Online Path Planning for Multi-Robot Coverage

Kevin Bradner

Department of Electrical Engineering and Computer Science

Case Western Reserve University

January, 2020

Contents

1 Introduction **1**

1.1 Robotic Coverage Background 1

1.1.1 Spanning-Tree based Coverage Algorithms 3

1.1.2 Multi-Agent Robotic Coverage 9

1.2 Problem Statement 12

1.2.1 Rules for Robot Motion 14

1.2.2 Environment Description 16

1.2.3 Field of View and Information Sharing 18

1.2.4 Relation to Robotic Mapping 19

1.2.5 Applications and Other Notes 22

2 Simulation Development **24**

2.1 Implementation of Problem Statement 25

2.1.1 Simulated Environment & Dynamics 25

2.1.2 Robot and Ensemble Representation 27

2.1.3 Environment Interface 29

2.1.4 Environment Generation 32

2.1.5 Running Simulations 36

2.1.6 Visualization Tools and Parsing 38

2.2 Extra Features 43

2.2.1 Drone Dropout 43
2.3 Operation and Software Utilities 44

3 Simple Coverage Policies **47**
3.1 The Environment-Policy Match up 47
3.1.1 Two Simple Policies 47
3.1.2 High Sweep First Policy 48
3.1.3 Low Sweep Policy 51
3.1.4 Performance on an Alternate Environment 51
3.1.5 Discussion 52
3.2 The Low Sweep Policy 54
3.2.1 Behavior of the Low Sweep Policy 55
3.2.2 The A-Star Algorithm 56
3.2.3 Optimizations of the Low Sweep Policy 59
3.3 The High Sweep First Policy 60
3.3.1 Details of the HSFP 60

4 Intelligent and Multi Agent Coverage Policies **63**
4.1 The Clustering Low Policy 64
4.1.1 K Means Clustering 64
4.1.2 Details of the Clustering Low Policy 66
4.2 A Single Agent Spanning Tree Policy 68
4.2.1 Depth First Spanning Tree Generation and Path Generation Algorithms 69
4.2.2 Details of the Low Spanning Tree Policy 72
4.3 The Clustering Low Spanning Tree Policy 72
4.4 Persistent (Learning) Policies 73

5 Conclusion **74**

5.1 Performance Comparison 74

5.2 Performance Comparison on Simple Environments 75

5.3 Performance Comparison on Complex Environments 75

5.4 Drone Dropout Performance Comparison 75

5.5 Efficiency of Drone Utilization 75

5.6 Future Work 75

Bibliography **77**

List of Tables

List of Figures

1.1 Boustrophedon Decomposition 3
1.2 Spanning Tree Example 4
1.3 Spanning Tree Coverage Path 5
1.4 Neighbor Visiting Behavior for Online STC 7
1.5 Path Deformations in Scan-STC 8
1.6 MSTC Path Following Behavior 10
1.7 Multi-Robot Forest Coverage Path Creation 11
1.8 Odometry Errors and Corrected Map 21

2.1 Sample Environment Visualization 25
2.2 Visualization of a Small Environment During Coverage 41

3.1 Tiny Square Environment 48
3.2 Partially Explored Environment 49
3.3 Complete information without complete coverage 50
3.4 Fully Covered Environment 50
3.5 Long Thin Environment 52
3.6 Minimal Environments with Uncertain Policies 53
3.7 Environment with non-continuous columns 56

Acknowledgments

Abstract

Robotic coverage problems task one or more robots with the goal of visiting every location in a region. Algorithms that can perform this kind of task in a time efficient manner are useful for purposes such as mapping, cleaning, or inspection. This work considers a multi agent robotic coverage problem in which the shape of the region to be explored is known in advance, but additional information and challenges are discovered during task performance. A software package is created to simulate such a scenario, generate virtual environments to be covered, and interface with policy programs that command the robots. Offline path planning algorithms are developed, and their performance on this task is evaluated. Next, online variants of these algorithms are developed to respond to events and information encountered during task execution. It is shown that the online algorithms are more robust and better performing than their offline counterparts.

Chapter 1

Introduction

1.1 Robotic Coverage Background

Robotic motion planning problems are typically concerned with finding an achievable path between two robot configurations. These configurations could be the current location of a robotic vehicle along with its desired destination, the current pose of a robotic arm and the pose required for it to grasp an object, or any other pair of robot states that correspond to "start" and "goal" configurations. *Robotic coverage* problems are another kind of motion planning problem. The goal of a robotic coverage problem is to find a path for the robot to follow during which it will visit every location in a region. This region could be an area of floor to be vacuumed, the surface of a car body that needs to be painted, or the interior of a building that needs to be mapped. In all of these cases, the task is only complete once the entire region has been visited, or *covered* by a robot. Usually, another goal of these problems is to achieve coverage in a way that minimizes the time or number of movements required to complete the task. The robotic coverage problem can be viewed as a continuous generalization of the traveling salesman problem [3]. Thus, the general form of the problem is NP-Hard. Applications for robotic cover-

age continue to be developed today. For example, work by Daltorio et al. from 2010 develops control policies necessary to achieve "obstacle edging" behavior [8]. This work effectively implements ideas from algorithms that perform online coverage of an unknown environment, discussed shortly.

One particularly well studied type of robotic coverage problem is the case where the coverage region is some subset of the plane, and the robots are mobile robots that can only move in the plane. A good overview of this kind of coverage problem can be found in [5]. Algorithms to perform robot coverage in the plane have been studied since the late 1980's. Early algorithms for this purpose relied on the use of heuristics, and these algorithms usually could not guarantee that complete coverage would be achieved for all possible environments. By around 2000, several planar coverage algorithms were able to guarantee complete coverage for a variety of simple environment types. Many of these algorithms proved completeness by showing that the algorithm breaks down the coverage space into smaller regions that are relatively trivial to cover completely. Coverage algorithms that employ this kind of technique are known as *cellular decomposition* algorithms.

One type of cellular decomposition fits a square grid to the coverage space. Because a general arbitrary set in the plane can not be exactly expressed as a union of grid-aligned square regions, this technique is often called approximate cellular decomposition. Typically, each cell in this kind of decomposition can be covered entirely by a single robot configuration. For example, consider dividing a grass lawn into squares for a robotic lawn mowing task. If each square in the cellular decomposition is small enough, a lawnmower whose center is aligned with a square's center will have mowed that entire square of the lawn. In coverage problems where this assumption applies, proving that a coverage path visits each cell in the approximate decomposition is sufficient to guarantee complete coverage.

In addition to approximate cellular decomposition, there are also techniques that

decompose a coverage space into a more diverse collection of cell shapes. Examples of such shapes include cells of fixed width whose top and bottom boundaries can vary in shape [16], or trapezoidal cells whose parallel sides all run in the same direction [6]. Depending on whether the union of these cells is exactly equal to the coverage space, such techniques are called either *semi-approximate* or *exact* cell compositions. In the case of trapezoidal cell decomposition, it is possible to combine the trapezoidal cells into a smaller number of large cells, each of which are able to be covered by a simple back-and-forth sweeping motion. This approach is known as a *boustrophedon* decomposition [6].

(a) Two new cells created at critical point **(b)** Transition from two cells to one new one

Figure 1.1: Boustrophedon cell creation at critical points

1.1.1 Spanning-Tree based Coverage Algorithms

When coverage is performed by a square shaped tool attached to a mobile robot, and when the coverage area can be exactly decomposed into a grid of squares that have double the side length of the robot tool, it is possible to compute an optimal covering path in linear time using an approach based on spanning trees [12]. Gabriely and Rimon developed several variants of this approach with different time, memory, and prior knowledge requirements. In all cases, it was assumed that a mobile robot would complete the coverage task with a square shaped tool of size D. It was also assumed that the tool could only move in the four cardinal directions when completing the task. Finally, it was assumed that the region to be covered could be approximated by cells in a grid of squares with size $2D$. The authors argue that

the final assumption is justified for realistic environments in which the size of the robot's tool is significantly smaller than the dimensions of the area to be covered. Under these assumptions, the *Spanning Tree Covering* (STC) algorithm is able to generate a path through the space which visits each grid-aligned square cell of size D exactly once.

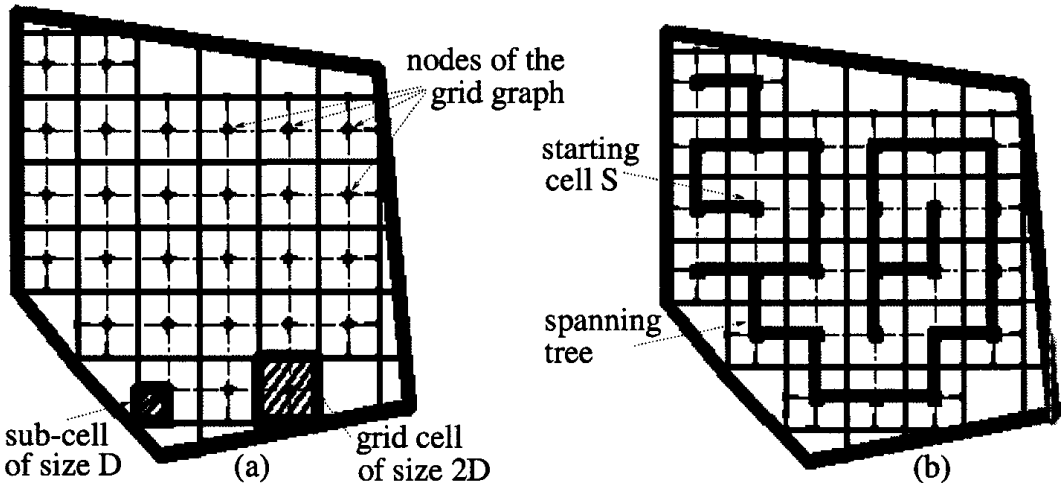


Figure 1.2: A small coverage environment and a spanning tree on that environment [12].

The offline variant of STC works by representing the coverage area as a graph G. Nodes on this graph represent the center of a square with size $2D$, and the set of nodes corresponds to the set of full cells when a grid with square size $2D$ is overlaid on the coverage space. Edges exist between any two nodes whose corresponding cells are adjacent in the coverage space. As the name suggests, STC's key step is the creation of a spanning tree on G, using any node in G as the root of the tree. Many algorithms exist to compute spanning trees, but two popular examples are based on breadth-first-search and depth-first-search [7].

Once this spanning tree has been created, the coverage area is subdivided into squares of size D. Each node of G coincides with the corners of four different squares in this subdivided map. Starting at any one of these squares, each of which is exactly the shape and size of the robot's coverage tool, the robot can simply move counterclockwise around the spanning tree as if it were performing wall following.

In this way, each cell in the coverage space will be visited exactly once by the robot. Because of this property along with the fact that the path generated by STC ends in a position adjacent to where it started, the path found by STC can be considered a Hamiltonian cycle on G. Although constructing a Hamiltonian path on a general planar grid is NP-complete [1], the assumed properties of G make it practical to compute such a path on this graph in linear time. The fact that this coverage path's start and end positions will be in adjacent cells can be beneficial in practice, as it allows deployment and retrieval of the robot to happen in approximately the same place.

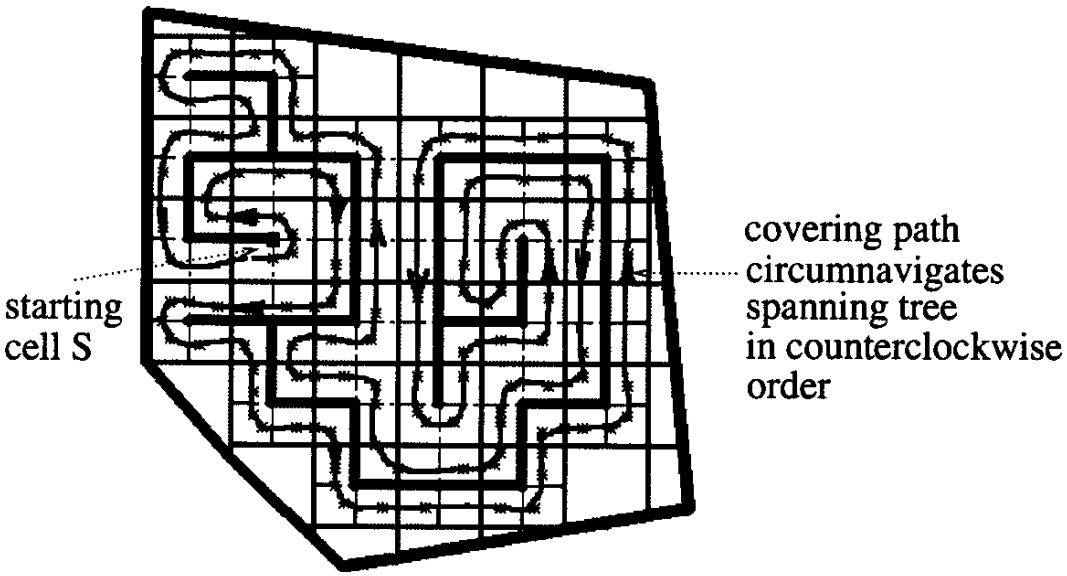


Figure 1.3: Following the spanning tree of G achieves exact coverage and creates a cycle [12].

This algorithm has a few interesting variants. The first runs online, and effectively circumnavigates a depth first search spanning tree that it constructs on the fly [12]. Restriction to use of depth first search is the primary disadvantage of this approach compared to offline STC. While the coverage paths based on depth first search are equivalent to those created through any other method in terms of final path length, other spanning tree algorithms have practical benefits. For example, spanning tree algorithms such as Prim's algorithm or Kruskal's algorithm can cre-

ate minimal spanning trees on weighted graphs. For details on Prim's algorithm and Kruskal's algorithm, refer to [7]. Setting the weights in a particular way can lead to coverage paths with desirable properties. For example, a minimal spanning tree on a graph where North-South edges are given less weight that East-West edges will tend to move in long, parallel lines that minimize turning. In later work, Gabriely and Rimon develop an online approach called Scan-STC that achieves similar behavior. Scan-STC also makes additional improvements on the original STC as discussed below.

The second online variant of the STC algorithm is described by Gabriely and Rimon as *ant-like*. In this case, the robot must be able to mark visited sub-cells through some means such as leaving pebbles on the ground. In addition, the robot must be able to sense the presence of these makers in the nodes of G adjacent to its current position. By marking each visited sub cell as the robot leaves that cell, the robot is able to proceed on the same path it would have created in the depth first search based online approach. Because this algorithm uses markers to identify visited nodes, it requires only O(1) memory. However, it requires the use of O(N) markers, where N is the number of sub-cells in the coverage area. Nonetheless, this algorithm presents interesting ideas for multi-agent approaches and prevention of drift in pose estimation. In both of these cases, physical and immobile records of a robot's past location can inform the decision making of any robot currently near some of those records.

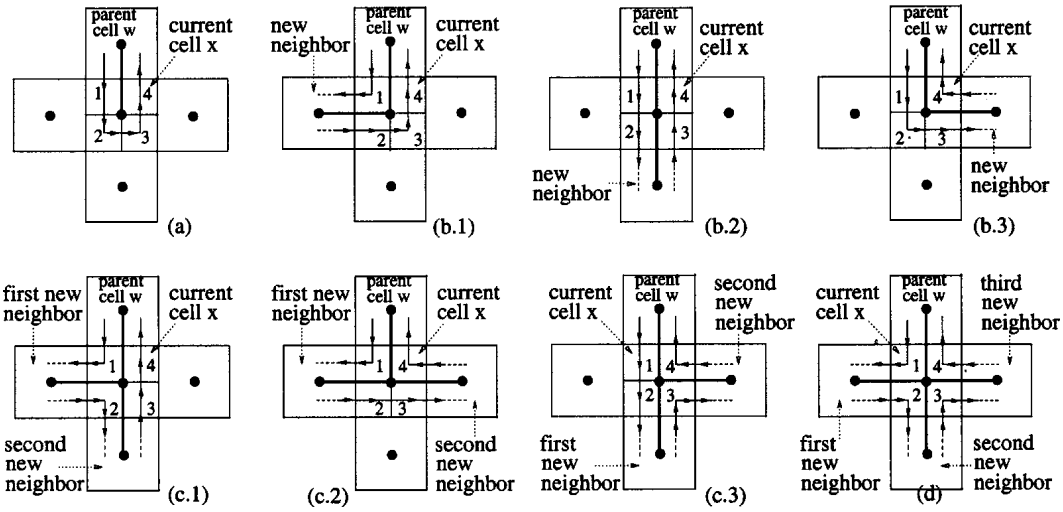


Figure 1.4: Online STC variants determine their behavior according to current position and whether neighbors have been visited before [12].

Finally, there is a variant of STC that achieves *exact* coverage of an environment, even when that environment can not be represented by completely filled cells on a grid of squares with size $2D$. However, this approach still requires that the connectivity of the environment is represented adequately by such a grid. The resulting algorithm simply notices any cells that are partially occupied by a previously unseen obstacle when it passes next to them in the course of normal online STC. When such a cell is encountered, a neighborhood is swept around the entire obstacle. Then, the robot returns to traversing the fully in-bounds cells according to its usual procedure. The ability to complete this sweeping motion is always guaranteed under assumptions used to model this problem. For details concerning the size of this neighborhood and the correctness of the resulting algorithm, see [12].

In the previously mentioned Scan-STC algorithm, Gabriely and Rimon improve STC to explicitly handle boundary cells, or graph cells that are partially occupied by an obstacle. Using the same notation as before, this algorithm generates a path with length no greater than $(n+m)D$, where n is the number of nodes in G, and m is the number of boundary cells [13]. The authors assume that no node will contain an arrangement of obstacles such that the graph of that node's sub-nodes is

disconnected. If an obstacle exists such that the normal grid of nodes would violate this assumption, the corresponding location in the coverage area is assigned two different nodes. With that assumption in place, avoidance of obstacles is done by simply displacing the normal path that goes around the obstacle. This displacement tactic is what leads to the m term appearing in the maximum path length bound. The authors claim that in practical environments, the path length is closer to $n + \frac{m}{2}$.

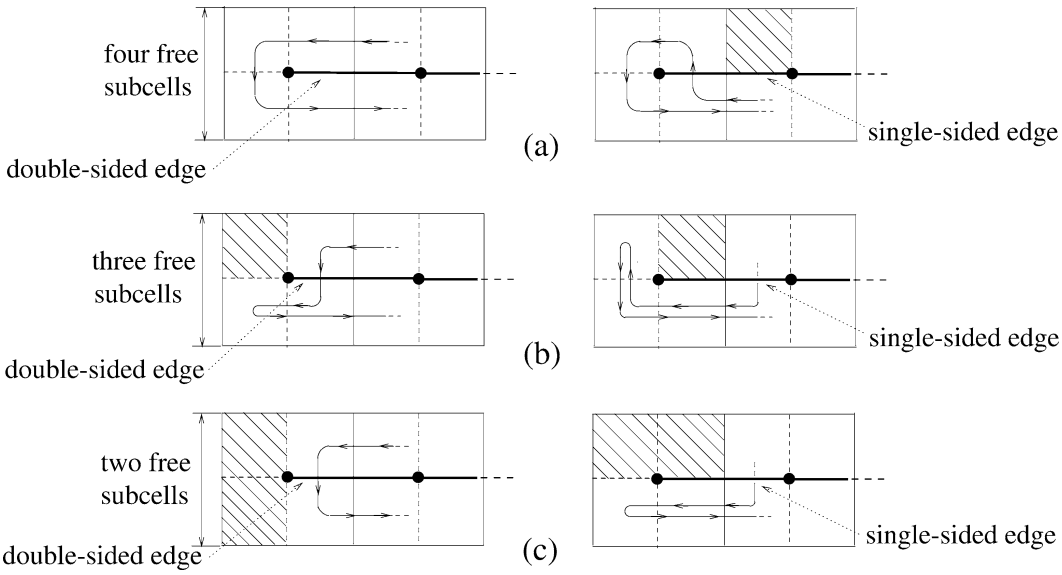


Figure 1.5: Scan-STC adapts its motion in a variety of situations with partially occupied cells [13].

The other significant insight of Scan-STC relates to its name. By allowing the mobile robot to sense the state of the eight cells the surround its current position, it is possible to discover that one of the horizontal neighbors of the current cell has a free vertical neighbor, and that the current cell has a free vertical neighbor in the same direction. In this case, it is guaranteed that a path into the node's horizontal neighbor exists such that the vertical neighbors of these cells appear between the current cell and its horizontal neighbor. With this insight, it is possible to skip the creation of a horizontal edge between the current node and its horizontal neighbor. The benefit of this approach is that it leads to coverage paths that move primarily in the vertical directions. This can be useful for limiting the amount of

turns performed, thereby reducing the number of opportunities to introduce dead reckoning error from imprecise turns.

1.1.2 Multi-Agent Robotic Coverage

Another imprint feature of a robotic coverage algorithm is whether that algorithm controls a single robot or a group. Multi-agent robotic coverage algorithms address the latter case. Multi agent approaches to coverage can often complete a coverage task several times faster than a single agent approach because of the division of work that use of multiple robots enables. Use of multiple robots can also enhance robotic coverage by allowing for more precise localization through information sharing. Finally, multi-agent methods are often able to adapt to the failure of one or more robots, as there may be remaining robots capable of completing the coverage task [5]. As robotic hardware decreases in price, taking full advantage of these benefits is increasingly viable in terms of up-front costs. However, multi-agent techniques often require more complex motion planning strategies to coordinate the motion of multiple robots in a way that takes full advantage of these potential efficiency improvements and other benefits.

A large amount of research adapts the spanning tree approach to the case of multi-agent coverage [19][15]. In their work on this subject, Noam Hazon and Gal Kaminka generalize the STC algorithm to the multi agent case, resulting in an algorithm called *Multirobot Spanning-Tree Coverage*, or MSTC. The MSTC problem assumes that the coverage area can be adequately approximated by a grid of square cells with side length $2D$, as in the spanning tree coverage algorithm. As with the *original* version of STC, it is assumed that all cells in the coverage space are free of obstacles. It is also assumed that all robots have the same speed and tool size. A version of this assumption, that all robots are the same, applies to most of the work on multi-agent robotic coverage.

With these assumptions in place, the first version of MSTC simply constructs a spanning tree of the space offline, then has all of the drones start to follow the spanning tree from their initial positions, much the same as single agent STC. If the robots are able to share information about their status, i.e. working or broken, and whether they have completed their assigned section, then it is possible to have all working robots continue to follow the spanning tree path until the coverage task is complete. Under these assumptions, complete coverage is guaranteed as long as one robot remains in working order until the end of the task.

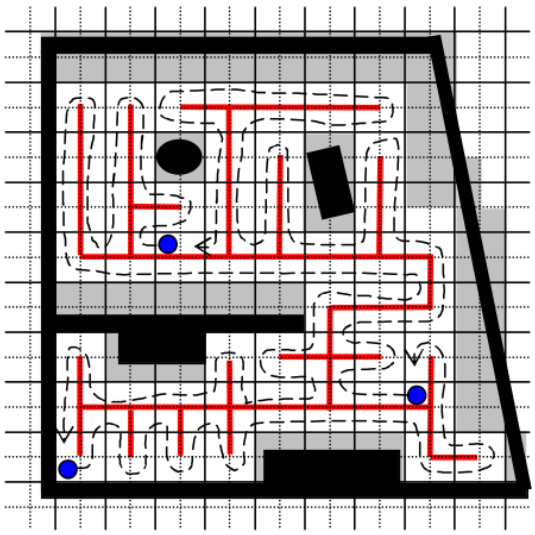


Figure 1.6: The MSTC algorithm assigns portions of a shared loop to each drone based on its starting position [15].

If there are k robots exploring a coverage space with n sub-cells as described above, then the best case time for this algorithm is shown to be $\frac{n}{k} - 1$. This performance is achieved when the starting points of the robots are distributed such that each robot is evenly spaced along the length of the spanning tree coverage path. The worst case is stated to be $n - k - 1$, for the case where all k robots start on k adjacent sub-cells that appear together in the spanning tree graph. Implicit in both bounds is the assumption that no robots fail. In a pathological case with the same initial conditions used to give the previously stated worst case bound, the $k - 1$ robots that

start directly in front of the last robot could all fail at time $n-k-2$, just before the front robot reached the back robot's position. In this case, it would take another $n-1$ steps for the remaining robot to complete the coverage task according to Algorithm 2 as described by the authors, leading to a total coverage time of $2n-k-2$. The authors note that many practical multi-robot coverage scenarios require starting all robots from the same location, meaning that performance close to the worst case bound $n-k-1$ is relatively common. This motivates the development of a more sophisticated coverage algorithm. By assuming that backtracking is allowed, the authors develop an algorithm with a worst case of $\frac{n}{2}-1$ for $k>2$. However, because all algorithms in [15] only consider the possibility of moving forward and backward along the initially calculated spanning tree path, many opportunities for further improvement remain.

Work by Zheng et al. surpasses the average-case performance of MSTC with their *Multi-Robot Forest Coverage* (MFC) algorithm [19]. As the name suggests, MFC computes a separate spanning tree for each of the robots working on the coverage task. For convenience, the authors of MFC also assume that multiple robots can occupy the same sub-cell simultaneously, and that the coverage area can be represented by grid-aligned cells with side length twice that of the robot's coverage tool. Experimental results showed that MFC achieved performance close to a theoretical lower bound in many cases.

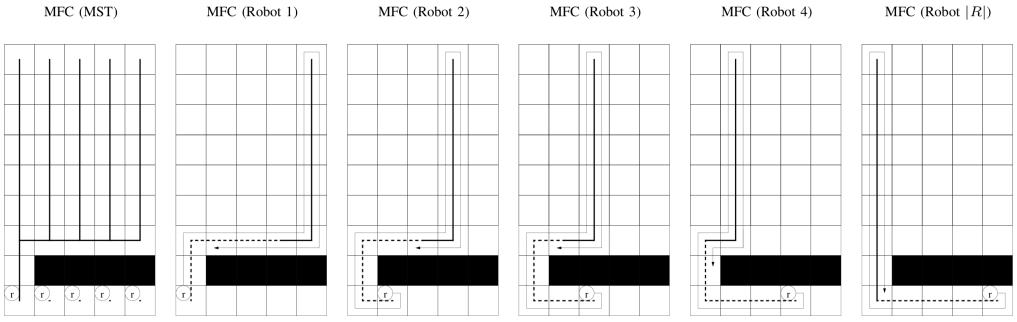


Figure 1.7: MFC assigns a different spanning tree to each robot.

In addition, Zheng et al. were able to prove that their algorithm's cover times are no worse than eight times the optimal value, thanks to properties of a tree cover algorithm for general graphs that they adapted to this purpose. The tree cover algorithm, developed by Even et al, finds a forest of $|R|$ trees that start from a set R of root positions and cover an entire graph $G = (V, E)$. This algorithm attempts to minimize the largest weight of its trees. The algorithm runs in polynomial time and finds forests such that the maximum tree weight is no more than four times larger than optimal. Zheng et al. use a lightly altered version of this algorithm to determine the individual trees assigned to each robot in the space (Figure 1.7).

1.2 Problem Statement

As with most of the previously discussed research, this work presents multi-agent coverage algorithms based on spanning trees. The details of the coverage problem stated here vary somewhat from other work - these differences are motivated by a use case in which a team of aerial drones (UAVs, robots) must use cameras or other 2D imaging sensors to perform coverage of an area affected by disaster. Practical applications of this work would employ a combination of human effort and image recognition / processing techniques to identify people in need of aid, key structural damage that requires immediate repair, or other points of interest. Once any such point of interest has been identified and located, the relevant information can be used to guide response efforts.

Two particularly novel features are introduced to fit this setting:

1. Robots that are able to change the size of their field of view
2. Environments with subsections that require different levels of scrutiny

These two changes are closely related to one another. Specifically, the varying scrutiny requirements motivates the use of techniques that trade off between an

observation's field of view and its level of detail. In turn, concrete motivation for an environment that requires varying levels of scrutiny is based on the idea that not all areas are worth viewing in great detail when gathering information for a search and rescue effort. For example, some areas of the coverage environment may consist of a fairly uniform surface such as a body of water or a large area of flat, rocky terrain. Such areas may not be worth covering in detail because their uniform nature makes it easy to conclude that nothing of interest exists in these regions. At the opposite extreme, trying to observe a region with dense tree cover may not be practical in a coverage scenario constrained by an urgent timeline. In contrast, it may be worth looking closely at areas with partial occlusions, areas with visually complex damaged structures, or places where people are particularly likely to be in need of help.

As with the identification of actual people and other points of interest, the visual identification of which regions are likely to contain useful information that can be practically observed is abstracted away by this work. In its place, this work assumes access to a perfectly accurate sensor that describes the level of scrutiny required to view each environment cell in a drone's field of view. Another assumption of this problem is that the distribution of locations requiring one level of scrutiny or another is not known in advance of coverage. In practice, it may not be possible to know the exact location of collapsed structures, flooded regions, or other environment features that may appear without warning in an emergency.

The other significant novel feature of this problem formulation, robots with a variable field of view, arises naturally as a strategy to adapt to the varying scrutiny requirements of an environment. In this setting, the dimensions of a drone's field of view corresponds to the size of a robot's covering tool in a more traditional presentation of robotic coverage. This work assumes that a drone's field of view is always centered on a point directly below the drone, roughly corresponding to

down-facing camera mounted on the bottom of a drone. Real drones can usually alter the size of a ground area being photographed. For example, increasing the size of the observed area may be achieved by ascending with a constant viewing angle or by using an optical camera zoom feature to increase the viewing angle. Likewise, decreasing the field of view may be achieved by descending, zooming in, or by some combination of these.

Increasing the field of view has the obvious benefit of allowing a robot to cover more of its environment at once. Decreasing the field of view increases the detail with which a drone observes the area directly below it, and this allows high detail viewing of those areas for which a highly detailed view has been deemed useful. While the distribution of environment sections that require low vs high scrutiny is unknown a priori, it is assumed that the boundaries of the coverage area and of the obstacles that border it are known in advance.

1.2.1 Rules for Robot Motion

As previously stated, each drone in this formulation has access to moves that increase and decrease its field of view. For simplicity, it is assumed that a drone's field of view can be in only two states that correspond to two possible altitudes. A drone flying low can view a single cell in the environment's approximate cellular decomposition. A drone flying high can view a 3x3 grid of these environment cells. Depending on which altitude the drone currently occupies, it has access one of two vertical motions: ascend and descend. A somewhat arbitrary assumption of this work is that these vertical motions each take about the same length of time as moving cardinally between the centers of adjacent environment cells.

Unlike most other work on spanning-tree based coverage, this work assumes that the robots can move along diagonals, rather than only in the four directions orthogonal to the sides of its tool. For convenience of notation, the four directions

commonly used by spanning tree robots will be referred to as *cardinal directions*, and the four directions that are offset by $\frac{1}{8}$ turn from the cardinal directions will be called intercardinal directions. In addition, concrete examples shown in this work may refer to a particular cardinal direction such as East, or a particular intercardinal direction such as South-West (SW).

One reasonable concern about the introduction of diagonal motion in a world represented by an approximate cellular decomposition is how it affects the connectivity of the coverage area. As with many coverage techniques that employ an approximate cellular decomposition of the environment, this work assumes that coverage paths will move directly from the center of one cell (node, patch, location) to the center of one of its neighbors. In addition, it is assumed that the robots are able to move directly between cells that share only a corner, even if one or both of the adjacent cells that would connect them are out of bounds. Since other work typically assumes that the robot is roughly the same size as the area it covers, this assumption requires justification. Because the problem explored here is motivated by a team of autonomous drones observing an area from above, it is safe to assume that the robots are significantly smaller than the area they are actively observing at any particular moment. In addition, the set of in-bounds cells is assumed to represent any cells that are completely covered by terrain to be observed, and so out of bounds cells may correspond to areas that are not *completely* obstructed. Finally, due to the ability of aerial drones to move in three dimensions, it is often possible for a drone to fly over any obstacles that aren't too close to the ground, even if these obstacles completely occlude the ground. As a result of these considerations, it seems justifiable to assume that the robots are able to move between diagonally connected in-bounds cells, as there is likely to be enough room for a drone to complete such a move in a realistic scenario.

1.2.2 Environment Description

In this work, the author assumes that all robots start out in the same location. It is possible that this assumption makes the work less general than other work that allows for arbitrary starting locations, such as the multi robot coverage work of Hazon and Kaminka [15]. However, starting all robots in one location is often a practical assumption, as it allows for easy deployment of robots from a single drop-off point. It is also worth noting that starting each robot in the same place often makes for a more challenging coverage task than starting with robots distributed throughout the space. This is because spatially distributed robots may already be close to evenly dispersed along a single Hamiltonian cycle through the coverage space, and so moving to this extremely convenient state may be a low cost operation. Note that in order to take advantage of this situation with the highest probability, robots must be able to move freely between their starting position and an idealized starting position for spanning tree following. This initialization step is not addressed by the MSTC algorithm or the MFC algorithm, and the MSTC algorithm in particular will often perform close to its worst case bound when it is necessary to deploy the robots near each other [19][15]. Most of the coverage algorithms developed in this work are also applicable to the case where each robot gets a different starting position. With these points in mind, assuming that all robots start in one location is unlikely to limit the usefulness of this work in practical applications.

Unlike many of the efforts to adapt spanning tree coverage to a multi agent case, this work allows for environments that can't be cleanly represented as a collection of obstacle-free cells twice the size of a robot's coverage footprint. This assumption complicates the issue of creating and traversing spanning trees significantly. However, it is likely to be an important practical assumption for many realistic tasks. For example, in a search and rescue setting, people may be trapped around collapsed structures and other difficult to navigate corners of the coverage area. The capabil-

ity to plan spanning tree paths through such complex environments is also useful for the case where a robot has viewed an area with low scrutiny and must now revisit a small portion of that area that requires high scrutiny. Although it is not necessary to forbid revisiting locations that have already been adequately covered, a good path planning strategy should not visit these locations unless doing so is the most efficient way of covering the locations that do require a visit. In addition, it may be known in advance that certain areas of the environment are simply not worth examining at all. For example, it is unnecessary for a drone to scout out areas where human search and rescue teams are already planning to visit due to ease of access or a likelihood of finding people there. As in the case of previously visited locations it is not strictly necessary to treat these areas as obstacles. However, any technique that can avoid visiting them without reducing the efficiency of its exploration of the rest of the environment will save time and perform better than if those areas were considered in-bounds. Finally, the drones that motivate this research have a field of view that is much larger than the tool sizes of robots used for cleaning, de-mining, or other popular applications of coverage. As a result, assuming that the environment contains only open spaces with at least four times this area is not realistic.

With these caveats in mind, the environment model used here is very similar to the models used in other research on spanning tree based coverage. It consists of a set of grid-aligned square cells that require coverage. This region is always finite, connected, and enclosed by a boundary that forms a single loop around the environment. As discussed previously, an arbitrary subset of the cells in an environment may be considered out of bounds. Because aerial drones are likely to occupy only a small footprint inside the three dimensional airspace over the relatively large region they are observing, it is assumed that multiple drones can occupy the same environment cell simultaneously without crashing. As discussed shortly, this does

not present an issue from the perspective of being centered over the observed cell(s) because of information sharing.

1.2.3 Field of View and Information Sharing

Like much of the work that addresses multi-agent robot coverage, this work assumes that the robots share information while performing coverage. This assumption avoids a significant and complex barrier to implementing the described algorithms on real robots. However, because this work is motivated by a use case in which the robots are substantially smaller than their coverage area, and because an approximate cellular decomposition is used to describe the set of locations in the coverage area, it is possible to communicate full information between the robots in this scenario using only small and relatively infrequent messages. In addition, realistic flying robots are almost always equipped with long range communication devices to allow manual control by a human operator and to transmit observations in real time. For a search and rescue scenario in particular, it seems likely that the drones would be equipped with the ability to transmit high resolution photos at the same rate that coverage information is passed, in which case passing key planning information would require only a small fraction of the drone's available bandwidth. With these considerations in mind, it seems safe to assume that the level of communication required to share planning information between drones is at least possible in a realistic scenario.

Specifically, it is assumed that all drones have access to the following information at all times:

1. A map of the in bounds locations in the coverage region
2. The current location of every drone participating in the coverage task

3. A partially filled map in which any previously observed locations are marked and classified as requiring high or low scrutiny

4. Information about which previously seen locations were not observed in adequate detail and must be revisited from a low altitude

In addition, it is assumed that a central controller implements all motion planning, so a global multi robot motion plan is implicitly maintained at all times. For all intents and purposes, this information can also be thought of as shared.

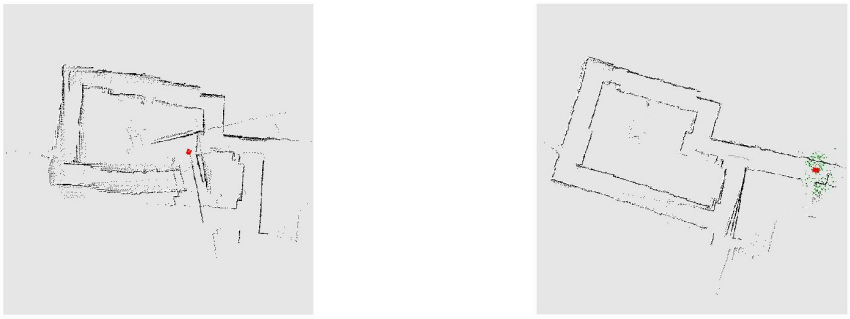
Because this problem statement allows for more types of robot state changes than the previously described work, it is worth addressing the optimization criteria and cost function for this task. Much of the existing work on multi agent coverage aims to find a set of robot paths that complete the coverage task while minimizing the length of the longest path taken by any of the robots. In this work, it makes more sense to assign time costs to all of the moves available to the drones. Then, the optimization goal for this task is to achieve complete coverage in the least possible time. It is assumed that ascending and descending each take as long as a cardinal motion between adjacent environment cells. Intercardinal motion between cells that share a vertex is assumed to take a factor of $\sqrt{2}$ more time than cardinal motions in order to correspond with the greater euclidean distance between the centers of these cells.

1.2.4 Relation to Robotic Mapping

Whereas many applications of robotic coverage involve mechanical interaction with the environment being covered (e.g. vacuum cleaning, humanitarian mine removal), this work addresses a use case where the robots are observing an environment with cameras or other sensors. As a result, it bears some resemblance to the robotic mapping problem. Despite this meaningful similarity, this work focuses on simply

visiting the entirety of an environment. The imagined application of this behavior in a search and rescue scenario is to capture anyone in need of help on camera. From there, it would be necessary to employ human effort or a complex image recognition tool to identify these people and send help. Note that providing this kind of information does not imply any of the other steps involved in creating an accurate map of some area.

This focus on online path planning for abstract environments turns out to have very little to do with the main considerations of robotic mapping research. In a survey of the topic, Thrun provides this definition: "Robotic mapping addresses the problem of acquiring spatial models of physical environments through mobile robots" [29]. In a world of perfect robots with completely reliable sensors and actuators, solving this problem might be a straightforward extension of the behavior of the online coverage algorithms discussed so far. Real robots are not perfect, though, and this is particularly true in applications where the robots must be inexpensive in order for the target application to be cost effective. As a result, approaching the robot mapping problem without accounting for noise in sensors and actuators leads to extremely bad performance. For example, odometry is one of the most widely used techniques for keeping track of a robot's position in space as it takes local measurements during a mapping operation. In principle, it is possible to determine a robot's precise location relative to some starting point by tracking all of its actuated motions and adding up the corresponding displacements in space. With noisy odometry, small errors in the robot's estimate of position and orientation can cascade, with one error affecting the accuracy of all of the measurements that follow. As the robot travels farther from its initial position, small errors in estimated orientation can be particularly damaging to the accuracy of the resulting map.



(a) Noisy odometry affects map quality **(b)** Corrected map

Figure 1.8: Use of belief filtering and feature correspondence with loop closure [29].

In order to fix this odometry problem, it is often possible for mobile robots to use trackable shapes in their environment as navigation beacons. Assuming that the walls, doorways, and other measurable objects in the environment are stationary, these can be used as reference points by which to navigate and correct odometry errors. Whereas humans naturally use vision to guide their motion through the world, robots must be explicitly programmed to reconcile the incompatible information coming from odometry, sensors, and any other onboard information sources. Some of the most successful approaches to this reconciliation have relied on probabilistic techniques. By explicitly modeling the noise from various sources of information, it is possible to maintain a probabilistic *belief* about the state of the world and the robot's pose.

With clever implementations, these techniques can be both robust and relatively inexpensive from a computational standpoint. For example, Kalman filtering with Gaussian assumptions about sensor noise creates a loop in which a possible belief about robot state is repeatedly adjusted during actuation and then corrected based on sensor measurements. Due to some convenient mathematical properties of Gaussian distributions, this technique can combine signals to maintain an up to date belief with fairly efficient matrix and arithmetic operations. Even in applications where sensor noise is not exactly Gaussian, techniques like this often work well. Kalman

filtering and related methods also rely on sensors and algorithms to perceive the robot's local environment, summarize this perception as a set of key features and their coordinates, and then repeatedly determine which of the features measured at one time correspond to a feature measured afterwards.

While all of these considerations are important for practical robotic mapping operations, they have little to do with the goal of discovering a time efficient multi agent path through an environment with the goal of achieving coverage. In addition, many realistic search and rescue scenarios involving drones may have reasonably accurate compass and GPS signals onboard the drones. While these signals are by no means free of noise, they do not suffer from the cumulative error development that tends to occur when using odometry. In addition, a large amount of robot mapping research applies to mobile robots measuring feature rich indoor settings with sophisticated sensors such as lidar. Because drones are unlikely to carry a lidar or any comparably useful distance measurement sensor, many of the techniques and results from mapping don't apply here. Nonetheless, mapping research is likely to have considerable applicability in a realistic search and rescue scenario. This is particularly true if the environment being explored is largely unknown or significantly changed by a recent disaster. Thus, while the goals and methods of robotic mapping do not apply to this work, it would surely be necessary to combine mapping approaches with the techniques here in order to produce a holistically useful robotic answer to the problems of search and rescue.

1.2.5 Applications and Other Notes

As has been stated throughout this introduction, the following work addresses robotic coverage from a mathematically idealized point of view. However, there are many high quality publications addressing techniques to handle sensor data and noise, actuation and control, localization, and a host of other important concerns. Some

examples follow:

CWRU Cutter, a robotic lawn mowing robot developed by the lab of Prof. Roger Quinn at Case Western Reserve University, required lots of effort addressing these practical issues. This includes work by Beno on the mechanical considerations of the project [4], work by Hughes on navigation [18], and work by Kreinar on localization and odometry noise handling [21].

In a 2003 article for the International Journal of Robotics Research, Acar et al. develop a robotic mine and unexploded ordinance removal system. Here, the use of a complete coverage algorithm makes sure that all potentially mine filled areas are passed over. The coverage approach used the detection of critical points to perform an exact cellular decomposition of the coverage area. Making this work on a real robot required robotic perception techniques that could go from range sensor data.

Finally, work by Liu, Ma, and Huang addresses a situation where obstacles in the environment are allowed to move [25]. In other words, this algorithm achieves coverage of dynamic environments. In addition to increasing the challenge of the path planning problem, allowing for dynamic environments complicates other parts of this problem. For example, especially once multi robot teams are introduced, the authors needed to introduce a sophisticated biologically inspired neural backtracking algorithm. This approach was successful at avoiding deadlocks and allowing all robots to contribute to complete coverage. Assuming the presence of nonstationary obstacles can also make robot localization more difficult, as robots can't easily use nonstationary obstacles as reference points to navigate by.

Chapter 2

Simulation Development

This chapter describes a software package, *oprc_env,* that simulates the coverage scenario described in the introduction. This software exposes a standard interface, the *Policy* typeclass, that allows for interaction between the simulation and a wide variety of control algorithms. The package also includes functionality to generate large environment datasets based on custom mixture distributions of environment generation algorithms. oprc_env also includes functionality to run large batches of simulations with policies that are capable of modifying themselves between scenarios, thereby enabling the development and testing of online machine learning approaches to this problem. Finally, the package features a replay animation utility to aid in communicating results visually. In addition to describing these utilities, this chapter will specify many of the implementation details required to fully understand the coverage scenario and problem statement used in this research.

Note: names of datatypes, type constructors, and type constants in oprc_env start with capital letters, e.g. Position. For brevity, the names of these datatypes may be used throughout this chapter and the next in reference to both the computational abstractions and the concepts they model.

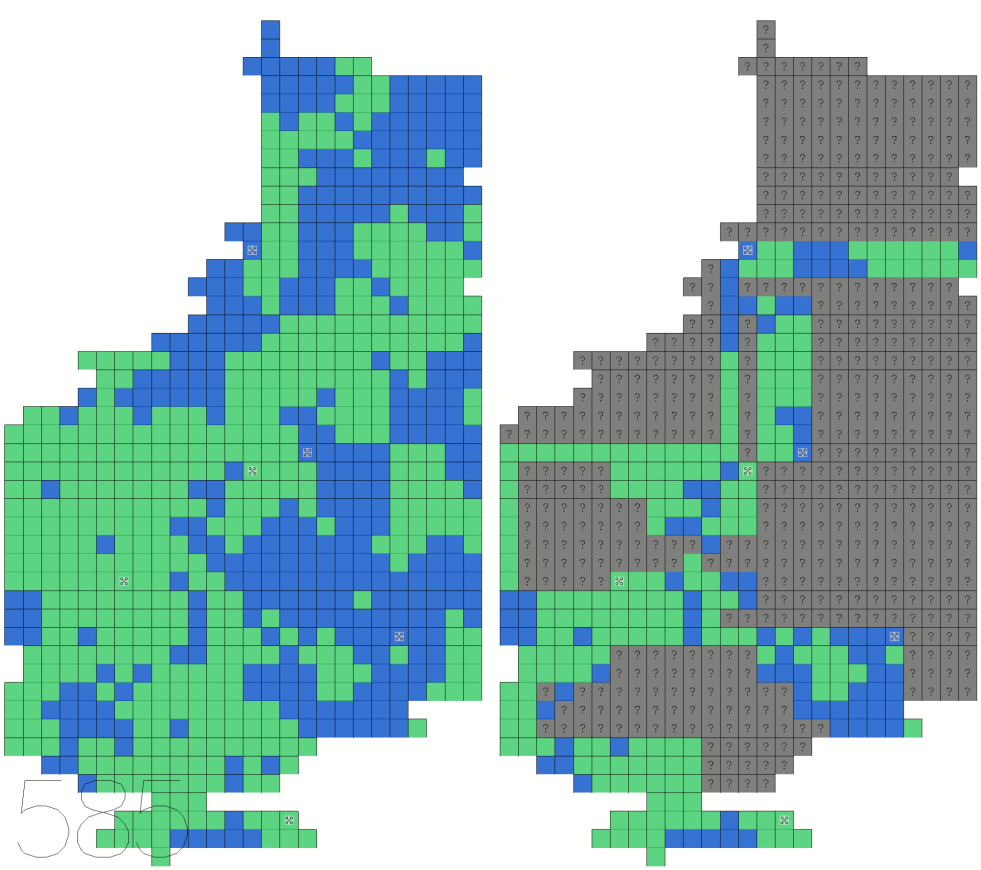


Figure 2.1: A sample environment (left) and the partial information available to robots exploring this environment (right).

2.1 Implementation of Problem Statement

2.1.1 Simulated Environment & Dynamics

The oprc_env package represents a multi agent coverage problem as a simulation in which all times and locations are represented with integer values. All ground positions in the environment are indexed by a pair of integers (x, y). For ease of communication, the x and y coordinates are assumed to correspond to longitude and latitude, respectively. For example, the position $(2, 1)$ is directly east of the position $(1, 1)$, and the position $(0, 0)$ is to the southwest of $(1, 1)$. The minimum x and y coordinates in an environment are both 0 by default, although this is not

guaranteed to be true in general. All simulations start at time T = 0.

This design choice allows for simulations to be computed and performance to be evaluated more efficiently than if the environment was modeled as a continuous area or if the dynamics were modeled with continuous time. Since this work claims to achieve coverage paths that are applicable to continuous environments, as was the case with STC and other closely related algorithms, the use of a discrete simulation requires some justification. The work by Gabriely and Rimon on STC provides the inspiration and precedent justification for this choice [12]. Because all robots following STC move directly between the centers of adjacent cells, it is possible to encode the continuous path followed by an STC-controlled robot as a sequence of environment cells (or subcells, in keeping with the original work's terminology) visited. Since these cells are aligned with a square grid, it is trivial to show that these cells can be indexed by pairs of integer coordinates.

From there, assuming that time is divided such that all drone motions take an integer number of time steps allows for discrete time in simulation. While the ratio of distances corresponding to a cardinal and an intercardinal motion is technically irrational, it is perfectly reasonable to approximate this ratio as 10:14. As long as two drones in the same scenario don't use wildly different proportions of cardinal and intercardinal movements over any extended period, paths that take an equal amount of time in simulation will take a roughly equal amount of time in a continuous environment. Furthermore, the best performing algorithms to solve these scenarios use online techniques that would readily adapt to the occasional one time step delay of move completion. More details on these algorithms can be found in the remaining chapters.

The physical environment of each coverage scenario represents the locations that are in bounds, the level of detailed scrutiny required to cover each of those locations, and the positions of obstacles inside the environment. This is achieved

by a map from Position to Patch, where a Patch can represent an environment cell that requires observation at a particular level of detail. This level of detail can be either Close or Far. Patches that require Close detail must be observed by a drone flying at a Low altitude in order to be considered covered, while patches that require only Far detail will be covered once observed by a drone flying at either a High or Low altitude.

```
newtype Environment = Environment {getMap :: (Map.Map Position Patch)}
```

Each Patch in an environment can have up to eight neighbors. Four of these neighbors are adjacent to the patch in the four cardinal directions North, East, South, West, while the remaining four share a vertex with the patch and lie in one of the four intercardinal directions NE, SE, NW, SW. Given an in-bounds location, any arbitrary subset of these eight neighboring locations may be in bounds.

2.1.2 Robot and Ensemble Representation

The datatype used to represent individual robots, or drones, appears in the *Drone* module. Each drone's identity is distinguished from those of any other drones by assigning it a unique integer identification number:

```
data Drone = DroneID Int
```

In addition to this simple definition, *Drone* contains several other datatypes that place a drone in the context of its environment. For example, a drone's position includes both a two dimensional ground position and an altitude from the set High, Low:

```
data DronePosition = DronePos
  {
    getEnvPos :: Env.Position
```

```
  , getEnvAlt :: Env.Altitude
  }
```

In addition, this module includes the previously unseen Action type, which expresses the eleven different moves a drone can make in terms of four categories:

```
data Action =
    MoveCardinal Env.CardinalDir
  | MoveIntercardinal Env.IntercardinalDir
  | MoveVertical VerticalDirection
  | Hover
```

In order to track the execution of these actions, most of which take multiple time steps, *Drone* also defines the DroneStatus type:

```
data DroneStatus =
  Unassigned DronePosition
  | Assigned Action DronePosition
  | Acting Action StepsRemaining DronePosition
```

Finally, *Drone* also defines some of the accessory datatypes and utility functions required to more easily work with the above types in other modules.

The comparatively small *Ensemble* module represents the state of an entire collection of drones. In addition to the usual utilities, the primary type provided here represents the state of such an ensemble as an association list between a drone and that drone's activity:

```
type EnsembleStatus = [(Drone, DroneStatus)]
```

2.1.3 Environment Interface

At any given time during an in-progress coverage scenario, only a limited subset of environment information is made available to the algorithm that controls drone motions. The set of in bounds locations in the environment, also known as that environment's *Footprint*, is always known. The partial information available to an agent is represented by the EnvironmentInfo datatype, and this type contains a map from Position to PatchInfo. The set of Position values in this map's domain is normally the same Footprint that belongs to the real environment. A PatchInfo represents a state of knowledge about the patch at a particular location. Patches can be Unseen, Classified, or FullyObserved. As the names suggest, Unseen and FullyObserved correspond to patches that have never been seen at all and to patches that have been completely covered, respectively. A Classified patch is one where the level of observation detail required (DetailReq) is known, but the patch has not been fully covered yet. In practice, this only comes up for patches that require Close scrutiny and have only been seen by one or more drones flying at a High altitude, in which case the patch would have only been observed with Far scrutiny.

The EnvironmentInfo datatype, together with the previously described EnsembleStatus, forms the content of the WorldView datatype. A WorldView represents all of the information about the current state of the simulation available to a control algorithm:

```
data WorldView =
  WorldView {
    getView :: EnvironmentInfo
  -- , getDroneList :: Ensemble.DroneList
  , getEnsembleStatus :: Ensemble.EnsembleStatus
  }
  deriving Eq
```

Control algorithms plug in to the simulation via an instance of the Policy typeclass, which abstracts the behavior of any type capable of commanding directions to an Ensemble and potentially maintaining a notion of internal state. Specifically, the typeclass is defined as:

```
class Policy p where
  nextMove :: p -> WorldView -> (NextActions, p)
```

As the name suggests, NextActions is a datatype that expresses newly issued commands for some drones to follow. This is implemented as an association list:

```
type NextActions = [(Drone, Action)]
```

The nextMove function provided by any type p with an instance of Policy is called by the core simulation code to determine what moves have been commanded to each drone in the simulation at every time step. As suggested by the type signature, most sensible implementations of nextMove use a particular value of the policy type p to decide what moves to command next. A simple type for which a Policy instance would be easy to provide might contain a direct mapping from WorldViews to NextActions. In this case, an instance of nextMove would determine what value of NextActions to return by simply performing a lookup on this map at each time step. Along with this value, nextMove would return the original policy value along with the result of that map query.

Of course, an instance of Policy like the one described above is not easy to implement for anything but the smallest environments and number of drones. This is because the number of possible WorldViews in an environment must be larger than the number of ways the drones can be distributed in that environment. It is easy to show that the latter notion corresponds to sampling from a bag of unique outcomes with replacement. Thus, for an environment with n cells and m drones, the number

of WorldViews is much greater than n^m. This is already an inconvenient number of entries to store in a data structure, and each of these outcomes corresponds to a distribution of drones in space that is achievable. As a result, some number of entries for each of these would need to be included in any Policy implemented as a lookup table.

Counting the number of achievable WorldViews exactly, complete with all of the possible partial information states, is much more difficult. However, the established lower bound makes it clear that this quantity is not practical to handle in any direct way. This discussion may seem pointless for an implementation of Policy that is obviously unwise. However, getting a sense of this scale has implications that will inform the design of internal representations used by more effective Policy instances as discussed in later chapters. Luckily, because nextMove yields another value of p along with its NextActions decision, it is possible and often desirable for p to encode a control strategy that learns and plans, evolving with the context in which it currently operates.

Some control algorithms are capable of learning from experience and modifying behavior in a given simulation based on results from a previous one. The PersistentPolicy typeclass exists in order to provide the interface required for this functionality:

```
class Policy p => PersistentPolicy p where
  cleanup :: p -> (WorldView -> p)
```

The exact purpose of cleanup's type may not be immediately apparent. In short, the most useful policy representations tend to need access to some information about the current environment before they can be initialized. The function type of cleanup's output shows that it will return the right kind of value to make this initialization fit nicely with tooling developed for more general Policy instances. The fact that cleanup must be applied to a value of type p before it returns this value

allows control algorithms to pass any encoded strategies learned during task execution to be applied to the completion of the next task. As later chapters will discuss, this capability is actually necessary for optimal performance when the distribution of environments is not known before task execution begins.

2.1.4 Environment Generation

In order to evaluate the coverage performance of any agent with a Policy instance, it is necessary to have a simulated environment for that algorithm to cover. It is also useful to have a large dataset of simulated environments for the agent to cover when collecting performance statistics. Finally, since some Policy instances may be capable of learning and specializing their behavior to the distribution of environments previously explored, it is useful to have fine control over the nature of the environments in a particular dataset. To meet all of these needs, *oprc_env* has a utility executable called *generate_environments* to create custom environments and environment datasets based on a specification of the distribution from which those environments should be drawn.

Before describing the capabilities of *generate_environments* in full, it is useful to consider the algorithms used to generate one specific environment. There are a few different algorithms for this purpose, and each one generates environments from a distribution that is qualitatively different from the others.

The simplest environment generation algorithm to describe is called the *BernoulliGen*. As the name suggests, this generator samples a required level of scrutiny from the same Bernoulli distribution for each location in the generated environment's Footprint. As a result, most of the content of the *BernoulliGen* algorithm comes from the procedure used to generate Footprints. The *randomFootprint* function supplies this functionality.

The *randomFootprint* algorithm generates a Footprint, or set of in-bounds loca-

tions. While a Footprint alone does not constitute a usable coverage environment, it can be augmented to form a complete environment description by algorithms that assign a DetailReq value to each of the positions in the set. The type signature and corresponding parameter nicknames used by randomFootprint are as follows:

```
randomFootprint :: StdGen --a source of randomness
                -> Int --allowed variation between border locations
                -> Int -> Int -> Int -> Int --coordinate limits for Positions
                -> Maybe Footprint
randomFootprint gen
                varLimit
                xMin xMax yMin yMax =
  undefined --bulk of implementation excluded for brevity
```

Roughly, this function randomly selects one point on each face of the "bounding box" of legal position coordinates defined by the x/y min and max arguments. Then, interpolation is used to connect turn these points into a discrete approximation of a quadrilateral. Each coordinate on this perimeter is randomly perturbed by an amount whose magnitude must not exceed varLimit. Finally, the set of all positions inside this perimeter is computed and returned. The output type of a fully applied randomFootprint, Maybe Footprint, indicates that this function may fail to produce a valid Footprint. This is because the arguments for randomFootprint are sometimes sourced from configuration files or user input, and information from these sources may correspond to unrealizable requirements.

In addition to this basic footprint generation function, it is possible to augment the complexity of the generated footprints by taking some locations out of bounds. One simple algorithm for this purpose creates the set of all locations whose Chebyshev distance from some center location is no greater than some limiting value. However, careless use of such a function can lead to unintended consequences. For

example, if the group of locations taken out of bounds includes one or more positions that were already on the perimeter of a Footprint, the resulting Footprint will not have a distinct cluster of out of bounds locations in the middle of an in bounds region. Since taking locations out of bounds is primarily useful for simulating obstacles in an environment, this is usually not a desirable result. Even worse is the case where the group of locations taken out of bounds intersects with multiple non-adjacent sections of the original Footprint's perimeter. In this case, the connectivity of the resulting environment could be changed such that a drone starting in one set of patches has no way of visiting the other set. This makes performance of complete coverage impossible, so this is almost always undesirable.

The main function used to take locations out of bounds, *hole*, implements a simple safeguard to avoid these pitfalls: it takes away positions in groups organized by ascending Chebyshev distance from the center. Starting from the center itself ($d = 0$), this procedure repeatedly removes another ring-shaped layer of locations until one of two things happens. The function can either reach the desired threshold for distance from the center, or it can detect that removing another layer would cause the removal of a border position. In either case, the algorithm terminates. With these safeguards in place, *hole* can be used to scatter virtual obstacles around an environment at different positions and with different obstacle sizes.

Given a random Footprint, the remainder of BernoulliGen simply assigns a DetailReq value to each location in the Footprint independently. The specific value assigned to each patch is determined by sampling a floating point number from the interval [0, 1] and then comparing this value to a threshold provided as input to the function. The value of this threshold corresponds to the probability that any individual patch will be assigned the value Close, meaning that it must be observed by a Low drone before it is covered.

Other algorithms available for use with *generate_environments* produce an En-

vironment in which the spatial distribution of DetailReq values features regions that are locally correlated in some way. A variety of these algorithms exist, and all of them rely on *randomFootprint* for the environment's basic shape. From there, however, these algorithms rely on procedures for filling in DetailReq values which are more sophisticated than the one used by *BernoulliGen*. In order to produce a somewhat structured output that still features randomness, many of these algorithms benefit from an efficient way of assigning DetailReq values to patches in a random order. This allows each new assignment to be drawn based on the presence of any other locations that have already been filled in.

The Fisher-Yates shuffle is an algorithm to reorder, or shuffle, the elements of a finite sequence. This algorithm provides a random reordering of a list of positions to support the many environment generation algorithms that require this functionality. It is also used for implementation of random drone failure and other simulation features that have not yet been described. It is possible to run this algorithm in-place on a sequence of length n in $O(n)$ time. In addition, given a way to randomly select an element from a range of integers with uniform probability, the Fisher-Yates shuffle has the useful property of producing any possible reordering of the sequence with uniform probability. The original work by Richard Durstenfeld to develop this algorithm for use on a computer can be found in [10]. The version presented here is a generalization of that algorithm to operate directly on sequences of arbitrary data.

The *Shuffle* function assumes access to a function g that can accept an integer argument i and return an integer sampled uniformly from 0 to i, inclusive. It operates on l, a finite sequence of values of any datatype. For simplicity, it also assumes access to functions that get the length of a finite sequence and swap the elements of that sequence at two indices. Finally, note that sequence indexing starts at 0.

function SHUFFLE(g, l)

$n \leftarrow length(l)$

while $n > 0$ **do**

$s \leftarrow g(0, n)$

$swap(l, s, n)$

$n \leftarrow n - 1$

end while

return l

end function

The actual source code for this algorithm can be found in the FisherYatesShuffle module in *oprc_env*.

Using the Fisher Yates shuffle, it is possible to create environment generation algorithms that tend to output locally correlated ”clusters” of similar patches. Each patch in the environment is given a DetailReq value by taking a supplied threshold value and altering that value by a small amount for any neighboring patches that have already been given an assignment. Each of these threshold alterations makes it slightly more likely to generate a patch of the same type as the one that triggered the alteration. The order in which patches are filled in is determined by applying the Fisher Yates shuffle to a list of all in bounds locations in the environment. This order randomization step makes it likely that a small number of completely unbiased ”seed” locations will get a DetailReq value sampled that is not affected by any neighbors. Depending on the exact parameters used, skipping this randomization step may result in environments that are dominated by one scrutiny requirement even when the initially supplied threshold value is not biased one way or the other.

2.1.5 Running Simulations

The Scenario type and the functions that operate on it are responsible for expressing and updating the state of a simulation. The type itself consists of a WorldState (discussed shortly) and a Policy for tracking the instantaneous state of the simulation,

along with time and history parameters to allow simulation replays and performance analysis:

```
data Scenario p =
  Scenario {
    getPolicy :: p
  , getWorldState :: WorldState
  , getTime :: Integer
  , getHist :: MoveHistory
  }
```

The function that steps simulations forward in time takes any `(Policy p => Scenario p)` and applies nextMove to that Scenario's Policy and a WorldView generated from the current WorldState. The resulting policy is assigned to the next Scenario value. The NextActions portion of nextMove's output is where most of the simulation's portion of the computation occurs. First, for replay and analysis, any newly commanded actions are added to the MoveHistory. In addition, time is incremented. The *Scenario* module also contains most of the code for managing the simulation of a single coverage scenario from a high level. This includes managing the interaction between a policy and the environment dynamics.

High level management of the environment dynamics falls to the *WorldState* module. Within that module, the WorldState data represents the state of an environment in a context agnostic to any policy or high level control mechanism:

```
data WorldState =
  WorldState {
    getEnv :: Env.Environment
  , getInfo :: EnvironmentInfo
  , getEnsemble :: Ensemble.EnsembleStatus
```

```
}
deriving Eq
```

Aside from the input of control instructions from the higher level *Scenario* context, the major source of dynamics in these simulations comes from the rules of the environment. A function, *updateState*, figures out the new instantaneous WorldState from a NextActions value and the current worldstate. Taking the environment through a single time step based on this information involves several steps distributed to subordinate functions. Briefly, this includes figuring out the new assignment status of each drone, stepping each one through its assigned action, figuring out the information that is instantaneously available to each drone, and augmenting the previous EnvironmentInfo value with any new information from that view. Except for the (variable) cost of policy computations, these functions take the lion's share of computation time during simulations.

2.1.6 Visualization Tools and Parsing

The most precise human-friendly way to communicate the activity of this simulation is via text written to the terminal by the various executable functions in *oprc_env*. Within this project, the typeclasses *Show* and *Pretty* must be provided for values of a particular datatype to be expressed in this way. Between these two typeclasses, *Show*'s instances tend to provide completely unambiguous and easily parsable representations of data. This is very convenient for some datatypes, such as *Hop*, where the bare programming language syntax is adequate for expressing such a simple idea in a form humans can understand at a glance. For example, a hop 3 positions east and one position south would be printed to the console as (3, -1) when using *Show*.

Pretty is at its most useful when expressing data of an intermediate level of complexity. Data that is list-like, that has two or three levels of hierarchy, or that

is otherwise large but mostly one-dimensional is not well suited to expression in the raw programming syntax that could, in principle, be used to specify such values directly in source code. While it may be possible to write Show instances that do an adequate job of expressing this kind of code, it is not straightforward to derive easily readable *Show* instances for large datatypes in terms of the instances of subordinate datatypes, which may have layers of hierarchy even before being included in the parent datatype. For example, indenting the contents of some string representing an already hierarchical datatype to fit inside a larger string with yet another layer would require meaningful parser development for every new instance of *Show*. Furthermore, while Show instances may compute arbitrarily sophisticated String values for the data they represent, convention dictates that instances should be expressed with the syntax required to directly program those values elsewhere. *Pretty* solves these problems by representing the hierarchical nature of its contents explicitly, rather than with repeated newline and tab characters or with nested delimiters in a string. In addition, the functions used to get these representations on a screen make rich use of IO such that they can adapt a presentation of data to fit on terminal windows of various sizes. This allows the contents of very small values of complex datatypes such as Scenario to be displayed on screen in a way that is easy to program and to understand.

However, the coverage behavior computed in these simulations can be difficult to interpret when expressed *exclusively* in terms of words and numbers. Because this software simulates the exploration of two dimensional worlds, some of the datatypes are best expressed in a medium better suited to expressing two continuous dimensions of data. While this kind of expression is less precise and less suitable for low-level debugging tasks than text representations, it enables an immediate and intuitive understanding of large quantities of data being updated on screen at up to 60 frames per second. To facilitate the design and presentation of such data, this

software makes use of the *Gloss* graphics library.

The animation software developed for *oprc_env* is primarily for visualizing environments with color coded two-dimensional drawings. An environment is displayed as a collection of colored squares. Each square in this collection represents the patch type associated with one position. All of these squares appear in grid aligned positions that are filled in with white wherever there is no in-bounds patch. In addition to empty environments, EnvironmentInfo data can be displayed. This datatype represents the state of partial information about an environment. Fully known EnvironmentInfo patches are drawn the same as they would be in an environment. Locations that are unobserved are drawn in grey with the question mark (?) character overlaid to indicate uncertainty. Partially observed patches are always classified by required scrutiny, so these are partitioned into a grey triangle and a triangle of whatever color corresponds to the fully observed patch.

Often, it is useful to visualize both the plain environment and the partial information known by a policy about that environment at the same time. The most common visualization used in this software shows an environment on the left and the information used by a policy on the right. In addition, the status of any drones in the environment are represented with symbols overlaid on the appropriate locations within the EnvironmentInfo representation. Finally, a timer appears in the lower left corner (Figure 2.2).

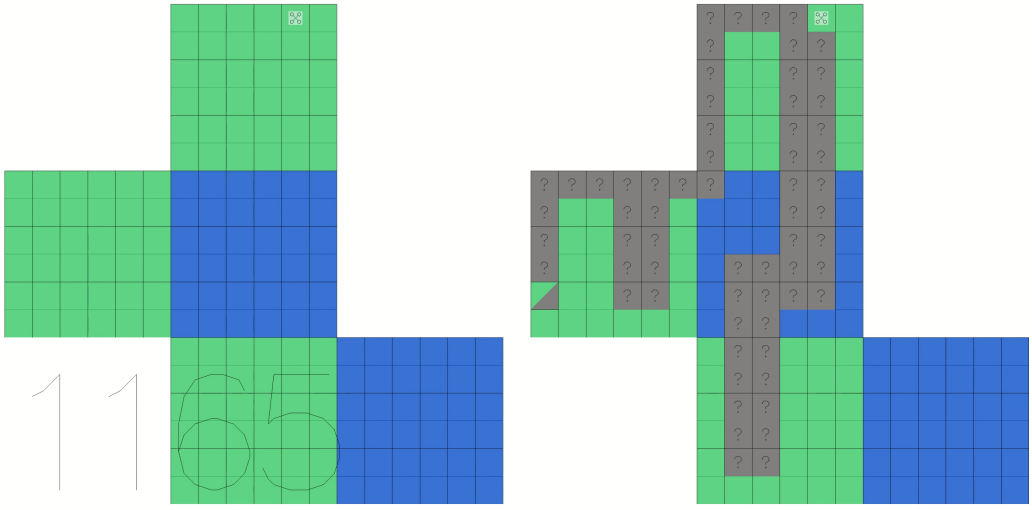


Figure 2.2: A sample environment (left) appears next to a visualization of the active coverage task (right). Unvisited locations are marked in grey on the right. A pictograph made of crossed lines with circles on the ends depicts the drone and shows its location on the map. The only partially observed location on the map appears on the west end of the EnvironmentInfo representation, and is partially covered by a grey triangle.

The modules containing this animation code, *AnimateReplay* and *AnimateScenario*, also provide functionality for drawing some of the data structures used by particular Policy instances. Some of these data structures can be thought of as walls for the drones to follow in a loop, and so are suitable for drawing over a map. For more information on this, refer to the next two chapters.

oprc_env also features significant parsing functionality. This functionality is useful for creating custom environments to load and run in the simulation, serializing and saving a collection of automatically generated environments for evaluating the performance of some policy, and allowing an executable to understand interactive input of control commands. The parsing used in this code does not occur in any speed critical areas, and so the best library for the job should emphasize ease of use over speed. Because of that, the Trifecta library by Edward Kmett is used here. This library provides monadic parser combinators and makes it very easy to sort through descriptions of datatypes in plain text input to a console, etc. The slightly more

complex parsing involves reading in environment configuration files. However, the resulting parser makes it possible to work with a representation of environments that is very easy for humans to read and write. For example, the following file comes from a plain text configuration that corresponds to the environment used in Figure 2.2:

```
      HHHHHHLLLLLL
      HHHHHHLLLLLL
      HHHHHHLLLLLL
      HHHHHHLLLLLL
      HHHHHHLLLLLL
      HHHHHHLLLLLL
HHHHHHLLLLLL
HHHHHHLLLLLL
HHHHHHLLLLLL
HHHHHHLLLLLL
HHHHHHLLLLLL
HHHHHHLLLLLL
      HHHHHH
      HHHHHH
      HHHHHH
      HHHHHH
      HHHHHH
      HHHHHH
```

Note that the plain text representation of an environment appears vertically flipped when compared with the resulting environment visualization. This is an intentional trade off that makes the line numbers of a plain text file correspond to the y coordinate of any patches specified on that line. Because many modern plain

text editors track both the line number and character position of a cursor, this makes it possible to easily specify environments with properties that change at precise coordinates. Other than that exception, it should be easy to see the correspondence between this plain text and the final illustrated environment on the left of 2.2. These features appear in the module *ParseOPRC*.

2.2 Extra Features

2.2.1 Drone Dropout

In order to fully test the adaptability of Policy implementations, *oprc_env* includes functionality to run simulations with drone dropout. When this feature is in use, up to one drone may be permanently disabled at each time step in the simulation. Whether or not a drone drops out during a given time step is based on comparison of a random number sampled from the uniform distribution over the interval [0, 1] and a customizable threshold value. It is not possible for more than one drone to fail during a single moment in time according to this formulation. Also, in order to make sure that all scenarios are possible to complete, at least one drone is guaranteed to be operational at all times.

These two assumptions and the basic model of drone dropout they apply to are not meant to be especially realistic. However, this model of dropout is sufficient for splitting Policy algorithms into a few coarse categories according to adaptability: those that fail when dropout occurs, those that complete a dropout scenario in an inefficient way, and those that smoothly adapt to each dropout event as it occurs.

Note that animations of scenarios with dropout do not show disabled drones in a unique way. Rather, disabled drones are simply frozen in place with whatever state they had just before being disabled. Theoretically, this makes it difficult to distinguish between a disabled drone and one which is active but repeatedly receiving

Hover commands. However, this issue does has not come up in practice.

2.3 Operation and Software Utilities

In addition to the several modules and algorithms detailed here, *oprc_env* contains some small pieces of code to do relatively mundane jobs. This section collects a few of them in one place.

RunEnvironments is responsible for managing a sequence of simulations to be performed as a particular policy instance is tested or trained on an entire dataset of environments. This module mostly handles the system IO functionality required to go from a folder full of environment specifications to a record of performance over all of those environments. The actual simulation functionality is accessed by library calls to the other modules that create this behavior. One exception to this has to do with PersistentPolicy instances. When working with PersistentPolicys, this module's *runPolicyAccum* function handles more of the high level manipulation of policy values between one scenario and the next. Because PersistentPolicy's cleanup function allows for a modified policy to emerge after every run, operating *RunEnvironments* in this mode demonstrates a policy's potential to learn and improve with experience.

RandomOPRC provides functionality to produce random behavior. Computer programs with behavior that depends on random number generation can be hard to debug because of a lack of repeatability. However, the modules throughout most of *oprc_env* don't access randomness directly from the system (i.e. through IO). Instead, most functions that rely on random behavior take a parameterized source of randomness as an explicit input. As a result, effort spent debugging random behavior can be reduced through tracking access to the random number source required to produce the same behavior every time. In all other cases, *RandomOPRC* provides a

pseudorandom seed obtained through IO that can simply be discarded once passed along to whatever function needs it.

SerializeOPRC transforms an Environment datatype into a string that can be saved and retrieved later. This string corresponds to the plain text visual representation of an environment shown in the parsing discussion.

oprc_env assigns different datatypes to cardinal and intercardinal directions in order to acknowledge the qualitative difference between them. This enables many other functions to be written more concisely, including the spanning tree generation code that only generates edges in cardinal directions (see Chapter 3). However, there are several cases where directions are best treated as actions. In these cases, it because useful to treat cardinal directions, intercardinal directions, vertical directions, and hovering in a uniform way. The *MoveCosts* module provides a part of this interface with the *Costed* typeclass:

```
class Costed c where
  cost :: c -> Int
```

This very simple typeclass provides a way of expressing the time cost associated with a particular action. Although it is not required to be this way, all values of any particular datatype map to the same cost as one another. For example, the following code states that all horizontal moves in cardinal directions require 10 time steps to complete:

```
instance Costed CardinalDir where
  cost _ = 10
```

Similarly, all intercardinal horizontal moves take 14 time steps. Ascending and descending both take 10 time steps, and hovering takes just one.

SampleVals provides a variety of instances for most of the datatypes defined as part of the *oprc_env* environment. It also provides values of some policy types

and various imported types, both of which allow for easier development runs in the *Main* module.

The *Env* module contains many of the core simulation data discussed throughout this chapter. In addition to those and many others, the following are worth noting:

A hop is simply defined as a 2-tuple of integers. It is distinguished from the Position type by its lack of constructor and by its reference to displacements and relative positions.

The function *chebyshevCluster* outputs the set of positions whose Chebyshev distance from some center location is no greater than some threshold value. In two dimensions, the Chebyshev distance between locations (x1, y1) and (x2, y2) is given by max(abs(x1 - x2), abs(y1 - y2)). The use of this function will be explained in the next chapter.

Chapter 3

Simple Coverage Policies

3.1 The Environment-Policy Match up

3.1.1 Two Simple Policies

It may not be immediately obvious that a strategy for switching between altitudes is helpful. The problem formulation used here is such that policies that only ever command drones to fly at Low altitude can still be guaranteed to complete coverage for all legal environments. However, this policy is not optimal for all environments. To understand this, consider the following toy instance of the coverage problem:

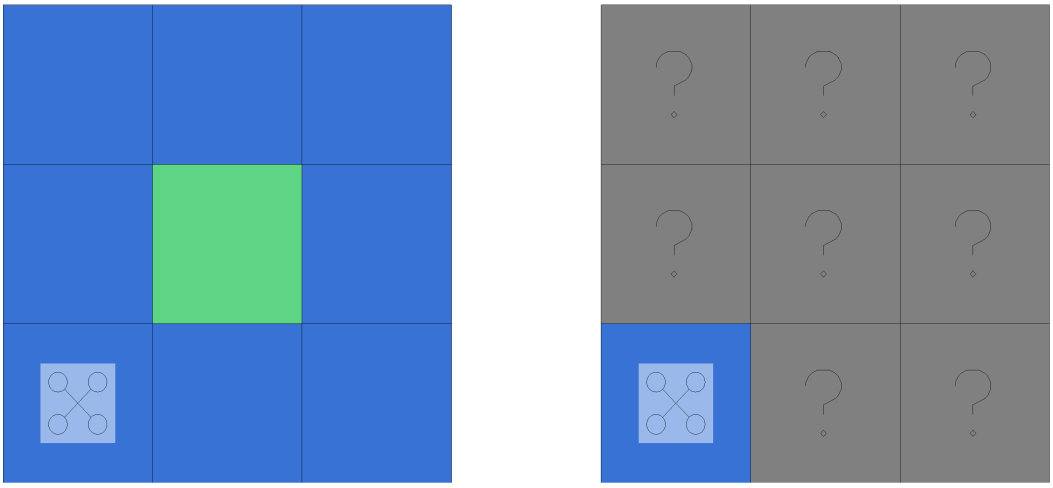


Figure 3.1: A drone sits at the lower left of a small environment. The right image indicates that it has only covered the patch directly below it so far.

Assume that a single drone must cover this environment starting from a low altitude over position (0, 0). Because of this starting position, the drone starts with limited information about the environment (Figure 3.1 right). From here, there are a few viable strategies the drone could use to cover this environment.

3.1.2 High Sweep First Policy

One option that works well for this and other policies is to ascend to a high altitude, view every location in the environment with low scrutiny, then descend to cover any locations with a patch requiring a highly detailed view. In this case, because the drone starts from a low altitude, the first step in following such a policy is to ascend. This takes 10 steps and results in the following view of the environment:

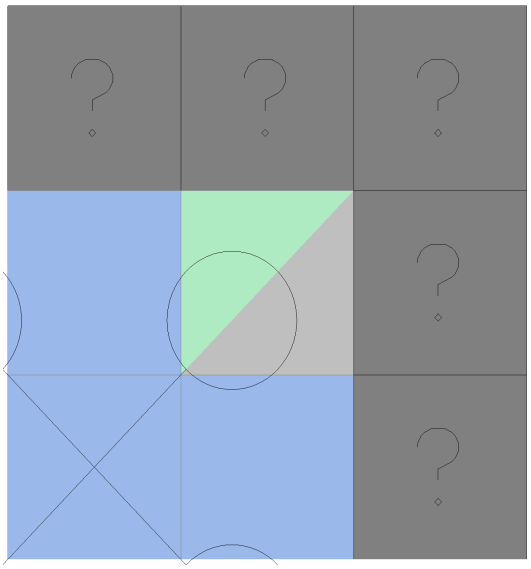


Figure 3.2: The drone has ascended, giving it a view of the 3x3 grid of patches around its ground position.

Because this drone has ascended while flying over a corner position, there is not a full set of nine in bound patches for the drone to view at this moment. Instead, it can see the 2x2 group of patches that are within its field of view. The larger scale of the drone's high altitude field of view is indicated by enlarging the drone icon to cover an amount of the screen that corresponds to this much area on the drawn map.

From here, the High Sweep First Policy (HSFP) sends the drone to a position that takes full advantage of its large field of view, (1, 1). This is directly northeast of the drone's current location, so this position can be achieved by an intercardinal motion for 14 steps.

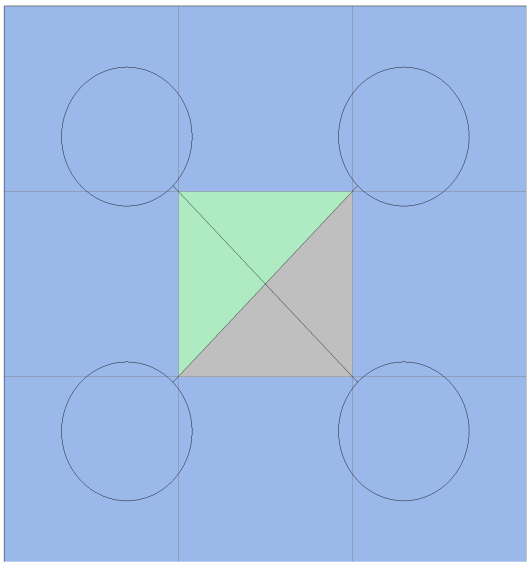


Figure 3.3: Complete information without complete coverage

From this vantage point, the drone is able to see a full 3x3 region of in bounds locations. This region happens to be the entire map. In addition, most of the locations in this environment have been drawn in blue, indicating that those locations can be covered with a low scrutiny view. Thus, all the drone needs to do is descend over the single patch requiring high scrutiny. Ten steps later, the task is complete:

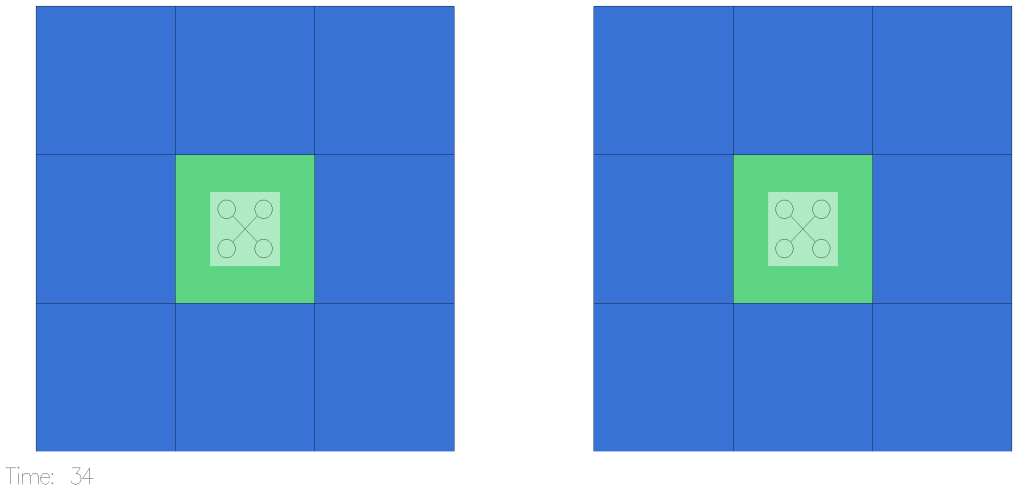


Figure 3.4: fully covered environment

As shown in Figure 3.4, this takes a total of 34 time steps. For this exact map,

it's possible to achieve coverage in just 24 steps by moving northeast before ascending. However, given the decision to ascend at the first time step, this is the fastest possible task performance.

3.1.3 Low Sweep Policy

A second type of policy suitable for covering this environment is the low sweep policy (LSP). This policy snakes back and forth across the environment one column at a time. In order to achieve coverage of environments with more complex shapes, there are a few exceptions to that description which might not seem obvious when covering a simple environment such as the 3x3 in this example. In short, all locations in the environment will eventually be visited. Because any drone commanded by this policy will view all of its visited patches with high scrutiny, this policy is guaranteed to achieve complete coverage.

When exploring the same environment as before, this policy commands the following eight cardinal motions: [North, North, East, South, South, East, North, North]. These eight motions take a total of 80 time steps and result in a completely explored environment.

3.1.4 Performance on an Alternate Environment

The above example makes it seem like the "High Sweep First" approach may be the best choice for a policy to complete this task. However, this is also not the case for all environments. Consider the following environment, for which the relative performance of the same two policies from last time is flipped:

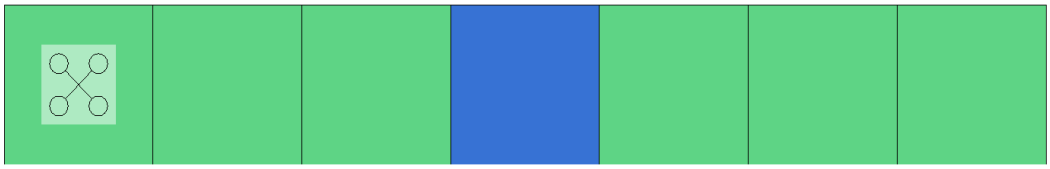


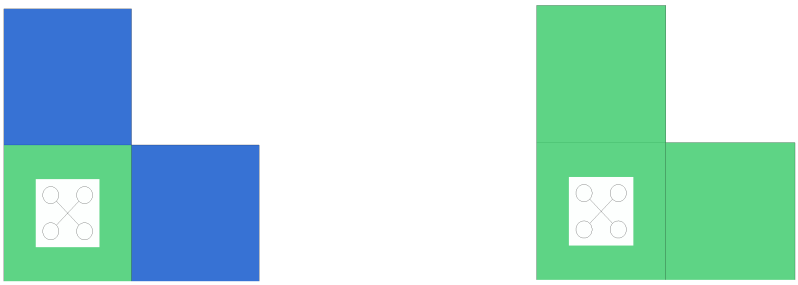
Figure 3.5: An environment with a different shape and dominant scrutiny requirement

The all low policy does well on this one: with six moves to the right, the task is complete in just 60 steps.

In this environment, HSFP does not do well at all. It ascends and flies most of the way across the environment, resulting in a complete view in which most of the patches are only partially covered. It must descend, move to the right, then sweep almost all the way back across the environment before finishing the task. This takes 130 time steps, over double the time taken by the all low policy.

3.1.5 Discussion

Both of these examples used very small environments in order to limit the length of explanations required. On this small a scale, the size and shape of the environment's Footprint provides a strong hint about the appropriate altitude switching strategy to employ when exploring the environment. Because the environment's Footprint is available to an agent before any decision needs to be made, it may be possible to create a policy that optimizes its behavior around that factor somewhat. However, for large environments that mostly consist of wide open space, the importance of this contribution on a case by case basis is likely to be minimal. Setting aside considerations about Footprint shape, the different ratios of High vs Low scrutiny patches used in these examples provide the first hint that the a priori unknown content of each environment has implications for the best possible strategy. As an extremely simple proof that this is the case, consider the following tiny environments:



(a) A tiny environment with a mix of scrutiny requirements.

(b) An environment with all high scrutiny patches and the same shape.

Figure 3.6: It is not possible to know the optimal policy to cover an environment with this shape without assuming a distribution of scrutiny requirements.

In both cases, assume that there is a single drone that starts at the bottom left position at a Low altitude.

In the first case, the best possible solution is to immediately Ascend. This action, whose move cost is as low or lower than all other move costs available in this simulation, completes the coverage task. Thus, this is clearly the optimal policy to use for exploring this environment in particular. In the second case, all environment cells require High scrutiny, and so must each be visited from a Low altitude. This means that the drone must move to each of the following states before the coverage tasks is complete:

1. Low over Position (0, 1)
2. Low over Position (1, 0)

In this case, one optimal policy would be to move East and then South-West. This policy causes the scenario to complete in 24 time steps. More importantly, any Policy that starts out with an Ascend would then need to Descend before any of the necessary observations to complete coverage can be performed. These two altitude switching moves require a total of 20 time steps to complete, making it clearly impossible to finish the task in less than the 24 time steps that another policy demonstrates.

With that, it has been demonstrated that a policy's chance at optimally exploring either of these environments can be ruined at the *very first* move performed. The set of moves which keep a possibility of optimality in the first environment, Ascend, and the set of moves that do the same in the second environment, MoveCardinal East, MoveCardinal South, are disjoint. All of this is true despite the fact that policies operating in these two environments must make a decision based on precisely the same initial information.

This proves that it is impossible to create a policy that makes optimal decisions without assuming some distribution of possible environments. The exact degree to which environment distributions will affect policy performance on larger environments is not addressed analytically in this work. However, as later sections will demonstrate, experimental results show that this difference is quite dramatic.

As discussed in the previous chapter, *oprc_env* is capable of generating quite a large number of different environments. Even if the exact parameters used by a particular environment generator are known in advance, it is not easy to characterize the distribution of resulting environments in a way that is useful for fine-tuning the design of policies. In addition, it seems intuitively likely that the distribution of real world environments in which this coverage behavior would be useful is similarly complex. This chapter will discuss the development and experimental optimization of Policy instances that do a good job on a complex distribution of environments.

3.2 The Low Sweep Policy

This section and the next go into more detail about the implementation of policies described in the previous example. These policies are considered offline. This distinguishes them from *online* policies, which may react to information discovered at every step of task performance. In addition, all policies that learn and update their

models over the course of multiple tasks are considered online.

These offline policies do most of their planning ahead of time and have limited potential for adaptation. While the implementations vary, most of the behavior achieved by these policies could be fully computed ahead of time. One significant exception to this is with offline policies that perform an initial high sweep of the environment. In order to gain any benefit at all from that initial effort, such a policy must develop a new plan to visit specific high scrutiny patches once the high altitude phase is complete. With that and other small exceptions, these policies don't take advantage of information discovered during task performance in any meaningful way. This limits their power somewhat compared to online policies (Chapter 4), and this limitation becomes especially apparent in a multi agent coverage setting.

3.2.1 Behavior of the Low Sweep Policy

The Low Sweep Policy (LSP) is the simplest policy implemented in this work. As stated in the previous section, its primary goal is to move through the environment one column at a time. This results in a coverage pattern of back and forth vertical stripes, and this pattern is often compared to the cutting pattern someone might create when mowing a lawn.

This behavior is achieved algorithmically by seeking out one patch at a time according to a particular heuristic. The first rule of this heuristic is that the algorithm prefers to seek out patches that are as far to the left as possible. Thus, at any given time, this algorithm is sending a drone on its way to some location that is further to the left than any other uncovered patch (or tied for this distinction). This results in every patch in a particular column being covered before any patch in the next column is actively sought out.

Nonetheless, this policy requires some meaningful algorithmic effort to handle situations in which the footprint of the environment has a complex shape. For exam-

ple, consider a drone seeking out the top left corner of the following environment:

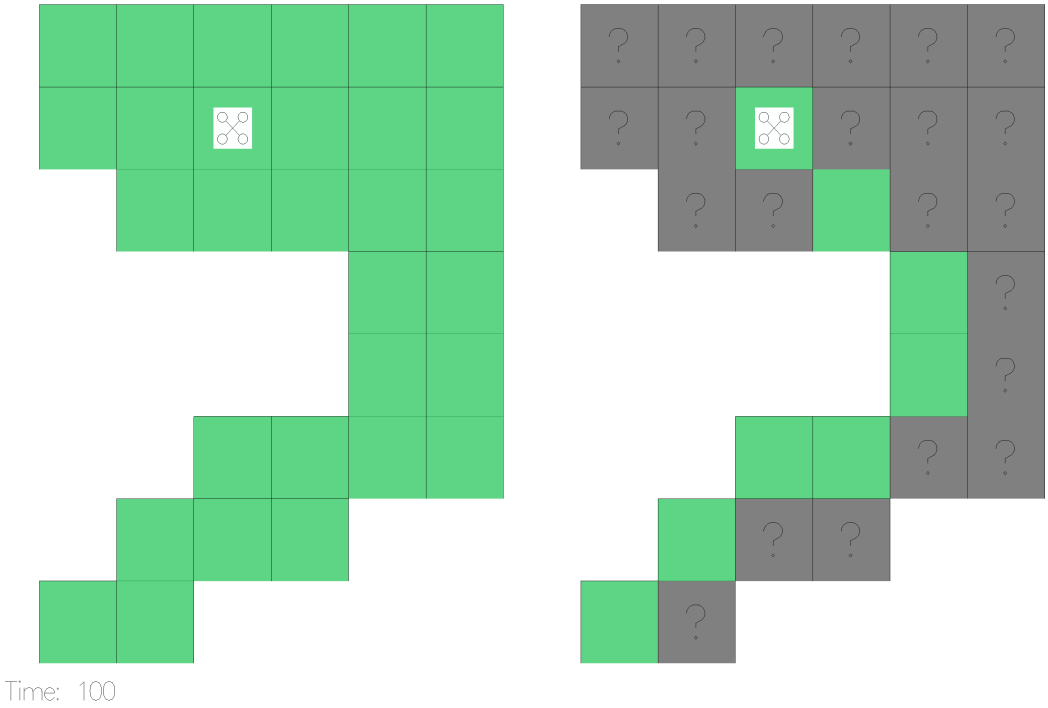


Figure 3.7: This environment has several columns that are not connected without access to the rest of the environment. On the right, the covered patches reveal the path taken by this drone when seeking out the top left section of the environment.

In this environment, it clearly is not possible to move from the bottom to the top of the left most column without visiting positions further to the right. Thus, in order to get from the bottom of the left column to the top, some sort of intelligent path finding algorithm must be used. In the simplest version of the Low Sweep Policy, this functionality is supplied by the popular A-Star algorithm.

3.2.2 The A-Star Algorithm

The A-Star algorithm is used to find the lowest cost path between two nodes in a graph in a computationally efficient way. It was developed in 1968 by Hart, Nilsson, and Raphael [14]. Given a graph G, A-Star attempts to connect a start node s and an end node e by repeatedly making informed guesses about which edges of the graph may be on the lowest cost path between these nodes.

At each step, A-Star uses a pair of functions, g and h, to evaluate which node is the best choice to explore next. g maps a node to the lowest known path cost between that node and s. h, short for *heuristic*, maps a node to an estimate of the lowest cost path from that node to e. A-Star repeatedly examines a set of "frontier" nodes to see which one has the lowest combined values of g and h. Initially, the frontier set contains only s. As each node n is explored, it is added to a closed set (initially empty). Then, each of the neighbors of n are examined. For each element x of these neighbors, the cost $g(x)$ of a path from s to x is computed by $g(n)$ plus the cost of the edge from n to x. Then $f(x)$ is computed as $g(x)+h(x)$. Any x such that no lower value for $f(x)$ has already been discovered is then added to the open set, and the process repeats. Once the neighbor e has been expanded, the algorithm terminates. Along with some simple additions that keep track of the parent of each node on the best known path between that node and s, this algorithm can then return the best found path from s to e.

The choice of heuristic function is critical to the performance of the A-Star algorithm. In this project, the graphs explored by A-Star represent physical space. As a result, it is possible to come up with a reasonable heuristic by simply comparing the physical locations of e and any other node. Assuming that there are no out of bounds locations, the shortest path between some node n and e is to move diagonally from n until either the x or y coordinate of the resulting position is equal to that of e. Then, the rest of the path is a straight line in some cardinal direction until e has been reached. As a result, one reasonable way to compute $h(n)$ is to compute the magnitudes of the x and y coordinate differences between n and e. Let a be the lesser of these magnitudes, and b be the greater. Then following a path as described above has cost $14*a+10*(b-a)$, or just $4*a+10*b$.

The A-Star algorithm is known to be optimal, in the sense that it finds the lowest cost path while exploring the fewest possible graph nodes, under certain conditions

on the heuristic used. These conditions are called admissibility and consistency.

A consistent heuristic is one where $h(x) \leq d(x,y) + h(y)$ for any neighbors x and y in the graph, where $d(x,y)$ is the cost of a path from x to y. With consistency, it is ensured that a node $n1$ that is explored before its neighbor $n2$ will not have its value of f decreased when $n2$ is expanded. The first time $n1$ is expanded, g(n1) must already have the lowest possible value. If $n2$ was such that $g(n2) + c(n1,n2) < g(n1)$, then the fact that $h(n2) \leq c(n1,n2) + h(n1)$ means that $f(n2) < f(n1)$. In addition, any neighbor $n3$ of $n2$ which was an ancestor on this shorter path from s to $n2$ must be such that $f(n3) \leq f(n2)$ because $h(n3) \leq c(n3,n2) + h(n2)$, and so $g(h3) + h(n3) \leq g(h3) + c(n3,n2) + h(n2)$. By induction, the first node on the lower cost path from s to $n1$, a neighbor of s, has a lower f value than $n1$, and so must always be explored before $n1$. This means that the next node on the path will likewise be explored before $n1$, and so on. Thus, by the time $n1$ is explored, no such lower cost path exists, and so the lowest cost path from s to $n1$ has been discovered.

An admissible heuristic h is one such that for any node n, $h(n)$ is less than the lowest cost path from n to e. Combined with the above conclusion based on consistency, admissibility guarantees that the first fully expanded path from s to e is the lowest cost path. By the definition of A-Star, the first time e is expanded, the value $f(e)$ is lower than the value $f(n)$ for any other node n in the open set. With consistency, it is guaranteed that the optimal value of $g(n)$ is known for any such node. Combined with that information, admissibility guarantees that the the true cost of any path from s to e through n is at least as great as $f(n)$. Thus, no unexplored path exists from s to e with a lower cost than the one discovered path.

While the above argument proves that admissibility and consistency are sufficient to guarantee the optimal efficiency of A-Star, it does not prove that these conditions are necessary. Dechter and Pearl proved that A-Star is not optimal for all graphs with admissibility alone, reversing a previously held belief on the subject.

With the A-Star algorithm and the described heuristic, it is possible to discover efficient paths between any pair of locations in a space to be covered. This allows the low sweep policy to command moves that always seek out locations in the left-most column that has not yet been completely covered.

3.2.3 Optimizations of the Low Sweep Policy

The necessity of taking occasional detours from the currently pursued column brings a new opportunity for optimization. Namely, can the low sweep policy do anything to avoid unnecessary visits to patches already covered in the course of a previously followed A-Star path? The answer is yes, but the extent to which these optimizations help or hurt is largely based on the geometry of the environment being explored.

As a heuristic, the cost of visiting a previously visited location can be thought of as the amount of time taken to move through that location. Assuming a policy makes mostly cardinal movements, this cost is close to 10 time steps. *oprc_env* includes functionality to run A-Star with an awareness of the status (covered / uncovered) of each of the nodes it explores. By running a particular variant of the function, it is also possible to make A-Star apply a custom penalty to the cost of each edge that moves into a previously covered patch.

Experimentation with this custom penalty amount showed that a penalty of 9 was most effective. This penalty encouraged A-Star to avoid proceeding blindly through an entire stretch of covered locations when a slightly longer detour contained mostly unvisited locations. Smaller penalties were also helpful, but often missed obvious opportunities. Larger penalties worked occasionally, but these tended to go to unreasonable lengths to avoid visiting previously covered patches.

3.3 The High Sweep First Policy

The other policy previewed in the introduction to this chapter was the High Sweep First Policy (HSFP). This policy saves time on certain environments by initially visiting all locations in the coverage space with low scrutiny. This allows for high altitude flight that can view new locations up to three times faster than a low flying drone. For environments dominated by large, uninterrupted sections of in bounds locations, this initial high sweep step views the entire environment close to three times faster than the low sweep policy.

However, after performing the high sweeping step, this policy must view any high scrutiny locations from a low altitude. As a result, environments with many locations which require high scrutiny are poor matches for this policy, since it must visit all such patches at a slower rate after wasting a lot of time viewing them from a high altitude. Environments with relatively few patches requiring high scrutiny are a better fit for this policy, but there is still significant variation in performance depending on the details. For example, some environments may have a distribution of high scrutiny patches that requires a low flying drone to make long trips between each of these locations. Such a distribution of locations might simply be isolated groups of very few high scrutiny patches that occur evenly throughout the environment. Since HSFP uses the relatively simple low sweep policy by default, a bad distribution of locations could also be one which causes the low flying portion of this policy to repeatedly move around complex environment borders.

3.3.1 Details of the HSFP

Briefly, the behavior of the high sweep policy attempts to do roughly the same thing as the low sweep policy, but at a high altitude. However, not all of the details of the low sweep policy translate directly into useful high altitude behavior. For

example, since this policy works from left to right, it often does not make sense to directly seek out whatever position contains the leftmost unobserved location. This is because a drone flying at a high altitude can see all patches with Chebyshev distance from the drone less than or equal to one. As a result, many cases in which the leftmost unobserved patch has the horizontal position x are best handled by sending the drone to some coordinates with horizontal position $x+1$. This is not always the case, as not every location is guaranteed to have a neighbor to the right.

However, in this work, locations with a full complement of in bounds neighbors are the rule rather than the exception. As a result, it is possible to guide the locations sought by the HSFP by a heuristic that handles most cases correctly. This heuristic works by preferentially seeking out positions with particular x coordinates. This x coordinate, modulo 3, should be one greater than the lowest x coordinate of any position in the environment. In other words, for an environment with a minimum x coordinate of 0, this policy prefers to seek x coordinates 1, 4, 7, 10, etc. In environments which are perfectly rectangular or otherwise easy to navigate, this results in a drone flight that visits columns ideally spaced to make full use of the width of the drone's 3x3 field of view.

Not all environments have such a clean shape, though. For cases where a location is not in view of a high flying drone at any positions with x coordinate fitting the pattern described above, HSFP uses A-Star search to seek those positions out directly. More precisely, whenever the lowest x coordinate of any unobserved position is x_1 and there are no unobserved positions with x coordinate $x_1 + 1 - (x_1 mod 3)$, the unobserved position is visited directly.

In most cases, following the primary 'coarse sweep' pattern accounts for most of the observations achieved by an instance of HSFP. In environments with positions missed by this part of the policy, these positions tend to be at related positions in space. For example, areas around the outer border of an environment or around

obstacles and other irregular shapes in the environment footprint tend to contain all of the misses. As a result, it is sometimes useful to simply let these positions be visited by the low flying 'cleanup' portion of HSFP. Depending on how these patches are distributed both spatially and with respect to their required level of scrutiny, this variant of the policy may perform better than the original described above.

Chapter 4

Intelligent and Multi Agent Coverage Policies

The coverage policies described in the last chapter make use of A-Star, an algorithm often classified as an example of 'artificial intelligence'. However, those policies attempt to work by following very simple rules most of the time, and A-Star is used only as a last resort. These policies are easy to understand due to their simplicity, and the adaptability of A-Star along with guarantees about properties of the environment make it easy to feel confident in their correctness. However, these policies may miss significant opportunities for behavior optimization in certain kinds of environment. In addition, the policies of the previous chapter did nothing to take advantage of the multi agent setting in which they operate. While the drone dropout and reintroduction feature introduced in this chapter will give these policies some redeeming properties for the multi agent case, more sophisticated approaches are necessary to take full advantage of the power of a team of covering robots.

4.1 The Clustering Low Policy

The first of these policies is the Clustering Low Policy (CLP). This policy manages a team of multiple drones by giving each of them a portion of the overall coverage space to complete. Because it is not straightforward to compute an optimal partitioning of an arbitrary coverage space between a team of covering robots, this policy uses an iterative clustering technique to come up with a reasonable approximation of this division. In addition, because this is the first of several policies designed to adapt to unforeseeable events such as drone dropout and reintroduction, this clustering operation is repeated throughout the coverage scenario. By adaptively redistributing assigned territory online, the CLP algorithm offers a single elegant solution to the dual problems of unforeseeable events and the difficulty of computing an optimal territory partition in one step.

4.1.1 K Means Clustering

The algorithm used to determine and change the division of territory assigned to each drone is a lightly modified variant of the K Means Clustering (KMC) algorithm. KMC was developed by Lloyd in 1957 and published in its standard form in 1982 [26]. As it relates to this problem, the goal of KMC is to find some number k of 'means' for a collection of data points such that the average squared distance from any point to the nearest mean is minimized. Once these means have been discovered, the original points can be divided into k groups according to which of the k means they are closest to.

The optimal solution to the problem of K Means Clustering is NP-Hard in general, so there is not a computationally efficient way to solve this problem perfectly. There are several algorithms which do a good job of approximating the solution to this problem with various claims to near-optimality and runtime. One of the most

popular such algorithms and the one used in this work is the variant developed by Lloyd, sometimes known as 'naive k means'.

This variant of the algorithm is best known perhaps because of its simplicity. Given an initialization, possibly random, of k means for some value of k, this algorithm simply repeats two steps. It assigns each point in the space to whichever mean is currently closest, then it replaces each mean with the centroid of the points assigned to it. This causes the location of the means to jump through the space with each iteration, gradually converging on a local minimum to the implicit cost function. It is common to simply run this algorithm until there is no change between two iterations, after which point the algorithm will make no further progress and terminates.

While Naive KMC is not the most computationally efficient way to perform K Means Clustering with a comparable level of optimality, it is more than adequate for this application. The number of points being divided into clusters tends to be small, and experimental runtime profiling suggests that these computations take a very small fraction of the overall time spent running simulations.

The exact implementation of K Means Clustering used in this work varies only slightly from the generic version described above. First, the mean of a collection of positions is always rounded into a discrete position value. This is because these means supply a possible location for the corresponding drone to be commanded to. In addition, because this variant of K Means Clustering will be run repeatedly on a smaller and smaller set of positions to be covered, it must eventually handle a case where some of the means actually have no positions associated with them. In this case, each drone gets a random uncovered position assigned to it. Finally, most runs of this K Means Variant use just one or a few iterations. This is because the previously described unconventional setting in which K Means CLustering is run can lead to issues with convergence. In addition, it is not necessary for a drone to have

the optimally computed cluster before making any moves. Instead, four iterations of this algorithm are run at the start of most scenarios involving this or other policies that use clustering. This is enough to give each drone a distinct section of territory to pursue, and the particulars of the Clustering Low Policy, discussed below, ensure that this clustering algorithm continues to be used appropriately through the end of the scenario.

In other words, the use of k means clustering results in a partitioning of the space containing the data points into Voronoi cells. Because the algorithm penalizes clusters with large spatial extents quadratically, the Voronoi cells produced by K Means Clustering tend to be of comparable spatial extent to each other. Because of this property, using this algorithm to divide territory between drones tends to give each one a comparable amount of space to cover. This does not guarantee that each drone will require the same amount of time to cover its assigned territory, but it provides a reasonable first guess.

4.1.2 Details of the Clustering Low Policy

With the details of this customized K Means Clustering algorithm in place, it is now possible to describe the details of how the Clustering Low Policy commands moves. The CLP maintains a data structure that, among other things, associates a 'mean' position and a set of territory in the environment with each drone. Every time nextMove is called on an instance of this policy, these means and information about the environment information observed so far are passed to a function that sets up a new iteration of K Means Clustering. In this iteration, the set of locations to be divided and assigned to each mean is generated by reviewing which locations have not been completely covered.

Note that the details of this criteria can very in this project's other uses of the K Means Clustering algorithm. For example, some coverage policies that use KMC

to command drones flying at a high altitude, discussed shortly, pass a function that will keep only completely unobserved locations in the set of points to be divided between means.

Once this information has been passed along, KMC returns a new set of means and territory assignments associated with each drone. Next, directions for each drone are computed. Each drone that is not already in the process of seeking a particular location is given a new location to seek. This location is selected from the most recently computed territory associated with the given drone. Within this set of possibilities, CLP attempts to select the location that is farthest from any other drone's mean position. Based on this assignment, a new sequence of movements is computed that will bring the drone from its current position to that newly assigned location. As in other policies, this sequence of moves uses A-Star search. The particular cost function used by A-Star applies a penalty to moves that would cause a drone to cover previously explored locations, and this penalty is especially effective at optimizing the paths created for the CLP.

The decision to make each drone seek out whatever position is farthest from all other means is a heuristic that attempts to have each drone explore the section of its territory that is most likely to be better suited to exploration by that drone than by any other. While this heuristic does not make optimal decisions with respect to that criteria, and while it is not clear that sequencing of positions to visit by such a heuristic will be useful, experimental results show that this policy works well compared to the simple sweep policies.

The rationale for assigning these 'middle-of-territory' positions to be visited first is that if another drone ends up needing more territory to explore, this heuristic ensures that the patches closest to that other drone are likely to go unexplored until that territory reassignment occurs. Likewise, if a drone exploring its own territory ends up needing to take on more work, the fact that other drones are behaving ac-

cording to this heuristic makes it likely that more territory near where the drone is exploring will be available to take over. In addition, a well tuned choice of penalty function for A-Star path planning seems to handle most of the issues with how commanded locations are sequenced: many locations that should have been visited before the currently commanded one end up on the path generated by A-Star. Some of these descriptions of good behavior break down slightly as the number of locations to be covered dwindles to just a few per drone, but the bulk of the coverage task is handled well by this policy.

4.2 A Single Agent Spanning Tree Policy

As discussed in chapter one, many robotic coverage algorithms have managed to achieve good performance through methods that compute spanning trees on a modified graph of the environment. In this thesis, the Low Spanning Tree Policy (LSTP) is a policy that incorporates these improvements for the environments in *oprc_env*. LSTP, like the policies in chapter 3, is primarliy a single agent policy. While there are several more sophisticated multi agent spanning tree based policies developed for this setting, LSTP provides the most direct point of comparison to the other single agent policies. This makes it possible to analyze the effect of introducing the superior planning capabilities of this policy without having to consider the effects of the other policy changes. While these additional changes produce policies that tend to have faster overall coverage performance, the single agent policies tend to have some of the best results when it comes to the efficient utilization of individual drones. Before describing the Low Spanning Tree Policy itself, however, it is necessary to discuss the spanning tree generation algorithm that this and other policies rely on.

4.2.1 Depth First Spanning Tree Generation and Path Generation Algorithms

The spanning tree algorithm used by this policy generates a spanning forest, or a collection of spanning trees, using a depth first search based algorithm. However, in the purely functional setting used in this work, the traditional approach to depth first search does not work well. Instead, this approach to spanning tree generation is defined in terms of mutual recursion between two functions: dfsInternal and processNeighbors.

dfsInternal is responsible for managing the high level collection and combination of subtrees created by exploring from some starting node *n*. Since *n* may in general be any node in an already partially spanned graph, dfsInternal takes a description of the already visited nodes in this graph. It then passes this information along to a processNeighbors call for each of the children of *n*. As each of these computations returns, any subtrees are added to a local tree with *n* as the root. In addition, each of these calls results in an updated description of the neighbors that has been added to the tree so far.

processNeighbors examines a candidate neighbor position called by dfsInternal. At the time processNeighbors is called on some node *m*, there is no guarantee that this candidate neighbor to *n* is in bounds or that it has not already been placed into the spanning tree that exists so far. Using the information passed down to it, processNeighbors simply performs these checks. If *m* is in the relevant sets, it is added to the spanning tree as a child of *n* by returning two parts to the instance of dfsInternal that called it. The first is a direct addition of *m* and a subtree for *m*, obtained by another call to dfsInternal within processNeighbors, to the list of children of *n* being built by the parent instance of dfsInternal. The second piece of information is the newly updated set of nodes that have been added to the tree so far. Passing this set up and down the chain of recursion is crucial to ensuring that

each function call adds the appropriate collection of nodes to the spanning tree in this purely functional and recursive setting.

Because the neighbor function used to generate these spanning trees only looks for neighbors in cardinal directions at a distance determined by the custom scale of the problem (although usually either 2 or 6), not all environments and choices of root position are guaranteed to produce a spanning tree that contains every node in the environment. This is why it is necessary to generate spanning *forests* for these environments. When one spanning tree fails to contain every coarse node with in bound locations in the environment, the set difference between the actual environment footprint and whatever nodes were added to the environment is calculated. This result is then passed to another call to the spanning tree generation function that starts with a new root. While it is possible and sometimes desirable to specify a custom root to the first spanning tree generated for an environment, these subsequent calls to generate additional trees simply use the minimum of the remaining positions as the new root. This process is repeated as necessary until a list of trees with all of the environment accounted for has been created. For most practical environments, which must be fully connected, it is often possible to generate a spanning forest that contains only a single spanning tree. The main exception to this is when a cell is only accessible through diagonal motion, and so most spanning forests contain one domainant tree and a small few subtrees. In the multi agent uses of spanning tree based algorithms, discussed later, there are additional reasons for a spanning forest as the footprints being spanned can be more irregular.

Once the spanning forest for an environment has been generated, this data can be used for multiple purposes. However, it is usually desirable to convert this spanning tree into the actual path that should be followed by a drone using this tree to guide its covering moves. The first step in providing this path is supplied by the cardinalCoveragePath function, which converts a sequence of positions to visit

that, once completed, should consitute complete coverage of the environment at the appropriate hierarchical level. As mentioned earlier, this hierarchical level tends to group locations in the environment into either 2x2 or 6x6 squares. A 2x2 square contains four positions that must be visited at a low altitude in order to cover the entire square with high scrutiny. The 6x6 case is for high altitude drones, and a 6x6 square contains four 'quadrant center' positions that should each be visited in order to view the entire square at a low level of scrutiny. The cardinalCoveragePath function returns the overall sequence of locations to visit that will cause all nodes under the top level spanning forest to be covered. This function simply calls another function on each tree in the spanning forest and strings the results together at the end.

The function that generates a path for a single spanning tree has a little more going on. While the details are somewhat messy, this is another rescursive function. Briefly, this function keeps track of which 'quadrant center' it is on, and which direction is just came from. This gives the function enough information to decide which child of the current node it should attempt to visit next, or if it should generate the moves to get back to a new quadrant in its parent node and return to the higher level of recursion. While the overall path generated to go around a particular spanning tree is complex, it can be approximately described as counterclockwise motion.

Once this path has been generated for the entire spanning forest, it must be converted into a path that is realizable. This could be an issue in cases where the quadrant centers of a 6x6 square of positions are not guaranteed to be in bounds. To deal with these cases, out of bounds path locations are substituted with one or more waypoints which collectively provide enough vantage points to view every in bounds location within that quadrant. Then, the default version of A-Star search is used to find valid paths between each of these waypoints. This results in a se-

quence of adjacent environment positions that can be converted directly into the drone directions necessary to go around the total spanning forest.

4.2.2 Details of the Low Spanning Tree Policy

The Low Spanning Tree Policy (LSTP) uses the functions described above to generate a spanning forest for the environment it attempts to cover. As with other policies, it contains a map from drones to cached directions. In predictable settings with no drone failure, this map only needs to be filled once for the entire simulation to run. The spanning forest and subsequent path generation functions, bundled together in the same module where they are defined, is called for any drone in need of new directions. This gives the LSTP a sequence of adjacent positions that must be converted into the directions required to visit each of those positions. In addition to making the obvious mappings (i.e. the subsequence [(0, 0), (1, 1)] corresponds to an intercardinal motion to the northeast), moves must be supplied to get the drone from its current position to the start of its sequence of waypoints. This does not come up often for the LSTP, but it becomes an important consideration for the cases with multiple drones, drone failure, and other considerations that necessitate online replanning. Once these directions are in place, LSTP simply commands them in sequence like many other policies.

4.3 The Clustering Low Spanning Tree Policy

As the name suggests, the Clustering Low Spanning Tree Policy (CLSTP) modifies the approach taken by LSTP by introducing the same K Means Clustering insights used by the Clustering Low Policy.

4.4 Persistent (Learning) Policies

In the last chapter, it was proved that the optimal policy with which to perform coverage depends on the distribution environments that policy will be asked to solve. In that chapter, this problem was approached by hand crafting policies to perform as well as possible on a complex distribution of environments through experimentation and tweaking of parameters. In this section, some policy implementations are developed that can recreate a version of this experimentation and tweaking across multiple scenarios. Importantly, these policies perform these tweaks *automatically*, using the PersistentPolicy typeclass and associated environment interface tools to refine an approach specifically tuned to a distribution of environments encountered online.

It's worth addressing why this online learning behavior could be superior to a hand crafted approach. The distribution of real world environments in which you might want to deploy drone based robotic coverage is not at all easy to characterize. Furthermore, practical applications of this kind of technology may see it deployed to various owners around the world. Any one of these teams of robots is likely to be deployed in environments with somewhat different properties the environments seen by other teams. As a result, there may be an opportunity to optimize the behavior of each of these teams of robots to the particular circumstances in which it is deployed. Such optimizations would not be easy to achieve through any sort of manual effort, so a well crafted policy that is capable of changing itself between individual operations may be the best way to achieve this optimization. In addition, there is some theoretical value in seeing how coverage policies can be automatically optimized for an experimentally determined distribution of environments. Much of the remainder of this chapter is devoted to exploring that topic.

Chapter 5

Conclusion

5.1 Performance Comparison

In order to evaluate the usefulness of each of these policy types, it is necessary to evaluate the speed with which they cover a wide variety of environments.

In addition to the policies described so far, one more policy is included as a benchmark: the random filtered policy. While all other policies have at least some high level strategy guiding the sequence of motions they command, the random filtered policy only rules out moves that obviously have no use in a coverage task. These moves include hovering in place or attempting to visit locations outside the environment boundaries. Such a policy is obviously not a serious contender for the most efficient way to cover an environment of any meaningful size or complexity. However, including its performance in these results puts the achievements of the other policies into perspective.

Two other policy instances exist in *oprc_env* besides these. One is the RandomPolicy, which is similar to the above except that it contains no restrictions whatsoever to the moves commanded at any given time. The other is the PolicyMap, which contains an explicit map of situations to the assocaited actions to command.

For obvious reasons, neither of these policies scales in performance to handle environments of even a modest size. As a result, the performance of these policies is not measured here.

5.2 Performance Comparison on Simple Environments

5.3 Performance Comparison on Complex Environments

5.4 Drone Dropout Performance Comparison

5.5 Efficiency of Drone Utilization

5.6 Future Work

Although this work is fairly permissive of complex environment shapes, it does not consider coverage of environment that can't be exactly represented as a square grid. As noted previously, the search-and-rescue scenarios that motivate this work could benefit from close inspection of tight corners and areas near the perimeter of environment boundaries. Since these boundaries may be due to collapsed or naturally occurring structures, their borders are unlikely to be aligned with cardinal directions. An algorithm that could adapt its behavior near these regions could be of great practical importance. For an example of other work that has achieved this kind of extension in the past, consider the exact coverage variant of STC as described in [12]. It may also be useful to represent the borders of regions requiring low or high scrutiny as arbitrary polygons, although the opportunity for improved algorithmic performance or other practical benefits are less clear in this case.

Another potential improvement of this work would allow for persistent coverage. Targets may be mobile in certain search and rescue situations, and so it is necessary to re-visit locations periodically in order to make sure nobody has moved into them.

Bibliography

[1] C. Papadimitrou A. Itai and J. Szwarcfiter. “Hamilton Paths in Grid Graphs”. In: *SIAM Journal on Computing* (Nov. 1982).

[2] Ercan Acar et al. “Path Planning for Robotic Demining: Robust Sensor-Based Coverage of Unstructured Environments and Probabilistic Methods”. In: *The International Journal of Robotics Research* 22.8 (July 2003), pp. 441–466.

[3] Esther M. Arkin, Sándor P. Fekete, and Joseph S. B. Mitchell. “The Lawnmower Problem”. In: *CCCG*. 1993.

[4] John Beno. “CWRU Cutter: Design and Control of an Autonomous Lawn Mowing Robot”. M.S. thesis available publicly on OhioLINK.

[5] Howie Choset. “Coverage for robotics – A survey of recent results”. In: *Annals of Mathematics and Artificial Intelligence* 31 (Jan. 2001), pp. 113–126.

[6] Howie Choset et al. “Exact Cellular Decompositions in Terms of Critical Points of Morse Functions”. In: *Proceedings of IEEE International Conference on Robotics and Automation (ICRA ’00)*. Vol. 3. Apr. 2000, pp. 2270–2277.

[7] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education, 2001. ISBN: 0070131511.

[8] K. Daltorio et al. “An Obstacle Edging Reflex for an Autonomous Lawnmower”. In: *IEEE/ION Position, Location and Navigation Symposium* (May 2010).

[9] Rina Dechter and Judea Pearl. “Generalized Best-First Search Strategies and the Optimality of A*”. In: *Journal of the ACM* 32 (3 July 1985).

[10] R. Durstenfeld. “Algorithm 235: Random Permutation”. In: *Communications of the ACM* (July 1964).

[11] I. Endo. “Cooperative Sweeping by Multiple Mobile Robots with Relocating Portable Obstacles”. In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems* (Nov. 1996).

[12] Y. Gabriely and E. Rimon. “Spanning-tree based coverage of continuous areas by a mobile robot”. In: *Annals of Mathematics and Artificial Intelligence* 31 (Oct. 2001).

[13] Yoav Gabriely and Elon Rimon. “Competitive on-line coverage of grid environments by a mobile robot”. In: *Computational Geometry* 24.3 (2003), pp. 197–224. ISSN: 0925-7721. DOI: https://doi.org/10.1016/S0925-7721(02)00110-4. URL: http://www.sciencedirect.com/science/article/pii/S0925772102001104.

[14] Peter Hart, Nils Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions of Systems Science and Cybernetics* 4 (2 July 1968).

[15] N. Hazon and G. Kaminka. “Redundancy, Efficiency and Robustness in Multi-Robot Coverage”. In: *Proceedings of the 2005 IEEE International Conference on Robotics and Automation* (Apr. 2005).

[16] Susan Hert, Sanjay Tiwari, and Vladimir Lumelsky. "A terrain-covering algorithm for an AUV". In: *Autonomous Robots* 3.2 (June 1996), pp. 91–119. ISSN: 1573-7527. DOI: 10.1007/BF00141150. URL: https://doi.org/10.1007/BF00141150.

[17] Ralf Hinze. *PSQueue*. Version 1.1.0.1. Dec. 9, 2019. URL: https://github.com/hackage-trustees/PSQueue.git.

[18] B. Hughes. "A Navigation Subsystem for an Autonomous Robot Lawn Mower". M.S. thesis available publicly on OhioLINK.

[19] D. Kempe. "Multi-Robot Forest Coverage". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems* (Aug. 2005).

[20] Edward Kmett. *Trifecta*. Version 2.1. Dec. 6, 2019. URL: http://github.com/ekmett/trifecta/.

[21] E. Kreinar. "Filter Based Slip Detection for a Complete-Coverage Robot". M.S. thesis available publicly on OhioLINK.

[22] Roman Leshchinskiy. *Data.Vector*. Version 0.12.0.3. Dec. 9, 2019. URL: http://github.com/haskell/vector.

[23] Haskell Libraries. *Containers*. Version 0.6.2.1. Dec. 6, 2019. URL: http://github.com/haskell/containers.git.

[24] Ben Lippmeier. *Gloss*. Version 1.13.1.1. Dec. 6, 2019. URL: http://gloss.ouroborus.net/.

[25] Hong Liu, Jiayao Ma, and Weibo Huang. "Sensor Based Complete Coverage Path Planning in Dynamic Environment for Cleaning Robot". In: *CAAI Transactions on Intelligence Technology* 3 (1 Mar. 2018).

[26] Stuart Lloyd. "Least Squares Quantization in PCM". In: *IEEE Transactions on Information Theory* 28 (2 Mar. 1982).

[27] *pretyprinter*. Version 1.5.1. Dec. 9, 2019. URL: http://github.com/quchen/prettyprinter.

[28] The GHC Team. *The Glasgow Haskell Compiler*. Version 8.6.5. Dec. 6, 2019. URL: http://www.haskell.org/ghc/.

[29] Sebastian Thrun. "Exploring Artificial Intelligence in the New Millennium". In: ed. by Gerhard Lakemeyer and Bernhard Nebel. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. Chap. Robotic Mapping: A Survey, pp. 1–35. ISBN: 1-55860-811-7. URL: http://dl.acm.org/citation.cfm?id=779343.779345.

[30] Ashley Yakeley. *time*. Version 1.9.3. Dec. 9, 2019. URL: https://github.com/haskell/time.