bresenham line drawing algorithm (Bresenham 1965) is unacceptable here because it does not return all cells that intersect the line. Instead, we use the Amanatides and Woo algorithm (Amanatides and Woo 1987), which is given in Algorithm 2.2. The CHECK_VISIBILITY function that implements this algorithm is applied to all grid cells in the environment and

used to construct the final viewshed (lines 5-8 in Algorithm 2.1). All points with a visibility greater than 0 are marked as visible in the viewshed.

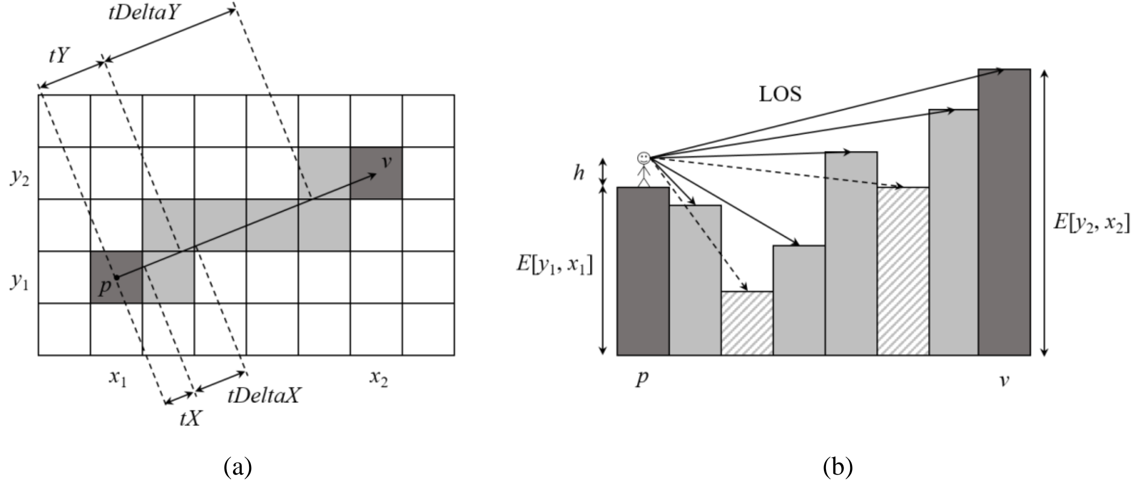


Figure 2.5 Viewshed analysis of the grid cell $v = (x_2, y_2)$ from the observation point $p = (x_1, y_1)$. (a) shows a line drawn from p to v and all grid cells that the line intersects (shaded). These cells are evaluated to see if any obstruct the line of sight (LOS). The variables *tX*, *tY*, *tDeltaX* and *tDeltaY* are used by the Amanatides and Woo line traversal algorithm in Algorithm 2.2. (b) shows the elevation profile of the shaded grid cells in (a). For a grid cell v to be visible from p, it must have a clear LOS from the observation point, set at a height h above the elevation of p. If the elevation angle of any grid cell between p and v is greater than the elevation angle from p to v, then the LOS is obstructed and the cell is not visible. The striped cells are not visible in this example.

Given the vectors **p** and **v** from the origin to the points $p = (x_1, y_1)$ and $v = (x_2, y_2)$, the vector representation of the line going from p to v is defined as $\mathbf{p} + t\mathbf{u}$, where $\mathbf{u} = \mathbf{v} - \mathbf{p}$. The Amanatides and Woo algorithm works by increasing t from 0 to 1 and identifying all the grid cell boundary crossings of this vector. These occur at the regular intervals *tDeltaX* and *tDeltaY*. *tDeltaX* represents the amount that t increases between vertical cell boundary crossings and is computed as

$$tDeltaX = \sqrt{\left(\frac{d_y}{d_x}\right)^2 + 1}, \tag{2.4}$$

where $d_x = x_2 - x_1$ and $d_y = y_2 - y_1$. Likewise, *tDeltaY* represents the amount that *t* increases between horizontal cell boundary crossings and is computed as

$$tDeltaY = \sqrt{\left(\frac{d_x}{d_y}\right)^2 + 1}. \tag{2.5}$$

These values are computed in lines 1-4 of Algorithm 2.2. Line 5 computes the value of *t* at the first vertical crossing as

$$tX = \sqrt{\left(\frac{d_y}{2d_x}\right)^2 + \left(\frac{1}{2}\right)^2}. \tag{2.6}$$

This variable will be updated to always store the value of *t* at the next vertical crossing. Line 6 computes the value of *t* at the first horizontal crossing as

$$tY = \sqrt{\left(\frac{d_x}{2d_y}\right)^2 + \left(\frac{1}{2}\right)^2}. \tag{2.7}$$

This variable will also be updated to store the value of *t* at the next horizontal crossing. Lines 7-8 determine the signs of d_x and d_y (+1, –1) and save these as *stepX* and *stepY* respectively. These values will be used to increment the current cell location, saved as *X* and *Y*, and initialized to x_1 and y_1 on lines 9-10. The main loop (lines 11-19) of the algorithm repeats until (*X*, *Y*) is equal to (x_2, y_2). Each iteration, the variables *tX* and *tY* are compared to see if the next boundary crossing is horizontal or vertical. If *tX* is less than *tY*, then the next crossing is a vertical boundary so *tX* is incremented by *tDeltaX* and *X* is incremented by *stepX* (lines 13-14). Otherwise, the next crossing is a horizontal boundary, so *tY* is incremented by *tDeltaY* and *Y* is incremented by *stepY* (lines 16-17). If at any time the current grid cell (*X*, *Y*) has an elevation angle from the observation point (x_1, y_1) that is

greater than the elevation angle of the target point (x_2, y_2), then the target point does not have a clear line of sight from the observation point and the algorithm returns 0 (lines 20-21). Lines 18-19 handle an edge case where there is an infinite wall obstructing the line of sight. If such a wall is detected, the algorithm returns –1, which is handled as a special case in our own algorithm in Section 4.1. If the target point is reached, then that indicates that there were no grid cells along the path that obstruct the view from the observation point, so the algorithm returns 1 (line 22).

Algorithm 2.2 Amanatides and Woo Line Traversal for Visibility

CHECK_VISIBILITY(A, x_1, y_1, x_2, y_2)
1: $d_x \leftarrow x_2 - x_1$
2: $d_y \leftarrow y_2 - y_1$
3: $tDeltaX \leftarrow \sqrt{\left(\frac{d_y}{d_x}\right)^2 + 1}$
4: $tDeltaY \leftarrow \sqrt{\left(\frac{d_x}{d_y}\right)^2 + 1}$
5: $tX \leftarrow \sqrt{\left(\frac{d_y}{2d_x}\right)^2 + \left(\frac{1}{2}\right)^2}$
6: $tY \leftarrow \sqrt{\left(\frac{d_x}{2d_y}\right)^2 + \left(\frac{1}{2}\right)^2}$
7: $stepX \leftarrow \mathrm{sign}(d_x)$
8: $stepY \leftarrow \mathrm{sign}(d_y)$
9: $X \leftarrow x_1$
10: $Y \leftarrow y_1$
11: **while** $(X, Y) \neq (x_2, y_2)$
12: **if** $tX < tY$
13: $tX \leftarrow tX + tDeltaX$
14: $X \leftarrow X + stepX$
15: **else**
16: $tY \leftarrow tY + tDeltaY$
17: $Y \leftarrow Y + stepY$
18: **if** $A[Y, X] = \mathrm{NIL}$
19: **return** -1
20: **if** $A[Y, X] > A[y_2, x_2]$
21: **return** 0
22: **return** 1

2.4 Least-Cost Paths in Fuzzy Weighted Graphs

The shortest path problem is one of the fundamental problems in graph theory that finds use in countless applications. These include finding the optimal path between two

points on a map, routing information through a computer network, and determining a series of actions that can solve a sequential decision problem. In general, solutions to these problems minimize some notion of the cost that is associated with each possible option. In many cases, the true cost of each solution component is unknown, or is dependent on multiple factors. For instance, when choosing a route between two locations in an environment, a decision-maker may have various objectives to satisfy such as minimizing the total distance and the maximum slope. The lengths and inclinations of each path segment may only be partially known due to limited visibility, leading to some uncertainty as to which path to choose. In these situations, it can be useful to model the problem using fuzzy cost values and a multiobjective framework (Buck, Keller, and Popescu 2014).

To represent an agent's mental map, we use a graph structure that models the spatial and semantic attributes of the environment. We define the structural component as a graph G with vertex set $V(G)$ and edge set $E(G)$. Each vertex $v \in V(G)$ represents a location or state, and edges represent possible actions or movements between locations. In a directed graph, the edge set $E(G) \subseteq V(G) \times V(G)$ consists of all ordered pairs of vertices (v_s, v_t) that are connected by an edge. An edge $e \in E(G)$ has both a starting vertex $v_s = \text{START}(e)$ and an ending vertex $v_t = \text{END}(e)$. A path p through the graph is represented as an n-tuple $(e_1, \ldots, e_n) \in \left(E(G)\right)^n$ where $\text{END}(e_i) = \text{START}(e_{i+1})$ for $i = 1, \ldots, n-1$. The starting and ending vertices of the path are denoted as $s = \text{START}(e_1)$ and $t = \text{END}(e_n)$ respectively. The set $P(s,t)$ is defined as the set of all paths between vertices s and t. Each edge is assigned a feature vector that represents the attributes of the environment. A multiobjective problem can have many feature dimensions, whereas a single-objective problem will only

have one dimensional features. The feature values are crisp numbers when the environment is fully observable, but fuzzy numbers are used in partially observable environments to represent uncertainty.

2.4.1 Least-Cost Path Problems

A standard weighted graph assigns a real-valued weight w_i to each edge $e_i \in E(G)$. The meaning of the weight value is arbitrary, although it typically represents some measure of the edge length or cost associated with including the edge in a path. The shortest path problem is defined as finding the path $(e_1, \ldots, e_n) \in P(s, t)$ between two vertices s and t in a graph G such that the sum $\sum_{i=1}^{n} w_i$ is minimized. There are several algorithms that are commonly used to solve shortest path problems.

Dijkstra's algorithm (Dijkstra 1959) can be used in graphs with non-negative weights to find a single-pair shortest path, or a tree of shortest paths to all vertices from a single source, known as the single-source shortest path problem. The algorithm works by expanding a search tree from the source vertex, always adding the edge with the minimum weight. A naïve implementation operates in $O(|V|^2)$ (where $|V|$ is the number of nodes), but this can be improved to $O(|E| + |V| \log|V|)$ (where $|E|$ is the number of edges) by using a min-priority queue such as a Fibonacci heap (Fredman and Tarjan 1984; Fredman and Tarjan 1987). If an admissible heuristic is available (Russell and Norvig 2009), the algorithm can be improved to select edges that minimize the sum of the edge weight and the estimated remaining distance to the goal. This algorithm is called A* (Hart, Nilsson, and Raphael 1968) and is commonly used to solve pathfinding problems where the path weight corresponds to total distance.

If the graph contains negative edge weights, the Bellman-Ford algorithm (also sometimes called the Bellman-Ford-Moore algorithm) (Ford Jr. 1956; Bellman 1958; Moore 1957) can be used to construct the shortest path tree from a single source and can detect negative cycles (path loops that have negative total weight thereby removing a lower bound on the minimum cost of a path). The Bellman-Ford algorithm operates by iteratively relaxing an upper bound on the cost to reach each vertex from the source. In the worst case, it operates in $O(|V||E|)$, but it can terminate early if no changes are detected.

For some applications, we may need to find the shortest paths between all pairs of vertices. This is called the all-pairs shortest path problem and it can be solved by finding a shortest path tree from each vertex using one of the above algorithms. Alternatively, the Floyd-Warshall algorithm (Floyd 1962; Warshall 1962) is specifically designed to solve this problem and does so by iteratively relaxing a $|V| \times |V|$ matrix containing the shortest path distances between each pair of vertices (and optionally a second matrix containing the predecessor of each vertex). The Floyd-Warshall algorithm runs in $O(|V|^3)$ and recursively computes for each triplet $(i, j, k) \in |V|^3$ the shortest path between i and j using only vertices $1, \dots, k$.

The optimal path in some contexts is not always the shortest path. For instance, to optimize traffic flow in transportation and computer networks, it can be useful to identify the maximum capacity route (Hu 1961; Pollack 1960) (sometimes called the bottleneck shortest path) that maximizes the minimum-weight edge in the path. A related problem that we consider is finding the minimax path (Berman and Handler 1987), which minimizes the maximum-weight edge in the path. This can be used to find paths that avoid certain high-

cost areas. In general, a least-cost path is a path that minimizes some measure of the path cost and may refer to either a shortest path or a minimax path.

2.4.2 Fuzzy Numbers

Partially observable environments introduce uncertainty into the wayfinding problem. Fuzzy sets (Zadeh 1965) are a way to model certain types of uncertainty that arise in the representation of partially observable environment features. A fuzzy number $A \subseteq \mathbb{R}$ is a normalized convex fuzzy set with a membership function $\mu_A: A \rightarrow [0, 1]$ that specifies the degree to which a real number $x \in \mathbb{R}$ is included in the set A. Fuzzy numbers provide a way to represent uncertainty in the true value of a number and to express linguistic approximations such as "about 3" or "nearly 10." Some common representations for fuzzy numbers include triangular and trapezoidal membership functions, which are defined by 3 or 4 parameters respectively. We use triangular fuzzy numbers throughout this work to demonstrate our approach, but other representations (such as trapezoidal membership functions or a list of alpha-cut endpoints) could be used when deemed appropriate by the problem domain. A triangular fuzzy number A is defined by a 3-tuple $\mathrm{Tri}(a, b, c)$, where the interval $[a, c]$ is the support for which $\mu_A(x) > 0$ and b is the single point where $\mu_A(x) = 1$. Its membership function is defined as

$$\mu_A(x; a, b, c) = \begin{cases} 0, & x \leq a \\ \dfrac{x-a}{b-a}, & a < x < b \\ 1, & x = b \\ \dfrac{c-x}{c-b}, & b < x < c \\ 0, & x \geq c \end{cases} \quad . \tag{2.8}$$

The arithmetic operators $(+, -, \times, \div)$, as well as other functions such as minimization and maximization, can be defined for fuzzy numbers using Zadeh's extension principle (Zadeh 1975a; Zadeh 1975b). The result of a function $f(A, B)$ operating on two fuzzy numbers A and B is given as

$$\mu_{f(A,B)}(z) = \sup_{z=f(x,y)} \min\big(\mu_A(x), \mu_B(y)\big). \tag{2.9}$$

In this work, we focus mainly on the summation and maximization operators for triangular fuzzy numbers. The summation of two triangular fuzzy numbers is derived from Equation 2.9 as

$$\mathrm{Tri}(a_1, b_1, c_1) + \mathrm{Tri}(a_2, b_2, c_2) = \mathrm{Tri}(a_1 + a_2, b_1 + b_2, c_1 + c_2). \tag{2.10}$$

The summation of two triangular fuzzy numbers will always result in a new triangular fuzzy number. However, because maximization is a nonlinear operator, the maximum of two triangular fuzzy numbers may not be triangular (see Figure 2.6). To keep the practical requirements of the CMM framework simple, we seek to maintain a consistent representation for all fuzzy numbers. Therefore, we define an approximate maximization operator that gives a triangular fuzzy number,

$$\begin{aligned} &\max'\big(\mathrm{Tri}(a_1, b_1, c_1), \mathrm{Tri}(a_2, b_2, c_2)\big) \\ &\qquad = \mathrm{Tri}(\max(a_1, a_2), \max(b_1, b_2), \max(c_1, c_2)). \end{aligned} \tag{2.11}$$

This approach maintains the true definition at the endpoints and peak of the fuzzy number, but may produce different values at the intermediate points.

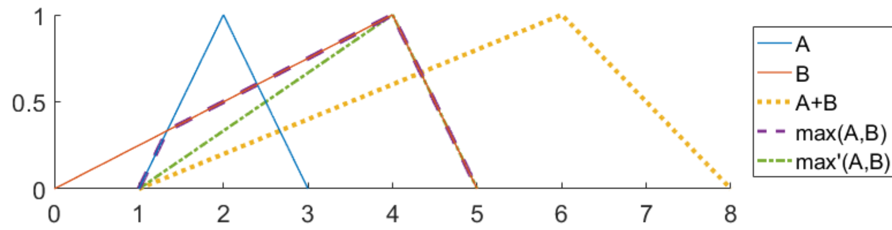


Figure 2.6 The summation of two triangular fuzzy numbers $A = \mathrm{Tri}(1, 2, 3)$ and $B = \mathrm{Tri}(0, 4, 5)$ is shown as $\mathrm{Tri}(1, 6, 8)$. The true maximum of A and B is not a triangular fuzzy number, but can be approximated as $\mathrm{Tri}(1, 4, 5)$.

In a least-cost path problem, the goal is to find a solution that minimizes some set of objectives. By representing the value of a solution as a fuzzy number, we can capture some of the uncertainty in a solution's true value. However, this uncertainty makes it difficult to assess whether one solution is better than another. While there is no universal definition for the ordering of fuzzy numbers that proves satisfactory for all cases (see for instance (Wang and Kerre 2001a; Wang and Kerre 2001b)), we adopt the following intuitive definitions.

Definition 2.1. Let $A_1 = \mathrm{Tri}(a_1, b_1, c_1)$ and $A_2 = \mathrm{Tri}(a_2, b_2, c_2)$ be two triangular fuzzy numbers. We say that A_1 is less than or equal to A_2 ($A_1 \leq A_2$) if and only if ($a_1 \leq a_2$ and $b_1 \leq b_2$ and $c_1 \leq c_2$).

Definition 2.2. Let $A_1 = \mathrm{Tri}(a_1, b_1, c_1)$ and $A_2 = \mathrm{Tri}(a_2, b_2, c_2)$ be two triangular fuzzy numbers. We say that A_1 is strictly less than A_2 ($A_1 < A_2$) if and only if $A_1 \leq A_2$ and ($a_1 < a_2$ or $b_1 < b_2$ or $c_1 < c_2$).

There may be many solutions for a given problem with no single solution that is less than all others. If $A_1 \not\prec A_2$ and $A_2 \not\prec A_1$, then it is not clear which of the two fuzzy numbers should be preferred (assuming $A_1 \neq A_2$). When a decision-maker is required to choose one of these, we can employ a weighted centroid defuzzification scheme to produce a crisp value for each solution that can be ranked directly. The centroid of a fuzzy number A is defined as

$$\bar{x} = \frac{\int x\mu_A(x)\,dx}{\int \mu_A(x)\,dx}. \tag{2.12}$$

For a triangular fuzzy number $\mathrm{Tri}(a, b, c)$, this evaluates to

$$\bar{x} = \frac{1}{3}(a + b + c). \tag{2.13}$$

The weighted centroid is defined by a control parameter $\xi \in [0, 1]$ that specifies the optimism/pessimism of the decision-maker. A value of $\xi = 0$ indicates extreme optimism, in which the fuzzy number is defuzzified to the smallest possible value, a. A value of $\xi = 1$ indicates extreme pessimism, where defuzzification results in the largest possible value, c. A value of $\xi = 0.5$ gives a balanced approach using the centroid, $\bar{x}$. The crisp weighted centroid value is linearly interpolated between these points as

$$C(A|\xi) = \begin{cases} a + 2\xi(\bar{x} - a), & \xi \leq 0.5 \\ \bar{x} + 2(\xi - 0.5)(c - \bar{x}), & \xi > 0.5\,. \end{cases} \tag{2.14}$$

Using a constant value for ξ, a decision-maker can defuzzify multiple fuzzy numbers using the weighted centroid approach and compare the resulting crisp values. If two fuzzy numbers result in the same crisp value when defuzzified, they are considered equivalent.

2.4.3 The Multiobjective Fuzzy Least-Cost Path Problem

The fuzzy shortest path problem (FSPP) was first analyzed by Dubois and Prade (Dubois and Prade 1980), who extended the classic Floyd-Warshall and Bellman-Ford algorithms for graphs with a single fuzzy weight assigned to each edge. A drawback of this early approach, however, was that a fuzzy cost could be obtained without an associated path to go with it. Many papers have since been written on the topic with improved algorithms for specific variations of the FSPP (e.g. (Klein 1991; Okada and Soper 2000; Cornelis, De Kesel, and Kerre 2004; Moazeni 2006; Hernandes et al. 2007)) with improved algorithms for specific variations of the FSPP. Typically, the problem is treated as a single objective optimization, although due to its fuzzy nature, there may be multiple non-dominated solutions. A defuzzification method is usually required to provide some recommendation to a decision-maker.

The multiobjective shortest path problem (MO-SPP) likewise has received considerable attention in the optimization literature (e.g. (Martins 1984; Loui 1983; Guerriero and Musmanno 2001; Tarapata 2007)). One of the common applications of the MO-SPP is in designing and using transportation networks, where travel time, distance, and other criteria may dictate which routes are considered optimal. These objectives are typically in conflict such that there is no single solution that outperforms all others. In this case, multiobjective decision-making techniques should be used to help the decision-maker choose a solution.

We define a general fuzzy weighted graph as a graph G that has a weight vector of fuzzy numbers assigned to each edge. For each edge $e \in E(G)$, we define a feature vector

$\boldsymbol{F}(e) = \big(F_1(e), \dots, F_m(e)\big)$, where each $F_i(e)$ is a fuzzy number that represents a feature attribute of that edge and m is the total number of features. Features are defined as components of a multiobjective cost function that are intended to be minimized. For instance, a vector might have one feature that represents distance and another that represents slope or travel time. In the CMM framework, each feature is assumed to be non-negative with zero being the minimum possible value. By using a vector of fuzzy numbers to represent edge weights, a fuzzy weighted graph can model different degrees of uncertainty for each component of the multiobjective cost function. When there is no uncertainty, the fuzzy numbers are reduced to crisp values.

A path $p = (e_1, \dots, e_n)$ in a fuzzy weighted graph is a sequence of n edges, each with an associated weight vector given as $\boldsymbol{F}(e_i)$. We compute an aggregated cost vector $\boldsymbol{A}(p) = \big(A_1(p), \dots, A_m(p)\big)$ for the path by either summing or taking the maximum value of the feature components of each edge. Let $\boldsymbol{\gamma} = (\gamma_1, \dots, \gamma_m)$ be an indicator vector where $\gamma_i = 0$ if feature i should be aggregated by summation and $\gamma_i = 1$ if feature i should be aggregated using maximization. For features where the decision-maker considers the total feature value ($\gamma_i = 0$), the aggregated value of feature i is

$$A_i(p) = \sum_{j=1}^{n} F_i\big(e_j\big). \tag{2.15}$$

For features where the decision-maker considers the maximum feature value ($\gamma_i = 1$), the aggregated value of feature i is

$$A_i(p) = \max_{j=1,\dots,n}{}' F_i\big(e_j\big). \tag{2.16}$$

Note that the aggregation method may be different for each feature. For instance, a feature measuring the total distance traveled would be aggregated using summation, whereas a feature measuring the steepest slope segment along a path would be aggregated using maximization.

Given a fuzzy weighted graph G with a starting vertex $s \in V(G)$ and an ending vertex $t \in V(G)$ where $s \neq t$, the multiobjective fuzzy least-cost path problem (MO-FLCPP) is defined as finding a path $p \in P(s,t)$ that minimizes the aggregated cost vector $\boldsymbol{A}(p)$. When the summation operator is used for aggregation, this is called the shortest path problem. When the max operator is used, it may be called the minimax path problem. We use the term least-cost path to refer to the general case that may have mixed aggregation methods. The MO-FLCPP may not have a single solution that minimizes each cost component $A_i(p)$ simultaneously for $i = 1, \dots, m$. Multiobjective optimization techniques should therefore be used to help the decision-maker choose a solution.

2.5 Multiobjective Optimization

Many real-world problems involve several criteria that influence the decision-making process. In these problems, a decision-maker must optimize multiple objectives simultaneously. Typically, the objectives conflict in some nontrivial way, forcing the decision-maker to make some tradeoff between various possible solutions. Multiobjective optimization is the study of decision problems with more than one objective. This section presents an overview of the multiobjective problem setting and defines several methods that allow a decision-maker to select an optimal solution.

2.5.1 Multiobjective Optimization Problem Definition

Formally, a multiobjective optimization problem (MOP) is defined as

$$\begin{array}{rl} \text{minimize} & \mathbf{f}(\mathbf{x}) = \big(f_1(\mathbf{x}), \dots, f_k(\mathbf{x})\big) \\ \text{subject to} & \mathbf{x} \in \Omega, \end{array} \tag{2.17}$$

where we have $k\ (\geq 2)$ real-valued objective functions $f_i\colon \mathbb{R}^n \to \mathbb{R}$ that are to be minimized (Miettinen 1999). A decision vector $\mathbf{x} = (x_1, \dots, x_n)$ represents a potential solution to the MOP and $\Omega \subseteq \mathbb{R}^n$ is the feasible region defined by the problem constraints. We sometimes use the terms *decision vector* and *solution* interchangeably. A decision variable x_i may represent some characteristic attribute of the solution, or some component of a complete solution. For example, a complete solution to the multiobjective path-planning problem is some path through an environment space and the decision variables represent the various path components[1].

We assume that no single solution $\mathbf{x}$ minimizes all objective functions simultaneously, otherwise there would be no conflict between the objectives and the problem could be solved using traditional single-objective methods. In the usual case with multiple conflicting objectives, we cannot determine a single optimal solution and are required to examine the tradeoffs between solutions in the objective space, $\mathbb{R}^k$. The subset of the objective space that forms the image of the feasible region Ω is called the feasible objective region and is denoted as $\Lambda = \mathbf{f}(\Omega)$. Elements of Λ are called objective vectors and are denoted by $\mathbf{f}(\mathbf{x})$ or $\mathbf{z} = (z_1, \dots, z_k)$, where $z_i = f_i(\mathbf{x})$ for all $i = 1, \dots, k$. Each objective value z_i represents some quantifiable feature of the decision vector that is to be

[1] The details of the decision vector representation for paths will be discussed further in Chapter 6.

minimized. Note that a feature that is to be maximized can be represented in this framework by defining $f_i^{\min}(\mathbf{x}) = -f_i^{\max}(\mathbf{x})$. Figure 2.7 shows a graphical representation of the mapping of a solution from decision space to objective space in a MOP.

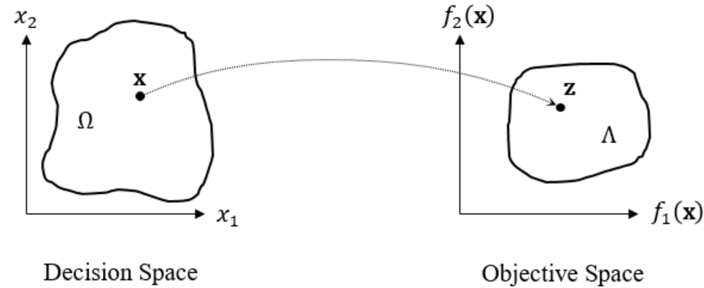


Figure 2.7 Mapping from decision space to objective space in a multiobjective optimization problem.

2.5.2 Pareto Optimality

Consider two solutions, $\mathbf{x}^1, \mathbf{x}^2 \in \Omega$ and their corresponding objective vectors, $\mathbf{z}^1 = \mathbf{f}(\mathbf{x}^1)$ and $\mathbf{z}^2 = \mathbf{f}(\mathbf{x}^2)$. If $z_i^1 < z_i^2$ for some objective i, then $\mathbf{z}^1$ is the preferred objective vector (and $\mathbf{x}^1$ is the preferred solution) based solely on objective i. If $z_i^1 \leq z_i^2$ for all $i = 1, \ldots, k$ and $z_j^1 < z_j^2$ for at least one index j, then $\mathbf{z}^1$ is said to dominate $\mathbf{z}^2$ because it is at least as good as $\mathbf{z}^2$ in all objectives and it is better than $\mathbf{z}^2$ in at least one objective. Similarly, we say that $\mathbf{x}^1$ dominates $\mathbf{x}^2$ if $\mathbf{z}^1$ dominates $\mathbf{z}^2$. A solution that is not dominated by any other solution in the feasible region is called Pareto optimal, named after the French-Italian economist and sociologist Vilfredo Pareto, who pioneered the notion of preference ordering in terms of ordinal utility rather than cardinal utility (Aspers 2001).

Definition 2.3. An objective vector $\mathbf{z}^* \in \Lambda$ is Pareto optimal if there does not exist another objective vector $\mathbf{z} \in \Lambda$ such that $z_i \leq z_i^*$ for all $i = 1, \dots, k$ and $z_j < z_j^*$ for at least one index j.

Definition 2.4. A decision vector $\mathbf{x}^* \in \Omega$ is Pareto optimal if there does not exist another decision vector $\mathbf{x} \in \Omega$ such that $f_i(\mathbf{x}) \leq f_i(\mathbf{x}^*)$ for all $i = 1, \dots, k$ and $f_j(\mathbf{x}) < f_j(\mathbf{x}^*)$ for at least one index j.

The set of all Pareto optimal decision vectors forms the Pareto optimal set PS, and the set of all Pareto optimal objective vectors forms the Pareto front PF. An illustration of the Pareto optimal set and corresponding Pareto front for a two-objective problem is shown in Figure 2.8.

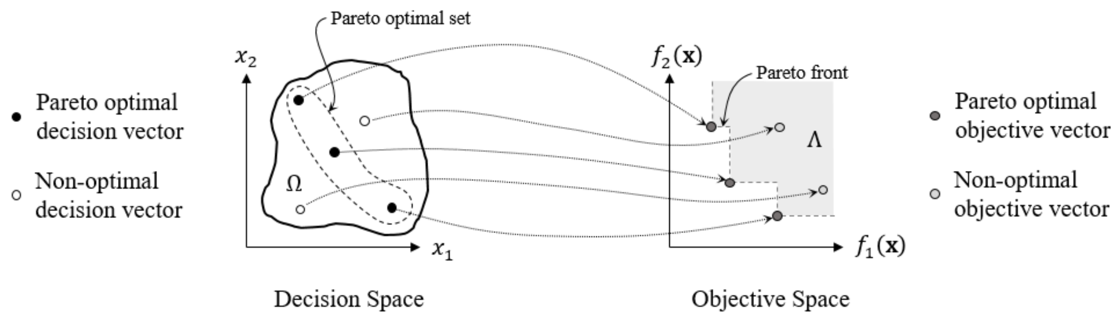


Figure 2.8 The mapping of solution vectors from decision space to objective space shows which solutions belong to the Pareto optimal set in decision space and the Pareto front in objective space.

2.5.3 Scalarization

We are not usually satisfied by simply determining the Pareto optimal set of solutions (or an approximation thereof); most problems require the decision-maker to actually decide upon a single solution. In general, multiobjective optimization problems are solved using the method of scalarization, in which a scalarizing function $g: \Lambda \rightarrow \mathbb{R}_{\geq 0}$ is defined over the multidimensional objective space that maps any given objective vector into a single non-negative real value. The decision-maker then chooses the solution from the feasible region that minimizes the scalarized objective value. We define the resulting single-objective optimization problem as

$$\begin{aligned} \text{minimize} \quad & g\big(\mathbf{f}(\mathbf{x})\big) \\ \text{subject to} \quad & \mathbf{x} \in \Omega. \end{aligned} \tag{2.18}$$

Once the problem has been scalarized into a single objective, we can use traditional optimization techniques to find a solution.

Before scalarizing the objective space, it is usually advisable to normalize the output range of each objective function into a common range. This helps to ensure that each objective is treated equally and that the natural scale of the objective values does not bias the decision toward certain objectives. One common strategy is to normalize the range of the Pareto front into the unit hypercube. This can be accomplished by defining the ideal and nadir objective vectors for a given multiobjective problem. The ideal objective vector is defined as $\mathbf{z}^{\star} = (z_1^{\star}, \ldots, z_k^{\star})$, where $z_i^{\star} = \min_{\mathbf{x} \in \Omega} f_i(\mathbf{x})$ for $i = 1, \ldots, k$. This point represents the best possible objective value in each dimension, although it is almost certainly outside of the feasible objective region. In contrast to the ideal objective vector, the nadir objective vector $\mathbf{z}^{\text{nad}}$ represents the upper boundary of the Pareto front and may

or may not lie within the feasible objective region. While the ideal objective vector is straightforward to define, the nadir objective vector is often unknown until the problem has been solved. For a given approximation of the Pareto optimal set PS', we define the nadir objective vector as $\mathbf{z}^{\text{nad}} = \left(z_1^{\text{nad}}, \dots, z_k^{\text{nad}}\right)$, where $z_i^{\text{nad}} = \max_{\mathbf{x} \in PS'} f_i(\mathbf{x})$ for $i = 1, \dots, k$. Figure 2.9 shows two examples of the Pareto front and the associated ideal and nadir objective vectors. Note that the ideal and nadir objective vectors form a bounding box around the complete Pareto front, which may be disjoint.

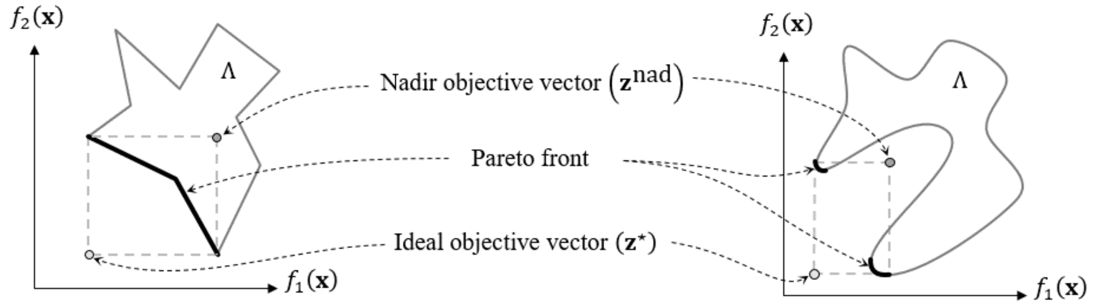


Figure 2.9 Examples of the range of the Pareto front. The ideal objective vector represents the minimum attainable value of each of objective and the nadir objective vector represents the upper boundary of the Pareto front.

Given the range of the Pareto front defined by $\mathbf{z}^{\star}$ and $\mathbf{z}^{\text{nad}}$, we normalize the objective function values into a common range $[0, 1]$ by defining

$$z_i' = f_i'(\mathbf{x}) = \frac{f_i(\mathbf{x}) - z_i^{\star}}{z_i^{\text{nad}} - z_i^{\star}} \quad \text{for } i = 1, \dots, k. \tag{2.19}$$

The normalized objective vectors are then defined as $\mathbf{z}' = \mathbf{f}'(\mathbf{x}) = (z_1', \dots, z_k')$. These vectors are used in place of the original objective vectors for the scalarization computation, although the decision-maker may still prefer to be presented with the original units when comparing alternatives.

2.5.4 Method of the Global Criterion

If the decision-maker considers all criteria to be equally important, we can use the method of the global criterion, sometimes also called compromise programming (Zeleny 1973). In this method, the decision-maker picks the solution from the Pareto optimal set that minimizes the distance to some ideal reference point. We use the L_p-metric to measure the distance from the ideal objective vector and define the scalarization function as

$$g_p^{\mathrm{gc}}(\mathbf{z}') = \left(\sum_{i=1}^{k} (z_i')^p\right)^{\frac{1}{p}}, \tag{2.20}$$

where each z_i' represents a normalized objective value from Equation 2.19 and $p > 0$. Because we only need to find the solution that minimizes the scalarized value, the exponent $\frac{1}{p}$ can be dropped without affecting the outcome. Common values of p are 1, 2, and ∞. The L_∞-metric is also called the Tchebycheff[1] metric, and the equivalent scalarization function can be expressed as

$$g_\infty^{\mathrm{gc}}(\mathbf{z}') = \max_{i=1,\ldots,k} z_i'. \tag{2.21}$$

Figure 2.10 shows the difference between the L_1-, L_2-, and L_∞-metrics. Each metric results in a different interpretation of the objective space and correspondingly selects a different solution from the Pareto front.

[1] An alternate spelling is the Chebychev metric.

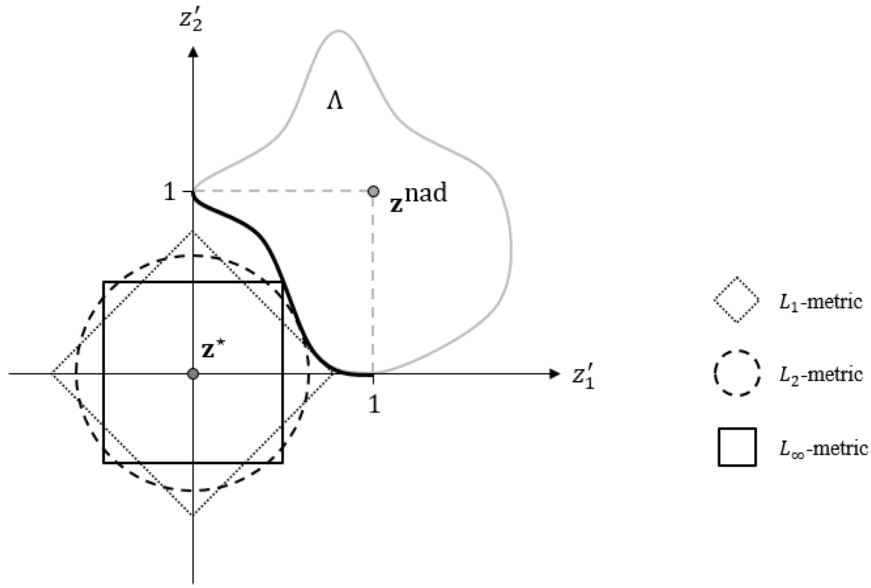


Figure 2.10 Different metrics applied in the global criterion method. The decision-maker chooses the solution on the Pareto front that minimizes the distance to the ideal objective vector $\mathbf{z}^{\star}$. The contours of each L_p-metric are shown at the minimum value that intersects the Pareto front.

2.5.5 *Method of Weighted Metrics*

A decision-maker will usually wish to express some preference for certain objectives over others. The method of the global criterion can be extended to include a weight term for each objective indicating its relative importance. Doing so effectively scales the objective space so that different points on the Pareto front are measured to be closer to the ideal objective vector. Assume that the decision-maker has defined a weight vector $\boldsymbol{\lambda} = (\lambda_1, \dots, \lambda_k)$ such that $\lambda_i \geq 0$ for all $i = 1, \dots, k$ and $\sum_{i=1}^{k} \lambda_i = 1$. We define the weighted L_p scalarization function as

$$g_p^{\text{wm}}(\mathbf{z}'|\boldsymbol{\lambda}) = \left(\sum_{i=1}^{k} \lambda_i (z_i')^p \right)^{\frac{1}{p}}, \tag{2.22}$$

for $p > 0$. If $p = 1$, the scalarization function is equivalent to a weighted sum of the objective values,

$$g^{\mathrm{ws}}(\mathbf{z}'|\boldsymbol{\lambda}) = \sum_{i=1}^{k} \lambda_i z_i'. \tag{2.23}$$

The weighted sum is one of the earliest and most commonly used scalarization methods for multiobjective problems as it maintains the linearity of the problem if the objective functions are linear (Gass and Saaty 1955; Zadeh 1963). If $p = 2$, the scalarization function uses Euclidean distance and becomes the method of least squares,

$$g^{ls}(\mathbf{z}'|\boldsymbol{\lambda}) = \sqrt{\sum_{i=1}^{k} \lambda_i (z_i')^2}. \tag{2.24}$$

When $p = \infty$, the scalarization function is equivalent to the Tchebycheff method,

$$g^{\mathrm{te}}(\mathbf{z}'|\boldsymbol{\lambda}) = \max_{i=1,\dots,k} \lambda_i z_i'. \tag{2.25}$$

The Tchebycheff scalarization function favors solutions toward the middle of the Pareto front, whereas the weighted sum approach tends to result in solutions near the edges. In fact, if the shape of the Pareto front is concave, the weighted sum approach can only return solutions at the extrema points of the Pareto front, whereas the Tchebycheff method can find any Pareto optimal solution with some setting of the weight vector (Bowman 1976). To illustrate this, consider the examples in Figure 2.11. The top row shows a convex Pareto front that has been scaled with three different weight vectors. The solutions closest to the origin are indicated for both the weighted sum and Tchebycheff approaches. Note that the Tchebycheff solution is closer to the midpoint of the Pareto front than the weighted sum solution, which moves closer to the endpoint with the larger weight value. The bottom

row shows another example with a concave Pareto front. In this case, the weighted sum approach cannot select a solution within the concave region because the endpoints are always closer to the origin regardless of the scaling applied by the weight vector. In contrast, the Tchebycheff approach can return solutions within the concave region because of the way the distance measurement is computed.

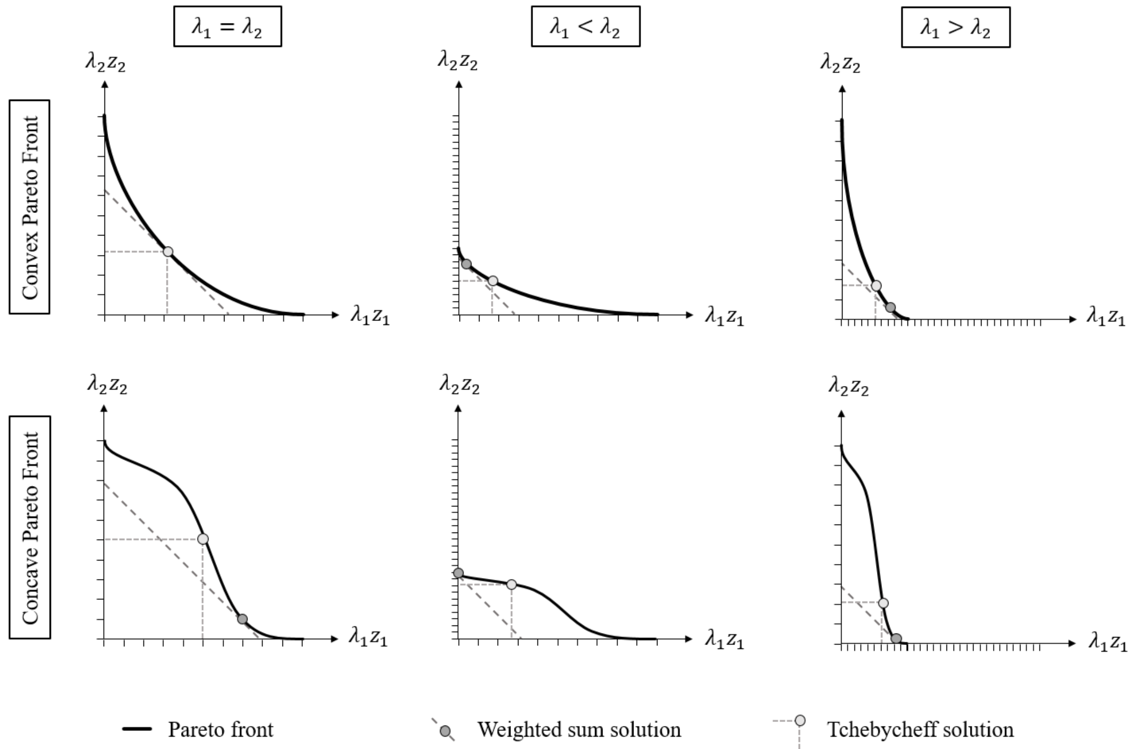


Figure 2.11 Comparison of the weighted sum and Tchebycheff scalarization approaches. The weighted sum can only return extrema points when the Pareto front is concave, whereas the Tchebycheff approach can find any Pareto optimal solution.

2.5.6 Ordered Weighted Average Approach

The final method we consider for scalarizing an objective vector is the ordered weighted average (OWA) operator (Yager 1988). The OWA operator is a flexible aggregation function that lies somewhere between the min and max operators. It allows for

a more natural interpretation of the scalarization function by defining the degree to which each of the criteria need to be satisfied. At one extreme, the max operator acts as the logical "and," requiring all criteria to meet some minimum degree of satisfaction. At the other extreme, the min operator acts as the logical "or," requiring only one criteria to be satisfied. The mean operator lies in between these two extremes and optimizes the average satisfaction of all criteria.

To apply the OWA operator to a normalized objective vector $\mathbf{z}' = (z'_1, \dots, z'_k)$, we first apply the criteria preference weights $\boldsymbol{\lambda} = (\lambda_1, \dots, \lambda_k)$ and define a scaled objective vector $\mathbf{a} = (a_1, \dots, a_k)$ where $a_i = \lambda_i z'_i$ for all $i = 1, \dots, k$. The elements of $\mathbf{a}$ are then sorted in descending order $\left(a_{(1)}, \dots, a_{(k)}\right)$ where $a_{(i)}$ is the ith largest of the a_i values. The OWA operator is parameterized by an additional weight vector $\mathbf{w} = (w_1, \dots, w_k)$ where $w_i \geq 0$ for all $i = 1, \dots, k$ and $\sum_{i=1}^{k} w_i = 1$. The OWA scalarization function is defined as

$$g^{\text{OWA}}(\mathbf{a}|\mathbf{w}) = \sum_{i=1}^{k} w_i a_{(i)}. \qquad (2.26)$$

By changing the OWA weight vector, the OWA operator can be made to represent different aggregation functions. The following are some notable operators.

- Average: $w_i = \frac{1}{k}$ for all $i = 1, \dots, k$. This method is equivalent to the weighted sum scalarization approach. All criteria are weighted equally (after scaling).
- Max: $w_1 = 1$ and $w_i = 0$ for all $i \neq 1$. This method is equivalent to the Tchebycheff scalarization approach and acts as the logical "and" operator. All criteria must be satisfied when using this method since the scalarized value is

equivalent to the objective value of the least satisfied criteria (the largest of the a_i values).

- Min: $w_k = 1$ and $w_i = 0$ for all $i \neq k$. This method acts as the logical "or" operator and requires only a single criterion to be satisfied. The scalarized value is equal to the most satisfied criteria (the smallest of the a_i values). In practice, this method is rarely used for multiobjective optimization because it does not consider the satisfaction of any other criteria other than the most satisfied criteria.

2.6 Multiobjective Evolutionary Algorithms

Evolutionary algorithms are well suited to solve multiobjective optimization problems. These algorithms create a population of potential solutions and use genetic operators such as selection, crossover, and mutation to iteratively improve the best individuals until a suitable solution is found. In a multiobjective problem, there may not be a single optimal solution, but rather a set of Pareto optimal solutions. The goal of a multiobjective evolutionary algorithm (MOEA) is to return a set of solutions that closely approximates the true Pareto optimal set (Fonseca and Fleming 1995; Zitzler, Laumanns, and Bleuler 2004; Zhou et al. 2011). There are several strategies that can be employed to modify an EA for a multiobjective problem. Typically, the algorithm needs to define the fitness of an individual solution and specify how the selection, crossover, and mutation operators should work. Whereas a single objective EA returns only a single solution, an MOEA needs to maintain population diversity so that the solutions are well distributed over the entire Pareto front. Additionally, there should be enough solutions to provide adequate coverage of the entire Pareto front. This can become a challenging issue when

there are many objectives. Two of the most influential MOEAs are the nondominated sorting genetic algorithm (NSGA-II) (Deb et al. 2002) and the multiobjective evolutionary algorithm based on decomposition (MOEA/D) (Qingfu Zhang and Hui Li 2007).

NSGA-II uses a fast non-dominated sorting approach to partition the population of N solutions into multiple nondominated fronts such that no solution dominates another solution in the same front, and each solution in front F_i is dominated by at least one solution in front F_{i-1} for $i > 1$. The solutions within each front are then sorted based on crowding distance, so that solutions that are more separated receive a lower (better) rank. Each generation, a new population is generated using binary tournament selection, and the combined parents and offspring are sorted using the nondominated sorting approach. The top N individuals survive to the next generation.

The NSGA-II algorithm achieves very good performance on problems with a small number of objectives, but suffers from the curse of dimensionality in high-dimensional spaces. Most solutions are nondominated in many-objective optimization problems, which weakens the selective pressure of the nondominated sorting approach. Furthermore, an exponentially greater number of solutions are required to model a high-dimensional Pareto front. These issues have given rise to a class of many-objective evolutionary algorithms (MaOEAs) (Ishibuchi, Tsukamoto, and Nojima 2008; Li et al. 2015). One of the key difficulties of MaOEAs is the visualization of the population, which can be used to evaluate methods and to observe the evolutionary process (He and Yen 2016). Several MaOEA algorithms have been proposed, including NSGA-III (Jain and Deb 2014), ϵ-MOEA (Deb, Mohan, and Mishra 2005), GrEA (Yang et al. 2013), HypE (Bader and Zitzler 2011), and

MOEA/D (Qingfu Zhang and Hui Li 2007). Of these, the decomposition strategy of MOEA/D has proven to be particularly adaptable to many problem domains and algorithmic variants (Trivedi et al. 2016).

The general MOEA/D approach is outlined in Algorithm 2.3. The method requires the definition of N evenly spread m-dimensional weight vectors $\boldsymbol{\lambda}^1, \dots, \boldsymbol{\lambda}^N$, representing N scalarized single-objective problems. The method maintains a population of N solutions $x^1, \dots, x^N$, where x^i is the current solution to subproblem i. A solution x^i is evaluated as $F(x^i) = \left(f_1(x^i), \dots, f_m(x^i)\right)$, where $f_j(x^i)$ represents the value of x^i against objective j. A reference point $\boldsymbol{z} = (z_1, \dots, z_k)$ maintains the ideal objective vector discovered so far and a scalarization function $g(x|\boldsymbol{\lambda}^i, \boldsymbol{z})$ provides a measure of how well solution x solves subproblem i. The algorithm maintains an external population EP that contains the nondominated solutions found during the search.

The first step of the algorithm creates initial solutions to each of the subproblems and computes the neighborhood of each weight vector. In the second step, each weight vector is evaluated and a new solution is created using crossover and mutation operators on the solutions in the neighborhood. If necessary, a problem-specific repair or heuristic can be applied to improve the solution. The new solution is then compared with the current solutions of the neighboring subproblems and it replaces the old solution if the new one is better. The objective values of the solutions are compared and used to update the external population of nondominated solutions, which is returned at the end of the search.

The decomposition approach allows MOEA/D to be applied to problems where a single-objective optimization strategy is readily available. For instance, the multiobjective

shortest path problem (MO-SPP) can define several weight vectors to scalarize the problem as a set of single-objective subproblems, and then use a shortest path algorithm such as Dijkstra's algorithm to find solutions to these subproblems. The MOEA/D algorithm can be used to find the Pareto optimal set of solutions to the MO-SPP before requiring a decision-maker to give any specific preferences.

Algorithm 2.3 MOEA/D

Input:

- MOP
- a stopping criterion
- N: the number of subproblems considered
- $\boldsymbol{\lambda}^1, \ldots, \boldsymbol{\lambda}^N$: m-dimensional objective weight vectors
- T: the size of each weight vector neighborhood
- g: a scalarization function

Output: EP

Step 1) Initialization:

Step 1.1) Set $EP = \emptyset$

Step 1.2) Compute $B(i) = \{i_1, \ldots, i_T\}$ as the T closest weight vectors to each weight vector $\boldsymbol{\lambda}^i$ for $i = 1, \ldots, N$

Step 1.3) Generate an initial population $x^1, \ldots, x^N$ and set $FV^i = F(x^i)$

Step 1.4) Initialize the reference point $\boldsymbol{z} = (z_1, \ldots, z_k)$

Step 2) Update:

For $i = 1, \ldots, N$, do

Step 2.1) Reproduction: Randomly select two indices k and l from $B(i)$ and generate a new solution y from x^k and x^l

Step 2.2) Improvement: Use a problem-specific heuristic on y to produce y'

Step 2.3) Update $\boldsymbol{z}$: Set $z_j = \min\{z_j, f_j(y')\}$ for $j = 1, \ldots, m$

Step 2.4) Update neighbors: For each index $j \in B(i)$, if $g(y'|\boldsymbol{\lambda}^j, \boldsymbol{z}) \leq g(x^j|\boldsymbol{\lambda}^j, \boldsymbol{z})$, then set $x^j = y'$ and $FV^j = F(y')$

Step 2.5) Update $\boldsymbol{EP}$:

Step 2.5.1) Remove all vectors from EP that are dominated by $F(y')$

Step 2.5.2) Add $F(y')$ to EP if no vectors in EP dominate $F(y')$.

Step 3) Stopping criteria: If the stopping criteria has been satisfied, then stop and output EP. Otherwise repeat Step 2.

2.7 Intelligent Agents

A software agent that acts with purposeful behavior in a problem domain may be described as intelligent to some degree. As environmental psychologists developed models that could be used to explain human and animal behavior in real world environments, computer scientists worked to implement analogous models that could be used in simulated environments. The field of artificial intelligence grew around the concept of developing intelligent agents that behave autonomously to maximize some notion of success (Russell and Norvig 2009). There are several classes of intelligent agents, ranging from the simple reflex agents used in multi-agent models, to learning agents that can operate in unknown environments. The general agent model is very similar to the cognitive model used in Figure 2.1**Error! Reference source not found.**. Agents have a set of actuators that define the actions they can perform and a set of sensors that receive percepts from the environment. The agent's actions are defined by the agent function, which maps percepts into actions and can be realized in various ways. For example, the agent function might use a lookup table or fuzzy rule base to decide which action to take. More complex agents can maintain some notion of the current state, and may be implemented as a finite state machine. The most advanced agents can adapt to unforeseen events and can learn the utilities of their actions.

In the physical world, intelligent agents have been designed to control mobile robots using a variety of cognitive models (Luke et al. 2005; Eliashberg 2002; Busch et al. 2007; Blisard and Skubic 2005; Phillips and Noelle 2005; Skubic et al. 2004). In some instances, the environment is not known completely and the robot must construct a map of its surroundings as it explores (Thrun 2002). The simultaneous localization and mapping

(SLAM) problem can be defined as determining the robot's location x_t and the environment map m_t from a sequence of observations $o_{1:t}$. SLAM has been typically treated as a probabilistic problem that uses Bayes rule to maximize the posterior distribution $P(m_t, x_t | o_{1:t})$ given sequentially defined updates for the location $P(x_t | o_{1:t}, m_t)$ and the map $P(m_t | x_t, o_{1:t})$.

2.8 The Traveling Salesman Problem

The traveling salesman problem (TSP) is a combinatorial optimization problem that has received wide attention in the fields of operations research and theoretical computer science as a benchmark for exploring issues of computational complexity (Applegate et al. 2007). In its canonical form, the problem is to find the order in which an agent should visit a set of cities separated by known distances so as to minimize the total distance traveled. Several variations of the TSP have been proposed (Gutin and Punnen 2007) including the probabilistic TSP (Jaillet 1985), physical TSP (Perez, Rohlfshagen, and Lucas 2012), partially observable TSP (Buck and Keller 2016), and the traveling purchaser problem (TPP) (Boctor, Laporte, and Renaud 2003; Riera-Ledesma and Salazar-González 2005). The TPP can be considered a generalization of the TSP, where an agent must visit a set of known market locations with various prices for goods in an order that allows it to purchase a given list of items at the lowest price while also accounting for the cost of travel. This form of the problem can be parameterized in many ways to create simpler problem types such as the resource collecting TPP. In this variation, each market offers a single type of resource and the agent needs to collect a set number of each resource type by visiting the appropriate markets in the shortest distance possible. An even simpler variation that can be

expressed in this form is the k-TSP in which the agent only needs to visit k of the market locations.

One application of the TSP and the TPP is to create problems that can be solved by agent-based systems. Computer simulations have been used in many domains to study the complex problems that often arise as the result of physical processes or agent interaction. The latter case is the subject of study in agent-based modeling, which seeks to explain the collective behavior of a population of autonomous agents that model natural systems by obeying simple rules (Bonabeau 2002). By using agents to represent individual decision-makers, complex systems can be created that show emergent behavior (Holland and Miller 1991). In the context of human geography, agent-based models have been used to study evacuation scenarios in disaster situations (Keller, Popescu, and Gibeson 2012), and utilized concepts such as bounded rationality to guide agent behavior (Popescu and Keller 2012) and rumor spreading models based on social networks (Zare et al. 2012). A related field is that of multi-agent systems, which focuses on using an interacting group of intelligent agents to model problems that are too complex for an individual agent (Niazi and Hussain 2011). In the area of computational intelligence, agent-based models have been used to solve optimization problems using techniques such as ant colony optimization (Dorigo, Maniezzo, and Colorni 1996) and particle swarms (Kennedy and Eberhart 1995).

3 CREATING GRID WORLD ENVIRONMENTS

In this chapter, we introduce the grid world environments used in the CMM framework. The environments contain various attributes and are constructed using several different methods. These environments provide the problem structure for studying various aspects of multicriteria and partially observable decision-making in later chapters.

3.1 Grid Worlds

The CMM framework presented in this work uses grid worlds exclusively as the problem domain. A gridded environment provides structure and regularity that simplifies many of the practical issues of representing a physical environment. In a grid world problem, the environment space $\mathcal{E}$ is discretized into a regular grid of cells so that cell $c_{(i,j)} \in \mathcal{E}$ is the cell in row i and column j. The world contains a single decision-making agent located in cell $c_{(a_i,a_j)}$. The agent can move to adjacent cells $c_{(a_{i-1},a_j)}$, $c_{(a_{i+1},a_j)}$, $c_{(a_i,a_{j-1})}$, or $c_{(a_i,a_{j+1})}$ by travelling up, down, left, or right respectively, so long as the adjacent cell is traversable. We restrict the agent's movement to these four directions to ensure that each step is of equal length, which simplifies the resulting analysis. Each cell has a set of attributes that describe the local state of the environment. In the CMM framework, we define the following attributes for each grid cell $c \in \mathcal{E}$:

- $\textsc{Open}(c) \in \{0, 1\}$ indicates if the cell is traversable. A value of 0 indicates that the cell contains a wall, whereas 1 indicates that the cell is open and the agent can move into it.

- $\textsc{Terrain}(c) \in \mathcal{T}$ indicates the type of terrain in the cell, where $\mathcal{T}$ is the set of all terrain types[1]. In simple environments, there may only be a single terrain type (e.g. $\mathcal{T} = \{open_space\}$). However, we can model more complex environments by including additional types of terrain such as *snow*, *rock*, *meadow*, *forest*, *water*, etc.
- $\textsc{Elevation}(c) \in [0,1]$ is a continuous-valued attribute that indicates the height of the grid cell. In our work, the domain is restricted to the unit interval.
- $\textsc{Resources}(c) \in \mathcal{R} \cup \emptyset$ indicates which resource type, if any, is present in the cell, where $\mathcal{R}$ is the set of all resource types. It may be preferable to associate a single resource type with each type of terrain, although this is not a strict requirement. Note that in our work we assume that each grid cell can have a maximum of one resource type.

In the following sections, these attributes are generated independently as a set of $n \times m$ matrices, where each grid cell $c_{(i,j)}$ references the corresponding matrix index $[i,j]$ and $1 \leq i \leq n$ and $1 \leq j \leq m$. In practice, these matrices are used to look up the attribute values of any grid cell in the environment.

[1] We could implicitly include cells that contain walls as an additional type of terrain, however we separate them in our notation to better indicate where the agent can travel and to aid in the computation of visibility.

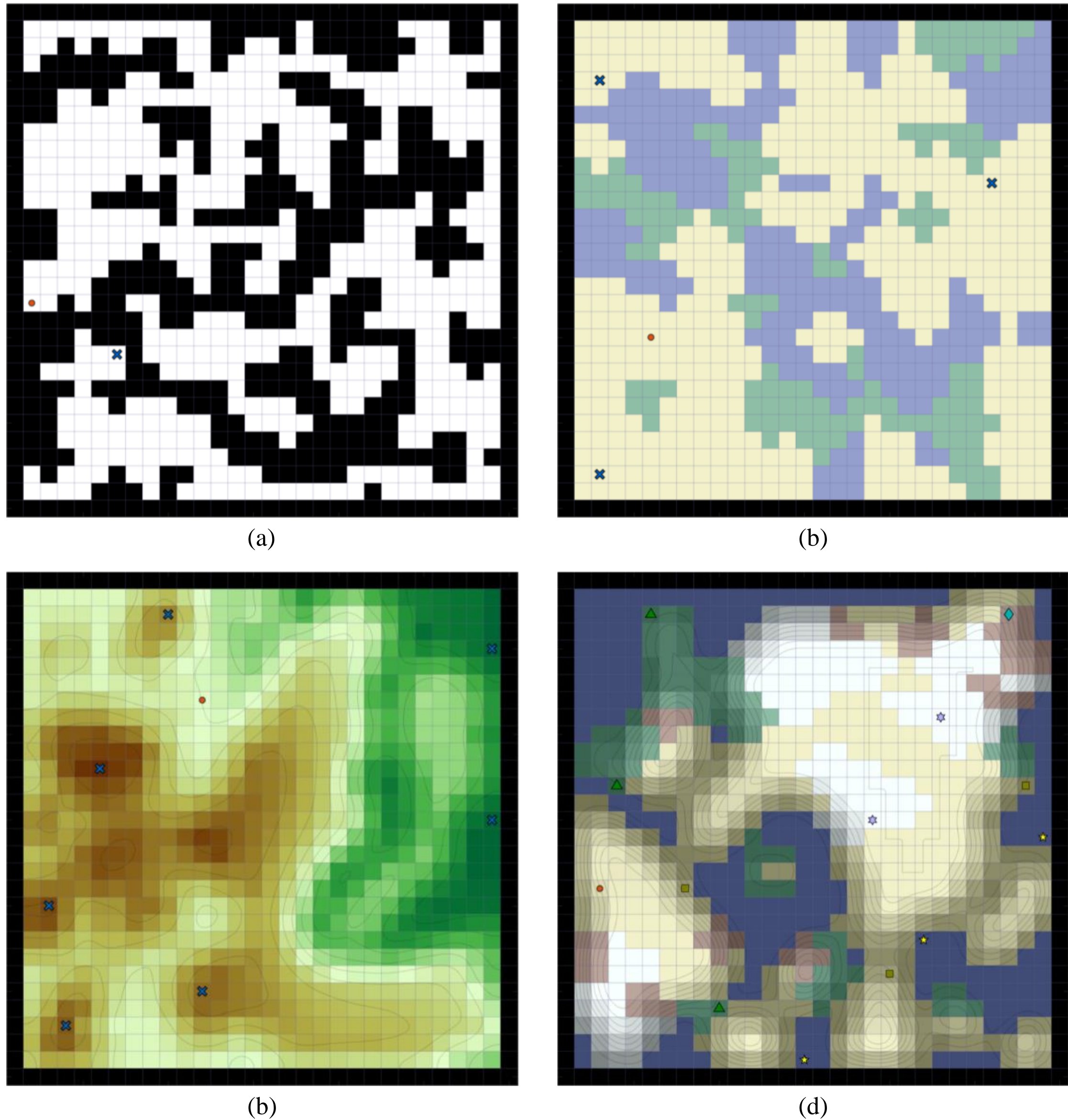


Figure 3.1 Examples of grid world problem domains generated in the CMM framework. (a) A cave-like occupancy grid environment with a single resource. (b) An environment with three different terrain types and multiple resources of the same type. (c) An environment with a real-valued elevation heightmap and multiple resources of the same type. (d) A synthetic world environment combining multiple terrain types with elevation and containing several different resource types.

The grid world environment model is a flexible problem domain that can be adapted to create many interesting scenarios. Some examples of grid world problems generated in the CMM framework are shown in Figure 3.1. Simple occupancy grid environments can

be created by defining a single region type with constant elevation. Multiobjective problems can be created by defining multiple region types and/or an elevation heightmap. The different resource types provide goals for the agent to pursue while moving through the environment. While these environments can be defined manually, either as hand-crafted models or based on real-world data, our focus in this work is on the generation and study of purely synthetic environments using procedural methods. The various approaches used by the CMM framework to generate environments are described over the next several sections.

Grid worlds are discrete domains that can simplify the representation of environment attributes, but they can also add significant overhead to the memory requirement and computational burden in large problems. We address this issue in two ways. First, we limit the size of the grid world environments to relatively small dimensions (typically between 30×30 and 100×100) to prevent an exponential growth in the number of grid cells. Second, we propose an approximate representation of the complete environment by clustering similar regions together using superpixels and working within this reduced domain. This latter approach forms the basis for the region graph representation of the mental map and will be discussed in more detail in Chapter 5. The remainder of this chapter is dedicated to explaining how a complete discrete grid-based environment model is generated from an initial random seed.

3.2 Generating Caverns

The first (and sometimes only) attribute layer we define is the occupancy grid that specifies which cells are traversable by the agent. Simple problem spaces can be defined

with few or no obstacles, but the problems become more interesting as the environments are filled with walls and passageways that limit the agent's navigation options. In the extreme case, the environment could be a complex maze that is challenging even for a human to solve. State-of-the-art path planning algorithms make solving mazes and other fully observable occupancy grid environments somewhat trivial, but when the environment is not completely visible, these obstacles serve to limit how much the agent can see. We model the occupancy grid layer as a 2D cavern map, generated using a cellular automation. These maps have an organic quality with both winding passageways and large open regions containing non-uniform random structure that leads to interesting shortest path problems and visibility constraints.

The general outline of our method for creating the occupancy grid is given in Algorithm 3.1. The process begins by initializing an $n \times m$ grid with random values, where each grid cell is assigned a value of 1 with probability p_0 and a value of 0 with probability $1-p_0$ (lines 1-4). The algorithm then iterates through k steps of cellular automation rules, represented by the CELLULAR_AUTOMATA function in Algorithm 3.2. The rules are applied simultaneously to all grid cells and are defined by two parameters, r_b and r_d, where $r_b \geq r_d$. For each grid cell, we check to see how many of the eight cells in the 3×3 Moore neighborhood around each cell are set to 1. If this number is greater than the birth rate (r_b), the cell is set to 1. Otherwise, if it is less than the death rate (r_d), the cell is set to 0. Between each generation, we apply an optional mask to constrain the cellular growth to a specified region (line 7). This is used when creating certain types of terrain in environments that

already have an occupancy grid layer defined. The border cells are always set to 0 between generations to ensure that the world is fully enclosed (line 8).

Algorithm 3.1 Cave Environment Generation

GENERATE_CAVE_ENVIRONMENT(n, m, p_0, r_b, r_d, k, *mask*, *opt*)
1: $W \leftarrow n \times m$ grid initalized to 0
2: **for** each $(i, j) \in \{(i, j) \mid 1 \leq i \leq n \wedge 1 \leq j \leq m\}$
3: **if** $r \sim U(0, 1) \leq p_0$ *// With probability* p_0
4: $W[i, j] \leftarrow 1$
5: **for** k iterations
6: $W \leftarrow$ CELLULAR_AUTOMATA(W, r_b, r_d) *// Algorithm 3.2*
7: $W[i, j] \leftarrow 0$ for all cells where $mask[i, j] = 1$
8: $W[i, j] \leftarrow 0$ for all cells where (i, j) is a border cell
9: **if** *opt.makeConnected*
10: **while** (# of connected components in W) > 1
11: $Z \leftarrow$ the smallest connected component in W
12: $Z' \leftarrow Z \oplus [0\ 1\ 0;\ 1\ 1\ 1;\ 0\ 1\ 0]$ *// Image dilation*
13: **if** *opt.method* = "dilate"
14: $W \leftarrow \max(W, Z')$
15: **elseif** *opt.method* = "random"
16: $Y \leftarrow Z' \wedge \neg Z$
17: $(i, j) \leftarrow$ random grid cell where $Y[i, j] = 1$
18: $W[i, j] \leftarrow 1$
19: $W[i, j] \leftarrow 0$ for all cells where $mask[i, j] = 1$
20: $W[i, j] \leftarrow 0$ for all cells where (i, j) is a border cell
21: $W \leftarrow$ REMOVE_DIAGONAL_PASSAGES(W) *// Algorithm 3.3*
22: **return** W

Algorithm 3.2 Cellular Automata

CELLULAR_AUTOMATA(W, r_b, r_d)

1: $(n, m) \leftarrow$ size of W
2: $W' \leftarrow W$
3: **for** each $(i, j) \in \{(i, j) \mid 1 \leq i \leq n \wedge 1 \leq j \leq m\}$
4: $\quad N \leftarrow \{(u, v) \mid i-1 \leq u \leq i+1 \wedge j-1 \leq v \leq j+1 \wedge (u, v) \neq (i, j) \wedge W[u, v] = 1\}$
5: $\quad$ **if** $|N| > r_b$
6: $\quad\quad W'[i, j] \leftarrow 1$
7: $\quad$ **elseif** $|N| < r_d$
8: $\quad\quad W'[i, j] \leftarrow 0$
9: **return** W'

In most cases, we want every open location in the environment to be reachable so that the agent can acquire all the resources. This is controlled by an options parameter that specifies if the environment is to be connected and which method should be used to make it connected. The two methods are referred to as *dilate* and *random*. In both approaches, the environment map is refined iteratively until there is only a single connected component. On each iteration, we identify the smallest connected component consisting of a contiguous set of 4-connected open cells (with a value of 1) and call this image Z, where $Z[i, j] = 1$ if cell (i, j) is part of the smallest connected component and $Z[i, j] = 0$ otherwise (line 11). We then compute Z' by dilating Z with a 4-connected mask that sets any cell in Z' to 1 if one of its 4-connected neighbors in Z is 1 (line 12). If the method is set to *dilate*, then the entirety of the newly dilated region is set to 1 (line 14). For the *random* method, only one of the newly opened grid cells is set to 1, chosen randomly (lines 16-18). The dilate method operates faster than the random method, but can produce irregular open regions if the smallest connected component is large. In contrast, the random method produces a more

uniform appearance, but operates more slowly. Between each iteration, we perform a clean-up step by setting all cells in the mask and on the border to zero (lines 19-20). We also remove any passageways that are only connected diagonally using the REMOVE_DIAGONAL_PASSAGES function presented in Algorithm 3.3. This procedure checks each grid cell to see if it is part of a diagonally connected passage and if it is found to be so, the cell is filled in and set to 0. This improves the visual appearance of the environment and helps the visibility computation, which is discussed further in Section 4.1.2. Several examples of cave environments are shown in Figure 3.2.

Algorithm 3.3 Remove Diagonal Passages

REMOVE_DIAGONAL_PASSAGES(W)

1: $(n, m) \leftarrow$ size of W
2: $changed \leftarrow True$
3: **while** $changed$
4: $\quad changed \leftarrow False$
5: $\quad$ **for** each $(i, j) \in \{(i, j) \mid 2 \le i \le n-1 \wedge 2 \le j \le m-1\}$
6: $\quad\quad$ **if** $W[i, j] = 1 \wedge$
$$\big((W[i-1, j-1] = 1 \wedge W[i-1, j] = 0 \wedge W[i, j-1] = 0) \vee$$
$$(W[i+1, j-1] = 1 \wedge W[i+1, j] = 0 \wedge W[i, j-1] = 0) \vee$$
$$(W[i-1, j+1] = 1 \wedge W[i-1, j] = 0 \wedge W[i, j+1] = 0) \vee$$
$$(W[i+1, j+1] = 1 \wedge W[i+1, j] = 0 \wedge W[i, j+1] = 0)\big)$$
7: $\quad\quad\quad W[i, j] \leftarrow 0$
8: $\quad\quad\quad changed \leftarrow True$
9: **return** W

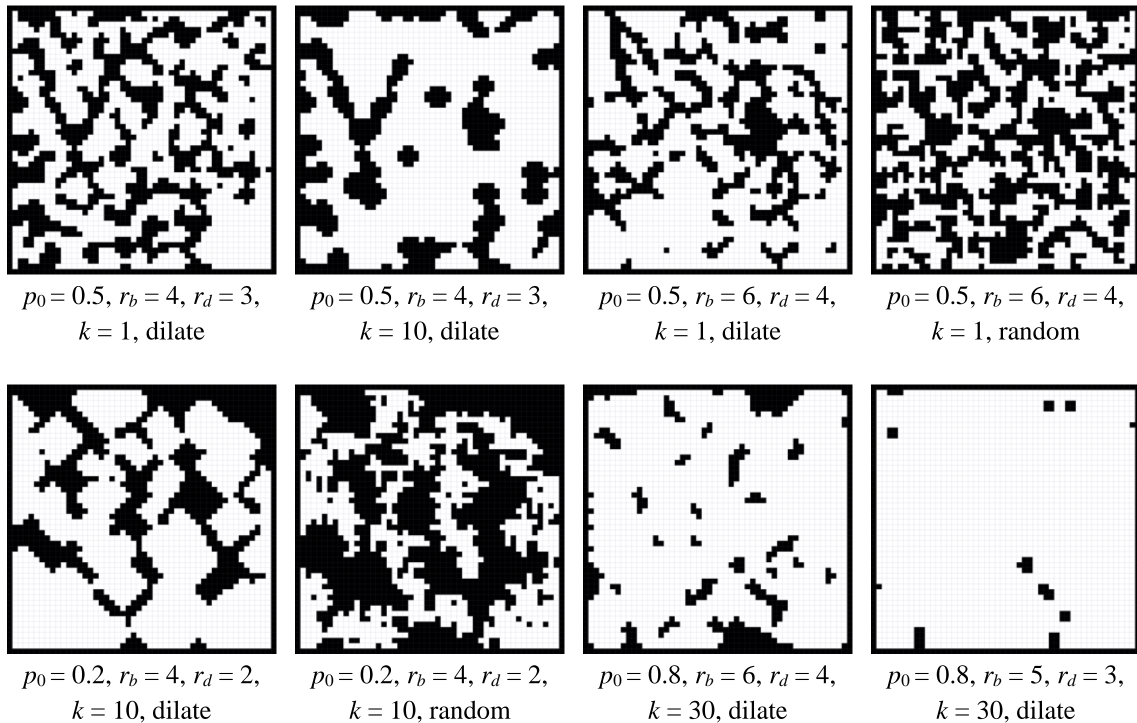


Figure 3.2 Examples of cavern maps generated using Algorithm 3.1. A wide range of map types can be created by varying the input parameters. These examples are 50×50 grids with all locations set to be reachable.

3.3 Region Partitioning

The grid world environments can be partitioned into discrete regions to aid in the generation of additional features and to simplify the mental map representation. Our method for partitioning the environment is based on the SLIC superpixel clustering algorithm (Achanta et al. 2012). SLIC stands for *simple linear iterative clustering* and is an adaptation of the *k*-means algorithm for producing superpixels in color imagery. The version of SLIC used in this work is modified slightly from the original method used on color imagery and is designed for use in grid world environments. Instead of three color channels, we use an elevation map to group similar grid cells. Distances are also computed

with respect to the cave wall boundaries. If no elevation map is provided, only spatial distance is used in determining region boundaries. We include the reference to an optional elevation map here since this procedure is used in the definition of region boundaries for the mental map representation in Chapter 5. However, since a region map may be needed to create an elevation map in Section 3.4, the elevation map is not required to define the region partitions. An overview of the region partitioning method is shown in Algorithm 3.4.

Algorithm 3.4 Region Partitioning

PARTITION_REGIONS(W, E, r, w_e, ϵ)
1: $C \leftarrow$ TABU_SAMPLING(W, r) // *Algorithm 3.5*
2: $C \leftarrow$ ADJUST_CLUSTER_CENTERS(E, C) // *Algorithm 3.7*
3: $(n, m) \leftarrow$ size of W
4: $R \leftarrow n \times m \times |C|$ matrix initalized to ∞
5: $e \leftarrow \infty$
6: **for** $\lceil \sqrt{r} \rceil$ iterations
7:     **for** k in 1 to $|C|$
8:         $(u, v) \leftarrow C[k]$
9:         $R[\ldots, \ldots, k] \leftarrow$ GRID_DISTANCE(W, u, v, $2r$) // *Algorithm 3.6*
10:     $L, e' \leftarrow$ ASSIGN_CELLS_TO_CLUSTERS(E, C, R, r, w_e) // *Algorithm 3.8*
11:     **if** $(e - e') / e < \epsilon$
12:         **break**
13:     $e \leftarrow e'$
14:     $C \leftarrow$ GET_REGION_CENTERS(L) // *Algorithm 3.9*
15: $L \leftarrow$ FIX_ORPHANS(C, L, R) // *Algorithm 3.10*
16: **return** L

The algorithm takes a cave wall map W and an optional elevation map E as inputs, along with a cluster separation radius r, an elevation weighting parameter w_e, and an improvement tolerance threshold ϵ. We start by sampling evenly spaced cluster centers. In

standard SLIC, this is accomplished by sampling on a grid at regular intervals. Because of the irregular nature of the cavern environments, this could lead to some narrow passageways being missed and having no local cluster center. Instead, we use tabu sampling to generate the cluster centers. Algorithm 3.5 shows the procedure for tabu sampling in a cavern map. The algorithm takes a cavern map W and a separation radius r as input. To begin, all open grid cells are added to a set of valid sample indices (line 3). While there are still grid cells in this set, a random location is sampled and added to the list of cluster centers (lines 5-7). After sampling a new cluster center, we compute the grid distance from that location and remove any cells within a specified radius r from the set of valid indices (lines 8-10). This ensures that no two samples are too close to one another. By continuing until there are no more valid locations, we obtain a set of samples that span the entire grid. Figure 3.3 shows several examples of tabu sampling using this approach with different values for the separation radius.

Algorithm 3.5 Tabu Sampling

TABU_SAMPLING(W, r)
1: $k \leftarrow 0$
2: $C \leftarrow$ empty list
3: $I \leftarrow \{(i, j) \mid W[i, j] = 1\}$ *// Get the set of valid indices*
4: **while** $|I| > 0$
5: $\quad k \leftarrow k + 1$
6: $\quad (i, j) \sim I$ *// Sample a random grid cell in I*
7: $\quad C[k] \leftarrow (i, j)$
8: $\quad D \leftarrow$ GRID_DISTANCE(W, i, j, r) *// Algorithm 3.6*
9: $\quad$ **for** each $(u, v) \in \{(u, v) \mid D[u, v] \leq r\}$
10: $\qquad I \leftarrow I \setminus \{(u, v)\}$ *// Remove from I*
11: **return** C

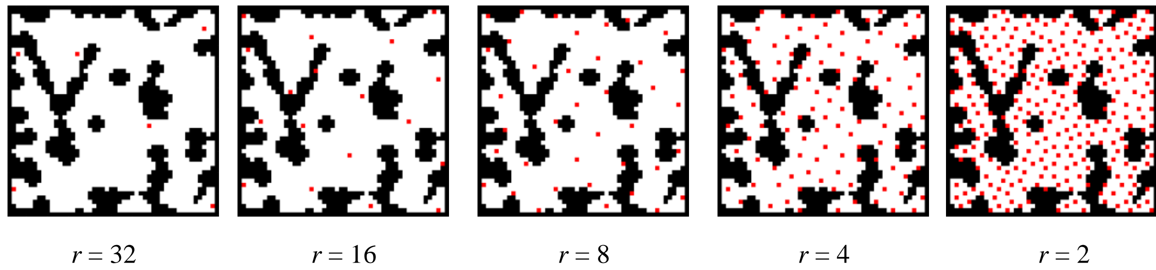


Figure 3.3 Tabu sampling on a 50×50 grid with different values for the separation radius.

The computation of grid distance on line 8 of the TABU_SAMPLING function is detailed in Algorithm 3.6. This function serves many roles throughout this work and will be revisited in Chapter 5 for computing edge attributes for the mental map representation of the environment. The input arguments are a cavern map W and a reference location (i, j), along with a maximum distance parameter d_{max}. The algorithm is based on the approach presented by (Lee 1961) and operates as a flood-fill, breadth-first search that radiates outward from the starting location. To summarize the procedure, we begin by initializing a matrix D to infinity (lines 1-2) that will store the shortest path grid distance from (i, j) that obeys the wall boundaries. A distance counter d is set to 0 and we initialize the *open* and *closed* sets, adding the starting location to the *open* set (lines 3-5). While the *open* set is not empty and the distance counter is less than the maximum value, we set the distance of each grid cell in the *open* set to the current distance counter value (line 9) and move the cell to the *closed* set (line 10). We then identify the neighbors of this cell that are not walls and have not yet been added to the *closed* set (lines 11-12). These locations are collected in the *frontier* set, which replaces the *open* set after all the previous cells in the *open* set have been evaluated (line 13) and the distance counter is incremented by one (line 14). This algorithm works as a flood fill method using breadth-first search to label each location

in the environment. The maximum distance parameter allows the search to terminate early, which can greatly reduce computation time when only close distances are needed. Once all cells have been evaluated and the *open* set is empty, the algorithm returns the distance matrix D.

Algorithm 3.6 Grid Distance

GRID_DISTANCE(W, i, j, d_{max})
1: $(n, m) \leftarrow$ size of W
2: $D \leftarrow n \times m$ matrix initalized to ∞
3: $d \leftarrow 0$
4: $open \leftarrow \{(i, j)\}$
5: $closed \leftarrow \emptyset$
6: **while** $|open| > 0 \wedge d \leq d_{max}$
7: $\quad frontier \leftarrow \emptyset$
8: $\quad$ **for** each $(u, v) \in open$
9: $\quad\quad D[u, v] \leftarrow d$
10: $\quad\quad closed \leftarrow closed \cup (u, v)$
11: $\quad\quad N \leftarrow \{(u-1, v), (u+1, v), (u, v-1), (u, v+1)\}$
12: $\quad\quad frontier \leftarrow frontier \cup \{(u', v') \mid (u', v') \in N \wedge (u', v') \notin closed \wedge W[u', v'] = 1\}$
13: $\quad open \leftarrow frontier$
14: $\quad d \leftarrow d + 1$
15: **return** D

After the initial cluster centers have been defined, they can be moved into a local minimum of the elevation gradient if an elevation map is provided. If no elevation is provided, the initial cluster centers are used without adjustment. Algorithm 3.7 shows the procedure for updating the set of cluster centers C using an elevation map E. The first step of this algorithm (line 1) is to compute the gradient of E which we denote as the matrix F.

Element $F[i,j]$ is defined as the magnitude of the central difference for interior points such that

$$F[i,j] = \sqrt{(F_x[i,j])^2 + (F_y[i,j])^2}, \quad \text{where} \tag{3.1}$$

$$F_x[i,j] = 0.5 * (E[i,j+1] - E[i,j-1]), \tag{3.2}$$

$$F_y[i,j] = 0.5 * (E[i+1,j] - E[i-1,j]). \tag{3.3}$$

For exterior points and points where one of the neighbors of $E[i,j]$ is undefined, the gradient is computed as the single-sided difference using only the defined values. For instance, if $E[i-1,j]$ is undefined, but $E[i+1,j]$, $E[i,j-1]$, and $E[i,j+1]$ are defined, then $F_x[i,j]$ keeps the same definition as Equation 3.2, but $F_y[i,j]$ is defined as

$$F_y[i,j] = (E[i+1,j] - E[i,j]). \tag{3.4}$$

Once the gradient matrix has been computed, the algorithm cycles through each cluster center and moves it to the location of the minimum gradient value within a 3×3 neighborhood. This helps prevent clusters from being located on natural region boundaries and generally improves the stability of the algorithm.

Algorithm 3.7 Adjust Cluster Centers

ADJUST_CLUSTER_CENTERS(E, C)
1: $F \leftarrow \nabla E$
2: **for** k in 1 to $|C|$
3: $(i, j) \leftarrow C[k]$
4: $G \leftarrow \{(u, v) \mid i-1 \le u \le i+1 \wedge j-1 \le v \le j+1\}$
5: $(u, v) \leftarrow (u, v) \in G$ s.t. $F[u, v] \le F[u', v'] \ \forall \ (u', v') \in G$
6: $C[k] \leftarrow (u, v)$
7: **return** C

At this point, we begin the main loop of Algorithm 3.4 and iteratively assign all grid cells to the nearest cluster center and then update the cluster center locations. This outer loop can be repeated until the cluster assignments do not change, or until some improvement threshold is reached. In practice, we have found it useful to scale the maximum number of iterations to be proportional to the effective cluster size, which is determined by the separation radius r. In our experiments, we use $\lceil \sqrt{r} \rceil$ as the maximum number of iterations and 0.01 as the improvement tolerance threshold ϵ. The first step during each iteration is to precompute the grid distance values for each cluster. These are stored in the $n \times m \times |C|$ matrix R, where n and m are the dimensions of the environment and $|C|$ is the number of clusters. This reference distance matrix is updated each iteration with new cluster center locations and allows for quick distance lookups for each grid cell. Note that the call to the GRID_DISTANCE function on line 9 of Algorithm 3.4 uses $2r$ as the maximum distance parameter to reduce computation time, which assumes that cells will not be assigned to clusters greater than $2r$ steps away.

The next step of the main loop is the assignment of each grid cell to the nearest cluster center. This is accomplished by the ASSIGN_CELLS_TO_CLUSTERS function in Algorithm 3.8. This function starts by initializing a distance matrix D and a label matrix L (lines 1-3). The distance matrix will store the distance from each grid cell to the nearest cluster center and the label matrix will store the index of the nearest cluster center. For each grid cell (i, j) and each cluster center (u, v), we compute both a spatial and an elevation distance. The spatial distance d_s is computed from the grid distance reference matrix R and normalized by the cluster separation radius r (line 7). The elevation distance is computed

as the absolute difference between the elevation of the two grid cells and is multiplied by a weighting parameter w_e (line 8). Note that since the elevation values come from the range [0, 1] the unweighted elevation distance is also restricted to the unit interval. If no elevation map is used, w_e can be set to zero. The two distance measures are combined using the ℓ^2-norm to give the overall cluster distance d (line 9). The index of the cluster center with the smallest distance to a given grid cell is stored in the label matrix L and the distance to this cluster center is saved in the distance matrix D (lines 11-12). After all clusters and grid cells have been evaluated, the error is computed as the sum of all non-infinite distances between grid cells and their assigned cluster centers. This error value and the label matrix are then returned to the main algorithm.

Algorithm 3.8 Assign Cells to Clusters

ASSIGN_CELLS_TO_CLUSTERS(E, C, R, r, w_e)
1: $(n, m) \leftarrow$ size of E
2: $L \leftarrow n \times m$ matrix initalized to 0
3: $D \leftarrow n \times m$ matrix initalized to ∞
4: **for** k in 1 to $|C|$
5: $\quad (u, v) \leftarrow C[k]$
6: $\quad$ **for** each $(i, j) \in \{(i, j) \mid 1 \leq i \leq n \wedge 1 \leq j \leq m\}$
7: $\quad\quad d_s \leftarrow R[i, j, k] \,/\, r$
8: $\quad\quad d_e \leftarrow w_e \times |E[i, j] - E[u, v]|$
9: $\quad\quad d \leftarrow \sqrt{d_s^2 + d_e^2}$
10: $\quad\quad$ **if** $d < D[i, j]$
11: $\quad\quad\quad D[i, j] \leftarrow d$
12: $\quad\quad\quad L[i, j] \leftarrow k$
13: $e \leftarrow \sum D[i, j]$ for all (i, j) such that $D[i, j] \neq \infty$
14: **return** L, e

Once all the grid cells have been assigned a cluster label, we can check if the error term has improved significantly from the previous iteration. Lines 11-13 of Algorithm 3.4 check to see if the improvement is less than the error tolerance threshold and break out of the main loop if it is small enough. If not, then the main loop continues and we use the GET_REGION_CENTERS function to get new cluster centers at the centroids of each region. Algorithm 3.9 gives the pseudocode for this procedure. We begin by initializing a new empty cluster list C (line 1). We then loop over each region label index and identify the grid cells that belong to each region (line 4). It should be noted that this algorithm assumes that there is at least one cell in the label map L for each index k in 1, ..., max(L). Next, we compute the centroid of the region, rounding to the nearest cell (lines 5-10). If this cell is not already part of the region, we move the region center to the cell closest to the centroid that is already labeled as part of the region (lines 11-12). The list of region centers is then returned to the main algorithm to be used as the new cluster centers.

Algorithm 3.9 Get Region Centers

GET_REGION_CENTERS(L)
1: $C \leftarrow$ empty list
2: $t \leftarrow 0$
3: **for** k in 1 to max(L)

/ Get the grid cells that belong to this region */*
4: $S \leftarrow \{(i, j) \mid L[i, j] = k\}$

/ Compute the centroid of the region */*
5: $(c_i, c_j) \leftarrow (0, 0)$
6: **for** each $(i, j) \in S$
7: $c_i \leftarrow c_i + i / |S|$
8: $c_j \leftarrow c_j + j / |S|$
9: $c_i \leftarrow \lfloor c_i + 0.5 \rfloor$
10: $c_j \leftarrow \lfloor c_j + 0.5 \rfloor$

/ Move the centroid to the closest cell in the region if not already in the region */*
11: **if** $L[c_i, c_j] \neq k$
12: $(c_i, c_j) \leftarrow (i, j) \in S$ s.t. $\sqrt{(i - c_i)^2 + (j - c_j)^2} \leq \sqrt{(i' - c_i)^2 + (j' - c_j)^2} \;\; \forall\, (i', j') \in S$

/ Save the region center */*
13: $t \leftarrow t + 1$
14: $C[t] \leftarrow (c_i, c_j)$

15: **return** C

The main loop in Algorithm 3.4 continues until the maximum number of iterations has been reached or the improvement threshold has been met. A final post-processing step is performed to remove orphaned grid cells that are not connected to a cluster center. This is done by the FIX_ORPHANS function in Algorithm 3.10. The function takes the current cluster centers C, the label matrix L, and the reference distance matrix R as input. The

algorithm loops repeatedly until it can be verified that there are no remaining orphans. For each cluster, we identify the main connected component by using the GRID_DISTANCE function seeded at the cluster center with the non-cluster cells acting as walls (lines 7-11). Any grid cells that were labeled as part of the cluster, but that were unreachable by the GRID_DISTANCE function are marked as orphan cells (line 12). For each orphan cell, we get the 4-connected neighbors (line 15) and check to see which of these have different labels from the label of the orphan cell (line 16). If none of the neighbor cells have different labels, then the algorithm moves on to the next orphan cell and will eventually return to this cell after the other orphans have been processed (lines 17-18). The label of the orphan cell is then set to the label of the neighboring cell with the smallest distance to the original cluster center (lines 19-20). After the outer loop of the FIX_ORPHANS function can verify that none of the clusters contain any orphans, the updated label matrix is returned to the main algorithm. Note that we do not explicitly check for race conditions that could lead to an infinite loop. In practice this is very rare and is best resolved by restarting with a different random seed or separation radius. Some examples of the final cluster labeling using different values of separation radius are shown in Figure 3.4.

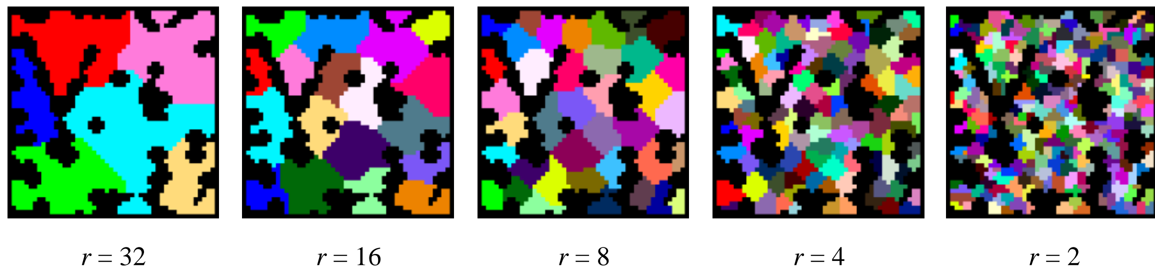


Figure 3.4 Results of the region partitioning algorithm on a 50×50 grid with different values for the separation radius.

Algorithm 3.10 Fix Orphans

```
FIX_ORPHANS(C, L, R)
1:  (n, m) ← size of L
2:  hasOrphans ← true
3:  while hasOrphans
4:      hasOrphans ← false
5:      for k in 1 to |C|
6:          (u, v) ← C[k]

            /* Get the orphans for this cluster */
7:          S ← {(i, j) | L[i, j] = k}
8:          B ← n × m matrix initalized to 0
9:          for each (i, j) ∈ S
10:             B[i, j] ← 1
11:         D ← GRID_DISTANCE(B, u, v, ∞)                    // Algorithm 3.6
12:         O ← {(i, j) | (i, j) ∈ S ∧ D[i, j] = ∞}

            /* Assign each orphan a neighboring label */
13:         for each (i, j) ∈ O
14:             hasOrphans ← true
15:             N ← {(i–1, j), (i+1, j), (i, j–1), (i, j+1)}
16:             G ← {(u, v) | (u, v) ∈ N ∧ L[u, v] ≠ 0 ∧ L[u, v] ≠ k}
17:             if |G| = 0
18:                 continue
19:             (u, v) ← (u, v) ∈ G s.t. R[u, v, k] ≤ R[u′, v′, k] ∀ (u′, v′) ∈ G
20:             L[i, j] ← L[u, v]

21: return L
```

3.4 Creating a Heightmap

A heightmap provides a real-valued elevation attribute for every grid cell. This can add a sense of realism to the problem domain and give a new set of features that influence how the agent makes decisions. The heightmaps for the grid world environments are

generated using the method of successive random additions. This approach is based on the fractal terrain methods presented in Section 2.2.3. Several random noise layers are created using the PARTITION_REGIONS function at multiple scales. These are combined to give the overall heightmap.

Algorithm 3.11 gives an overview of the GENERATE_HEIGHTMAP procedure. The algorithm takes as input a cave wall map W and two control parameters, p and q, which influence the overall shape and texture of the heightmap. The algorithm starts by initializing the elevation map E to zero and the scale factor to one (lines 1-3). The main loop of the algorithm repeats while the scale factor is less than the largest dimension of the grid world. At the end of each iteration, the scale factor is doubled (line 17). At the start of each iteration, the environment is partitioned into regions proportional to the current scale factor (line 5). Each labeled region is then assigned a random value from the uniform distribution $U(0, 1)$ (lines 6-10). This effectively creates a random noise image R at the current scale. Figure 3.5 and Figure 3.6 show several examples of random noise images generated at different scales with and without a cave wall map. Note that because we use the PARTITION_REGIONS function to define the region boundaries, the individual regions at larger scales will not cross the cells marked as walls. This allows high and low regions to be separated by only a thin wall, which is a difficult effect to achieve with other image scaling methods. Each noise image R is smoothed using a mean filter in a 3×3 window (lines 11-14). This is repeated q times, with larger values of q producing smoother terrain. The image R is then multiplied by a scale factor $s^{1/p}$ and added to the current elevation map E (lines 15-16). Larger values of p produce more homogeneous noise, as all scales

begin to be weighted equally. Smaller values of p give sharply distinct regions, as the larger image scales dominate the overall elevation map. After all iterations have completed, the elevation map is normalized to the range [0, 1] and retuned. Figure 3.7 shows the effect of p and q on the generated heightmap.

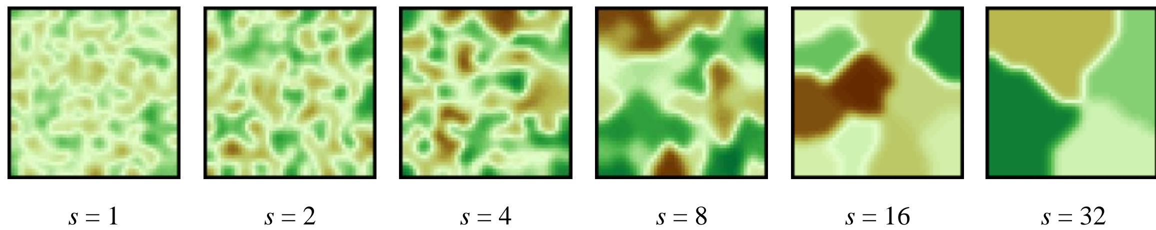


Figure 3.5 Random noise images at different scales on a 50×50 grid with no cave walls. ($q = 3$)

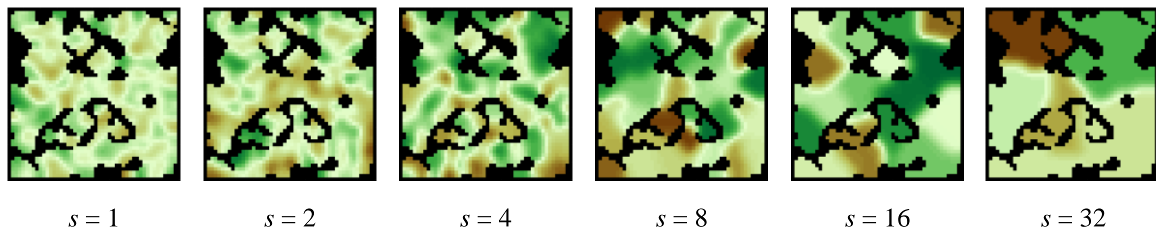


Figure 3.6 Random noise images at different scales on a 50×50 grid with a provided cave wall map. ($q = 3$)

Algorithm 3.11 Heightmap Generation

GENERATE_HEIGHTMAP(W, p, q)
1: $(n, m) \leftarrow$ size of W
2: $E \leftarrow n \times m$ matrix initalized to 0
3: $s \leftarrow 1$
4: **while** $s < n \vee s < m$

/ Create a random elevation map at the current scale */*
5: $L \leftarrow$ PARTITION_REGIONS(W, E, s, 0, 0.01) *// Algorithm 3.4*
6: $R \leftarrow n \times m$ matrix initalized to 0
7: **for** k in 1 to max(L)
8: $r \sim U(0, 1)$
9: **for** each $(i, j) \in \{(i, j) \mid L[i, j] = k\}$
10: $R[i, j] \leftarrow r$

/ Smoothing */*
11: **for** q iterations
12: **for** each $(i, j) \in \{(i, j) \mid 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge W[i, j] = 1\}$
13: $G \leftarrow \{(u, v) \mid i-1 \leq u \leq i+1 \wedge j-1 \leq v \leq j+1\}$
14: $R[i, j] \leftarrow \text{mean}(R[u, v])$ for all $(u, v) \in G$

/ Add to existing heightmap */*
15: **for** each $(i, j) \in \{(i, j) \mid 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge W[i, j] = 1\}$
16: $E[i, j] \leftarrow E[i, j] + R[i, j] \cdot s^{1/p}$

17: $s \leftarrow 2s$ *// Increase scale*

18: $E \leftarrow \frac{E - \min(E)}{\max(E) - \min(E)}$ *// Normalize*

19: **return** E

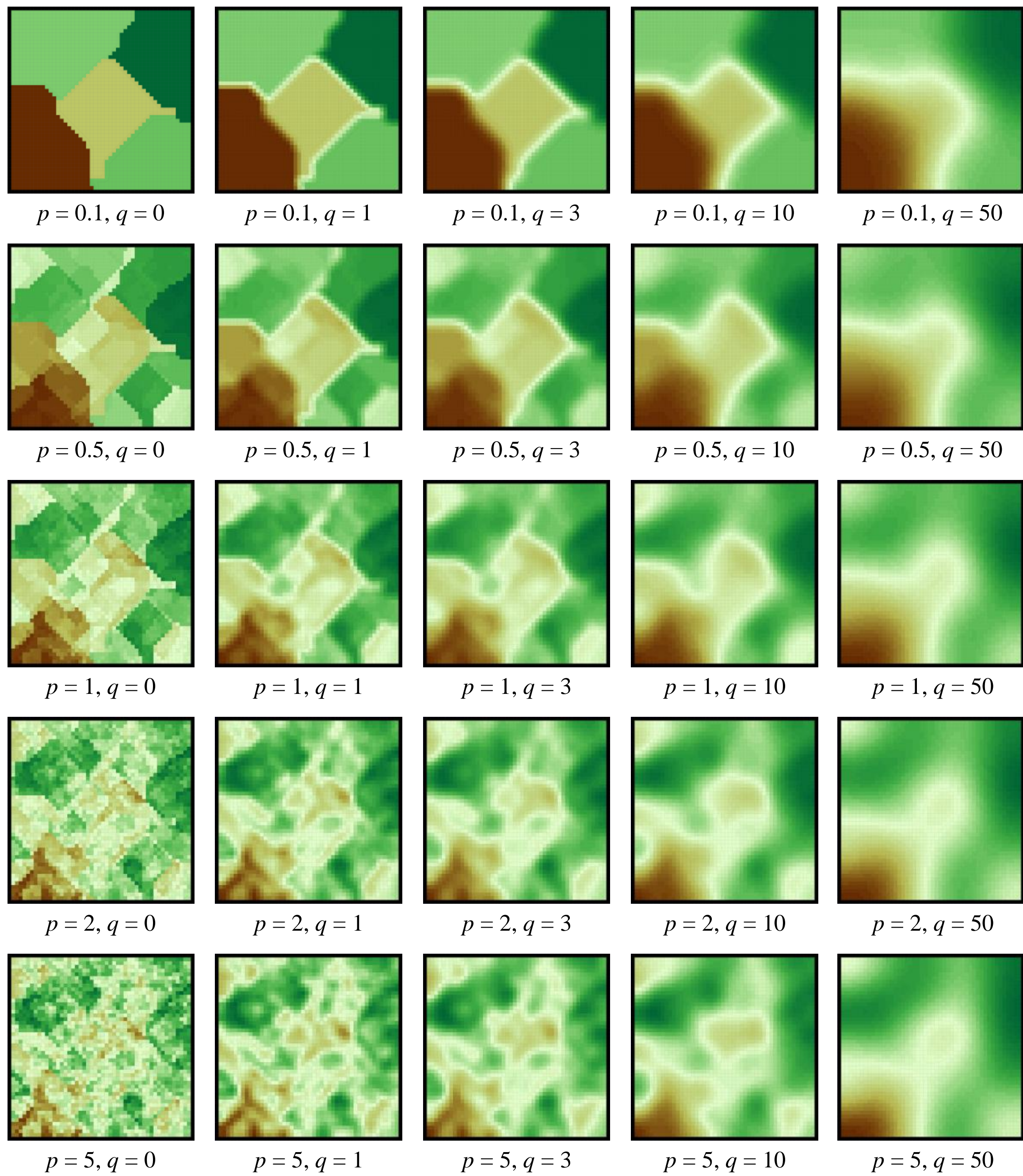


Figure 3.7 Heightmaps generated on a 50×50 grid with different values of p and q using the same random seed. Small values of p give more uniform regions, whereas larger values produce more noise. As q increases, the heightmap becomes smoother and sharp boundaries are eliminated.

3.5 Defining Terrain Types

The terrain feature of the grid world environments provides a discrete attribute that agents can use in the decision-making process. There are several ways that the terrain feature can be generated depending on the desired characteristics of the problem environment. We consider four different problem types in this work. The first uses only a single terrain type, *open_space*, and is used when studying single-objective problems in cavern maps or when elevation is the only relevant environment feature. In this case, there is no need to define additional terrain types. The second problem type uses two terrain types, *meadow* and *forest*, and is used mainly to study bi-objective problems where one type of terrain is preferred over the other and in some multi-objective problems. The third problem type adds *water* to the *meadow* and *forest* terrain types and is used to demonstrate problems that have directional terrain transition preferences, where the agent prefers to move from one type of terrain into another. The last problem type utilizes all the procedural content generation methods discussed and simulates a real-world environment with five terrain types: *meadow*, *forest*, *water*, *rock*, and *snow*.

3.5.1 Binary Terrain Environments

In the binary terrain environment problem type, we define two types of terrain: *meadow* and *forest*. The meadow represents open space where the agent can move freely and the forest offers concealment that may be desirable to some agents. We begin by constructing a cave wall map using the cellular automata method of Section 3.2. The cave walls provide the mask for generating the forested region, which is also created using the GENERATE_CAVE_ENVIRONMENT function. If the initial probability p_0 used to generate the

cave wall map is set to 1, then the wall map mask is set entirely to open space except for the border cells. In this case, the problem is focused entirely on the terrain and the heightmap if provided.

Algorithm 3.12 gives the procedure for generating the binary terrain map. The call to the GENERATE_CAVE_ENVIRONMENT function on line 1 returns a binary map where the "walls" represent the *forest* terrain type and the open space represents *meadow*. Because the output of this function is binary and we wish to indicate three types of grid cells (*wall*, *meadow*, and *forest*), we remap the indices by subtracting the terrain index from 2 if the grid cell was open space in the original cave wall map (lines 2-3). The resulting matrix T maps 0 to walls, 1 to *meadow*, and 2 to *forest* and can be added as an attribute layer in the grid world environment. Several examples of binary environments are shown in Figure 3.8. One interesting parameter is the *opt.makeConnected* setting. If set to true, the *meadow* terrain will be completely connected, such that an agent could get from any *meadow* grid cell to any other *meadow* grid cell without ever going into the forest. If *makeConnected* is set to false, the agent may be forced to go through at least some *forest* terrain.

Algorithm 3.12 Generate Binary Terrain

GENERATE_BINARY_TERRAIN(W, p_0, r_b, r_d, k, opt)
1: $T \leftarrow$ GENERATE_CAVE_ENVIRONMENT(n, m, p_0, r_b, r_d, k, W, opt) *// Algorithm 3.1*
2: **for** each $(i, j) \in \{(i, j) \mid 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge W[i, j] = 1\}$
3: $T[i, j] \leftarrow 2 - T[i, j]$
4: **return** T

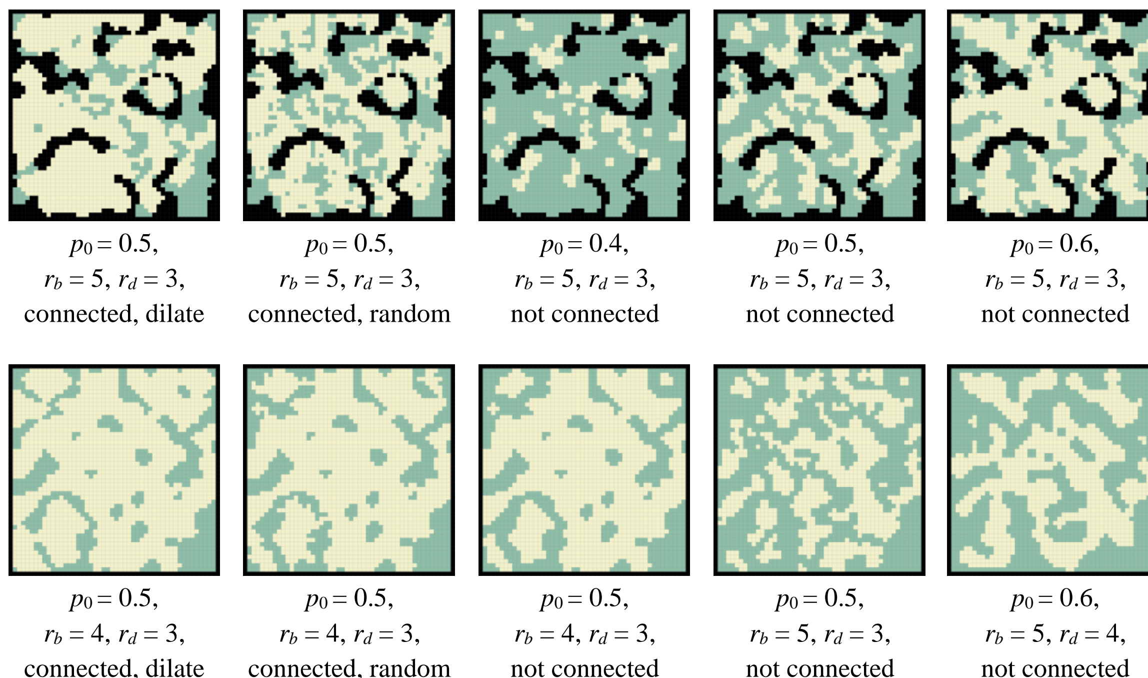


Figure 3.8 Examples of binary environments containing *forest* and *meadow* terrain types. The top row shows the binary environments in grid worlds with a cave wall map generated using parameters p_0=0.5, r_b=4, r_d=3, k=10, and the dilate connection method. The bottom row shows binary environments created without a cave wall map.

3.5.2 Trinary Terrain Environments

For the trinary environment problem type, we add *water* to the *forest* and *meadow* terrain types. We use fashion-based cellular automata such as the one presented in Section 2.2.2 to generate the three terrain types. The outline of our approach is given in Algorithm 3.13. We start by sampling an initial terrain type from a prior distribution P_0 for every grid cell that is open in the cave wall map W (lines 1-4). For trinary terrain environments, P_0 is a multinomial distribution over the domain {1, 2, 3}, indicating the probability of a grid cell starting as *meadow*, *forest*, or *water* respectively. We then apply the fashion-based cellular automation rules for k iterations (line 5-6). Algorithm 3.14 gives the pseudocode

for the FASHION_BASED_CELLULAR_AUTOMATA function. The first part of this function computes the score of each cell that has been assigned a terrain type using a supplied rule matrix R. For each cell, we examine its 4 adjacent neighbors and lookup the score value $R[i, j]$ that corresponds to a cell with terrain type i having a neighbor of terrain type j. The sum of these scores for each neighbor gives the overall score for the cell. Once all the scores have been computed, we set the terrain type of each open grid cell to that of its highest scoring neighbor. If a cell has a higher score than any of its neighbors, it keeps its current label. In this way, the cells "follow the fashion" of the neighborhood.

Figure 3.9 shows several examples of the fashion-based cellular automata for creating trinary terrain environments. The resulting patterns are highly dependent on the rule matrix and the initial distribution of terrain types. It can be difficult to anticipate the type of pattern that any given rule will produce, and it may take several tries to generate a valid environment containing at least some cells of every terrain type.

Algorithm 3.13 Generate Trinary Terrain

GENERATE_TRINARY_TERRAIN (W, P_0, R, k)

/ Sample the initial terrain type for each cell */*
1: $(n, m) \leftarrow$ size of W
2: $T \leftarrow n \times m$ grid initalized to 0
3: **for** each $(i, j) \in \{(i, j) \mid 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge W[i, j] = 1\}$
4: $\quad T[i, j] \sim P_0$ *// Sample a terrain type from the initial distribution*

5: **for** k iterations
6: $\quad T \leftarrow$ FASHION_BASED_CELLULAR_AUTOMATA(T, R) *// Algorithm 3.14*

7: **return** T

Algorithm 3.14 Fashion-Based Cellular Automata

FASHION_BASED_CELLULAR_AUTOMATA(T, R)

/ Compute the score for each cell */*

1: $(n, m) \leftarrow$ size of T
2: $S \leftarrow n \times m$ grid initalized to 0
3: **for** each $(i, j) \in \{(i, j) \mid 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge T[i, j] \neq 0\}$
4: $\quad N \leftarrow \{(i-1, j), (i+1, j), (i, j-1), (i, j+1)\}$
5: $\quad$ **for** each $(u, v) \in N$ s.t. $T[u, v] \neq 0$
6: $\quad\quad S[i, j] \leftarrow S[i, j] + R\big[T[i, j], T[u, v]\big]$

/ Assign the terrain type of the highest-scoring neighbor to each cell */*

7: **for** each $(i, j) \in \{(i, j) \mid 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge T[i, j] \neq 0\}$
8: $\quad N \leftarrow \{(i, j), (i-1, j), (i+1, j), (i, j-1), (i, j+1)\}$
9: $\quad (u, v) \leftarrow (u, v) \in N$ s.t. $S[u, v] \geq S[u', v'] \ \forall \ (u', v') \in N$
10: $\quad T[i, j] \leftarrow T[u, v]$

11: **return** T

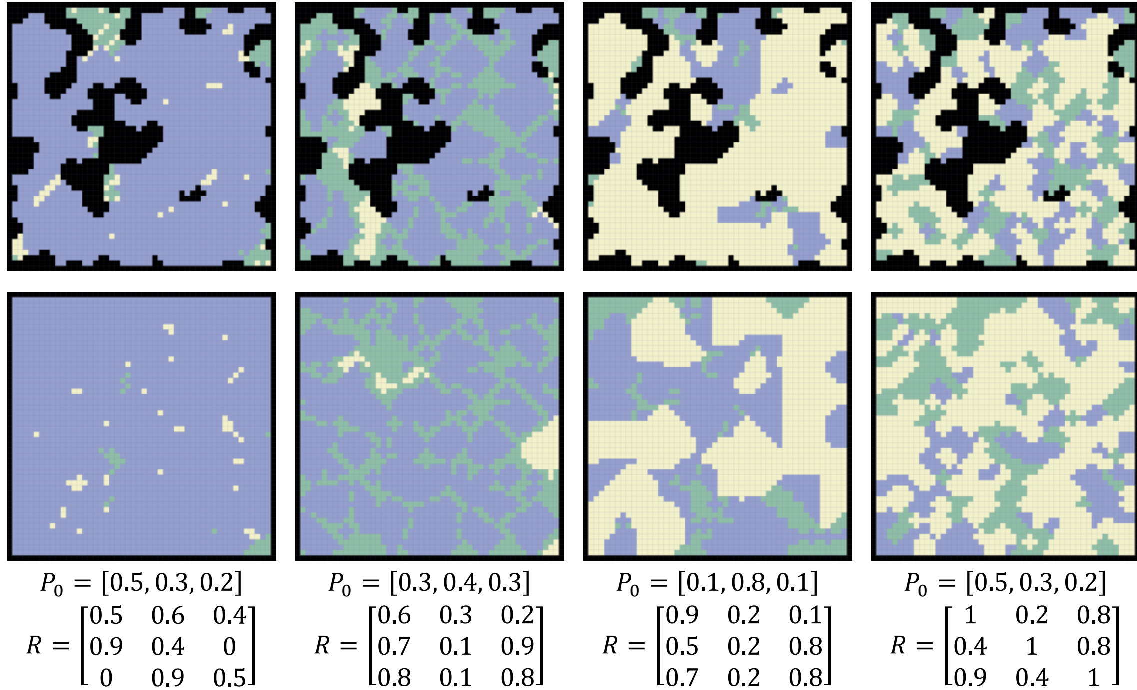


Figure 3.9 Examples of the fashion-based cellular automata algorithm for creating trinary terrain environments. The top row shows the results of Algorithm 3.14 using a cave wall map generated using parameters p_0=0.5, r_b=4, r_d=3, k=10, and the dilate connection method. The bottom row shows trinary environments created without a cave wall map. The initial distribution P_0 and the rule matrix R are the same for each column.

3.5.3 Full World Environments

The last environment type we present simulates a real-world environment with optional cave walls, elevation, and five terrain types: *meadow*, *forest*, *water*, *rock*, and *snow*. For this type of environment, we have hand-chosen some of the parameters after careful experimentation to ensure consistency. The pseudocode of our approach is given in Algorithm 3.15. We begin by creating a cave wall map *W* using the supplied parameters for the GENERATE_CAVE_ENVIRONMENT function (lines 1-2). Next, we create a heightmap *E* using the GENERATE_HEIGHTMAP function (line 3). The lowest elevations should be filled with water, but we would like for the above water elevations to remain scaled in the

range [0, 1]. Lines 4-5 achieve this effect by scaling the elevation map by a factor of 1.2 and then subtracting 0.2. Any grid cells that result in an elevation below zero are marked as water cells and the elevation is set back to zero to indicate sea-level. The gradient of the heightmap is computed on line 6, which is used in the next step to initialize the terrain map.

Lines 7-15 describe how the terrain map is initialized. We construct the prior distribution P_0 independently for each grid cell (i, j) from a set of unnormalized values. A constant value of 0.8 is assigned to $P_0[1]$ representing *meadow*. $P_0[2]$ represents *forest* and is given a value of $1 - E[i, j]$ to give greater weight to lower elevations. Terrain type 3 is used to represent *water*, which is handled separetely, so $P_0[3]$ is set to 0. $P_0[4]$ represents *rock* and is given a value proportional to the square root of the heightmap gradient so that steep slopes have a higher chance of being initialized with *rock*. $P_0[5]$ represents *snow* and is given a value of $(E[i, j])^5$ to strongly favor high elevations. These values are normalized and used to construct a multinomial distribution from which the initial terrain type is sampled.

The terrain is updated using the FASHION_BASED_CELLULAR_AUTOMATA function for 10 iterations using the rule specified on line 16. This simple rule indicates that *meadow* and *forest* terrain types prefer their own types and each gives half weight to the other. The *rock* and *snow* terrain types give full weight to themselves and each other. The third row and column is set to zero to ignore the *water* terrain type since it has already been defined with the heightmap. After each iteration, the terrain type for each grid cell where the elevation is zero is set to *water*. After the terrain has been defined, the cave wall map, heightmap, and terrain map are returned to be used as attribute layers in the grid world environment. Several full world environment examples are shown in Figure 3.10.

Algorithm 3.15 Generate Full World Environment

GENERATE_FULL_WORLD_ENVIRONMENT(n, m, p_0, r_b, r_d, k, opt, p, q)

/ Create the cave wall map */*
1: $mask \leftarrow n \times m$ grid initalized to 0
2: $W \leftarrow$ GENERATE_CAVE_ENVIRONMENT(n, m, p_0, r_b, r_d, k, $mask$, opt) *// Algorithm 3.1*

/ Create the heightmap */*
3: $E \leftarrow$ GENERATE_HEIGHTMAP(W, p, q) *// Algorithm 3.11*
4: **for** each $(i, j) \in \{(i, j) \mid 1 \le i \le n \wedge 1 \le j \le m\}$
5: $\quad E[i, j] \leftarrow \max(0,\ E[i, j] * 1.2 - 0.2)$
6: $F \leftarrow \nabla E$ *// Compute the slope of each grid cell*

/ Initialize the terrain map */*
7: $T \leftarrow n \times m$ grid initalized to 0
8: **for** each $(i, j) \in \{(i, j) \mid 1 \le i \le n \wedge 1 \le j \le m \wedge W[i, j] = 1\}$
9: $\quad P_0[1] \leftarrow 0.8$ *// Meadow*
10: $\quad P_0[2] \leftarrow 1 - E[i, j]$ *// Forest*
11: $\quad P_0[3] \leftarrow 0$ *// Water (handled separately)*
12: $\quad P_0[4] \leftarrow \sqrt{\frac{F[i,j] - \min(F)}{\max(F) - \min(F)}}$ *// Rock*
13: $\quad P_0[5] \leftarrow (E[i, j])^5$ *// Snow*
14: $\quad P_0 \leftarrow P_0 / \text{sum}(P_0)$ *// Normalize*
15: $\quad T[i, j] \sim P_0$ *// Sample a terrain type*

/ Create the terrain map */*
16: $R \leftarrow \begin{bmatrix} 1 & 0.5 & 0 & 0 & 0 \\ 0.5 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$
17: **for** 10 iterations
18: $\quad T \leftarrow$ FASHION_BASED_CELLULAR_AUTOMATA(T, R) *// Algorithm 3.14*
19: $\quad$ **for** each $(i, j) \in \{(i, j) \mid 1 \le i \le n \wedge 1 \le j \le m \wedge E[i, j] = 0\}$
20: $\quad\quad T[i, j] \leftarrow 3$ *// Set water terrain type*

21: **return** W, E, T

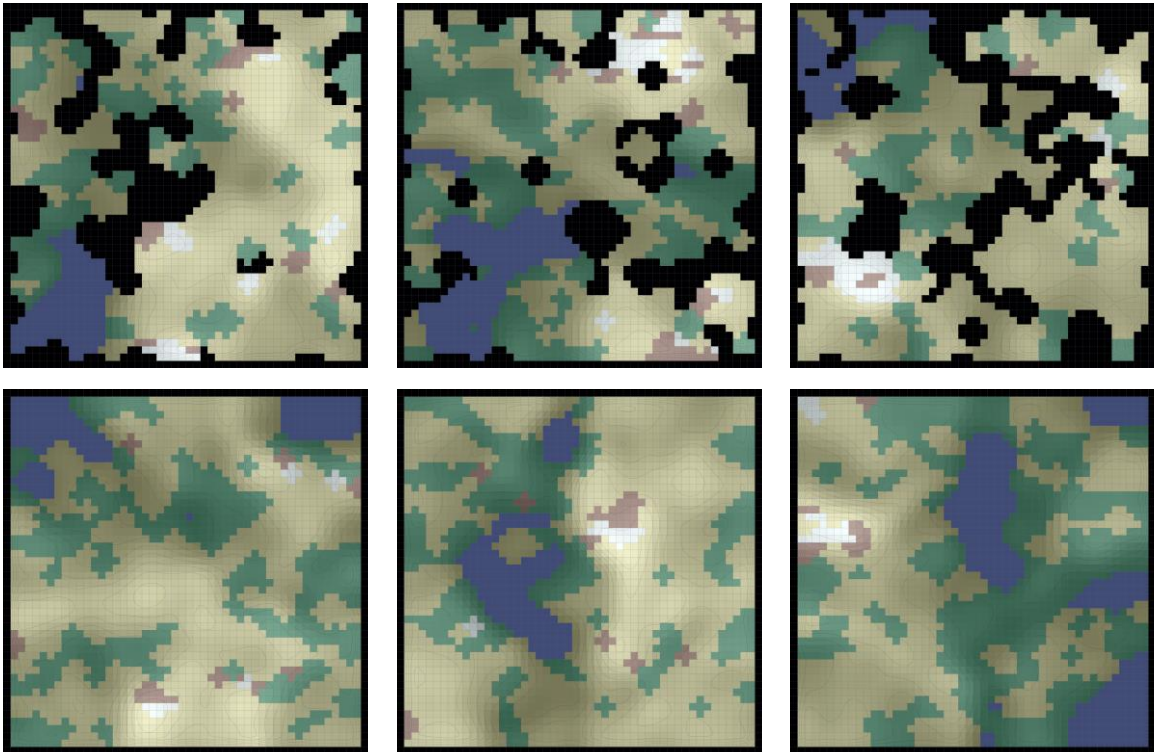


Figure 3.10 Examples of full world environments generated using Algorithm 3.15. The top row uses a cave wall map generated using parameters p_0=0.5, r_b=4, r_d=3, k=10, and the dilate connection method. The heightmap generation parameters are p=2 and q=3. The bottom row shows examples without any cave wall map.

3.6 Resource Placement

The last attribute layer specifies the resources that are present in the environment. The resources represent goal locations that the agent needs to visit to satisfy the problem requirements. There are many ways that the resources can be placed in the environment and some problems may require a different approach from the methods presented here. We consider three general classes of problems: Shortest path problems (SPP), traveling salesman problems (TSP), and traveling purchaser problems (TPP).

In shortest path problems, the agent must navigate through the environment to a single resource location, choosing a route that minimizes its objectives. To initialize the problem, we only need to place the agent and a single resource. One straightforward way to accomplish this is to use tabu sampling with Algorithm 3.5 to sample many possible locations with some minimum separation, and then select two locations that are reasonably far apart. For instance, a SPP can be initialized by using tabu sampling to sample locations with a 5-cell separation radius and then placing the agent at the sampled location closest to the origin and the goal at the location farthest from the origin. This approach is simple to implement, but may not utilize the entire environment space. An alternative is to compute the all-pairs shortest path distances between every open grid cell and place the agent and the goal at the two locations that have the maximum distance between them. This requires a graph representation of the environment, which will be discussed further in Chapter 4. Figure 3.11 shows an example cavern environment with the SPP initialized using these two approaches.

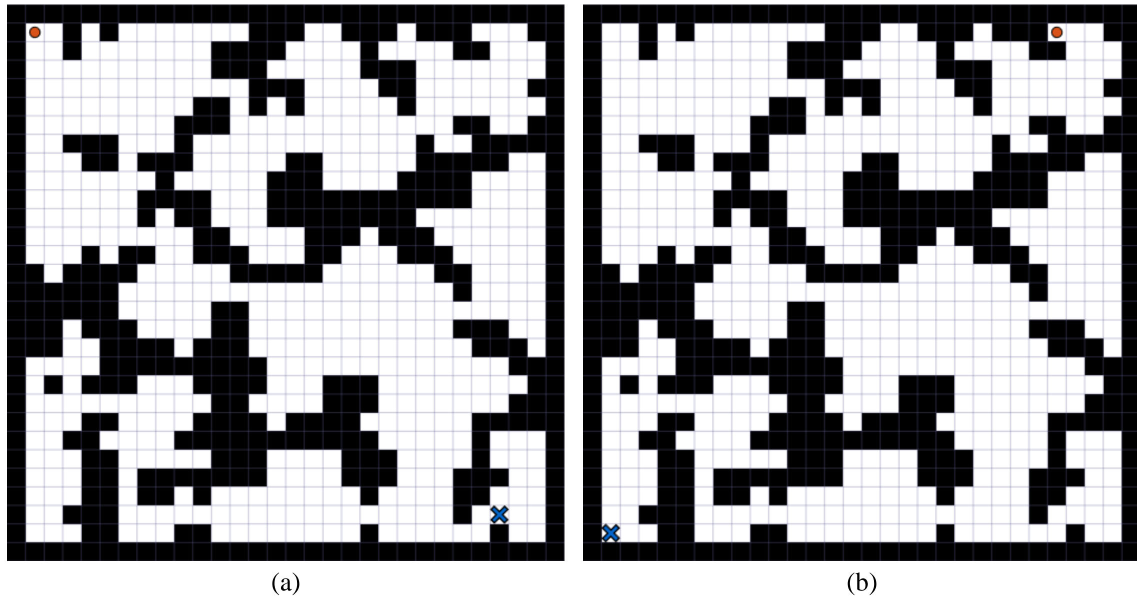


Figure 3.11 Examples of shortest path problems in a cavern environment using the tabu sampling approach (a) and the longest path approach (b). The agent is shown as a red circle and the goal is a blue cross.

In the traveling salesman problem, the agent must plan a minimum-cost route that visits a set of known waypoints. In some variants, the agent only needs to visit a certain number of waypoints. For this type of problem, we can again use tabu sampling to sample the desired number of waypoints and one additional point to use as the agent starting location. This ensures that the agent and all the waypoints meet some minimum separation distance. Depending on the specifics of the problem, we can choose to restrict the valid sampling area to one type of terrain, such as *meadow*. When using elevation as a feature, some interesting problems can be created by placing waypoints at extrema locations in the environment. Placing the agent in a relatively flat location at a middle elevation helps maximize the difference between the agent's possible choices, particularly if only some of the waypoints need to be visited. Figure 3.12 shows two environments with the TSP demonstrating these approaches.

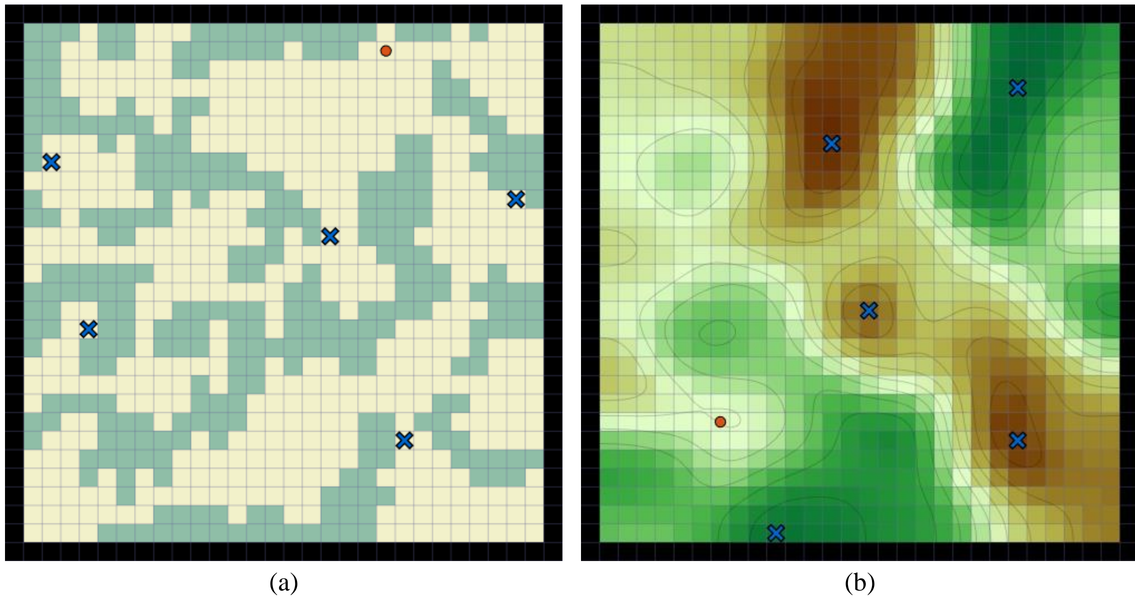

(a) (b)

Figure 3.12 Examples of traveling salesman problems initialized using the tabu sampling method in meadow terrain only (a) and using extrema locations in the elevation (b). The agent is a red circle and the waypoints are blue crosses.

The last problem type we consider is the traveling purchaser problem. In the CMM framework, the TPP is presented as a resource collecting problem. Various resource types are distributed throughout the environment and the agent is required to collect a specified number of each type. We use five resource types; one for each type of terrain in the full world environment. Just as with the TSP, tabu sampling is used to sample the resource locations, with each terrain type handled independently. For each type of terrain, we create a mask that leaves only grid cells of that terrain type and sample the resources using a separation distance of 10 for *meadow*, 4 for *forest*, 8 for *water*, 2 for *rock*, and 3 for *snow*. For *meadow* and *forest*, we sample resource locations until 100% of the feasible area has selected. For *water*, *rock*, and *snow*, we sample 50%, 5% and 20% respectively. Other values can be used, but these were found to create suitable problems for this work. The

agent location is sampled from an open location in the *meadow* terrain type. Figure 3.13 shows two examples of full world environments with the TPP.

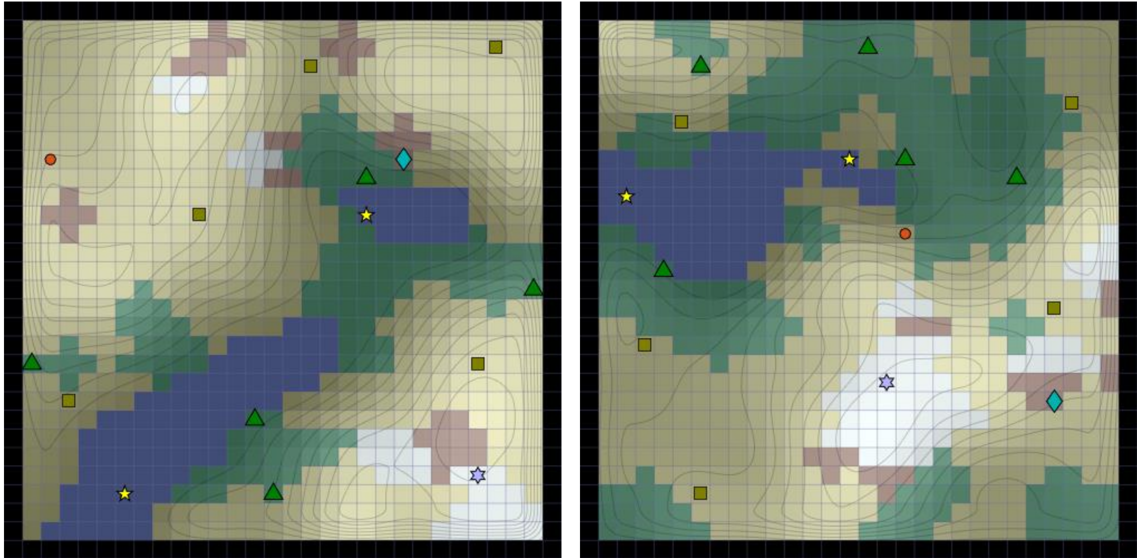


Figure 3.13 Examples of traveling purchaser problems in full world environments. The agent is a red circle and the other symbols represent different resource types. Each type of resource is restricted to a specific type of terrain.

3.7 Summary

This chapter described how the grid world problem environments are procedurally generated in the CMM framework. The environments are represented as multiple matrices representing the attributes of each grid cell, including the presence or absence of a wall, the type of terrain, the elevation, and the locations of any resources. Not all properties need to be defined, depending on the problem being studied. Maze-like environments can be created by only using the cave wall layer and discrete problems can be created by using only the terrain layer. Adding an elevation layer introduces a continuous feature that can lead to interesting agent strategies. Finally, the full world environment uses all of these

layers to simulate a synthetic real-world environment, although an agent need not consider all of the environment attributes when deciding where to go.

Each environment presents a problem for an agent to solve, designed as a resource gathering game. In the simplest problem type, there is a single resource (goal) placed somewhere in the environment for the agent to collect. The traveling salesman problem can be simulated by placing several resources of the same type throughout the environment. It may be worthwhile to consider a problem in which the agent only needs to collect one or a few resources, to observe how different destinations are compared. Finally, if the resource types are different, the problem is modeled as the traveling purchaser problem, in which the agent needs to collect a certain number of each resource. To decide which resources to pursue and the routes to take, the agent needs to develop a working model of the environment and understand the cost of each movement action. The next chapter introduces the mental map grid and defines the fundamental features that the agent can use to evaluate problems in the CMM framework.

4 THE MENTAL MAP GRID

The grid world environments defined in the previous chapter provide rich problem domains for studying agent movement and planning. The agent's actions are influenced by how the agent sees and interprets the environment. This chapter introduces the mental map that formalizes how the agent observes the environment and assigns costs to each movement action. These costs are represented as features in an action graph, which defines all possible actions that the agent can take. Later chapters summarize this information and use it to develop plans that guide the agent's movements.

4.1 The Mental Map

The CMM framework consists of two basic components: the simulation server and the agent. The server is responsible for defining the grid world environment using the procedural content generation methods presented in Chapter 3. The server also provides information about the environment to the agent in the form of observations. The knowledge that the agent accumulates through these observations is stored in a mental map representation of the environment. This mental map contains the only information that the agent can use to develop a plan that will solve the specified problem. Initially, the mental map is empty and represents complete uncertainty about the environment. As the agent moves, it discovers new regions and adds these to the mental map.

Formally, the server provides a grid world environment $\mathcal{E}$ consisting of several attribute layers as defined in Chapter 3. For each cell $c \in \mathcal{E}$, the following attributes are defined:

- $\text{OPEN}(c) \in \{0, 1\}$,
- $\text{TERRAIN}(c) \in \mathcal{T}$,
- $\text{ELEVATION}(c) \in [0,1]$, and
- $\text{RESOURCES}(c) \in \mathcal{R} \cup \emptyset$.

$\mathcal{T}$ is the set of all terrain types and $\mathcal{R}$ is the set of all resource types. In our examples, $\mathcal{T} = \{0, 1, 2, 3, 4, 5\}$ representing the terrain types *wall*, *meadow*, *forest*, *water*, *rock*, and *snow*. Likewise, $\mathcal{R} = \{0, 1, 2, 3, 4, 5\}$, where 0 indicates no resource and 1-5 indicate the resource that appears in the respective terrain type. In practice, these attribute layers are stored as $n \times m$ matrices $\mathcal{E}.W$, $\mathcal{E}.T$, $\mathcal{E}.E$, and $\mathcal{E}.R$ for the open, terrain, elevation, and resource attributes respectively.

The agent maintains an internal representation of the environment as a mental map $\mathcal{M}$. Each grid cell $c \in \mathcal{M}$ has the same attribute properties as $\mathcal{E}$ and one additional attribute, $\text{OBSERVED}(c) \in \{0,1\}$ (represented as the matrix V) indicating if the grid cell has been observed by the agent (1) or not (0). Initially, $\text{OBSERVED}(c) = 0$ for all $c \in \mathcal{M}$ and the other attributes are undefined. As the environment is revealed to the agent, $\text{OBSERVED}(c)$ is set to 1 for the grid cells that have been observed, and the other attributes are defined as equal to the corresponding values in $\mathcal{E}$. We assume that a grid cell is either fully observed, in which case all other attributes are defined, or completely unobserved, in which case the other attributes are undefined.

Algorithm 4.1 shows the INITIALIZE_MENTAL_MAP function used at the beginning of the simulation to initialize $\mathcal{M}$ with a specified size of $n \times m$, and with terrain types $\mathcal{T}$ and resource types $\mathcal{R}$. The size of the map is saved in $\mathcal{M}.size$ (line 2), the terrain and resource types are saved for later use (lines 3 and 4), and the other attributes are initialized to default values. The position of the agent $\mathcal{M}.pos$ is set to NIL (line 5) and a map of all visited grid cells is initialized to zero (line 6). The visibility matrix $\mathcal{M}.V$ is initialized to zero (line 7) and the other attribute layers $\mathcal{M}.W$, $\mathcal{M}.E$, $\mathcal{M}.T$, and $\mathcal{M}.R$ are initialized to NIL (lines 8-11). Lines 12 and 13 initialize the region labels $\mathcal{M}.L$ and the local region $\mathcal{M}.localRegion$. These are used to construct the region graph and will be discussed further in Chapter 5.

Algorithm 4.1 Initialize the Mental Map

INITIALIZE_MENTAL_MAP(n, m, $\mathcal{T}$, $\mathcal{R}$)
1: $\mathcal{M} \leftarrow$ empty structure
2: $\mathcal{M}.size \leftarrow (n, m)$
3: $\mathcal{M}.\mathcal{T} \leftarrow \mathcal{T}$
4: $\mathcal{M}.\mathcal{R} \leftarrow \mathcal{R}$
5: $\mathcal{M}.pos \leftarrow$ NIL
6: $\mathcal{M}.visited \leftarrow n \times m$ grid initalized to 0
7: $\mathcal{M}.V \leftarrow n \times m$ grid initalized to 0
8: $\mathcal{M}.W \leftarrow n \times m$ grid initalized to NIL
9: $\mathcal{M}.E \leftarrow n \times m$ grid initalized to NIL
10: $\mathcal{M}.T \leftarrow n \times m$ grid initalized to NIL
11: $\mathcal{M}.R \leftarrow n \times m$ grid initalized to NIL
12: $\mathcal{M}.L \leftarrow n \times m$ grid initalized to 1
13: $\mathcal{M}.localRegion \leftarrow n \times m$ grid initalized to 0
14: **return** $\mathcal{M}$

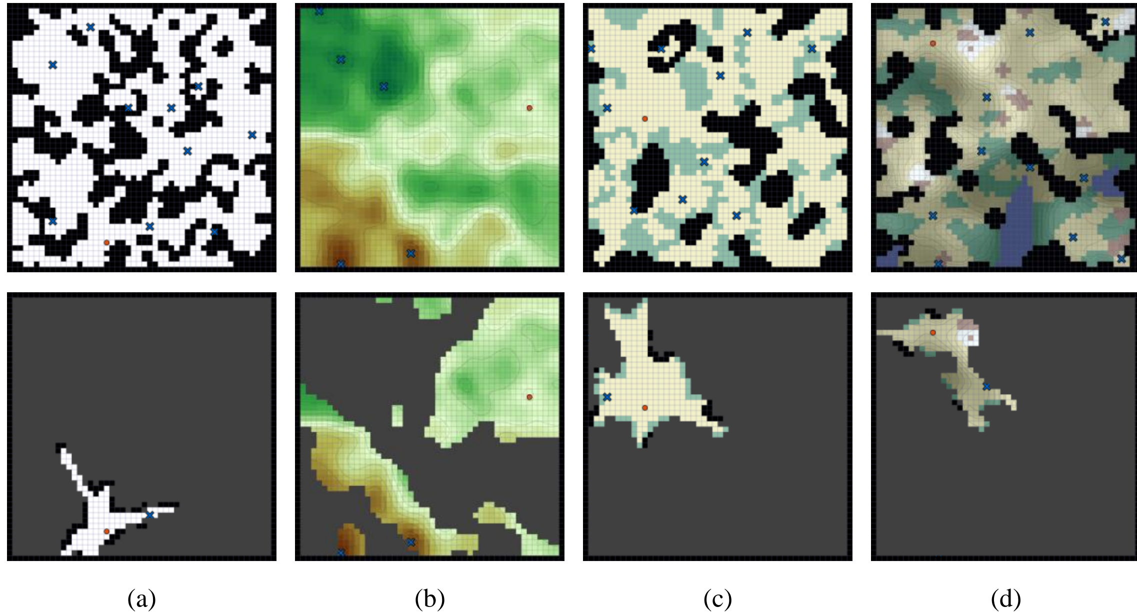
4.1.1 Creating Observations

At the start of the simulation and after each movement action by the agent, the server provides an observation of the environment to the agent. An observation $\mathcal{O}$ has the same form as the mental map $\mathcal{M}$, but contains only the immediately visible image of the environment. In contrast, $\mathcal{M}$ maintains a record of everything that has been seen since the start of the simulation and remembers what the environment looks like in places that are no longer visible. Since we assume that the environment does not change during the course of the simulation, portions of the mental map that have been observed but are no longer visible are considered to be accurate.

The observation data structure is assembled using Algorithm 4.2. The first step in this algorithm is to determine the visible region. There are two ways to do this that are controlled by the *opt.obsMode* parameter: using the line of sight viewshed method or declaring the entire environment to be visible. If we use the second method, then we can bypass the visibility computation altogether and declare all cells to be in the visible region, V (line 10). This is useful for studying problems without any uncertainty arising from visibility. Using the first method, the visible region is computed from the viewshed (lines 3-8), which will be discussed next. We return to the definition of the overall observation structure in Section 4.1.3. Figure 4.1 shows several examples of the observations computed at the current agent location in different environments.

Algorithm 4.2 Get Observation

```
GET_OBSERVATION(ℰ, a_i, a_j, opt)
1:  (n, m) ← size of ℰ

    /* Get the visible region */
2:  if opt.obsMode = "viewshed"
3:      E ← ℰ.E
4:      E[ℰ.W = 0 ∨ ℰ.T = 2] ← NIL
5:      E[a_i, a_j] ← ℰ.E[a_i, a_j]
6:      for opt.k iterations
7:          E ← E * G                          // Apply a 3×3 Gaussian blur
8:      V ← GET_VIEWSHED(E, a_j, a_i, opt.h)   // Algorithm 4.3
9:  else
10:     V ← n × m grid initalized to 1

11: V′ ← V ⊕ [0 1 0; 1 1 1; 0 1 0]            // 4-connected viewshed neighbors
12: V″ ← V ⊕ [1 1 1; 1 1 1; 1 1 1]            // 8-connected viewshed neighbors

13: V[V′ = 1 ∧ ℰ.T = 2] ← 1      // Mark adjacent forest cells as visible

    /* Get wall observation */
14: W ← n × m grid initalized to NIL
15: W[V = 1] ← 1
16: W[V″ = 1 ∧ ℰ.W = 0] ← 0
17: W[y, x] ← 0 for all cells where c_(y, x) is a border cell

    /* Create the observation structure */
18: 𝒪 ← empty structure
19: 𝒪.pos ← (a_i, a_j)
20: 𝒪.V ← [W = 0 ∨ W = 1]
21: 𝒪.W ← W
22: 𝒪.E ← ℰ.E
23: 𝒪.E [𝒪.V = 0] ← NIL
24: 𝒪.T ← ℰ.T
25: 𝒪.T [𝒪.V = 0] ← NIL
26: 𝒪.R ← ℰ.R
27: 𝒪.R [𝒪.V = 0] ← NIL
28: return 𝒪
```



(a) (b) (c) (d)

Figure 4.1 Examples of observations in various environments computed using Algorithm 4.2 and Algorithm 4.3. The top row shows the full environment $\mathcal{E}$ and the bottom row shows the observation $\mathcal{O}$ at the current agent location (shown as a red dot). The environment values are observed within the visible region and hidden everywhere else.

4.1.2 Viewshed Computation

If the server is configured to use the partially observable viewshed method, then the visible region V is computed using the GET_VIEWSHED function in Algorithm 4.3. The elevation map E that is sent to this function has cells that completely obstruct the line of sight, such as walls and forest terrain marked as NIL (lines 3-4 of Algorithm 4.2). The forest terrain in our model is designed to be traversable by the agent, but with restricted visibility, so these cells are marked as NIL in E. For the GET_VIEWSHED function to work properly, the current agent location should not be marked as NIL, so line 5 of Algorithm 4.2 copies the true elevation value at the agent location into E. The qualitative appearance of the viewshed region computed by GET_VIEWSHED can be improved by first applying

Gaussian smoothing to E. This removes elevation noise and results in a more continuous viewshed region that is less sensitive to the discretized heightmap. Our implementation convolves a 3×3 Gaussian filter G with the heightmap $\mathcal{E}.E$ for *opt.k* iterations, while maintaining the NIL values (lines 6-7). We found that repeated applications of small filter sizes produced more pleasing results than large filters when accounting for the NIL values. At this point, the smoothed elevaiton map E is passed to the GET_VIEWSHED function with the agent location (a_i, a_j) and the height parameter *opt.h* (line 8).

The GET_VIEWSHED function is given in Algorithm 4.3. It follows the basic premise of the R3 viewshed algorithm given in Algorithm 2.1, but is optimized to avoid computing the line of sight to every grid cell. The function starts by computing the elevation angle A from the current agent location to every grid cell, and copying the NIL flag for cells that are marked NIL in the elevation map (lines 1-7). Rather than evaluating the visibility of all grid cells at once, the function starts at the agent location and works outward. We initialize the visibility map V and a processed map P to all zeros (lines 8-9). The visibility map is then set to 1 at the current agent location and this cell is added to the current working set, C (lines 10-11). The algorithm then cycles through the main loop (lines 12-22) while the current working set is not empty. Lines 13-16 get the next set of cells to process, N, which are determined as the neighbors of C. Each cell in C is also marked with a 1 in the processed map, P. Lines 17-22 check the visibility of each cell in N that has not already been processed. The variable v in line 19 is computed using Algorithm 2.2 and is either 1 if the grid cell is visible from the agent location, 0 if it is not visible, or −1 if the line of sight encountered an obstruction marked as NIL. If $v = 1$, it is updated in the visibility map. If

$v \neq -1$, then the cell is added to C and its neighbors will be evaluated on the next iteration. If $v = -1$, then this cell will not be evaluated further. This allows the algorithm to stop looking in a direction that has a wall or forest cell that obstructs the line of sight regardless of the elevation.[1] The algorithm can only stop looking in a direction once it encounters such a cell, because there is always the possibility that a distant mountain ridge is visible beyond a hidden valley. Figure 4.1 shows several examples of the viewshed region computed in different environments.

[1] A more accurate visibility model might account for tree height in a forested region and allow the agent to look over a forest cell if the elevation permits. This would allow the agent to observe an entire forest region on a distant mountainside, for instance.

Algorithm 4.3 Get Viewshed

```
GET_VIEWSHED(E, x1, y1, h)

    /* Precompute the elevation angle to each grid cell */
1:  (n, m) ← size of E
2:  A ← n × m grid initalized to 0
3:  for each (x2, y2) ∈ {(x2, y2) | 1 ≤ y2 ≤ n ∧ 1 ≤ x2 ≤ m ∧ (x1, y1) ≠ (x2, y2)}
4:      if E[y2, x2] = NIL
5:          A[y2, x2] ← NIL
6:      else
7:          A[y2, x2] ← tan^-1( (E[y2,x2] − E[y1,x1] − h) / sqrt((x2 − x1)^2 + (y2 − y1)^2) )

8:  V ← n × m grid initalized to 0
9:  P ← n × m grid initalized to 0
10: V[y1, x1] ← 1
11: C ← {(x1, y1)}
12: while |C| > 0

        /* Determine the next set of cells to process */
13:     N ← ∅
14:     for each (x, y) ∈ C
15:         P[y, x] ← 1
16:         N ← N ∪ {(x–1, y), (x+1, y), (x, y–1), (x, y+1)}

        /* Check the visibility */
17:     C ← ∅
18:     for each (x, y) ∈ N s.t. P[y, x] = 0
19:         v ← CHECK_VISIBILITY(A, x1, y1, x, y)                    // Algorithm 2.2
20:         V[y, x] ← [v > 0]
21:         if v ≥ 0
22:             C ← C ∪ {(x, y)}

23: return V
```

Line 7: $A[y_2, x_2] \leftarrow \tan^{-1}\left(\frac{E[y_2,x_2]-E[y_1,x_1]-h}{\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}}\right)$

4.1.3 Finalizing the Observation

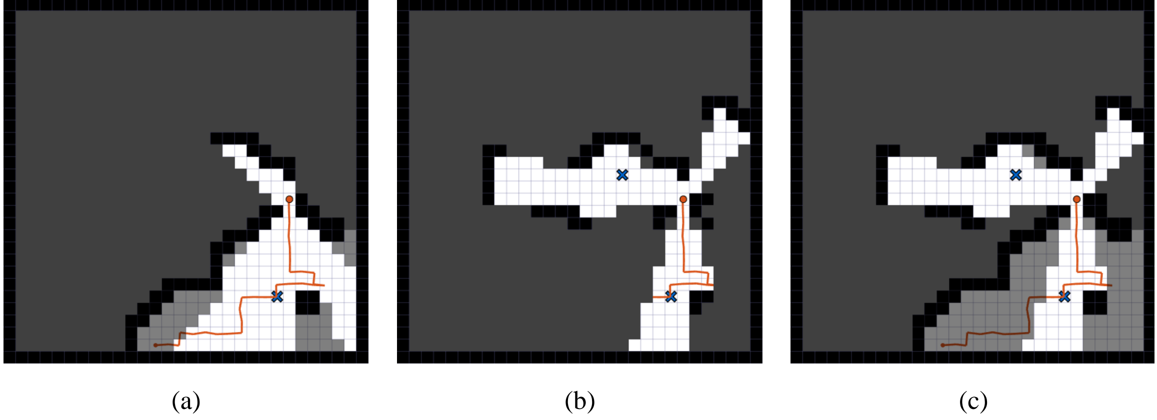
Returning to the GET_OBSERVATION function in Algorithm 4.2, we have now defined the visible region, V. We use this as a mask to define which cells have known attributes in the observation structure $\mathcal{O}$ and which cells are unobserved and marked as NIL. Some heuristics are used to update the knowledge in the observation for cells adjacent to the visible region. First, we define two dilations of V using both 4- and 8-connected neighbors (lines 11-12 of Algorithm 4.2). We want to make sure that the cells adjacent to the agent location are always visible, even in forest terrain, so line 13 marks these cells as visible in V. Next, we create the observation wall map W from the environment wall map $\mathcal{E}.W$ and the visible region. Line 14 initializes W as a grid with all cells marked as NIL to indicate complete uncertainty. Line 15 marks any cells in the visible region as being traversable open space in W. Because the edges of the visible region include cells that are adjacent to walls but not the walls themselves, we use the 8-connected neighbors of the visible region to identify which wall cells to include in the observation. Any cells in this expanded visible region that are marked as walls in the environment wall map are marked as walls in W (line 16). We also mark the border cells of the environment as walls to prevent the agent from moving off the edge of the map (line 17).

Lines 18-27 of Algorithm 4.2 create the actual observation structure $\mathcal{O}$. The observation consists of several information layers and the current agent position, which is stored as $\mathcal{O}.pos$ (line 19). The visible region of the observation $\mathcal{O}.V$ is defined as any cell that has been identified as either open space or a wall. Note that this may be somewhat different from the visible region computed using the GET_VIEWSHED function since we use additional heuristics to determine where the walls are. The observation wall map is saved

as $\mathcal{O}.W$ and the remaining attribute layers are saved using the values from the environment. Any cells that are not marked as visible in the observation visible region $\mathcal{O}.V$ are set to NIL in the elevation $\mathcal{O}.E$, terrain $\mathcal{O}.T$, and resource $\mathcal{O}.R$ attribute layers. The complete observation data structure $\mathcal{O}$ is returned on line 28.

4.1.4 Updating the Mental Map

After receiving an observation from the server, the agent identifies any new information and updates its mental map. An example is shown in Figure 4.2. Algorithm 4.4 gives the UPDATE_MENTAL_MAP procedure that takes an existing mental map structure $\mathcal{M}$ and an observation $\mathcal{O}$ and integrates the new information from $\mathcal{O}$ into $\mathcal{M}$. Lines 1-2 update the agent position and the map of visited locations. Line 3 identifies the grid cells that have new information, defined as cells that are visible in the observation but have not yet been observed in the mental map. These cells are saved as $\mathcal{M}.new$ and are used to avoid recomputing regions and features that have not changed since the last observation. The new cells are marked as observed in $\mathcal{M}.V$ (line 5) and the corresponding attribute values are copied from the observation to the mental map (lines 6-9). Line 10 applies some heuristics to the cave wall attribute layer to mark additional cells as walls based on inference rules. These are given in the CAVE_WALL_HEURISTICS function in Algorithm 4.5. After updating the known wall locations, line 11 updates the region labels to remove the label from any cell that is known to be a wall. The UPDATE_MENTAL_MAP procedure ends by returning the updated mental map structure on line 12.



(a) (b) (c)

Figure 4.2 Updating the mental map from an observation. (a) The agent's mental map before receiving the observation. The agent's path is shown in red, and the agent has just moved north into a cell that opens to the west. The previous observation is highlighted and previously observed cells are darkened. (b) The observation returned by the server at the agent's new position. (c) The agent's updated mental map after integrating the new information from the observation. The new observation is highlighted and previously observed cells are darkened.

Algorithm 4.4 Update Mental Map

UPDATE_MENTAL_MAP($\mathcal{M}$, $\mathcal{O}$)

1: $\mathcal{M}.pos \leftarrow \mathcal{O}.pos$
2: $\mathcal{M}.visited[\mathcal{M}.pos] \leftarrow 1$

3: $new \leftarrow \{(i, j) \mid \mathcal{O}.V[i, j] = 1 \wedge \mathcal{M}.V[i, j] = 0\}$
4: $\mathcal{M}.new \leftarrow new$
5: $\mathcal{M}.V[new] \leftarrow 1$
6: $\mathcal{M}.W[new] \leftarrow \mathcal{O}.W[new]$
7: $\mathcal{M}.E[new] \leftarrow \mathcal{O}.E[new]$
8: $\mathcal{M}.T[new] \leftarrow \mathcal{O}.T[new]$
9: $\mathcal{M}.R[new] \leftarrow \mathcal{O}.R[new]$

10: $\mathcal{M}.W \leftarrow$ CAVE_WALL_HEURISTICS($\mathcal{M}.W$) *// Algorithm 4.5*

11: $\mathcal{M}.L[\mathcal{M}.W = 0] \leftarrow 0$

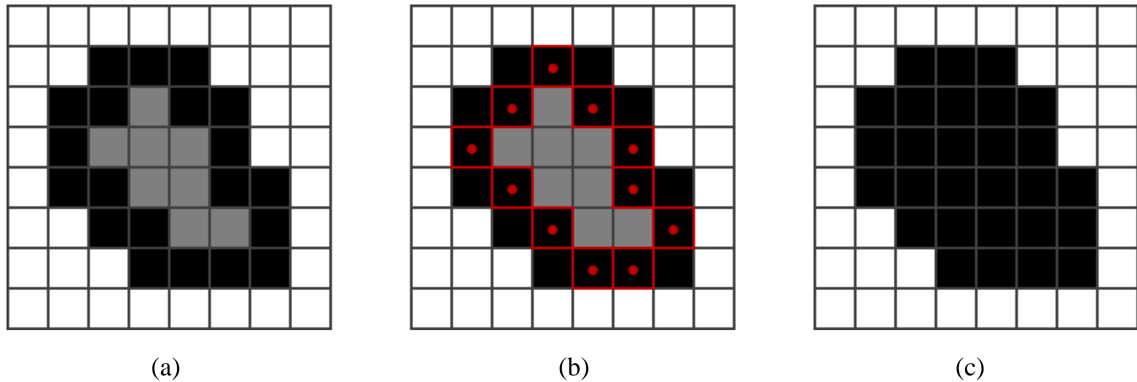
12: **return** $\mathcal{M}$

Algorithm 4.5 Cave Wall Heuristics

```
CAVE_WALL_HEURISTICS(W)

    /* Fill in unreachable areas */
1:  U ← {(i, j) | W[i, j] = NIL}                  // Get all unobserved cells
2:  L ← 4-connected component labeling of U
3:  for k in 1 to max(L)
4:      Z ← {(i, j) | L[i, j] = k}
5:      D ← Z ⊕ [0 1 0; 1 1 1; 0 1 0]              // Image dilation
6:      B ← {(i, j) | Z[i, j] = 0 ∧ D[i, j] = 1}   // Get the boundary cells
7:      if W[i, j] = 0 for all (i, j) ∈ B
8:          W[Z] ← 0

    /* Fix diagonals */
9:  (n, m) ← size of W
10: for each (i, j) ∈{(i, j) | 1 ≤ i ≤ n−1 ∧ 1 ≤ j ≤ m−1}
11:     if W[i, j] = 0 ∧ W[i+1, j+1] = 0
12:         if W[i+1, j] = 1
13:             W[i, j+1] ← 0
14:         if W[i, j+1] = 1
15:             W[i+1, j] ← 0
16:     if W[i, j+1] = 0 ∧ W[i+1, j] = 0
17:         if W[i, j] = 1
18:             W[i+1, j+1] ← 0
19:         if W[i+1, j+1] = 1
20:             W[i, j] ← 0

21: return W
```

There are two main heuristics applied by the CAVE_WALL_HEURISTICS function in Algorithm 4.5. The first seeks to fill in unreachable areas that have been surrounded completely by walls such as the example in Figure 4.3. These cells are inaccessible to the agent, and are therefore assumed to be wall cells. Line 1 of the function identifies all unobserved wall cells that are currently labeled as NIL. The 4-connected components are

identified on line 2, and for each one we apply an image dilation to identify the boundary cells of this unobserved region (lines 4-6). If all the boundary cells are marked as walls, then there is no way for an agent to access the cells in the unobserved region, so they are marked as walls (lines 7-8).



(a) (b) (c)

Figure 4.3 Filling in unreachable areas with walls. (a) An unobserved region (gray) is surrounded by wall cells (black) and is inaccessible to the agent. (b) The boundary cells (marked with red dots) of the unobserved region are checked and if they are all walls, then the unobserved region is filled in. (c) The filled in region.

The second heuristic is used to improve the wall boundary on diagonal edges. Because Algorithm 3.3 removed any diagonal passages during the generation of the cave wall map, the agent can assume that there will be no diagonal passages in the environment. This means that if two diagonally adjacent grid cells are both observed to be wall cells and one of the two cells between them is observed to be open, then the other cell must be a wall to prevent the creation of a diagonal passage. Lines 9-20 apply this rule to the entire map and mark cells that meet these criteria as walls. An example is shown in Figure 4.4.

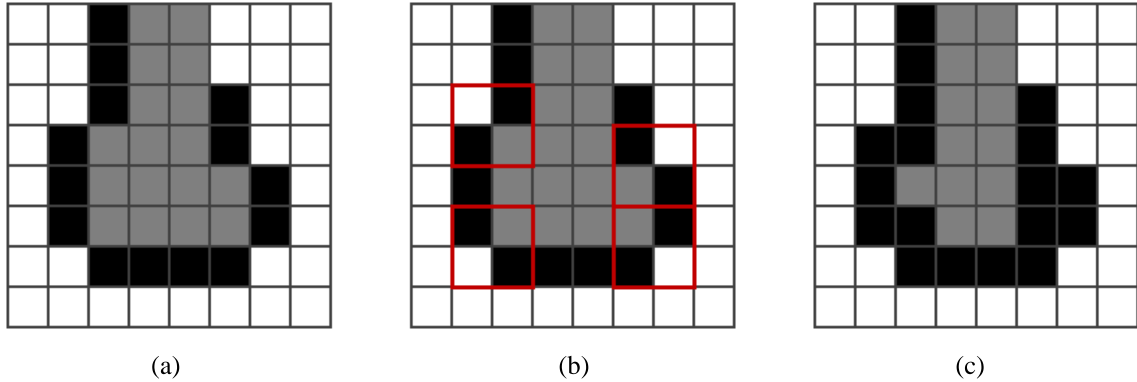


Figure 4.4 Fixing diagonal boundaries. (a) The wall layer of a mental map before fixing diagonals. Open space is white, walls are black, and the unobserved cells are gray. (b) Locations that match the pattern of diagonally adjacent wall cells with one open and one unobserved cell in between are marked in red. (c) The unobserved cells in these patterns are marked as walls.

4.2 The Action Graph

The mental map structure $\mathcal{M}$ is stored internally as a set of raster image layers, representing the agent's knowledge of the environment at each grid cell location. However, for planning future actions, it is useful to represent the mental map as an attributed weighted graph. We begin by defining the action graph G_A that is the most granular representation of the knowledge stored in $\mathcal{M}$. In Chapter 5 we will introduce the region graph, which summarizes the information in G_A for distinct regions in the environment. Each vertex in G_A represents a grid cell where the agent can be located and edges represent the movement actions between adjacent grid cells. The vertex set is defined as $V(G_A) = \{c \in \mathcal{M} \mid \textsc{Open}(c) = 1 \vee \textsc{Observed}(c) = 0\}$, which represents every grid cell that has either been observed to be traversable, or has not yet been observed and therefore may be traversable. Each vertex $v \in V(G_A)$ inherits the attributes of the grid cell associated with it. Adjacent grid cells are connected by a directed edge $e = (u, v)$ where $u, v \in V(G_A)$ and

the cell represented by u is adjacent to the cell represented by v. We denote $\text{START}(e) = u$ as the starting vertex and $\text{END}(e) = v$ as the ending vertex of edge e. The set of all edges in the graph forms the edge set $E(G_A)$. Figure 4.5 shows the action graphs computed from mental maps of two example environments.

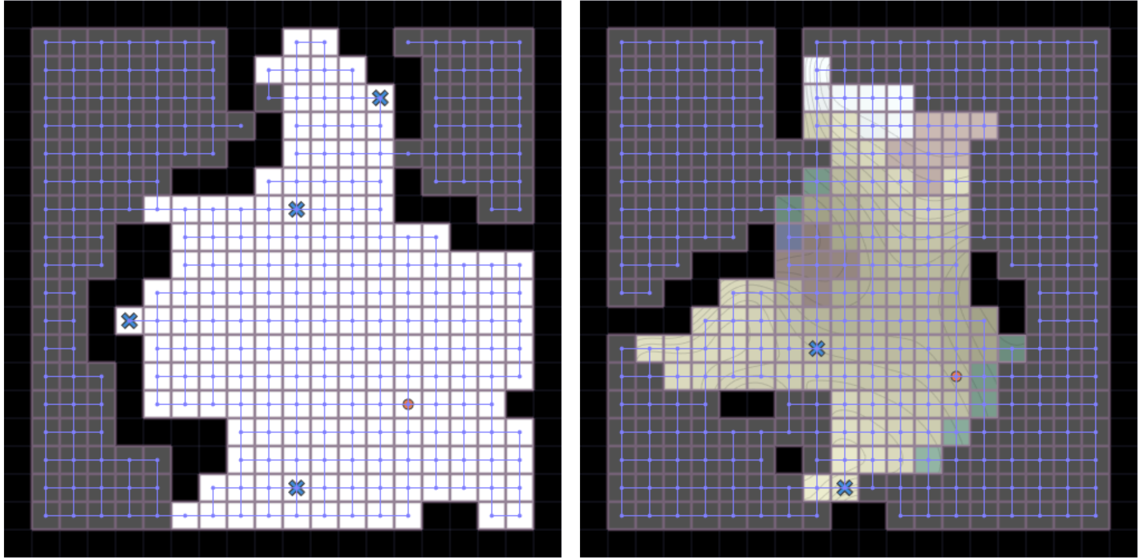


Figure 4.5 Examples of the action graph for two mental maps. All grid cells that are not walls are included as vertices, regardless of observability. Adjacent grid cells are connected with edges.

Consider a single edge $e = (u, v) \in E(G_A)$ where u represents grid cell c_1 and v represents an adjacent grid cell c_2. If a grid cell is observed, then its attributes are known; otherwise the attributes are marked as NIL. We denote the terrain type of cell c_k as t_k, the elevation (height) as h_k, and the observability as o_k, where $k \in \{1, 2\}$. (Note that we use h to represent elevation to avoid confusion with the notation for graph edges.)

$$o_k = \text{OBSERVED}(c_k) \tag{4.1}$$

$$t_k = \begin{cases} \text{TERRAIN}(c_k), & o_k = 1 \\ \text{NIL}, & o_k = 0 \end{cases} \tag{4.2}$$

$$h_k = \begin{cases} \text{ELEVATION}(c_k), & o_k = 1 \\ \text{NIL}, & o_k = 0 \end{cases} \tag{4.3}$$

The attributes of grid cell c_k are defined as the pair (t_k, h_k) and the attributes of an edge can be written as the pair of pairs, $\big((t_1, h_1), (t_2, h_2)\big)$. For notational convenience, we define $e.t_1$ and $e.t_2$ as the starting and ending terrain types of an edge, $e.h_1$ and $e.h_2$ as the starting and ending elevations, and $e.o_1$ and $e.o_2$ as the observability of the starting and ending cells. We occasionally drop the e prefix when referring to only a single edge. The terrain, elevation, and observability of c_1 and c_2 are used to define several features $f(e)$ for an edge e. Multiple edge features are combined into a feature vector $\mathbf{f}(e) = \big(f_1(e), \dots, f_m(e)\big)$, where $f_i: e \mapsto \mathbb{R}_{\geq 0}$ for all $i = 1, \dots, m$. We assume that each feature maps into a non-negative real number to aid in the formulation of agent objective functions, which will be defined so that edge features are minimized. The next two sections give several possible feature functions that an agent can use to define $\mathbf{f}(e)$. We first consider the case in which both grid cells are observed, resulting in a crisp feature vector with no uncertainty. Then, we show how these features become fuzzy when one or both grid cells are unobserved.

4.3 Crisp Feature Functions

In the case where both grid cells of an edge e are observed (i.e. $e.o_1 = e.o_2 = 1$), the attributes are known exactly and the resulting feature vector contains no uncertainty. The following subsections define several crisp features for action graph edges. An edge feature $f(e)$ may depend only on some of the cell attributes, so for notational clarity, only

the required arguments are included in the following feature definitions. An example showing the computation of these features is given in Section 4.3.6.

4.3.1 Distance Feature

The simplest feature we consider is a basic measure of the distance the agent has traveled. We denote this feature as f_d. For a single edge in a uniform grid, the distance feature is defined as a constant value,

$$f_d = 1. \tag{4.4}$$

The distance feature is independent of the grid cell attributes and can be applied in any environment.

4.3.2 Terrain Type Features

In environments with multiple types of terrain, we can define a separate feature for each terrain type. These features indicate how much of each type of terrain is represented by an edge. For terrain type i, we denote this feature as $f_{t(i)}$, defined as

$$f_{t(i)}(t_1, t_2) = \begin{cases} 0, & t_1 \neq i \wedge t_2 \neq i \\ 0.5, & t_1 \neq i \wedge t_2 = i \\ 0.5, & t_1 = i \wedge t_2 \neq i \\ 1, & t_1 = i \wedge t_2 = i\,. \end{cases} \tag{4.5}$$

There are four possible cases considered, where each cell either is or is not of terrain type i. Equation 4.5 can be expressed more compactly as

$$f_{t(i)}(t_1, t_2) = \frac{1}{2}[t_1 = i] + \frac{1}{2}[t_2 = i], \tag{4.6}$$

where the notation $[*]$ evaluates to 1 if the condition in the brackets is true and 0 otherwise. A terrain type feature is computed for each terrain type $i \in \mathcal{T}$. The feature will be 1 if both

grid cells are of type i and 0 if neither grid cell is of type i. If only one grid cell is of type i, the feature will evaluate to 0.5.

4.3.3 Terrain Transition Features

In some circumstances, it may be important for the agent to consider the transition between different types of terrain (e.g. when getting into or out of a boat at the edge of a lake). For these types of features, we define a transition matrix $\mathrm{T} \in \{0,1\}^{|\mathcal{T}| \times |\mathcal{T}|}$ where t_{ij} is 1 if the transition between two grid cells is from terrain type i to terrain type j and 0 otherwise. If the direction of the transition is important, we can use each element of T as a separate feature. We denote these directional terrain transition features as $f_{t\langle i,j\rangle}$ and define them formally as

$$f_{t\langle i,j\rangle}(t_1, t_2) = \begin{cases} 1, & t_1 = i \wedge t_2 = j \\ 0, & \text{otherwise} \end{cases} . \tag{4.7}$$

This can also be written as

$$f_{t\langle i,j\rangle}(t_1, t_2) = [(t_1 = i \ \wedge t_2 = j)]. \tag{4.8}$$

If the direction of the transition is unimportant, we can reduce the number of features by accounting for symmetries. The symmetric terrain transition features are denoted as $f_{t\{i,j\}}$ where we assume that $i \leq j$ and they are defined as

$$f_{t\{i,j\}}(t_1, t_2) = \begin{cases} 1, & t_1 = i \wedge t_2 = j \\ 1, & t_1 = j \wedge t_2 = i \\ 0, & \text{otherwise} \end{cases} . \tag{4.9}$$

Again, this can be written in square bracket notation as

$$f_{t\{i,j\}}(t_1, t_2) = [(t_1 = i \ \wedge t_2 = j) \ \vee (t_1 = j \ \wedge t_2 = i)]. \tag{4.10}$$

Table 4.1 shows the difference between the terrain type features and the terrain transition features for all possible combinations of terrain types for the two cells.

Table 4.1 Crisp terrain type and terrain transition features

t_1	t_2	$f_{t(i)}$	$f_{t(j)}$	$f_{t\{i,i\}}$	$f_{t\{i,j\}}$	$f_{t\{j,j\}}$	$f_{t\langle i,i\rangle}$	$f_{t\langle i,j\rangle}$	$f_{t\langle j,i\rangle}$	$f_{t\langle j,j\rangle}$
i	i	1	0	1	0	0	1	0	0	0
i	j	0.5	0.5	0	1	0	0	1	0	0
i	$\neg(i \vee j)$	0.5	0	0	0	0	0	0	0	0
j	i	0.5	0.5	0	1	0	0	0	1	0
j	j	0	1	0	0	1	0	0	0	1
j	$\neg(i \vee j)$	0	0.5	0	0	0	0	0	0	0
$\neg(i \vee j)$	i	0.5	0	0	0	0	0	0	0	0
$\neg(i \vee j)$	j	0	0.5	0	0	0	0	0	0	0
$\neg(i \vee j)$	$\neg(i \vee j)$	0	0	0	0	0	0	0	0	0

Note that we define a separate directional terrain transition feature for each terrain type pair (i, j) and (j, i) whereas we only need to define the symmetric terrain transition feature for the pair (i, j) where $i \leq j$. Also, note that the self-transition features $f_{t\{i,i\}}$ and $f_{t\langle i,i\rangle}$ are not quite the same as the terrain type feature $f_{t(i)}$ since the terrain transition features can only take binary values. However, an agent need only use one of the terrain-based feature sets because the same information is simply distributed across a different number of features. For N terrain types, there are N terrain type features, N^2 directional terrain transition features, and $\frac{N^2+N}{2}$ symmetrical terrain transition features.

4.3.4 Elevation Features

Whereas terrain is a discrete feature type, the difference in elevation between two grid cells is a continuous feature domain. Recall that the elevation values of the starting

and ending edge cells are given as h_1 and h_2. The absolute difference in elevation for the edge is a feature that we denote as f_h and define as

$$f_h(h_1, h_2) = |h_1 - h_2|. \tag{4.11}$$

Often, an agent will want to differentiate between an uphill slope and a downhill slope. To account for this, we define the uphill slope feature $f_{h\uparrow}$ as

$$f_{h\uparrow}(h_1, h_2) = \max(0, h_2 - h_1), \tag{4.12}$$

And the downhill slope feature $f_{h\downarrow}$ as

$$f_{h\downarrow}(h_1, h_2) = \max(0, h_1 - h_2). \tag{4.13}$$

Note that the features are always non-negative to ensure that the objective values never go below zero. The uphill and downhill slope features are complementary and at least one of them will always be zero. The absolute elevation difference feature represents a combination of the two directional elevation difference features, so an agent will usually only use either just f_h or the pair $f_{h\uparrow}$ and $f_{h\downarrow}$. Figure 4.6 shows plots of the elevation difference features for all values of h_1 and h_2 within the allowed range of $[0, 1]$.

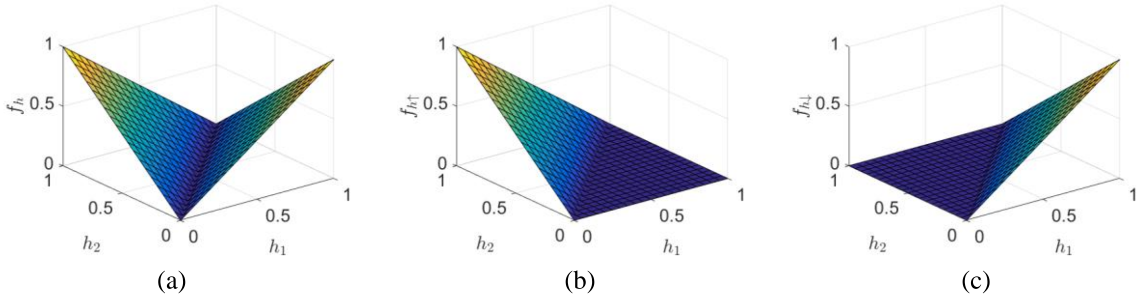


Figure 4.6 Plots of the elevation difference features. (a) The absolute elevation difference f_h. (b) the uphill elevation difference $f_{h\uparrow}$. (c) The downhill elevation difference $f_{h\downarrow}$.

4.3.5 Other Features

We limit our study in this work to the above features, but this list is by no means exhaustive. Many different problems can be expressed in this framework so long as it is possible to compute a feature for each edge based only on the attributes of its vertices. For instance, one could compute additional features in the environment such as proximity to a wall or the amount of terrain that is visible from a grid cell and develop edge features based on these environment attributes. We should mention that features that depend on multiple edges, such as the curviness or uniqueness of a path might be more difficult to use in this framework. These types of features would be more suitably defined over paths rather than individual edges, which would require different agent strategies than the ones presented here.

4.3.6 Example

Figure 4.7 shows four examples of the edge features computed between two grid cells. Each example shows a pair of cells representing a single edge going from the left cell to the right cell. The colors indicate the terrain type with light tan representing terrain type 1 (meadow) and green representing terrain type 2 (forest). The numbers inside each cell indicate the elevation, and the computed features are shown below each example.

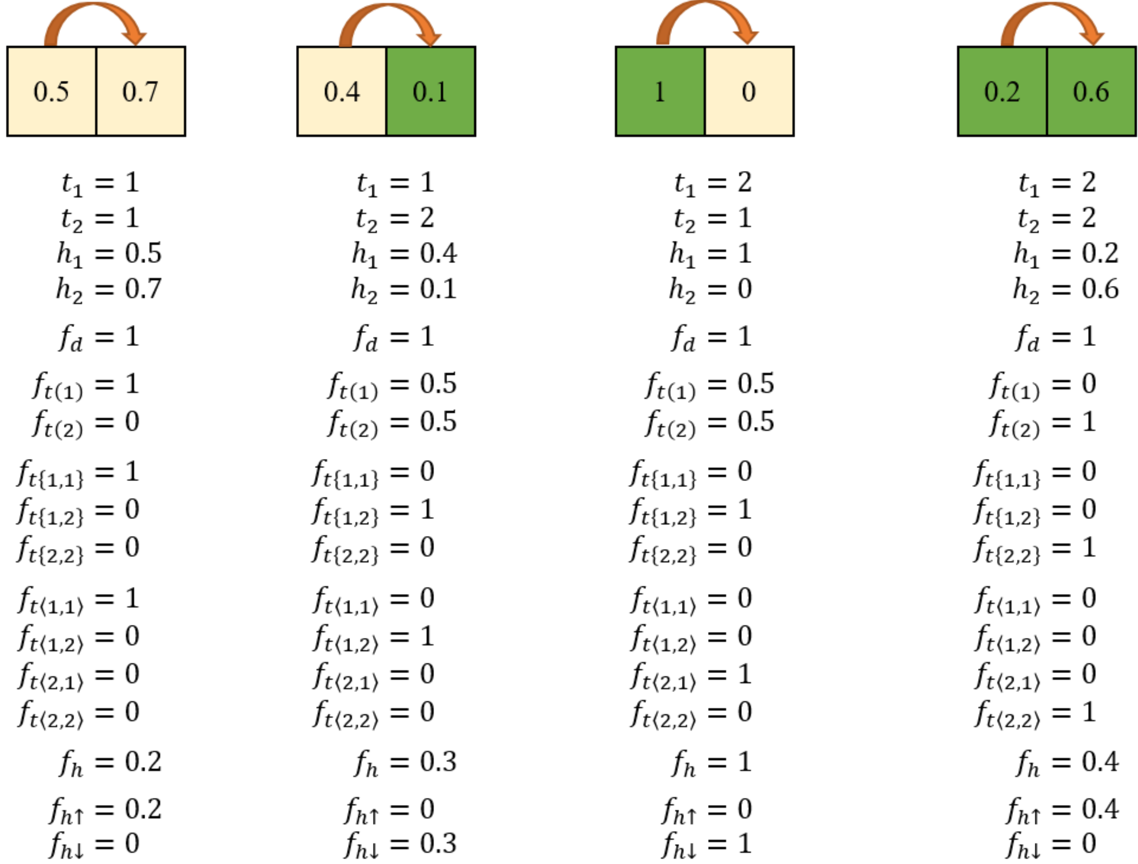


Figure 4.7 Four examples demonstrating the computation of the feature functions considered in this work for a single transition between two grid cells. The light tan region represents terrain type 1 (meadow) and the green region represents terrain type 2 (forest). The numbers in each cell indicate the elevation value.

The distance feature f_d is a constant 1 for each of the examples. The terrain type features $\left(f_{t(1)}, f_{t(2)}\right)$ are either (1,0), (0,1), or (0.5, 0.5) depending on if the pair of cells is all of terrain type 1, 2, or both. The symmetric terrain transition features $\left(f_{\{1,1\}}, f_{\{1,2\}}, f_{\{2,2\}}\right)$ are either (1,0,0), (0,1,0), or (0,0,1), with the single nonzero element indicating which pair of terrain types is present in each example. Likewise, the directional terrain transition features $\left(f_{t\langle 1,1\rangle}, f_{t\langle 1,2\rangle}, f_{t\langle 2,1\rangle}, f_{t\langle 2,2\rangle}\right)$ each have a single nonzero element that indicates the appropriate configuration of the two terrain types. The absolute elevation difference feature f_h is simply the absolute difference in elevation between the two grid

cells. The directional elevation difference features $(f_{h\uparrow}, f_{h\downarrow})$ indicate the direction of the slope and have one element equal to f_h and the other set to zero. An agent could use various subsets of these features to define its objective functions, which will be discussed further in Chapter 6.

4.4 Fuzzy Feature Functions

We now consider the case in which one or both grid cells of an edge e are unobserved (i.e. $e.o_1 = 0$ and/or $e.o_2 = 0$). When this occurs, we introduce uncertainty into the feature vector, which now needs to capture the range and distribution of possible feature values given the unobserved attributes. Fuzzy numbers are well-suited for this task, as they can represent a range of values with different weights specified by a membership function. A fuzzy number $A \subseteq \mathbb{R}$ is a normalized convex fuzzy set with a membership function $\mu_A: A \to [0, 1]$ that specifies how well a number $x \in A$ is represented by A. We use triangular fuzzy numbers in this work for their relative simplicity. For a value $x \in \mathbb{R}$, the membership function of a triangular fuzzy number $A = \mathrm{Tri}(a, b, c)$ is defined as

$$\mu_A(x; a, b, c) = \begin{cases} 0, & x \leq a \\ \dfrac{x-a}{b-a}, & a < x < b \\ 1, & x = b \\ \dfrac{c-x}{c-b}, & b < x < c \\ 0, & x \geq c \end{cases} . \tag{4.14}$$

Whereas a crisp feature function $f(e)$ only needs to define a single value, a fuzzy feature function $\tilde{f}(e)$ needs to define the three control parameters for the triangular fuzzy number. A natural interpretation of these parameters is the min, mean, and max values that the corresponding crisp feature function could take if the hidden grid cells were observed.

Using the notation from Equation 4.14, we denote these as $a = \tilde{f}^{\min}(e)$, $b = \tilde{f}^{\text{mean}}(e)$, and $c = \tilde{f}^{\max}(e)$. The following sections define these values for each of the feature functions considered in this work.

4.4.1 Distance Feature

Unlike the other features, the distance feature for a single edge is unaffected by observability. Because all edges in the action graph have the same length, the distance feature is defined as a crisp value of 1, regardless of whether the grid cells are observed or not.

$$\tilde{f}_d^{\min}(e) = \tilde{f}_d^{\text{mean}}(e) = \tilde{f}_d^{\max}(e) = 1. \tag{4.15}$$

The resulting fuzzy feature is defined as

$$\tilde{f}_d(e) = \text{Tri}(1, 1, 1). \tag{4.16}$$

4.4.2 Terrain Type Features

The terrain type features measure the amount of an edge that occurs within terrain type i. In the crisp case, the possible values are 0, 0.5, and 1, indicating that neither, one, or both grid cells were of type i. In the fuzzy case, we need to consider what the minimum, maximum, and expected values of the crisp feature would be over all possible configurations of the unknown terrain types. We start by defining some additional notation.

Let $t_1^*, t_2^* \in \mathcal{T}$ be the true terrain types of the starting and ending cells respectively. The value of the terrain type feature in the fully observable case is given by Equation 4.5 or Equation 4.6 as $f_{t(i)}(t_1^*, t_2^*)$. When either t_1^* or t_2^* is unknown, this value cannot be

evaluated directly, but we can determine the range and most likely value. Let T_{ki} be the event that $t_k^* = i$ for $k \in \{1, 2\}$. The probability that event T_{ki} occurs is defined as

$$p(T_{ki}) = \begin{cases} 1, & o_k = 1 \wedge t_k = i \\ 0, & o_k = 1 \wedge t_k \neq i \\ p(i), & o_k = 0 \end{cases}, \tag{4.17}$$

where $p(i)$ is the prior likelihood of observing terrain type i. Often, $p(i) = \frac{1}{|\mathcal{T}|}$, where $|\mathcal{T}|$ is the number of possible terrain types, but other priors are possible. The complementary event T_{ki}^c is defined as the event that $t_k^* \neq i$ for $k \in \{1, 2\}$. Since these are the only two possible events describing the state of a single cell,

$$p(T_{ki}^c) = 1 - p(T_{ki}). \tag{4.18}$$

For the two cells involved in a graph edge, there are four possible states that need to be considered:

- $s_{12} = (T_{1i}, T_{2i})$, (4.19)
- $s_1 = (T_{1i}, T_{2i}^c)$, (4.20)
- $s_2 = (T_{1i}^c, T_{2i})$, and (4.21)
- $s_0 = (T_{1i}^c, T_{2i}^c)$. (4.22)

Here, s_{12} is the state where both grid cells have terrain type i, s_0 is the state where neither cell has terrain i, and s_1 and s_2 are the states where just one of the cells is of terrain type i. We call the set of all possible states $S = \{s_{12}, s_1, s_2, s_0\}$, and each of these states results in a crisp terrain type feature vector for the edge. Adapting Equation 4.5 gives

$$f_{t(i)}(s) = \begin{cases} 0, & s = s_0 \\ 0.5, & s = s_1 \vee s = s_2 \\ 1, & s = s_{12} \end{cases}. \tag{4.23}$$

We assume that the terrain types of the two cells are independent, so the following expressions give the probability that each state is the true state of the environment.

$$p(s_{12}) = p(T_{1i})p(T_{2i}) \tag{4.24}$$

$$p(s_1) = p(T_{1i})\big(1 - p(T_{2i})\big) \tag{4.25}$$

$$p(s_2) = \big(1 - p(T_{1i})\big)p(T_{2i}) \tag{4.26}$$

$$p(s_0) = \big(1 - p(T_{1i})\big)\big(1 - p(T_{2i})\big) \tag{4.27}$$

If the probability of a state is greater than zero, then it has some chance of occurring. We define the possibility that a state occurs as

$$\text{pos}(s) = [p(s) > 0], \tag{4.28}$$

and the set of all possible states is given as

$$S_{\text{pos}} = \{s \in S | \text{pos}(s) > 0\}. \tag{4.29}$$

We can now express the minimum, maximum, and expected values of the terrain type feature for an edge e and terrain type i.

$$\tilde{f}_{t(i)}^{\min}(e) = \min_{s \in S_{\text{pos}}} f(s) \tag{4.30}$$

$$\tilde{f}_{t(i)}^{\max}(e) = \max_{s \in S_{\text{pos}}} f(s) \tag{4.31}$$

$$\tilde{f}_{t(i)}^{\text{mean}}(e) = \sum_{s \in S} f(s)p(s) \tag{4.32}$$

In practice, the fuzzy terrain type feature values are computed using the following equivalent definitions.

$$\tilde{f}_{t(i)}^{\min}(e) = \frac{1}{2}[t_1 = i \wedge o_1 = 1] + \frac{1}{2}[t_2 = i \wedge o_1 = 1] \tag{4.33}$$

$$\tilde{f}_{t(i)}^{\max}(e) = \frac{1}{2}[t_1 = i \vee o_1 = 0] + \frac{1}{2}[t_2 = i \vee o_2 = 0] \tag{4.34}$$

$$\begin{aligned} \tilde{f}_{t(i)}^{\text{mean}}(e) = \frac{1}{2}[t_1 = i \wedge o_1 = 1] + \frac{1}{2}p(i)[o_1 = 0] + \\ \frac{1}{2}[t_2 = i \wedge o_2 = 1] + \frac{1}{2}p(i)[o_2 = 0] \end{aligned} \tag{4.35}$$

The fuzzy number representing the overall terrain type feature is defined as

$$\tilde{f}_{t(i)}(e) = \text{Tri}\left(\tilde{f}_{t(i)}^{\min}(e),\ \tilde{f}_{t(i)}^{\text{mean}}(e),\ \tilde{f}_{t(i)}^{\max}(e)\right). \tag{4.36}$$

A summary of the triangular fuzzy number feature values is given in Table 4.2 for the case where $|\mathcal{T}| = 2$ and $p(i) = 0.5$. Note that when both grid cells are observed, the fuzzy numbers are equivalent to the crisp version.

Table 4.2 Example of the fuzzy terrain type feature when $|\mathcal{T}| = 2$ and $p(i) = 0.5$

$\tilde{f}_{t(i)}(e)$		$o_2 = 1$		$o_2 = 0$
		$t_2 = i$	$t_2 \neq i$	
$o_1 = 1$	$t_1 = i$	Tri(1,1,1)	Tri(0.5, 0.5, 0.5)	Tri(0.5, 0.75, 1)
	$t_1 \neq i$	Tri(0.5, 0.5, 0.5)	Tri(0, 0, 0)	Tri(0, 0.25, 0.5)
$o_1 = 0$		Tri(0.5, 0.75, 1)	Tri(0, 0.25, 0.5)	Tri(0, 0.5, 1)

To demonstrate, consider the examples in Figure 4.8. In (a), both cells are observed, so the fuzzy terrain type feature value is equivalent to the crisp case and the fuzzy number is a singleton value of 0.5 for both terrain type 1 (meadow) and 2 (forest). In (b), the first cell is observed to be terrain type 1 and the second cell is unobserved. Both terrain types have equal priors, so the second grid cell is equally likely to be either terrain type. If the second cell is type 1, then the crisp terrain type features would be $f_{t(1)} = 1$ and $f_{t(2)} = 0$.

If the second cell is type 2, then they would be $f_{t(1)} = 0.5$ and $f_{t(2)} = 0.5$. Clearly, the minimum value of $f_{t(1)}$ is 0.5 and the maximum is 1. Likewise, the minimum of $f_{t(2)}$ is 0 and the maximum is 0.5. Since the priors are equal, the mean values are the averages of these two extremes. In (c), the first grid cell is unobserved and the priors favor terrain type 1. The resulting triangular fuzzy numbers are skewed to reflect that the most likely outcome is that the unobserved cell is type 1 and the resulting crisp feature value would be 0.5. In (d), both cells are unobserved, so both terrain type features span the entire range [0, 1]. Because the priors slightly favor terrain type 1, the mean value of $\tilde{f}_{t(1)}$ is slightly higher than that of $\tilde{f}_{t(2)}$.

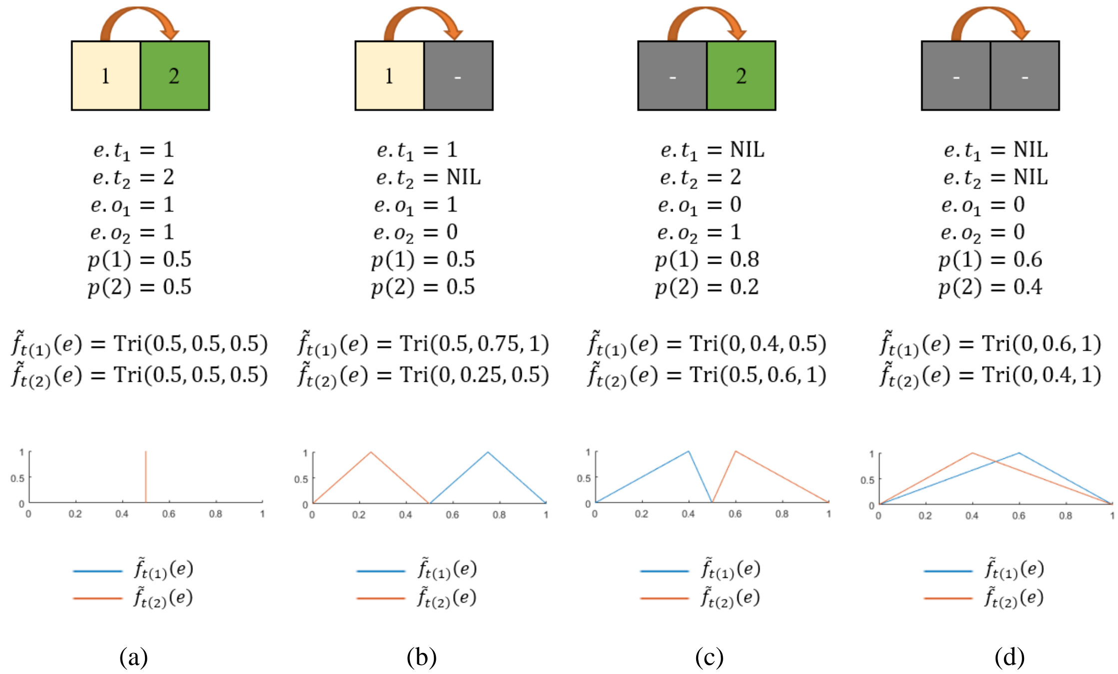


Figure 4.8 Four examples demonstrating the computation of the fuzzy terrain type features for a single transition between two adjacent grid cells. The light tan region represents terrain type 1 (meadow) and the green region represents terrain type 2 (forest). Gray cells are unobserved. The plots compare the fuzzy terrain type features for each example.

4.4.3 Terrain Transition Features

The directional terrain transition feature $\tilde{f}_{t\langle i,j\rangle}(e)$ indicates if an edge e goes from terrain type i to terrain type j, whereas the symmetric terrain transition feature $\tilde{f}_{t\{i,j\}}(e)$ only checks if an edge e includes both terrain types i and j. In the fully observable case, these features could only take binary values. However, in the partially observable case, the terrain transition features are represented as triangular fuzzy numbers. Following the notation from the previous section, let $t_1^*, t_2^* \in \mathcal{T}$ be the true terrain types of the starting and ending grid cells for an edge e, and let T_{ki} be the event that $t_k^* = i$ and T_{kj} the event that $t_k^* = j$ for $k \in \{1, 2\}$. Equation 4.17 gives the probability of each event as $p(T_{ki})$ and $p(T_{kj})$. Note that the terrain priors $p(i)$ and $p(j)$ may be different, but the terrain priors for all terrain types must satisfy the requirements of a multinomial probability distribution (i.e. $\sum_{k\in\mathcal{T}} p(k) = 1$ and $p(k) \geq 0$ for all $k \in \mathcal{T}$). For the directional terrain transition features, the only environment state that gives a feature value of one is (T_{1i}, T_{2j}); all other states give a feature value of zero. The probability of this state is defined as $p(T_{1i}, T_{2j})$, which is equivalent to $p(T_{1i})p(T_{2j})$ since the two terrain types are independent of each other. For the symmetric terrain transition features, both (T_{1i}, T_{2j}) and (T_{1j}, T_{2i}) give a feature value of one with all other states being zero. The probability of this occurring is $p(T_{1i})p(T_{2j}) + p(T_{1j})p(T_{2i})$ if $i \neq j$ and $p(T_{1i})p(T_{2i})$ if $i = j$. The different expression for when i and j refer to the same terrain type is because there is only a single environment state where both terrain types are the same (type i), so it should only be counted once.

As with the terrain type features, the fuzzy terrain transition features are defined using the minimum, maximum, and expected value of the crisp features over all possible

environment states. For the fuzzy directional terrain transition features, the expected value is given as

$$\tilde{f}_{t\langle i,j\rangle}^{\text{mean}}(e) = p(T_{1i})p(T_{2j}), \tag{4.37}$$

and for the fuzzy symmetric terrain transition features, the expected value is given as

$$\tilde{f}_{t\{i,j\}}^{\text{mean}}(e) = \begin{cases} p(T_{1i})p(T_{2j}) + p(T_{1j})p(T_{2i}), & i \neq j \\ p(T_{1i})p(T_{2i}), & i = j\,. \end{cases} \tag{4.38}$$

Because the crisp feature value is binary, we can infer that the minimum possible feature value will be zero, unless both terrain types are known and the feature value is observed to be one. In other words, if the expected value is less than one, then the minimum value will be zero; otherwise it will be one. Likewise, the maximum possible feature value will be one if the expected value is greater than zero; otherwise it will be zero. Formally,

$$\tilde{f}_{t\langle i,j\rangle}^{\text{min}}(e) = \begin{cases} 1, & p(T_{1i})p(T_{2j}) = 1 \\ 0, & \text{otherwise} \end{cases}, \text{and} \tag{4.39}$$

$$\tilde{f}_{t\langle i,j\rangle}^{\text{max}}(e) = \begin{cases} 0, & p(T_{1i})p(T_{2j}) = 0 \\ 1, & \text{otherwise} \end{cases} \tag{4.40}$$

for the fuzzy directional terrain transition features and

$$\tilde{f}_{t\{i,j\}}^{\text{min}}(e) = \begin{cases} 1, & p(T_{1i})p(T_{2j}) + p(T_{1j})p(T_{2i}) = 1 \\ 0, & \text{otherwise} \end{cases}, \text{and} \tag{4.41}$$

$$\tilde{f}_{t\{i,j\}}^{\text{max}}(e) = \begin{cases} 0, & p(T_{1i})p(T_{2j}) + p(T_{1j})p(T_{2i}) = 0 \\ 1, & \text{otherwise} \end{cases} \tag{4.42}$$

for the fuzzy symmetric terrain transition features. In practice, the fuzzy terrain transition features are computed using the following equivalent definitions.

$$\tilde{f}_{t\langle i,j\rangle}^{\text{min}}(e) = [t_1 = i \wedge o_1 = 1 \wedge t_2 = j \wedge o_2 = 1] \tag{4.43}$$

$$\tilde{f}_{t\langle i,j\rangle}^{\text{max}}(e) = [(t_1 = i \vee o_1 = 0) \wedge (t_2 = j \vee o_2 = 0)] \tag{4.44}$$

$$\tilde{f}^{\text{mean}}_{t\langle i,j\rangle}(e) = \begin{cases} 1, & t_1 = i \wedge o_1 = 1 \wedge t_2 = j \wedge o_2 = 1 \\ p(j), & t_1 = i \wedge o_1 = 1 \wedge o_2 = 0 \\ p(i), & o_1 = 0 \wedge t_2 = j \wedge o_2 = 1 \\ p(i)p(j), & o_1 = 0 \wedge o_2 = 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.45}$$

$$\tilde{f}^{\text{min}}_{t\{i,j\}}(e) = \big[\big((t_1 = i \wedge t_2 = j) \vee (t_1 = j \wedge t_2 = i)\big) \wedge o_1 = 1 \wedge o_2 = 1\big] \tag{4.46}$$

$$\tilde{f}^{\text{max}}_{t\{i,j\}}(e) = [(t_1 = i \vee t_1 = j \vee o_1 = 0) \wedge (t_2 = i \vee t_2 = j \vee o_2 = 0)] \tag{4.47}$$

$$\tilde{f}^{\text{mean}}_{t\{i,j\}}(e) = \begin{cases} 1, & \big((t_1 = i \wedge t_2 = j) \vee (t_1 = j \wedge t_2 = i)\big) \wedge \\ & o_1 = 1 \wedge o_2 = 1 \\ p(j), & (t_1 = i \wedge o_1 = 1 \wedge o_2 = 0) \vee \\ & (o_1 = 0 \wedge t_2 = i \wedge o_2 = 1) \\ p(i), & (t_1 = j \wedge o_1 = 1 \wedge o_2 = 0) \vee \\ & (o_1 = 0 \wedge t_2 = j \wedge o_2 = 1) \\ p(i)^2, & o_1 = 0 \wedge o_2 = 0 \wedge i = j \\ 2p(i)p(j), & o_1 = 0 \wedge o_2 = 0 \wedge i \neq j \\ 0, & \text{otherwise} \end{cases} \tag{4.48}$$

The fuzzy numbers representing the directional and symmetric terrain transition features are defined as

$$\tilde{f}_{t\{i,j\}}(e) = \text{Tri}\left(\tilde{f}^{\text{min}}_{t\{i,j\}}(e),\ \tilde{f}^{\text{mean}}_{t\{i,j\}}(e),\ \tilde{f}^{\text{max}}_{t\{i,j\}}(e)\right), \text{and} \tag{4.49}$$

$$\tilde{f}_{t\langle i,j\rangle}(e) = \text{Tri}\left(\tilde{f}^{\text{min}}_{t\langle i,j\rangle}(e),\ \tilde{f}^{\text{mean}}_{t\langle i,j\rangle}(e),\ \tilde{f}^{\text{max}}_{t\langle i,j\rangle}(e)\right). \tag{4.50}$$

A summary of these features is shown in Table 4.3 and Table 4.4. Note that this differs from the terrain type feature summary in Table 4.2 since we consider a problem with more than two terrain types ($|\mathcal{T}| > 2$), and unequal terrain type priors. There are a few noticeable differences between the symmetric and directional versions. In the directional version, only the case where $t_1 = i$ and $t_2 = j$ (blue) is a crisp 1, whereas in

the symmetric version, the case where $t_1 = j$ and $t_2 = i$ (red) is also a crisp 1. The bottom row and rightmost column indicate configurations where at least one of the cells is unobserved. The nonzero configurations are shaded to show the similarity between the two versions of the feature. Note that the mean feature value of the configuration where both cells are unobserved is twice as large in the symmetric version of the feature as compared to the directional version. (We assume that $i \neq j$.)

Table 4.3 Example of the fuzzy symmetric terrain transition feature when $|\mathcal{T}| > 2$, $p(i) = 0.7$, and $p(j) = 0.2$ $(i \neq j)$

$\tilde{f}_{t\{i,j\}}(e)$		$o_2 = 1$			$o_2 = 0$
		$t_2 = i$	$t_2 = j$	$t_2 \neq i \land t_2 \neq j$	
$o_1 = 1$	$t_1 = i$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(1,1,1)$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0.2,1)$
	$t_1 = j$	$\mathrm{Tri}(1,1,1)$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0.7,1)$
	$t_1 \neq i \land t_1 \neq j$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0,0)$
$o_1 = 0$		$\mathrm{Tri}(0,0.2,1)$	$\mathrm{Tri}(0,0.7,1)$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0.28,1)$

Table 4.4 Example of the fuzzy directional terrain transition feature when $|\mathcal{T}| > 2$, $p(i) = 0.7$, and $p(j) = 0.2$ $(i \neq j)$

$\tilde{f}_{t\langle i,j\rangle}(e)$		$o_2 = 1$			$o_2 = 0$
		$t_2 = i$	$t_2 = j$	$t_2 \neq i \land t_2 \neq j$	
$o_1 = 1$	$t_1 = i$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(1,1,1)$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0.2,1)$
	$t_1 = j$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0,0)$
	$t_1 \neq i \land t_1 \neq j$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0,0)$
$o_1 = 0$		$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0.7,1)$	$\mathrm{Tri}(0,0,0)$	$\mathrm{Tri}(0,0.14,1)$

Figure 4.9 shows the same four examples from Figure 4.8, now evaluated for the terrain transition features. We assume that $|\mathcal{T}| = 2$ in all cases. In (a), both cells are observed, so the terrain transition features are crisp binary values. In (b), the second cell is unobserved and is equally likely to be either terrain type 1 (meadow) or type 2 (forest). The feature values are either zero if the observation is incompatible with the feature type, or a completely uncertain fuzzy number spanning the range [0, 1] with a mean of 0.5. In (c), the first cell is unobserved and the priors favor terrain type 1. This changes the means of the fuzzy numbers to reflect the greater likelihood that the unobserved region is type 1. In (d), both cells are unobserved with unequal priors, making each feature span the range [0, 1], but with different mean values. Note that $\tilde{f}_{t\{i,j\}}$ and $\tilde{f}_{t\langle i,j\rangle}$ are identical when $i = j$, and that $\tilde{f}_{t\{1,2\}}^{\text{mean}} = 2\tilde{f}_{t\langle 1,2\rangle}^{\text{mean}}$. Again, this is because when $i \neq j$, the symmetric version of the feature will consider the both cases where (t_1, t_2) is (i, j) and (j, i), but when $i = j$, there is only a single case, (i, i).

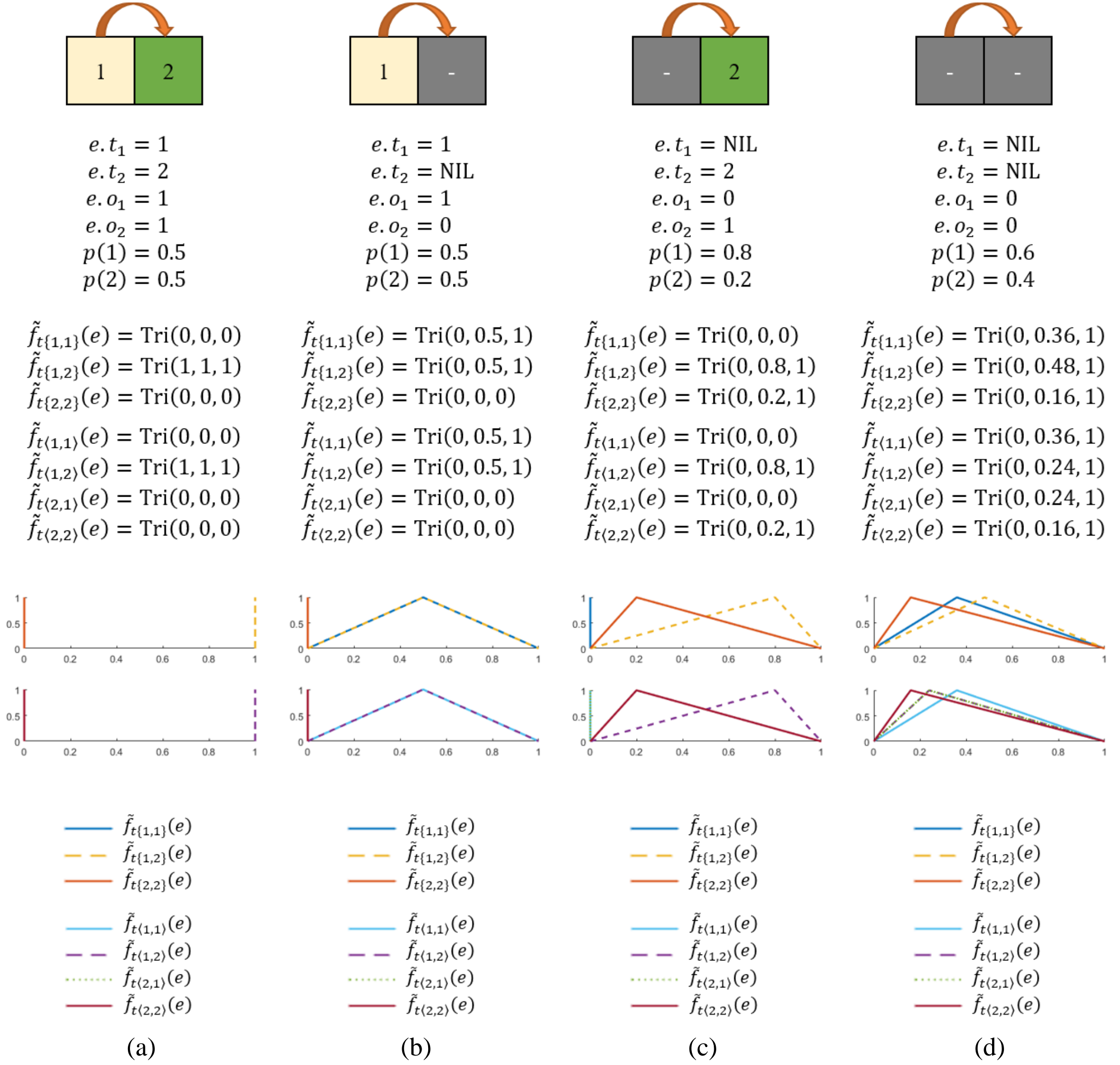


Figure 4.9 Four examples demonstrating the computation of the fuzzy terrain transition features for a single transition between two adjacent grid cells. The light tan region represents terrain type 1 (meadow) and the green region represents terrain type 2 (forest). Gray cells are unobserved. The plots compare the fuzzy terrain transition features for each example.

4.4.4 Elevation Features

Unlike the discrete terrain features, the crisp elevation features come from the continuous domain $[0, 1]$. We defined three elevation features in Section 4.3.4, given by Equations 4.11, 4.12, and 4.13: the absolute, uphill, and downhill elevation difference. In

the partially observed case, we need to compute the minimum, maximum, and expected values of these features over all possible configurations. To simplify the analysis, we assume that the elevation attributes of a grid cell are bounded by the range $[0, 1]$ and that the values are distributed uniformly over this range, so that all elevations are equally likely. Recall that the elevation values of the starting and ending edge cells are given as $h_1, h_2 \in [0, 1]$ and we denote the observability of the cells as o_1 and o_2. If one or both cells of an edge are unobserved, then the minimum elevation difference for all three feature types will always be zero, because it is possible that both cells have the same elevation.

$$\tilde{f}_h^{\min}(e) = \begin{cases} |h_1 - h_2|, & o_1 = 1 \wedge o_2 = 1 \\ 0, & \text{otherwise} \end{cases} \tag{4.51}$$

$$\tilde{f}_{h\uparrow}^{\min}(e) = \begin{cases} \max(0, h_2 - h_1), & o_1 = 1 \wedge o_2 = 1 \\ 0, & \text{otherwise} \end{cases} \tag{4.52}$$

$$\tilde{f}_{h\downarrow}^{\min}(e) = \begin{cases} \max(0, h_1 - h_2), & o_1 = 1 \wedge o_2 = 1 \\ 0, & \text{otherwise} \end{cases} \tag{4.53}$$

If neither cell is observed, the maximum elevation difference is one, and if just one cell is observed, the maximum elevation difference is determined by the observed elevation value.

$$\tilde{f}_h^{\max}(e) = \begin{cases} |h_1 - h_2|, & o_1 = 1 \wedge o_2 = 1 \\ \max(h_1, 1 - h_1), & o_1 = 1 \wedge o_2 = 0 \\ \max(h_2, 1 - h_2), & o_1 = 0 \wedge o_2 = 1 \\ 1, & o_1 = 0 \wedge o_2 = 0 \end{cases} \tag{4.54}$$

$$\tilde{f}_{h\uparrow}^{\max}(e) = \begin{cases} \max(0, h_2 - h_1), & o_1 = 1 \wedge o_2 = 1 \\ 1 - h_1, & o_1 = 1 \wedge o_2 = 0 \\ h_2, & o_1 = 0 \wedge o_2 = 1 \\ 1, & o_1 = 0 \wedge o_2 = 0 \end{cases} \tag{4.55}$$

$$\tilde{f}_{h\downarrow}^{\max}(e) = \begin{cases} \max(0, h_1 - h_2), & o_1 = 1 \wedge o_2 = 1 \\ h_1, & o_1 = 1 \wedge o_2 = 0 \\ 1 - h_2, & o_1 = 0 \wedge o_2 = 1 \\ 1, & o_1 = 0 \wedge o_2 = 0 \end{cases} \tag{4.56}$$

For instance, if the first cell is observed but not the second, then the biggest absolute elevation difference is the greater of h_1 and $1 - h_1$, since these are the differences between the two extremes of the possible range for h_2. Similar reasoning follows for the other cases and feature types.

To get the expected elevation difference, we need to integrate over all possible unobserved values. Consider the plots shown in Figure 4.10. These plots show how the value of the elevation difference features change when just one grid cell is observed. In this case, we can express the expected value of the elevation difference features as

$$\tilde{f}_{h^*}^{\text{mean}}(e) = \int_0^1 f_{h^*}(x|z)p(x)dx, \quad (o_1 = 1 \land o_2 = 0) \lor (o_1 = 0 \land o_2 = 1). \quad (4.57)$$

Here, f_{h^*} is the crisp feature function for either the absolute elevation difference f_h, the uphill elevation difference $f_{h\uparrow}$, or the downhill elevation difference $f_{h\downarrow}$. The function parameter x is the unobserved elevation value and z is the elevation value from the observed cell. The probability of observing x is given as $p(x)$, which can be ignored since we assume a uniform distribution over the interval $[0, 1]$ and therefore $p(x) = 1$.

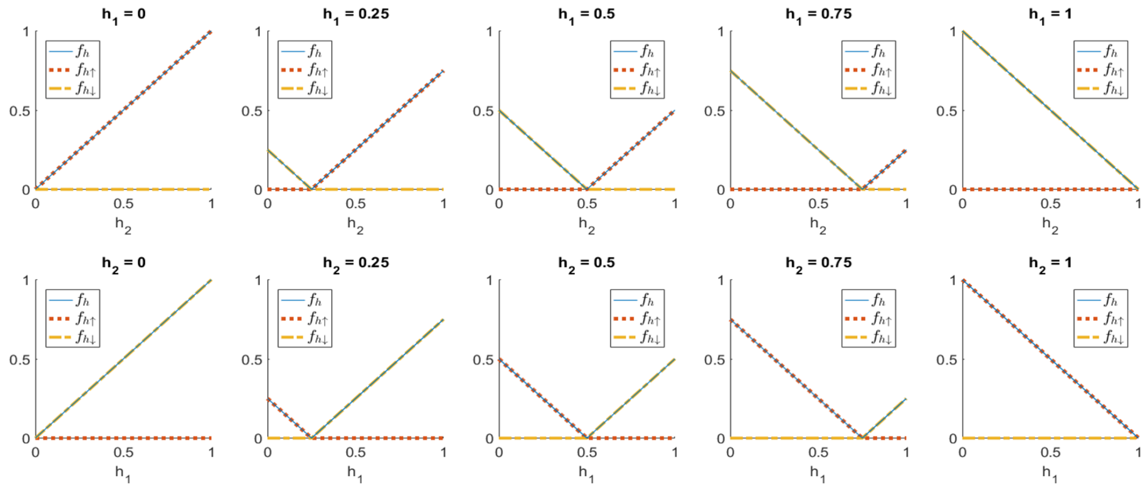


Figure 4.10 Plots of the elevation difference features when one cell is unobserved. The top row assumes that the second cell is unobserved ($o_1 = 1$ and $o_2 = 0$) and the first cell has the value given in the plot title. The bottom row shows the opposite case, where the first cell is unobserved ($o_1 = 0$ and $o_2 = 1$). These plots are cross-sections of the bivariate feature plots shown in Figure 4.6.

As an example, consider the case where the first cell is observed with an elevation of z and the second cell is unobserved (top row of Figure 4.10). The value of the absolute elevation difference can be written as a piecewise linear function of x,

$$f_h(x|z) = \begin{cases} z - x, & x < z \\ x - z, & x \geq z \end{cases}. \tag{4.58}$$

The expected value is then calculated as

$$\tilde{f}_h^{\text{mean}}(z) = \int_0^z (z - x)dx + \int_z^1 (x - z)dx \tag{4.59a}$$

$$= \left[xz - \frac{1}{2}x^2\right]_0^z + \left[\frac{1}{2}x^2 - xz\right]_z^1 \tag{4.59b}$$

$$= \left(\frac{1}{2}z^2\right) + \left(\frac{1}{2}z^2 - z + \frac{1}{2}\right) \tag{4.59c}$$

$$= z^2 - z + \frac{1}{2}. \tag{4.59d}$$

The uphill and downhill elevation difference functions each contain only one of the linear segments from the absolute elevation difference with the other set to zero.

$$f_{h\uparrow}(x|z) = \begin{cases} 0, & x < z \\ x - z, & x \geq z \end{cases} \tag{4.60}$$

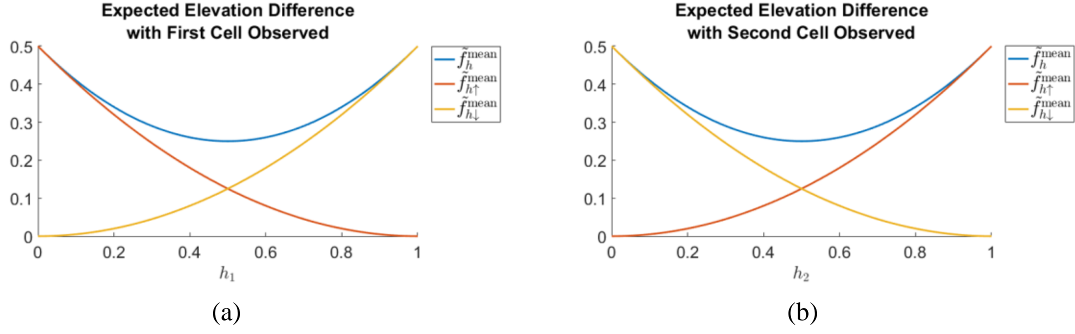
$$f_{h\downarrow}(x|z) = \begin{cases} z - x, & x < z \\ 0, & x \geq z \end{cases} \tag{4.61}$$

The expected values of these functions are the corresponding components of the overall integral from Equation 4.59.

$$\tilde{f}_{h\uparrow}^{\text{mean}}(z) = \int_z^1 (x - z)dx = \frac{1}{2}z^2 - z + \frac{1}{2} \tag{4.62}$$

$$\tilde{f}_{h\downarrow}^{\text{mean}}(z) = \int_0^z (z - x)dx = \frac{1}{2}z^2 \tag{4.63}$$

These functions are shown in Figure 4.11 for all possible values where only one cell is observed. Note that the expected elevation difference value is bounded by the range $[0, 0.5]$.



(a) (b)

Figure 4.11 Plots of the expected elevation difference features when only the first cell (a) or the second cell (b) is observed. Note that the uphill and downhill elevation difference features are switched in the two cases.

Finally, we consider the case where both grid cells are unobserved. The expected elevation difference in this case is the double integral over both possible elevation values,

$$\tilde{f}_{h^*}^{\text{mean}}(e) = \int_0^1 \int_0^1 f_{h^*}(x, y) p(x) dx\, p(y) dy, \quad o_1 = 0 \land o_2 = 0. \tag{4.64}$$

Here, $f_{h^*}(x, y)$ is one of the crisp elevation difference feature functions for two elevation values, x and y. These are given by Equations 4.11, 4.12, and 4.13. Again, since we assume a uniform distribution for x and y, $p(x) = p(y) = 1$. First, consider the absolute elevation difference feature. To compute the double integral, the expression is divided into two parts,

$$\tilde{f}_h^{\text{mean}}(e) = \int_0^1 \int_0^1 |x - y| dx\, dy \tag{4.65a}$$

$$= \int_0^1 \int_0^y (y - x) dx\, dy + \int_0^1 \int_y^1 (x - y) dx\, dy \tag{4.65b}$$

$$= \int_0^1 \left[xy - \frac{1}{2}x^2\right]_0^y dy + \int_0^1 \left[\frac{1}{2}x^2 - xy\right]_y^1 dy \tag{4.65c}$$

$$= \int_0^1 \left(\frac{1}{2}y^2\right) dy + \int_0^1 \left(\frac{1}{2}y^2 - y + \frac{1}{2}\right) dy \tag{4.65d}$$

$$= \left[\frac{1}{6}y^3\right]_0^1 + \left[\frac{1}{6}y^3 - \frac{1}{2}y^2 + \frac{1}{2}y\right]_0^1 \tag{4.65e}$$

$$= \frac{1}{6} + \frac{1}{6} = \frac{1}{3}, \qquad o_1 = 0 \wedge o_2 = 0. \tag{4.65f}$$

As before, the expected values of the uphill and downhill elevation difference features each contain only the corresponding component of the absolute elevation difference.

$$\tilde{f}_{h\uparrow}^{\text{mean}}(e) = \int_0^1 \int_0^y (y - x) dx\, dy = \frac{1}{6}, \qquad o_1 = 0 \wedge o_2 = 0 \tag{4.66}$$

$$\tilde{f}_{h\downarrow}^{\text{mean}}(e) = \int_0^1 \int_y^1 (x - y) dx\, dy = \frac{1}{6}, \qquad o_1 = 0 \wedge o_2 = 0 \tag{4.67}$$

Collecting all the above definitions, we have the following expressions for the expected elevation difference.

$$\tilde{f}_h^{\text{mean}}(e) = \begin{cases} |h_1 - h_2|, & o_1 = 1 \wedge o_2 = 1 \\ h_1^2 - h_1 + \frac{1}{2}, & o_1 = 1 \wedge o_2 = 0 \\ h_2^2 - h_2 + \frac{1}{2}, & o_1 = 0 \wedge o_2 = 1 \\ \frac{1}{3}, & o_1 = 0 \wedge o_2 = 0 \end{cases} \tag{4.68}$$

$$\tilde{f}_{h\uparrow}^{\text{mean}}(e) = \begin{cases} \max(0, h_2 - h_1), & o_1 = 1 \wedge o_2 = 1 \\ \frac{1}{2}h_1^2 - h_1 + \frac{1}{2}, & o_1 = 1 \wedge o_2 = 0 \\ \frac{1}{2}h_2^2, & o_1 = 0 \wedge o_2 = 1 \\ \frac{1}{6}, & o_1 = 0 \wedge o_2 = 0 \end{cases} \tag{4.69}$$

$$\tilde{f}_{h\downarrow}^{\text{mean}}(e) = \begin{cases} \max(0, h_1 - h_2), & o_1 = 1 \wedge o_2 = 1 \\ \frac{1}{2}h_1^2, & o_1 = 1 \wedge o_2 = 0 \\ \frac{1}{2}h_2^2 - h_2 + \frac{1}{2}, & o_1 = 0 \wedge o_2 = 1 \\ \frac{1}{6}, & o_1 = 0 \wedge o_2 = 0 \end{cases} \tag{4.70}$$

As before, the triangular fuzzy numbers for each of the elevation difference features are defined using the min, mean, and max values computed above.

$$\tilde{f}_h(e) = \text{Tri}\left(\tilde{f}_h^{\text{min}}(e),\ \tilde{f}_h^{\text{mean}}(e),\ \tilde{f}_h^{\text{max}}(e)\right) \tag{4.71}$$

$$\tilde{f}_{h\uparrow}(e) = \text{Tri}\left(\tilde{f}_{h\uparrow}^{\text{min}}(e),\ \tilde{f}_{h\uparrow}^{\text{mean}}(e),\ \tilde{f}_{h\uparrow}^{\text{max}}(e)\right) \tag{4.72}$$

$$\tilde{f}_{h\downarrow}(e) = \text{Tri}\left(\tilde{f}_{h\downarrow}^{\text{min}}(e),\ \tilde{f}_{h\downarrow}^{\text{mean}}(e),\ \tilde{f}_{h\downarrow}^{\text{max}}(e)\right) \tag{4.73}$$

Figure 4.12 shows the computation of the fuzzy elevation difference features for four different cases. In (a), both cells are observed, so the features are all crisp values. In

(b), only the first cell is observed with a height value of 0.4. The minimum possible value for all three features is 0. If the unobserved cell were to have a height of 1, $f_{h\uparrow}$ would have its maximum value of 0.6, whereas if it were to have a height of 0, $f_{h\downarrow}$ would have its maximum value of 0.4. The maximum of f_h is the greater of these two, 0.6. The mean values of all three features are given by the above definitions, and one can see that because the observed value is less than 0.5, $f_{h\downarrow}$ has the smallest expected value. The expected value of f_h is greater than that of $f_{h\uparrow}$ because the latter will be zero if the true height of the second cell is anything less than 0.4. In (c), the first cell is unobserved and the second cell is observed to be zero. Since the height of the first cell cannot be less than 0, there is no possibility of an uphill slope, so $f_{h\uparrow}$ is a crisp 0. The other two features, f_h and $f_{h\downarrow}$, both scale linearly with the unobserved height value and have a maximum value of 1 and an average value of 0.5. In (d), both cells are unobserved, so the feature definitions are given by the expressions derived previously. The minimum value of all three features is 0 and the maximum is 1. The expected value of f_h is $\frac{1}{3}$ (≈ 0.33), and the expected value of both $f_{h\uparrow}$ and $f_{h\downarrow}$ is $\frac{1}{6}$ (≈ 0.17).

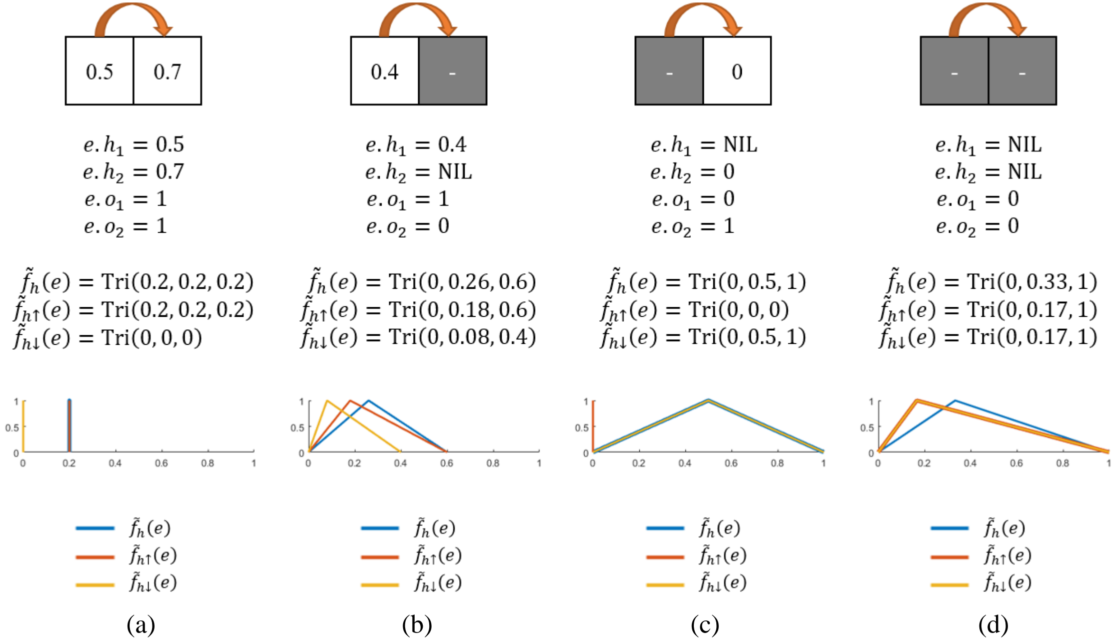


Figure 4.12 Four examples demonstrating the computation of the fuzzy elevation difference features for a single transition between two adjacent grid cells. The numbers inside the cells indicate the height value. Gray cells are unobserved. The plots compare the fuzzy elevation difference features for each example.

4.5 Summary

This chapter introduced the mental map grid and the action graph used by the agent to represent the observed environment in the CMM framework. The simulation server keeps track of the agent's location within the environment and computes the viewshed region using line-of-sight, considering obstructions from elevation and terrain. The agent maintains a record of all the observations it receives and stores the information in grid layers representing the attributes of the environment. An additional layer indicates which cells have been observed and which cells still have unknown properties.

The action graph is defined over all grid cells that are potentially traversable and indicates possible movement actions by the agent. Each movement step is an edge in the graph between adjacent grid cells. Several features are defined for each edge representing

distance, the terrain type of each cell, and the change in elevation. When both cells are observed, the feature values are known with no uncertainty and are stored as crisp values. If one or both grid cells is unobserved, then there is some uncertainty in the feature values, which are represented as triangular fuzzy numbers. While the action graph provides a low-level analysis of the cost of moving through the environment, it can often be helpful to summarize this information, both to reduce the number of decision points in the planning process and to more closely model the humanistic concepts of spatial reasoning. The next chapter introduces the region graph, which provides this summary by grouping similar nearby grid cells into regions and computing the feature costs between adjacent regions.

5 THE REGION GRAPH

In this chapter, we introduce the concept of the region graph, which summarizes the information in the mental map and allows the agent to develop plans at a higher level. Whereas the action graph specifies the cost of individual actions, the region graph specifies the cost of multiple aggregated actions that cannot be performed immediately, but may be used in future plans. The region graph must therefore deal with the uncertainty inherent in extending the single-step feature definitions to multi-step features defined over regions of the environment.

5.1 The Region Graph

Up to this point, we have considered only single-step transitions between adjacent grid cells. These short edges comprise the action graph G_A and represent the actual steps that an agent can take within the environment. Each edge e in the action graph is attributed with one or more feature values to give a feature vector $\mathbf{f}(e)$, which in the general case is comprised of triangular fuzzy numbers. While the action graph gives a low-level representation of the information in the mental map $\mathcal{M}$, the decision-maker is often unable to fully utilize all this knowledge. Planning typically occurs at a higher level of cognition where the spatial information and feature values have been summarized into a more succinct form. We introduce the region graph G_R to provide this summary of the information in the action graph. Note that the region graph will be less precise than the more granular action graph, but will allow planning to take place at a higher level with fewer decision points.

Our concept of the region graph is to combine similar nearby grid cells into a single region that is represented as one vertex in the graph. Adjacent regions are connected by bidirectional edges. This can drastically reduce the size of the graph and make it easier to develop high-level plans. To construct the region graph, we use the region partitioning algorithm introduced in Section 3.3. Each terrain type and the unobserved areas are partitioned separately to ensure that each resulting region is either completely unobserved or contains only a single type of terrain. This is an important restriction that we employ to facilitate the computation of fuzzy features in the region graph, which will be described in Sections 5.2 and 5.3. Additionally, we define a local region around the agent and any observed resources that will not be clustered. This ensures that the grid cells immediately surrounding the agent and any goal locations are given their own vertices in the region graph. The region graph within the local region is identical to the action graph, which means that the immediate decision actions available to the agent are actual movement steps that the agent can take in the environment. Without this restriction, an agent might develop a plan to move into an adjacent region that is accessible from multiple directions, but not specify to the simulation server which direction to move. The region graph is updated after each movement action by the agent, which will be discussed in Section 5.4.

Algorithm 5.1 provides a high-level overview of the process for creating the initial region graph. The function takes the current mental map $\mathcal{M}$ as input and a set of options specified by the variable *opt*. The first step is to get the local region (lines 1 and 2), which is given in Algorithm 5.2. Then, a clustering mask Q is created to define all areas outside of the local region as regions that need to be clustered (lines 3-5). The region boundaries

are defined on line 6 using Algorithm 5.3. Line 7 creates the structure and features of the region graph using Algorithm 5.5. The updated mental map structure is returned on line 8.

Algorithm 5.1 Create the Initial Region Graph

INITIALIZE_REGION_GRAPH($\mathcal{M}$, *opt*)

/ Get the local region */*
1: *LR* ← GET_LOCAL_REGION($\mathcal{M}$, *opt*) *// Algorithm 5.2*
2: $\mathcal{M}$*.localRegion* ← *LR*

/ Create the region boundaries */*
3: $(n, m) \leftarrow \mathcal{M}.size$
4: $Q \leftarrow n \times m$ grid initalized to 1
5: $Q[LR] \leftarrow 0$
6: $\mathcal{M}.L$ ← CLUSTER_MENTAL_MAP_REGIONS($\mathcal{M}$, *LR*, *Q*, *opt*) *// Algorithm 5.3*

/ Construct the region graph */*
7: $\mathcal{M}.G_R$ ← CREATE_REGION_GRAPH($\mathcal{M}$) *// Algorithm 5.5*

8: **return** $\mathcal{M}$

5.1.1 Defining the Local Region

The local region is first defined on line 1 of Algorithm 5.1 using the GET_LOCAL_REGION function in Algorithm 5.2. There are two methods we consider for defining the local region, specified by the option *opt.lrMethod*. If *opt.lrMethod* = “all”, then the entire traversable space is marked as part of the local region (lines 3-4 of Algorithm 5.2). This is essentially a control parameter to allow for experimentation with no region clustering. In the default case, the local region is first defined as all observed grid cells within a distance of *opt.lrDist* from the current agent position using the GRID_DISTANCE

function from Algorithm 3.6 (lines 6-10). Next, any observed resources are included as part of the local region (line 11). This is done to ensure that cells that contain resources are given their own vertices in the region graph. This also ensures that a region will not contain more than one resource. Figure 5.1. shows an example of defining the local region.

Algorithm 5.2 Get the Local Region

```
GET_LOCAL_REGION(𝓜, opt)
1:  (n, m) ← 𝓜.size
2:  LR ← n × m grid initalized to 0
3:  if opt.lrMethod = "all"
4:      LR[𝓜.W ≠ 0] ← 1            // Include all potentially traversable cells
5:  else
6:      (a_i, a_j) ← 𝓜.pos
7:      W ← 𝓜.W
8:      W[𝓜.W ≠ 1] ← 0
9:      D ← GRID_DISTANCE(W, a_i, a_j, opt.lrDist)                 // Algorithm 3.6
10:     LR[D ≤ opt.lrDist] ← 1      // Include observed cells near the agent
11:     LR[𝓜.R > 0] ← 1            // Include observed resource locations
12: return LR
```

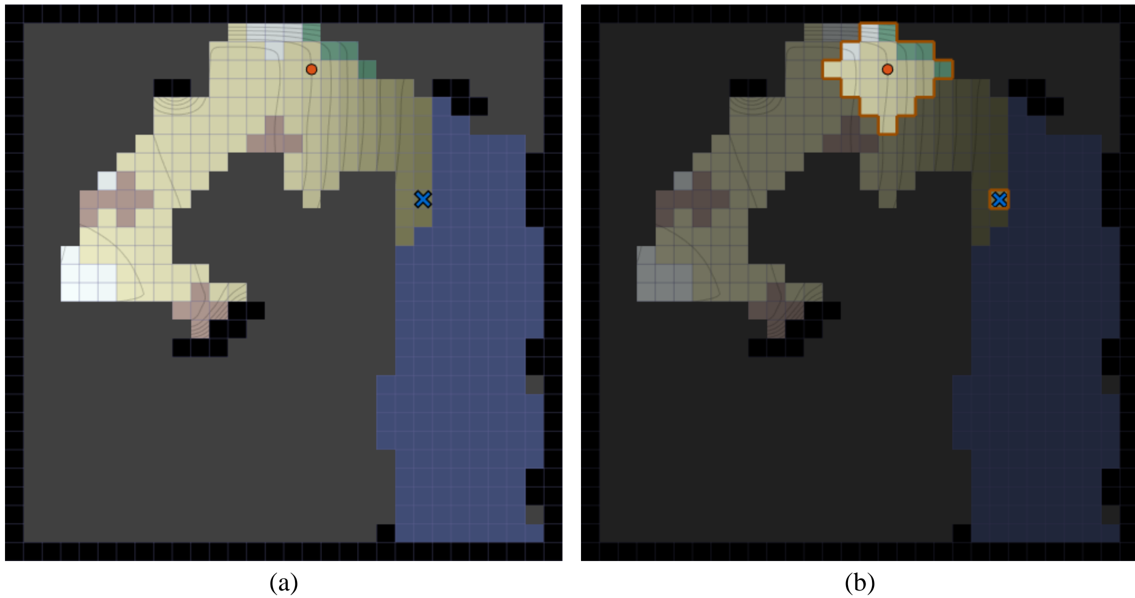


Figure 5.1 An example of determining the local region. (a) The initial observation an agent receives in a new environment. (b) The local region is highlighted within a distance of 3 cells from the agent and in the cell containing an observed resource. Note that unobserved cells and walls are excluded from the local region.

5.1.2 Creating the Region Boundaries

After determining the local region, the next step is to cluster the remaining area to define the region boundaries. This is done by the CLUSTER_MENTAL_MAP_REGIONS function in Algorithm 5.3. Lines 2-7 initialize the region label matrix L by assigning a unique label to each cell in the local region. Then, each type of terrain is clustered separately (lines 8-17). Line 9 identifies the grid cells within the clustering mask Q with terrain type t, and if there are none, the loop proceeds to the next terrain type (lines 10-11). A wall matrix is defined for these cells (lines 12-13) and the corresponding elevation values are extracted from the mental map (lines 14-15). These are passed to the PARTITION_REGIONS function from Algorithm 3.4 with a cluster separation radius defined by *opt.regionSize* to get a set of labels U (line 16). These labels are added to the region

label matrix using the UPDATE_REGION_MAP function given in Algorithm 5.4. This function ensures that the label indices from U do not conflict with those already in L.

Once the terrain has been clustered, lines 18-24 of Algorithm 5.3 cluster the unobserved areas. The elevation matrix is set to all zeros and the separation radius for the PARTITION_REGIONS function is set by *opt.hiddenSize*, which we usually set to be larger than *opt.regionSize* to reduce the number of unobserved regions. After the entire environment has been clustered, the region labels are returned on line 25. The final region boundaries from the example in Figure 5.1 are shown in Figure 5.2 (a).

Algorithm 5.3 Create the Initial Mental Map Regions

```
CLUSTER_MENTAL_MAP_REGIONS(𝓜, LR, Q, opt)
1:  (n, m) ← 𝓜.size

    /* Assign labels to cells in the local region */
2:  L ← n × m grid initalized to 0
3:  I ← {(i, j) | LR[i, j] = 1}
4:  k ← 1
5:  for each (i, j) ∈ I
6:      L[i, j] ← k
7:      k ← k + 1

    /* Cluster each terrain type separately */
8:  for each t ∈ 𝓜.𝒯
9:      I ← {(i, j) | 𝓜.T[i, j] = t ∧ Q[i, j] = 1}
10:     if I = ∅
11:         continue
12:     W ← n × m grid initalized to 0
13:     W[I] ← 1
14:     E ← 𝓜.E
15:     E[W = 0] ← NIL
16:     U ← PARTITION_REGIONS(W, E, opt.regionSize, opt.w_e, opt.ϵ)     // Algorithm 3.4
17:     L ← UPDATE_REGION_MAP(L, U)                                      // Algorithm 5.4

    /* Cluster the unobserved areas */
18: I ← {(i, j) | 𝓜.W[i, j] = NIL ∧ Q[i, j] = 1}
19: if |I| > 0
20:     W ← n × m grid initalized to 0
21:     W[I] ← 1
22:     E ← n × m grid initalized to 0
23:     U ← PARTITION_REGIONS(W, E, opt.hiddenSize, opt.w_e, opt.ϵ)     // Algorithm 3.4
24:     L ← UPDATE_REGION_MAP(L, U)                                      // Algorithm 5.4

25: return L
```

Algorithm 5.4 Update Region Map

UPDATE_REGION_MAP(L, U)
1: $kmax \leftarrow \max(L)$
2: **for** k in 1 to $\max(U)$
3: $\quad I \leftarrow \{(i, j) \mid U[i, j] = k\}$
4: $\quad L[I] \leftarrow kmax + k$
5: **return** L

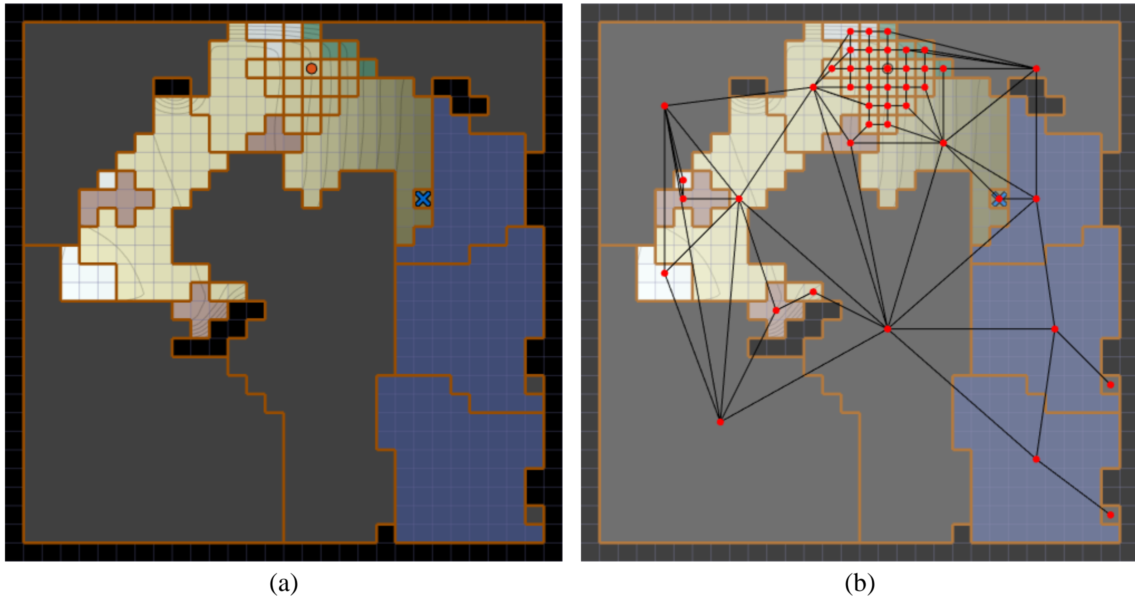


Figure 5.2 (a) Region boundaries computed from the example in Figure 5.1 using Algorithm 5.3. (b) The region graph defined from the region labels. Each pair of adjacent regions is connected by a bidirectional edge in the graph.

5.1.3 Constructing the Region Graph

The final step in creating the initial region graph is to define the structure and features of the graph itself. The region graph G_R is built from the current information stored in the mental map $\mathcal{M}$. Each region with a unique label in the label map $\mathcal{M}.L$ is defined as a vertex of the region graph, and adjacent regions are connected with bidirectional edges.

Sections 5.2 and 5.3 describe the process for computing features for edges between regions. Within the local region, and anywhere that adjacent regions each contain only a single grid cell, the region graph and its edge attributes are identical to the corresponding vertices and edges of the action graph. Elsewhere, the region graph summarizes the information in the action graph using the structure presented in this section and the features computed in the next two sections. For consistency, we always use the region graph for planning agent actions, even in special circumstances where the region graph is identical to the action graph, such as when the local region method is defined using *opt.lrMethod* = "all", making each region a single cell.

Algorithm 5.5 shows the procedure for creating the region graph. The function takes the mental map structure $\mathcal{M}$ as input and returns a structure representing the region graph. We begin by creating the graph vertices on lines 2-6, which are stored in the list *V*. The center point of each region is found on line 2 using the GET_REGION_CENTERS function from Algorithm 3.9. This is saved along with a the cells belonging to each region on lines 4-6. After defining the graph vertices, lines 7-20 define the graph edges. We start with an empty adjacency matrix *A* on line 7 and an empty list of edge features on line 8. Line 9 initalizes the edge index, which is used to associate edge features with the adjacency matrix. For each vertex, lines 11 and 12 construct a mask of the region assigned to this vertex. This mask is dilated on line 13 to get the 4-connected neighbors. Line 14 identifies the region labels of the neighboring regions, and each one is added as an edge in lines 15-20. The current edge index is incremented on line 16 and saved in the adjacency matrix on line 17. This allows for a quick lookup into the edge feature list *E*, which will contain the features computed by the COMPUTE_REGION_FEATURES function, discussed further in

Sections 5.2 and 5.3. Lines 18 and 19 create a region map R for each edge, with cells belonging to the first region labeled 1, cells belonging to the second region labeled 2, and all other cells labeled 0. This map is used to compute the region features for the edge using Algorithm 5.10 on line 20. After defining all the graph edges, the vertices, adjacency matrix, and edge features are all saved and returned as the region graph G_R on lines 21-25. Figure 5.2 (b) shows the region graph for the example in Figure 5.1 with vertices drawn at the region centers. Note that while the edge is only drawn between the center points of adjacent regions, the edge exists conceptually between the two regions as a whole.

Algorithm 5.5 Create Region Graph

```
CREATE_REGION_GRAPH(𝓜)
1:  (n, m) ← 𝓜.size

    /* Add vertices for each region */
2:  C ← GET_REGION_CENTERS(𝓜.L)                                  // Algorithm 3.9
3:  V ← list of |C| uninitialized vertices
4:  for k in 1 to |C|
5:      V[k].region ← {(i, j) | 𝓜.L[i, j] = k}
6:      V[k].center ← C[k]

    /* Add edges for adjacent regions */
7:  A ← |C| × |C| adjacency matrix initalized to 0
8:  E ← empty list of edge features
9:  i ← 0
10: for k in 1 to |C|
11:     U ← n × m grid initalized to 0
12:     U[V[k].region] ← 1
13:     U′ ← U ⊕ [0 1 0; 1 1 1; 0 1 0]        // Dilate to get neighboring cells
14:     N ← {l | l ∈ 𝓜.L[U′ = 1] ∧ l ≠ 0 ∧ l ≠ k}
15:     for n in 1 to |N|
16:         i ← i + 1
17:         A[k][n] ← i
18:         R ← U
19:         R[V[n].region] ← 2
20:         E[i] ← COMPUTE_REGION_FEATURES(𝓜, R)                 // Algorithm 5.10

    /* Save the graph structure */
21: G_R ← empty graph structure
22: G_R.V ← V
23: G_R.A ← A
24: G_R.E ← E

25: return G_R
```

5.2 Fuzzy Region Distance

Each edge e of the region graph G_R connects two adjacent regions and is annotated with the same features defined in the previous chapter. Let R_1 be the starting region and R_2 be the ending region. We define the fuzzy region features of the edge $e_{R_1R_2} \in E(G_R)$ as triangular fuzzy numbers that represent the minimum, maximum, and average feature values that the agent could expect to encounter when moving from any grid cell in R_1 to any grid cell in R_2. Consider the example in Figure 5.3 (a) that shows two adjacent regions of different terrain types with labeled elevation values. Let G_{12} be a subgraph of the action graph G_A that contains only the vertices belonging to R_1 or R_2. We define three additional subgraphs of G_{12} that will be used to compute the fuzzy region features. G_1 is the subgraph of G_{12} that contains only the vertices belonging to grid cells in R_1. Likewise, G_2 is the subgraph of G_{12} for R_2. The boundary graph G_{bnd} consists of only the edges and vertices belonging to the transition between the two regions. For every edge $e \in E(G_{\text{bnd}})$, $\text{START}(e) \in R_1$ and $\text{END}(e) \in R_2$. Note that the only edges from G_{12} that are not assigned to G_1, G_2, or G_{bnd} are those that return from R_2 back to R_1. These three graphs are shown in Figure 5.3 (b). All edges except the boundary edges are bidirectional, indicating that only one boundary edge can be used in a path from R_1 to R_2. In this section, we define a measure of the distance between two adjacent regions. This will be used to define the distance and terrain-based fuzzy region features. Section 5.3 will extend this approach to the elevation feature.

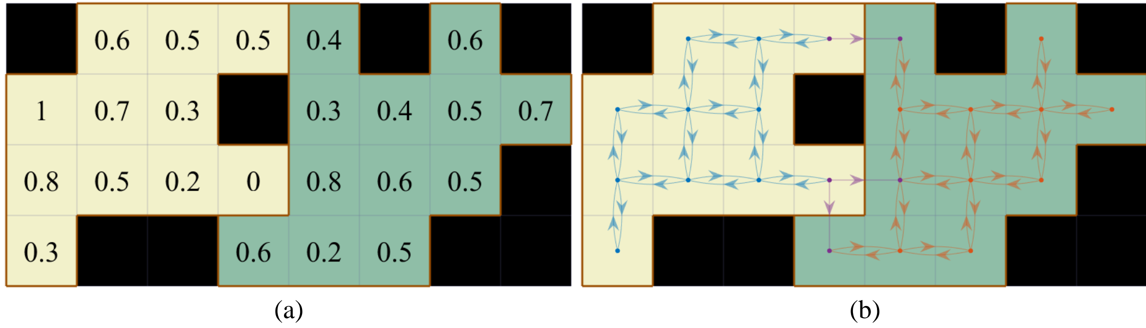


Figure 5.3 (a) An example of two regions used to demonstrate the computation of fuzzy region features. The left region R_1 is terrain type 1 (meadow) and the right region R_2 is terrain type 2 (forest). The numbers in each cell indicate the elevation. (b) There are three graphs for the two regions. G_1 (blue) and G_2 (orange) are bidirectional graphs that are each completely contained in R_1 and R_2 respectively. G_{bnd} (purple) consists of only the edges that start in R_1 and end in R_2.

5.2.1 Computing the Distance Cost Matrix

A prerequisite for many of the fuzzy region features is a measure of the distance between the two regions. Assume that each edge $e \in E(G_{12})$ is assigned a crisp cost value of 1, corresponding to the distance feature in Equation 4.4. (In the following sections, we may consider a different cost for each edge.) An agent in R_1 could exist in any one of the cells belonging to this region and need to know the minimum total distance cost required to get to any one of the cells in R_2. We define all possible costs using the matrix C, where C_{ij} represents the minimum cost required to move from cell $i \in R_1$ to cell $j \in R_2$. In the special case where all edge costs are 1, this is equivalent to the distance between the two cells, restricted to only using cells from the two regions. This can be cast as a special case of the all-pairs shortest path problem where we are only interested in paths that originate in R_1 and end in R_2. One way to compute this is to run the Floyd-Warshall algorithm (Floyd 1962; Warshall 1962) on G_{12} and then extract the submatrix corresponding to only the

paths that start in R_1 and end in R_2. The Floyd-Warshall algorithm has a computational complexity of $O(|V|^3)$, and results in significant overhead for this problem, since most of the computed distances are disregarded. Our approach improves on this by utilizing the regular grid structure of the graph and an additional requirement that each path can only contain one transition edge between the two regions. This ensures that the agent moves directly from R_1 to R_2 without moving repeatedly between the two regions.

To compute the cost matrix C efficiently, we consider each boundary edge independently and analyze the costs of all paths using that boundary edge. This allows us to only compute the single-source shortest path costs to and from each boundary edge, as opposed to the shortest paths between all pairs of cells.[1] For each boundary edge $k \in E(G_{\text{bnd}})$, let u_{ik}^1 be the minimum cost required to get from cell $i \in R_1$ to the start of boundary edge k. Likewise, let u_{jk}^2 be the minimum cost required to get from the end of boundary edge k to cell $j \in R_2$. Also, let u_k^{bnd} be the cost of boundary edge k (set to 1 for the distance feature). The distance feature computes the cost of a path as the total sum of the individual edge costs. This is an example of summation aggregation. Alternatively, the cost of a path for some features may be evaluated as the maximum cost of an edge in the path, such as when planning a path that minimizes the maximum change in elevation for each edge (see Section 5.3). We define the minimum cost of a path from cell i to cell j using boundary edge k as u_{ijk} where

$$u_{ijk} = u_{ik}^1 + u_k^{\text{bnd}} + u_{jk}^2 \tag{5.1}$$

[1] This approach is different from computing the shortest paths to any cell on the region boundary using a shortest path algorithm with multiple source cells. Such an approach would overlook the cost of traveling along the region boundary, essentially allowing free travel from one end of the border to the other.

when using the summation aggregation method (as with the distance feature) and

$$u_{ijk} = \max\left(u_{ik}^{1}, u_{k}^{\text{bnd}}, u_{jk}^{2}\right) \tag{5.2}$$

when using the maximization aggregation method. The overall minimum cost to get from cell i to cell j is defined over all boundary edges as

$$C_{ij} = \min_{k} u_{ijk}. \tag{5.3}$$

The cost matrix for the distance feature is somewhat of a special case, since all edges are given a uniform cost of 1. This is true even if a region is unobserved. Since we define the region boundaries with no uncertainty, the only factor that influences the distance feature is the shape and arrangement of the two adjacent regions. We define the distance cost matrix as C^{d} and the individual region cost matrices as U^{d1} and U^{d2}. These can be computed from the region map R using the GET_REGION_DISTANCE function in Algorithm 5.6. The input R is a grid that spans the two regions, with cells in R_1 marked 1, cells in R_2 marked 2, and all other cells marked 0. Line 1 gets the indices of the two regions using Algorithm 5.7. Note that these are stored as ordered lists of tuples that define a lexographic ordering of the grid cells. Lines 2-5 construct the individual regions maps W_1 and W_2 that are 1 inside of their respective regions and 0 elsewhere. Line 6 gets the boundary edges E_{bnd} between the two regions using Algorithm 5.8. This function also defines an ordering of the boundary edges to maintain consistency between the various cost matrices. Each edge in E_{bnd} is represented as a 4-tuple (i_1, j_1, i_2, j_2), where (i_1, j_1) is a cell in R_1 and (i_2, j_2) is an adjacent cell in R_2. Lines 7-9 initialize the output matrices, where U^{d1} stores the distances from cells in R_1 to each boundary edge, U^{d2} stores the distances from each boundary edge to cells in R_2, and C^{d} stores the distances between all pairs of cells in

the two regions. Lines 10-15 compute the distances to and from each boundary edge using the GRID_DISTANCE function from Algorithm 3.6. The starting and ending cells of each boundary edge are used as the starting points for the distance computations using the region map for each region. After computing the distances for the entire grid on lines 12 and 13, the distance values within each region are saved to U^{d1} and U^{d2} on lines 14 and 15. Finally, we compute the overall distance cost matrix C^{d} for each pair of cells in the two regions using Equations 5.1 and 5.3 on lines 16 and 17. Note that the u_k^{bnd} values are set to 1, since the distance cost of each boundary edge is always 1. The cost matrices are returned on line 18.

Algorithm 5.6 Get Fuzzy Distance Cost Matrices for Two Regions

GET_REGION_DISTANCE(R)

/ Get the indices of the two regions */*
1: $I_1, I_2 \leftarrow$ GET_REGION_INDICES(R) *// Algorithm 5.7*

/ Create individual region maps */*
2: $(n, m) \leftarrow$ size of R
3: $W_1, W_2 \leftarrow n \times m$ matrices initalized to 0
4: $W_1[I_1] \leftarrow 1$
5: $W_2[I_2] \leftarrow 1$

/ Get the boundary edges */*
6: $E_{\text{bnd}} \leftarrow$ GET_BOUNDARY_EDGES(n, m, I_1, I_2) *// Algorithm 5.8*

/ Initialize the output matrices */*
7: $U^{d1} \leftarrow |I_1| \times |E_{\text{bnd}}|$ matrix initalized to ∞
8: $U^{d2} \leftarrow |I_2| \times |E_{\text{bnd}}|$ matrix initalized to ∞
9: $C^{d} \leftarrow |I_1| \times |I_2|$ matrix initalized to ∞

/ Compute region distances */*
10: **for** k in 1 to $|E_{\text{bnd}}|$
11: $(i_1, j_1, i_2, j_2) \leftarrow E_{\text{bnd}}[k]$
12: $D_1 \leftarrow$ GRID_DISTANCE(W_1, i_1, j_1, ∞) *// Algorithm 3.6*
13: $D_2 \leftarrow$ GRID_DISTANCE(W_2, i_2, j_2, ∞) *// Algorithm 3.6*
14: $U^{d1}[:, k] \leftarrow D_1[I_1]$
15: $U^{d2}[:, k] \leftarrow D_2[I_2]$

/ Find the boundary edge that gives the minimum cost */*
16: **for** each $(i, j) \in$ in $\{(i, j) \mid 1 \le i \le |I_1| \wedge 1 \le j \le |I_2|\}$
17: $C^{d}[i, j] \leftarrow \min_k\{U^{d1}[i, k] + 1 + U^{d2}[j, k]\}$

18: **return** C^{d}, U^{d1}, U^{d2}

Algorithm 5.7 Get Region Indices

GET_REGION_INDICES(R)
1: $(n, m) \leftarrow$ size of R
2: $I_1, I_2 \leftarrow$ empty lists
3: $N_1, N_2 \leftarrow 0$
4: **for** j in 1 to m
5: **for** i in 1 to n
6: **if** $R[i, j] = 1$
7: $N_1 \leftarrow N_1 + 1$
8: $I_1[N_1] \leftarrow (i, j)$
9: **else if** $R[i, j] = 2$
10: $N_2 \leftarrow N_2 + 1$
11: $I_2[N_2] \leftarrow (i, j)$
12: **return** I_1, I_2

Algorithm 5.8 Get Boundary Edges

GET_BOUNDARY_EDGES(n, m, I_1, I_2)
1: $E_{\text{bnd}} \leftarrow$ empty list
2: $K \leftarrow 0$
3: **for** j in 1 to m
4: **for** i in 1 to n
5: **if** $(i, j) \in I_1$
6: **if** $(i, j-1) \in I_2$
7: $K \leftarrow K + 1$
8: $E_{\text{bnd}}[K] = (i, j, i, j-1)$
9: **if** $(i, j+1) \in I_2$
10: $K \leftarrow K + 1$
11: $E_{\text{bnd}}[K] = (i, j, i, j+1)$
12: **if** $(i-1, j) \in I_2$
13: $K \leftarrow K + 1$
14: $E_{\text{bnd}}[K] = (i-1, j, i, j)$
15: **if** $(i+1, j) \in I_2$
16: $K \leftarrow K + 1$
17: $E_{\text{bnd}}[K] = (i+1, j, i, j)$
18: **return** E_{bnd}

Figure 5.4 shows the composite distance grids computed for the example problem in Figure 5.3. These are the values returned by the GRID_DISTANCE function on lines 12 and 13 of Algorithm 5.6. The individual region and overall distance cost matrices for this example are shown in Figure 5.5. The grid cells are indexed by consecutive columns from left to right, and from top to bottom within each column. Note that the values of each column of U^{d1} and U^{d2} match the values of the corresponding distance grid region in Figure 5.4.

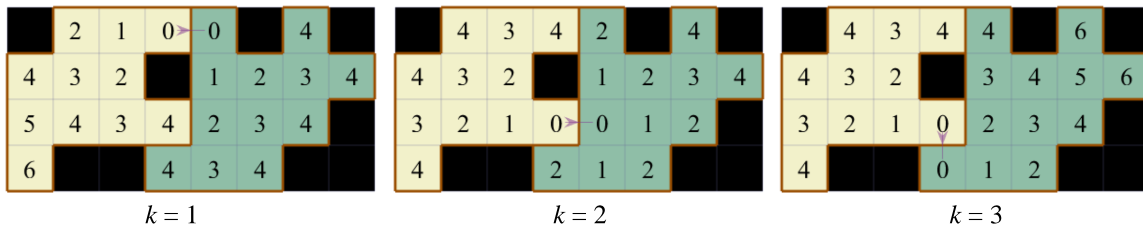


Figure 5.4 Composite distance grids for each of the three boundary edges for the example in Figure 5.3. The numbers indicate the number of steps required to get to or from the boundary edge. The index k is used to reference each of the three boundary edges.

$$U^{d1} = \begin{bmatrix} 4 & 4 & 4 \\ 5 & 3 & 3 \\ 6 & 4 & 4 \\ 2 & 4 & 4 \\ 3 & 3 & 3 \\ 4 & 2 & 2 \\ 1 & 3 & 3 \\ 2 & 2 & 2 \\ 3 & 1 & 1 \\ 0 & 4 & 4 \\ 4 & 0 & 0 \end{bmatrix} \quad U^{d2} = \begin{bmatrix} 4 & 2 & 0 \\ 0 & 2 & 4 \\ 1 & 1 & 3 \\ 2 & 0 & 2 \\ 3 & 1 & 1 \\ 2 & 2 & 4 \\ 3 & 1 & 3 \\ 4 & 2 & 2 \\ 4 & 4 & 6 \\ 3 & 3 & 5 \\ 4 & 2 & 4 \\ 4 & 4 & 6 \end{bmatrix} \quad C^{d} = \begin{bmatrix} 5 & 5 & 6 & 5 & 6 & 7 & 6 & 7 & 9 & 8 & 7 & 9 \\ 4 & 6 & 5 & 4 & 5 & 6 & 5 & 6 & 8 & 7 & 6 & 8 \\ 5 & 7 & 6 & 5 & 6 & 7 & 6 & 7 & 9 & 8 & 7 & 9 \\ 5 & 3 & 4 & 5 & 6 & 5 & 6 & 7 & 7 & 6 & 7 & 7 \\ 4 & 4 & 5 & 4 & 5 & 6 & 5 & 6 & 8 & 7 & 6 & 8 \\ 3 & 5 & 4 & 3 & 4 & 5 & 4 & 5 & 7 & 6 & 5 & 7 \\ 4 & 2 & 3 & 4 & 5 & 4 & 5 & 6 & 6 & 5 & 6 & 6 \\ 3 & 3 & 4 & 3 & 4 & 5 & 4 & 5 & 7 & 6 & 5 & 7 \\ 2 & 4 & 3 & 2 & 3 & 4 & 3 & 4 & 6 & 5 & 4 & 6 \\ 5 & 1 & 2 & 3 & 4 & 3 & 4 & 5 & 5 & 4 & 5 & 5 \\ 1 & 3 & 2 & 1 & 2 & 3 & 2 & 3 & 5 & 4 & 3 & 5 \end{bmatrix}$$

Figure 5.5 Individual region and overall distance cost matrices for the example in Figure 5.4, given as the output of Algorithm 5.6.

5.2.2 Region Distance Feature

As mentioned previously, the fuzzy region features are defined to represent the minimum, mean, and maximum feature values that the agent could encounter when moving between regions. Let $e_{R_1R_2} \in E(G_R)$ be the region graph edge from R_1 to R_2 for which we need to compute a fuzzy feature value, and let G_{12} be the subgraph of the action graph G_A that is completely within R_1 and R_2. We define C^d as the cost matrix computed by Algorithm 5.6 using the distance feature for all edges, i.e. $f(e) = f_d = 1 \; \forall \; e \in E(G_{12})$. The min, mean, and max region distance features are defined as

$$\tilde{f}_d^{\min}\left(e_{R_1R_2}\right) = C_{\min}^d = \min_{\substack{i=1,\ldots,|R_1| \\ j=1,\ldots,|R_2|}} C_{ij}^d, \tag{5.4}$$

$$\tilde{f}_d^{\text{mean}}\left(e_{R_1R_2}\right) = C_{\text{mean}}^d = \frac{1}{|R_1||R_2|} \sum_{\substack{i=1,\ldots,|R_1| \\ j=1,\ldots,|R_2|}} C_{ij}^d \text{, and} \tag{5.5}$$

$$\tilde{f}_d^{\max}\left(e_{R_1R_2}\right) = C_{\max}^d = \max_{\substack{i=1,\ldots,|R_1| \\ j=1,\ldots,|R_2|}} C_{ij}^d. \tag{5.6}$$

The resulting fuzzy region distance feature is

$$\tilde{f}_d\left(e_{R_1R_2}\right) = \text{Tri}\left(\tilde{f}_d^{\min}\left(e_{R_1R_2}\right),\ \tilde{f}_d^{\text{mean}}\left(e_{R_1R_2}\right),\ \tilde{f}_d^{\max}\left(e_{R_1R_2}\right)\right). \tag{5.7}$$

To get the fuzzy region distance feature for the example problem in Figure 5.3, we compute the overall and individual distance cost matrices using Algorithm 5.6. The distance grids for each boundary edge shown in Figure 5.4 are used to define the individual region cost matrices U^{d1} and U^{d2}, shown in Figure 5.5. The overall cost matrix C^d is computed using Equations 5.1 and 5.3. Using this as the input for the above equations gives a fuzzy region distance feature value of $\tilde{f}_d\left(e_{R_1R_2}\right) = \text{Tri}(1, 5.03, 9)$.

5.2.3 Region Terrain Type Features

The terrain type features measure the amount of distance traveled in each type of terrain. For the fully observed single-step features defined in Section 4.3.2, this is a value between 0 and 1 that depends only on the two terrain types t_1 and t_2. In Section 4.4.2, we consider the fuzzy case where we include the observability of each cell o_1 and o_2 and define the feature as a triangular fuzzy number that represents the minimum, maximum, and expected crisp feature values based on the prior likelihoods of each terrain type. For the region terrain type features, we extend this definition to account for the greater distance within each region. Equations 5.4-5.6 define the min, mean, and max values of the overall distance cost matrix C^d. As a shorthand, we notate these as C^d_{min}, C^d_{mean}, and C^d_{max}. For the individual region cost matrices U^{d1} and U^{d2}, we first determine the minimum distance from each grid cell to one of the boundary edges. We define these matrices as V^{d1} and V^{d2} where

$$V_i^{d1} = \min_{k=1,\ldots,K} U_{ik}^{d1}, \tag{5.8}$$

$$V_j^{d2} = \min_{k=1,\ldots,K} U_{jk}^{d2}, \tag{5.9}$$

and K is the number of boundary edges. The min, mean, and max values of these two matrices are given as V^{d1}_{min}, V^{d1}_{mean}, V^{d1}_{max}, and V^{d2}_{min}, V^{d2}_{mean}, V^{d2}_{max}, respectively. They represent the expected distances that an agent would need to travel to reach the nearest boundary edge from each cell within a region and assumes that the nearest boundary edge is the best option when moving to the adjacent region. We make this assumption to avoid computing an explicit probability distribution of which boundary edge is used for each pair of cells in R_1 and R_2. Such a distribution likely depends on other factors (such as elevation, which is

evaluated separately), and may be infeasible to compute accurately. The nearest boundary assumption is a simple and straightforward heuristic that works in most cases and provides a reasonable approximation of the required distances. Note that the minimum values $V_{\min}^{d1}$ and $V_{\min}^{d2}$ will always equal zero, since at least one of the cells in each region is already part of a boundary edge.

Because we have defined each region to be only a single terrain type or completely unobserved, we can use the same approach as Section 4.4.2, treating each region as one of the two adjacent cells, but multiplying by some measure of the size of each region. Let t_1^* and t_2^* be the true terrain types of the two regions and let T_{ki} be the event that $t_k^* = i$ for $k \in \{1, 2\}$. The probability that event T_{ki} occurs is defined as

$$p(T_{ki}) = \begin{cases} 1, & o_k = 1 \wedge t_k = i \\ 0, & o_k = 1 \wedge t_k \neq i \\ p(i), & o_k = 0 \end{cases}, \tag{5.10}$$

where $p(i)$ is the prior likelihood of observing terrain type i. The four possible state configurations that need to be considered are given as $S = \{s_{12}, s_1, s_2, s_0\}$, where

$$p(s_{12}) = p(T_{1i})p(T_{2i}), \tag{5.11}$$

$$p(s_1) = p(T_{1i})\big(1 - p(T_{2i})\big), \tag{5.12}$$

$$p(s_2) = \big(1 - p(T_{1i})\big)p(T_{2i}), \text{ and} \tag{5.13}$$

$$p(s_0) = \big(1 - p(T_{1i})\big)\big(1 - p(T_{2i})\big). \tag{5.14}$$

States that have a probability greater than zero have some chance of occurring and are added to the set of possible states,

$$S_{\text{pos}} = \{s \in S \mid p(s) > 0\}. \tag{5.15}$$

For each of the possible states, we consider the minimum, average, and maximum of the terrain type feature values. The minimum value for each state is defined as

$$f_{t(i)}^{\min}(s) = \begin{cases} 0, & s = s_0 \\ 0.5, & s = s_1 \vee s = s_2 \\ 1, & s = s_{12} \end{cases} . \tag{5.16}$$

This is equivalent to the single-step feature, since the minimum value occurs when the agent only needs to take one step across the boundary edge. The maximum value for each state is defined as

$$f_{t(i)}^{\max}(s) = \begin{cases} 0, & s = s_0 \\ V_{\max}^{d1} + 0.5, & s = s_1 \\ V_{\max}^{d2} + 0.5, & s = s_2 \\ C_{\max}^{d}, & s = s_{12} \end{cases} . \tag{5.17}$$

This uses the maximum values from each of the region cost matrices: $V_{\max}^{d1}$ when only region 1 is of terrain type i, $V_{\max}^{d2}$ when only region 2 is of terrain type i, and $C_{\max}^{d}$ when both regions are of terrain type i. When only one region is the appropriate terrain type, 0.5 is added to the feature value to include half of the cost of traveling the boundary edge. Since we may not know the true state if one or both regions are unobserved, the min and max overall terrain type feature values are given as the minimum and maximum of the costs for all possible states.

$$\tilde{f}_{t(i)}^{\min}\left(e_{R_1R_2}\right) = \min_{s \in S_{\text{pos}}} f_{t(i)}^{\min}(s) \tag{5.18}$$

$$\tilde{f}_{t(i)}^{\max}\left(e_{R_1R_2}\right) = \max_{s \in S_{\text{pos}}} f_{t(i)}^{\max}(s) \tag{5.19}$$

To get the average feature value, we sum up the mean distance costs multiplied by the expected likelihood of each state,

$$\tilde{f}_{t(i)}^{\text{mean}}(e_{R_1R_2}) = p(s_1)(V_{\text{mean}}^{d1} + 0.5) + p(s_2)(V_{\text{mean}}^{d2} + 0.5) + p(s_{12})C_{\text{mean}}^{d}. \tag{5.20}$$

Note that we do not need to consider s_0 since the feature value in this case would be zero. The resulting fuzzy region terrain type feature is

$$\tilde{f}_{t(i)}(e_{R_1R_2}) = \text{Tri}\left(\tilde{f}_{t(i)}^{\text{min}}(e_{R_1R_2}),\ \tilde{f}_{t(i)}^{\text{mean}}(e_{R_1R_2}),\ \tilde{f}_{t(i)}^{\text{max}}(e_{R_1R_2})\right). \tag{5.21}$$

For the example in Figure 5.3, the two terrain type features are computed as follows. Since both regions are observed, the true state is known with no uncertainty. For terrain type 1 (meadow), the state is s_1 and for terrain type 2 (forest) the state is s_2. For both of these states, $f_{t(i)}^{\text{min}} = 0.5$. With only one possible state, $\tilde{f}_{t(i)}^{\text{min}}(e_{R_1R_2}) = 0.5$. From Figure 5.5 we can see that $V_{\text{max}}^{d1} = 4$ and $V_{\text{max}}^{d2} = 4$. For both s_1 and s_2, $f_{t(i)}^{\text{max}} = 4.5$ and therefore $\tilde{f}_{t(i)}^{\text{max}}(e_{R_1R_2}) = 4.5$. The average values of the individual region cost matrices are computed as $V_{\text{mean}}^{d1} = 2$ and $V_{\text{mean}}^{d2} \approx 1.67$. Since there is no uncertainty, $\tilde{f}_{t(1)}^{\text{mean}}(e_{R_1R_2}) = 2.5$ and $\tilde{f}_{t(2)}^{\text{mean}}(e_{R_1R_2}) \approx 2.17$. The overall fuzzy region terrain type features are then defined as $\tilde{f}_{t(1)}(e_{R_1R_2}) = \text{Tri}(0.5, 2.5, 4.5)$ and $\tilde{f}_{t(2)}(e_{R_1R_2}) = \text{Tri}(0.5, 2.17, 4.5)$.

Consider now if both regions were unobserved. All four states would be possible and their likelihoods would be determined by the terrain type priors. Assume that $p(t_1) = 0.75$ and $p(t_2) = 0.25$. For terrain type 1, the state probabilities are computed as:

- $p(s_{12}) = (0.75)(0.75) \approx 0.56$
- $p(s_1) = (0.75)(1 - 0.75) \approx 0.19$
- $p(s_1) = (1 - 0.75)(0.75) \approx 0.19$
- $p(s_0) = (1 - 0.75)(1 - 0.75) \approx 0.06$

For terrain type 2, the state probabilities are computed as:

- $p(s_{12}) = (0.25)(0.25) \approx 0.06$
- $p(s_1) = (0.25)(1 - 0.25) \approx 0.19$
- $p(s_1) = (1 - 0.25)(0.25) \approx 0.19$
- $p(s_0) = (1 - 0.25)(1 - 0.25) \approx 0.56.$

The minimum feature value $\tilde{f}_{t(i)}^{\min}(e_{R_1R_2})$ would be 0, since s_0 has a nonzero probability for both terrain types. The maximum feature value $\tilde{f}_{t(i)}^{\max}(e_{R_1R_2})$ would be $C_{\max}^{d} = 9$, since s_{12} is possible for both terrain types. For the average value, we would use $V_{\text{mean}}^{d1} = 2$ and $V_{\text{mean}}^{d2} \approx 1.67$ as calculated before, and $C_{\text{mean}}^{d} \approx 5.03$. Using Equation 5.20, for terrain type 1 we compute

$$\tilde{f}_{t(1)}^{\text{mean}}(e_{R_1R_2}) \approx (0.19)(2 + 0.5) + (0.19)(1.67 + 0.5) + (0.56)(5.03) \approx 3.70.$$

and for terrain type 2 we compute

$$\tilde{f}_{t(2)}^{\text{mean}}(e_{R_1R_2}) \approx (0.19)(2 + 0.5) + (0.19)(1.67 + 0.5) + (0.06)(5.03) \approx 1.19.$$

Using Equation 5.21, the overall fuzzy region terrain type features are defined as $\tilde{f}_{t(1)}(e_{R_1R_2}) = \text{Tri}(0, 3.70, 9)$ and $\tilde{f}_{t(2)}(e_{R_1R_2}) = \text{Tri}(0, 1.19, 9)$. Comparing these features to the observed case, we see that being unable to observe the regions increases the overall uncertainty.

5.2.4 Region Terrain Transition Features

In the same way that the previous section extended the single-step terrain type features to compute region features, the fuzzy region directional terrain transition features $\tilde{f}_{t\langle i,j\rangle}(e_{R_1R_2})$ and symmetric terrain transition features $\tilde{f}_{t\{i,j\}}(e_{R_1R_2})$ are computed as an

extension of the definitions presented in Sections 4.3.3 and 4.4.3. Recall from Section 4.3.3 that the directional and symmetric terrain transition features always take binary values in the observable case. Given two terrain types i and j, the feature is 1 if the edge represents a transition from i to j and 0 otherwise. (The transition from j to i is also allowed in the symmetric feature version.) When one or both cells (now regions) are unobserved, the fuzzy feature is defined in Section 4.4.3 using the possibility and probability that the true state is the specified type. Consider first the fuzzy region directional terrain transition features $\tilde{f}_{t\langle i,j\rangle}(e_{R_1R_2})$ and symmetric terrain transition features $\tilde{f}_{t\{i,j\}}(e_{R_1R_2})$ when $i \neq j$. Since we have defined the terrain within each region to be uniform and have the restricted the agent to only cross the region boundary once, the only edge on a path from R_1 to R_2 that could have a different starting and ending terrain type is the boundary edge. Therefore, if $i \neq j$, the $\tilde{f}_{t\langle i,j\rangle}(e_{R_1R_2})$ and $\tilde{f}_{t\{i,j\}}(e_{R_1R_2})$ feature definitions are identical to those presented in Section 4.4.3 for the single-step case. The maximum value of the feature in this case is 1, regardless of the region sizes.

We start by defining the true terrain types of the two regions as t_1^* and t_2^*. Let T_{ki} be the event that $t_k^* = i$ and T_{kj} the event that $t_k^* = j$ for $k \in \{1,2\}$. The directional terrain transition feature is nonzero only when the environment state is (T_{1i}, T_{2j}), which occurs with probability $p(T_{1i})p(T_{2j})$. These values can be obtained from the observed terrain types and terrain priors using Equation 5.10. The symmetric terrain transition feature is nonzero for environment states (T_{1i}, T_{2j}) and (T_{1j}, T_{2i}), which occur with probability $p(T_{1i})p(T_{2j}) + p(T_{1j})p(T_{2i})$ if $i \neq j$ and $p(T_{1i})p(T_{2i})$ if $i = j$. For the case where $i \neq j$,

the fuzzy region directional terrain transition features are defined using the following equations based on those in Section 4.4.3.

$$\tilde{f}_{t\langle i,j\rangle}^{\min}\left(e_{R_1R_2}\right) = \begin{cases} 1, & p\left(T_{1i}\right)p\left(T_{2j}\right) = 1 \\ 0, & \text{otherwise} \end{cases} \tag{5.22}$$

$$\tilde{f}_{t\langle i,j\rangle}^{\max}\left(e_{R_1R_2}\right) = \begin{cases} 0, & p\left(T_{1i}\right)p\left(T_{2j}\right) = 0 \\ 1, & \text{otherwise} \end{cases} \tag{5.23}$$

$$\tilde{f}_{t\langle i,j\rangle}^{\text{mean}}\left(e_{R_1R_2}\right) = p\left(T_{1i}\right)p\left(T_{2j}\right) \tag{5.24}$$

For the symmetric terrain transition feature when $i \neq j$, the equations are as follows.

$$\tilde{f}_{t\{i,j\}}^{\min}\left(e_{R_1R_2}\right) = \begin{cases} 1, & p\left(T_{1i}\right)p\left(T_{2j}\right) + p\left(T_{1j}\right)p\left(T_{2i}\right) = 1 \\ 0, & \text{otherwise} \end{cases} \tag{5.25}$$

$$\tilde{f}_{t\{i,j\}}^{\max}\left(e_{R_1R_2}\right) = \begin{cases} 0, & p\left(T_{1i}\right)p\left(T_{2j}\right) + p\left(T_{1j}\right)p\left(T_{2i}\right) = 0 \\ 1, & \text{otherwise} \end{cases} \tag{5.26}$$

$$\tilde{f}_{t\{i,j\}}^{\text{mean}}\left(e_{R_1R_2}\right) = p\left(T_{1i}\right)p\left(T_{2j}\right) + p\left(T_{1j}\right)p\left(T_{2i}\right) \tag{5.27}$$

When $i = j$, both the directional and symmetric terrain transition features behave like the terrain type feature, essentially measuring the number of steps taken within the specified terrain type. The only real difference between the two is the handling of the region boundary transition. In the previous section, we added 0.5 to the cost values for states that only had one region of the specified terrain type to represent half the cost of crossing the region boundary. This meant that the boundary edge could have a cost of 0.5 instead of a binary value like the terrain transition features. Therefore, we redefine the equations from the previous section for the terrain transition features when $i = j$. Equations 5.10-5.15

remain the same, noting that $p(T_{ki}) = p(T_{kj})$ since $i = j$. The minimum feature value for each state is defined as

$$f_{tt(i)}^{\min}(s) = \begin{cases} 0, & s = s_0 \vee s = s_1 \vee s = s_2 \\ 1, & s = s_{12} \end{cases} \tag{5.28}$$

and the maximum feature values are defined as

$$f_{tt(i)}^{\max}(s) = \begin{cases} 0, & s = s_0 \\ V_{\max}^{d1}, & s = s_1 \\ V_{\max}^{d2}, & s = s_2 \\ C_{\max}^{d}, & s = s_{12}\,. \end{cases} \tag{5.29}$$

From this it follows that the minimum and maximum of the feature values are the minimum and maximum values of all possible states for both the directional and symmetric feature versions.

$$\tilde{f}_{t\langle i,j\rangle}^{\min}(e_{R_1R_2}) = \tilde{f}_{t\{i,j\}}^{\min}(e_{R_1R_2}) = \min_{s \in S_{\text{pos}}} f_{tt(i)}^{\min}(s) \tag{5.30}$$

$$\tilde{f}_{t\langle i,j\rangle}^{\max}(e_{R_1R_2}) = \tilde{f}_{t\{i,j\}}^{\max}(e_{R_1R_2}) = \max_{s \in S_{\text{pos}}} f_{tt(i)}^{\max}(s) \tag{5.31}$$

The mean value for both types is defined by multiplying the likelihood of each state by the mean distance costs and computing the sum.

$$\tilde{f}_{t\langle i,j\rangle}^{\text{mean}}(e_{R_1R_2}) = \tilde{f}_{t\{i,j\}}^{\text{mean}}(e_{R_1R_2}) = p(s_1)V_{\text{mean}}^{d1} + p(s_2)V_{\text{mean}}^{d2} + p(s_{12})C_{\text{mean}}^{d}. \tag{5.32}$$

The resulting fuzzy region directional terrain transition feature is given as

$$\tilde{f}_{t\langle i,j\rangle}(e_{R_1R_2}) = \text{Tri}\left(\tilde{f}_{t\langle i,j\rangle}^{\min}(e_{R_1R_2}),\ \tilde{f}_{t\langle i,j\rangle}^{\text{mean}}(e_{R_1R_2}),\ \tilde{f}_{t\langle i,j\rangle}^{\max}(e_{R_1R_2})\right), \tag{5.33}$$

and the fuzzy region symmetric terrain transition feature is given as

$$\tilde{f}_{t\{i,j\}}(e_{R_1R_2}) = \text{Tri}\left(\tilde{f}_{t\{i,j\}}^{\min}(e_{R_1R_2}),\ \tilde{f}_{t\{i,j\}}^{\text{mean}}(e_{R_1R_2}),\ \tilde{f}_{t\{i,j\}}^{\max}(e_{R_1R_2})\right). \tag{5.34}$$

For the example in Figure 5.3, the fuzzy region directional terrain transition features are defined as

- $\tilde{f}_{t\langle 1,1\rangle}(e_{R_1R_2}) = \text{Tri}(0, 2, 4)$,
- $\tilde{f}_{t\langle 1,2\rangle}(e_{R_1R_2}) = \text{Tri}(1, 1, 1)$,
- $\tilde{f}_{t\langle 2,1\rangle}(e_{R_1R_2}) = \text{Tri}(0, 0, 0)$,
- $\tilde{f}_{t\langle 2,2\rangle}(e_{R_1R_2}) = \text{Tri}(0, 1.67, 4)$,

and the symmetric terrain transition features are defined as

- $\tilde{f}_{t\{1,1\}}(e_{R_1R_2}) = \text{Tri}(0, 2, 4)$,
- $\tilde{f}_{t\{1,2\}}(e_{R_1R_2}) = \text{Tri}(1, 1, 1)$,
- $\tilde{f}_{t\{2,2\}}(e_{R_1R_2}) = \text{Tri}(0, 1.67, 4)$.

Note that when $i = j$, both feature versions are 0.5 less than the corresponding terrain type feature in the previous section. When $i \neq j$, the feature is a crisp binary value indicating if the terrain transition is of the appropriate type.

If we consider the situation where both regions are unobserved as in the previous section with the same terrain priors, $p(t_1) = 0.75$ and $p(t_2) = 0.25$, the fuzzy region terrain transition features are defined as

- $\tilde{f}_{t\langle 1,1\rangle}(e_{R_1R_2}) = \tilde{f}_{t\{1,1\}}(e_{R_1R_2}) = \text{Tri}(0, 3.52, 9)$,
- $\tilde{f}_{t\langle 2,2\rangle}(e_{R_1R_2}) = \tilde{f}_{t\{2,2\}}(e_{R_1R_2}) = \text{Tri}(0, 1.00, 9)$,
- $\tilde{f}_{t\langle 1,2\rangle}(e_{R_1R_2}) = \tilde{f}_{t\langle 2,1\rangle}(e_{R_1R_2}) = \text{Tri}(0, 0.19, 1)$,
- $\tilde{f}_{t\{1,2\}}(e_{R_1R_2}) = \text{Tri}(0, 0.38, 1)$.

Note that the features where $i = j$ have slightly lower mean values than the corresponding terrain type features in the previous section. This comes from the possibility that only one region is of the specified type and the 0.5 cost of the boundary edge is not incurred. The symmetric $\tilde{f}_{t\{1,2\}}(e_{R_1R_2})$ feature also has a mean value that is the sum of the two directional variants, indicating that both terrain configurations would contribute to the feature value.

5.3 General Fuzzy Region Features

In the previous section, we defined the graph G_{12} for every pair of adjacent regions R_1 and R_2 in the region graph G_R. By assigning a uniform cost of 1 to each edge of G_{12}, we computed the distance cost matrices C^d, U^{d1}, and U^{d2}, and used these to compute the distance and terrain-based fuzzy region features between the two regions. For the elevation feature, we can no longer assume that each edge has a uniform weight since the cost is defined as the difference in elevation between adjacent grid cells. Because of this, we introduce a more generic algorithm in this section for computing the cost matrices that can handle non-uniform edge weights.

5.3.1 General Framework for Computing Region Features

The three subgraphs of G_{12} are G_1, G_2, and G_{bnd}, where G_1 contains only the vertices from R_1, G_2 contains only the vertices from R_2, and G_{bnd} contains the boundary edges. In practice, we represent the three subgraphs G_1, G_2, or G_{bnd} as edge sets, where each edge e is a 4-tuple (i_1, j_1, i_2, j_2). The pair (i_1, j_1) indicates the starting cell, $\text{START}(e)$, and (i_2, j_2) is the ending cell, $\text{END}(e)$. The edges for the G_1 and G_2 subgraphs are separated by direction

into four sets: *up*, *down*, *left*, and *right*. This makes it straightforward to define the edge sets and allows the shortest path algorithm to be optimized for grid world domains.

Algorithm 5.9 Create Region Edge Sets

CREATE_REGION_EDGE_SETS(R)
1: $(n, m) \leftarrow$ size of R
2: $I_1, I_2 \leftarrow$ GET_REGION_INDICES(R) *// Algorithm 5.7*
3: $E_1, E_2 \leftarrow$ empty structures
4: **for** r in $\{1, 2\}$
5: $E_r.up \leftarrow \{(i_1, j_1, i_2, j_2) \mid (i_1, j_1) \in I_r \wedge (i_2, j_2) \in I_r \wedge i_2 = i_1 - 1\}$
6: $E_r.down \leftarrow \{(i_1, j_1, i_2, j_2) \mid (i_1, j_1) \in I_r \wedge (i_2, j_2) \in I_r \wedge i_2 = i_1 + 1\}$
7: $E_r.left \leftarrow \{(i_1, j_1, i_2, j_2) \mid (i_1, j_1) \in I_r \wedge (i_2, j_2) \in I_r \wedge j_2 = j_1 - 1\}$
8: $E_r.right \leftarrow \{(i_1, j_1, i_2, j_2) \mid (i_1, j_1) \in I_r \wedge (i_2, j_2) \in I_r \wedge j_2 = j_1 + 1\}$
9: $E_{\text{bnd}} \leftarrow$ GET_BOUNDARY_EDGES(n, m, I_1, I_2) *// Algorithm 5.8*
10: **return** E_1, E_2, E_{bnd}

Algorithm 5.9 gives the procedure for creating the edge sets E_1, E_2, and E_{bnd} for each of the three subgraphs using the function CREATE_REGION_EDGE_SETS. The algorithm takes a region map R as input, where cells are labeled 1 for R_1, 2 for R_2, and 0 elsewhere. The indices for each region are found on line 2 using Algorithm 5.7, which provides an ordering that is consistant with the distance cost matrices computed in the previous section. Lines 3-8 create the directional edge sets E_1 and E_2 by identifying the adjacent grid cells in each direction. Note that these sets do not need to be ordered. Line 9 gets the boundary edges using Algorithm 5.8, which also maintains the same ordering as the previous section. These three edge sets are returned on line 10 and are used in conjunction with the attributes of the mental map to compute the fuzzy features between the two regions.

The general algorithm for computing all the region features for a given edge of the region graph is given in Algorithm 5.10. The COMPUTE_REGION_FEATURES function is called on line 20 of the CREATE_REGION_GRAPH function from Algorithm 5.5 and takes the current mental map structure $\mathcal{M}$ and the region map R as inputs. The first half of the function computes the distance and terrain-based features from the previous section. Lines 1-5 get the terrain types and observability of each region and line 6 initializes an empty structure to hold the features. Line 7 gets the region distance matrices using Algorithm 5.6, which will be used to compute many of the features. Line 8 computes the distance feature using the formulas from Section 5.2.2. Lines 9-14 loop over each terrain type in the set of all terrain types, $\mathcal{M}.\mathcal{T}$. The terrain type feature from Section 5.2.3 is computed on line 10 using the distance matrices computed previously by the GET_REGION_DISTANCE function. Lines 11-14 loop again over each terrain type to compute the terrain transition features from Section 5.2.4. The directional terrain transition features are computed on line 12, and if $i \leq j$, then the symmetrical features are also computed on line 14. It is possible to skip any of these feature computations if they are not required by the problem.

Algorithm 5.10 Compute Region Features

COMPUTE_REGION_FEATURES($\mathcal{M}$, R)

1: $(n, m) \leftarrow \mathcal{M}.size$

2: $t_1 \leftarrow \{\mathcal{M}.T[i,j] \mid 1 \le i \le n \wedge 1 \le j \le m \wedge R[i,j] = 1\}$

3: $t_2 \leftarrow \{\mathcal{M}.T[i,j] \mid 1 \le i \le n \wedge 1 \le j \le m \wedge R[i,j] = 2\}$

4: $o_1 \leftarrow \{\mathcal{M}.V[i,j] \mid 1 \le i \le n \wedge 1 \le j \le m \wedge R[i,j] = 1\}$

5: $o_2 \leftarrow \{\mathcal{M}.V[i,j] \mid 1 \le i \le n \wedge 1 \le j \le m \wedge R[i,j] = 2\}$

6: $F \leftarrow$ empty structure

/ Compute distance cost matrices */*

7: C^d, U^{d1}, U^{d2} ← GET_REGION_DISTANCE(R) *// Algorithm 5.6*

/ Compute distance and terrain-based features (Sections 5.2.2-5.2.4) */*

8: $F.\tilde{f}_d$ ← DISTANCE_FEATURE(C^d)

9: **for** i in 1 to $|\mathcal{M}.\mathcal{T}|$

10: $F.\tilde{f}_{t(i)}$ ← TERRAIN_TYPE_FEATURE(i, C^d, U^{d1}, U^{d2}, t_1, t_2, o_1, o_2)

11: **for** j in 1 to $|\mathcal{M}.\mathcal{T}|$

12: $F.\tilde{f}_{t\langle i,j\rangle}$ ← DIR_TERRAIN_FEATURE(i, j, C^d, U^{d1}, U^{d2}, t_1, t_2, o_1, o_2)

13: **if** $i \le j$

14: $F.\tilde{f}_{t\{i,j\}}$ ← SYM_TERRAIN_FEATURE(i, j, C^d, U^{d1}, U^{d2}, t_1, t_2, o_1, o_2)

/ Get edge sets */*

15: E_1, E_2, E_{bnd} ← CREATE_REGION_EDGE_SETS(R) *// Algorithm 5.9*

/ Compute elevation edge costs (Algorithm 5.11) */*

16: E_{1_abs}, E_{2_abs}, $E_{\text{bnd_abs}}$ ← GET_ELEVATION_EDGE_COSTS($\mathcal{M}.E$, E_1, E_2, E_{bnd}, "*abs*")

17: $E_{1_\uparrow}$, $E_{2_\uparrow}$, $E_{\text{bnd}_\uparrow}$ ← GET_ELEVATION_EDGE_COSTS($\mathcal{M}.E$, E_1, E_2, E_{bnd}, "*up*")

18: $E_{1_\downarrow}$, $E_{2_\downarrow}$, $E_{\text{bnd}_\downarrow}$ ← GET_ELEVATION_EDGE_COSTS($\mathcal{M}.E$, E_1, E_2, E_{bnd}, "*down*")

/ Compute elevation features (Algorithm 5.12) */*

19: $F.\tilde{f}_{h_\text{max}}$ ← ELEV_FEATURE(R, U^{d1}, U^{d2}, o_1, o_2, E_{1_abs}, E_{2_abs}, $E_{\text{bnd_abs}}$, "*abs*", "*max*")

20: $F.\tilde{f}_{h\uparrow_\text{max}}$ ← ELEV_FEATURE(R, U^{d1}, U^{d2}, o_1, o_2, $E_{1_\downarrow}$, $E_{2_\uparrow}$, $E_{\text{bnd}_\uparrow}$, "*up*", "*max*")

21: $F.\tilde{f}_{h\downarrow_\text{max}}$ ← ELEV_FEATURE(R, U^{d1}, U^{d2}, o_1, o_2, $E_{1_\uparrow}$, $E_{2_\downarrow}$, $E_{\text{bnd}_\downarrow}$, "*down*", "*max*")

22: $F.\tilde{f}_{h_\text{sum}}$ ← ELEV_FEATURE(R, U^{d1}, U^{d2}, o_1, o_2, E_{1_abs}, E_{2_abs}, $E_{\text{bnd_abs}}$, "*abs*", "*sum*")

23: $F.\tilde{f}_{h\uparrow_\text{sum}}$ ← ELEV_FEATURE(R, U^{d1}, U^{d2}, o_1, o_2, $E_{1_\downarrow}$, $E_{2_\uparrow}$, $E_{\text{bnd}_\uparrow}$, "*up*", "*sum*")

24: $F.\tilde{f}_{h\downarrow_\text{sum}}$ ← ELEV_FEATURE(R, U^{d1}, U^{d2}, o_1, o_2, $E_{1_\uparrow}$, $E_{2_\downarrow}$, $E_{\text{bnd}_\downarrow}$, "*down*", "*sum*")

25: **return** F

The second half of Algorithm 5.10 computes the elevation features between the two specified regions. This process begins on line 15, where the edge sets E_1, E_2, and E_{bnd} are constructed using the CREATE_REGION_EDGE_SETS function from Algorithm 5.9. These sets provide the starting and ending grid cells for the edges in each region and the boundary set. On lines 16-19, we append these edges with the elevation features defined in Chapter 4. This is accomplished by the GET_ELEVATION_EDGE_COSTS function in Algorithm 5.11, which is called three times. Each function call computes a different feature: the absolute value of the elevation difference, the uphill difference, or the downhill difference.

The inputs to Algorithm 5.11 are the heightmap H from the mental map, the edge sets E_1, E_2, and E_{bnd}, and a *type* flag indicating which elevation feature to compute. Lines 1-14 compute the crisp elevation features for the two region edge sets. Since each region is either completely observed or unobserved, the feature values of each edge will either be known exactly or be unknown with maximum uncertainty. If the region is unobserved, the feature value of each edge is set to NIL (lines 5-6). We will discuss how unobserved regions are handled in more detail in the next section. In each direction, the edge (i_1, j_1, i_2, j_2) is used to compute the appropriate feature value c using Equations 4.11-4.13 (lines 7-12). These features are appended to the edges creating a 5-tuple (i_1, j_1, i_2, j_2, c), which is saved back to the edge set (lines 13-14). Note that it is not necessary to maintain the edge order within each of the region edge sets.

Algorithm 5.11 Get Elevation Edge Costs

```
GET_ELEVATION_EDGE_COSTS(H, E1, E2, Ebnd, type)

    /* Get costs for each region */
1:  for r in {1, 2}
2:      for dir in {up, down, left, right}
3:          F ← ∅
4:          for each (i1, j1, i2, j2) ∈ Er.{dir}
5:              if H[i1, j1] = NIL ∨ H[i2, j2] = NIL
6:                  c ← NIL
7:              else if type = "abs"
8:                  c ← |H[i1, j1] – H[i2, j2]|                    // Equation 4.11
9:              else if type = "up"
10:                 c ← max(0, H[i2, j2] – H[i1, j1])              // Equation 4.12
11:             else if type = "down"
12:                 c ← max(0, H[i1, j1] – H[i2, j2])              // Equation 4.13
13:             F ← F ∪ (i1, j1, i2, j2, c)
14:         Er.{dir} ← F

    /* Get boundary edge costs */
15: for k in 1 to |Ebnd|
16:     (i1, j1, i2, j2) ← Ebnd[k]
17:     h1 ← H[i1, j1]
18:     h2 ← H[i2, j2]
19:     o1 ← [H[i1, j1] ≠ NIL]
20:     o2 ← [H[i2, j2] ≠ NIL]
21:     if type = "abs"
22:         c ← f̃h(h1, h2, o1, o2)                                 // Equation 4.71
23:     else if type = "up"
24:         c ← f̃h↑(h1, h2, o1, o2)                                // Equation 4.72
25:     else if type = "down"
26:         c ← f̃h↓(h1, h2, o1, o2)                                // Equation 4.73
27:     Ebnd[k] ← (i1, j1, i2, j2, c)

28: return E1, E2, Ebnd
```

The elevation features for the boundary edges are computed on lines 15-27 of Algorithm 5.11. The order of these edges is maintained, and for each edge, we determine the starting and ending heights and observability (lines 16-20). These values are used to compute the appropriate fuzzy elevation features using Equations 4.71-4.73 (lines 21-26). Note that unlike the edge sets for each region, it is possible for only one side to be observed. Therefore, we save the complete fuzzy feature for each boundary edge, represented as a triangular fuzzy number. In practice, we only save the min, mean, and max points used to define the membership function. Once all the edge features have been computed, the updated edge sets are returned on line 28.

Each call to the GET_ELEVATION_EDGE_COSTS function on lines 16-18 of Algorithm 5.10 returns three edge sets with either the absolute, uphill, or downhill elevation difference features appended to each edge. These are saved as the sets E_{1_abs}, E_{2_abs}, and E_{bnd_abs} for the absolute elevation difference, $E_{1_\uparrow}$, $E_{2_\uparrow}$, and $E_{bnd_\uparrow}$ for the uphill elevation difference, and $E_{1_\downarrow}$, $E_{2_\downarrow}$, and $E_{bnd_\downarrow}$ for the downhill elevation difference. The actual elevation features are computed on lines 19-24 using different subsets of these edge sets, which will be discussed in the following sections. Note that for the uphill and downhill elevation features, the edge set for region 1 is opposite that of the boundary edges and region 2. This is done because costs are aggregated in the direction moving away from the boundary edge. For region 1, this is opposite of the direction of agent movement, so the edge set is replaced with that of the other elevation difference feature. This works because the uphill cost in one direction is equal to the downhill cost in the opposite direction. As with the distance and terrain type features, any features not required by the problem can be

skipped. The final set of features for this edge of the region graph is returned on line 25 to the CREATE_REGION_GRAPH function in Algorithm 5.5.

Figure 5.6 shows the elevation edge costs computed for the example in Figure 5.3. The three images show the absolute, uphill, and downhill elevation difference features. These are saved in the corresponding edge sets and separated by edge direction. Note that the uphill costs in each direction are equal to the downhill costs in the opposite direction.

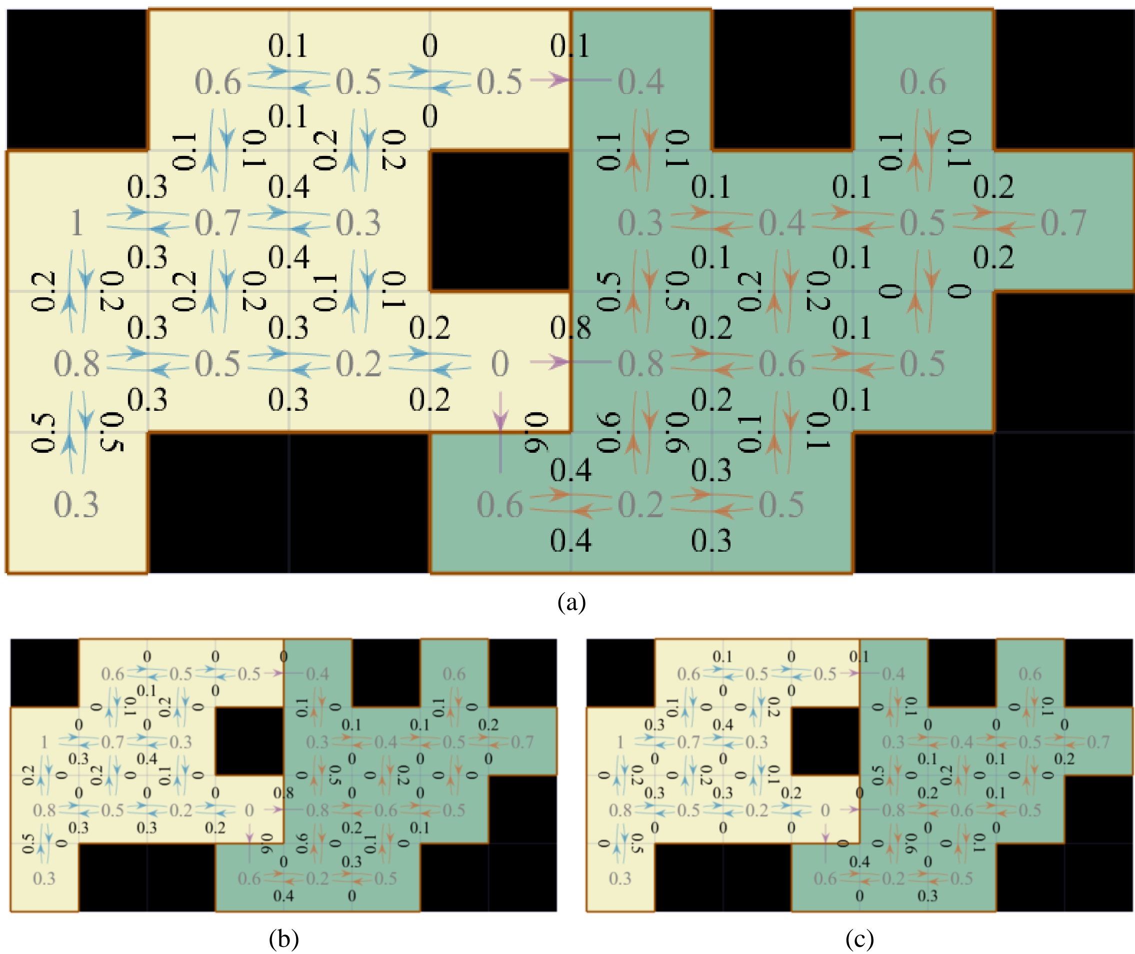


Figure 5.6 Elevation edge costs computed for the example in Figure 5.3. The elevation of each cell is shown in gray and the edge costs are displayed next to each edge. (a) Absolute elevation difference. (b) Uphill elevation difference. (c) Downhill elevation difference.

5.3.2 Region Elevation Features

As mentioned previously, computing the region elevation features requires a generalization of the distance cost algorithm presented in the last section to account for non-uniform edge weights. Algorithm 5.12 shows the approach we use to compute the elevation features, which is very much like the GET_REGION_DISTANCE function in Algorithm 5.6. The function takes the following input arguments:

- an $n \times m$ grid map R, where cells in R_1 are marked 1, cells in R_2 are marked 2, and all other cells are 0,
- region distance matrices U^{d1} and U^{d2}, obtained as the outputs of Algorithm 5.6 on the grid map R,
- the observability of the two regions o_1 and o_2,
- weighted edge sets E_1, E_2, and E_{bnd}, obtained as the outputs of Algorithm 5.11,
- a *type* parameter set to either "*abs*", "*up*", or "*down*" to indicate which elevation feature to compute, and
- an *agg* parameter set to either "*sum*" or "*max*" to indicate if summation or maximization aggregation should be used.

The algorithm starts by obtaining the indices of the two regions using Algorithm 5.7 (line 2) and initializing the cost matrices (lines 3-5). U^1 and U^2 will hold the expected aggregated elevation feature costs from each cell in R_1 and R_2 respectively to each boundary edge. U^{bnd} will hold the three triangular fuzzy number parameters (min, mean, and max) for each boundary edge.

Algorithm 5.12 Elevation Feature

ELEV_FEATURE(R, U^{d1}, U^{d2}, o_1, o_2, E_1, E_2, E_{bnd}, *type*, *agg*)

```
1:  (n, m) ← size of R

    /* Get the indices of the two regions */
2:  I1, I2 ← GET_REGION_INDICES(R)                                  // Algorithm 5.7

    /* Initialize the cost matrices */
3:  U^1 ← |I1| × |Ebnd| matrix initalized to ∞
4:  U^2 ← |I2| × |Ebnd| matrix initalized to ∞
5:  U^bnd ← |Ebnd| × 3 matrix initalized to ∞

    /* Compute region costs */
6:  for r in {1, 2}
7:      if o_r = 1
8:          for k in 1 to |Ebnd|
9:              (i1, j1, i2, j2, c) ← Ebnd[k]
10:             D ← n × m matrix initalized to ∞
11:             D[i_r, j_r] ← 0
12:             D ← BELLMAN_FORD_GRID_DIST(D, E_r, agg)              // Algorithm 5.13
13:             U^r[:, k] ← D[I_r]
14:     else
15:         U^r ← UNOBSERVED_ELEVATION_COST(U^dr, type, agg)       // Algorithm 5.14

    /* Get boundary edge costs */
16: for k in 1 to |Ebnd|
17:     (i1, j1, i2, j2, Tri(f_min, f_mean, f_max)) ← Ebnd[k]
18:     U^bnd[k, 1] ← f_min
19:     U^bnd[k, 2] ← f_mean
20:     U^bnd[k, 3] ← f_max

    /* Compute the feature (Algorithm 5.15) */
21: F ← COMBINE_ELEVATION_COSTS(U^1, U^2, U^bnd, U^d1, U^d2, o1, o2, type, agg)

22: return F
```

The first part of Algorithm 5.12 computes the region costs to fill in the U^1 and U^2 matrices. For each region that is observed, we cycle over each boundary edge and compute the costs from that edge to each grid cell in the region. Unobserved regions are treated as a special case and are discussed in the next section. Whereas Algorithm 5.6 used the GRID_DISTANCE function from Algorithm 3.6 for each boundary edge in both regions, we rely here on a variation of the Bellman-Ford algorithm presented in Algorithm 5.13, which allows for non-uniform edge weights. Lines 9-11 prepare a distance grid D for the algorithm, where all values in D are set to infinity except for the source grid cell, which is set to zero at one of the boundary edge cells.

The Bellman-Ford algorithm (Bellman 1958; Ford Jr. 1956) operates by iteratively relaxing an upper bound on the cost to each vertex from some source. Each vertex starts with an initial value of infinity except for the source, which starts with zero. The entire set of edges E is evaluated for a maximum of $|V| - 1$ iterations, and each time an edge is found that results in a smaller cost to reach a vertex, the upper bound is relaxed. The algorithm can terminate early if no edges are relaxed in an iteration. The worst-case runtime performance is given as $O(|V||E|)$, but the best-case performance is only $O(|E|)$. Our implementation of the Bellman-Ford algorithm is specifically designed to operate on grid graphs, where the edges can be divided into four sets: *up*, *down*, *left*, and *right*. This allows the cost updates to occur simultaneously for each set. This is possible because within each set, each vertex has at most one incoming edge that could change its current value. In practice, this results in a wave propagation of settled costs radiating outward from the source, similar to the breadth-first search approach of the Lee algorithm (Lee 1961) and our previous GRID_DISTANCE function.

Algorithm 5.13 Bellman-Ford Grid Distance

```
BELLMAN_FORD_GRID_DIST(D, E, agg)
    /* D is a grid initialized to ∞ with source cells set to 0 */

1:  (n, m) ← size of D
2:  D_old ← D
3:  while True

        /* Loop over the edges in each direction */
4:      for dir in {up, down, left, right}
5:          for each (i1, j1, i2, j2, c) ∈ E.{dir}
6:              s ← D[i1, j1]                // Get value of D at edge starting point
7:              t ← D[i2, j2]                // Get value of D at edge ending point
8:              if agg = "sum"
9:                  u ← s + c
10:             else if agg = "max"
11:                 u ← max(s, c)
12:             D[i2, j2] ← min(u, t)

        /* Check if finished */
13:     if D = D_old
14:         break
15:     else
16:         D_old ← D

17: return D
```

The BELLMAN_FORD_GRID_DIST function presented in Algorithm 5.13 takes the following inputs:

- a distance grid D, where all cells have been initialized to infinity except for the source cell, which is set to zero,
- a set of edges E, that has been subdivided into four sets, *up*, *down*, *left*, and *right*, based on edge direction, and

- an option parameter *agg* that indicates if summation or maximization aggregation is to be used.

The distance grid D represents an upper bound on the minimum cost required to reach each cell from the source. As the algorithm proceeds, the values in D are replaced with better estimates. The main loop of Algorithm 5.13 begins on line 3 and continues until D does not change, which is checked on lines 13-16. For each main loop iteration, each of the four directional edge sets is evaluated in sequence (lines 4-12). Lines 5-12 loop over each edge in the edge set. For each edge, we get the current values in D of the starting cell (i_1, j_1) and ending cell (i_2, j_2), saved as the variables s and t (lines 6 and 7). The cost of the edge is given as c and is aggregated with the value s, which represents the best-known cost from the source to (i_1, j_1). If using summation aggregation, this is evaluated as $s + c$ (line 9), whereas it is evaluated as $\max(s, c)$ if using maximization aggregation (line 11). The resulting value is saved as the variable u and is compared with t, which represents the best-known cost from the source to (i_2, j_2). If u is less than t, then the edge offers a better path to (i_2, j_2). The value $D[i_2, j_2]$ is updated to be the minimum of u and t on line 12. Note that we do not save the shortest paths themselves, but only the costs associated with the paths.

Since we only consider edges in one direction at a time, each cell can have only a single incoming edge and a single outgoing edge. This ensures that there are no conflicts when the values in D are updated and allows lines 5-12 to operate in parallel, which can greatly improve the speed of the algorithm. If no values in D have changed after evaluating the edges in each direction (checked on lines 13-16), the algorithm terminates and D is returned on line 17.

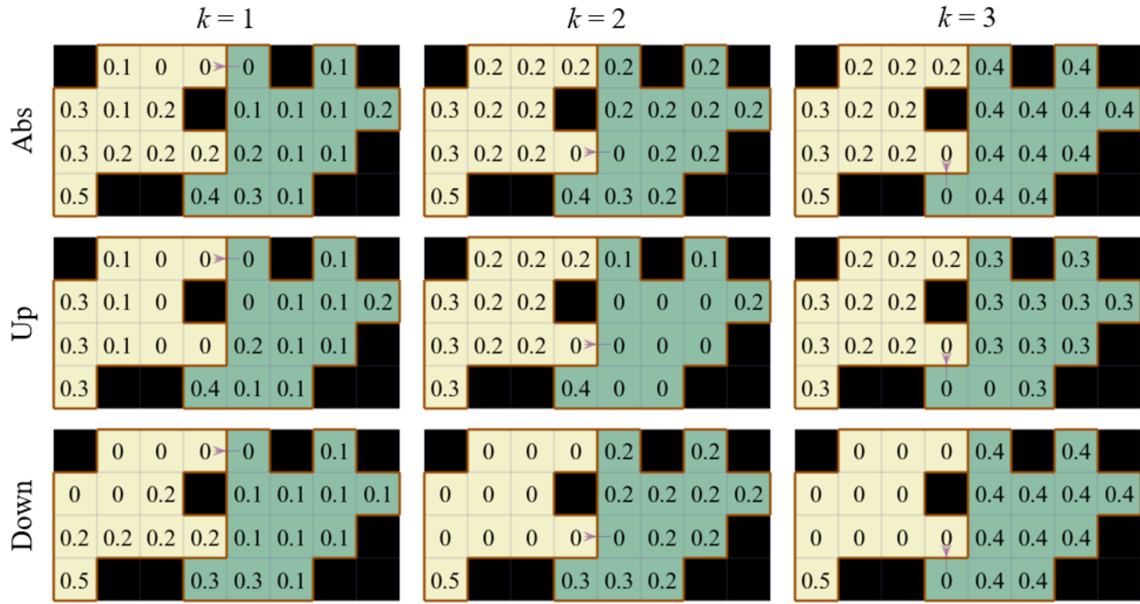


Figure 5.7 Composite distance grids computed using Algorithm 5.13 for the example in Figure 5.3 using the maximum aggregation method. The top row shows the absolute elevation difference feature costs, the middle row shows the uphill costs, and the bottom row shows the downhill costs. The three columns show the different costs for reaching each of the three boundary edges.

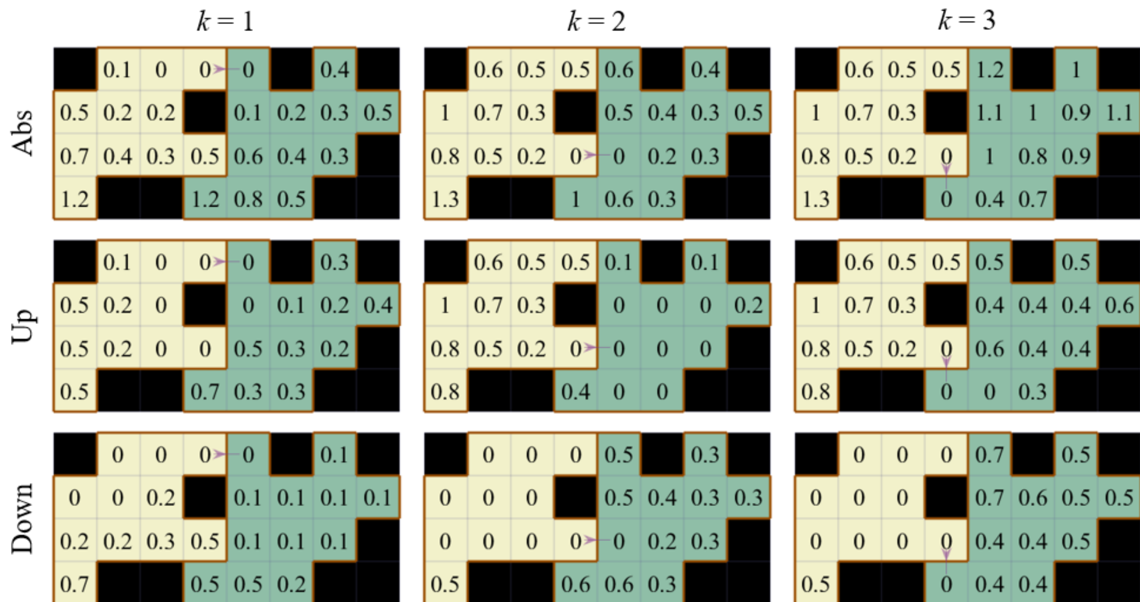


Figure 5.8 Composite distance grids computed using Algorithm 5.13 for the example in Figure 5.3 using the summation aggregation method. The top row shows the absolute elevation difference feature costs, the middle row shows the uphill costs, and the bottom row shows the downhill costs. The three columns show the different costs for reaching each of the three boundary edges.

Figure 5.7 shows the output of the BELLMAN_FORD_GRID_DIST function on the example from Figure 5.3 using the maximization aggregation method. Likewise, Figure 5.8 shows the output when using summation. Each image shows the distance values computed for each grid cell in both regions using a specific boundary edge and feature type. The absolute, uphill, and downhill elevation difference features are shown. It can be helpful to reference the elevation edge costs computed in Figure 5.6 when examining these figures.

5.3.3 Unobserved Elevation Costs

We now return to the first part of Algorithm 5.12 where we compute the individual region elevation costs for each boundary edge. If the region is observed, the edge costs are computed using the BELLMAN_FORD_GRID_DIST function in Algorithm 5.13. However, if the region is unobserved, then each edge in the region will have an unknown cost. In this case, all edges can be assigned a fuzzy cost using the equations in Section 4.4.4. Because each edge has the same fuzzy cost value, we can make use of the distance cost matrices computed by the GET_REGION_DISTANCE function from Algorithm 5.6.

Consider first the average cost value of reaching one of the boundary edges from each grid cell within a region. The matrices U^{d1} and U^{d2} give the number of steps required to reach each boundary edge from any location within one of the regions. In general, n steps are required where $n \geq 0$. Using the values computed in Equations 4.65-4.67, we know that the mean elevation feature value for a single unobserved edge is $\frac{1}{3}$ for the absolute elevation difference and $\frac{1}{6}$ for both the uphill and downhill elevation differences.

Therefore, for the summation aggregation method, the average total cost to reach one of the boundary edges is $\frac{1}{3}U^d$ for the absolute elevation difference and $\frac{1}{6}U^d$ for the uphill and downhill elevation differences.

For maximization, the approach is not so straightforward. The average cost of reaching each boundary edge using maximization aggregation is the expected maximum value of n randomly sampled elevation feature values. Consider a set of n independent[1] and identically distributed (i.i.d.) random variables $X_1, X_2, \dots, X_n$ where each variable X_i is sampled from a probability distribution $f_X(x)$. Let $Y_n = \max\{X_1, X_2, \dots, X_n\}$ be the maximum value of the set. We can define the expected value of Y_n as

$$\mathbb{E}[Y_n] = \int_{-\infty}^{\infty} y f_{Y_n}(y)\, dy, \tag{5.35}$$

where $f_{Y_n}(y)$ is the probability distribution function (PDF) of Y_n. If the PDF is continuous, it can be computed as

$$f_{Y_n}(y) = \frac{d}{dy} F_{Y_n}(y), \tag{5.36}$$

where $F_{Y_n}(y)$ is the cumulative distribution function (CDF) of Y_n, which is defined as

$$F_{Y_n}(y) = P(Y_n \le y). \tag{5.37}$$

Replacing Y_n with its definition, we get

$$F_{Y_n}(y) = P(\max\{X_1, X_2, \dots, X_n\} \le y). \tag{5.38}$$

[1] The elevation features within a region are not actually independent since they depend on the shared heights of the grid cells. The heightmap is also generated in such a way that adjacent cells are more likely to have the same elevation, biasing the elevation difference features toward zero. However, we assume independence here for the sake of analysis and recognize that the resulting estimate will likely be larger than the true value.

Since the X_i variables are independent, this can be expressed as

$$F_{Y_n}(y) = P(X_1 \leq y)P(X_2 \leq y) \dots P(X_n \leq y) \tag{5.39a}$$

$$F_{Y_n}(y) = [P(X_i \leq y)]^n \tag{5.39b}$$

$$F_{Y_n}(y) = [F_X(y)]^n. \tag{5.39c}$$

Here, $F_X(y)$ is the CDF of the random variable X, which is one of the elevation feature values.

Each of the elevation features introduced in Section 4.3.4 are defined in terms of the starting and ending grid cell heights, h_1 and h_2. In the completely unobserved case, both values are assumed to be randomly sampled from a uniform distribution over the unit interval $h_1, h_2 \sim U(0, 1)$. Let X_h be the absolute elevation difference feature computed from h_1 and h_2 such that

$$X_h = |h_1 - h_2|. \tag{5.40}$$

The CDF of X_h is defined as

$$F_{X_h}(x) = P(X_h \leq x). \tag{5.41}$$

The easiest way to evaluate this expression is to imagine a unit square representing all possible values of the pair (h_1, h_2). The 3D surface plots of the elevation difference features were shown in Figure 4.6, and a top-down view is shown in Figure 5.9. For any value x, the area of the square where $|h_1 - h_2| \leq x$ represents the probability that $X_h \leq x$. From Figure 5.9 (a), we can see that this area is $1 - (1 - x)^2$. Simplifying this expression gives,

$$F_{X_h}(x) = 2x - x^2, \;\; 0 \leq x \leq 1. \tag{5.42}$$

For the directional elevation difference features, we define $X_{h\uparrow}$ and $X_{h\downarrow}$ as

$$X_{h\uparrow} = \max(0, h_2 - h_1)\,, \text{and} \tag{5.43}$$

$$X_{h\downarrow} = \max(0, h_1 - h_2). \tag{5.44}$$

Again, the CDFs of $X_{h\uparrow}$ and $X_{h\downarrow}$ are defined as

$$F_{X_{h\uparrow}}(x) = P(X_{h\uparrow} \le x), \text{and} \tag{5.45}$$

$$F_{X_{h\downarrow}}(x) = P(X_{h\downarrow} \le x). \tag{5.46}$$

From Figure 5.9 (b) and (c), we see that the areas of the unit square where $\max(0, h_2 - h_1) \le x$ and $\max(0, h_1 - h_2) \le x$ are both $1 - \frac{1}{2}(1-x)^2$. Simplifying gives

$$F_{X_{h\uparrow}}(x) = F_{X_{h\downarrow}}(x) = -\frac{x^2}{2} + x + \frac{1}{2},\ \ 0 \le x \le 1. \tag{5.47}$$

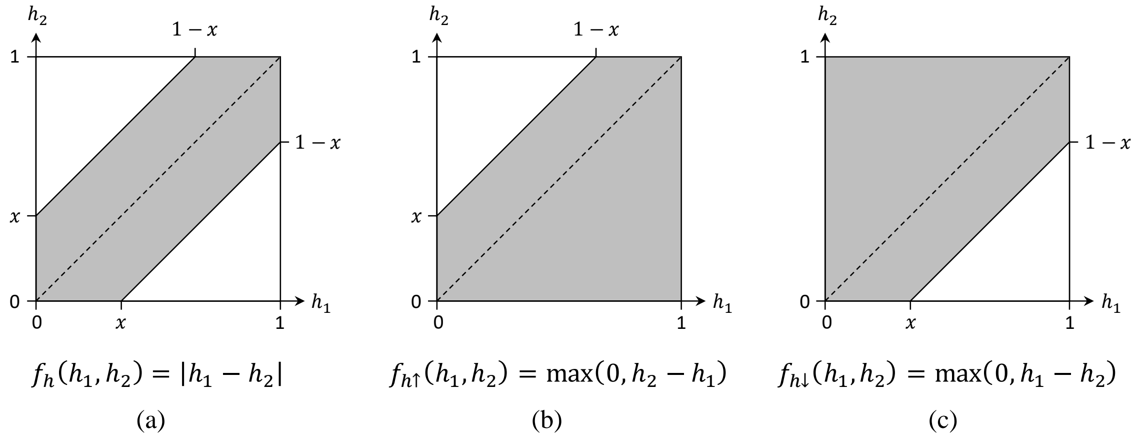


Figure 5.9 Plots of the elevation difference features over the unit square, with a shaded region showing the area where the function is less than a value x. (a) The absolute elevation difference f_h. (b) The uphill elevation difference $f_{h\uparrow}$. (c) The downhill elevation difference $f_{h\downarrow}$. These represent top-down views of the 3D surface plots shown in Figure 4.6.

Since the CDFs of the uphill and downhill elevation difference features are identical, we simplify our notation and refer to the two types of features as absolute and

directional elevation difference features. We notate these two CDFs as $F_{X_{\text{abs}}} = F_{X_h}$ and $F_{X_{\text{dir}}} = F_{X_{h\uparrow}} = F_{X_{h\downarrow}}$. Figure 5.10 shows the plots of these CDFs as x ranges between 0 and 1. The value of each function for a given x represents the probability that the feature value will be less than or equal to x. Note that $F_{X_{\text{abs}}}(0) = 0$, whereas $F_{X_{\text{dir}}}(0) = 0.5$. This is because for half of the possible values of h_1 and h_2, the directional features are zero.

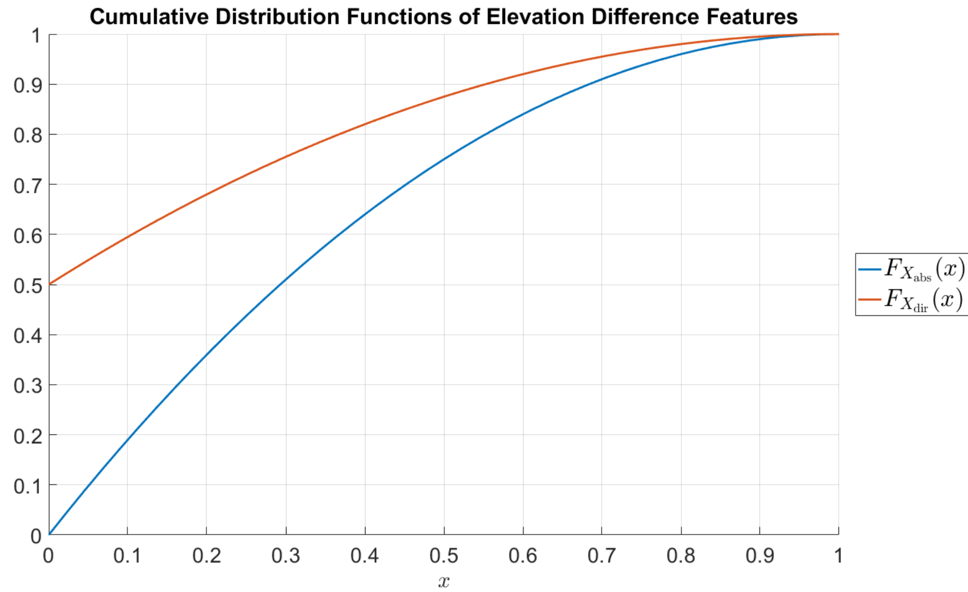


Figure 5.10 Plots of the cumulative distribution functions of the elevation difference features.

Returning to Equation 5.39, let Y_n^{abs} be the maximum of n values sampled from the distribution f_{X_h}, and let Y_n^{dir} be the maximum of n values sampled from either $f_{X_{h\uparrow}}$ or $f_{X_{h\downarrow}}$. We can express the CDF of Y_n^{abs} as

$$F_{Y_n^{\text{abs}}}(y) = \left[F_{X_{\text{abs}}}(y)\right]^n \tag{5.48a}$$

$$= (2y - y^2)^n, \;\; 0 \le y \le 1 \tag{5.48b}$$

and the CDF of Y_n^{dir} as

$$F_{Y_n^{\text{dir}}}(y) = \left[F_{X_{\text{dir}}}(y)\right]^n \tag{5.49a}$$

$$= \left(-\frac{y^2}{2} + y + \frac{1}{2}\right)^n, \;\; 0 \le y \le 1. \tag{5.49b}$$

Plots of these CDFs are shown in Figure 5.11. Note that $F_{Y_n^{\text{abs}}}(0) = 0$ and $F_{Y_n^{\text{dir}}}(0) = 2^{-n}$. The functions shift toward higher values of y as n increases, indicating that the expected maximum value should increase with more samples.

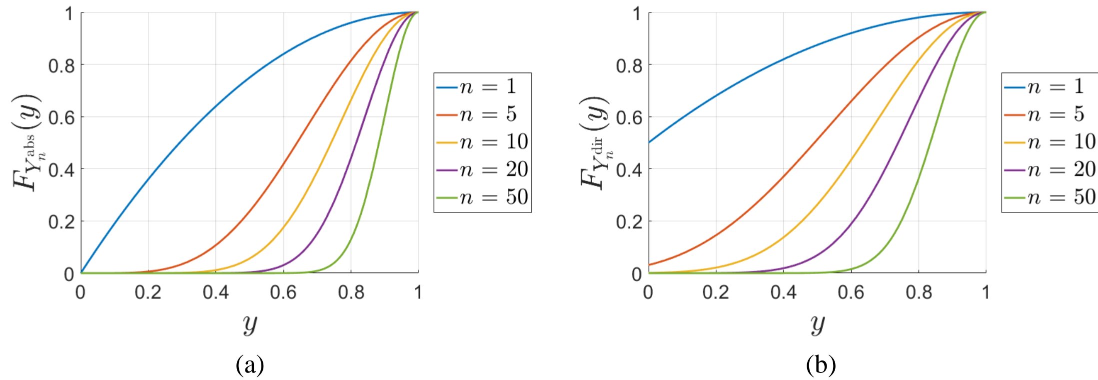


Figure 5.11 CDFs of the maximum of n elevation difference feature values. (a) CDF of Y_n^{abs} for various values of n. (b) CDF of Y_n^{dir} for various values of n.

To get the PDFs of these functions, we differentiate using Equation 5.36. For the absolute elevation difference, this gives

$$f_{Y_n^{\text{abs}}}(y) = \frac{d}{dy} F_{Y_n^{\text{abs}}}(y) \tag{5.50a}$$

$$= -n(2y - 2)(2y - y^2)^{n-1}, \;\; 0 \le y \le 1. \tag{5.50b}$$

For the directional elevation difference, this gives

$$f_{Y_n^{\text{dir}}}(y) = \frac{d}{dy} F_{Y_n^{\text{dir}}}(y) \tag{5.51a}$$

$$= -n(y-1)\left(-\frac{y^2}{2} + y + \frac{1}{2}\right)^{n-1} + \frac{\delta(y)}{2^n}, \;\; 0 \le y \le 1 \tag{5.51b}$$

where δ is the Dirac delta function that models the probability point mass at $y = 0$. This represents the case where all sampled values are zero, and it is included to ensure that $\int_0^1 f_{Y_n^{\text{dir}}}(y)\, dy = 1$. Figure 5.12 shows these PDFs for several values of n. Notice that the PDFs of the directional elevation features are skewed towards slightly lower values than those of the absolute elevation difference. This is in addition to the probability mass from the Dirac delta function at $y = 0$, which is not shown.

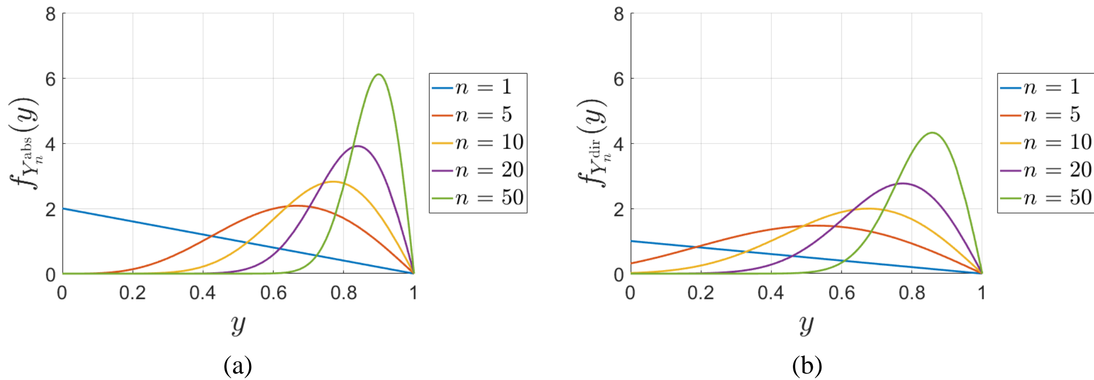


Figure 5.12 PDFs of the maximum of n elevation difference feature values. (a) PDF of Y_n^{abs} for various values of n. (b) PDF of Y_n^{dir} for various values of n. Note that $\delta(y)$ is not shown in these plots.

We use Equation 5.35 to get the expected values of Y_n^{abs} and Y_n^{dir} from their PDFs. For the expected absolute elevation difference, we obtain

$$\mathbb{E}\left[Y_n^{\text{abs}}\right] = \int_0^1 y f_{Y_n^{\text{abs}}}(y)\, dy \tag{5.52a}$$

$$= \int_0^1 -ny(2y-2)(2y-y^2)^{n-1}\, dy. \tag{5.52b}$$

For the uphill and downhill elevation differences, we obtain

$$\mathbb{E}\left[Y_n^{\text{dir}}\right] = \int_0^1 y f_{Y_n^{\text{dir}}}(y)\, dy \tag{5.53a}$$

$$= \int_0^1 -ny(y-1)\left(-\frac{y^2}{2} + y + \frac{1}{2}\right)^{n-1} + \frac{y\delta(y)}{2^n}\, dy \tag{5.53b}$$

$$= \int_0^1 -ny(y-1)\left(-\frac{y^2}{2} + y + \frac{1}{2}\right)^{n-1} dy. \tag{5.53c}$$

Note that we can ignore the Dirac delta function here, since $y\delta(y) = 0$. Evaluating these integrals for large values of n can become costly for real-time operation, so we precompute the expected values up to some limit $(n = 100)$ and save these in a lookup table. Table 5.1 shows the expected values of Y_n^{abs} and Y_n^{dir} for several values of n.

Table 5.1 Expected values of Y_n^{abs} and Y_n^{dir} for various values of n

n	1	2	3	4	5	10	20	50	100
$\mathbb{E}\left[Y_n^{\text{abs}}\right]$	0.333	0.467	0.543	0.594	0.631	0.730	0.806	0.876	0.912
$\mathbb{E}\left[Y_n^{\text{dir}}\right]$	0.167	0.283	0.368	0.431	0.480	0.618	0.725	0.824	0.875

To verify that our method is correct, we randomly sampled n values of h_1 and h_2 from a uniform distribution $U(0, 1)$ and computed the maximum value of the absolute and directional elevation features. This was repeated 100 times for each value of n in $1, \dots, 100$. Figure 5.13 shows the box plots of the distributions of computed values for each n along with a curve representing the expected values $\mathbb{E}\left[Y_n^{\text{abs}}\right]$ and $\mathbb{E}\left[Y_n^{\text{dir}}\right]$ computed using Equations 5.52 and 5.53. The box plots show the maximum, minimum, median, and upper and lower quartiles of each distribution, along with any outliers. A green 'x' is plotted on each box plot at the location of the mean value. For both the absolute and directional elevation features, the red line representing the computed expected values is in alignment with the means of the sampled distributions.

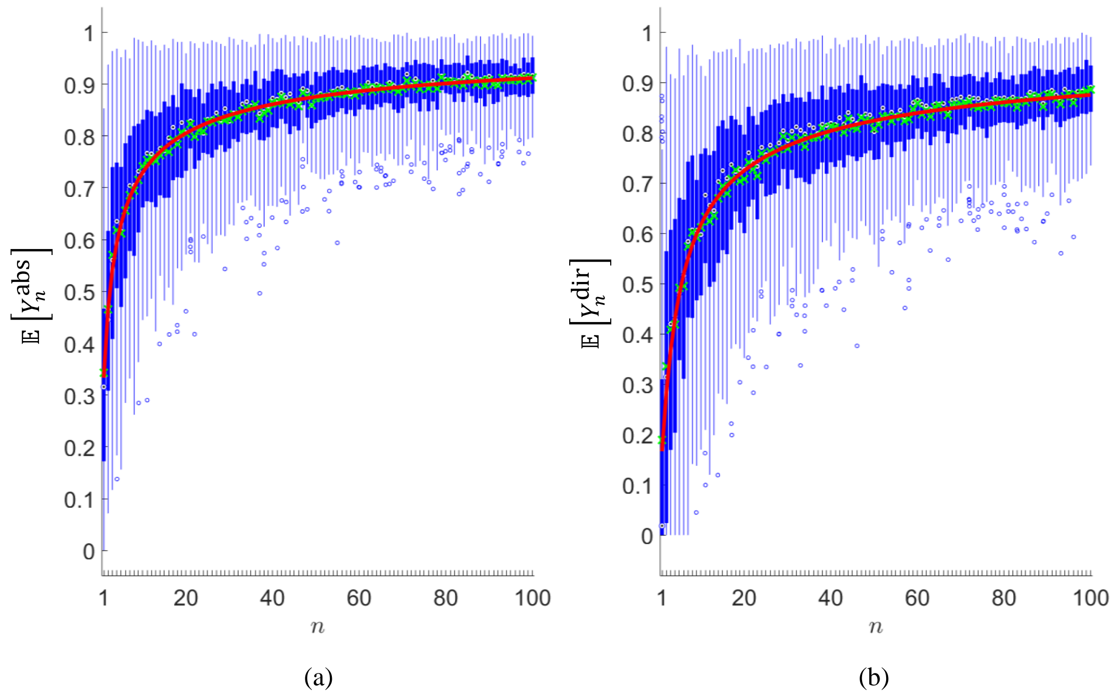


Figure 5.13 Expected values of (a) Y_n^{abs} and (b) Y_n^{dir} for n in $1, \ldots, 100$. For each n, 100 samples of Y_n were computed for the absolute and directional elevation features and the resulting distributions are shown in blue as box plots. The mean value of each sampled distribution is shown as a green 'x'. The red line indicates the expected value computed using Equations 5.52 and 5.53.

Algorithm 5.14 shows how the above computations are implemented as part of the ELEV_FEATURE function in Algorithm 5.12. Lines 6-15 of Algorithm 5.12 compute the region cost matrices U^1 and U^2, using the BELLMAN_FORD_GRID_DIST function in Algorithm 5.13 when the regions are observed and the UNOBSERVED_ELEVATION_COST function in Algorithm 5.14 when the regions are unobserved. Algorithm 5.14 provides a cost matrix U^r containing the expected elevation feature costs for a region using the distance cost matrix U^d and the above definitions. For the maximum aggregation method, the values are precomputed in a lookup table to avoid unnecessary computation.

Algorithm 5.14 Unobserved Elevation Costs

UNOBSERVED_ELEVATION_COST(U^d, *type*, *agg*)
1: (N, K) ← size of U^d
2: $U^r \leftarrow N \times K$ matrix initalized to ∞
3: **for** each $(i, k) \in \{1 \le i \le N \wedge 1 \le k \le K\}$
4: **if** *agg* = "*sum*"
5: **if** *type* = "*abs*"
6: $U^r[i, k] \leftarrow \frac{1}{3} \times U^d[i, k]$
7: **else if** *type* = "*up*" or *type* = "*down*"
8: $U^r[i, k] \leftarrow \frac{1}{6} \times U^d[i, k]$
9: **else if** *agg* = "*max*"
10: $n \leftarrow U^d[i, k]$
11: **if** *type* = "*abs*"
12: $U^r[i, k] \leftarrow \mathbb{E}\left[Y_n^{\text{abs}}\right]$
13: **else if** *type* = "*up*" or *type* = "*down*"
14: $U^r[i, k] \leftarrow \mathbb{E}\left[Y_n^{\text{dir}}\right]$
15: **return** U^r

5.3.4 Combining Region Elevation Costs

The second half of Algorithm 5.12 combines the region elevation costs from both regions to compute the actual elevation difference feature. Lines 16-20 of Algorithm 5.12 construct the U^{bnd} matrix from the provided boundary edge list E_{bnd}. Each row of U^{bnd} contains the min, mean, and max feature values of one of the boundary edges. Line 21 calls the COMBINE_ELEVATION_COSTS function, which is given in Algorithm 5.15. This function takes the following input arguments:

- expected aggregated elevation cost matrices U^1 and U^2, containing the average costs of traveling between each grid cell and each boundary edge,
- the boundary edge cost matrix U^{bnd},

- region distance cost matrices U^{d1} and U^{d2}, containing the number of steps required to travel between each grid cell and each boundary edge,
- the observability of the two regions o_1 and o_2, and
- an *agg* parameter set to either "*sum*" or "*max*" to indicate if summation or maximization aggregation should be used.

The algorithm starts on lines 1 and 2 by getting the number of cells in each region (N_1 and N_2) and the number of boundary edges (K). These are used to produce an index set I of starting and ending cells from the two regions (line 3) and to initialize the mean and max $N_1 \times N_2$ cost matrices C_{mean} and C_{max} (line 4). Note that we do not need a cost matrix for the minimum feature value because this will always be represented by a single-step transition along one of the boundary edges.

We first consider the computation of the mean cost matrix C_{mean} for both aggregation types. Recall from Equations 5.1 and 5.2 that the minimum cost of a path from cell i in R_1 to cell j in R_2 using boundary edge k is given as $u_{ik}^1 + u_k^{\text{bnd}} + u_{jk}^2$ for summation aggregation and as $\max\left(u_{ik}^1, u_k^{\text{bnd}}, u_{jk}^2\right)$ for maximization. The expected values of u_{ik}^1 and u_{jk}^2 are provided by the U^1 and U^2 matrices respectively, and the mean cost of the boundary edge u_k^{bnd} is given by the second column of U^{bnd}. From Equation 5.3, the minimum expected cost is defined using the boundary edge that gives the minimum value. Since U^1 and U^2 have already been defined to contain the expected region elevation costs for the appropriate aggregation type regardless of observability, we can apply the above expressions to each pair of cells in the index I to get C_{mean} on line 6 for the max type and on line 13 for the sum type.

To compute the maximum cost matrix $C_{\max}$, we need to update the definitions of U^1 and U^2 depending on if each region has been observed. If a region has been observed, then the corresponding cost matrix contains the actual observed costs and does not need to be changed. However, if a region is unobserved ($o_1 = 0$ or $o_2 = 0$), then we need to update the cost matrix to contain the maximum cost that the agent could encounter along the path between each cell and each boundary edge.

First, consider an unobserved region with the maximum aggregation type. If the region distance cost matrix U^{d1} or U^{d2} indicates that a grid cell is one or more steps away from the boundary edge, then there is at least one completely unobserved edge that could take the maximum value of 1. Lines 8 and 10 apply this test to each cost matrix element in an unobserved region, setting the value of U^1 or U^2 to 1 if the corresponding value in U^{d1} or U^{d2} is greater than zero and setting it to 0 otherwise. The only reason the expected cost is not simply set to 1 for any unobserved region is to handle the edge case where one region is unobserved and contains only a single grid cell. In this case, the maximum cost would be determined by the maximum region cost of the other region and the boundary edge, since there would be no edges completely within the unobserved region.

Next, consider an unobserved region with the summation aggregation type. For summation, each unobserved step could mean the addition of the largest possible elevation difference feature value. In the worst case, an elevation pattern of (0, 1, 0, 1, 0, ...) would produce an absolute elevation difference of 1 for each grid step. Therefore, when the feature type is set to "*abs*" and one of the regions is unobserved with summation aggregation, the corresponding cost matrix U^1 or U^2 is set to be the same as the corresponding region

distance cost matrix U^{d1} or U^{d2} (lines 16 and 21). The situation is only half as bad for the directional elevation costs. Even in the worst case, the uphill or downhill feature value can only take its maximum every other step. Each element of the corresponding cost matrix is set to $\left\lceil\frac{d}{2}\right\rceil$ where d is the value of the distance cost matrix U^{d1} or U^{d2} (lines 18 and 23).

After updating the cost matrices U^1 and U^2 to account for unobserved regions, we compute the maximum cost matrix $C_{\max}$, using Equations 5.1 or 5.2 and Equation 5.3 as before, but using the third column of U^{bnd}, which contains the maximum feature value of each boundary edge (lines 11 and 24). To get the final feature as a triangular fuzzy number, we first define the minimum feature value $f_{\min}$ as the minimum of the first column of U^{bnd}, which contains the minimum cost of each boundary edge (line 25). The mean feature value f_{mean} is the average of all values in the mean cost matrix C_{mean} (line 26). The maximum feature value $f_{\max}$ is the maximum of all values in $C_{\max}$ (line 27). The final feature is constructed as the triangular fuzzy number $\text{Tri}(f_{\min}, f_{\text{mean}}, f_{\max})$ on line 28 and returned to the ELEV_FEATURE function in Algorithm 5.12 on line 29. This is eventually returned to the original COMPUTE_REGION_FEATURES function in Algorithm 5.10.

Algorithm 5.15 Combine Elevation Costs

COMBINE_ELEVATION_COSTS(U^1, U^2, U^{bnd}, U^{d1}, U^{d2}, o_1, o_2, *type*, *agg*)
1: $(N_1, K) \leftarrow$ size of U^1
2: $(N_2, K) \leftarrow$ size of U^2
3: $I \leftarrow \{(i, j) \mid 1 \le i \le N_1 \wedge 1 \le j \le N_2\}$
4: $C_{\text{mean}}, C_{\text{max}} \leftarrow N_1 \times N_2$ matrices initalized to ∞

/* *Combine costs* */
5: **if** agg = "*max*"
6: $C_{\text{mean}}[i, j] \leftarrow \min_k\{\max(U^1[i, k], U^{\text{bnd}}[k, 2], U^2[j, k])\}\ \forall\ (i, j) \in I$
7: **if** $o_1 = 0$
8: $U^1[i, k] \leftarrow [U^{d1}[i, k] > 0]\ \forall\ (i, k) \in \{(i, k) \mid 1 \le i \le N_1 \wedge 1 \le k \le K\}$
9: **if** $o_2 = 0$
10: $U^2[j, k] \leftarrow [U^{d2}[j, k] > 0]\ \forall\ (j, k) \in \{(j, k) \mid 1 \le j \le N_2 \wedge 1 \le k \le K\}$
11: $C_{\text{max}}[i, j] \leftarrow \min_k\{\max(U^1[i, k], U^{\text{bnd}}[k, 3], U^2[j, k])\}\ \forall\ (i, j) \in I$
12: **else if** agg = "*sum*"
13: $C_{\text{mean}}[i, j] \leftarrow \min_k\{U^1[i, k] + U^{\text{bnd}}[k, 2] + U^2[j, k]\}\ \forall\ (i, j) \in I$
14: **if** $o_1 = 0$
15: **if** *type* = "*abs*"
16: $U^1 \leftarrow U^{d1}$
17: **else if** *type* = "*up*" or *type* = "*down*"
18: $U^1[i, k] \leftarrow \lceil U^{d1}[i, k] / 2 \rceil\ \forall\ (i, k) \in \{(i, k) \mid 1 \le i \le N_1 \wedge 1 \le k \le K\}$
19: **if** $o_2 = 0$
20: **if** *type* = "*abs*"
21: $U^2 \leftarrow U^{d2}$
22: **else if** *type* = "*up*" or *type* = "*down*"
23: $U^2[j, k] \leftarrow \lceil U^{d2}[j, k] / 2 \rceil\ \forall\ (j, k) \in \{(j, k) \mid 1 \le j \le N_2 \wedge 1 \le k \le K\}$
24: $C_{\text{max}}[i, j] \leftarrow \min_k\{U^1[i, k] + U^{\text{bnd}}[k, 3] + U^2[j, k]\}\ \forall\ (i, j) \in I$

/* *Construct feature* */
25: $f_{\text{min}} \leftarrow \min_k\{U^{\text{bnd}}[k, 1]\}$
26: $f_{\text{mean}} \leftarrow \frac{1}{N_1 N_2} \sum_{i,j} C_{\text{mean}}[i, j]$
27: $f_{\text{max}} \leftarrow \max_{i,j}\{C_{\text{max}}[i, j]\}$
28: $F \leftarrow \text{Tri}(f_{\text{min}}, f_{\text{mean}}, f_{\text{max}})$

29: **return** F

To demonstrate the computation of the elevation features for the example in Figure 5.3, consider the maximum absolute elevation difference feature when region 1 is observed but region 2 is unobserved. The cost matrix U^1 is implicitly given from the distance grid values from region 1 in the top row of Figure 5.7,

$$U^1 = \begin{bmatrix} 0.3 & 0.3 & 0.3 \\ 0.3 & 0.3 & 0.3 \\ 0.5 & 0.5 & 0.5 \\ 0.1 & 0.2 & 0.2 \\ 0.1 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 \\ 0 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 \\ 0 & 0.2 & 0.2 \\ 0.2 & 0 & 0 \end{bmatrix}.$$

The boundary edge matrix U^{bnd} comes from the boundary edge list E_{bnd} computed in Algorithm 5.11. Using Equation 4.71, we compute

$$U^{\text{bnd}} = \begin{bmatrix} 0 & 0.25 & 0.5 \\ 0 & 0.5 & 1 \\ 0 & 0.5 & 1 \end{bmatrix}.$$

The region distance cost matrices U^{d1} and U^{d2} are the same as those in Figure 5.5. Because region 2 is unobserved, U^2 is defined by Algorithm 5.14 using U^{d2} and the "max-abs" configuration. The values are given by $\mathbb{E}\left[Y_n^{\text{abs}}\right]$, where n comes from U^{d2}. Using the precomputed values shown partially in Table 5.1, the U^2 matrix is computed as

$$U^2 = \begin{bmatrix} 0.594 & 0.467 & 0 \\ 0 & 0.467 & 0.594 \\ 0.333 & 0.333 & 0.543 \\ 0.467 & 0 & 0.467 \\ 0.543 & 0.333 & 0.333 \\ 0.467 & 0.467 & 0.594 \\ 0.543 & 0.333 & 0.543 \\ 0.594 & 0.467 & 0.467 \\ 0.594 & 0.594 & 0.659 \\ 0.543 & 0.543 & 0.631 \\ 0.594 & 0.467 & 0.594 \\ 0.594 & 0.594 & 0.659 \end{bmatrix}.$$

The mean cost matrix C_{mean} is computed on line 6 of Algorithm 5.15 as

$$C_{\text{mean}} = \begin{bmatrix} 0.5 & 0.3 & 0.33 & 0.47 & 0.5 & 0.47 & 0.5 & 0.5 & 0.59 & 0.54 & 0.5 & 0.59 \\ 0.5 & 0.3 & 0.33 & 0.47 & 0.5 & 0.47 & 0.5 & 0.5 & 0.59 & 0.54 & 0.5 & 0.59 \\ 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.59 & 0.54 & 0.5 & 0.59 \\ 0.5 & 0.25 & 0.33 & 0.47 & 0.5 & 0.47 & 0.5 & 0.5 & 0.59 & 0.54 & 0.5 & 0.59 \\ 0.5 & 0.25 & 0.33 & 0.47 & 0.5 & 0.47 & 0.5 & 0.5 & 0.59 & 0.54 & 0.5 & 0.59 \\ 0.5 & 0.25 & 0.33 & 0.47 & 0.5 & 0.47 & 0.5 & 0.5 & 0.59 & 0.54 & 0.5 & 0.59 \\ 0.5 & 0.25 & 0.33 & 0.47 & 0.5 & 0.47 & 0.5 & 0.5 & 0.59 & 0.54 & 0.5 & 0.59 \\ 0.5 & 0.25 & 0.33 & 0.47 & 0.5 & 0.47 & 0.5 & 0.5 & 0.59 & 0.54 & 0.5 & 0.59 \\ 0.5 & 0.25 & 0.33 & 0.47 & 0.5 & 0.47 & 0.5 & 0.5 & 0.59 & 0.54 & 0.5 & 0.59 \\ 0.5 & 0.25 & 0.33 & 0.47 & 0.5 & 0.47 & 0.5 & 0.5 & 0.59 & 0.54 & 0.5 & 0.59 \\ 0.5 & 0.25 & 0.33 & 0.47 & 0.5 & 0.47 & 0.5 & 0.5 & 0.59 & 0.54 & 0.5 & 0.59 \end{bmatrix}.$$

The max cost matrix $C_{\max}$ is computed on line 11 of Algorithm 5.15 after replacing the U^2 matrix with 1 for any value where $U^{d2} > 0$,

$$C_{\max} = \begin{bmatrix} 1 & 0.5 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0.5 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0.5 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0.5 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0.5 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0.5 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0.5 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0.5 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0.5 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0.5 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0.5 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

From the boundary cost matrix, we compute $f_{\min} = 0$. The mean value of C_{mean} is computed as $f_{\text{mean}} = 0.483$, and the max value of $C_{\max}$ is $f_{\max} = 1$. The fuzzy number cost for the maximum absolute elevation difference feature for this example is therefore computed as $\tilde{f}_{h_\max} = \text{Tri}(0, 0.483, 1)$. Other configurations can be computed in a similar way using the above algorithms and definitions.

5.4 Approximate Fuzzy Region Features

The region feature definitions presented in Sections 5.2 and 5.3 are sometimes too computationally intensive for use in real-time applications or when performing many Monte Carlo simulations. This can also be true in large environments or where regions share long borders, resulting in many region boundary edges. In these situations, it may be acceptable to approximate the region features using simpler approaches. Any problem with partial observability or region partitioning will be subject to some approximation in the feature definitions since there is uncertainty represented in the triangular fuzzy numbers. The quality of any feature approximation depends on the type of environment and region clustering parameters. The agent designer must decide what is an acceptable tradeoff for any given problem.

By far, the costliest operations in the computation of region features are the grid distance searches used to define the region distance and elevation cost matrices. The GRID_DISTANCE function is called twice for each boundary edge regardless of observability, and for each elevation feature type that needs to be computed, the BELLMAN_FORD_GRID_DIST function is called once for each boundary edge in an observed region. The result of these functions is a region cost matrix that specifies the

feature cost between each grid cell in a region and one of the boundary edges. One way to significantly reduce the computation time is to eliminate the distance searches for each boundary edge and instead use a single approximation of the distance from each grid cell to the region boundary. This approach is similar to the method used to compute the terrain-based region features using the V^{d1} and V^{d2} matrices defined by Equations 5.8 and 5.9.

The distance to the region boundary can be approximated quickly by using the region centroids. The centroid of each region is computed using Algorithm 3.9 during the creation of the region graph in Algorithm 5.5. Consider two regions R_1 and R_2 with centroids $\left(c_x^1, c_y^1\right)$ and $\left(c_x^2, c_y^2\right)$. For every cell $(x, y) \in R_1$, the distance to the centroid of R_2 can be approximated as

$$D_c^1(x,y) = |x - c_x^2| + \left|y - c_y^2\right|. \tag{5.54}$$

Likewise, for every cell $(x, y) \in R_2$, the distance to the centroid of R_1 is

$$D_c^2(x,y) = |x - c_x^1| + \left|y - c_y^1\right|. \tag{5.55}$$

This is simply the Manhattan distance from the centroids of each region. The minimum distance required to reach the region boundary from a cell $(x, y) \in R_1$ can be approximated as

$$V_{xy}^{d1} = D_c^1(x,y) - \min_{(u,v)\in R_1} D_c^1(u,v). \tag{5.56}$$

Likewise, for a cell $(x, y) \in R_2$,

$$V_{xy}^{d2} = D_c^2(x,y) - \min_{(u,v)\in R_2} D_c^2(u,v). \tag{5.57}$$

These matrices can be reshaped into single column vectors that include only the distances for the grid cells in each region. The distance cost matrix is then defined to approximate

the distance between all pairs of cells in the two regions. For a cell $i = (x_1, y_1) \in R_1$ and a cell $j = (x_2, y_2) \in R_2$, the distance cost is defined as

$$C_{ij}^{d} = V_{x_1 y_1}^{d1} + V_{x_2 y_2}^{d2} + 1, \tag{5.58}$$

where 1 is added to account for the cost of crossing the boundary edge. If there are no walls or obstacles to navigate around, then this is an accurate measure of the distance to a single point on the region boundary. It becomes less accurate when the regions have irregular shapes or when the shortest path between the regions is not a straight line. The region distance feature and terrain-based features can be computed using these substitutions for the C^d, V^{d1}, and V^{d2} matrices.

Figure 5.14 shows this approximation approach applied to the example in Figure 5.3. The distances to each region centroid are shown in the top of each cell and the estimated distances to the region boundary are shown in the bottom of each cell. Figure 5.15 shows the resulting cost matrices V^{d1}, V^{d2}, and C^d for this example. Using the approximated C^d matrix with Equations 5.4-5.7, we compute an approximate fuzzy region distance feature value of $\tilde{f}_d\left(e_{R_1R_2}\right) = \mathrm{Tri}(1, 5.38, 8)$. To compare, the original definition was $\tilde{f}_d\left(e_{R_1R_2}\right) = \mathrm{Tri}(1, 5.03, 9)$. A comparison for all features is shown at the end of this section in Table 5.2 and Table 5.3.

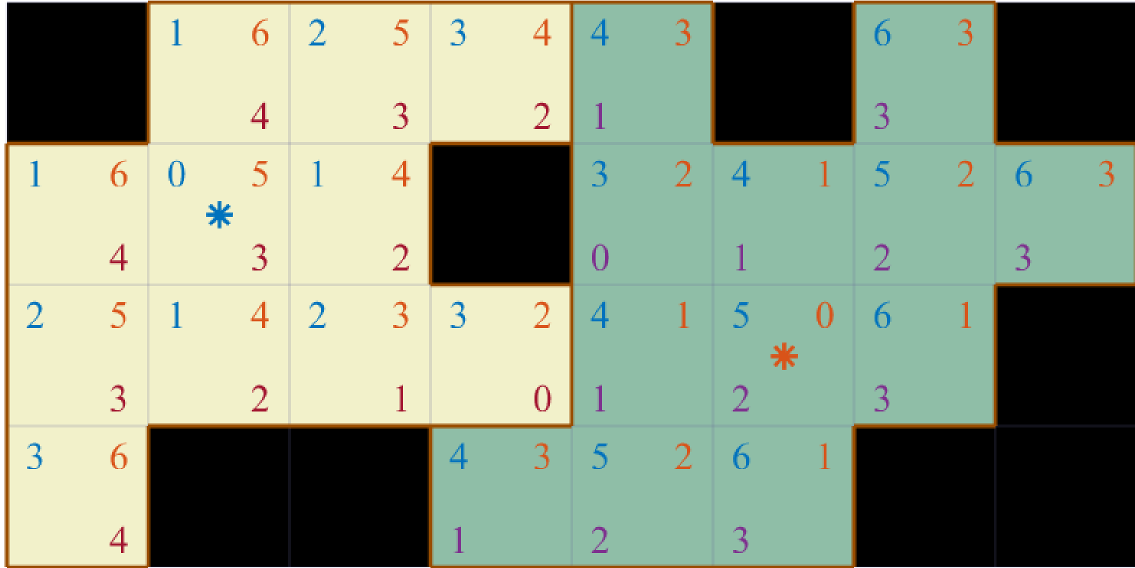


Figure 5.14 Approximation of the region distances using the region centroids for the example in Figure 5.3. The region centroids are marked with asterisks. The D_c^1 values (distance from the right centroid) are shown in the top right of each cell. The D_c^2 values (distance from the left centroid) are shown in the top left of each cell. The values in the bottom of each cell indicate the V^{d1} and V^{d2} values, which are the approximated distances to the region boundary.

$$V^{d1} = \begin{bmatrix} 4 \\ 3 \\ 4 \\ 4 \\ 3 \\ 2 \\ 3 \\ 2 \\ 1 \\ 2 \\ 0 \end{bmatrix} \quad V^{d2} = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 2 \\ 1 \\ 2 \\ 3 \\ 3 \\ 2 \\ 3 \\ 3 \end{bmatrix} \quad C^d = \begin{bmatrix} 6 & 6 & 5 & 6 & 7 & 6 & 7 & 8 & 8 & 7 & 8 & 8 \\ 5 & 5 & 4 & 5 & 6 & 5 & 6 & 7 & 7 & 6 & 7 & 7 \\ 6 & 6 & 5 & 6 & 7 & 6 & 7 & 8 & 8 & 7 & 8 & 8 \\ 6 & 6 & 5 & 6 & 7 & 6 & 7 & 8 & 8 & 7 & 8 & 8 \\ 5 & 5 & 4 & 5 & 6 & 5 & 6 & 7 & 7 & 6 & 7 & 7 \\ 4 & 4 & 3 & 4 & 5 & 4 & 5 & 6 & 6 & 5 & 6 & 6 \\ 5 & 5 & 4 & 5 & 6 & 5 & 6 & 7 & 7 & 6 & 7 & 7 \\ 4 & 4 & 3 & 4 & 5 & 4 & 5 & 6 & 6 & 5 & 6 & 6 \\ 3 & 3 & 2 & 3 & 4 & 3 & 4 & 5 & 5 & 4 & 5 & 5 \\ 4 & 4 & 3 & 4 & 5 & 4 & 5 & 6 & 6 & 5 & 6 & 6 \\ 2 & 2 & 1 & 2 & 3 & 2 & 3 & 4 & 4 & 3 & 4 & 4 \end{bmatrix}$$

Figure 5.15 Approximation of the region distance cost matrices for the example in Figure 5.14.

To get the approximations of the elevation difference features, we can use the COMBINE_ELEVATION_COSTS function in Algorithm 5.15 with some substitutions for the distance matrices. For the U^{d1} and U^{d2} matrices, we use V^{d1} and V^{d2} as computed above

using the region centroids. This implies only one boundary edge, so U^{bnd} is computed as a 1×3 matrix of the minimum, mean, and maximum feature values over all boundary edges between the two regions. The expeced aggregated elevation feature cost matrices U^1 and U^2 are approximated using the V^{d1} and V^{d2} matrices and any observed elevation values. If a region is unobserved, the UNOBSERVED_ELEVATION_COST function from Algorithm 5.14 is used with the appropriate substitution of V^{d1} or V^{d2} for the U^d matrix. For observed regions, the most accurate cost measure requires a distance search with the BELLMAN_FORD_GRID_DIST function from Algorithm 5.13, either from one (good) or all (better) boundary edges. If just one boundary edge is used, it is prefereable to choose one near the center of the region boundary. If all boundary edges are used, then the original elevation feature definition from Algorithm 5.12 should be used. To avoid any iterative distance search when computational resources are extremely limited, the following procedure can be used to approximate the U^1 and U^2 matrices.

First, let V_i^d be the approximated distance to the region boundary of grid cell i. Then, let E be the set of all edges in a region where each edge $e_{ij} \in E$ represents the transition from grid cell i to j, and let $f(e_{ij})$ be the elevation feature cost of that edge. Next, split the edge features into sets where set $S_k = \{f(e_{ij}) | V_i^d < V_j^d = k\}$. Let U_i be the entry in the expeced aggregated elevation feature cost matrix U^1 or U^2 for grid cell i. When using summation aggregation, define

$$U_i = \sum_{k=1}^{V_i^d} \left\{ \frac{1}{|S_k|} \sum_{f(e) \in S_k} f(e) \right\}, \tag{5.59}$$

and when using maximization aggregation, define

$$U_i = \max_{k=1,\dots,V_i^d} \left\{ \max_{f(e) \in S_k} f(e) \right\}. \tag{5.60}$$

This assigns the same cost value to each grid cell using the feature values of each edge set up to the given distance value. For summation, the cost is the sum of the average feature values in each edge set, and the cost is the overall maximum feature value in each edge set for maximization. After defining U^1 and U^2, the elevation features can be computed using the COMBINE_ELEVATION_COSTS function in Algorithm 5.15.

Figure 5.16 shows the edge sets used to approximate the elevation difference features for the example in Figure 5.3. Using these feature sets, we compute the expeced aggregated elevation feature cost matrices U^1 and U^2 for each feature type that needs to be computed. For example, to compute the $\tilde{f}_{h_\text{sum}}$ feature using the absolute elevation difference and summation aggregation, we would assign the following values to each element $U^1_{(k)}$, where $V_i^d = k$:

- $U^1_{(0)} = 0$
- $U^1_{(1)} = 0.2$
- $U^1_{(2)} = 0.2 + \frac{1}{2}(0.1 + 0.3) = 0.4$
- $U^1_{(3)} = 0.4 + \frac{1}{5}(0.3 + 0.2 + 0.4 + 0.2 + 0) = 0.62$
- $U^1_{(4)} = 0.62 + \frac{1}{5}(0.2 + 0.5 + 0.3 + 0.1 + 0.1) = 0.86$

For elements $U^2_{(k)}$ in region 2, we compute

- $U^2_{(0)} = 0$
- $U^2_{(1)} = \frac{1}{3}(0.1 + 0.5 + 0.1) = 0.233$
- $U^2_{(2)} = 0.23 + \frac{1}{5}(0.2 + 0.6 + 0.2 + 0.2 + 0.1) = 0.493$
- $U^2_{(3)} = 0.493 + \frac{1}{6}(0.3 + 0.2 + 0.4 + 0.2 + 0) = 0.627$

The resulting U^1 and U^2 matrices are used in Algorithm 5.15 to compute an overall feature value of $\tilde{f}_{h_\text{sum}} = \text{Tri}(0.1, 1.46, 2.33)$. To compare, the original definition was $\tilde{f}_{h_\text{sum}} = \text{Tri}(0.1, 0.87, 2.1)$. Table 5.2 and Table 5.3 at the end of this section compare all the feature values using the original and approximate definitions for observed and unobserved cases. In the unobserved cases, we define the terrain type priors as $p(t_1) = 0.75$ and $p(t_2) = 0.25$.

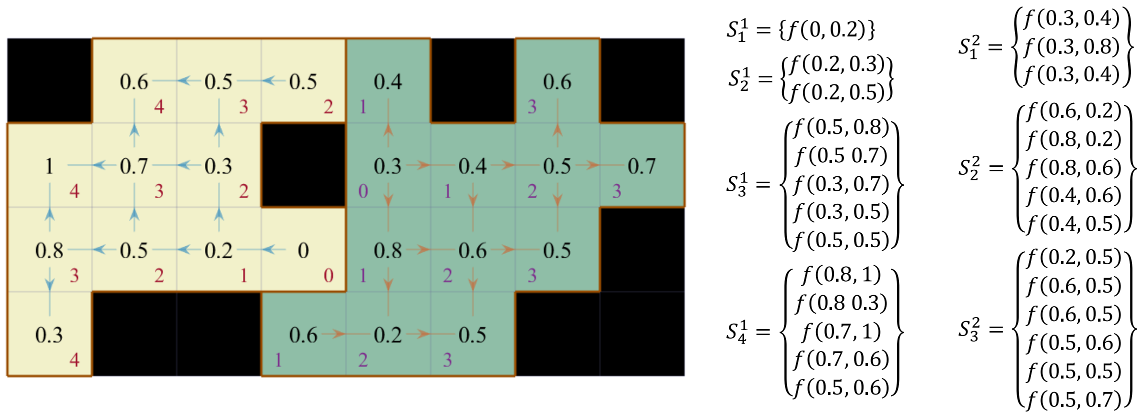


Figure 5.16 Elevation feature edge sets used to approximate the elevation difference features for the example in Figure 5.14. The numbers in the bottom corners of each cell show the approximate distances to the region boundary. Only edges that increase in distance are used. The elevation values are shown for each cell, and the feature sets are split based on the farthest distance of each edge. The sets S_k^1 and S_k^2 show the feature values from regions 1 and 2 respectively with max distance k. The actual feature values depend on which feature is being computed using one of the Equations 4.11-4.13.

All approximations result in some loss of accuracy, but these methods can be used in many cases to reduce computation time. This can lead to significant time savings that may allow for more analysis and planning to occur between updates. These approximations work best when the regions are generally convex and have relatively smooth elevation changes. When there are many sudden, unexpected elevation changes or when the region boundary does not lie between the region centroids, the approximations become less accurate.

Table 5.2 Original and approximate region features with both regions either observed or unobserved. For unobserved cases, the terrain type priors are $p(t_1) = 0.75$ and $p(t_2) = 0.25$.

	Both Regions Observed						Both Regions Unobserved					
	Original			Approximate			Original			Approximate		
	f_{min}	f_{mean}	f_{max}	f_{min}	f_{mean}	f_{max}	f_{min}	f_{mean}	f_{max}	f_{min}	f_{mean}	f_{max}
$\tilde{f}_d$	1	5.03	9	1	5.38	8	1	5.03	9	1	5.38	8
$\tilde{f}_{t(1)}$	0.5	2.5	4.5	0.5	3.05	4.5	0	3.70	9	0	4.03	8
$\tilde{f}_{t(2)}$	0.5	2.17	4.5	0.5	2.33	3.5	0	1.12	9	0	1.34	8
$\tilde{f}_{t\{1,1\}}$	0	2	4	0	2.55	4	0	3.52	9	0	3.85	8
$\tilde{f}_{t\{1,2\}}$	1	1	1	1	1	1	0	0.38	1	0	0.38	1
$\tilde{f}_{t\{2,2\}}$	0	1.67	4	0	1.83	3	0	1.00	9	0	1.16	8
$\tilde{f}_{t\langle 1,1\rangle}$	0	2	4	0	2.55	4	0	3.52	9	0	3.85	8
$\tilde{f}_{t\langle 1,2\rangle}$	1	1	1	1	1	1	0	0.19	1	0	0.19	1
$\tilde{f}_{t\langle 2,1\rangle}$	0	0	0	0	0	0	0	0.19	1	0	0.19	1
$\tilde{f}_{t\langle 2,2\rangle}$	0	1.67	4	0	1.83	3	0	1.00	9	0	1.16	8
$\tilde{f}_{h_\text{max}}$	0.1	0.24	0.5	0.1	0.56	0.8	0	0.52	1	0	0.52	1
$\tilde{f}_{h\uparrow_\text{max}}$	0	0.20	0.5	0	0.50	0.8	0	0.35	1	0	0.35	1
$\tilde{f}_{h\downarrow_\text{max}}$	0	0.17	0.3	0	0.48	0.6	0	0.35	1	0	0.35	1
$\tilde{f}_{h_\text{sum}}$	0.1	0.87	2.1	0.1	1.46	2.33	0	1.68	9	0	1.79	8
$\tilde{f}_{h\uparrow_\text{sum}}$	0	0.44	1.2	0	0.78	1.31	0	0.84	5	0	0.90	5
$\tilde{f}_{h\downarrow_\text{sum}}$	0	0.43	1.1	0	0.68	1.11	0	0.84	5	0	0.90	5

Table 5.3 Original and approximate region features with only one region observed. For unobserved cases, the terrain type priors are $p(t_1) = 0.75$ and $p(t_2) = 0.25$.

	Region 1 Observed; Region 2 Unobserved						Region 1 Unobserved; Region 2 Observed					
	Original			Approximate			Original			Approximate		
	$f_{\min}$	f_{mean}	$f_{\max}$	$f_{\min}$	f_{mean}	$f_{\max}$	$f_{\min}$	f_{mean}	$f_{\max}$	$f_{\min}$	f_{mean}	$f_{\max}$
$\tilde{f}_d$	1	5.03	9	1	5.38	8	1	5.03	9	1	5.38	8
$\tilde{f}_{t(1)}$	0.5	4.40	9	0.5	4.80	8	0	1.88	4.5	0	2.28	4.5
$\tilde{f}_{t(2)}$	0	0.54	4.5	0	0.58	3.5	0.5	2.88	9	0.5	3.09	8
$\tilde{f}_{t\{1,1\}}$	0	4.27	9	0	4.67	8	0	1.5	4	0	1.91	4
$\tilde{f}_{t\{1,2\}}$	0	0.25	1	0	0.25	1	0	0.75	1	0	0.75	1
$\tilde{f}_{t\{2,2\}}$	0	0.42	4	0	0.46	3	0	2.51	9	0	2.72	8
$\tilde{f}_{t\langle 1,1\rangle}$	0	4.27	9	0	4.67	8	0	1.5	4	0	1.91	4
$\tilde{f}_{t\langle 1,2\rangle}$	0	0.25	1	0	0.25	1	0	0.75	1	0	0.75	1
$\tilde{f}_{t\langle 2,1\rangle}$	0	0	0	0	0	0	0	0	0	0	0	0
$\tilde{f}_{t\langle 2,2\rangle}$	0	0.42	4	0	0.46	3	0	2.51	9	0	2.72	8
$\tilde{f}_{h_\max}$	0	0.48	1	0	0.48	1	0	0.45	1	0	0.57	1
$\tilde{f}_{h\uparrow_\max}$	0	0.36	1	0	0.41	1	0	0.31	1	0	0.49	1
$\tilde{f}_{h\downarrow_\max}$	0	0.28	1	0	0.35	1	0	0.29	1	0	0.48	1
$\tilde{f}_{h_\text{sum}}$	0	1.38	5.7	0	1.56	4.86	0	1.28	5.4	0	1.57	5.47
$\tilde{f}_{h\uparrow_\text{sum}}$	0	0.70	3.2	0	0.71	3.12	0	0.65	3	0	0.90	3.19
$\tilde{f}_{h\downarrow_\text{sum}}$	0	0.69	3	0	0.85	3.24	0	0.64	2.8	0	0.67	2.87

5.5 Updating the Region Graph

Up to this point, the region graph G_R has been defined for static problems where the agent does not move. Section 5.1 gave the initial region boundaries and created the graph structure. Sections 5.2 and 5.3 defined the fuzzy feature values for each graph edge, and Section 5.4 provided a way to approximate these features quickly. These region boundaries and features are valid until the agent moves. When the agent does move, new parts of the environment may be discovered and the local region can change. These updates

need to be integrated into the existing region graph. Rather than recompute the entire region graph from scratch after each agent movement, we utilize the existing region graph and update only the parts of the graph that have changed, maintaining the existing graph structure and features where possible. This drastically improves the runtime efficiency of the algorithm by only changing portions of the region graph that have new information.

Consider an existing mental map structure $\mathcal{M}$ and a new observation $\mathcal{O}$ provided by the environment server. Upon receiving the observation, the agent will update its current position and the mental map grid layers using Algorithm 4.4. The updated mental map is then passed to the UPDATE_MENTAL_MAP_REGIONS function in Algorithm 5.16 to update the regions and the region graph. This function begins on line 1 by computing a new local region from the agent's updated position using Algorithm 5.2. This is the same method used to initialize the local region at the start of the simulation. However, since the agent has just moved to a new location, the updated local region may have changed.

An additional option given is by *opt.lrMemory* that dictates whether the new local region should replace or extend the existing local region in the mental map. If *opt.lrMemory* is true, then the local region will continue to grow as the agent explores the environment. This can be thought of as an agent that does not forget what it has learned about the feature values of the action graph and can help to reduce ocilatory behavior. Without this parameter setting, it becomes possible for the agent to wander back and forth between two grid cells when the unique region graphs computed from each location indicate that the best action is to move to the other cell. This will be explored further in Section 6.6.

Algorithm 5.16 Update Mental Map Regions

UPDATE_MENTAL_MAP_REGIONS($\mathcal{M}$, *opt*)

/* Get the new local region */
1: $LR \leftarrow$ GET_LOCAL_REGION($\mathcal{M}$, *opt*) // *Algorithm 5.2*
2: **if** *opt.lrMemory*
3: $LR[\mathcal{M}.localRegion = 1] \leftarrow 1$

/ * Determine the regions that need to be reclustered */
4: $Q \leftarrow$ GET_REGION_CLUSTERING_MASK($\mathcal{M}$, LR) // *Algorithm 5.17*

/* Create new region boundaries */
5: $L \leftarrow$ CLUSTER_MENTAL_MAP_REGIONS($\mathcal{M}$, LR, Q, *opt*) // *Algorithm 5.3*

/* Merge with existing regions */
6: $L \leftarrow$ MERGE_REGION_LABELS($\mathcal{M}.L$, L, Q, LR) // *Algorithm 5.18*

/* Update the region graph */
7: $G_R \leftarrow$ UPDATE_REGION_GRAPH($\mathcal{M}$, L) // *Algorithm 5.19*

/* Save the updated variables */
8: $\mathcal{M}.L \leftarrow L$
9: $\mathcal{M}.localRegion \leftarrow LR$
10: $\mathcal{M}.G_R \leftarrow G_R$

11: **return** $\mathcal{M}$

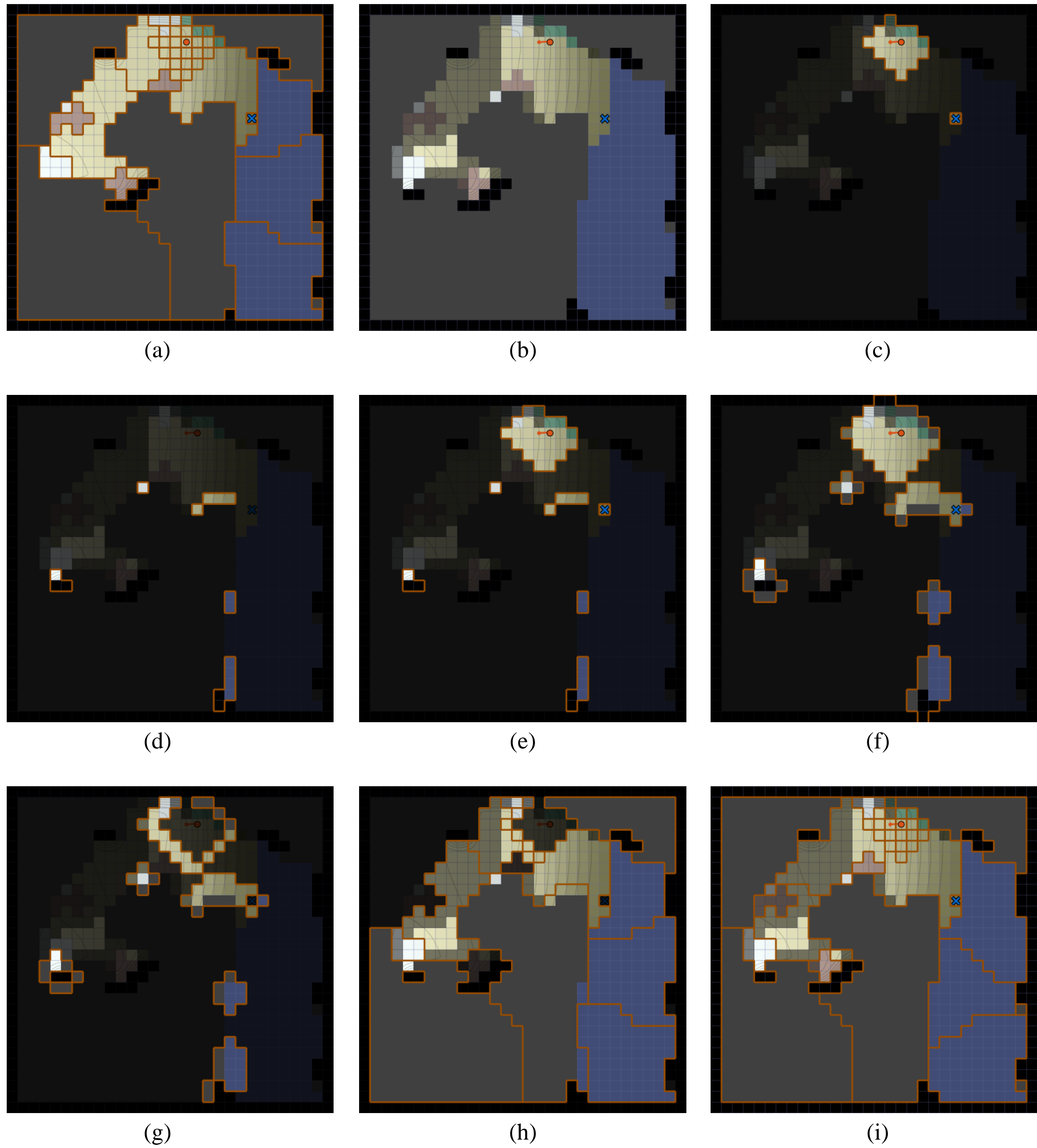


Figure 5.17 Step-by-step example of determining new regions. (a) Old regions before agent moves. (b) New observation. (c) New local region. (d) Cells with new information. (e) Update mask before dilation. (f) Update mask after dilation. (g) Removing walls and the new local region. (h) Old regions that need to be reclustered. (i) New regions after clustering.

Figure 5.17 shows a step-by-step example of updating the region boundaries after an agent moves. This is a continuation of the example in Figure 5.2. Subfigures (a) through (c) show the definition of the new local region. (a) shows the old regions that were computed at the initial agent location. (b) shows the new observation after the agent moves one cell to the right. Grid cells that are no longer visible are darkened, whereas cells that have never been observed are solid gray. (c) shows the new local region after the agent has moved one grid cell to the right. The parameter *opt.lrDist* is set to 3, but notice that the agent still cannot observe beyond one cell into the forest region directly to the north.

After determining the local region, the next step is to determine the set of cells that need to be reclustered. The GET_REGION_CLUSTERING_MASK function is called on line 4 of Algorithm 5.16 and is defined in Algorithm 5.17. The function starts on lines 1-3 by defining a mask U that identifies the cells that have just been observed. This is given by the set $\mathcal{M}.new$, which was defined as part of Algorithm 4.4 and is shown in subfigure (d) of Figure 5.17. These cells will need to be integrated into their adjacent regions, which may affect where the region boundaries are drawn. Additionally, since the local region may have changed, cells that have just been added to the local region will need to have their old regions redrawn and cells that left the local region will need to be included in neighboring regions. Line 4 of Algorithm 5.17 adds cells from the current and previous local region to the mask U, which is shown in subfigure (e).

To identify the cells from the neighboring regions, the mask is dilated on line 5, which is shown in subfigure (f). After dilation, line 6 removes from the mask any cells that are known walls or are part of the new local region. The remaining cells are shown in subfigure (g). These cells and their current regions will all need to be reclustered. Line 7

identifies K as the set of unique region labels in the updated mask. Finally, lines 8-10 construct the mask Q that will be used for clustering the regions. This mask consists of all grid cells that are currently assigned one of the labels in K and are not part of the new local region. Subfigure (h) shows the region clustering mask Q for the example with the old region boundaries marked. The final regions obtained after re-clustering are shown in subfigure (i).

Algorithm 5.17 Get the Region Clustering Mask

GET_REGION_CLUSTERING_MASK($\mathcal{M}$, LR)
1: $(n, m) \leftarrow \mathcal{M}.size$

/ Get a mask of the cells that need to be updated */*
2: $U \leftarrow n \times m$ grid initalized to 0
3: $U[\mathcal{M}.new] \leftarrow 1$
4: $U[\mathcal{M}.localRegion = 1 \vee LR = 1] \leftarrow 1$

/ Dilate to include neighboring cells */*
5: $U' \leftarrow U \oplus [0\ 1\ 0;\ 1\ 1\ 1;\ 0\ 1\ 0]$

/ Remove cells that are walls and cells that are part of the new local region */*
6: $U'[\mathcal{M}.W = 0 \vee LR = 1] \leftarrow 0$

/ Get the labels of the regions that need to be reclustered */*
7: $K \leftarrow \{k \mid k \in \mathcal{M}.L[U' = 1]\}$

/ Construct a mask of all cells in the identified regions */*
8: $Q \leftarrow n \times m$ grid initalized to 0
9: **for** each $(i, j) \in \{(i, j) \mid \mathcal{M}.L[i, j] \in K \wedge LR[i, j] = 0\}$
10: $Q[i, j] \leftarrow 1$

11: **return** Q

Returning to Algorithm 5.16, we have now identified the local region LR and the region clustering mask Q. The next step is to cluster the grid cells within the clustering mask on line 5 using the CLUSTER_MENTAL_MAP_REGIONS function in Algorithm 5.3. As opposed to the first time this function was called during the creation of the initial region boundaries, the clustering mask Q now restricts where the new region labels are assigned outside of the local region. Only cells within the new local region and the clustering mask Q are assigned region labels greater than zero. These labels need to be merged with the existing region labels in $\mathcal{M}.L$. This is accomplished by the MERGE_REGION_LABELS function on line 6, which is defined in Algorithm 5.18.

Algorithm 5.18 Merge Region Labels

MERGE_REGION_LABELS(U, L, Q, LR)

/ Remove old region labels that have been replaced */*
1: $S \leftarrow \{(i, j) \mid Q[i, j] = 1 \lor LR[i, j] = 1\}$
2: $U[S] \leftarrow 0$

/ Merge the old and new region labels */*
3: $L \leftarrow$ UPDATE_REGION_MAP(L, U) *// Algorithm 5.4*

/ Renumber regions */*
4: $K \leftarrow \{k \mid \exists(i, j)(L[i, j] = k \land k > 0)\}$
5: $L' \leftarrow L$
6: $t \leftarrow 1$
7: **for** each $k \in K$
8: $\quad I \leftarrow \{(i, j) \mid L[i, j] = k\}$
9: $\quad L'[I] \leftarrow t$
10: $\quad t \leftarrow t + 1$

11: **return** L'

The MERGE_REGION_LABELS function in Algorithm 5.18 takes a set of old region labels *U*, a set of new region labels *L*, a clustering mask *Q*, and the current local region *LR*. The first step in the algorithm is to eliminate old region labels that do not need to be included in the new label map. Line 1 identifies the cells that belong to either the clustering mask *Q* or the local region *LR* and line 2 sets these cells in *U* to zero. This allows us to use the UPDATE_REGION_MAP function from Algorithm 5.4 to combine the two sets of region labels (line 3). The combined region labels are likely not continuous, since some of the old regions were removed. Lines 4-10 renumber these regions so that each region is assigned a value between 1 and the total number of regions.

Once the new regions have been defined, the region graph itself can be updated. Line 7 of Algorithm 5.16 calls the UPDATE_REGION_GRAPH function in Algorithm 5.19. This function provides a high-level overview of the region graph update. First, the graph vertices are reassigned on line 1 using Algorithm 5.20. Then the graph edges are updated on line 2 using Algorithm 5.21. These functions use the existing region graph from the mental map structure $\mathcal{M}$ to avoid recomputing features that have not changed since the last update. Once the new graph has been defined, it is saved and returned on lines 3-7 as G_R.

Algorithm 5.19 Update Region Graph

UPDATE_REGION_GRAPH($\mathcal{M}$, L)

/ Create a lookup table between old and new regions and get vertices */*
1: B, $V \leftarrow$ UPDATE_REGION_GRAPH_VERTICES($\mathcal{M}$, L) *// Algorithm 5.20*

/ Add edges for adjacent regions */*
2: A, $E \leftarrow$ UPDATE_REGION_GRAPH_EDGES($\mathcal{M}$, L, B, V) *// Algorithm 5.21*

/ Save the graph structure */*
3: $G_R \leftarrow$ empty graph structure
4: $G_R.V \leftarrow V$
5: $G_R.A \leftarrow A$
6: $G_R.E \leftarrow E$

7: **return** G_R

The UPDATE_REGION_GRAPH_VERTICES function in Algorithm 5.20 uses the existing mental map structure $\mathcal{M}$ and the new region label map L to construct a list of vertices V and a lookup table B that maps new region indices to old region indices. The function begins on line 1 by defining K as the number of regions in the new region map. Lines 2 and 3 initialize the lookup table B and the vertex list V with K elements. Then the algorithm loops on lines 4-16 for each region index k in K. Line 5 finds the grid cells in the new region map L that have index k and saves these as the set S. Line 6 saves S as the region cells of vertex $V[k]$. On line 7, we define T as the set of old region indices from $\mathcal{M}.L$ that are included in the set S. We examine each index t in T on lines 8-12. Line 9 gets the set of cells from the old region map that have index t and saves these as the set U. If the sets S and U are identical (line 10), then we save the index t to $B[k]$ (line 11) and copy the old vertex center from index t to the new vertex list at index k (line 12). The lookup

table B will contain the corresponding old region index for each new region index if there is a matching region; otherwise it will be zero. Line 13 checks if $B[k] = 0$, which indicates that there was no matching region. In this case, the region center needs to be recomputed using Algorithm 3.9. Lines 14 and 15 prepare a grid R for the GET_REGION_CENTERS function, which is called on line 16. The lookup table B and new vertex list V are returned on line 17.

Algorithm 5.20 Update Region Graph Vertices

UPDATE_REGION_GRAPH_VERTICES($\mathcal{M}$, L)
1: $K \leftarrow \max(L)$
2: $B \leftarrow K$-dimensional vector initialized to 0
3: $V \leftarrow$ list of K uninitialized vertices
4: **for** k in 1 to K
5: $S \leftarrow \{(i, j) \mid L[i, j] = k\}$ *// Get cells with this new label*
6: $V[k].region \leftarrow S$
7: $T \leftarrow \{\mathcal{M}.L[i, j] \mid (i, j) \in S\}$ *// Get all old labels assigned to these cells*
8: **for** each $t \in T$
9: $U \leftarrow \{(i, j) \mid \mathcal{M}.L[i, j] = t\}$ *// Get cells with this old label*
10: **if** $S = U$ *// New and old regions are identical*
11: $B[k] \leftarrow t$
12: $V[k].center \leftarrow \mathcal{M}.G_R.V[t].center$
13: **if** $B[k] = 0$ *// New region needs to be recomputed*
14: $R \leftarrow n \times m$ grid initalized to 0
15: $R[S] \leftarrow 1$
16: $V[k].center \leftarrow$ GET_REGION_CENTERS(R) *// Algorithm 3.9*
17: **return** B, V

After creating the lookup table B and the new vertex list V, these variables are passed to the UPDATE_REGION_GRAPH_EDGES function in Algorithm 5.21 along with the current mental map structure $\mathcal{M}$ and the new region label map L. This function also begins on line 1 by defining K as the number of regions in L. Line 2 initializes the adjacency

matrix A as a $K \times K$ matrix set to all zeros and line 3 creates an empty edge list E. Line 4 starts the edge index counter at zero and lines 5-19 loop over each region index k in K. For each index, lines 6 and 7 create a mask U of the grid cells in the region. Lines 8 and 9 get the neighboring grid cells using image dilation and store the neighboring region labels as the set N. Lines 10-19 loop over each neighboring region label n in N. An edge is created for each neighbor between the vertices assigned to region indices k and n. For each neighbor, the edge index i is incremented (line 11) and then stored in the adjacency matrix at $A[k][n]$ (line 12). We then use the lookup table B, to check if both regions already exist in the region graph (line 13). If so, then line 14 gets the index of the existing edge using the old adjacency matrix $\mathcal{M}.G_R.A$ and line 15 saves the features in the new edge list E. If either region does not already exist in the old region graph, the features need to be recomputed. Lines 17 and 18 prepare the region map R used to compute the region features on line 19 using Algorithm 5.10. After evaluating all the neighbors for each region, the final adjacency matrix A and edge list E are returned on line 20.

The UPDATE_REGION_GRAPH function in Algorithm 5.19 uses the vertex list computed by the UPDATE_REGION_GRAPH_VERTICES function and the adjacency matrix and edge list computed by the UPDATE_REGION_GRAPH_EDGES function to construct the new region graph G_R. This is returned to the UPDATE_MENTAL_MAP_REGIONS function on line 7 of Algorithm 5.16. The updated region labels, local region, and region graph are all saved to the mental map structure $\mathcal{M}$ on lines 8-10 of Algorithm 5.16 and the new mental map can then be used to plan future actions as will be discussed in the next chapter.

Algorithm 5.21 Update Region Graph Edges

```
UPDATE_REGION_GRAPH_EDGES(𝓜, L, B, V)
1:  K ← max(L)
2:  A ← K × K adjacency matrix initalized to 0
3:  E ← empty list of edge features
4:  i ← 0
5:  for k in 1 to K
6:      U ← n × m grid initalized to 0
7:      U[V[k].region] ← 1
8:      U' ← U ⊕ [0 1 0; 1 1 1; 0 1 0]        // Dilate to get neighboring cells
9:      N ← {l | l ∈ L[U' = 1] ∧ l ≠ 0 ∧ l ≠ k}
10:     for n in 1 to |N|
11:         i ← i + 1
12:         A[k][n] ← i
13:         if B[k] > 0 ∧ B[n] > 0
14:             t ← 𝓜.G_R.A[B[k], B[n]]       // Get index of existing edge
15:             E[i] ← 𝓜.G_R.E[t]             // Save existing edge features
16:         else
17:             R ← U
18:             R[V[n].region] ← 2
19:             E[i] ← COMPUTE_REGION_FEATURES(𝓜, R)       // Algorithm 5.10
20: return A, E
```

5.6 Summary

This chapter defined how an agent can represent the contents of its mental map as a region graph. The region graph is a fuzzy weighted graph that represents the minimum, maximum, and average feature values that an agent could expect to encounter when moving between adjacent regions in the environment. The first step in creating the region graph is to define the region boundaries. We start with a local region around the agent that contains a copy of the action graph to ensure that the next immediate action chosen by the agent corresponds to an edge in the region graph. Observed resources are also placed in

single-cell regions so that each vertex in the region graph will contain no more than one resource. Observed terrain types and unobserved areas are clustered separately to make sure that each region has only one type of terrain. The region graph is then constructed with a vertex for each region and an edge connecting adjacent regions.

To compute the feature values, we begin by calculating the shortest path distance between each pair of cells in two bordering regions. We presented an algorithm to compute the distance matrices based on the distances within a region to each grid cell on the region boundary. From these matrices, we can compute the distance and terrain-based features. For the elevation features, we adapted the algorithm for non-uniform edge weights and minimax paths. We derived an estimate of the elevation features for unobserved regions using the expected feature values at different distances from the region boundary. A method for approximating these features without computing the shortest path distances was also presented.

Lastly, we looked at how the region graph is updated when the agent moves and discovers new information. The local region is redefined and regions that border newly observed grid cells are reclustered. Features between regions that have not changed are copied into the new graph and new features are computed for the remaining edges. The region graph provides the computational problem for the agent to solve each timestep in the form of a least-cost path problem. The next chapter addresses how an agent can solve these problems for a given set of objectives and decide a course of action.

6 LEAST-COST PATH PROBLEMS

This chapter shows how a fuzzy weighted graph, such as the region graph computed in the previous chapter, can be used to solve least-cost path problems in gridded domains and presents a greedy agent algorithm for solving these problems within the CMM framework. We begin with a discussion on the issue of selection bias that can arise when computing shortest paths in grid worlds. Next, we introduce the multiobjective fuzzy least-cost path problem and present a method to solve it using a pre-scalarized decomposition approach. This is then compared with an evolutionary method using MOEA/D. We show several experiments to demonstrate these methods and conclude with a description of a greedy algorithm that uses these techniques to solve generic problems in the CMM framework.

6.1 Shortest Paths in Grid Worlds

The most straightforward pathfinding problem is to determine the shortest path between two points in an environment. In the simplest case, where there are no obstacles and the agent is permitted to travel freely in Euclidean space, the shortest path is just a straight line. However, in grid-based environments, the agent can only move at right angles, similar to how one navigates the grid layout of most city blocks. This geometry is sometimes called a taxicab geometry, and the associated distance metric can be referred to as the taxicab metric, city block distance, or Manhattan distance. Formally, the distance

$d_1(c_p, c_q)$ between two grid cells $c_p = c(p_i, p_j)$ and $c_q = c(q_i, q_j)$ is defined as the L_1 norm,

$$d_1(c_p, c_q) = |p_i - q_i| + |p_j - q_j|. \tag{6.1}$$

When two grid cells are not in the same row or column, there may be many paths between the two cells that all have the same shortest distance. When the transition costs are all identical, any path that has the minimum required number of horizontal and vertical steps is a shortest path. If the two grid cells are far apart and diagonally separated, then the number of equidistant paths can grow to be very large.

If all the shortest paths between two grid cells are otherwise equivalent[1], they should each have an equal likelihood of being selected by the agent. However, selection bias can occur if the agent decides each step sequentially by randomly breaking ties between cells with the same remaining distance to the goal. To see this, consider the 3x4 grid world shown in Figure 6.1 (a). There are 10 unique shortest paths between the grid cells (1,1) and (3,4). The L_1 distance from cell (3,4) is shown in subfigure (b). To select a path, the agent starts at (1,1) and picks one of the adjacent cells with the smallest value. Since there are two cells with a distance of 4, the agent picks one uniformly at random. This process is repeated until the agent reaches the goal. Following this approach, the agent has a $(0.5)^2 = 0.25$ chance of passing through cell (3,1) and a $(0.5)^3 = 0.125$ chance of passing through cell (1,4) even though both cells only have a single path out of the 10 possible paths passing through them.

[1] Even when all transitions have the same cost, a perceptive agent might notice that some paths have fewer turns or some other desirable criteria. In the CMM framework, these preferences would be modeled as additional objectives in addition to shortest distance, leading to a multiobjective problem in which the paths are not considered identical.

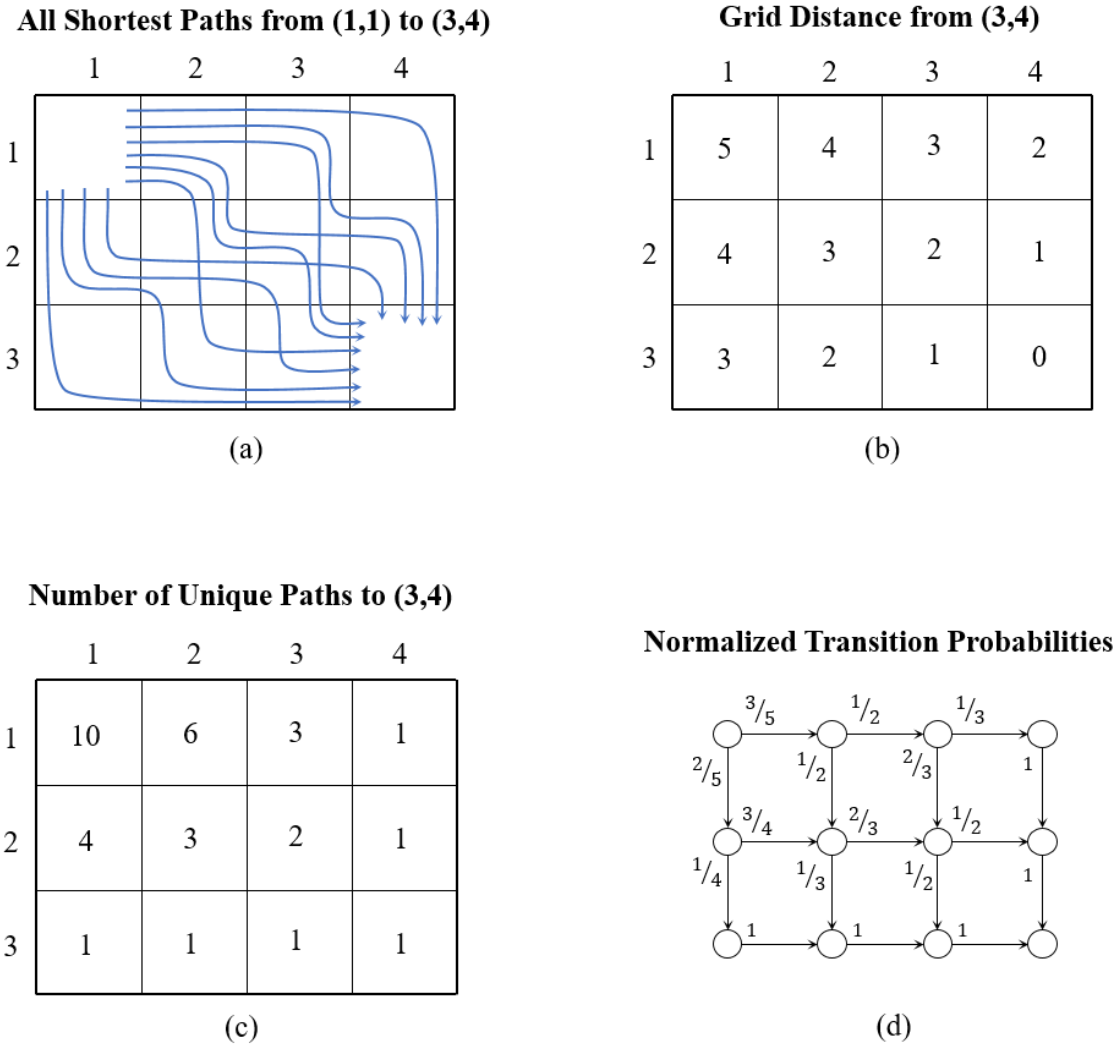


Figure 6.1 Example of the selection bias problem for choosing paths in grid-world domains. (a) There are 10 unique shortest paths between grid cells (1,1) and (3,4). (b) The L_1 distance is computed from (3,4). An agent at (1,1) picks a (biased) shortest path by repeatedly picking an adjacent cell with the smallest remaining distance, breaking ties uniformly at random. (c) The number of unique shortest paths leading to (3,4) is used to compute the weighted probabilities for each cell transition. (d) The normalized transition probabilities.

This issue is known as the selection bias (or label bias) problem, and is commonly addressed in the domain of conditional random fields (Lafferty, Mccallum, and Pereira 2001). A decision made early in the sequence can adversely affect the likelihood that certain options will be available later. For example, once the agent reaches the cell (3,1),

there is only a single way to reach the goal in three steps, yet this path will be chosen 25% of the time instead of 10%, which would better reflect the true likelihood of this path being chosen out of the 10 total paths.

To resolve this issue, the options at each decision point can be weighted by the total number of unique paths from each grid cell to the destination as shown in Figure 6.1 (c). These values are calculated at the same time as the grid distance values for each cell using the NORMALIZED_GRID_DISTANCE function in Algorithm 6.1. This function is similar to the GRID_DISTANCE function in Algorithm 3.6, but includes an additional output N that aggregates the number of unique paths from each grid cell to the target cell (i, j). This matrix is initialized with zeros on line 3 and will be filled in along with the distance map D as cells progressively farther from the target cell are examined. The algorithm sets the distance counter d to 0 on line 4 and creates an open set on line 5 containing only the cell (i, j). The main loop (lines 7-20) is evaluated for each distance d up to a maximum of d_{max} while there are still cells in the open set. For each distance, a new frontier set is initialized (line 8) and the current N matrix is copied as N' (line 9). Each cell (u, v) in the open set is then examined (lines 10-18) and assigned the current distance value (line 11). The frontier set is updated with all unprocessed neighboring cells (lines 12-14) and the number of unique paths is calculated. For the first iteration, $N[u, v]$ is set to 1 (lines 15 and 16), and for later iterations, $N[u, v]$ is set to the sum of the N' values of all neighboring cells (line 18). Since only the neighbors that were added in the previous iteration will have non-zero values in N' at the time of calculation, their sum represents the total number of unique paths to the target cell. It should be noted that this number can grow large very quickly if the grid

has large open spaces. For example, applying Algorithm 6.1 on a 100x100 open grid yields $2.3\text{x}10^{58}$ unique paths between the opposite corners!

Algorithm 6.1 Normalized Grid Distance

NORMALIZED_GRID_DISTANCE(W, i, j, d_{max})
1: $(n, m) \leftarrow$ size of W
2: $D \leftarrow n \times m$ matrix initalized to ∞
3: $N \leftarrow n \times m$ matrix initalized to 0
4: $d \leftarrow 0$
5: $open \leftarrow \{(i, j)\}$
6: $closed \leftarrow \emptyset$
7: **while** $|open| > 0 \wedge d \leq d_{max}$
8: $\quad frontier \leftarrow \emptyset$
9: $\quad N' \leftarrow N$
10: $\quad$ **for** each $(u, v) \in open$
11: $\quad\quad D[u, v] \leftarrow d$
12: $\quad\quad closed \leftarrow closed \cup (u, v)$
13: $\quad\quad B \leftarrow \{(u-1, v), (u+1, v), (u, v-1), (u, v+1)\}$
14: $\quad\quad frontier \leftarrow frontier \cup \{(u', v') \mid (u', v') \in B \wedge (u', v') \notin closed \wedge W[u', v'] = 1\}$
15: $\quad\quad$ **if** $d = 0$
16: $\quad\quad\quad N[u, v] \leftarrow 1$
17: $\quad\quad$ **else**
18: $\quad\quad\quad N[u, v] \leftarrow \sum_{(u',v') \in B} N'[u', v']$
19: $\quad open \leftarrow frontier$
20: $\quad d \leftarrow d + 1$
21: **return** D, N

Having computed the number of paths through each grid cell, the agent can use these values to normalize the probability of selecting the next location. Rather than giving equal weight to each option, the probability of transitioning from cell u to cell v is computed as $N(v)/N(u)$, where $N(u)$ is the number of paths passing through cell u and $N(v)$ is the number of paths passing through cell v as computed by Algorithm 6.1. These transition probabilities are shown in Figure 6.1 (d).

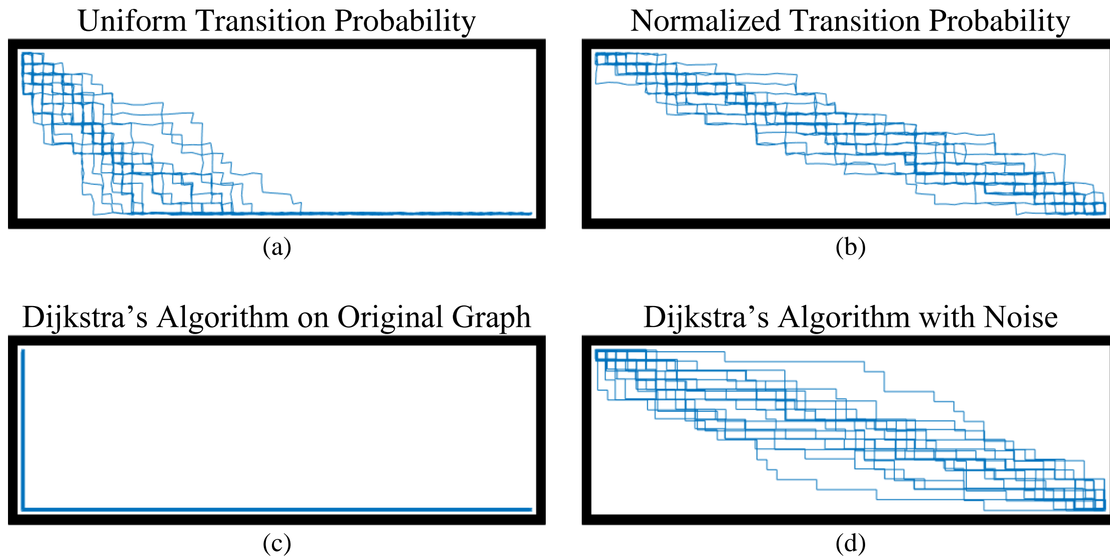


Figure 6.2 Examples of shortest paths chosen between opposite corners of an open grid world. (a) Paths are sampled by starting in the upper-left and selecting each transition step with uniform probability. (b) Paths are sampled by starting in the upper-left and selecting each transition step according to the normalized transition probabilities. (c) Dijkstra's shortest path algorithm with a deterministic tie-breaking rule. (d) Dijkstra's shortest path algorithm on a graph with a small amount of uniform random noise applied to the edge weights.

To demonstrate the effect of selection bias, consider the example in Figure 6.2. Each of the subfigures shows 20 of the shortest paths sampled from the upper-left corner to the lower-right corner of an open grid world. Subfigure (a) shows the paths produced by using a uniform transition probability to select each subsequent grid cell. Notice how the paths generally follow a 45° angle to the lower-right until reaching the bottom edge and then follow the same path to the goal. With this selection strategy, the direction of the path is clearly apparent as having originated in the upper-left and terminating in the lower-right. It is very unlikely for a path to approach the lower-right corner from above with this method. This contrasts with the paths sampled in subfigure (b), where the transition probabilities are normalized using Algorithm 6.1. In this approach, the paths tend to lie on the true diagonal between the two corners and the direction is symmetric.

Clearly the normalized transition probabilities lead to a better sampling of the true set of shortest paths without selection bias. In practice, however, we may wish to use an existing implementation of Dijkstra's algorithm to compute the shortest path between two points[1]. This is especially true when the environment is represented as a graph rather than a grid. Most implementations of Dijkstra's algorithm will return only a single path from a vertex to the source, if a path is returned at all. (Some implementations return only the shortest path distances from which a path can be constructed by backtracking.) Figure 6.2 (c) shows the single path returned by the Matlab implementation of Dijkstra's algorithm when computing the shortest path between the opposite corners of an open world grid. The path is constructed using a deterministic rule based on the lexicographic ordering of the graph vertices and is not very representative of the distribution of all shortest paths. The returned path can be improved by adding a small amount of uniform random noise[2] to each edge weight before computing the shortest path distances. This makes it highly unlikely for any two paths to have the same distance, resulting in a single shortest path returned by the algorithm. A sample of these paths are shown in Figure 6.2 (d), where each path is found using different noise values. Notice that the path distribution closely matches the ideal distribution in Figure 6.2 (b). This approach of adding a small amount of random noise when computing shortest paths is used throughout our experiments with the CMM framework to produce more natural looking paths when path length is otherwise equivalent.

---

[1] Efficient implementations of Dijkstra's algorithm will utilize a priority queue data structure such as a Fibonacci heap and can make other domain-specific optimizations.

[2] The noise values should be much smaller than the default edge weights, otherwise the shortest path algorithm may return a longer path than the true shortest path distance. Unless otherwise stated, we sample noise values from a uniform random distribution on the interval $(0, 10^{-14})$.

6.2 The Multiobjective Fuzzy Least-Cost Path Problem

The fundamental component of any pathfinding algorithm in the CMM framework is to find a least-cost path between two locations. This can be modeled as a multiobjective fuzzy least-cost path problem (MO-FLCPP) between two vertices in the region graph defined in Chapter 5. The region graph G_R is a fuzzy weighted graph in which vertices represent regions in the environment and edges between adjacent regions are weighted with multiple fuzzy feature values indicating the minimum, maximum, and average costs associated with traversing each edge. The set $P(s,t)$ includes all paths from vertex s to vertex t through the graph. The goal of the MO-FLCPP is to find a path $p \in P(s,t)$ that minimizes the aggregated cost vector $\boldsymbol{A}(p) = \big(A_1(p), \dots, A_m(p)\big)$, where each component $A_i(p)$ represents the aggregated cost of feature i along path p. The agent specifies an indicator vector $\boldsymbol{\gamma} = (\gamma_1, \dots, \gamma_m)$, where $\gamma_i = 0$ if feature i should be aggregated by summation and $\gamma_i = 1$ if feature i should be aggregated using maximization. Recall from Section 2.4 that the aggregated value of feature i is defined as

$$A_i(p) = \begin{cases} \sum_{j=1}^{n} F_i(e_j), & \gamma_i = 0 \\ \max'_{j=1,\dots,n} F_i(e_j), & \gamma_i = 1 \end{cases}, \tag{6.2}$$

where $F_i(e_j)$ is a triangular fuzzy number $\mathrm{Tri}\big(a_{ij}, b_{ij}, c_{ij}\big)$ that represents the cost of feature i for edge j in the path $p = (e_1, \dots, e_n)$. The $\max'$ operator approximates the maximum of a set of triangular fuzzy numbers as a triangular fuzzy number and is defined in Equation 2.11. We can find a path that solves the MO-FLCPP using multiobjective optimization techniques.

6.2.1 Multiobjective Optimization for the MO-FLCPP

In the jargon of multiobjective optimization from Section 2.5, the MO-FLCPP is defined as

$$\begin{array}{ll} \text{minimize} & \boldsymbol{A}(p) = \left(A_1(p), \dots, A_m(p)\right) \\ \text{subject to} & p \in P(s,t), \end{array}$$

where $m \geq 2$. If $m = 1$, then there is only a single objective and the least-cost path can be found using a standard implementation of Dijkstra's algorithm. When there are multiple conflicting objectives, the minimum value of one objective cannot be obtained without some tradeoff in the other objectives. Nevertheless, some solutions (paths) are clearly better than others. We say that a path p dominates path p' ($p \prec p'$) if and only if $A_i(p) \leq A_i(p')$ for all $i = 1, \dots, m$ and there exists a $j \in \{1, \dots, m\}$ such that $A_j(p) < A_j(p')$. A path that dominates another path is at least as good as the other path in all objectives and is better in at least one objective. A path that is not dominated by any other known solution is said to be Pareto optimal. Formally, the Pareto optimal set PS is defined as

$$PS = \{p \in P(s,t) \mid \{p' \in P(s,t) \mid p' \prec p\} = \emptyset\}. \tag{6.3}$$

The multiobjective cost vectors of the paths in PS define the Pareto front,

$$PF = \{\boldsymbol{A}(p) \mid p \in PS\}. \tag{6.4}$$

The native units of each objective may be incomparable, making it difficult to assess the relative value of each solution. To make the comparison between solutions meaningful, the original cost vectors are normalized into a unit hypercube. This ensures that each objective is treated equally. For instance, if the distance cost is measured in meters and the slope cost is measured as a percentage of some reference angle, the magnitudes of these

two dimensions should be normalized before being compared. To normalize the vectors, the minimum value of each objective is defined as zero and the maximum value is defined by the reference point $\boldsymbol{z}^* = (z_1^*, \dots, z_m^*)$. Determining the optimal value of $\boldsymbol{z}^*$ is not a trivial task and the value that is chosen can greatly affect the resulting decision. Ideally, $\boldsymbol{z}^*$ should be the nadir vector of the Pareto front, in which each z_i^* is defined as

$$z_i^* = \max_{p \in PS'} c_{ip}, \tag{6.5}$$

where $A_i(p) = \text{Tri}\left(a_{ip}, b_{ip}, c_{ip}\right)$. Here, PS' is the current best approximation of the Pareto optimal set since the true set may be unknown. The normalized cost vectors are then computed as $\boldsymbol{A}'(p) = \left(A_1'(p), \dots, A_m'(p)\right)$, where

$$A_i'(p) = \text{Tri}\left(\frac{a_{ip}}{z_i^*}, \frac{b_{ip}}{z_i^*}, \frac{c_{ip}}{z_i^*}\right) \tag{6.6}$$

for each $i = 1, \dots, m$.

6.2.2 Scalarization

All solutions that are members of the Pareto optimal set would be rational choices for the decision-maker. However, the agent must ultimately choose a single path to follow. Typically, this decision is made using a scalarization function that reduces the multiobjective optimization problem into a single objective optimization problem. Given a multidimensional fuzzy cost vector $\boldsymbol{X} = (X_1, \dots, X_m)$ where each X_i is a fuzzy number, and an objective weight vector $\boldsymbol{\lambda} = (\lambda_1, \dots, \lambda_m)$ where $\lambda_i \geq 0$ and $\sum_i \lambda_i = 1$ for $i = 1, \dots, m$, the scalarization function $g(\boldsymbol{X}|\boldsymbol{\lambda})$ reduces the cost vector $\boldsymbol{X}$ to a single fuzzy number. This

value can then be used to rank and compare various alternatives, with smaller values being preferred. The scalarized MO-FLCPP is defined as

$$\begin{array}{rl} \text{minimize} & g(\boldsymbol{A}'(p)|\boldsymbol{\lambda}) \\ \text{subject to} & p \in P(s,t). \end{array}$$

The path p that minimizes the scalarized value of the normalized aggregated cost vector $\boldsymbol{A}'(p)$ is chosen as the preferred solution. The objective weight vector $\boldsymbol{\lambda}$ represents the relative importance of each objective to the decision-maker, with more important objectives receiving higher weights. We consider three different scalarization functions: weighted sum, Tchebycheff, and ordered weighted average.

One of the most common scalarization methods is the weighted sum, which maintains a linear relationship between the decision-maker's preferences and the scalarized cost value. This is defined as

$$g^{\text{ws}}(\boldsymbol{X}|\boldsymbol{\lambda}) = \sum_{i=1}^{m} \lambda_i X_i, \tag{6.7}$$

where the multiplication of a scalar λ and a triangular fuzzy number $\text{Tri}(a,b,c)$ is defined as $\text{Tri}(\lambda a, \lambda b, \lambda c)$. If the shape of the Pareto front is convex, then the weighted sum can be a good choice because every Pareto optimal solution can be made to have the lowest scalarized cost by changing the objective weight vector. However, if the shape of the Pareto front is non-convex, then there will always be some Pareto optimal solution that can never be chosen. For more details, refer to Section 2.5.5.

A simple alternative to the weighted sum approach is the Tchebycheff method, which can be parameterized with different objective weight vectors to make any Pareto optimal

solution have the lowest scalarized cost. The Tchebycheff scalarization function is defined as

$$g^{\text{te}}(\boldsymbol{X}|\boldsymbol{\lambda}) = \max_{i=1,\ldots,m}{}' \lambda_i X_i. \tag{6.8}$$

This method evaluates the quality of a solution as the least satisfied weighted objective value. A single high cost for one objective can penalize an otherwise good solution.

The last scalarization approach we consider is based on the ordered weighted average operator (OWA) proposed by Yager (Yager 1988). This method requires the definition of an additional scalar weight vector $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_m)$ where $\theta_i \geq 0$ and $\sum_i \theta_i = 1$ for $i = 1, \ldots, m$. Each θ_i represents the weighted contribution of the ith largest scaled vector component. First, the cost vector $\boldsymbol{X}$ is scaled by the objective weight vector λ to give the scaled cost vector $\boldsymbol{Y} = (Y_1, \ldots, Y_m)$, where $Y_i = \lambda_i X_i = \text{Tri}(a_i^Y, b_i^Y, c_i^Y)$ for $i = 1, \ldots, m$. Next, we independently sort all the a_i^Y, b_i^Y, and c_i^Y values and define the lists $\left(a_{(1)}^Y, \ldots, a_{(m)}^Y\right)$, $\left(b_{(1)}^Y, \ldots, b_{(m)}^Y\right)$, and $\left(c_{(1)}^Y, \ldots, c_{(m)}^Y\right)$, where $a_{(i)}^Y$, $b_{(i)}^Y$, and $c_{(i)}^Y$, are the ith largest values in their respective lists. Once this is done, the OWA scalarization function is defined as

$$g^{\text{OWA}}(\boldsymbol{X}|\boldsymbol{\lambda}, \boldsymbol{\theta}) = \sum_{i=1}^{m} \theta_i \text{Tri}\left(a_{(i)}^Y, b_{(i)}^Y, c_{(i)}^Y\right). \tag{6.9}$$

The OWA scalarization method can be made to represent many different functions by changing the weight vector $\boldsymbol{\theta}$. For instance, the OWA operator behaves as the weighted sum when $\theta_i = \frac{1}{m}$ for all $i = 1, \ldots, m$. (Although the ordering of solutions in this case is the same as the weighted sum, the actual values may be different due to the additional scaling.) The Tchebycheff method is equivalent to setting $\theta_1 = 1$ and $\theta_i = 0$ for all $i \neq 1$.

We can implement a form of bounded rationality (Simon 1955) for the decision-maker by defining a weight vector of the form $\theta_i = \frac{1}{q}$ for $i = 1, \dots, q$ and $\theta_i = 0$ for all $i > q$. This represents the case where the decision-maker does not have the necessary computational resources to consider all objectives simultaneously and bases the decision on only the q least satisfied objective values. Typically, $\boldsymbol{\theta}$ is fixed for a given decision-maker, so for notational conciseness, we omit the $\boldsymbol{\theta}$ parameter of the OWA scalarization function when referring to a general scalarization function.

We use one of these scalarization functions $g(\boldsymbol{X}|\boldsymbol{\lambda})$ to reduce the fuzzy cost vector $\boldsymbol{X}$, representing a possible solution to the MO-FLCPP, to a single fuzzy value $S(\boldsymbol{X})$. To compare different solutions, the decision-maker uses the weighted centroid approach (Section 2.4.2) to defuzzify each alternative and ranks the resulting crisp values, favoring solutions with smaller values. The weighted centroid method allows the decision-maker to specify a degree of optimism or pessimism, given as the parameter $\xi \in [0, 1]$. When $\xi = 0$, the decision-maker is extremely optimistic and uses the smallest possible value, whereas when $\xi = 1$, the decision-maker is extremely pessimistic and uses the largest possible value. A value of $\xi = 0.5$ provides a balanced approach using the centroid of the fuzzy number. The crisp defuzzified value is computed as $C(S(\boldsymbol{X})|\xi)$ using Equation 2.14.

6.2.3 Example

To demonstrate the MO-FLCPP, consider the example graph in Figure 6.3. This graph has two features assigned to each edge representing distance and slope. The features come from different unrelated domains and are represented as linguistic variables defined

by the triangular fuzzy numbers in Figure 6.4. One can imagine that this graph represents an environment with a tall hill at vertex 3 and various ways of navigating over or around the hill to get from vertex 1 to vertex 5. The multiobjective cost function consists of a distance feature and a slope feature, where the decision-maker seeks to find a path with the shortest total distance and the smallest maximum slope. In this case, the distance feature is aggregated using summation, whereas the slope feature uses maximization. There are five unique paths between vertices 1 and 5 in the example graph. The aggregated feature values of the paths are given in Table 6.1 and are plotted in Figure 6.5. All paths except the yellow path (1-3-4-5) are members of the Pareto optimal set. The yellow path is dominated by both the red (1-3-5) and green (1-2-3-5) paths.

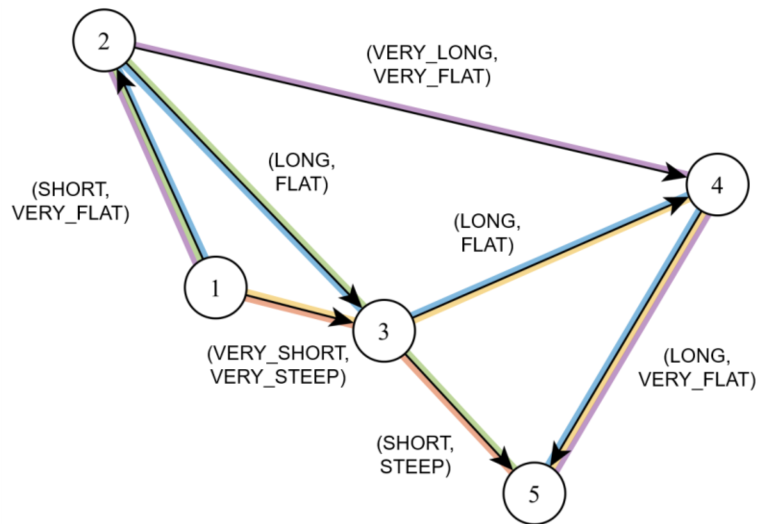


Figure 6.3 An example fuzzy weighted graph with two features per edge, distance and slope, represented as triangular fuzzy numbers given in Figure 6.4. There are five unique paths between the vertices 1 and 5 colored red, yellow, green, blue, and purple.

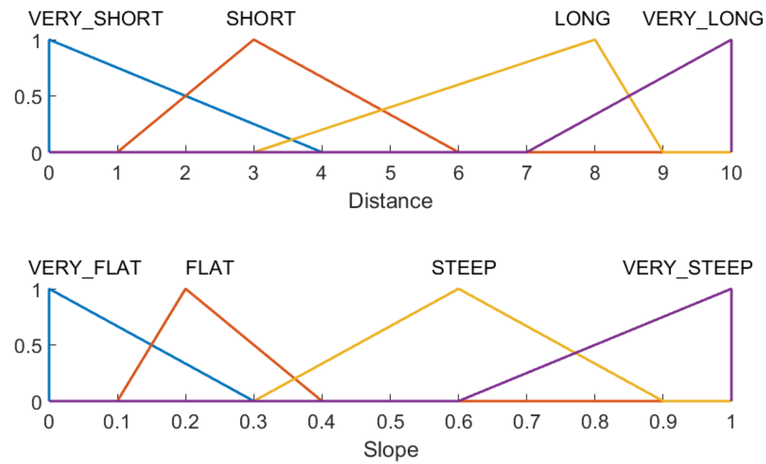


Figure 6.4 Triangular fuzzy numbers used to represent the distance and slope features for the example graph in Figure 6.3.

Table 6.1 Aggregated feature values of the example graph in Figure 6.3

Path	Color	Total Distance	Max Slope
1-3-5	Red	$\text{Tri}(1, 3, 10)$	$\text{Tri}(0.6, 1, 1)$
1-3-4-5	Yellow	$\text{Tri}(6, 16, 22)$	$\text{Tri}(0.6, 1, 1)$
1-2-3-5	Green	$\text{Tri}(5, 14, 21)$	$\text{Tri}(0.3, 0.6, 0.9)$
1-2-3-4-5	Blue	$\text{Tri}(10, 27, 33)$	$\text{Tri}(0.1, 0.2, 0.4)$
1-2-4-5	Purple	$\text{Tri}(11, 21, 25)$	$\text{Tri}(0, 0, 0.3)$

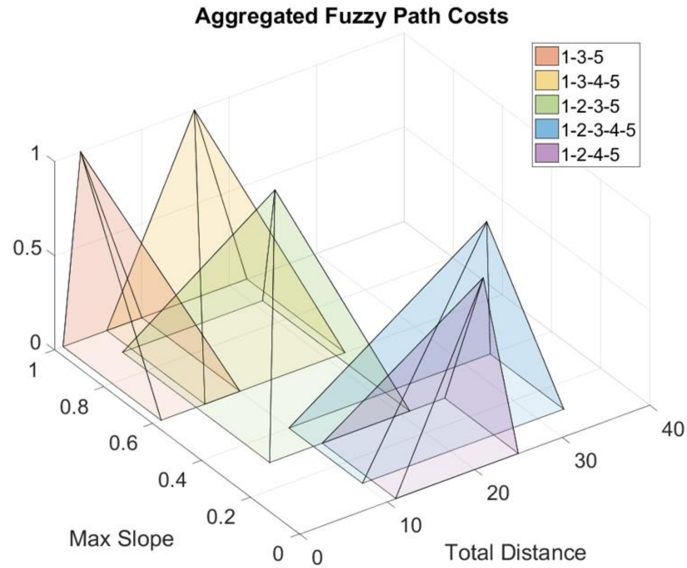


Figure 6.5 Plots of the two-dimensional aggregated fuzzy cost vectors for each path in the example graph from Figure 6.3.

Since there are only five paths to consider, the Pareto optimal set can be determined directly and the reference point is evaluated as the nadir vector $\boldsymbol{z}^* = (33, 1)$, as these are the largest possible values of the aggregated distance and slope features. Figure 6.6 shows each of the normalized cost vectors after applying weighted centroid defuzzification to each feature. (We typically wait until after scalarizing the cost vectors to apply defuzzification, but this example helps show the process.) The black dotted lines show the location of the Pareto front for different values of ξ. From this we can see that the yellow path is always dominated, whereas the blue path (1-2-3-4-5) is dominated by the purple path (1-2-4-5) when $\xi = 0.5$ and $\xi = 1$. The blue path is only Pareto optimal when the decision-maker is very optimistic (i.e. expects the true cost of each path segment to be small).

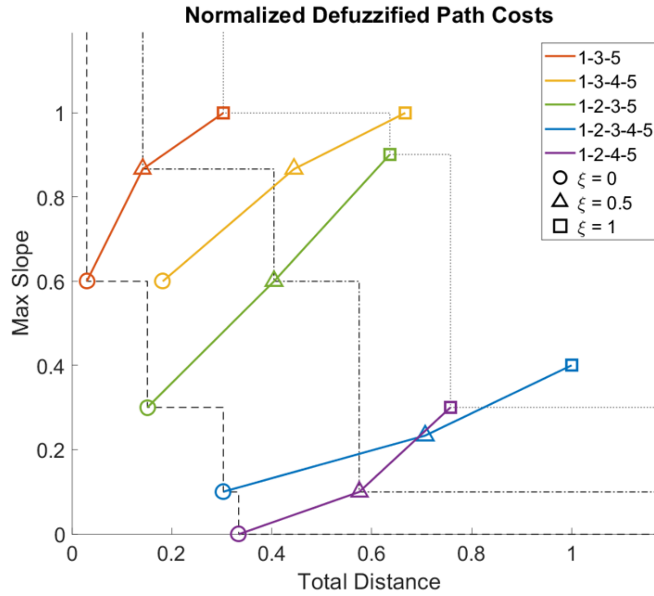


Figure 6.6 The aggregated fuzzy cost vectors from Figure 6.5 are normalized using the nadir vector and defuzzified using weighted centroid defuzzification. The black dotted lines show the Pareto fronts for different values of ξ.

To demonstrate the scalarization process, consider three different decision-makers that must choose a solution for this example problem. The first uses the weighted sum scalarization method with $\boldsymbol{\lambda} = (0.5, 0.5)$ and $\xi = 0.5$. Applying g^{ws} to each of the aggregated fuzzy path cost vectors in Table 6.1 gives the fuzzy values shown in Figure 6.7 (a). The weighted centroid of each path is shown with a circle and a vertical line. The decision-maker chooses the path with the smallest defuzzified cost, which is the purple path. A different decision-maker using the Tchebycheff method with $\boldsymbol{\lambda} = (0.25, 0.75)$ and $\xi = 0$ computes the values shown in Figure 6.7 (b). This is one of the few conditions where the blue path is evaluated as the lowest cost option. The last decision-maker shown in Figure 6.7 (c) uses the OWA scalarization method with $\boldsymbol{\lambda} = (0.9, 0.1)$, $\boldsymbol{\theta} = (0.7, 0.3)$ and

$\xi = 1$. This represents extreme pessimism with a strong bias towards minimizing the distance feature, which results in giving the red path the lowest cost.

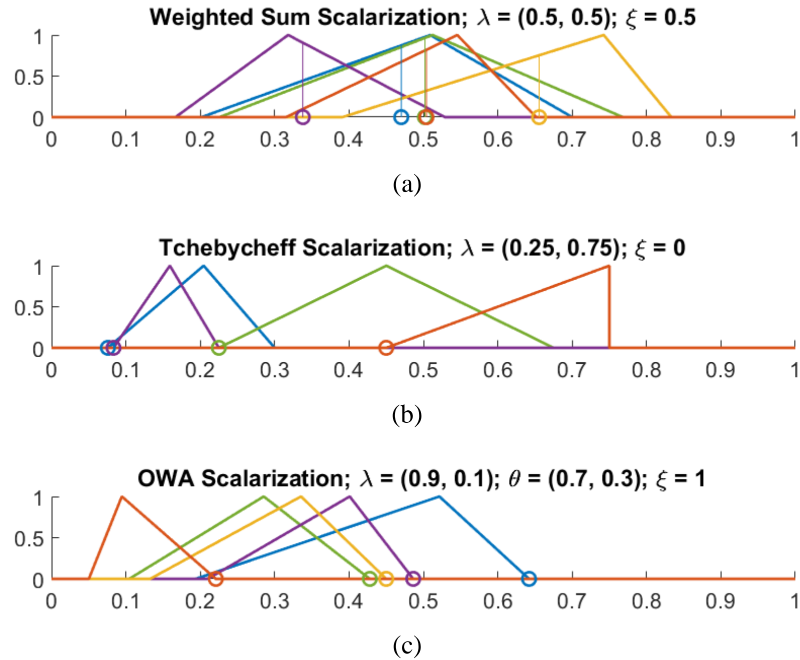


Figure 6.7 Examples of different scalarization methods applied to the aggregated fuzzy cost vectors given in Table 6.1. Each method represents a decision-maker with different preferences. The scalarized fuzzy numbers shown in the plots are defuzzified (shown as a circle and vertical line) and the decision-maker chooses the path with the lowest defuzzified cost.

6.3 Decomposition of the MO-FLCPP

In the previous section, we assumed that the candidate paths to be evaluated have already been provided or are straightforward to determine, but in larger problems such as those generated by the CMM framework, this may not be the case. If a decision-maker can express their preferences *a priori*, then we can scalarize the MO-FLCPP into a single-objective shortest path problem (SPP) that can be solved directly using Dijkstra's algorithm (Dijkstra 1959). However, this may not find the same ideal solution that would be obtained

if the same preferences were to be applied to all possible paths using the multiobjective approach presented in the previous section. Nevertheless, it can still be useful to decompose a multiobjective problem into many simpler single-objective problems that are easier to solve. These can be used to construct initial candidate solutions and may be sufficient for some applications.

6.3.1 Edge Normalization

Each subproblem in the decomposition is defined by a unique objective weight vector $\boldsymbol{\lambda}$. In addition, the following parameters are required to specify how the edge costs of the SPP should be computed:

- g: a scalarization function (either g^{ws}, g^{te}, or g^{OWA});
- $\boldsymbol{z}^*$: a reference point for normalizing the edge features;
- $\boldsymbol{\gamma}$: an aggregation indicator vector;
- ξ: a defuzzification parameter;
- $\boldsymbol{\theta}$: an OWA weight vector if using g^{OWA}.

Except for the reference point $\boldsymbol{z}^*$, these parameters are defined by the decision-maker *a priori*. If $\boldsymbol{z}^*$ is not clear from the problem context, it will need to be estimated from the cost vectors of the graph edges.

Given a MO-FLCPP defined between two vertices s and t in a graph G, the feasible region of the decision space is defined as all paths in the set $P(s,t)$. The cost vectors $\boldsymbol{A}(p)$ are normalized after aggregating the individual edge costs $\boldsymbol{F}(e)$ for each edge e in a path $p \in P(s,t)$. However, to decompose the problem as a SPP, each edge must be scalarized to a single value before aggregating. Since the nadir vector of the Pareto front is unknown

beforehand, we use the maximum values of each objective feature over all edges to define $\boldsymbol{z}^*$. Each value of z_i^* is then defined as

$$z_i^* = \max_{e \in E(G)} c_{ie}, \tag{6.10}$$

where the value of feature i for edge e is given as $F_i(e) = \text{Tri}(a_{ie}, b_{ie}, c_{ie})$. The normalized edge costs are computed as $\boldsymbol{F}'(e) = \left(F_1'(e), \ldots, F_m'(e)\right)$, where

$$F_i'(e) = \text{Tri}\left(\frac{a_{ie}}{z_i^*}, \frac{b_{ie}}{z_i^*}, \frac{c_{ie}}{z_i^*}\right) \tag{6.11}$$

for each $i = 1, \ldots, m$. Note that this method can cause summation objectives $(\gamma_i = 0)$ to become greater than one when aggregated over an entire path. This is not a great concern for the decomposed problem, since length of a path in the SPP just needs to be proportional to the aggregated cost of that path in the MO-FLCPP. Still, the difference between the value of $\boldsymbol{z}^*$ that is found by Equation 6.10 and the value found by Equation 6.5, which was based on the objective values in the Pareto front, can change which path is found as the optimal solution. For the example problem in Section 6.2.3, $\boldsymbol{z}^*$ would be computed as $(10, 1)$ using max-edge normalization, as opposed to the nadir vector of $(33, 1)$. In general, the reference point should be defined as the nadir vector of the Pareto front when possible and the max-edge normalization should only be used to create initial solutions.

6.3.2 Exponential Scaling

When scalarization is performed after the path costs have already been aggregated, the scalarization function can ignore which aggregation operator was used for each objective. However, the SPP only uses *total* path length as the objective to minimize. To decompose the problem, the edge costs need to be scalarized so that maximization objectives $(\gamma_i = 1)$

can be summed. While there is no perfect encoding that can achieve this, a sufficiently good approximation can be found using exponential scaling.

Consider a crisp single-objective minimax path problem where the goal is to find a path p that minimizes the maximum value of each edge $e \in p$. The total cost of the path is defined as

$$A(p) = \max_{e \in p} F'(e), \tag{6.12}$$

where $F'(e) \in [0, 1]$ is the normalized crisp scalar cost of edge e. The purpose of exponential scaling is to adjust the individual edge costs so that only the largest (maximum) values have any significant contribution when the costs are summed over the length of a path. Each edge cost x is scaled exponentially as

$$h(x) = \frac{\exp\big(-\,log(\varepsilon)\,(x-1)\big) - \varepsilon}{1-\varepsilon}, \tag{6.13}$$

where ε is a small positive number ($\ll 1$) that defines the amount of scaling. The effect of this scaling is shown in Figure 6.8, where smaller values of x are pushed closer to zero and larger values are distributed across a wider output range. If the edge costs are fuzzy values, then exponential scaling is applied using the extension principle,

$$h\big(\mathrm{Tri}(a, b, c)\big) = \mathrm{Tri}\big(h(a), h(b), h(c)\big). \tag{6.14}$$

The resulting edge costs can then be defuzzified to crisp values using the weighted centroid method and a defuzzification parameter ξ. A shortest path algorithm (i.e. Dijkstra) will find a path that minimizes the sum of all $h(x)$ values, which is an approximation of the minimax path. Smaller values of the scaling parameter ε give better approximations, but care should

be taken to avoid numerical underflow. We set $\varepsilon = 10^{-12}$ in our experiments with the CMM framework.

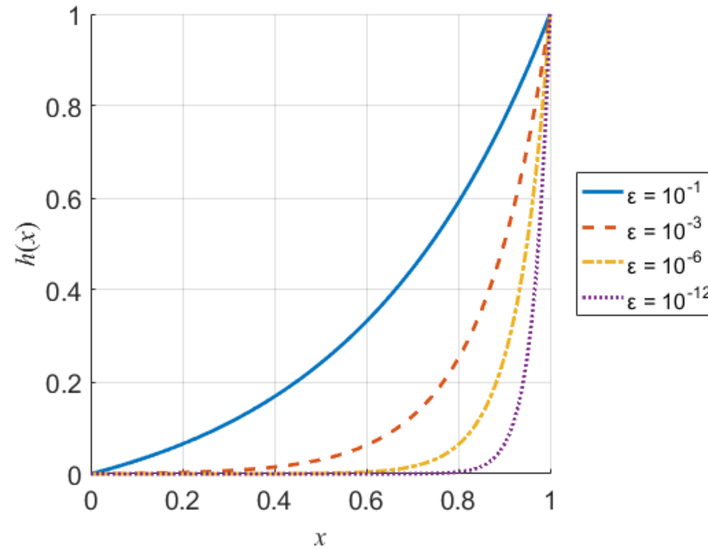


Figure 6.8 Exponential scaling of a normalized edge cost x. The resulting $h(x)$ values can be used to find an approximate minimax path via summation.

When there are multiple objectives, the scalarization function is applied to each edge before running a shortest path algorithm. If both summation and maximization aggregation operators are used, the objective weight vector $\boldsymbol{\lambda}$ needs to be adjusted to account for the exponential scaling of the maximized edge costs. Let $\eta = \sum_{i=1}^{m} \lambda_i (1 - \gamma_i)$ be the fraction of the original objective weight vector that represents summation objectives. Instead of scaling the cost values associated with these objectives, we scale the objective weights

using the same function given in Equation 6.13. A new scaled objective weight vector $\boldsymbol{\lambda}' = (\lambda_1', \dots, \lambda_m')$ is defined where

$$\lambda_i' = \begin{cases} \dfrac{h(\eta)\lambda_i}{\eta}, & \gamma_i = 0 \\ \dfrac{\left(1 - h(\eta)\right)\lambda_i}{1 - \eta}, & \gamma_i = 1 \end{cases}. \tag{6.15}$$

This effectively redistributes the objective weights so that summation objectives are given exponentially smaller weights to compensate for the exponentially smaller maximization cost values.

6.3.3 Pre-scalarized Decomposition

Once the edge features of a fuzzy weighted graph have been scalarized and defuzzified into non-negative crisp values, any shortest path algorithm can be used to find an approximate solution to the MO-FLCPP. Algorithm 6.2 gives an overview of the pre-scalarized decomposition approach for solving the MO-FLCPP as a shortest path problem.

Algorithm 6.2 Pre-scalarized Decomposition of the MO-FLCPP

Input:

- MO-FLCPP
- g: a scalarization function (either g^{ws}, g^{te}, or g^{OWA})
- $\boldsymbol{\lambda}$: an objective weight vector
- $\boldsymbol{\gamma}$: an aggregation indicator vector
- ξ: a defuzzification parameter
- $\boldsymbol{\theta}$: an OWA weight vector if using g^{OWA}

Step 1) Get reference point: Compute $\boldsymbol{z}^*$ from Equation 6.10

Step 2) Compute edge costs:

For each edge $e \in E(G)$, do

Step 2.1) Normalize: Get $\boldsymbol{F}'(e)$ from Equation 6.11.

Step 2.2) Apply exponential scaling:

Step 2.2.1) For all i where $\gamma_i = 1$, set $F_i'(e) = h\big(F_i'(e)\big)$

Step 2.2.2) Get $\boldsymbol{\lambda}'$ from Equation 6.15.

Step 2.3) Scalarize: Compute $S(e) = g(\boldsymbol{F}'(e)|\boldsymbol{\lambda}')$.

Step 2.4) Defuzzify: Set the edge cost as $C(S(e)|\xi)$.

Step 3) Find the shortest path: Use Dijkstra's algorithm to find the shortest path $p \in P(s,t)$ using the computed edge costs.

Output: A path p that approximates the ideal path that minimizes the scalarized MO-FLCPP objective function, $g(\boldsymbol{A}'(p)|\boldsymbol{\lambda})$.

The pre-scalarized decomposition method can be used to find solutions to the example problem from Section 6.2.3 with various decision-maker preferences. Table 6.2 shows the best paths found using this approach and compares them to those found using the post-aggregation scalarization method over all paths as in Section 6.2. The columns labeled p^{D} indicate which path has the lowest cost after pre-scalarizing each edge. This method uses max-edge normalization to compute $\boldsymbol{z}^*$ using Equation 6.10. The columns labeled p^{M} and

p^* show the paths that minimize the post-aggregation scalarized cost, where p^{M} uses max-edge normalization to compute $\boldsymbol{z}^*$ and p^* uses the known nadir vector. Comparing p^{D} with p^{M} shows how well exponential scaling applied to each edge can approximate the true cost of a path, whereas the difference between p^{M} and p^* shows the effect of choosing the appropriate reference point. If we assume that p^* represents the ideal path chosen with the best possible information (knowledge of the true Pareto front), then it is clear that the pre-scalarized decomposition approach does not always find the ideal path. We also note that the weighted sum scalarization approach only finds the red and purple paths in the ideal case (the endpoints of the Pareto front), whereas the Tchebycheff method is able to find other non-dominated paths.

Table 6.2 Best paths found in the example graph in Figure 6.3 using various methods. p^{D} is the path found using pre-scalarized decomposition. p^{M} and p^{*} are the best paths found using post-aggregation scalarization, where p^{M} uses max-edge normalization and p^{*} uses the nadir vector to normalize. Paths are notated with the first letter of their color.

		Weighted Sum			Tchebycheff		
ξ	$\boldsymbol{\lambda}$	p^{D}	p^{M}	p^{*}	p^{D}	p^{M}	p^{*}
0	(0, 1)	P	P	P	P	P	P
	(0.25, 0.75)	B	P	P	B	G	B
	(0.5, 0.5)	G	R	P	G	G	G
	(0.75, 0.25)	R	R	R	R	R	G
	(1, 0)	R	R	R	R	R	R
0.5	(0, 1)	P	P	P	P	P	P
	(0.25, 0.75)	P	P	P	P	G	P
	(0.5, 0.5)	P	R	P	P	R	P
	(0.75, 0.25)	P	R	R	P	R	R
	(1, 0)	R	R	R	R	R	R
1	(0, 1)	P	P	P	P	P	P
	(0.25, 0.75)	P	P	P	P	P	P
	(0.5, 0.5)	P	R	P	P	R	P
	(0.75, 0.25)	P	R	R	P	R	R
	(1, 0)	R	R	R	R	R	R

By scalarizing the multidimensional edge costs before aggregating all the costs in a path, the edges are evaluated in isolation, independent of their contributions to complete paths. If only the summation aggregation method is used and the scalarization function is the weighted sum, then the problem is linear and the cost of a full path can be represented exactly as the sum of the scalarized edge costs. However, in the non-linear case, some approximation is required. While the pre-scalarized decomposition approach can provide some initial candidate solution for a given objective weight vector, this solution can often be improved by conducting a more thorough search using the MOEA discussed in the next section.

6.4 MOEA/D for the MO-FLCPP

Each decomposition of the MO-FLCPP can only find a single solution based on the predefined parameters of the scalarization method. To gain a better understanding of the tradeoffs between the various objectives, we can search for an approximation of the Pareto optimal set of solutions. Multiobjective evolutionary algorithms (MOEAs) are well-suited for this task, since they maintain a population of solutions that can approximate the true Pareto optimal set. Rather than search for a single optimal solution, MOEAs use diversity preserving techniques to keep the population distributed along the Pareto front. In the CMM framework, we use decomposition as the primary way of maintaining population diversity using the MOEA/D algorithm (Qingfu Zhang and Hui Li 2007). This approach uses a set of many different objective weight vectors to create several single-objective problems that are optimized simultaneously.

The MOEA/D algorithm for the MO-FLCPP takes a set of N weight vectors $\boldsymbol{\lambda}^1, \dots, \boldsymbol{\lambda}^N$ and uses the pre-scalarized decomposition approach of the previous section to construct solutions to each subproblem. Each generation of MOEA/D for the MO-FLCPP maintains the following:

- a population of N solution paths $p^1, \dots, p^N$, where $p^i \in P(s,t)$ is the current solution to the ith subproblem;
- the aggregated cost values $\boldsymbol{A}(p^1), \dots, \boldsymbol{A}(p^N)$ of each path in the population;
- an external population (EP) that stores the most recent set of nondominated solutions;

- the reference point $\boldsymbol{z}^* = (z_1^*, \dots, z_m^*)$ defined by Equation 6.5 where EP is used as the approximation of the Pareto optimal set PS'.

An overview of the procedure is given in Algorithm 6.3.

Algorithm 6.3 MOEA/D for the MO-FLCPP

Input:
- MO-FLCPP
- a stopping criterion
- N: the number of subproblems considered
- $\boldsymbol{\lambda}^1, \dots, \boldsymbol{\lambda}^N$: m-dimensional objective weight vectors
- T: the size of each weight vector neighborhood
- g: a scalarization function (either g^{ws}, g^{te}, or g^{OWA})
- $\boldsymbol{\gamma}$: an aggregation indicator vector
- ξ: a defuzzification parameter
- $\boldsymbol{\theta}$: an OWA weight vector if using g^{OWA}

Output: EP

Step 1) Initialization:

Step 1.1) Compute the Manhattan distances (L_1 norm) between all pairs of objective weight vectors and determine each vector's T closest neighbors. For each $i = 1, \dots, N$, set $B(i) = \{i_1, \dots, i_T\}$, where $\boldsymbol{\lambda}^{i_1}, \dots, \boldsymbol{\lambda}^{i_T}$ are the T closest weight vectors to $\boldsymbol{\lambda}^i$, self-inclusive.

Step 1.2) Create the initial population of solutions $p^1, \dots, p^N$ using the decomposition method presented in Section 6.3. The reference point $\boldsymbol{z}^*$ is computed using Equation 6.10 and each path p^i for $i = 1, \dots, N$ is a solution to the scalarized SPP defined by the weight vector $\boldsymbol{\lambda}^i$.

Step 1.3) Define the initial external population EP as all nondominated paths in $p^1, \dots, p^N$.

Step 1.4) Update the reference point $\boldsymbol{z}^*$ using Equation 6.5 where EP is used as the approximation of the Pareto optimal set PS'.

(continued on next page)

(continued from previous page)

Step 2) Update:

For $i = 1, \ldots, N$, do

Step 2.1) Crossover: Randomly select two indices k and l from $B(i)$. Perform crossover on the paths p^k and p^l to produce a new path p'.

Step 2.2) Mutation: Randomly select a vertex v from the path p' and replace it with a valid substitute vertex v'.

Step 2.3) Improve: Remove any loops from path p'.

Step 2.4) Evaluate: Compute the aggregated cost $\boldsymbol{A}(p')$.

Step 2.5) Normalize: Use Equation 6.6 to compute $\boldsymbol{A}'(p')$ with $\boldsymbol{z}^*$.

Step 2.6) Compare:

For each neighboring weight index $j \in B(i)$, do

Step 2.6.1) Get the normalized cost vector $\boldsymbol{A}'\left(p^j\right)$ using $\boldsymbol{z}^*$

Step 2.6.2) Scalarize the cost vectors:
$S^j(p') = g\left(\boldsymbol{A}'(p')\middle|\boldsymbol{\lambda}^j\right)$ and $S^j\left(p^j\right) = g\left(\boldsymbol{A}'\left(p^j\right)\middle|\boldsymbol{\lambda}^j\right)$

Step 2.6.3) Defuzzify:
$f' = C\left(S^j(p')\middle|\xi\right)$ and $f^j = C\left(S^j\left(p^j\right)\middle|\xi\right)$

Step 2.6.4) If $f' \leq f^j$, then
set $p^j = p'$ and $\boldsymbol{A}\left(p^j\right) = \boldsymbol{A}(p')$.

Step 2.7) Update external population:

Step 2.7.1) Remove all paths from EP that p' dominates.

Step 2.7.2) Add p' to EP if no paths in EP dominate p'.

Step 2.7.3) Update the reference point $\boldsymbol{z}^*$.

Step 3) Stopping criteria: If the stopping criteria has been satisfied, then stop and output EP. Otherwise repeat Step 2.

The first step in MOEA/D is to construct the initial population of solutions. We begin by defining a set $\boldsymbol{\lambda}^1, \ldots, \boldsymbol{\lambda}^N$ of objective weight vectors. For problems with only a small number of objectives ($m \leq 3$), it is straightforward to produce a set of evenly spaced

vectors. However, as the number of objectives increases, it becomes increasingly desirable to limit the total number of vectors to some value N. For many-objective problems (MaOPs) where $m > 3$, we can uniformly sample N weight vectors from the unit simplex (Smith and Tromble 2004). It may be desirable to combine this approach with Mitchell's best-candidate sampling algorithm (Mitchell 1991) to ensure that the resulting vectors are well-distributed. When using random weight vectors, we often include the weight vectors at the corners of the unit simplex (i.e. all λ_i values set to zero except one) to help find the extrema points along the Pareto front. Each weight vector defines a different decomposition of the MOP as a single-objective problem.

In the initialization step, we first determine the neighborhood of each weight vector using the Manhattan distance metric. While the Euclidean distance can be used if there are only a few objectives, the Manhattan distance has better performance in high dimensional space (Aggarwal, Hinneburg, and Keim 2001). For each weight vector $\boldsymbol{\lambda}^i$, we define $B(i)$ as the indices of the T closest neighbors, including $\boldsymbol{\lambda}^i$, so that $i \in B(i)$.

The initial population is constructed by decomposing the MO-FLCPP into N subproblems corresponding to the N weight vectors and solving each scalarized SPP. The initial external population is determined from the nondominated paths and the reference point is updated to reflect the range of the Pareto front.

In each update step, we cycle over each weight vector index i and construct a new child solution p' from two of the neighbors in $B(i)$. Crossover on two paths p^k and p^l can be implemented by first selecting a vertex v that is common to both paths and is neither the starting vertex s nor the ending vertex t. If no such vertex exists, p' is set to either p^k

or p^l randomly. Otherwise, p' is defined as (s, p_1^k, v, p_2^l, t) where p_1^k is the first part of p^k from s to v and p_2^l is the second part of p^l from v to t. Mutation can be applied to paths with at least one vertex $v \notin \{s, t\}$. We select one of these vertices randomly and define the previous vertex as v_{-1} and the following vertex as v_{+1}. We then identify all vertices v' that could replace v for which there exists an edge (v_{-1}, v') and (v', v_{+1}). We pick one of these vertices randomly and set $v = v'$ (It may be the same vertex.) Figure 6.9 shows an example of path crossover and mutation.

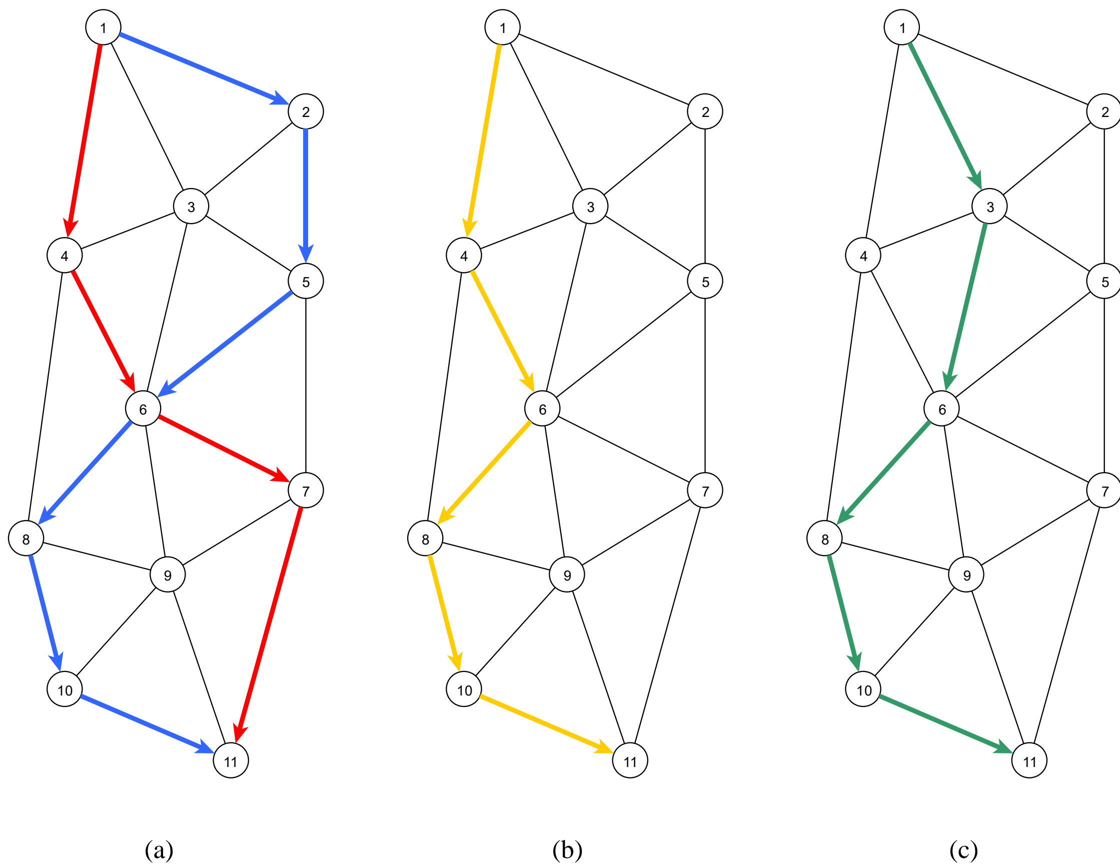


Figure 6.9 Example of crossover and mutation on paths. (a) There are two paths between vertices 1 and 11, p^k (red) and p^l (blue). (b) Crossover between p^k and p^l occurs at vertex 6 and results in a new path p' (yellow). (c) Mutation on p' changes vertex 4 to vertex 3.

After constructing a child path using crossover and mutation, we remove any loops in the path and compute the normalized cost $\boldsymbol{A}'(p')$ using the current value of $\boldsymbol{z}^*$. This cost vector is compared with the current best normalized solution cost $\boldsymbol{A}'(p^j)$ for each neighboring solution $j \in B(i)$. If the scalarized and defuzzified value of the new path is less than that of the current best path for index j when using weight vector $\boldsymbol{\lambda}^j$, then the new path is a better solution than the current one for index j. We replace any neighboring solutions that are outperformed with the new path p'.

Once the new path has been compared with all neighboring solutions, we update the external population. Any paths in EP that are dominated by p' are removed, and if no path in EP dominates p', it is added to the set.

Finally, after generating new solutions for each weight vector, we check to see if the stopping criteria has been met. If a maximum number of iterations has been reached or there is no clear improvement, then the algorithm stops and EP is returned. The decision-maker can evaluate the quality of EP directly, or choose the solution that minimizes some predefined preferred scalarization method.

6.5 Experiments

To evaluate the MOEA/D algorithm for the MO-FLCPP, we create several different test scenarios using the CMM framework. First, we consider problems with only two objectives since they are easier to interpret and visualize. We consider different region clustering approaches and scalarization methods, finding a set of Pareto optimal solutions for each configuration in a binary terrain environment. We then investigate both summation and maximization elevation features in a hilly environment, and show a pair of examples

using the terrain transition features. Finally, we show some of the challenges of assessing the results of problems with many objectives. We conclude these experiments with a series of tests designed to show improvement by the MOEA/D algorithm over the pre-scalarized decomposition approach.

6.5.1 Two Objective Shortest Paths in Binary Terrain Environments

The first scenario we consider is a two objective shortest path problem in a flat environment with two terrain types. The two features to minimize are the terrain distance features $\tilde{f}_{t(1)}$ and $\tilde{f}_{t(2)}$. Both features are aggregated over the solution paths using summation. Figure 6.10 shows the solutions found by MOEA/D using Algorithm 6.3 with the weighted sum scalarization method. A different region clustering method is used in each of the three subfigures. For each problem instance, 20 initial reference weight vectors are uniformly sampled, including the “one-hot” vectors $(1, 0)$ and $(0, 1)$. The algorithm proceeds for 100 iterations before returning the external population containing the Pareto optimal solutions.

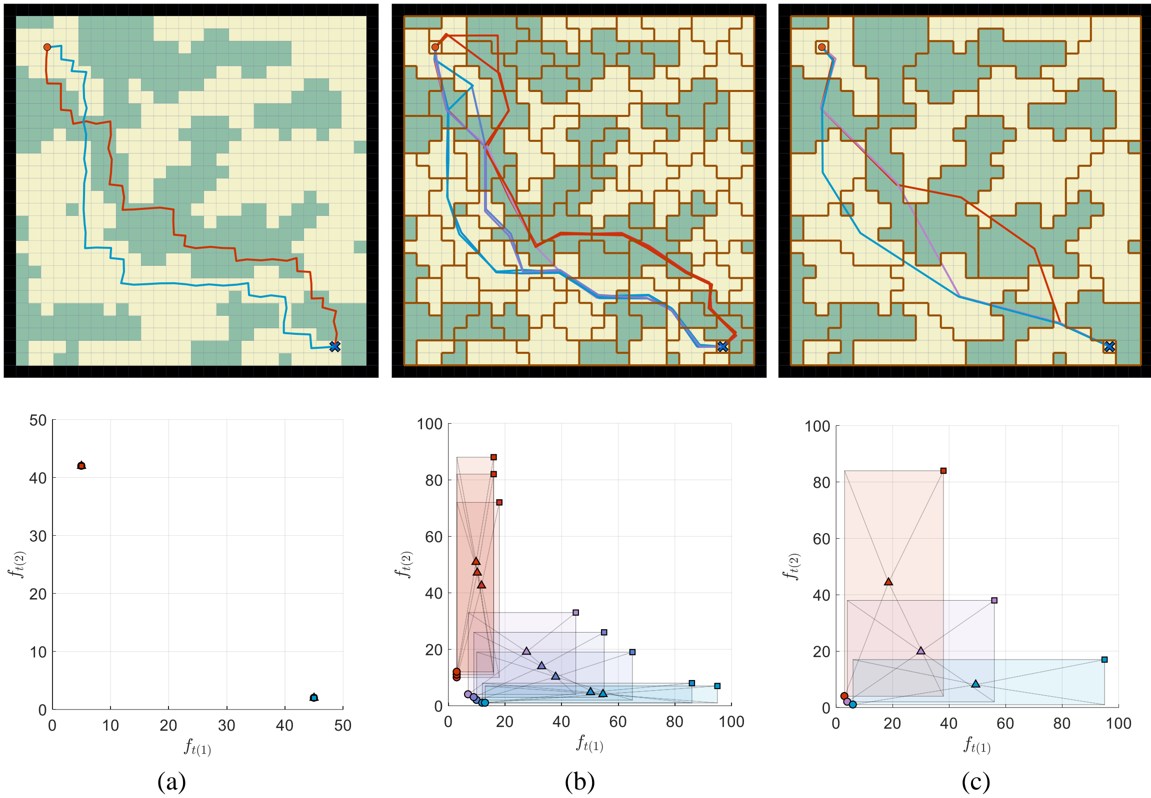


Figure 6.10 Shortest paths found by the MOEA/D algorithm for the MO-FLCPP in a binary terrain environment using the weighted sum scalarization method to minimize $\tilde{f}_{t(1)}$ and $\tilde{f}_{t(2)}$. (a) No region partitioning. (b) Region cluster size = 3. (c) Region cluster size = 10.

Subfigure (a) shows the results found when no region clustering is used. (This can be obtained by setting *opt.lrMethod* = “all” in Algorithm 5.2.) Since the environment is fully observable in this case, there is no uncertainty and the solution costs are crisp values. The weighted sum scalarization method finds the extrema Pareto optimal solutions that minimize each of the objectives. The red path stays in the forest and the blue path stays in the meadow terrain as much as possible. In this scenario, there is no solution that only stays in one terrain type, so each solution travels some amount through both terrain types. As discussed in Section 6.1, the edge costs are modified by a very small random value (on the

order of 10^{-14}) to mitigate the selection bias problem and select a single path out of the many equivalent cost solutions.

Subfigure (b) shows the results found using a region clustering size of 3, set with *opt.regionSize* in Algorithm 5.3. There are several nondominated paths that are found, drawn with colors ranging from red to blue. As with the previous subfigure, paths that favor the forest terrain are drawn in red and paths that favor the meadow are drawn in blue, with several compromise paths drawn in purple. Paths are drawn through the environment between the centers of each region, so there may be some regions that appear to be crossed that are not actually part of the path. The fuzzy cost of each solution in the objective space is shown in a plot below the environment figure. The triangular fuzzy numbers of the two cost values form a box, with the minimum value marked as a circle, the maximum value marked as a square, and the peak (mean value) marked as a triangle. Recall that only one of these points needs to be nondominated among the same type of points (min, mean, or max) for the entire solution to be considered nondominated. These objective space plots are analogous to a top-down view of the example in Figure 6.5.

Subfigure (c) shows the results found when the region cluster size is set to 10. This results in larger regions and an overall smaller search space. Only three nondominated solutions are found, but they provide a reasonable summary of the paths found using the smaller region size. Note that we use these examples only to demonstrate the differences between the various path planning options. To follow one of these paths, the agent would need to define a preference weight vector and select the path that minimizes the scalarized cost. A local region would also need to be defined. In subfigures (b) and (c), it is not clear

in which direction the agent should move to follow one of the paths. Using a local region ensures that the next immediate step in the path is one that the agent can actually take.

Figure 6.11 shows the same scenario setup solved using Algorithm 6.3 with the Tchebycheff scalarization method. As expected from the discussion in Section 2.5.5, this method returns many more solutions along the Pareto front. In subfigure (a), all the paths have the same total crisp length since there is no region partitioning, and nearly every possible combination of forest and meadow terrain is represented. Subfigures (b) and (c) have additional paths compared to Figure 6.10, but they generally follow the same route. Most differences arise from moving between the two terrain types at different locations.

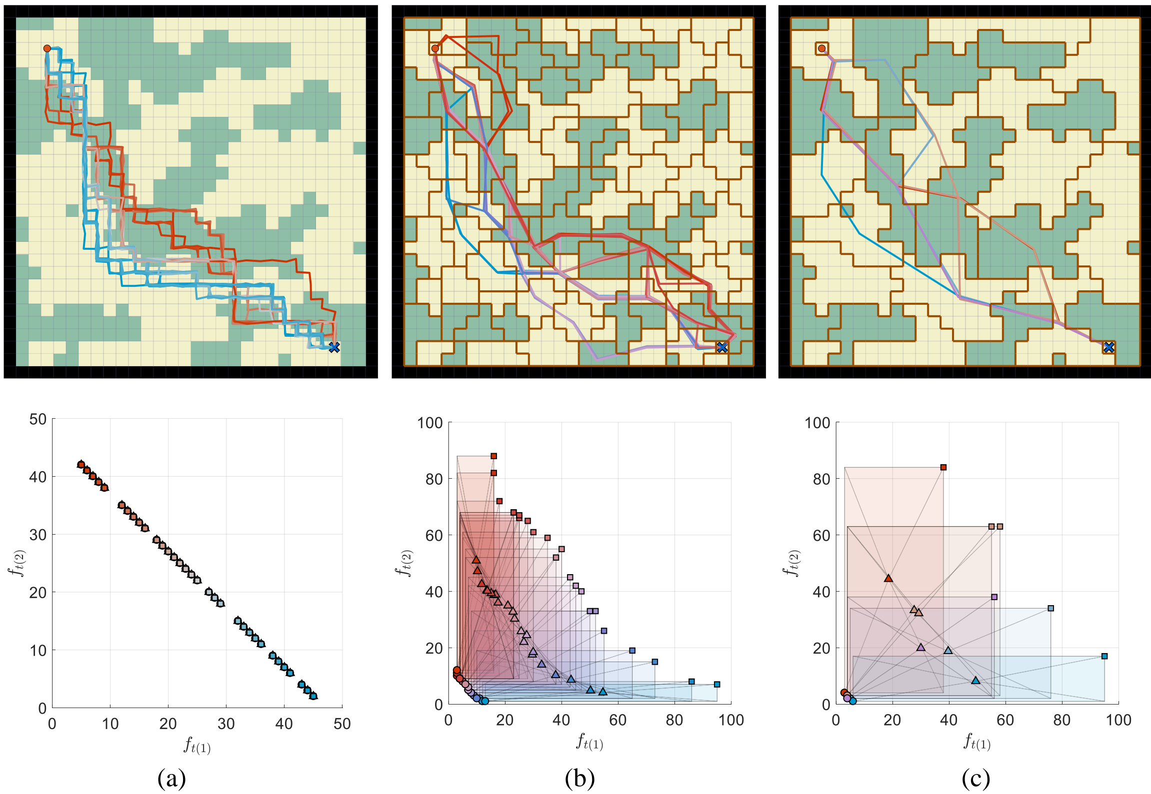


Figure 6.11 Shortest paths found by the MOEA/D algorithm for the MO-FLCPP in a binary terrain environment using the Tchebycheff scalarization method to minimize $\tilde{f}_{t(1)}$ and $\tilde{f}_{t(2)}$. (a) No region partitioning. (b) Region cluster size = 3. (c) Region cluster size = 10.

As a balance between the weighted sum (WS) and Tchebycheff (TE) scalarization methods, we consider the same problem scenario using an ordered weighted average (OWA) operator. In the two objective case, the weighted sum and Tchebycheff methods are equivalent to OWA weight vectors of $\boldsymbol{\theta}^{WS} = (0.5, 0.5)$ and $\boldsymbol{\theta}^{TE} = (1, 0)$ respectively. We consider a hybrid of these two weight vectors and set $\boldsymbol{\theta} = (0.67, 0.33)$, giving twice as much weight to the least satisfied objective as the most satisfied one. Figure 6.12 shows the resulting paths found using Algorithm 6.3. The number of solutions for the first two configurations is greater than the WS method, but less than the TE method, and the solutions for the last configuration is the same as the WS method. This shows that the OWA scalarization method does act as a hybrid operator with this parameterization and finds a balance of solutions between those found by the weighted sum and Tchebycheff methods.

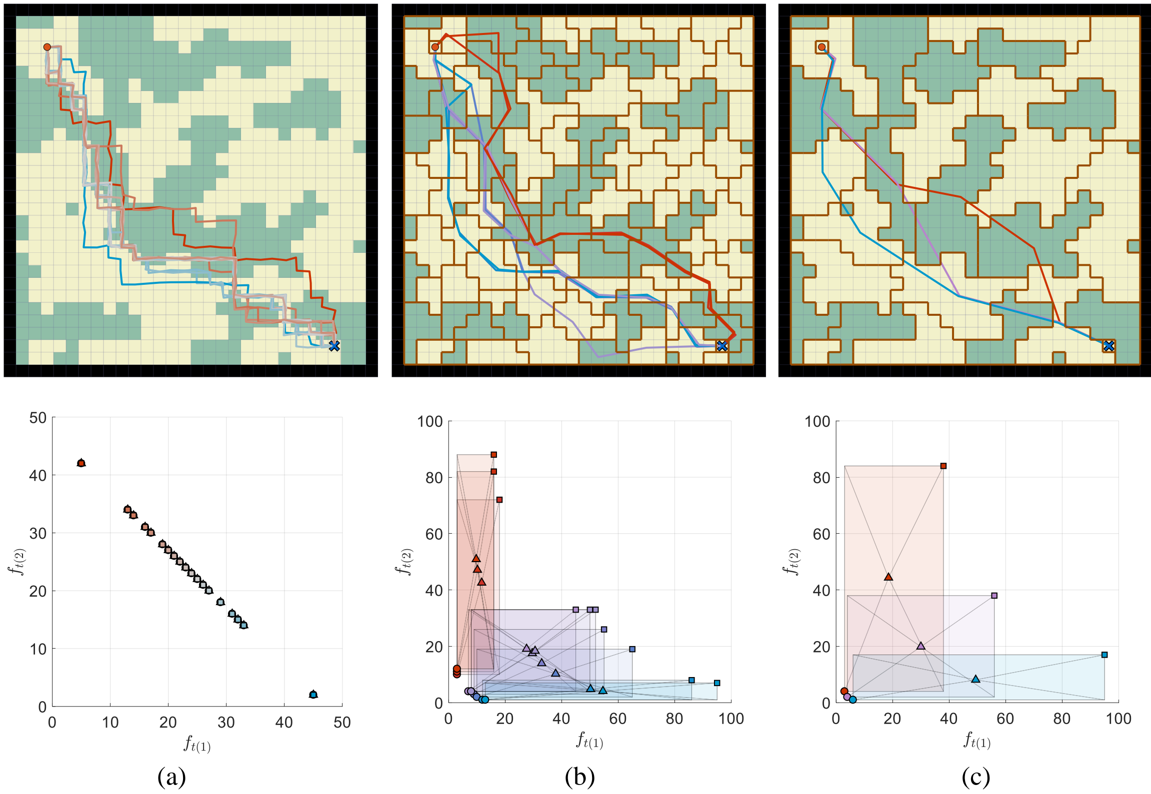


Figure 6.12 Shortest paths found by the MOEA/D algorithm for the MO-FLCPP in a binary terrain environment using the ordered weighted average (OWA) scalarization method with weight vector $\boldsymbol{\theta} = (0.67, 0.33)$ to minimize $\tilde{f}_{t(1)}$ and $\tilde{f}_{t(2)}$. (a) No region partitioning. (b) Region cluster size = 3. (c) Region cluster size = 10.

6.5.2 Two Objective Least-Cost Paths Using Elevation

The second scenario we consider is a two objective least-cost path problem in a hilly environment with only one terrain type. The two features to minimize are the total distance $\tilde{f}_d$ and the maximum absolute value of the elevation change $\tilde{f}_{h_\max}$. Figure 6.13 shows the solutions found in an example environment by MOEA/D using Algorithm 6.3 with the OWA scalarization method with $\boldsymbol{\theta} = (0.67, 0.33)$. The red path shows a route that minimizes the maximum slope in the shortest distance possible. The blue path shows

a route that prioritizes a gentle slope over distance and contains switchbacks to minimize the elevation change. The purple path shows a route that is a compromise between these two extremes.

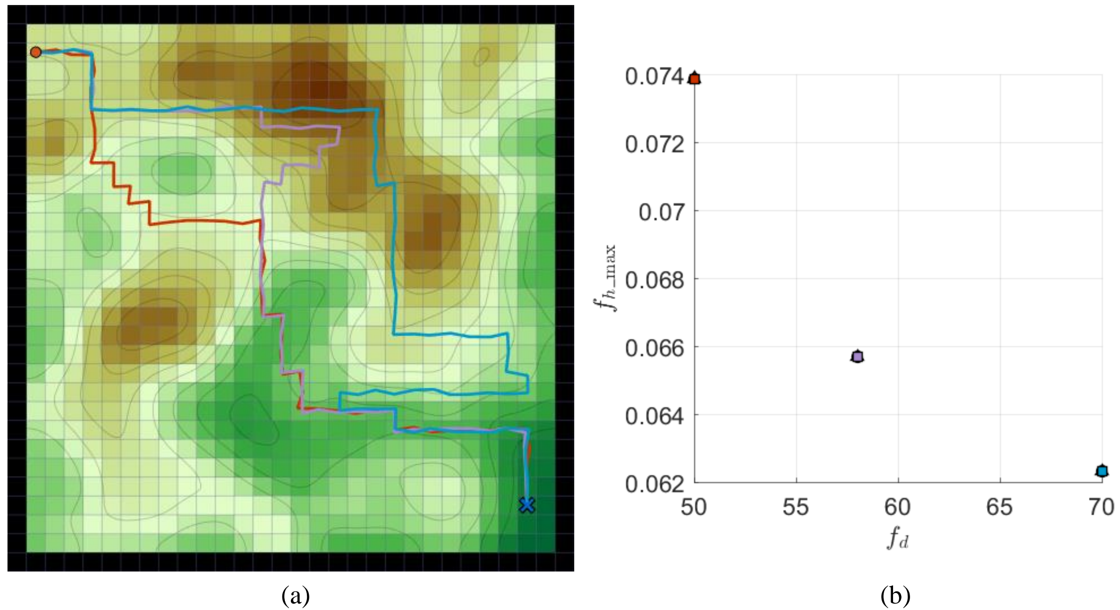


Figure 6.13 Least-cost paths found by the MOEA/D algorithm for the MO-FLCPP in a hilly environment using the ordered weighted average (OWA) scalarization method with weight vector $\boldsymbol{\theta} = (0.67, 0.33)$ to minimize $\tilde{f}_d$ and $\tilde{f}_{h_max}$. (a) Pareto optimal paths found from the starting agent location to the goal. (b) Solution costs plotted in objective space.

Figure 6.14 shows a scenario with the same objectives, but with multiple goal locations. The agent is placed at a middle elevation and the goals are placed at minimum and maximum extrema locations. This can be implemented in the CMM framework as a TSP-type problem in which the agent only needs to collect one resource. In subfigure (a), each of the four resources are selected by different paths, with bluer paths favoring slope over distance and redder paths favoring distance over slope. Subfigure (b) shows the same scenario with a region size of 3. The solutions are similar to those found without region clustering, but the path costs in objective space show much greater uncertainty with lots of

overlap between the minimum and maximum values of each Pareto optimal solution. Subfigure (c) uses a region size of 10, which simplifies the resulting region graph and results in a summary of fewer solutions. However, the difference between solutions is less clear as the amount of uncertainty has increased.

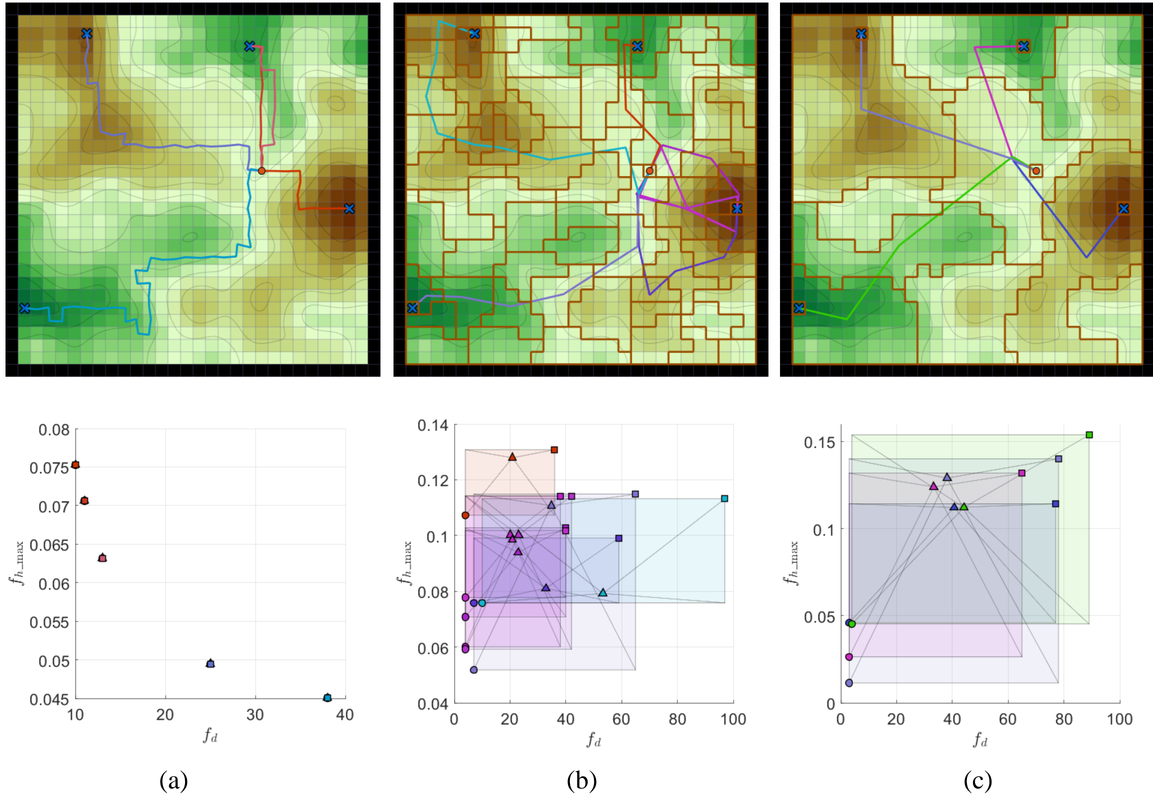


Figure 6.14 Least-cost paths to the nearest goal found by the MOEA/D algorithm for the MO-FLCPP in a hilly environment using the ordered weighted average (OWA) scalarization method with weight vector $\boldsymbol{\theta} = (0.67, 0.33)$ to minimize $\tilde{f}_d$ and $\tilde{f}_{h_max}$. (a) No region partitioning. (b) Region cluster size = 3. (c) Region cluster size = 10.

In Figure 6.15, we consider an example that minimizes different pairs of elevation features. In subfigure (a), the maximum uphill and downhill elevation features are used. There are a total of four Pareto optimal solutions found by MOEA/D for this example. The red paths prioritize minimizing the uphill slope and find routes that mainly go downhill,

whereas the blue paths prioritize minimizing the downhill slope and find routes that mainly go uphill. In subfigure (b), the total uphill and downhill elevation features are used. Here, any path that has the same starting and ending point and goes either entirely uphill or downhill will have the same feature value. Because of this, it is possible to have many unique paths that all evaluate to the same point in objective space. We keep only one representative path from any set of paths with the same feature value. This is accomplished by rounding feature values to a very fine grid (on the order of 10^{-12}) when computing dominance to avoid small errors in numerical precision. The result is that only two Pareto optimal paths are found for this example when using the uphill and downhill summation elevation features, one going up and the other down. In subfigure (c), we use the maximum and total absolute elevation difference features. This results in solutions that either minimize the slope or the total elevation change. The red path in this example shows a longer downhill route with a gentle slope, and the blue and purple paths show shorter uphill routes that have steeper slopes. Some of the routes are found using several different elevation feature objectives, suggesting that an agent does not need to use all of these features simultaneously.

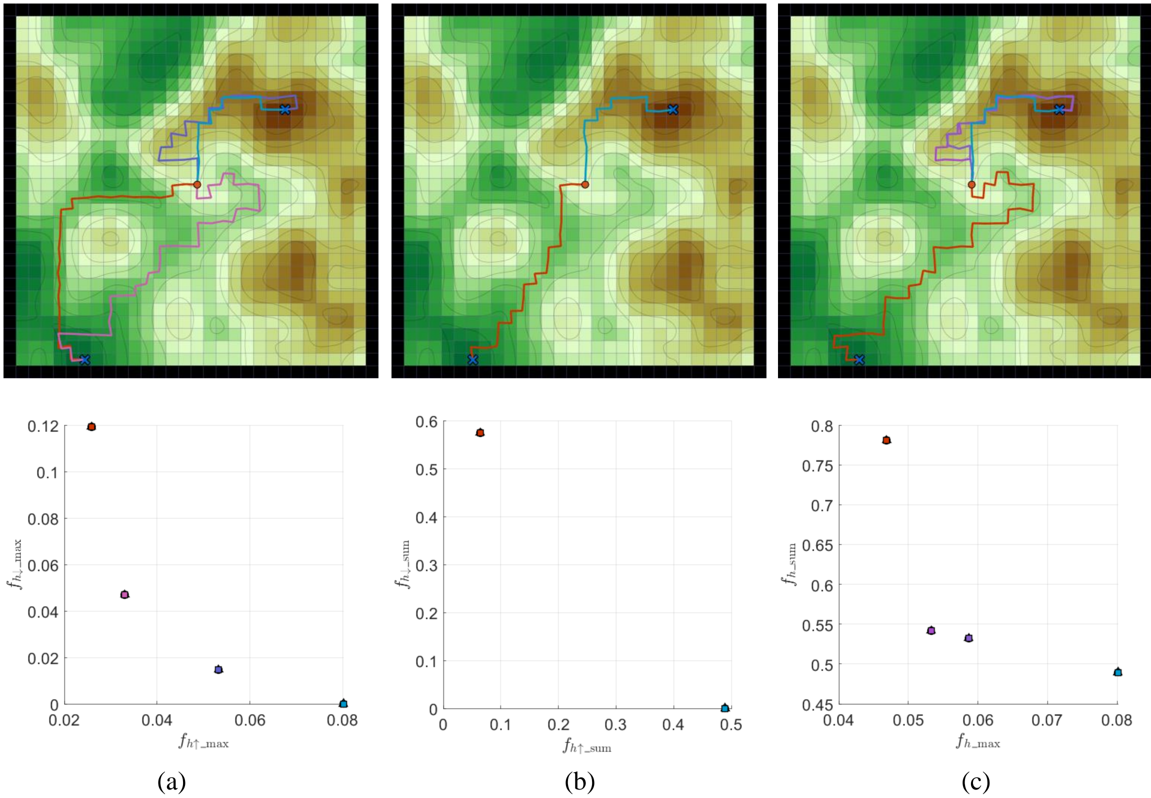


Figure 6.15 Least-cost paths to the nearest goal found by the MOEA/D algorithm for the MO-FLCPP in a hilly environment with no region clustering using the ordered weighted average (OWA) scalarization method with weight vector $\boldsymbol{\theta} = (0.67, 0.33)$. (a) Minimizing features $f_{h\uparrow_\text{max}}$ and $f_{h\downarrow_\text{max}}$. (b) Minimizing features $f_{h\uparrow_\text{sum}}$ and $f_{h\downarrow_\text{sum}}$. (c) Minimizing features f_{h_max} and f_{h_sum}.

6.5.3 Shortest Paths Using Terrain Transition Features

The terrain transition features offer a way for the agent to indicate additional specific preferences for finding paths through multiple types of terrain. Using these features quickly increases the number of objectives and can make it difficult to visualize the entire Pareto optimal set of solutions. To demonstrate the effect of both the symmetric and directional terrain transition features, Figure 6.16 shows two example scenarios that are solved using specific agent preferences. The preferences are encoded by the objective

weight vector $\boldsymbol{\lambda}$, which we arrange as a matrix $\boldsymbol{\lambda}^{\mathbf{T}}$, where $\boldsymbol{\lambda}_{ij}^{\mathbf{T}}$ is the weight of the terrain transition feature from terrain type i to terrain type j. A greater weight indicates a higher cost when using this type of transition.

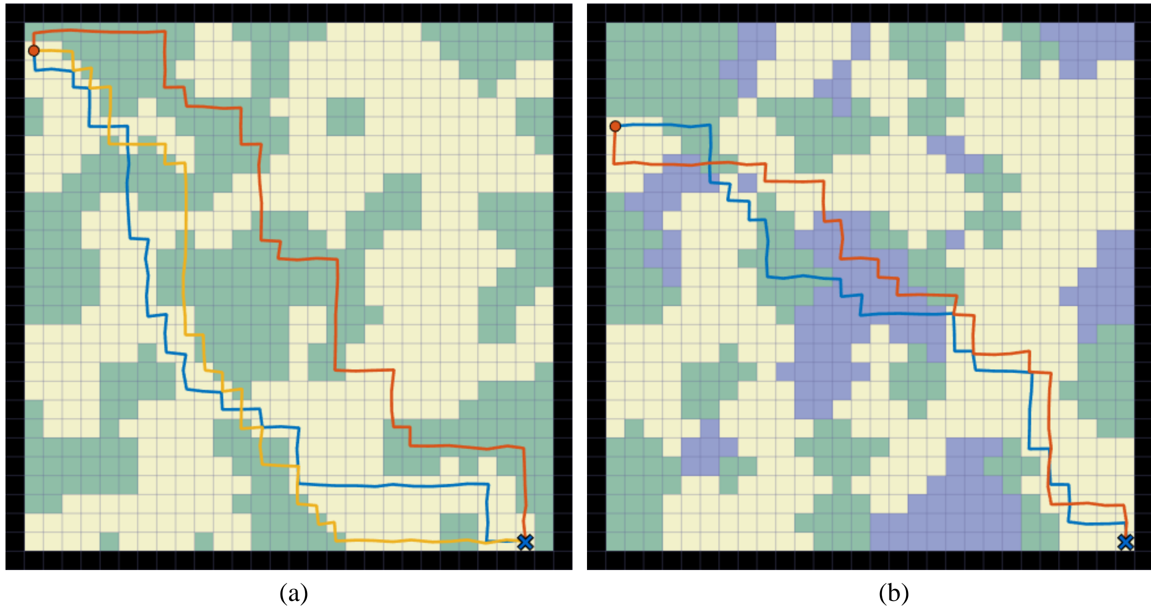


Figure 6.16 Shortest paths found using terrain transition features. (a) Three agents solve this problem using symmetric terrain transition features. The blue agent prefers the meadow, the red agent prefers the forest, and the yellow agent prefers the transitions between the two terrain types. (b) Two agents solve this problem using directional terrain transition features. The red agent prefers the transitions from meadow to water and from water to forest, whereas the blue agent prefers the transitions from forest to water and from water to meadow.

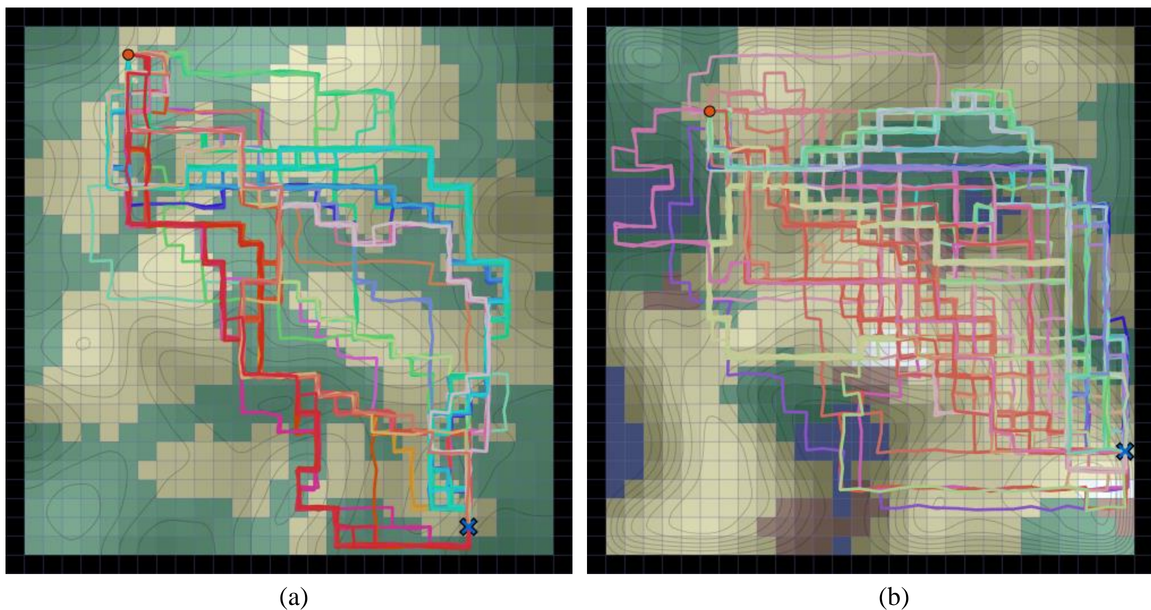
In subfigure (a), a shortest path problem in a binary terrain environment is solved by three different agents. The blue agent uses an objective weight matrix of $\begin{bmatrix} 0.1 & 0.6 \\ 0.6 & 0.3 \end{bmatrix}$, indicating a low cost of 0.1 for staying in the meadow and a slightly higher cost of 0.3 for staying in the forest. The cost of the transition between the two types is symmetric and has a higher cost value of 0.6. (Note that only the upper triangular part of the matrix is used and these three costs sum to one.) The resulting path chosen by the blue agent stays in the

meadow as much as possible and avoids the forest. The red agent uses a reverse weight matrix of $\begin{bmatrix} 0.3 & 0.6 \\ 0.6 & 0.1 \end{bmatrix}$, which favors the forest over the meadow. Both agents use the fewest number of transitions between terrain types as possible. The yellow agent uses an objective weight matrix of $\begin{bmatrix} 0.4 & 0.2 \\ 0.2 & 0.4 \end{bmatrix}$, which gives a lower cost to transitioning between terrain types than staying in one type of terrain. The path chosen by the yellow agent stays on the border between terrain types as much as possible while still choosing one of the shortest paths.

To demonstrate the effect of the directional terrain transition features, at least three terrain types are required. Subfigure (b) shows a shortest path problem in a trinary terrain environment solved by two different agents using the directional terrain transition features. The blue agent uses an objective weight matrix of $\begin{bmatrix} 1 & 2 & 9 \\ 9 & 1 & 2 \\ 2 & 9 & 1 \end{bmatrix} / 36$, where all nine features are used. This matrix gives a low cost to staying in the same type of terrain, but different higher costs for terrain transitions. A high cost is given to transitions from type 1 to 3 (meadow to water), type 2 to 1 (forest to meadow), and type 3 to 2 (water to forest). The resulting path chosen by the blue agent never uses these terrain transition types. The red agent uses a reversed objective weight matrix of $\begin{bmatrix} 1 & 9 & 2 \\ 2 & 1 & 9 \\ 9 & 2 & 1 \end{bmatrix} / 36$. This agent uses only the above terrain transition types. These features give even greater control over the types of paths produced and offer a way to explore problems with many objectives.

6.5.4 Many-Objective Least-Cost Paths

When there are many objectives to optimize, the number of Pareto optimal solutions can grow very large. This is especially true when region clustering is used to add uncertainty to the feature values. We consider two additional types of environments, designed to provide a high number of features for many-objective problems. Figure 6.17 shows two example least-cost path problems in environments that combine multiple terrain types and elevation.



(a) (b)

Figure 6.17 Pareto optimal least-cost paths found by the MOEA/D algorithm for the MO-FLCPP in many-objective environments. Solutions are colored based on objective weight similarity. (a) A hilly binary terrain environment. (b) A full world environment with five terrain types and elevation.

Subfigure (a) shows a hilly binary terrain environment that is solved by an agent using three symmetric terrain transition features and the maximum absolute elevation difference feature using the weighted sum scalarization method. Using 40 uniformly sampled reference weight vectors for the MOEA/D algorithm, including the one-hot

vectors, a total of 71 Pareto optimal paths were found. These are shown in different colors based on objective weight similarity. To compute the colors, the objective values are projected into two dimensions using PCA and assigned hue and saturation values based on the 2D coordinates. In this example, red paths favor the forest terrain and green paths favor the meadow. A spectrum of other solutions shows the variety of different agent preferences that can lead to unique paths.

Subfigure (b) shows a full world environment with five terrain types and elevation. In this example, the agent uses 15 symmetric terrain transition features and the maximum absolute elevation difference feature for a total of 16 objectives. The weighted sum scalarization method is used for the MOEA/D algorithm with 160 reference weight vectors (10 times the number of objectives), including the one-hot vectors. A total of 176 Pareto optimal paths were found for this problem, spanning all different agent preferences. The solutions are colored based on similarity, although it can be difficult to distinguish what exactly each path is optimizing based on color alone. Here, the red paths seem to prefer paths through open meadow, and blue paths prefer forest and water. The pink paths in the upper left show some switchbacks that indicate a preference for gentle slopes. Although any one agent can specify a particular set of objective weight preferences and choose one of these solutions, the quality of the entire solution set is difficult to quantify. The next section discusses a set of experiments to measure the effectiveness of the MOEA/D algorithm for the MO-FLCPP in various configurations.

6.5.5 Comparing MOEA/D to Pre-scalarized Decomposition

The MOEA/D algorithm for the MO-FLCPP presented in Algorithm 6.3 uses an iterative evolutionary procedure to find Pareto optimal solutions over the entire set of possible decision-maker preferences. For a specific set of preferences, the pre-scalarized decomposition method presented in Algorithm 6.2 can be used to quickly find a single solution that may be satisfactory in some cases. In this section, we evaluate many different problem scenarios to determine how much of an improvement in solution quality the MOEA/D approach offers over the pre-scalarized decomposition method.

We chose 10 problem configurations to evaluate, which are summarized in Table 6.3. Each problem configuration represents an environment type and a specified feature set. Figure 6.18 shows the five different environment types used in these experiments. The first type (a) is a flat binary terrain environment used to study problems with two objectives. The next type (b) is a hilly environment with a single type of terrain used to study elevation features. The third type (c) is a flat trinary terrain environment used to study the directional terrain transition features. The fourth type (d) is a hilly binary terrain environment that combines the terrain type and elevation features. The last type (e) is a full world environment that has five terrain types and elevation, which is used to study problems with many objectives.

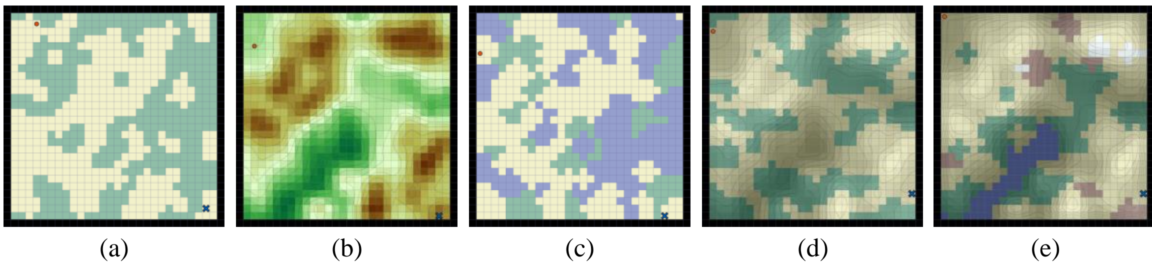


Figure 6.18 Environment types used to evaluate the MOEA/D algorithm for the MO-FLCPP. (a) Flat binary terrain. (b) Hilly uniform terrain. (c) Flat trinary terrain. (d) Hilly binary terrain. (e) Full world environment.

The environment type of each problem is listed in the second column of Table 6.3. The third column lists the total number of objectives used, and the fourth and fifth columns indicate how many of these are summation objectives and how many are maximization objectives. The sixth column lists how many reference vectors (N) are used in the MOEA/D algorithm. This is set to be 10 times the total number of objectives for each problem. The last column gives the features that are used for each problem type.

Table 6.3 Summary of problem types used to compare MOEA/D to pre-scalarized decomposition

Prob. #	Env. Type	Total # of Obj.	# of Sum Obj.	# of Max Obj.	N	Features
1	a	2	2	0	20	$\tilde{f}_{t(1)}, \tilde{f}_{t(2)}$
2	b	2	1	1	20	$\tilde{f}_{d}$, $\tilde{f}_{h_\max}$
3	b	2	0	2	20	$\tilde{f}_{h\uparrow_\max}, \tilde{f}_{h\downarrow_\max}$
4	b	3	3	0	30	$\tilde{f}_{d}$, $\tilde{f}_{h\uparrow_\text{sum}}, \tilde{f}_{h\downarrow_\text{sum}}$
5	d	3	2	1	30	$\tilde{f}_{t(1)}, \tilde{f}_{t(2)}$, $\tilde{f}_{h_\max}$
6	e	6	5	1	60	$\tilde{f}_{t(i)}\ \forall\ i \in \{1, \dots, 5\}$, $\tilde{f}_{h_\max}$
7	c	6	6	0	60	$\tilde{f}_{t\{i,j\}}\ \forall\ i, j \in \{1, \dots, 3\}\ s.t.\, i \leq j$
8	e	15	15	0	150	$\tilde{f}_{t\{i,j\}}\ \forall\ i, j \in \{1, \dots, 5\}\ s.t.\, i \leq j$
9	e	17	15	2	170	$\tilde{f}_{t\{i,j\}}\ \forall\ i, j \in \{1, \dots, 5\}\ s.t.\, i \leq j$, $\tilde{f}_{h\uparrow_\max}, \tilde{f}_{h\downarrow_\max}$
10	e	29	26	3	290	$\tilde{f}_{t\langle i,j\rangle}\ \forall\ i, j \in \{1, \dots, 5\}$, $\tilde{f}_{h\uparrow_\max}, \tilde{f}_{h\downarrow_\max}$, $\tilde{f}_{h\uparrow_\text{sum}}, \tilde{f}_{h\downarrow_\text{sum}}$

The first three problem types are two objective problems. Problem 1 uses the terrain type features in binary terrain as two summation objectives. Problems 2 and 3 use the hilly uniform terrain environment to investigate maximization objectives. Problem 2 uses one summation objective, $\tilde{f}_d$, and one maximization objective, $\tilde{f}_{h_\max}$. Problem 3 uses the two directional elevation features $\tilde{f}_{h\uparrow_\max}$ and $\tilde{f}_{h\downarrow_\max}$ as maximization objectives. Problems 4 and 5 use different combinations of the distance, terrain type, and elevation features to investigate three objective problems. Problems 6 and 7 consider problems with six objectives in either the trinary terrain for full world environments. Problems 8-10 use

terrain transition and directional elevation features to study problems with many objectives in full world environments.

For each problem configuration, we generate 30 environments of the specified type with an agent and a single resource in opposite corners. Each environment is used to define a MO-FLCPP using either no region clustering, or a cluster size of 3 or 10. These problems are then evaluated by nine different sets of decision-maker parameters. We consider defuzzification values of 0, 0.5, and 1. For scalarization, we consider the weighted sum (WS), Tchebycheff (TE), and ordered weighted average (OWA) methods. The OWA method acts as a hybrid between the WS and TE methods, and we define the OWA weights as

$$\boldsymbol{\theta} = \left(\frac{1}{H_m}, \frac{1}{2H_m}, \dots, \frac{1}{mH_m}\right), \tag{6.16}$$

where $H_m = \sum_{k=1}^{m} 1/k$ is the m-th harmonic number and m is the total number of objectives. This gives a harmonic sequence that assigns greater weight to the first (largest) terms and is roughly equivalent to a decision-maker that prioritizes only a few objectives.

Next, we randomly sample a set of N weight vectors for each problem, where $N = 10m$. These include the one-hot weight vectors where all values are zero except one. For each weight vector, apply the pre-scalarized decomposition method in Algorithm 6.2 to create an initial candidate solution. This solution is scored using the max-edge reference point where $\boldsymbol{z}^*$ is defined by Equation 6.10.

The N solutions are then used as the initial population for MOEA/D in Algorithm 6.3. We use a neighborhood size of $T = 5$ for each weight vector and run the algorithm for 100 iterations. The resulting EP contains a set of Pareto optimal solutions and lets us

compute a new reference point where $\boldsymbol{z}^*$ is defined by Equation 6.5 using the nadir vector. Each of the original weight vectors $\boldsymbol{\lambda}^j$ where $j = 1, \dots, N$ is then used to find two solutions to the MO-FLCPP:

1) The path $p^* \in EP$ that minimizes $g\left(\boldsymbol{A}'(p^*)\middle|\boldsymbol{\lambda}^j\right)$ using the nadir vector reference point $\boldsymbol{z}^*$.
2) The path $p^{PSD} \in P(s,t)$ found using pre-scalarized decomposition and the same reference point $\boldsymbol{z}^*$.

The scalarized values of the two paths are defined as S^* and S^{PSD} respectively, and since both solutions use the same value for $\boldsymbol{z}^*$, the values can be compared directly. We measure the percent improvement of the MOEA/D algorithm over the pre-scalarized approach as

$$PI = \left(\frac{S^{PSD} - S^*}{S^{PSD}}\right) * 100. \qquad (6.17)$$

Table 6.4 shows the average PI values of each MO-FLCPP configuration, for each decision-maker, averaged over all weight vectors from the 30 problem instances of all 10 problem types when using a region clustering size of 3. The first column gives the problem number and the second and third columns give the number of summation and maximization objectives respectively. The fourth and fifth columns list the average number of nodes and edges over the 30 graphs created for each problem type. The results show an overall improvement in the scalarization scores of the MOEA/D algorithm over the pre-scalarized decomposition method. In general, we note the largest improvement for problems that contain at least one maximization objective. The Tchebycheff scalarization method also tends to show the greatest improvement compared to the weighted sum. The OWA method

usually scores somewhere between the other two approaches, reflecting its role as a hybrid operator.

Table 6.4 Average percent improvement of MOEA/D over pre-scalarization (region cluster size = 3)

Prob. #	# of Sum Obj.	# of Max Obj.	Avg. Nodes	Avg. Edges	$\xi = 0$			$\xi = 0.5$			$\xi = 1$		
					WS	OWA	TE	WS	OWA	TE	WS	OWA	TE
1	2	0	103	507	0.00	1.39	8.49	0.00	1.67	9.30	0.00	1.78	9.94
2	1	1	66	318	5.35	7.10	13.12	5.27	5.08	5.44	5.57	5.75	7.02
3	0	2	64	300	12.43	12.01	12.94	5.71	5.87	6.55	0.93	0.79	1.17
4	3	0	65	311	-0.02	0.83	5.03	0.00	0.14	0.67	0.00	0.09	0.56
5	2	1	92	447	9.64	13.73	20.53	7.93	7.08	8.51	9.80	8.55	9.27
6	5	1	93	454	17.02	21.75	30.27	5.32	5.93	11.00	5.46	5.17	9.91
7	6	0	109	545	-0.47	2.32	8.87	-0.15	2.39	9.82	-0.15	3.06	12.35
8	15	0	93	454	-0.08	1.64	8.02	-0.03	1.53	8.00	-0.04	2.02	11.73
9	15	2	91	445	26.31	31.75	42.06	7.14	9.14	16.35	4.36	5.38	12.22
10	26	3	93	450	15.68	21.66	29.27	6.34	8.01	13.46	4.55	5.85	11.46

We can conclude that problems that have many nonlinearities, either from the aggregation method or the scalarization function, benefit the most from performing a search with the MOEA/D algorithm. In contrast, when the problem contains only summation objectives, or the weighted sum is used to scalarized, there is less reason to use MOEA/D. If both are true, then the problem is entirely linear and the pre-scalarized decomposition method can find the ideal solution directly. In this case, the MOEA/D algorithm can perform worse if the reference point changes from the initial solutions and the ideal path is never encountered through the crossover and mutation operators. This is the primary cause for the negative improvement scores in the table. One way this effect could be ameliorated is by re-running the initialization procedure each time the reference point changes to migrate the population toward solutions that are evaluated better with the new reference point.

Table 6.5 shows the average PI values when using no region clustering and Table 6.6 shows the values when using a region cluster size of 10. The outcomes are mostly the same as the results using a region cluster size of 3 with a few notable exceptions. With no region clustering, all graphs are the same size and there is no uncertainty in these fully-observable problems, so all feature values are crisp. The results are therefore the same for each value of the defuzzification parameter ξ. Problems 3 and 4 have very low improvement scores across all scalarization methods. This would suggest that the pre-scalarized approach produces very good results using crisp distance and elevation features when all objectives use either summation or maximization aggregation. Problems 7 and 8 show both worse scores for the WS method and better scores for the OWA and TE methods than when using a region cluster size of 3. These problems use only the terrain transition features with summation aggregation, which suggests that the MOEA/D method is most effective for problems with nonlinear operators. The improvement is more noticeable in problems with less uncertainty, which is confirmed by the corresponding rows in Table 6.6 that show less difference between scalarization methods when using a region cluster size of 10. This effect is observed throughout Table 6.6, where the improvement scores for each scalarization method tend to be more similar than in the other two tables. We also note a few instances in the later problems of Table 6.6 where the OWA method shows less improvement than the other two approaches, rather than behaving as a hybrid operator that gives a result somewhere in-between. The additional parameters of the OWA method, the added complexity of the many-objective problems, and the increased uncertainty from a larger region size could render the MOEA/D search less effective than the pre-scalarized approach in these instances.

Table 6.5 Average percent improvement of MOEA/D over pre-scalarization (no region clustering)

Prob. #	# of Sum Obj.	# of Max Obj.	Avg. Nodes	Avg. Edges	$\xi = 0$			$\xi = 0.5$			$\xi = 1$		
					WS	OWA	TE	WS	OWA	TE	WS	OWA	TE
1	2	0	784	3024	0.00	1.69	12.00	0.00	1.69	12.00	0.00	1.69	12.00
2	1	1	784	3024	9.62	11.53	14.37	9.62	11.53	14.37	9.62	11.53	14.37
3	0	2	784	3024	0.01	0.00	0.00	0.01	0.00	0.00	0.01	0.00	0.00
4	3	0	784	3024	0.00	0.10	0.36	0.00	0.10	0.36	0.00	0.10	0.36
5	2	1	784	3024	6.44	5.50	10.21	6.44	5.50	10.21	6.44	5.50	10.21
6	5	1	784	3024	5.00	7.67	18.25	5.00	7.67	18.25	5.00	7.67	18.25
7	6	0	784	3024	-1.49	4.97	19.53	-1.49	4.97	19.53	-1.49	4.97	19.53
8	15	0	784	3024	-2.18	4.00	17.49	-2.18	4.00	17.49	-2.18	4.00	17.49
9	15	2	784	3024	1.57	7.74	22.34	1.57	7.74	22.34	1.57	7.74	22.34
10	26	3	784	3024	2.31	7.68	19.35	2.31	7.68	19.35	2.31	7.68	19.35

Table 6.6 Average percent improvement of MOEA/D over pre-scalarization (region cluster size = 10)

Prob. #	# of Sum Obj.	# of Max Obj.	Avg. Nodes	Avg. Edges	$\xi = 0$			$\xi = 0.5$			$\xi = 1$		
					WS	OWA	TE	WS	OWA	TE	WS	OWA	TE
1	2	0	35	139	0.00	0.85	4.81	0.00	0.99	5.90	0.00	0.97	6.48
2	1	1	12	40	8.79	9.57	14.71	6.16	6.19	6.38	9.34	9.08	9.46
3	0	2	12	40	1.69	1.13	2.27	0.66	0.76	1.27	0.12	0.37	1.18
4	3	0	11	38	0.00	0.16	0.71	0.00	0.00	0.09	0.00	0.01	0.08
5	2	1	34	129	8.99	13.08	20.73	11.20	9.24	9.92	9.38	9.45	11.74
6	5	1	35	146	17.12	20.55	26.42	15.41	12.94	13.23	17.12	14.73	14.02
7	6	0	44	191	-0.12	1.61	6.12	-0.01	2.02	8.01	-0.01	2.43	9.56
8	15	0	35	146	-0.01	1.10	5.04	-0.01	0.80	4.69	-0.01	1.11	7.47
9	15	2	32	126	12.99	12.49	16.51	6.64	7.42	9.26	3.77	4.08	5.41
10	26	3	35	144	18.52	20.34	24.47	10.94	7.18	8.11	8.41	5.43	8.85

6.6 A Greedy Algorithm for the CMM Framework

A problem in the CMM framework can be expressed as a resource gathering task. The agent begins with a list of demanded resources and begins to move through the environment in pursuit of these goals. In this chapter, we have considered fully observable least-cost path problems. To solve one of these problems in the CMM framework, the agent starts with only one resource in its list of demands. The agent then uses one of the methods

presented in this chapter to plan a least-cost route to one of the resource locations. The set of options can be evaluated in terms of the quality of the Pareto optimal set, but to solve the problem, the agent must pick one of these path options and physically move to the goal location. In fully observable problems with no region clustering the region graph does not change as the agent moves, so there may be no need to iteratively step through the simulation server. However, in partially observable environments or those using region clustering, the region graph can change, so a new plan needs to be developed after each step.

Algorithm 6.4 shows a greedy algorithm for the CMM framework that can be used to solve generic resource gathering problems. The algorithm is greedy in the sense that it always moves toward the closest objective without any additional planning. Steps 1 and 2 of the algorithm take the current mental map data structure $\mathcal{M}$ and use the latest observation from the simulation server $\mathcal{O}$ to recompute the region graph $\mathcal{M}.G_R$. In Step 3, any observed resources that are still in demand are set as target locations, and if no visible resources are in demand, all unobserved regions are used instead. Step 4 uses either the pre-scalarized decomposition method from Algorithm 6.2 or the MOEA/D method from Algorithm 6.3 to find a solution path p that minimizes the cost to one of the target locations determined in Step 3. In problems where computation time is a factor, it may be desirable to use the pre-scalarized approach, rather than the more exhaustive MOEA/D search. Finally, in Step 5, the path p is saved to $\mathcal{M}$ as the current plan and the first step in the plan is returned to the simulation server as the agent's action. This algorithm is called any time the region graph changes, either from observing new areas of the environment or when the agent moves and the regions are reclustered.

Algorithm 6.4 A Greedy Algorithm for the CMM Framework

Input:

- $\mathcal{M}$: a mental map data structure
- $\mathcal{O}$: the most recent observation of the environment
- *opt*: region clustering options

Step 1) Update the mental map: Use Algorithm 4.4 to integrate the most recent observation $\mathcal{O}$ into the mental map $\mathcal{M}$.

Step 2) Update the region graph: Use Algorithm 5.16 to update the region graph $\mathcal{M}.G_R$ using the options defined in *opt*.

Step 3) Determine the target locations:
If there are visible resources that are still in demand,
Set each of these resources as a target location.
Otherwise,
Set all unobserved regions as target locations.

Step 4) Solve the MO-FLCPP: Using the target locations from Step 3 and the current agent location, use either the pre-scalarized decomposition method from Algorithm 6.2 or the MOEA/D method from Algorithm 6.3 to find a solution path p.

Step 5) Choose an action: Save the path p to the mental map $\mathcal{M}$ as the current plan and set the next action as the first step in the plan.

Output: A movement action

The greedy algorithm can be used to solve any generic problem in the CMM framework. To demonstrate, we show an overview of the steps taken by an agent solving a travelling purchaser problem (TPP) in a simulated world environment. The environment is a 30×30 grid constructed with five region types using the method outlined in Section 3.5.3. Each region type is assigned a unique type of resource, and these resources are

distributed randomly throughout the environment. The agent's list of demands requires it to collect one of each type of resource. The environment is only partially observable, so to discover and collect all the necessary resources, the agent will need to explore the environment. We define six objectives for the agent to minimize, the same as Problem 6 from Table 6.3: five terrain type features and the maximum absolute value of the elevation change. The feature weights are defined in Table 6.7. The agent uses the OWA scalarization method with harmonic weights defined by Equation 6.16 and a defuzzification value of $\xi = 0.5$. Figures 6.19-6.22 show the simulation output for the greedy agent using four different region clustering methods.

Table 6.7 Feature weights for the example greedy agent

	$\tilde{f}_{t(1)}$ *meadow*	$\tilde{f}_{t(2)}$ *forest*	$\tilde{f}_{t(3)}$ *water*	$\tilde{f}_{t(4)}$ *rock*	$\tilde{f}_{t(5)}$ *snow*	$\tilde{f}_{h_max}$ *slope*
Unnormalized	1	5	10	3	8	2
Normalized	0.0345	0.1724	0.3448	0.1034	0.2759	0.0690

Figure 6.19 shows the path traveled by the agent using the greedy algorithm with no region clustering. The last image shows the winding route that the agent took, searching nearly the entire environment before discovering the final two resources on top of the hill. The agent makes several sweeps through the forest region since observability is limited to only the adjacent grid cells in the forest, taking a total of 233 simulation steps. The final scalarized path cost is given in Table 6.8 as 0.1611 when normalized with the other three approaches.

Figure 6.20 shows the solution path when using a local region size of 3, an observed cluster size of 5, and an unobserved cluster size of 20. Because of the way the environment is dynamically partitioned as the agent moves, the agent takes a different route than when using no region clustering, occasionally taking less direct paths. The solution takes 125 simulation steps and has a scalarized cost of 0.1854.

Figure 6.21 shows the path followed by an agent using the same region clustering parameters as above, but while keeping a memory of the local region. This approach grows the size of the graph as the environment is explored and can provide more efficient planning through previously explored terrain. The solution takes 96 simulation steps and has a scalarized cost of 0.1666.

Lastly, Figure 6.22 shows the path of an agent that only uses region clustering for the unobserved regions (cluster size = 20). This is accomplished by setting the local region size to infinity, allowing all observed areas to be represented with no imprecision. The solution of this agent is the most efficient, taking 78 simulation steps with a scalarized cost of 0.1341.

Table 6.8 Solution costs of the example greedy agent

Region Clustering Type	Simulation Steps	Normalized and Scalarized Cost	Aggregated Feature Costs					
			$f_{t(1)}$ *meadow*	$f_{t(2)}$ *forest*	$f_{t(3)}$ *water*	$f_{t(4)}$ *rock*	$f_{t(5)}$ *snow*	f_{h_max} *slope*
No region clustering	233	0.1611	88.5	124	13	4	3.5	0.1385
Small local region with no memory	125	0.1854	70.5	27	19	5	3.5	0.1200
Small local region with memory	96	0.1666	71.5	1	8.5	6	9	0.1200
Region clustering only if unobserved	78	0.1341	52.5	6	6.5	6	7	0.1200

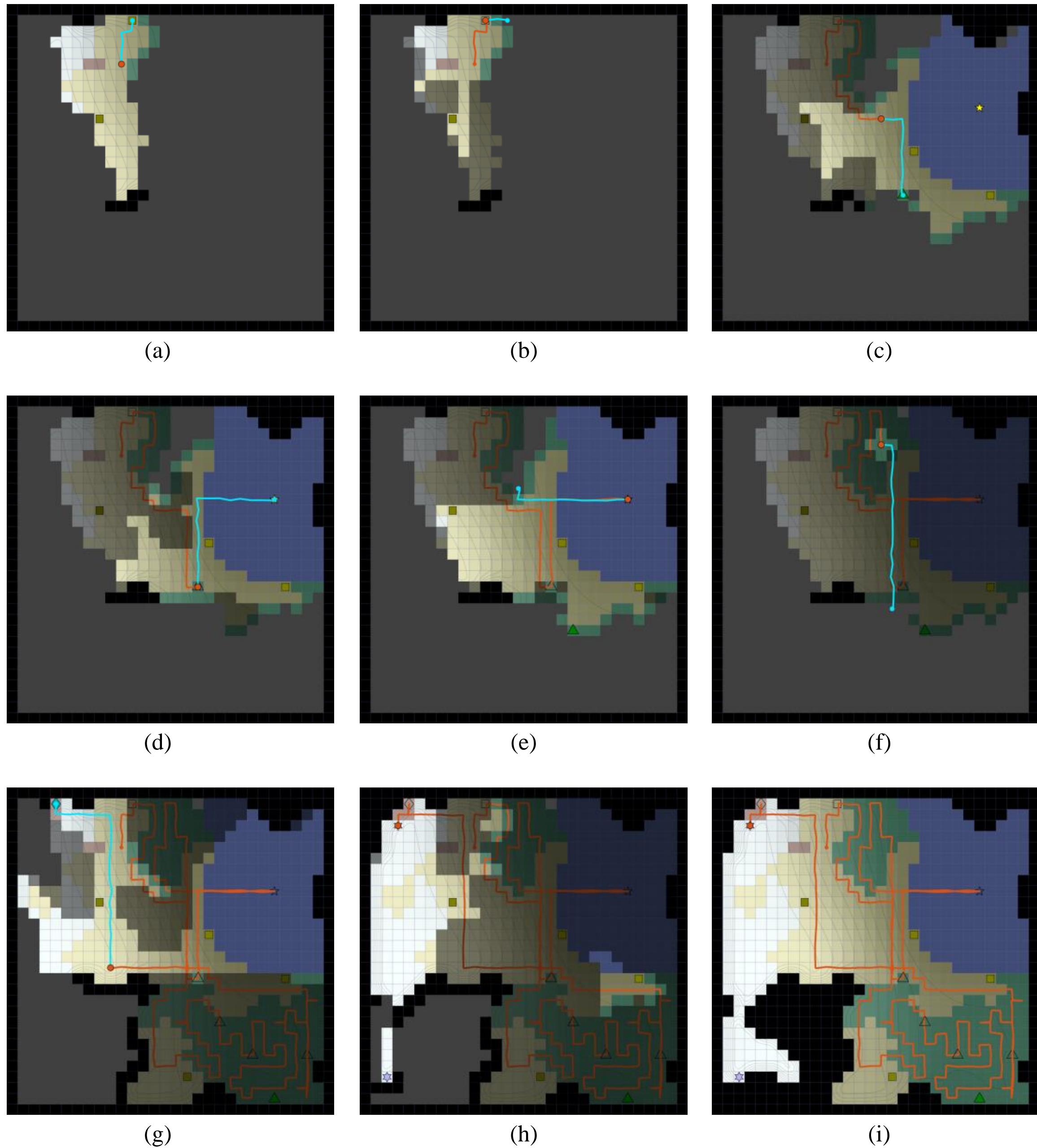


Figure 6.19 An agent solving a TPP in the CMM framework with no region clustering. (a) The agent plans a route to the nearest observed resource in the open meadow. (b) The remaining resource types have not yet been observed, so the agent plans a route to the nearest unobserved region. (c) After exploring part of the unobserved region, two new resources are discovered. The agent plans a route to the closest one in a forest. (d) The agent plans a route to the resource in the water. (e) The agent continues to explore the unobserved region at the top of the map. (f) The agent decides to begin exploring the unobserved region at the bottom of the map. (g) After wandering through the forest at the bottom of the map, the agent finally discovers one of the remaining needed resources at the top of the hill. (h) The agent collects the final resources and ends the problem. (i) Final path traveled by the agent.

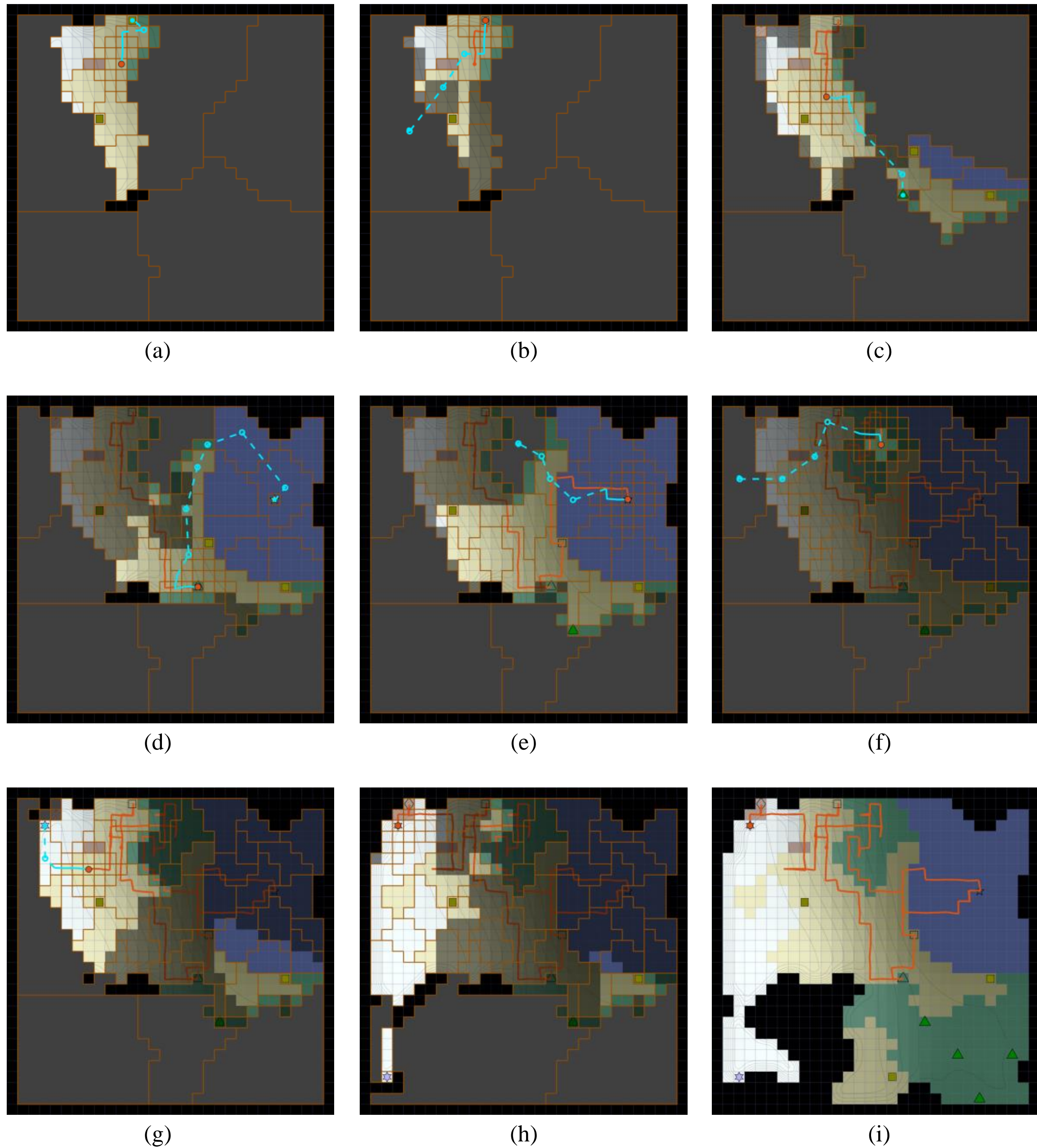


Figure 6.20 An agent solving a TPP in the CMM framework using a small local region with no memory and region clustering in all areas. (a) The agent plans a route to the nearest observed resource in the open meadow. (b) The remaining resource types have not yet been observed, so the agent plans a route to the nearest unobserved region. (c) En route to the unobserved region, the agent discovers the forest resource and plans a new route. (d) After collecting the forest resource, the agent plans a route to the newly discovered water resource. (e) The agent decides to explore the unobserved region at the top of the map. (f) After exploring this area, the agent plans a route to the unobserved region on top of the hill. (g) The agent discovers the snow resource and plans a new route. (h) The agent discovers and collects the rock resource on the way to the snow resource, finishing the problem. (i) Final path traveled by the agent.

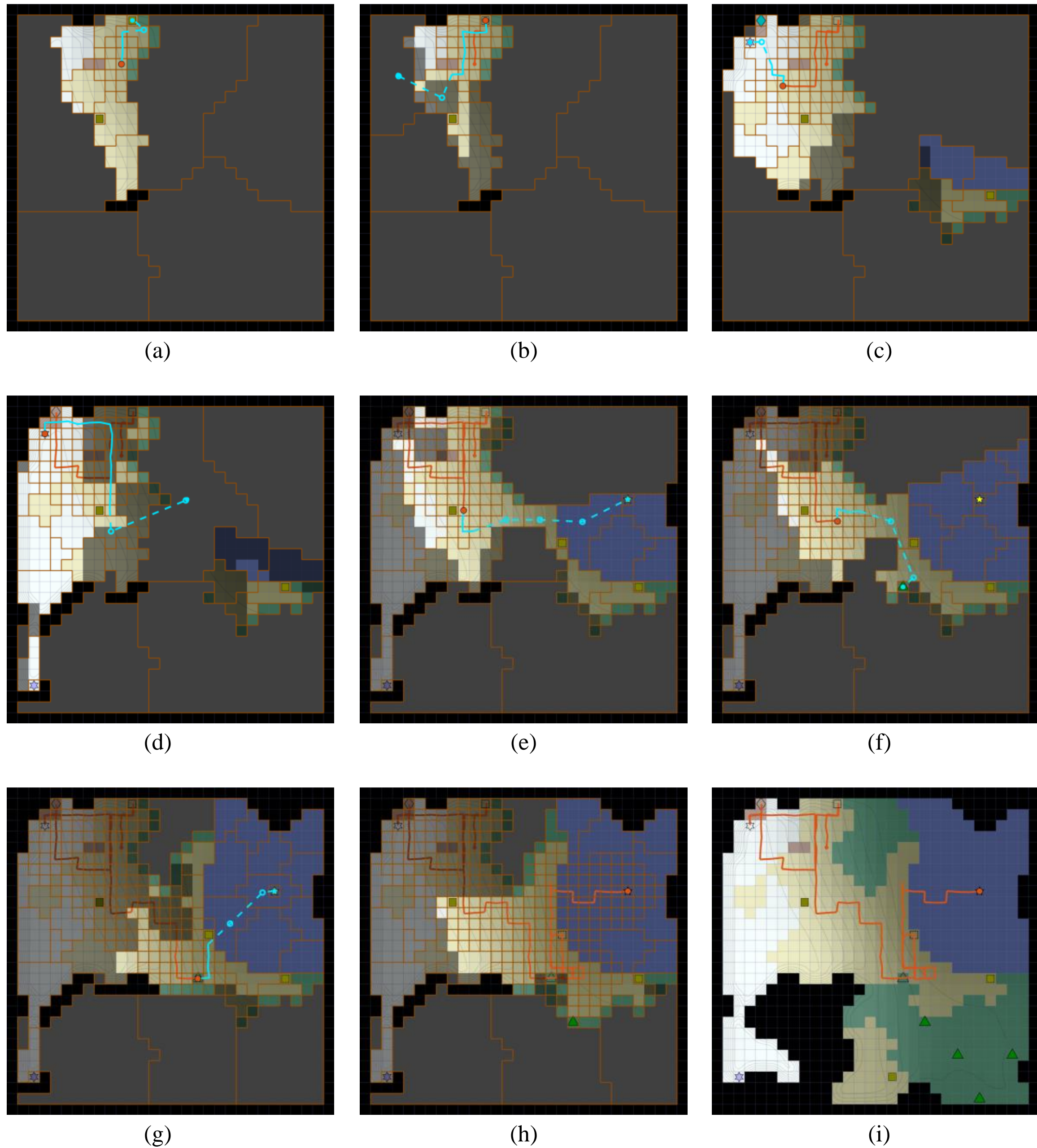


Figure 6.21 An agent solving a TPP in the CMM framework using a small local region with memory and region clustering in all areas. (a) The agent plans a route to the nearest observed resource in the open meadow. (b) After collecting the meadow resource, the agent plans a route to the unobserved area on top of the hill. (c) The agent discovers the snow and rock resources and plans a new route to collect them. (d) The agent plans a route to the large unobserved region in the top center of the map. (e) The agent discovers the water resource and plans a route to collect it. (f) On the way, the agent discovers the forest resource and plans a new route. (g) After collecting the forest resource, the agent plans a route to the final water resource. (h) The agent wanders along the shore, trying to find the least-cost path to the last resource. (i) Final path traveled by the agent.

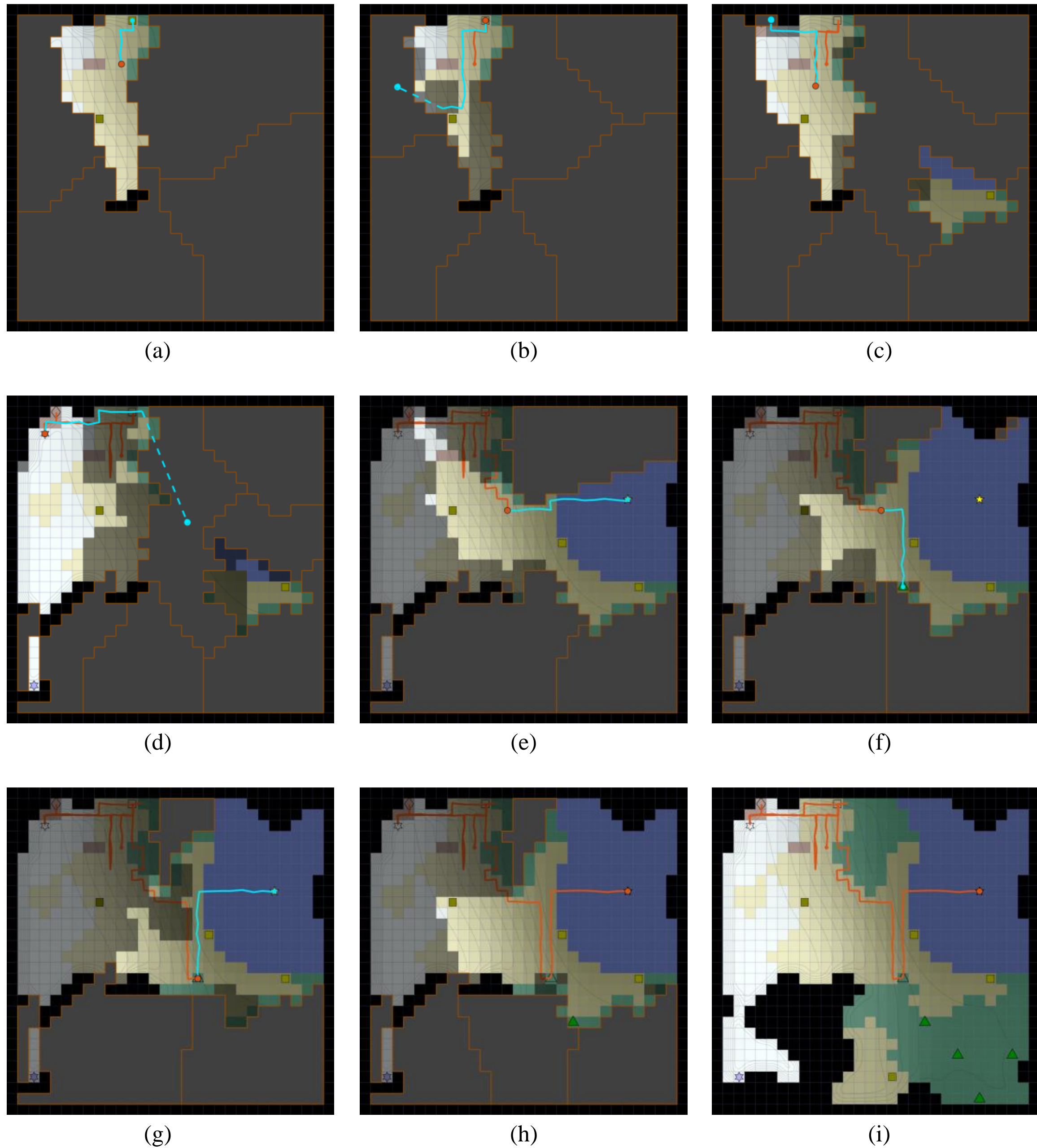


Figure 6.22 An agent solving a TPP in the CMM framework using region clustering only for unobserved regions and no region clustering elsewhere. (a) The agent plans a route to the nearest observed resource in the open meadow. (b) After collecting the meadow resource, the agent plans a route to the unobserved area on top of the hill. (c) On the way, the agent discovers the rock resource and decides to turn around to collect it. (d) After collecting both the rock and snow resources, the agent plans a route to the large unobserved region in the top center of the map. (e) The agent discovers the water resource and plans a route to collect it. (f) On the way, the agent discovers the forest resource and plans a detour to retrieve it first. (g) After collecting the forest resource, the agent plans a route to the last water resource. (h) The agent travels along the shore and travels directly to the final resource, ending the problem. (i) Final path traveled by the agent.

In this example scenario, the effect of region clustering seems to offer the greatest benefit when applied only to the unobserved regions. However, it is difficult to make general claims as to the effect of various region clustering approaches. In some environments, applying region clustering to observed regions can degrade performance, as the agent must continually recluster these areas and may end up taking less direct routes. If no memory is used for the local region, the agent can get stuck in a cycle moving back and forth between two grid cells as the regions are reclustered. This can still occur in some instances when using local region memory, but it is less common. Increasing the size of the local region can also help to avoid oscillatory behavior. Ultimately, the individual differences in each environment can make any one region clustering method perform better than another.

Perhaps the best reason to use region clustering is to reduce the size of the planning search space. The greedy algorithm does not utilize this property to its full potential, but more advanced planning algorithms may benefit greatly from this. For example, the approach used in (Buck and Keller 2016) for solving the partially observable traveling salesman problem used a Monte Carlo sampling method to construct a distribution of possible target locations. By reducing the number of possible locations and adopting a fuzzy methodology, it may be possible to improve upon this approach and allow for the development of long-term agent strategies. This is a possible topic for future work with the CMM framework and will be discussed further in the next and final chapter.

6.7 Summary

This chapter defined the multiobjective least-cost path problem (MO-FLCPP) and presented a greedy agent strategy for solving generic problems in the CMM framework. We first showed the issues regarding selection bias in grid world domains and proposed to overcome these issues by adding a small amount of random noise to the graph edge weights. The MO-FLCPP was introduced with an example that demonstrated how to determine the set of Pareto optimal solutions using different aggregation and scalarization methods. We showed that a decision-making agent could choose different solution paths depending on its preferences and that the choice of which path to take depends largely on the objective weights and scalarization method used.

We then described an approach to pre-scalarize the feature values in a fuzzy weighted graph, accounting for both summation and maximization aggregation methods so that a solution path could be found using a standard implementation of Dijkstra's algorithm. This approach can be used to quickly find a solution using a specific set of objective weights. The solution can often be improved, however, by applying an evolutionary search procedure using MOEA/D. This method returns an approximation of the entire Pareto optimal set of solutions. Selecting a path from this set will often yield an improvement in the scalarized cost and provides additional context, allowing for better normalization to judge how well a solution compares with other possible options.

We concluded the chapter with several example problems to demonstrate how different feature sets, scalarization functions, and region clustering approaches can impact the solution quality. We considered several two objective scenarios with binary terrain and elevation features, and then investigated problems with many objectives. A series of

experiments on several different problem types showed that the MOEA/D search often produces a better result, particularly when the problem includes many nonlinearities. We ended with a description of a greedy agent algorithm that can be used to solve any type of resource collecting problem in the CMM framework, such as the shortest path problem, traveling salesman problem, and traveling purchaser problem. An example demonstrated how different configurations of the local region can affect the performance of the agent. Some of the methods presented in this chapter can be extended to perform long-term planning, which can improve the solution quality when visiting multiple locations in sequence. Such extensions are beyond the scope of this current work and are discussed briefly in the last chapter.

7 CONCLUSION

The CMM framework has great potential for use in many applications. This final chapter summarizes the contributions of the CMM framework and presents several possibilities for future work.

7.1 Summary of the CMM Framework

In this work, we introduced the CMM framework as a simulation environment for studying multiobjective pathfinding problems with partial observability. These problems allow decision-making agents to demonstrate purposeful behavior in pursuit of a goal. The scenarios are interpretable and can be adapted for use in other problem domains of interest where it is beneficial to have a highly customizable, controllable, and repeatable test environment. Beyond developing sequential decision-making models, the CMM framework can also be used to generate synthetic trajectories that can be used for behavior modeling and anticipatory analysis.

The models in the CMM framework embrace uncertainty using the machinery and logic of fuzzy sets. Procedurally generated grid world environments are represented using fuzzy weighted graphs where vertices represent spatial regions and edges indicate the cost of moving between adjacent regions. Movement costs are represented as fuzzy numbers to capture the uncertainty of the minimum, maximum, and expected feature values of each edge. We employ various graph search techniques and approximations to compute the edge weights of the region graph. In fully observable environments with no region clustering, the features are crisp values. However, edges that are only partially observable or represent

movement between large regions are weighted with fuzzy feature values. This allows the agent to express some degree of optimism or pessimism when choosing an action.

Multiple objectives are managed by defining a relative weight for each objective and a scalarization function to reduce the problem to a single objective. We consider the weighted sum, Tchebycheff, and OWA methods for scalarization. The OWA approach can be implemented as a hybrid operator that represents a form of bounded rationality in which the agent can only consider a few objectives at once. Feature values along a path are aggregated using either summation or maximization, and an approximate scalarized path cost for mixed aggregation methods can be computed using exponential scaling. A multiobjective evolutionary algorithm with decomposition (MOEA/D) is used to find a Pareto optimal set of nondominated paths.

We showed several examples that demonstrate how the CMM framework can be used to solve multiobjective fuzzy least-cost path problems (MO-FLCPPs) in grid world environments. The choice of features, region clustering parameters, and scalarization method can greatly impact the solutions that are found. An experiment with many randomly generated test instances across several problem types found that in general, the MOEA/D search method can improve the quality of the solutions found over a pre-scalarized approach. The amount of improvement is most apparent for problems with many nonlinearities coming from either the aggregation method or scalarization function. We finished with a demonstration of a greedy algorithm that can be used to solve generic resource collecting problems. It was shown that the path followed by the greedy agent is very sensitive to the interpretation of the environment formed by different region clustering methods.

7.2 Future Work

The full potential of the CMM framework extends far beyond the material discussed in this work. Our initial goal was to create a simulation environment that could be used to study models of sequential multiobjective decision-making behavior in partially observable environments for anticipatory analysis. To achieve this, we required an environment in which an agent could demonstrate purposeful behavior while solving some problem that could be recognized by an analyst. In particular, the pathfinding problem was of considerable interest due to the nature of representing spatial uncertainty. The traveling salesman problem and its more generic variant, the traveling purchaser problem, provided highly configurable scenarios for the agent to solve.

Although reasoning about ideal models of environment representation and agent knowledge yielded many promising ideas, an important goal of this work was to produce a working computational framework that could facilitate many diverse experiments. This required some design compromises to build a functional model. The CMM framework is implemented in the Matlab programming language, which was chosen for its prototyping efficiency and ease of visualization. Several optimizations helped to improve the runtime performance, but certain assumptions were made for the sake of simplifying the architectural requirements. Perhaps chiefly among these was the restriction to grid world domains. Many of the methods in the CMM framework—such as procedural environment generation, feature computation, and region clustering—assume a discrete grid structure. Although these ideas could be adapted for continuous domains, it would not be a straightforward exercise. One issue that arises with continuous domains is the increased size of the search space when planning actions. This could be mitigated to some degree by

sampling a traversal graph and using a continuous form of region clustering. The spatial relationships between regions could be represented as fuzzy attributes, giving greater flexibility in the representation of the environment and allowing for uncertainty in the location of each graph vertex. Such an approach would likely require significant changes to the framework, but could make it easier to apply the models to real-world data.

The greedy agent strategy presented in this work is a straightforward solution to a complex problem. A greedy approach can often create an acceptable solution to a problem such as the TSP, but rarely an optimal one. To create more intelligent agents, an algorithm such as ant colony optimization (ACO) can be used to plan the optimal sequence in which goal locations should be visited. A fully connected planning graph would be defined over all resources and the agent location. Simulated "ants" would stochastically solve the problem and deposit pheromones on promising edges of the planning graph, attracting future ants and eventually the agent itself. This approach could be used in a multiobjective framework by combining ACO with MOEA/D (Ke, Zhang, and Battiti 2013).

In partially observable environments, the planning graph would need to include unobserved regions if there is a possibility that the region contains a needed resource. Depending on the region clustering method used, this could result in a very large graph. An alternate approach is to sample potential resource locations in the environment and build a distribution of the best paths to each sampled location. These paths can be used to construct a value map of how beneficial it would be for the agent to move toward a given area. This is the approach used by the Myopic Monte Carlo (MMC) agent policy in (Buck and Keller 2016). Figure 7.1 shows an example of a partially observable traveling salesman

problem (PO-TSP) solved by a greedy agent policy such as the one given in Algorithm 6.4. The same problem is solved by an MMC agent policy in Figure 7.2.

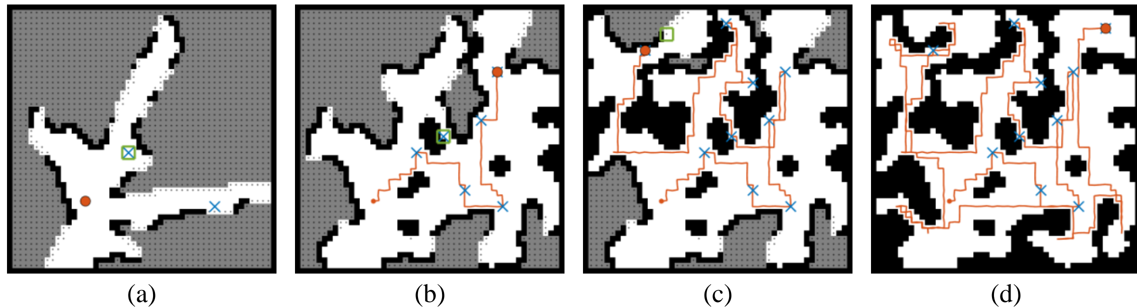


Figure 7.1 Some selected moments from the greedy policy's solution for the PO-TSP with symbols enlarged for clarity. (a) The initial mental map shows only what is visible from the agent's starting location (red circle), which includes two waypoints (blue crosses). The closest of these is chosen as the target objective (green square). Grey areas indicate unknown areas of the environment and dots signify the possibility of a waypoint. (b) The first five waypoints are acquired greedily and the sixth target is chosen as a waypoint that was discovered along the route, but requires the agent to backtrack. (c) Nine targets are acquired by always moving toward the nearest unvisited waypoint if one is visible, or the nearest unexplored area otherwise. (d) The final target is hidden behind a corner that was not fully explored on the first pass and is not discovered until the entire environment has been explored.

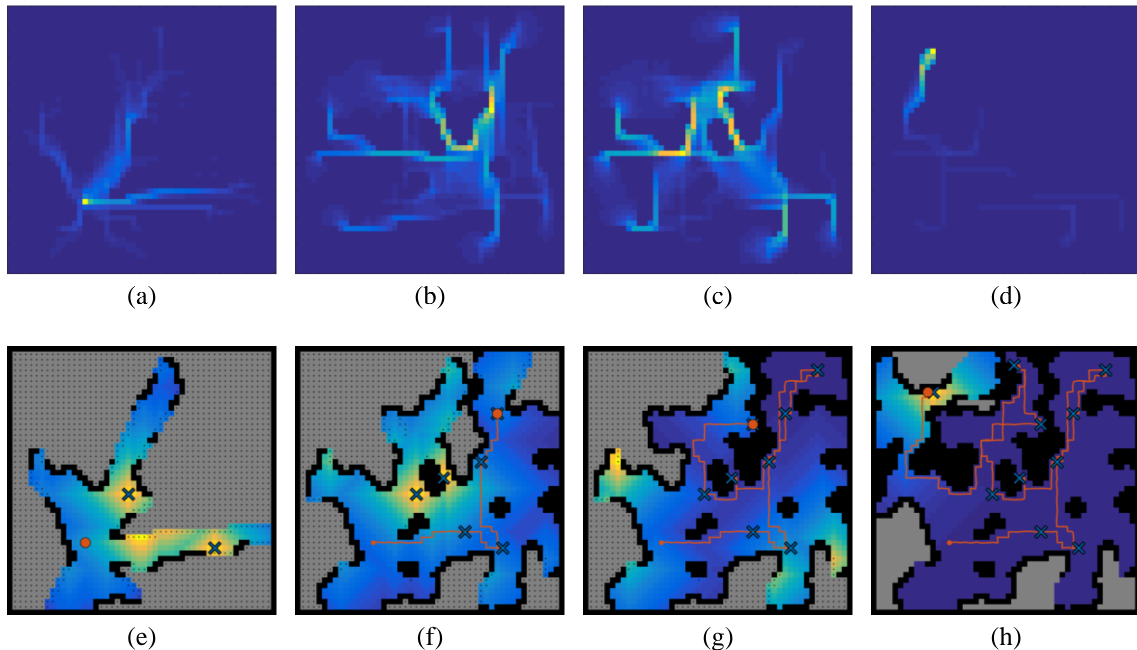


Figure 7.2 Some selected moments from the MMC policy's solution to the same PO-TSP environment used in Figure 7.1. The top row (a-d) shows the pheromone map, which aggregates the shortest paths from the agent to sampled waypoints. The bottom row (e-h) shows the value map, which defines a gradient that the agent follows. Symbols are enlarged for clarity. The initial observation (e) is identical to Figure 7.1a, but instead of picking a target location, 1000 waypoint locations are sampled and the shortest paths back to the agent are aggregated in a persistent pheromone map (a). Performing value iteration on the unobserved regions of (a) gives the gradient map (e) that the agent follows. (b) and (f) show the maps after reaching four waypoints as the MMC agent recognizes the possibility of a waypoint in the top-right and discovers the waypoint that was missed by the greedy agent. (c) and (g) show the maps after visiting eight waypoints when the agent could proceed to the high value area in the left, but instead follows the local gradient toward the top and discovers the ninth waypoint. The final maps are shown in (d) and (h) where most of the pheromone has evaporated except for a single trail and only a single peak is left in the gradient map.

Although the MMC policy is an improvement over the greedy approach, it still does not develop a complete plan to collect all waypoints as with ACO in fully observable environments. To adapt ACO for partially observable environments, we can treat each ant as a solution to the problem in an environment sampled from the distribution of all possible environments based on the current state of the mental map. This is effectively the approached used by Monte Carlo Tree Search (MCTS) (Browne et al. 2012) adapted for partially observable domains (Silver and Veness 2010). MCTS has been very effective at

learning agent policies for problems modeled as Markov decision processes (MDPs) and partially observable MDPs (POMDPs). A multiobjective version of MCTS was used to solve the multiobjective physical traveling salesman problem (Perez et al. 2015), which was based on a competition to design a controller for an agent solving the single objective physical traveling salesman problem (Perez, Rohlfshagen, and Lucas 2012). MCTS methods could be very effective at solving problems in the CMM framework, which is well-suited for evaluating different agent strategies in a competition setting.

Lastly, we mention that the CMM framework can be used to develop advanced visualization techniques for many-objective problems with fuzzy parameters. Solutions to pathfinding problems are easy to understand and interpret in isolation, but it can be difficult to convey the tradeoffs between many different Pareto optimal solutions. The visualization approach in (He and Yen 2016) can be used to show high-dimensional Pareto fronts, and the fuzzy rose diagrams introduced in (Buck and Keller 2014) can show fuzzy weighted graphs and the expected costs of multiple route options. These methods and others could help analysts better understand the results of multiobjective pathfinding algorithms.

In closing, there are many potential avenues of research involving the CMM framework that could be further explored. From designing long-term agent strategies using MOEA/D-ACO and MCTS, to using the simulator to generate synthetic trajectories for use in anticipatory analysis applications, there are many possible use cases. Ultimately, the tools and methods presented in this work should prove to be valuable resources in the understanding of sequential multicriteria decision-making problems with uncertainty.

REFERENCES

Achanta, Radhakrishna, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk. 2012. "SLIC Superpixels Compared to State-of-the-Art Superpixel Methods." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34 (11): 2274–82. doi:10.1109/TPAMI.2012.120.

Aggarwal, Charu C., Alexander Hinneburg, and Daniel A. Keim. 2001. "On the Surprising Behavior of Distance Metrics in High Dimensional Space." In *Database Theory – ICDT 2001*, 420–34. doi:10.1007/3-540-44503-X_27.

Amanatides, John, and Andrew Woo. 1987. "A Fast Voxel Traversal Algorithm for Ray Tracing." *Eurographics* 87 (3): 3–10. doi:10.1.1.42.3443.

Applegate, David L., Robert E. Bixby, Vasek Chvátal, and William J. Cook. 2007. *The Traveling Salesman Problem: A Computational Study*. *Princeton University Press*. http://press.princeton.edu/titles/8451.html.

Ashlock, Daniel. 2015. "Evolvable Fashion-Based Cellular Automata for Generating Cavern Systems." In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, 306–13. IEEE. doi:10.1109/CIG.2015.7317958.

Aspers, Patrik. 2001. "Crossing the Boundary of Economics and Sociology: The Case of Vilfredo Pareto." *American Journal of Economics and Sociology* 60 (2): 519–45. doi:10.1111/1536-7150.00073.

Bader, Johannes, and Eckart Zitzler. 2011. "HypE: An Algorithm for Fast Hypervolume-Based Many-Objective Optimization." *Evolutionary Computation* 19 (1): 45–76. doi:10.1162/EVCO_a_00009.

Bellman, Richard. 1958. "On a Routing Problem." *Quarterly of Applied Mathematics* 16 (1): 87–90.

Berlekamp, Elwyn R., John H. Conway, and Richard K. Guy. 1982. *Winning Ways for Your Mathematical Plays*. *American Mathematical Monthly*. Vol. 1–2. New York: Academic Press. doi:10.2307/2323620.

Berman, Oded, and Gabriel Y. Handler. 1987. "Optimal Minimax Path of a Single Service Unit on a Network to Nonservice Destinations." *Transportation Science* 21 (2): 115–22. doi:10.1287/trsc.21.2.115.

Blisard, Samuel N., and Marjorie Skubic. 2005. "Modeling Spatial Referencing Language for Human-Robot Interaction." In *ROMAN 2005. IEEE International Workshop on Robot and Human Interactive Communication, 2005.*, 698–703. IEEE. doi:10.1109/ROMAN.2005.1513861.

Boctor, Fayez F., Gilbert Laporte, and Jacques Renaud. 2003. "Heuristics for the Traveling Purchaser Problem." *Computers & Operations Research* 30 (4): 491–504. doi:10.1016/S0305-0548(02)00020-5.

Bonabeau, Eric. 2002. "Agent-Based Modeling: Methods and Techniques for Simulating Human Systems." *Proceedings of the National Academy of Sciences* 99 (Supplement 3): 7280–87. doi:10.1073/pnas.082080899.

Bowman, V. Joseph. 1976. "On the Relationship of the Tchebycheff Norm and the Efficient Frontier of Multiple-Criteria Objectives." In *Multiple Criteria Decision Making: Proceedings of a Conference Jouy-En-Josas, France May 21--23, 1975*, edited by Hervé Thiriez and Stanley Zionts, 76–86. Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-87563-2_5.

Bresenham, J. E. 1965. "Algorithm for Computer Control of a Digital Plotter." *IBM Systems Journal* 4 (1): 25–30. doi:10.1147/sj.41.0025.

Briggs, Ronald. 1973. "Urban Cognitive Distance." In *Image and Environment*, edited by R. M. Downs and D. Stea, 361–88. Chicago: Aldine.

Browne, Cameron B., Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. "A Survey of Monte Carlo Tree Search Methods." *IEEE Transactions on Computational Intelligence and AI in Games* 4 (1): 1–43. doi:10.1109/TCIAIG.2012.2186810.

Buck, Andrew R., and James M. Keller. 2014. "Visualizing Uncertainty with Fuzzy Rose Diagrams." In *2014 IEEE Symposium on Computational Intelligence for Engineering Solutions (CIES)*, 30–36. IEEE. doi:10.1109/CIES.2014.7011827.

———. 2016. "A Myopic Monte Carlo Strategy for the Partially Observable Travelling Salesman Problem." In *2016 IEEE Congress on Evolutionary Computation (CEC)*, 632–39. IEEE. doi:10.1109/CEC.2016.7743852.

Buck, Andrew R., James M. Keller, and Mihail Popescu. 2014. "An α-Level OWA Implementation of Bounded Rationality for Fuzzy Route Selection." In *Studies in Fuzziness and Soft Computing*, 312:253–60. doi:10.1007/978-3-319-03674-8_24.

Busch, Mark A., Marjorie Skubic, James M. Keller, and Kevin E. Stone. 2007. "A Robot in a Water Maze: Learning a Spatial Memory Task." In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, 1727–32. Roma: IEEE. doi:10.1109/ROBOT.2007.363572.

Cadwallader, M. T. 1976. "Cognitive Distance in Intra-Urban Space." In *Environmental Knowing*, edited by G. T. Moore and R. G. Golledge, 316–24. Stroudsberg, PA: Dowden, Hutchinson, and Ross.

Chapman, Paul. 2002. "Life Universal Computer." http://www.igblan.free-online.co.uk/igblan/ca/.

Chown, Eric, Stephen Kaplan, and David Kortenkamp. 1995. "Prototypes, Location, and Associative Networks (PLAN): Towards a Unified Theory of Cognitive Mapping." *Cognitive Science* 19 (1): 1–51. doi:10.1207/s15516709cog1901_1.

Cornelis, Chris, Peter De Kesel, and Etienne E. Kerre. 2004. "Shortest Paths in Fuzzy Weighted Graphs." *International Journal of Intelligent Systems* 19 (11): 1051–68. doi:10.1002/int.20036.

Coucleis, H., R. G. Golledge, N. Gale, and W. Tobler. 1987. "Exploring the Anchor-Point Hypothesis of Spatial Cognition." *Journal of Environmental Psychology* 7 (2): 99–122. doi:10.1016/S0272-4944(87)80020-8.

Deb, Kalyanmoy, Manikanth Mohan, and Shikhar Mishra. 2005. "Evaluating the Epsilon-Domination Based Multi-Objective Evolutionary Algorithm for a Quick Computation of Pareto-Optimal Solutions." *Evolutionary Computation* 13 (4): 501–25. doi:10.1162/106365605774666895.

Deb, Kalyanmoy, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2002. "A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II." *IEEE Transactions on Evolutionary Computation* 6 (2): 182–97. doi:10.1109/4235.996017.

Dijkstra, E. W. 1959. "A Note on Two Problems in Connexion with Graphs." *Numerische Mathematik* 1 (1): 269–71.

Dorigo, Marco, Vittorio Maniezzo, and Albert Colorni. 1996. "Ant System: Optimization by a Colony of Cooperating Agents." *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)* 26 (1): 29–41. doi:10.1109/3477.484436.

Downs, Roger M., and David Stea. 1977. *Maps in Minds: Reflections on Cognitive Mapping*. New York: Harper & Row.

Dubois, D., and H. Prade. 1980. *Fuzzy Sets and Systems: Theory and Applications*. New York: Academic Press.

Durand, F. 2000. "A Multidisciplinary Survey of Visibility." In *ACM SIGGRAPH Course Notes Visibility, Problems, Techniques, and Applications*. doi:10.1.1.15.8992.

Eliashberg, Victor. 2002. "What Is Working Memory and Mental Imagery? A Robot That Learns to Perform Mental Computations." *System*. Palo Alto, CA. http://arxiv.org/abs/cs/0309009.

Fishman, Jeremy, Herman Haverkort, and Laura Toma. 2009. "Improved Visibility Computation on Massive Grid Terrains." In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - GIS '09*, 121. New York, New York, USA: ACM Press. doi:10.1145/1653771.1653791.

Floriani, Leila De, and Paola Magillo. 1994. "Visibility Algorithms on Triangulated Digital Terrain Models." *International Journal of Geographical Information Systems* 8 (1): 13–41. doi:10.1080/02693799408901985.

Floyd, Robert W. 1962. "Algorithm 97: Shortest Path." *Communications of the ACM* 5 (6): 345. doi:10.1145/367766.368168.

Fonseca, Carlos M., and Peter J. Fleming. 1995. "An Overview of Evolutionary Algorithms in Multiobjective Optimization." *Evolutionary Computation* 3 (1): 1–16. doi:10.1162/evco.1995.3.1.1.

Ford Jr., R. L. 1956. "Network Flow Theory." Santa Monica, California.

Fournier, Alain, Don Fussell, and Loren Carpenter. 1982. "Computer Rendering of Stochastic Models." *Communications of the ACM* 25 (6): 371–84. doi:10.1145/358523.358553.

Franklin, Randolph, and Clark K. Ray. 1994. "Higher Isn't Necessarily Better: Visibility Algorithms and Experiments." In *Advances in GIS Research: Sixth International Symposium on Spatial Data Handling*, 2:1–22. doi:10.1.1.17.5634.

Fredman, Michael L., and Robert Endre Tarjan. 1984. "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms." In *25th Annual Symposium on Foundations of Computer Science, 1984.*, 34:338–46. IEEE. doi:10.1109/SFCS.1984.715934.

———. 1987. "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms." *Journal of the ACM* 34 (3): 596–615. doi:10.1145/28869.28874.

Gardner, Martin. 1970. "Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game 'life.'" *Scientific American* 223 (October): 120–123. doi:10.1038/scientificamerican0169-116.

Gärling, T., A. Book, and E. Lindberg. 1985. "Adults' Memory Representations of the Spatial Properties of Their Everyday Physical Environment." In *The Development of Spatial Cognition*, edited by R. Cohen, 141–84. Hillsdale, NJ: Erlbaum Lawrence.

Gass, Saul, and Thomas Saaty. 1955. "The Computational Algorithm for the Parametric Objective Function." *Naval Research Logistics Quarterly* 2 (1–2). Wiley Subscription Services, Inc., A Wiley Company: 39–45. doi:10.1002/nav.3800020106.

Gould, Peter, and Rodney White. 1992. *Mental Maps*. Second edi. London: Routledge.

Guerriero, F., and R. Musmanno. 2001. "Label Correcting Methods to Solve Multicriteria Shortest Path Problems." *Journal of Optimization Theory and Applications* 111 (3): 589–613. doi:10.1023/A:1012602011914.

Gutin, Gregory, and Abraham Punnen. 2007. *The Traveling Salesman Problem and Its Variations*. Edited by Gregory Gutin and Abraham P. Punnen. Vol. 12. Combinatorial Optimization. Boston, MA: Springer US. doi:10.1007/b101971.

Hart, Peter, Nils Nilsson, and Bertram Raphael. 1968. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *IEEE Transactions on Systems Science and Cybernetics* 4 (2): 100–107. doi:10.1109/TSSC.1968.300136.

Haverkort, Herman, Laura Toma, and Yi Zhuang. 2009. "Computing Visibility on Terrains in External Memory." *Journal of Experimental Algorithmics* 13 (1): 1.5.1-1.5.23. doi:10.1145/1412228.1412233.

He, Zhenan, and Gary G. Yen. 2016. "Visualization and Performance Metric in Many-Objective Optimization." *IEEE Transactions on Evolutionary Computation* 20 (3): 386–402. doi:10.1109/TEVC.2015.2472283.

Hernandes, Fábio, Maria Teresa Lamata, José Luis Verdegay, and Akebo Yamakami. 2007. "The Shortest Path Problem on Networks with Fuzzy Parameters." *Fuzzy Sets and Systems* 158 (14): 1561–70. doi:10.1016/j.fss.2007.02.022.

Holland, John H., and John H. Miller. 1991. "Artificial Adaptive Agents in Economic Theory." *The American Economic Review*. http://www.jstor.org/stable/10.2307/2006886.

Hu, T. C. 1961. "The Maximum Capacity Route Problem." *Operations Research* 9 (6): 898–900.

Ishibuchi, Hisao, Noritaka Tsukamoto, and Yusuke Nojima. 2008. "Evolutionary Many-Objective Optimization: A Short Review." In *2008 IEEE Congress on Evolutionary Computation, CEC 2008*, 2419–26. doi:10.1109/CEC.2008.4631121.

Jaillet, Patrick. 1985. "Probabilistic Traveling Salesman Problems." MIT.

Jain, Himanshu, and Kalyanmoy Deb. 2014. "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Nondominated Sorting Approach, Part II: Handling Constraints and Extending to an Adaptive Approach." *IEEE Transactions on Evolutionary Computation* 18 (4): 602–22. doi:10.1109/TEVC.2013.2281534.

Johnson, Lawrence, Georgios N. Yannakakis, and Julian Togelius. 2010. "Cellular Automata for Real-Time Generation of Infinite Cave Levels." In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games - PCGames '10*, 1–4. New York, New York, USA: ACM Press. doi:10.1145/1814256.1814266.

Ke, Liangjun, Qingfu Zhang, and Roberto Battiti. 2013. "MOEA/D-ACO: A Multiobjective Evolutionary Algorithm Using Decomposition and Ant Colony." *IEEE Transactions on Cybernetics* 43 (6): 1845–59.

Keller, James M., Mihail Popescu, and Dustin Gibeson. 2012. "An Extension of a Confined Space Evacuation Model to Human Geography." In *2012 IEEE International Geoscience and Remote Sensing Symposium*, 531–34. IEEE. doi:10.1109/IGARSS.2012.6350861.

Kennedy, James, and Russell Eberhart. 1995. "Particle Swarm Optimization." In *Proceedings of ICNN'95 - International Conference on Neural Networks*, 4:1942–48. IEEE. doi:10.1109/ICNN.1995.488968.

Kitchin, Rob, and Mark Blades. 2002. *The Cognition of Geographic Space*. London: I.B.Tauris.

Klein, Cerry M. 1991. "Fuzzy Shortest Paths." *Fuzzy Sets and Systems* 39 (1): 27–41. doi:10.1016/0165-0114(91)90063-V.

Kreveld, Marc Van. 1996. "Variations on Sweep Algorithms: Efficient Computation of Extended Viewsheds and Class Intervals." In *In Proc. 7th Int. Symp. on Spatial Data Handling*, 1–14.

Lafferty, John, Andrew Mccallum, and Fernando C. N. Pereira. 2001. "Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data." In *Proceedings of the 18th International Conference on Machine Learning 2001 (ICML 2001)*, 2001:282–89.

Lee, C. Y. 1961. "An Algorithm for Path Connections and Its Applications." *IEEE Transactions on Electronic Computers* EC-10 (3): 346–65. doi:10.1109/TEC.1961.5219222.

Li, Bingdong, Jinlong Li, Ke Tang, and Xin Yao. 2015. "Many-Objective Evolutionary Algorithms." *ACM Computing Surveys* 48 (1): 1–35. doi:10.1145/2792984.

Loui, Ronald Prescott. 1983. "Optimal Paths in Graphs with Stochastic or Multidimensional Weights." *Communications of the ACM* 26 (9): 670–76. doi:10.1145/358172.358406.

Luke, Robert H., James M. Keller, Marjorie Skubic, and Steven Senger. 2005. "Acquiring and Maintaining Abstract Landmark Chunks for Cognitive Robot Navigation." In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2566–71. IEEE. doi:10.1109/IROS.2005.1545556.

Lynch, Kevin. 1960. *The Image of the City*. Cambridge, Mass.: M.I.T. Press.

Mandelbrot, Benoit B. 1983. *The Fractal Geometry of Nature*. *American Journal of Physics*. Vol. 51. doi:10.1017/CBO9781107415324.004.

Mandelbrot, Benoit B., and John W. Van Ness. 1968. "Fractional Brownian Motions, Fractional Noises and Applications." *SIAM Review* 10 (4): 422–37. doi:10.1137/1010093.

Martins, Eernesto Queiros Vieira. 1984. "On a Multicriteria Shortest Path Problem." *European Journal of Operation Research* 16 (2): 236–45.

Miettinen, Kaisa. 1999. *Nonlinear Multiobjective Optimization*. Boston: Kluwer Academic Publishers.

Mitchell, Don P. 1991. "Spectrally Optimal Sampling for Distribution Ray Tracing." In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '91*, 25:157–64. New York, New York, USA: ACM Press. doi:10.1145/122718.122736.

Moazeni, Somayeh. 2006. "Fuzzy Shortest Path Problem with Finite Fuzzy Quantities." *Applied Mathematics and Computation* 183 (1): 160–69. doi:10.1016/j.amc.2006.05.067.

Moore, E. F. 1957. "The Shortest Path through a Maze." In *Proceedings of an International Symposium on the Theory of Switching*, 285–92. Cambridge, Massachusetts: Harvard University Press.

Musgrave, F. Kenton, Craig E. Kolb, and Robert S. Mace. 1989. "The Synthesis and Rendering of Eroded Fractal Terrains." *ACM SIGGRAPH Computer Graphics* 23 (3): 41–50. doi:10.1145/74334.74337.

Niazi, Muaz, and Amir Hussain. 2011. "Agent-Based Computing from Multi-Agent Systems to Agent-Based Models: A Visual Survey." *Scientometrics* 89 (2): 479–99. doi:10.1007/s11192-011-0468-9.

Okada, Shinkoh, and Timothy Soper. 2000. "A Shortest Path Problem on a Network with Fuzzy Arc Lengths." *Fuzzy Sets and Systems* 109 (1): 129–40. doi:10.1016/S0165-0114(98)00054-2.

Packard, Norman H., and Stephen Wolfram. 1985. "Two-Dimensional Cellular Automata." *Journal of Statistical Physics* 38 (5–6): 901–46. doi:10.1007/BF01010423.

Perez, Diego, Sanaz Mostaghim, Spyridon Samothrakis, and Simon M. Lucas. 2015. "Multiobjective Monte Carlo Tree Search for Real-Time Games." *IEEE Transactions on Computational Intelligence and AI in Games* 7 (4): 347–60. doi:10.1109/TCIAIG.2014.2345842.

Perez, Diego, Philipp Rohlfshagen, and Simon M. Lucas. 2012. "The Physical Travelling Salesman Problem: WCCI 2012 Competition." *2012 IEEE Congress on Evolutionary Computation, CEC 2012*, 10–15. doi:10.1109/CEC.2012.6256440.

Perlin, Ken. 1985. "An Image Synthesizer." *ACM SIGGRAPH Computer Graphics* 19 (3): 287–96. doi:10.1145/325165.325247.

Phillips, Joshua L., and David C. Noelle. 2005. "A Biologically Inspired Working Memory Framework for Robots." In *ROMAN 2005. IEEE International Workshop on Robot and Human Interactive Communication, 2005.*, 599–604. IEEE. doi:10.1109/ROMAN.2005.1513845.

Pollack, Maurice. 1960. "The Maximum Capacity Through a Network." *Operations Research* 8 (5): 733–36. doi:10.1287/opre.8.5.733.

Popescu, Mihail, and James M. Keller. 2012. "Implementing Bounded Rationality in Disaster Agent Behavior Using OGA Operators." In *2012 IEEE International Geoscience and Remote Sensing Symposium*, 5379–81. IEEE. doi:10.1109/IGARSS.2012.6352391.

Qingfu Zhang, and Hui Li. 2007. “MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition.” *IEEE Transactions on Evolutionary Computation* 11 (6): 712–31. doi:10.1109/TEVC.2007.892759.

Riera-Ledesma, Jorge, and Juan José Salazar-González. 2005. “A Heuristic Approach for the Travelling Purchaser Problem.” *European Journal of Operational Research* 162 (1): 142–52. doi:10.1016/j.ejor.2003.10.032.

Russell, Stuart, and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach, 3rd Edition*. *Prentice Hall*.

Shaker, Noor, Julian Togelius, and Mark J. Nelson. 2016. *Procedural Content Generation in Games*. Computational Synthesis and Creative Systems. Cham: Springer International Publishing. doi:10.1007/978-3-319-42716-4.

Silver, David, and Joel Veness. 2010. “Monte-Carlo Planning in Large POMDPs.” In *Advances in Neural Information Processing Systems (2010)*, 2164–72. http://discovery.ucl.ac.uk/1347369/.

Simon, Herbert A. 1955. “A Behavioral Model of Rational Choice.” *The Quarterly Journal of Economics* 69 (1): 99–118.

Skubic, Marjorie, David Noelle, Mitch Wilkes, Kazuhiko Kawamura, and James M. Keller. 2004. “A Biologically Inspired Adaptive Working Memory for Robots.” In *AAAI Fall Symp., Workshop on the Intersection of Cognitive Science and Robotics: From Interfaces to Intelligence*, 68–75. http://www.aaai.org/Papers/Symposia/Fall/2004/FS-04-05/FS04-05-010.pdf.

Smith, Noah A., and Roy W. Tromble. 2004. “Sampling Uniformly from the Unit Simplex.” *Johns Hopkins University, Tech. Rep*. http://www.cs.cmu.edu/~nasmith/papers/smith+tromble.tr04.pdf.

Tarapata, Zbigniew. 2007. “Selected Multicriteria Shortest Path Problems: An Analysis of Complexity, Models and Adaptation of Standard Algorithms.” *International Journal of Applied Mathematics and Computer Science* 17 (2): 269–87. doi:10.2478/v10006-007-0023-2.

Thrun, Sebastian. 2002. “Robotic Mapping: A Survey.” In *Exploring Artificial Intelligence in the New Millennium*. Morgan Kaufmann. http://robots.stanford.edu/papers/thrun.mapping-tr.pdf.

Togelius, Julian, Emil Kastbjerg, David Schedl, and Georgios N. Yannakakis. 2011. “What Is Procedural Content Generation? Mario on the Borderline.” In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games - PCGames ’11*, 1–6. New York, New York, USA: ACM Press. doi:10.1145/2000919.2000922.

Tolman, Edward C. 1948. “Cognitive Maps in Rats and Men.” *Psychological Review* 55 (4): 189–208.

Trivedi, Anupam, Dipti Srinivasan, Krishnendu Sanyal, and Abhiroop Ghosh. 2016. “A Survey of Multiobjective Evolutionary Algorithms Based on Decomposition.” *IEEE Transactions on Evolutionary Computation* 21 (3): 1–1. doi:10.1109/TEVC.2016.2608507.

Wang, Xuzhu, and Etienne E. Kerre. 2001a. “Reasonable Properties for the Ordering of Fuzzy Quantities (I).” *Fuzzy Sets and Systems* 118 (3): 375–85. doi:10.1016/S0165-0114(99)00062-7.

———. 2001b. “Reasonable Properties for the Ordering of Fuzzy Quantities (II).” *Fuzzy Sets and Systems* 118 (3): 387–405. doi:10.1016/S0165-0114(99)00063-9.
Warshall, Stephen. 1962. “A Theorem on Boolean Matrices.” *Journal of the ACM* 9 (1): 11–12. doi:10.1145/321105.321107.
Yager, Ronald R. 1988. “On Ordered Weighted Averaging Aggregation Operators in Multicriteria Decisionmaking.” *IEEE Transactions on Systems, Man, and Cybernetics* 18 (1): 183–90. doi:10.1109/21.87068.
Yang, Shengxiang, Miqing Li, Xiaohui Liu, and Jinhua Zheng. 2013. “A Grid-Based Evolutionary Algorithm for Many-Objective Optimization.” *IEEE Transactions on Evolutionary Computation* 17 (5): 721–36. doi:10.1109/TEVC.2012.2227145.
Zadeh, L. A. 1963. “Optimality and Non-Scalar-Valued Performance Criteria.” *IEEE Transactions on Automatic Control* 8 (1): 59–60. doi:10.1109/TAC.1963.1105511.
———. 1965. “Fuzzy Sets.” *Information and Control* 8 (3): 338–53. doi:10.1016/S0019-9958(65)90241-X.
———. 1975a. “The Concept of a Linguistic Variable and Its Application to Approximate reasoning—I.” *Information Sciences* 8 (3): 199–249. doi:10.1016/0020-0255(75)90036-5.
———. 1975b. “The Concept of a Linguistic Variable and Its Application to Approximate Reasoning—II.” *Information Sciences* 8 (4): 301–57. doi:10.1016/0020-0255(75)90046-8.
Zare, Alina, Zachary Fields, James M. Keller, and Joshua Horton. 2012. “Agent-Based Rumor Spreading Models for Human Geography Applications.” In *2012 IEEE International Geoscience and Remote Sensing Symposium*, 5394–97. IEEE. doi:10.1109/IGARSS.2012.6352387.
Zeleny, M. 1973. “Compromise Programming.” In *Multiple Criteria Decision Making*, edited by J Cochrane and M Zeleny, 262–301. Columbia: University of South Carolina Press.
Zhao, Yanli, Anand Padmanabhan, and Shaowen Wang. 2013. “A Parallel Computing Approach to Viewshed Analysis of Large Terrain Data Using Graphics Processing Units.” *International Journal of Geographical Information Science* 27 (2): 363–84. doi:10.1080/13658816.2012.692372.
Zhou, Aimin, Bo-Yang Qu, Hui Li, Shi-Zheng Zhao, Ponnuthurai Nagaratnam Suganthan, and Qingfu Zhang. 2011. “Multiobjective Evolutionary Algorithms: A Survey of the State of the Art.” *Swarm and Evolutionary Computation* 1 (1). Elsevier B.V.: 32–49. doi:10.1016/j.swevo.2011.03.001.
Zitzler, Eckart, Marco Laumanns, and Stefan Bleuler. 2004. “A Tutorial on Evolutionary Multiobjective Optimization.” In *Metaheuristics for Multiobjective Optimisation*, 3–37. doi:10.1007/978-3-642-17144-4_1.

VITA

Andrew (Drew) Robert Buck grew up in Kansas City, MO and graduated as a valedictorian of the Park Hill HS class of 2004. He attended the University of Missouri in Columbia, MO and earned Bachelor of Science degrees in Computer Engineering and Electrical Engineering in 2009. During his final undergraduate year, he began working with Dr. James Keller on a research project titled "Text-to-Sketch," which convinced him to pursue a graduate education at the University of Missouri. In 2012, he earned a Master of Science degree in Computer Engineering and continued to work toward a Ph.D. in Electrical and Computer Engineering. Following the conclusion of the "Text-to-Sketch" project, he joined the human geography research group and studied models of agent decision-making behavior, which led to the present work. He currently holds a research assistantship and performs target detection on side-looking 3D radar imagery.

Andrew has presented at multiple international conferences and earned numerous awards, including the Donald K. Anderson Graduate Research Assistant Award, Outstanding Ph.D. Student Award, and Outstanding Masters Student Award. He has coauthored several published research papers, winning best paper and best student paper awards at two IEEE symposiums. Throughout his undergraduate career, he played the trombone in Marching Mizzou and other university ensembles, and continues to play with the Columbia Community Band. His interests include computational intelligence, spatial reasoning, visualization, robotics, games, and the outdoors.