An Adaptive Memory-Based Reinforcement Learning Controller

Keith August Cissell

June 2018

Abstract

Recently, the use of autonomous robots for exploration has drastically expanded–largely due to innovations in both hardware technology and the development of new artificial intelligence (AI) methods. The wide variety of robotic agents and operating environments has led to the creation of many unique control strategies that cater to each specific agent and their goal within an environment. Most control strategies are single purpose, meaning they are built from the ground up for each given operation. Here we present a single, reinforcement learning control solution for autonomous exploration intended to work across multiple agent types, goals, and environments. The solution presented here includes a memory of past actions and rewards to efficiently analyze an agent's current state when planning future actions. The agent's objective is to safely navigate an environment and collect data to achieve a defined goal. The control solution is first compared with random and heuristic control schemas. To test the controller for adaptability, the controller is next subjected to changes in the agent's sensors, environments, and goals. Control strategies are compared by examining goal completion rates, the number of actions taken, and the agent's remaining health and energy at the end of a simulation. Results indicate that the newly developed control strategy is adaptable to new situations. A reinforcement learning based controller, such as the one presented in this research, could help provide a universal solution for controlling autonomous robots in the field of exploration.

Contents

Chapter 1

Introduction

As studies in autonomous computing and robotics have become more extensive, the two categories have overlapped to create a field of research combining both areas. Autonomous robots of many different forms have been built and applied to a wide variety of use cases. Rovers, drones and even aquatic robots, combined with artificial intelligence (AI), have been created to interact with diverse types of environments. The tasks that these robots carry out greatly vary based on the robot's abilities and the environmental limitations they may face. This variance causes a demand for distinct software and hardware configurations to achieve each robot's given task. Exploration-based research is one area that has seen a drastic shift to the use of autonomous robotics. Exploration involves the navigation of a previously unknown area to learn new information or seek out specific features that may be held within. While the field of exploration presents many different use cases and demands, similarities can be drawn between all of them. Almost all operations involving exploration are focused on navigation within unknown environments to gather and analyze data. Both autonomous computing and robotics offer unique and clever solutions for this problem space.

While a great deal of research has been conducted in adaptive hardware for robotics, there is not an extensive amount of research on adaptive software for integrating the variety of robot setups with exploration-based tasks. Most of this is due to the fact that each robot has

a unique set of capabilities and control schema designed for a single purpose. Autonomous robots tend to focus in on a certain niche, which requires them to be built from the ground up for each task. This diversity in agent-task pairs could greatly benefit from a single adaptive solution of operation setup and control. Here we discuss how AI can be applied to exploration and present an adaptive solution for encompassing the variety of use cases of autonomous robotics in the field of exploration.

The field of robotics has benefited tremendously through growing research in artificial intelligence. AI methods are commonly used in situations when there is a known number of controllable variables and a wide solution space, making them valuable tools in exploration. For example, in a search and rescue setting, if an earthquake had collapsed a building and agents were deployed to find survivors, the operation has no definition to the landscape of the altered environment, or the potential location of survivors. There are many AI approaches that have been applied to create decision making models for robotic agents operating within unknown environments. In particular, artificial neural networks (ANN) [3, 4, 16, 36] and reinforcement learning (RL) [3, 16, 35] decision models have yielded promising solutions for creating optimal control patterns. Through data analytics, ANNs and RL models will often find correlations in data that are not always obvious. Their ability to learn from experience make them adaptable to new situations and uses. Learning is typically achieved through training in simulation, which has many benefits over real-world training. Simulations allow agent setups and control schemas to be tested and observed without the potential of damaging equipment or the environment. Additionally, they offer the ability to conduct multiple tests in short periods of time without the need for any physical setup or supervision.

The presented solution to the strong diversity of use cases for autonomous robotics in the field of exploration is a unified Surveillance Coordination and Operations Utility (SCOUt). SCOUt takes a top down approach to agent-task definition and provides an adaptive control schema for a wide variety of robotic agents and their uses. The combination of an agent-task definition and simulation platform, and an adaptive control schema creates a tool which can

be applied to both new and existing use cases of autonomous robots in exploration. This is achieved by abstracting the very basics of autonomous robotics. SCOUt's control schema repeatedly follows the process of collecting data from sensors, analyzing the agent's state, and the choosing actions based on previous experiences for completing exploration-based operations. The schema uses reinforcement learning to build a memory-based decision model. When applied, the decision model compares the agent's current state to previous states in memory. By analyzing similar states, the controller will then be able to predict what actions will yield the best results given the current situation. Data stored in the controller's memory is highly abstracted so that it can be applied to a wide variety of agent setups, goals, and environments. For example, locating a human or mapping the presence of water in unknown environments, with a land-based robotic agent. SCOUt also provides a simulation platform for training and testing the controller in the variety of situations it can be applied to. The platform's architecture is also highly abstracted to create an easy to use tool for defining and simulating different agent abilities, goals, and types of environments.

For testing the adaptive control schema, interactions between several configurations of agent setups, goals, and environments are simulated to observe performance. Performance is measured by the controller's ability to complete a defined goal, the number of actions that the controller had to perform before completing the goal, and the remaining health and energy levels of the agent. As a base line, both random and heuristic control schemas are used in testing for comparison against SCOUt's control solution. Heuristic control schemas use a defined logical analysis process to analyze a situation and choose an act. This schema is meant to reflect a specialized, non-adaptive control solution that might be applied in these scenarios. The SCOUt controller is trained to complete specific goals and then experiments are conducted to test its usefulness. First, tests are conducted to observe the performance of the memory-based learning schema. Next, adaptability is tested through presentation of agent setups, goals and environments that the controller has experienced.

The layout of this paper is as follows. Chapter 2 covers related works in the field of

autonomous exploration. Chapter 3 describes the data structures and methodology of the simulation and control platforms. Chapter 4 then goes into detail about how the random, heuristic and SCOUt controllers operate. Experimentation and results are covered in chapter 5, a conclusion is drawn in chapter 6, and ideas for future work on this project are discussed in section 6.1.

Chapter 2

Related Work

Recent advances in hardware capabilities and computational intelligence techniques have led to the introduction of robotics in numerous complex use cases. Robotic agents continue to phase out human agents for tasks that are considered mundane or dangerous, or simply because a machine can provide better results for less cost. Here we focus on the application of autonomous robots in exploration-based operations. Examples of this are search and rescue missions to find survivors after a natural disaster such as an earthquake, and research missions to map water on the surface of Mars. Exploration-based operations have shown promising boosts in performance through the use of autonomous agents for a few reasons. Most notably, there are typically certain levels of hazard involved in exploration, and search and rescue that limit or prevent a human agent's performance. In most cases, a robotic agent is less susceptible to the same environmental hazards as a human. Use of robotics can help eliminate the risks of injury, disease, and death of humans involved in an operation. The other advantages to using autonomous robots is the diverse number of sensors that a robotic agent can use, as well as their ability to analyze data quickly and without bias. Sensors, such as an infrared camera, can accurately collect data that humans do not have the capability to observe themselves. Large amounts of sensor data can also be processed and analyzed by a computer more efficiently than a human. This supports the idea that a robotic agent

can operate at higher performance levels than a human could in many exploration-based operations.

The basics of modeling and controlling agents within an environment were studied in the works of Poole et al. [28], LaVelle [19], and Sutton et al. [35]. Using this platform of knowledge, we next reviewed recent implementations of autonomous control designed for real-world environments. There is an abundant amount of existing research on the use of autonomous robots for exploration. Both intelligent hardware and software approaches have been researched to create robotic agents that can safely and efficiently navigate in an environment. Many of the hardware focused solutions explore clever designs for mobility features that enable an agent to traverse hazardous and complex environments [7, 8, 13–15, 17, 18, 33]. Software solutions typically focus on the application of AI control schemas to plan hazard avoidance, and maximize efficiency [4, 9, 11, 27, 30, 34, 36]. Each of these autonomous exploration-based research experiments share three key components: there is a robotic agent with a set of actions that it can perform, the agent must use intelligence to navigate in an environment, and the agent is goal driven. While these components are present in all of these experiments, there is a large variation in the use cases and control schemas. Some are designed for mapping indoor or outdoor environments [27, 34, 36], others focus on hazard avoidance [11, 30], use of multiple agents working together [9, 30], and operation efficiency [4]. A variety of intelligent approaches for controlling the agents range from the use of Bayesian prediction models [30], artificial neural networks [36] and machine learning [4] to name a few. Each of these experiments focused on AI controllers that were hand crafted for a specific goal. The SCOUt project aims to address all of these use cases for autonomous agents in a unified operation setup process and adaptive control schema.

While preliminary research did not uncover any existing work on a unified process for the setup and control of exploration-based operations, the idea of a single adaptive controller for completing multiple goals is not a new concept. Arora et al. [1,2] have created control schemas that are both adaptive to an agent's capabilities, as well as a diversity of environments. In

their research conducted in 2017 [1], they use a high-level approach to generate a controller that can model and analyze scientific data in a task-based approach across multiple goals. Their later research in 2018 [2] achieves efficient path planning and sensor usage policies through an adaptive Bayesian framework. SCOUt exhibits similar functionality through its use of a memory-based learning model to plan actions that will maximize goal completion and minimize damage and energy usage for a variety of operations. Memory-based control schemas have also been used in existing experiments for autonomous decision models. Experiments such as [12, 38], used a genetic algorithm (GA) to generate decision policies through the use of existing knowledge. Arulkumaran et al. [3] cover an approach to handling memory sets, as large pools of memory can yield more accurate results but lead to issues of increased complexity and storage requirements. There has also been a heavy use of machine learning (ML) in the field of robotics. Specifically, [3, 4, 16] all implement reinforcement learning (RL) systems in action decision models. SCOUt's decision model follows a similar approach to [16], using an adaptive process for choosing actions through the use of a reward system. The reward system will generate long-term and short-term rewards for each agent's performance in an operation. Long-term rewards reflect the overall outcome of an operation, while short-term rewards reflect the outcome of each action taken during the operation. SCOUt uses a variation of RL by building a memory of state-action rewards (SAR) to predict and critique the performance of agents acting within an environment.

Our memory-based learning model primarily borrows concepts from both partially observable Markov decision process (POMDP) and learning classifier systems (LCS). A POMDP is a generalized decision making process used in situations when an agent's state-space cannot be fully modeled [29, 31]. A state-space is a collection of finite possible configurations of a problem that could occur during a defined operation. For example, in the game of chess, the state-space is a collection of legal game positions that could occur based on the moves that each player makes within the game. When states cannot be fully modeled, POMDPs are applied to create a probability distribution of states that might result from the actions

an agent can take. The predicted state-space is then used in action decision models. In this project, agents must navigate through previously unknown environments. Because the agents set of actions and the types of features within each environment will vary from operation to operation, the state-space cannot be modeled in a finite set. For this reason, POMDP cannot fully be apply to our problem. However, SCOUt borrows the general process of making a probability distribution of potential rewards for each action, base on the current state of an agent. A memory of past SARs is used to predict future rewards that an agent will receive for each valid action it could take.

LCS models combine data discovery systems with a ML component to build a set of rules that can be applied in a piecewise approach to decision making [32]. We see this reflected in the combination of SCOUt's state comparison system 4.2.3 with the reward system 3.4.2. While SCOUt's decision model uses an LCS approach in regards to state comparisons, the full decision process is more broad in the sense that multiple comparisons are made to contribute in action-reward prediction based on a memory of previous SARs. When information about the environment is discovered, the new agent state is passed through piecewise functions to compare it against states in the learned memory. These functions use weight sets that were optimized using a genetic algorithm (discussed in section 4.2). State comparisons will then help the decision model predict rewards that an agent will receive for each of the valid actions it can choose.

Chapter 3

SCOUt

This project explores the reliability and flexibility of using a single intelligent controller to complete exploration-based operations in diverse environments. The Surveillance Coordination and Operation Utility (SCOUt) is used to generalize environments, agents, states and actions into abstract data structures. This data then builds a platform for creating controllers, running simulations, and visualizing outputs.

3.1 Framework

Several coding languages and libraries are used in this project to provide a simple and expandable framework. It is laid out in a client-server architecture (figure 3.1) to allow separation of data handling and data visualization. The server portion provides full functionality to generate unique environments, build agents and controllers, run test operations, and collect results. Data structures on the server side are implemented using an object-oriented architecture of traits, classes and class instances. Traits are abstract objects that can be inherited by multiple classes. Each class that inherits a trait can add specific values and behaviors to the object. All classes that inherit from the same trait can be handled using the same logic, yet each class can behave in a unique way. Different instances of a class can then be declared for repeated usage within code. This data architecture was chosen to give the SCOUt framework

simple extendibility into future projects. The client is a graphical user interface (GUI) for requesting actions to be executed by the server and visualizing the data structures returned. Because the majority of the data structures used in this platform are abstracted, the front end can be generalized to handle any new classes created without any maintenance required for the GUI.

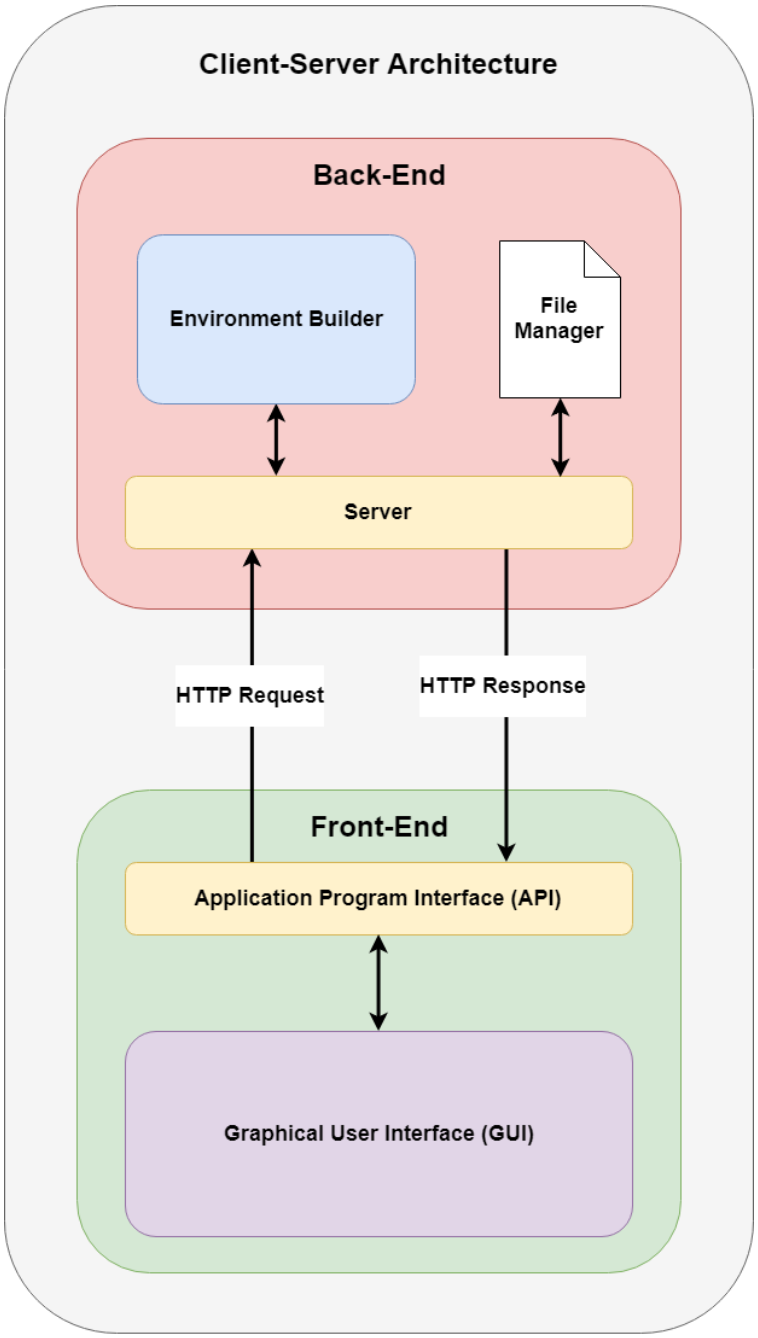

Figure 3.1: Client-server architecture of SCOUt's framework.

3.1.1 Simulation Back End

The back end is written in the Scala programing language. Scala is a Java based, paradigm language that combines object-oriented and functional programming methods. Object-oriented programming provides the flexibility needed for the trait-class-instance architecture, while functional programming provides immutability when working on large sets of diverse data. Immutability is important when working with a large code base as it will assure that data types and values are not changed once set. All data storage and manipulation takes place on the back end of the platform to ensures consistency. Data is only imported or exported on the back end in two scenarios: file storage and client-server communication. In both cases, it is assumed that immutability is maintained. File storage is the only case where data is open to manipulation outside of the back end. Client-server communication only allows variables to be passed into the back end via requests, and a copy of data structures are returned to the front end for visualization only. Any alterations on the front end will have no effect on the original copy on the back end. To allow storage and communication, the back end encodes and decodes data into JSON objects. When imported, JSON objects are immediately decoded and parsed into Scala data structures before usage. The circe Scala library [6] is used for the encoding and decoding of JSON data. Circe provides integration of JSON objects in the Scala language to allow seamless encoding and decoding. Communication for passing and receiving JSON objects between the front and back end is achieved with the http4s library [26]. The SCOUt server is setup using http4s' blaze-server to create a local service for handling HTTP communication.

3.1.2 Visualization Front End

The platform's front end is built around Electron [25], a framework that allows building a native desktop application with JavaScript, HTML and CSS. The GUI is written using all three of these languages. HTML structures the page within Electron, CSS provides styling and JavaScript handles all of the logic. The SCOUt platform uses D3 [5], node-fetch [21]

and jQuery [37] JavaScript libraries to assist with data visualization and communication to the back end. D3 (Data-Driven Documents) is a visualization library that uses SVG (an XML-Based format for vector graphics) to create graphical representation of data sets. Node-fetch is used for HTTP communication with the back end through HTTP request and response handling. jQuery provides integration with JSON data that is passed back and forth between the client and server, as well as several functions to simplify working with DOM elements within HTML. Node Package Manager (NPM) [22] is used to maintain all of the dependencies between Electron, the three languages and the JavaScript libraries on the platform's front end. In addition to dependency management, NPM has packages of its own that simplify the process for compiling code into a single file for Electron to handle. The Babel [23] package "transpiles" JavaScript into a browser friendly format, then webpack [24] integrates the resulting JavaScript into the HTML code for a single JavaScript file for Electron. The GUI can then be launched using an NPM script to compile all of the code, launch Electron and load the content.

3.2 Environment

Modeling a real-world environment in a simulation is tricky. Each model needs to balance simplicity and coverage. If too much is left out of the model, it won't reflect real-word scenarios. On the other hand, attempting to model too much can be impractical as it consumes effort and resources that could instead be applied to running real world experiments. For SCOUt's simulation platform, environments are modeled as a high-level class containing a collection of sub-classes that together form a simplified representation of a real world environment (figure 3.2).

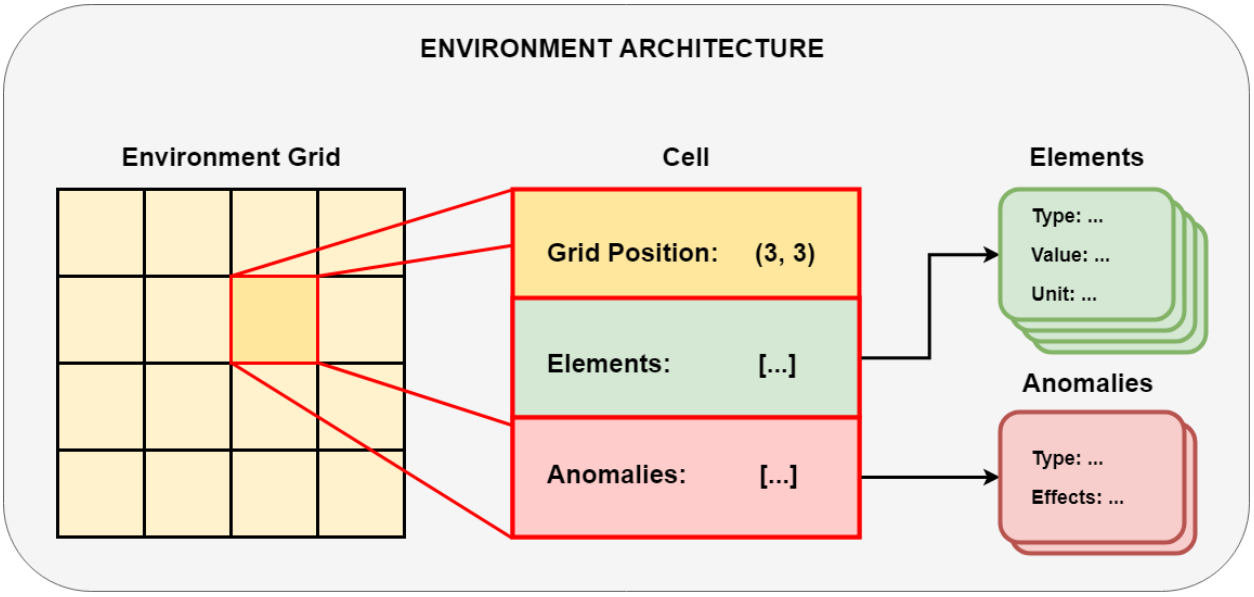

Figure 3.2: A visual model of SCOUt's `Environment` class architecture. An environment is made up of cells which hold information about the environment. This information includes any present element type or anomaly.

The `Environment` class (Appendix 7.1) holds an nxm 2D grid of uniformly sized sxs square cells, where nxm is the total area of the environment and sxs is the area each cell represents within the environment. A `Cell` (Appendix 7.2) holds x and y coordinates for its relative position within the `Environment`'s grid. These coordinates do not reflect the actual size of the cell, only an index value for the order they appear within the 2-dimensional array data structure (e.g., see "Position" of "Cell" in figure 3.2). The environment's scale can easily be applied to the physical location of a cell as it is a shared global attribute within the `Environment` class. If an element type or an anomaly is present within the area of the environment that the cell covers, it will be stored in an appropriate list within the `Cell` instance.

An element can be any measurable attribute within an environment. For example, temperature, elevation and decibel levels are all attributes of the environment whose values can be measured. All element types are all generalized by the abstract trait, `Element` (Appendix 7.3). The trait has a set of attributes that an inheriting class must define to identify the element's type and how it behaves. Each element type has `name` and `unit`

attributes that are used for identification and denoting the unit of measurement. The `value` attribute holds a numerical value for the measurement of each instance. For example, an instance of the `Elevation` class (Appendix 7.4) stores a measurement of the elevation level in feet for the area within a cell. The `radial` flag, `lowerBound` and `upperBound` attributes guide and limit the values that can be set for each element type. These are used when procedurally generating an environment (covered in section 3.5).

An anomaly is any object that may be of significance to an agent, such as a human or precious mineral. Anomalies each have their own effects on element values in the environment around them. Like `Element`, `Anomaly` and `Effect` (Appendix 7.5 and Appendix 7.6) are both defined as a single trait that specific classes can inherit from. An `Anomaly` class can occupy multiple cells, but must occupy at least one cell in an environment. Anomalies can also have multiple effects on the values of different element types in its surrounding area. Each `Effect` class defines a "seed" element and a range of the effect. The `seed` attribute holds a specific instance of an `Element` class that will be present in the `Cell`s that the anomaly occupies. The `range` then defines the radius of the area beyond the attribute's position that the effect will "radiate." The term radiate is used because the effect will alter the element type's values in surrounding cells based upon how close they are to the source of the effect (the anomaly).

For example, `Human` (Appendix 7.7) is an anomaly that takes the area of a single cell, and affects the temperature and decibel values in their environment. If the human is much louder than the ambient noise level in the environment, there will be a sharp spike in decibel values in cells nearest the human, with a diminishing increase for values in surrounding cells. Figure 3.3 shows a heatmap of decibel values in an environment that a human is present within.

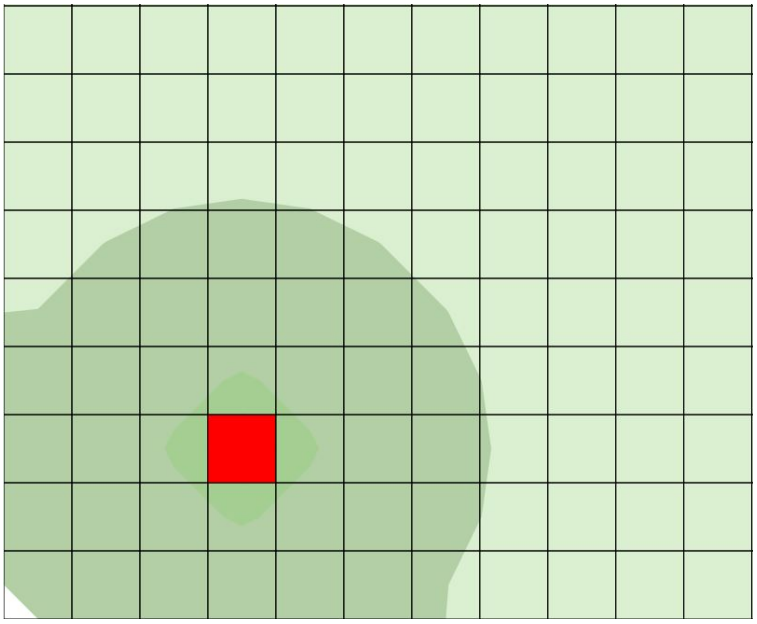

Figure 3.3: Heatmap of the layer of decibel values within an environment that contains a human. A darker shade of green represents higher values of decibel levels, lighter shades represent lower values, and the human's location is shown in red.

One last important data structure is a `Layer` (Appendix 7.8). A layer is designed in the same 2-dimensional structure as the `Environment` grid, but holds a collection of `Element` instances instead of cells (figure 3.3 is an example of a layer of decibel elements). While layers are not direct members of the `Environment` class structure, they are crucial to building and analyzing the environment. For this reason, instances of the `Layer` class are only generated on demand through method calls.

3.3 Agents

Agents within this experiment are modeled based on a set of abilities that dictate how it will interact within an environment. The `Agent` class defines these with state related attributes, physical limitations, a set of sensors that can be used, and a controller for decision making (Appendix 7.9). The state related attributes of an `Agent` are health, energy level, an internal map, and its current position relative to the `Environment` grid. Because SCOUt is focused on exploration-based operations, agent behaviors can be divided into two categories of actions:

movement and scanning. The agent can attempt to move one cell at a time in any of the four cardinal directions. This allows the agent to reassess after each movement attempt. Agents perform scanning actions to collect information about the immediate environment. Any new information learned about the environment is stored in the agent's internal map. The type of scanning actions that an agent can perform is based on the set of sensors the agent has equipped. The agent's controller is in charge of analyzing the current state and deciding the next action to be performed. To simulate the interactions that will occur between an agent and an environment, mobility and durability attributes are defined for each agent. These will dictate how an agent can move within the environment and the types of effects each element type within the environment will have on the agent. Figure 3.4 shows the internal agent architecture and how it applies to an external environment.

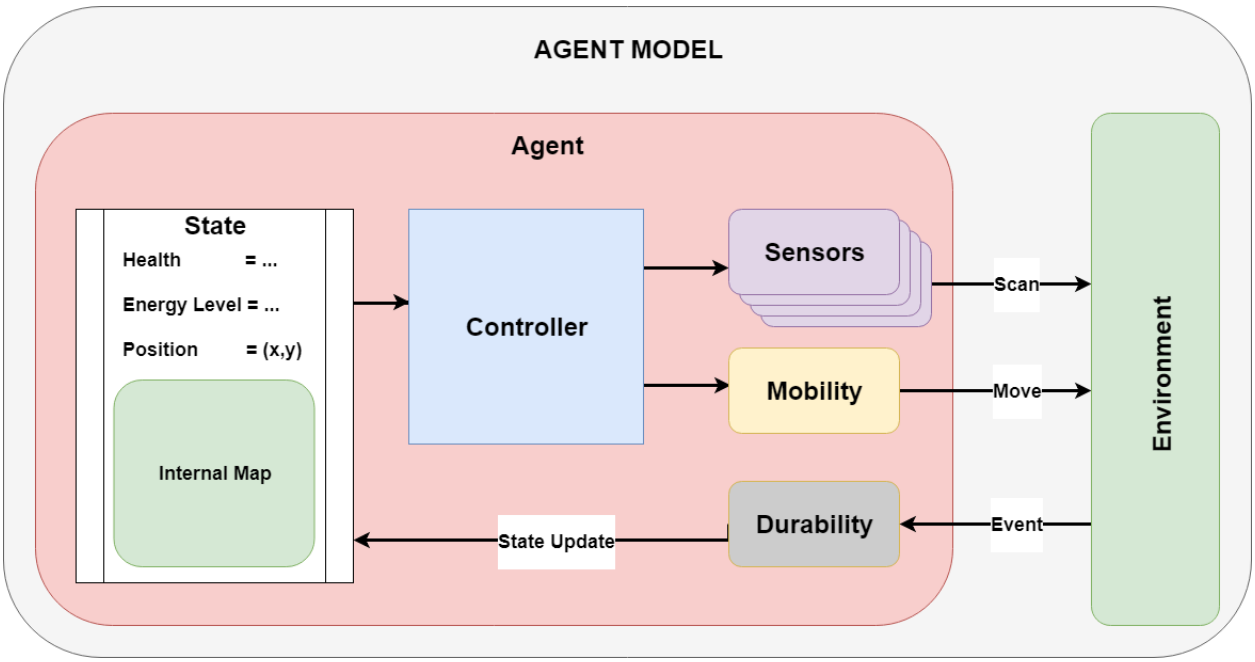

Figure 3.4: Visual model of the `Agent` class and how it interacts with an environment. A controller makes action decisions based on the agents current state. The agent will then attempt to perform the selected action within its environment. The agent's state will then be updated based on the resulting event.

3.3.1 Sensor

Sensors are defined using the same trait-class-instance architecture as `Element` and `Anomaly`. The `Sensor` class (Appendix 7.10) models a scientific instrument that could be used for gathering data measurements of a specific element type. Each class defines the element type it is able to measure, the energy it costs to use, its effective range, and two flags indicating if the element type is hazardous or considered an indicator for the given operation. When performing a scan, the sensor will sweep 360 degrees around the agents location and gather data within the circular area (figure 3.5). The circular scan area is calculated with the sensors range as the radius and the agent's position as the center. Any element values that were previously unknown to the agent are then added to the internal map. Hazardous elements are flagged in a `Sensor` class instance when its element type has the potential to cause harm to the agent. The indicator flag is set when the element type is believed to be an important data type related to the operation. For example, if an agent was searching for a human, temperature and decibel sensors would be flagged as indicators since their values can potentially help lead the agent to the human.

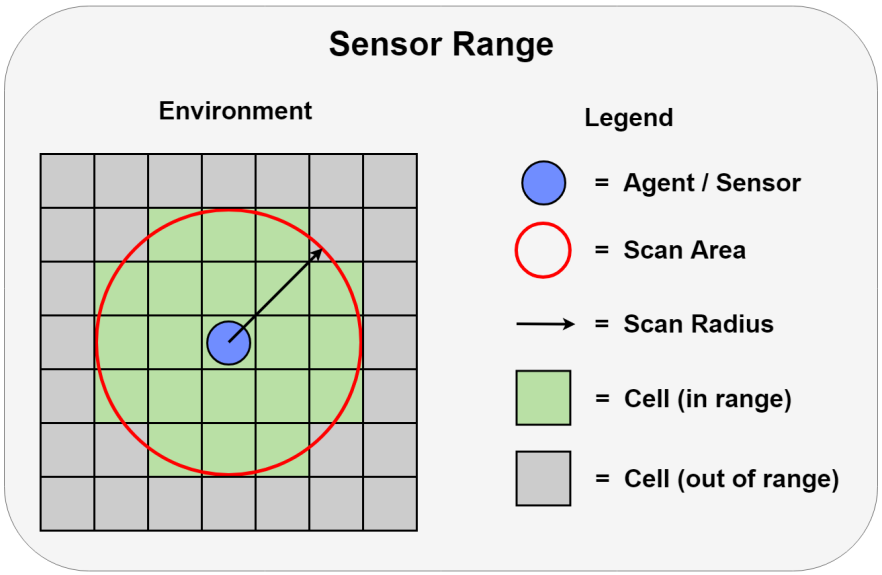

Figure 3.5: Diagram of a sensor's effective range within a grid of environment cells. The sensor will collect present element values for cells in range. The type of values collected is dependent upon the element type that the sensor is designed for.

3.3.2 Mobility

`Mobility` is a standalone class that will determine the physical limitations of the agent when moving within an environment (Appendix 7.11). In real-world scenarios, agents often have to trade off functionality and mobility. For example, if a drone was modeled as the agent, it would have a higher range of mobility, but would likely sacrifice the number of sensors that could be carried. A wheeled robot loaded with multiple sensors would likely have decreased mobility but could collect a wider variety of data. An agent's mobility is defined by the maximum slope an agent can climb, the minimum slope an agent can traverse before taking "fall damage," a resistance factor, and the energy cost required for movement. The resistance factor is used to scale the amount of fall damage that the agent may take. Movement and slope costs are used to calculate how much energy is used when attempting a movement action. The total amount of energy used is calculated based on the distance moved, and the slope of the elevation between the agent's current position and the position that it attempts to move to (equation 3.1).

$$energyCost = (1.0 + (slope * slopeCost)) * dist * movementCost \tag{3.1}$$

Equation 3.1: Equation to calculate the energy required for an agent to move within an environment. This is based on the agent's *movementCost* and *slopeCost*, and the physical *slope* and *distance* that is traveled.

3.3.3 Durability

Like many other data structures in this platform, `Durability` (Appendix 7.12) factors are defined per element type. These factors model how an agent will be affected by different element types they come in contact with during an operation. For example, consider an environment with pools of water in it. Most robots would be damaged when emerged in water, but an amphibious robot's durability could be modeled such that it would be impervious to water damage. Durability is defined by an upper and lower value threshold, and a resistance

factor. The thresholds define what values of an element type the agent can be exposed to before it begins to take damage. The resistance factor between 0 and 1 (0 being no resistance and 1 being immune) then influences how much damage the agent will take when in contact with values beyond these thresholds (equation 3.2).

$$damage = |v - threshold| * (1.0 - resistance) \tag{3.2}$$

Equation 3.2: Damage calculation based on an agent's *resistance* to an element type with value v that is above or below a durability *threshold*.

3.3.4 Actions

Agents interact with an environment via movement or scanning actions. These two categories cover the exploration and research aspects that are required for most surveillance operations. The agent's controller is in charge of deciding what action to perform. Figure 3.6 shows an example of an agent within an environment considering both a movement and a scanning action, and the new states that would result from selecting either of them.

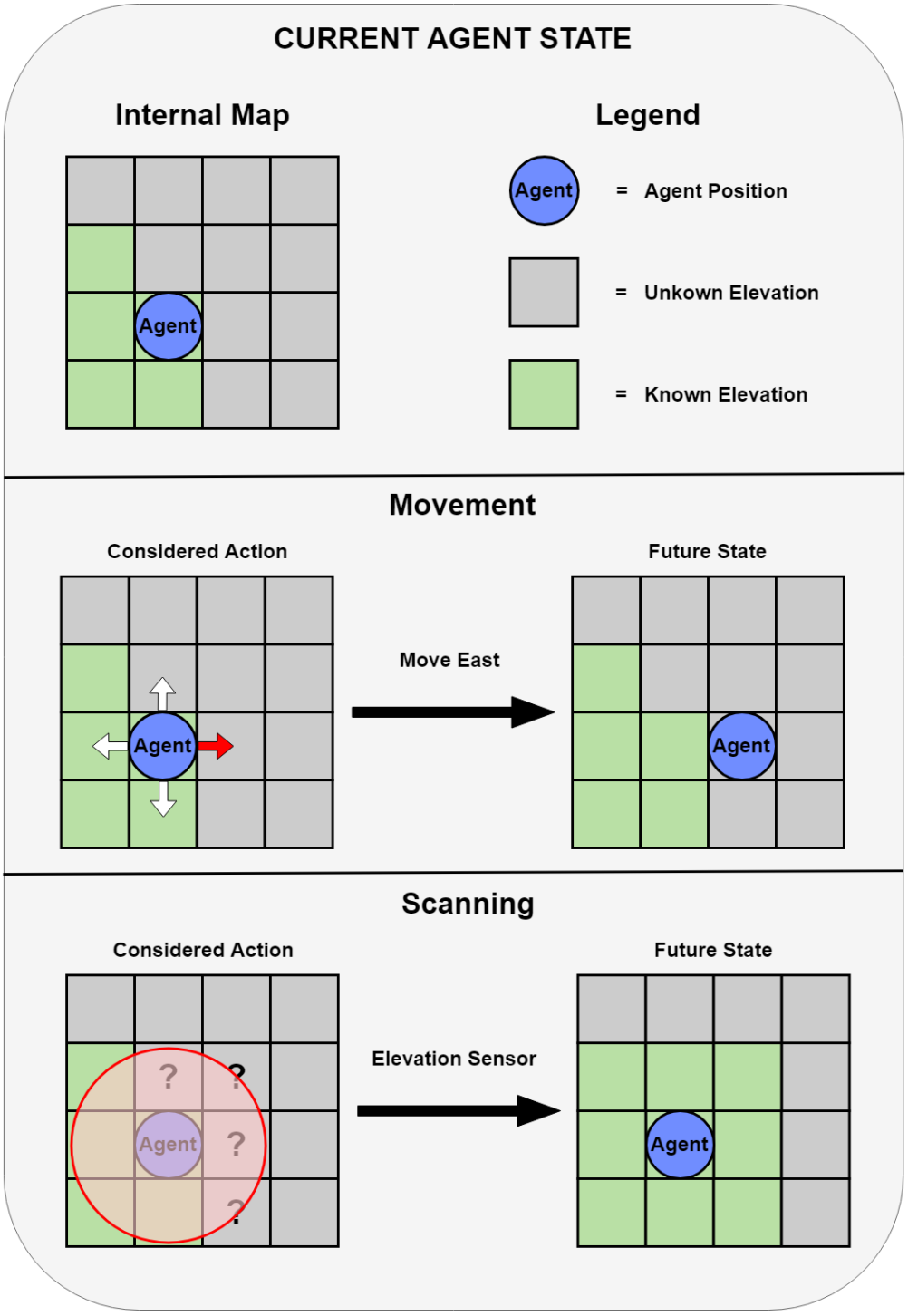

Figure 3.6: Example of an agent considering a movement action and a scanning action, along with the resulting states caused by each of these actions.

In simulation, movement is handled by changing the agent's current position to an adjacent cell in one of four direction ("north," "south," "west," or "east") based on the orientation of

the environment's grid of cells. Moving a single cell at a time gives the agent the opportunity to reassess its current state before selecting the next action. Distance covered by successful movement will inherently be equal to the size of the cells within the simulated environment. Each time an agent attempts to move to a new cell, elevation levels will be compared between the current and new cell to check if movement is possible, or if it results in damage (based upon the agent's mobility). After the action has been attempted, changes to health and energy level are calculated based upon the agent's durability factors. If the movement action is successful, the current position is updated.

An Agent can also perform scans of the environment using equipped sensors. For each cell that falls within the sensor's search radius, the value for the sensor's given element type is extracted. These values are then added to the agent's internal map if they did not previously exist there. Through repeated scanning, the agent will begin to fill its internal map with data collected from the surrounding environment. Data collected in the internal map can then be used by the controller to determine what actions would be most beneficial to the goal at hand.

3.3.5 State Representation

For controllers to intelligently choose among actions, they need to have sufficient data about the agent and surrounding environment. The agent's position, health and energy level can easily be analyzed, but the internal map containing the known environment is a very large data structure to analyze each time the controller must choose an action. For this reason, the data structure is simplified to reduce memory usage and the computational effort required to analyze a state. `AgentState` (Appendix 7.13) is a minimal data structure that contains all of the useful information necessary for a controller to make intelligent decisions. Instead of a 2-dimensional array of cells, the internal map is represented as a list of `ElementState`s (Appendix 7.14), where each `ElementState` is a summary of the data known about a specific element type.

Element states contain useful information about what is currently known about a specific element type during an operation. The indicator flag can cue the controller on whether the element type is being analyzed in order to progress the goal at hand. If the goal was to map out the elevation levels in an environment, the elevation `ElementState` would be flagged true. If the goal was to find a human, the temperature and decibel `ElementState`s would be marked true, as irregular changes in these values could indicate the presence of the human. The hazard flag is used to mark any element that could potentially cause harm to the agent. For example, the presence of water, large changes in elevation, and extreme temperatures could potentially cause damage and would be flagged as hazardous. We also track the percent of known element values that are within range of the corresponding sensor. Technical information of each `ElementState` is divided into four quadrants, where each quadrant has its own state. Because agent movement is defined as north, south, west and east, we can collapse known information from the internal map into four quadrants (figure 3.7).

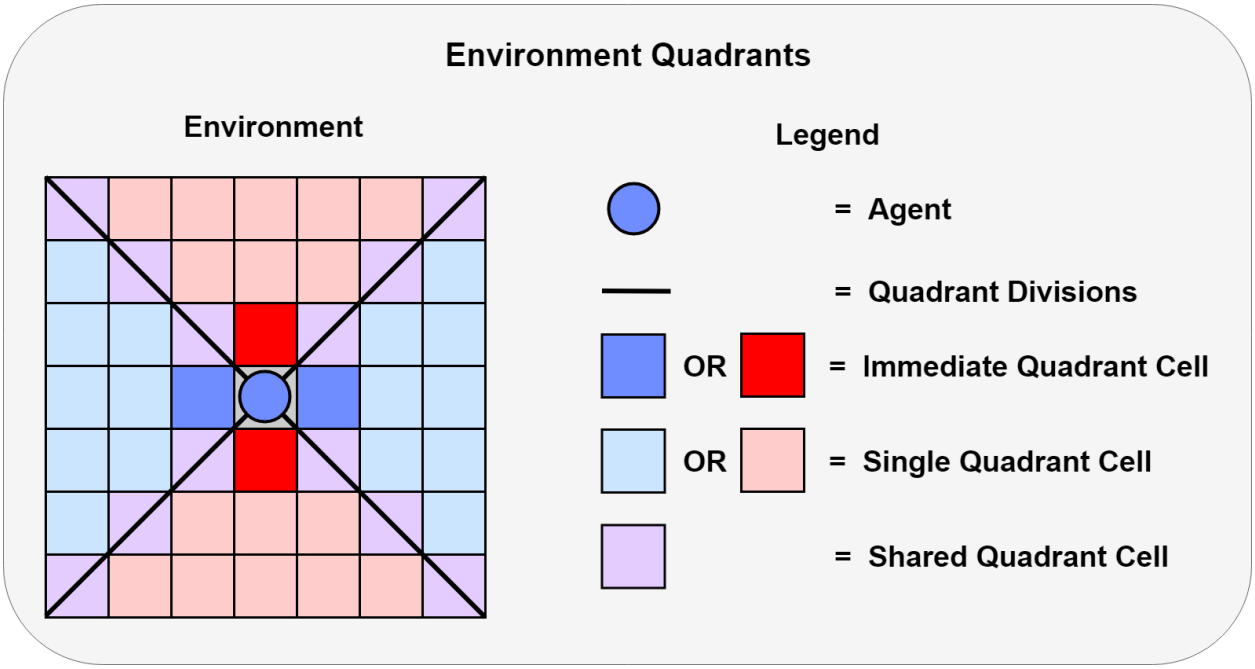

Figure 3.7: Diagram showing how an environment is divided into quadrants relative to an agent's position.

Each quadrant is defined using a `QuadrantState` (Appendix 7.15). To reflect the amount

of information that is known about an element type within the quadrant, the `QuadrantState` first looks at the percentage of cells in the agent's internal map that hold known values. If there are known values within the quadrant, the averages will be reflected in the `QuadrantState`. Additionally, if the value in the cell immediately adjacent to the agent's current position is known, it will also be reflected in the `QuadrantState`. Average and immediate values are stored as "differentials" relative to the value in the current cell. Average differential takes the difference between the current cell's value and the average of all known values in the quadrant's collection of cells. Immediate differential takes the difference between the current cell's value and the cell immediately adjacent to it. In doing so, the controller will have data that is relative to the agent's current position.

3.3.6 Controller

A controller is the goal driven decision model that an agent will use when navigating within an environment. The agent will pass its current state to the controller, along with a list of valid actions that it can execute. The controller will then decide which action to take. Controllers are defined as an extendable trait (Appendix 7.16). The `Controller` trait requires that inheriting classes define a `setup` and `shutdown` function to perform any initializations or final actions once an operation has ended, and a `selectAction` function. The `selectAction` function is where the controller's decision model is implemented. This function takes the list of valid action and the current `AgentState` as input, and returns a single action for the agent to perform. Specific Controllers and their schemas are analyzed and discussed in chapter 4.

3.4 Operations

To explore the efficiency and adaptability of the intelligent controller, many different scenarios need to be simulated. All simulations are made up of three main components: an agent, a goal, and the environment. Different combinations of each component allow the creation of

a large variety of scenarios. Each simulation follows a defined process called an operation (figure 3.8). An operation will simulate an agent's attempt to complete a goal within an environment, record data for each event that occurs between the agent and environment, and the final outcome. These data collections are denoted as short-term and long-term events respectively.

Short-term events are collected each time the agent performs an action. They include:

- The agent's state when the action was selected
- Any changes to the agent's internal state (health or energy reduction)
- If the action performed was successful (could it move, did it have enough energy to complete the action)
- A short term reward for the outcome of the action

A long-term event is only collected once at the simulated operation ends. It includes:

- Status of internal variables (health and energy)
- Number of actions taken during operation
- Goal completion
- An overall long term reward
- Long term reward given to each action taken

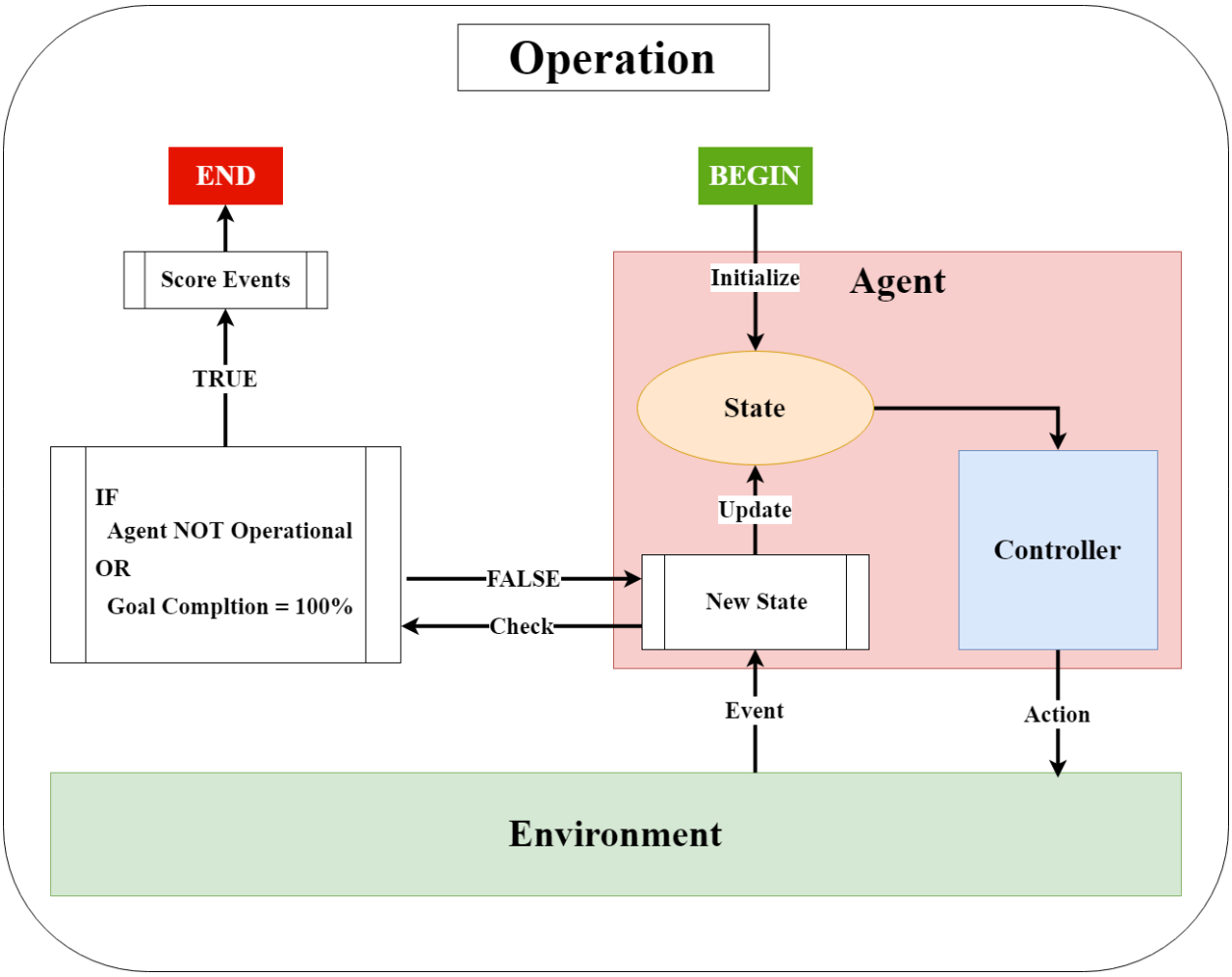

Figure 3.8: Diagram depicting the flow of an operation where an agent is attempting to complete a goal within an environment. The agent will continuously select actions to perform until the goal is successfully completed or it is no longer operational. An agent is no longer operational if its health or energy are depleted due to the events that occur as it interacts with the environment. Once an operation has completed, each action the agent attempted will receive a short-term and long-term score.

3.4.1 Goals

Because SCOUt is designed for exploration, two goal types are analyzed: anomaly searching and element mapping. This is not to say that SCOUt would be limited to operations which only involve these types of tasks. SCOUt is intended to be *integrated* with other tasks. For example, if the entire task of a robot was to traverse a hazardous area to find a certain mineral for extraction. SCOUt would be used to guide exploration in the environment and

detect the mineral. When the mineral is found, SCOUt's process would then be completed successfully, and a separate process could take over for the actual extraction, or the location could be recorded by SCOUt and another agent could be deployed. After the other agent or process completes its task, SCOUt could continue to search for more deposits of the mineral or return to base. For anomaly searching goals, the agent is required to find a specified anomaly within an environment. This tests SCOUt's ability to use environmental clues to track down the anomaly. For example, if the agent was looking for a human after a natural disaster, it could use data such as temperature and decibel readings to locate the person. The element mapping goal is straitforward. The agent must map out as much data about the specified element type as possible.

3.4.2 Rewards

In addition to goal completion, an agent must also be observant of its health and energy throughout the operation. The more efficiently an agent can complete an operation the better. The overall performance of an agent is measured by both its ability to complete the task at hand, and the safety and efficiency of the actions taken. These performance measurements are calculated on a short-term and long-term basis and come in the form of "rewards."

Short-Term Rewards

Short-term rewards are given each time an agent performs an action (algorithm 1). Energy and health depletion are major factors in this reward. If the action required an excessive amount of energy or resulted in damage to the agent, the reward is decreased. Other factors that come into play depend on the specific action taken. If the agent attempted to move to a new area and failed to move (e.g., a hill was too steep to climb) a deduction is made. A small increase in reward is applied if the agent moves into an unexplored area. If the agent uses a scanner, the reward is calculated to reflect the amount of new information learned.

This penalizes the agent from using a scanner multiple times in the same area, as it is not an efficient use of energy.

Long-Term Rewards

Long-term rewards are calculated once an operation is over using algorithm 3. Operations end when the agent has successfully completed its goal or has depleted its health or energy. To reflect these scenarios, reward is determined by the goal completion, remaining health, and remaining energy. Even if a goal is completed, the agent could receive a low score if it was "reckless" and took lots of damage or used large amounts of energy. The long-term reward is then propagated backwards through all the actions that were taken. The actions performed immediately before the end of the operation are given highest score. Previous actions then receive diminishing reward based on equation 3.3.

[H]

$$actionLongTermReward(i) = longTermReward * 0.9^{\frac{i*10}{n}} \tag{3.3}$$

Equation 3.3: Equation representing the diminishing long-term reward that each of the n actions will be assigned based on the $longTermReward$ the agent received for completing an operation, where i is the current action's index within the operation event log.

3.5 Environment Builder

The SCOUt `EnvironmentBuilder` is a tool for creating diverse environment models. The tool is highly abstracted so that more details can easily be added to the model as needed while still maintaining a defined build process. Environments are procedurally generated based upon a collection of parameters called an environment template. These templates require minimal input to build unique environments. An environment can be tweaked or even built entirely by hand, but the procedural generation process removes this overhead. Procedural generation process:

Algorithm 1 Calculate the short-term reward for an event caused by an agent's action within an environment. W_{item} denotes the attributed weight for *itemReward*.

Require: $W_{health} \in [0, \infty)$
Require: $W_{energy} \in [0, \infty)$
Require: $W_{action} \in [0, \infty)$
Require: $maxHealth \leftarrow 1$
Require: $maxEnergy \leftarrow 1$
Require: $actionType \in \{movement, scanning\}$
Require: $cellVisitedBefore \in \{true, false\}$
Require: $scanDatas \in [0, \infty)$
Require: $newScanDatas \in [0, scanDatas]$
Require: $remainingHealth \in [0, 1]$
Require: $damageTaken \in [0, 1]$
Require: $energyUsed \in [0, 1]$
Ensure: $shortTermScore \in [0, 1]$

```
if remainingHealth == 0 then
    return shortTermScore ← 0
else
    damageInverse ← (maxHealth − damageTaken)/maxHealth
    energyUseInverse ← (maxEnergy − energyUsed)/maxEnergy
    healthReward ← damageInverse × W_health
    energyReward ← energyUseInverse × W_energy
    if actionType == movement then
        if cellVisitedBefore == false then
            actionReward ← W_action
        else
            actionReward ← 0
        end if
    else
        actionReward ← (newScanDatas/scanDatas) × W_action
    end if
    rewardsTotal ← healthReward + energyReward + actionReward
    W_total ← W_health + W_energy + W_action
    return shortTermScore ← rewardsTotal/W_total
end if
```

Algorithm 2 Calculate the long-term reward for an agent's performance in an operation. W_{item} denotes the attributed weight for $itemReward$.

Require: $W_{goal} \in [0, \infty)$

Require: $W_{health} \in [0, \infty)$

Require: $W_{energy} \in [0, \infty)$

Require: $goalCompletion \in [0, 1]$

Require: $remainingHealth \in [0, 1]$

Require: $remainingEnergy \in [0, 1]$

Ensure: $longTermScore \in [0, 1]$

$gReward \leftarrow goalCompletion \times W_{goal}$

$hReward \leftarrow remainingHealth \times W_{health}$

$eReward \leftarrow remainingEnergy \times W_{energy}$

$rewardsTotal \leftarrow gReward + hReward + eReward$

$W_{total} \leftarrow W_{goal} + W_{health} + W_{energy}$

return $longTermReward \leftarrow rewardsTotal / W_{total}$

1. Environment Template is given
2. Builder initializes a grid of empty cells
3. `ElementSeed`s are used to populate each present element type into the gird of cells
4. `TerrainModification`s are applied to manipulate their related element(s)
5. Anomalies are placed randomly within the environment
6. Anomaly effect(s) are applied to corresponding element(s) in neighboring cells

3.5.1 Environment Templates

Each `EnvironmentTemplate` (Appendix 7.17) will act as a guide in the creation of an instance of the `Environment` class. A template can create similar, but unique environments each time it is used by the `EnvironmentBuilder`. This allows testing and training agent controllers multiple times in similar conditions, while still providing a dynamic range of sce-

narios that the agent may face in each generated environment. Each template is comprised of the name, dimensions and scale of the environment along with lists of `ElementSeed`s, `TerrainModification`s and anomalies to be applied.

3.5.2 Element Seeds

The environment builder begins by procedurally generating one layer of elements at a time. Each `Element` class has a companion class called an `ElementSeed` which holds parameters used to produce a layer of its element type, and a unique function defining how procedural generation will take place to produce the layer. The generation of each layer is modeled on how the element type's values may vary in a real-world scenario. Parameters within each Seed are set to default values that can also be overridden by creating a new instance of the `ElementSeed` (Appendix 7.18). This can change how the values within the layer will vary.

The environment builder will use each `ElementSeed` to produce each layer of elements. Resulting layers will be temporarily stored in a list until the end of the build process so they can easily be manipulated before being stored into corresponding cells within the `Environment` grid. Some layers are easier to generate than others. Latitude and Longitude layers can be generated by calculating the distance each cell is from the origin point on the `Environment` grid). For an example, let's look at the `ElementSeed` for producing the elevation layer.

```
case class ElevationSeed(
  val elementName: String = "Elevation",
  val average: Double = 0.0,
  val deviation: Double = 0.15
) extends ElementSeed {
  def randomDeviation(mean: Double, scale: Double): Double = {
    val lowerBound = mean - (deviation * scale)
    val upperBound = mean + (deviation * scale)
    randomDouble(lowerBound, upperBound)
  }
  def buildLayer(height: Int, width: Int, scale: Double): Layer = {
    val layer: Layer = new Layer(AB.fill(height)(AB.fill(width)(None)))
    for {
      x <- 0 until height
      y <- 0 until width
    } {
      val value = randomDeviation(average, scale)
      layer.setElement(x, y, new Elevation(value))
    }
    layer.smoothLayer(3, 3)
    return layer
  }
}
```

The `ElevationSeed`'s `buildLayer` algorithm first initializes an empty `Layer`. Next, it sets each (x,y) coordinate in the layer to an a random elevation value within a standard deviation of the average value provided [(average - deviation), (average + deviation)]. Once every Cartesian position has been set to an instance of elevation, the layer is then smoothed. Smoothing is a function defined within the `Layer` class that will reduce strong variations of element values within the layer. For elevation, this would equate to transforming a highly rigid surface into a smoother, more natural surface.

3.5.3 Terrain Modifications

Terrain modifications influence the basic landscape of the environment. After each `ElementSeed` has produced a layer for the environment, `TerrainModification`s are applied one after another, taking care not to overlap modifications (e.g., we wouldn't want a hill to overlap with a valley and cancel each other out). Each modification represents a severe alteration of one or more element type layers within the environment. Following a similar process laid out by Doran and Parberry [10], desired alterations are incorporated into the environment while allowing unique variations of each to develop. Their controlled procedural generation process is used to produce landmasses that potentially have bodies and channels of water. SCOUt's environment builder generalizes this process and extends it to allow multitudes of element types to be modified. The `TerrainModification` trait provides an extendable template from which all types of modifications that can be applied (Appendix 7.19).
For an example, lets look at elevation again.

Here we have an `ElevationModification` that will allow us to create hills and valleys within the environment. Again following the approach of [10], random, unmodified (x,y) positions are selected from the layer to begin with and updates their value to the specified modification value. The modifier then performs "walks" to random, unmodified neighboring cells, updating their values within a standard deviation of the specified modification value. These walks continue until the specified coverage area has been modified, or until there are no neighboring cells that can be modified. A special layer smoothing algorithm is then applied to the elevation values in the modified area, as well as the immediate surrounding unmodified area to reduce rigidity and give a more natural change in values between neighboring cells. This type of smoothing applies a given slopping factor within the modified area, allowing the `ElevationModification` to generate gentle hills or valleys or sharp cliffs depending on the slope defined.

Algorithm 3 Algorithm for applying a terrain modification to a layer of elevation values denoted as cells. Note that the `Layer` is a collection of `Elevation` objects, not `Cells`. However, it is easier to think of each location in the layer's grid as a cell, and is therefore denoted as such. The algorithm begins by initializes a random, unmodified cell. Next, it will modify neighboring cells until it has modified the requested `coverage` area or there are no unmodified, neighboring cells remaining. Once the modification process is completed, a smoothing function will be applied to cells in the modified area and cells in the surrounding area to "blend" the modified area into the rest of the environment (this prevents a plateau effect). The number of cells outside of the modified area that are smoothed is determined by the modification's `coverage` and `slope` set.

Require: $layer \leftarrow Array[Array[Elevation]]$
Require: $modification \in \mathbb{Z}$
Require: $deviation \in \mathbb{Z}$
Require: $coverage \in [0,1]$
Require: $slope \in [0,1]$
  $startCell \leftarrow layer.getRandomUnmodifiedCell$
  if $\exists startCell$ **then**
    $startCell.value \leftarrow modification$
    for $coverage \times layer.numCells$ **do**
      $nextCell \leftarrow layer.getRandomUnmodifiedNeighbor$
      if $\exists nextCell$ **then**
        $mod \leftarrow [modification - deviation, modification + deviation]$
        $nextCell \leftarrow nextCell.currentValue + mod$
      else
        BREAK
      end if
    end for
    $slopeRadius \leftarrow coverage/slope$
    for $modCell \in modifiedCells$ **do**
      for $cell \in layer$ **do**
        $dist \leftarrow distance(startCell, cell)$
        if $dist \leq slopeRadius$ **then**
          $layer.smooth(cell.x, cell.y, 2, dist)$
        end if
      end for
    end for
    return $layer_{modified}$
  else
    return $layer_{unmodified}$
  end if

3.5.4 Anomaly Placement

Once all `TerrainModification`s have been applied, anomalies are placed into the environment. Each specified anomaly is randomly placed into cell(s) in the `Environment`. For anomalies that occupy more than one cell, neighboring cells are chosen at random until the `Anomaly`'s coverage area is met, or there are no neighboring cells that can contain the `Anomaly`. The anomaly type is appended to each occupied cell's anomalies list for reference within the simulation. After an anomaly has been placed, each of the anomaly's effects are applied. An effect will alter the element values for the occupied and surrounding cells in the effected area. These alterations are typically applied as a "radiation." For example, a `Human Anomaly` might radiate heat and sound.

Now that all `ElementSeed`s, `TerrainModification`s and `Anomaly` placements have occurred, the resulting layers containing their respective elements are populated into their corresponding (x,y) cell location within the `Environment` grid. The resulting instance of an `Environment` class is then returned by the builder to the requesting party.

3.6 Visualization Tool

The environment build tool provides a Graphical User Interface (GUI) for creating and visualizing environments. Electron [25] is used to render a web page contained within a standalone desktop application. This allows the front end to be written in JavaScript, HTML and CSS and handle communication to the back end via HTTP over a localhost network. A Scala library named http4s [26] is used to create a server on a localhost network for handling the HTTP requests from the front end. Launching the GUI starts up the Scala server in a new terminal and opens the Electron window which will begin attempts to establish communication with the server.

3.6.1 Home Page

Once a connection between the server and GUI has been established, the user is brought to the home page (figure 3.9), where they can choose to generate a random environment, build a custom environment, load in an environment, or view an operation. For a random environment, the user inputs the name and size of the environment and all other variables are set by the server. Building a custom environment steps the user through a series of form pages to create an `EnvironmentTemplate`. Loading an environment allows the user to select a saved environment or a saved template. Selecting an operation will load the environment and log of all actions taken by an agent during a specific operation run that is saved in memory. Once an environment has been generated and/or loaded by the server, it is returned to the GUI to be displayed. The environment build tool will parse the returned environment into a graphical data representation, with interactive capabilities to explore the specific variables within the environment. In the case that the user selected an operation, the user will additionally be able to step through the event log of an Operation.

Figure 3.9: Home page for SCOUt's graphical user interface. From here the user can select to generate a random environment, define a new environment template for generating an environment, load in existing environment or templates, or load an operation log to view the environment used in the operation and events that occurred.

3.6.2 Template Forms Page

To create a template, the user will be presented with a series of forms with parameter input fields. The forms are generated based on the available `Element`, `TerrainModification` and `Anomaly` classes that are defined in SCOUt's back end. The first three forms will ask the user which element types, terrain modification and anomaly types they would like to include (e.g., figure 3.13). Some elements such as elevation, latitude, and longitude, are required in all environments. Once the user selects which features will be present in the environment templates, more forms will be generated for defining element seed data, anomaly seed data, and terrain modification parameter (e.g., figure 3.11). Each form within this process will

save the user's input data on the front end. This allows the user to go back and edit form data while moving through the different form pages, as well as return and edit the form data after an environment has already been generated and loaded into the visualization page. Form data is also set within required bounds and checked before submission. Once the user has filled out all required form info, they can review their entire form entry from a single page and then submit it. When submitted, the front-end data is converted into JSON data and sent in a request to the SCOUt server via a JavaScript fetch request. An `Environment` instance is then built on the back end using the template parameters provided and returned to the front end where it is loaded into the visualization page. Figure 3.13 shows an example of an environment template form page, and figure 3.12 shows an overview of the form input process.

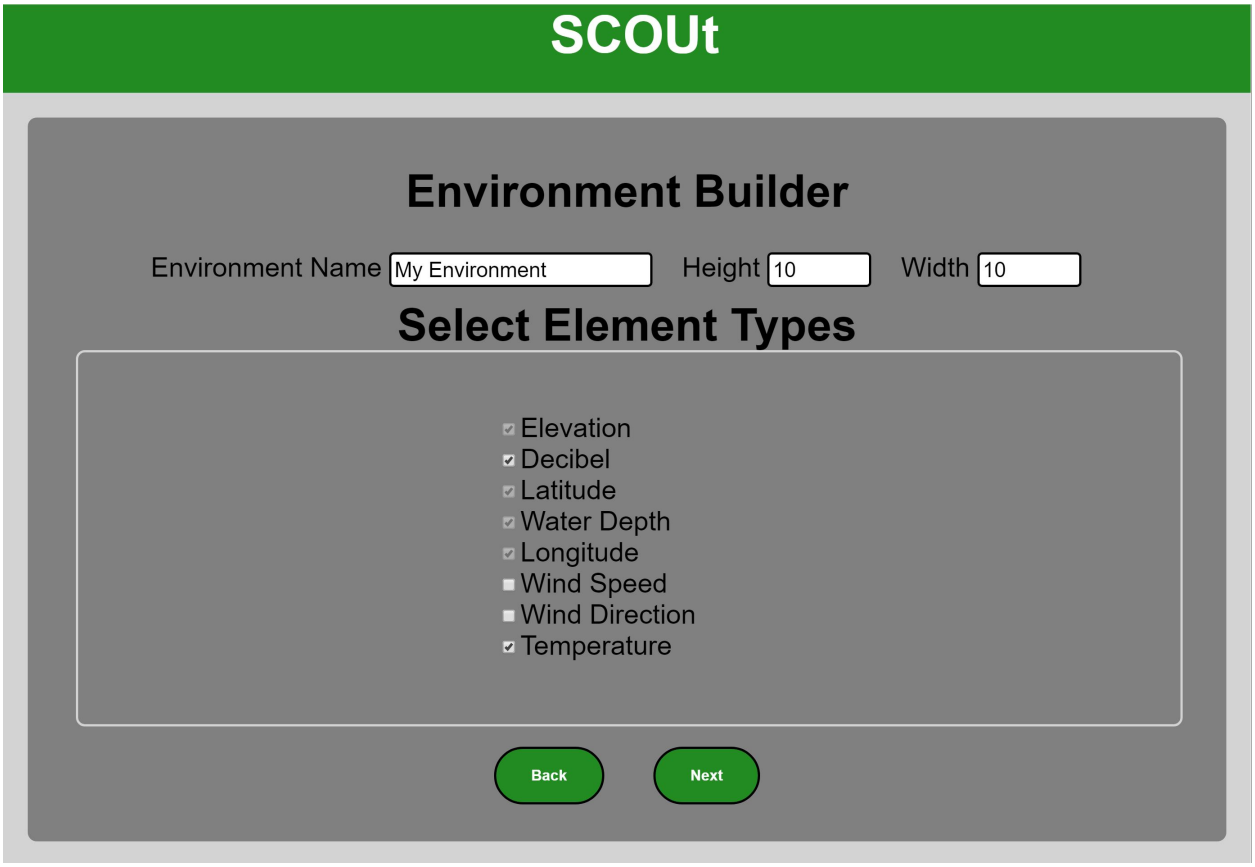

Figure 3.10: User-input form page for selecting element types that will be present in an environment template. For each element type selected a new form page will be generated for the user to define seed data to guide how the element type will be procedurally generated into an environment.

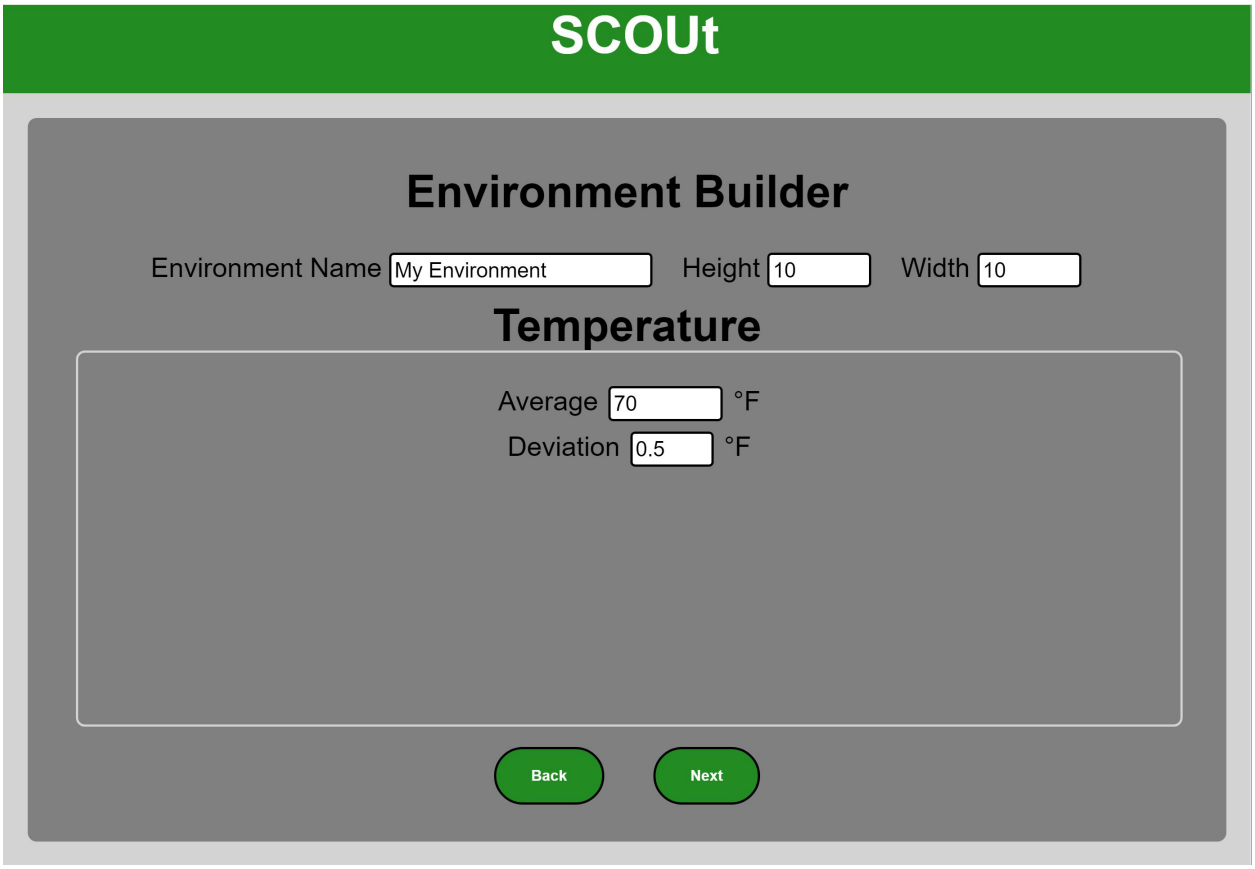

Figure 3.11: User-input form page for specifying element seed data for the "Temperature" element type. This seed data will be used in an environment template to procedurally generate an environment with an average ambient temperature of 70 degrees Fahrenheit and a standard deviation of 0.5 degrees from the average throughout the environment.

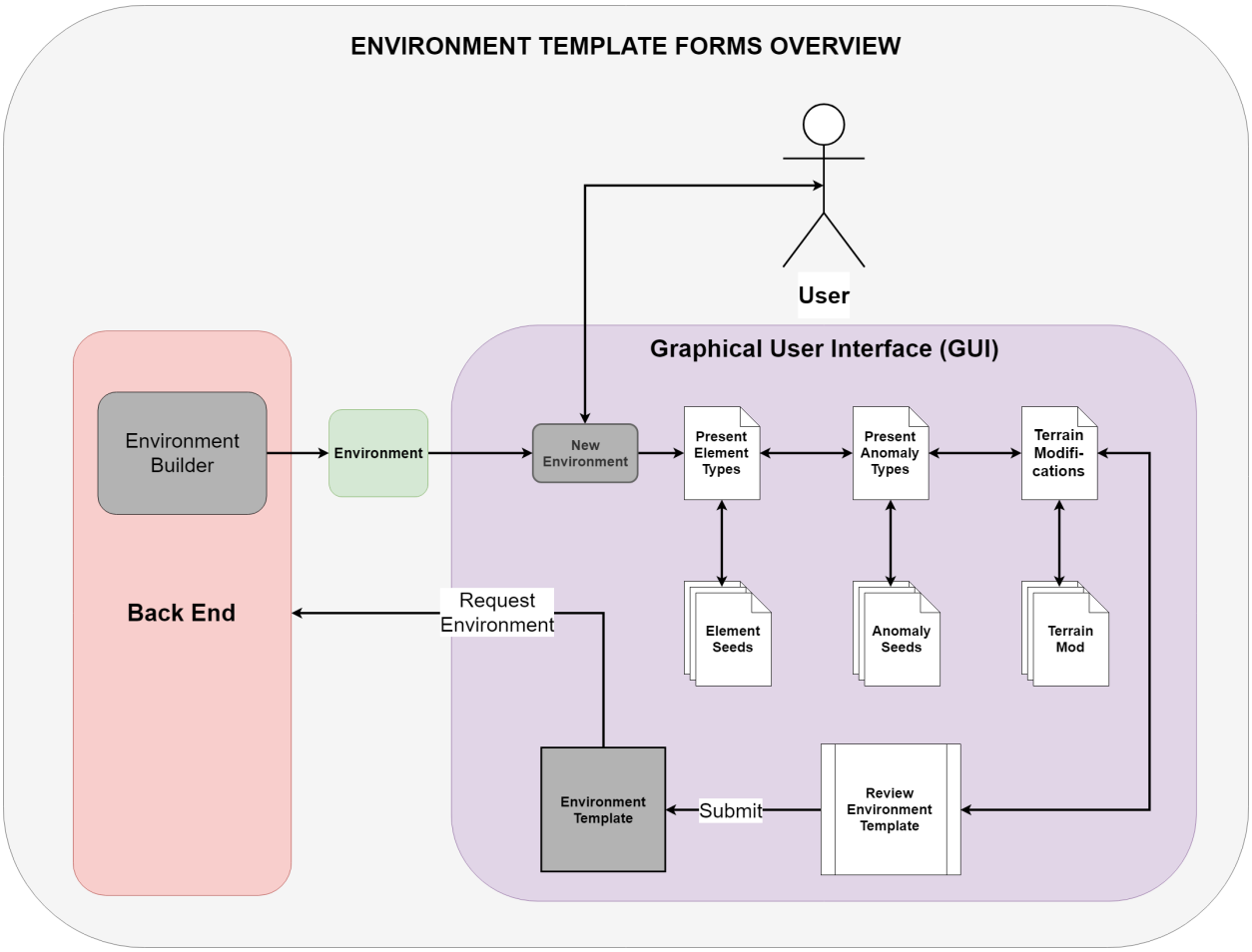

Figure 3.12: Visual overview of the form process that a user will complete to generate an environment. As the presence of element types, anomalies, and terrain modifications are selected, more forms will be generated for each of one. Once the user has filled out all the generated forms and has submitted them, an environment template will be created and sent to the back end. Here, the `EnvironmentBuilder` will create an instance of an environment based on the user's input and return it to the front end.

3.6.3 Visualization Page

The visualization page provides an interactive overview of any given environment. The main focus is on the display section where the entire environment grid is represented using heatmaps. Different element layers can be viewed independently, anomaly locations can be highlighted, and specific element type values of a single cell can be viewed. A main menu is also present to allow a user to perform higher level actions. All of these interactive features are controlled by action, toggle and radio buttons within different sections of the visualization

page. The primary use of the visualization page is for creating environment templates and for debugging. Debugging usage ranges from analyzing an environment that was used for testing an agent-controller setup or new features. Examples of new features would be adding new classes (e.g., a new element type), or altering the process in which environments are generated and stored on the back end.

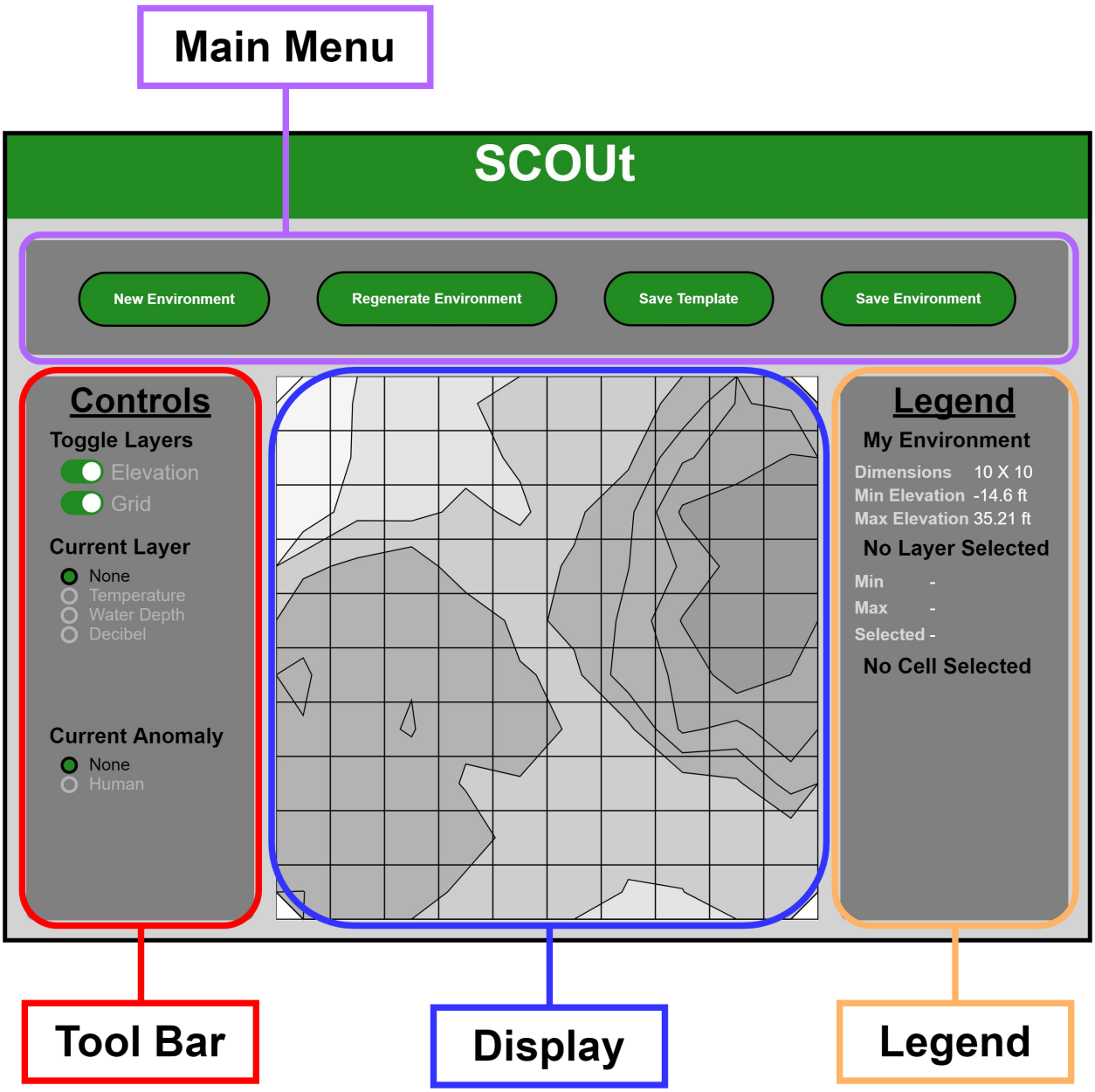

Figure 3.13: Labeled view of the interactive environment-visualization tool with an example environment loaded. Each section of the page either displays information about the environment viewed or allows the user to interact with the view. The main menu also provides features to leave this view (via "New Environment").

Main Menu

The main menu provides high level functions to perform while using the environment build tool. The main menu options are displayed at the top of the visualizer as a series of

buttons. The user can select buttons to return to the home page, regenerate or save the current `EnvironmentTemplate` (if an environment template is being used), or save the current `Environment` instance that is currently being viewed. If the user wants to tweak the current template that is in use, they can do so by returning to the home page via the home button and choosing "Custom Environment" again. Their previously set parameters will be loaded back into the form fields for editing.

Display

The display is laid out in a grid of display cells corresponding to the environment's cell grid. A display layer is created for each element type present within the environment using the D3 library [5]. D3's heatmap creates a graphical representation of data values over a 2-dimensional space, providing a solution for visually differentiating each cell's value within the environment cell grid. Heatmaps display the variation of values using a color scale where higher values are indicated by darker sections and lower values as a lighter section of the map. For example, when viewing the elevation's heatmap, a hill will appear darker than a valley. The user can also select individual cells by clicking on their region in the displayed cell grid. The display will highlight a display cell once it has been selected and then load the cells data into the legend.

Tool Bar

The tool bar is divided into three subsections: Toggle Layers, Current Layer and Current Anomaly. The Toggle Layers subsection provides two toggle buttons for the user to turn on and off the display of the Elevation layer and the Grid layer. The Elevation layer is a grey-scale contour map of the Elevation layer created by D3 heatmap (a contour map is the same as a heatmap, with the distinction of boarder lines between each value layer). Because Elevation is the most fundamental piece to any environment, it is the only element type whose layer has the option of always being displayed. The grid layer displays solid black lines

between each cell within the cell grid. The Current Layer subsection provides a list of radio buttons for all other element types present within the environment. When one of these radio buttons is selected, a green-scale, transparent heatmap of the selected element type will be populated into the display. This element type layer will be displayed on top of the Elevation layer (if Elevation is toggled on). Only one element type layer can be viewed at a time to prevent crowding the display. The Current Anomaly subsection is also a set of radio buttons for each anomaly type present in the environment. Selecting one of these will highlight all display cells containing the given anomaly type in red. Just as the case with element type layers, only one anomaly type can be viewed at a time. Figure 3.14 shows an example of how the display is manipulated by selecting different controls within the toolbar.

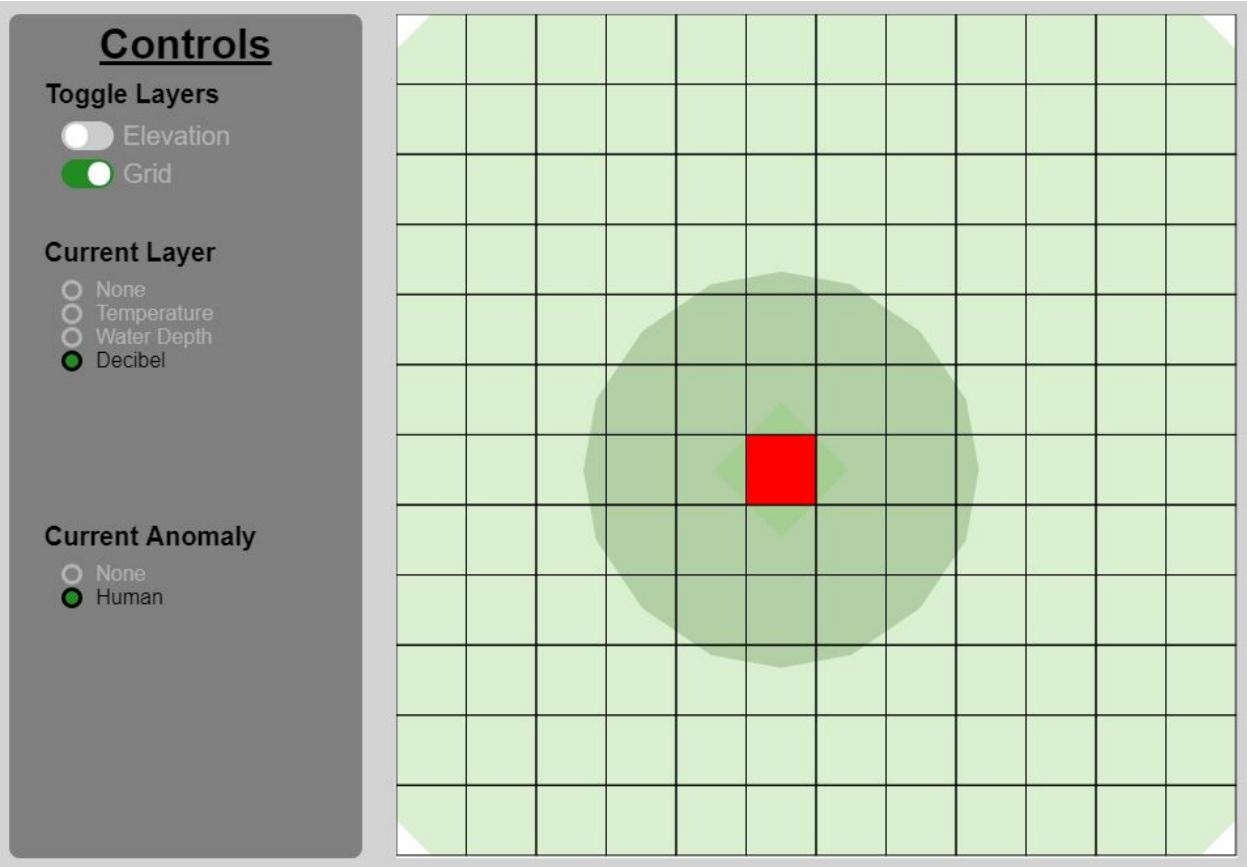

Figure 3.14: Example of the environment-visualizer's toolbar being used to manipulate the display section. The elevation layer has been toggled off, the decibel layer has been selected to be displayed (as a green-scale heatmap), and the human anomaly has been selected and its location is highlighted as the red cell within the display.

Legend

The legend provides an overview of the environment in three main subsection: environment, layer and cell. For the environment subsection, the name of the current environment is displayed along with the dimensions and minimum and maximum elevation within the environment. The layer subsection displays the minimum and maximum values of the selected display layer (if one is selected), as well as the display layer's value at the selected cell (if a cell is selected). When a cell is selected, the values of all element types in the area covered by the cell are presented in a list, as well as the cell's relative coordinates in the grid (e.g., figure 3.15).

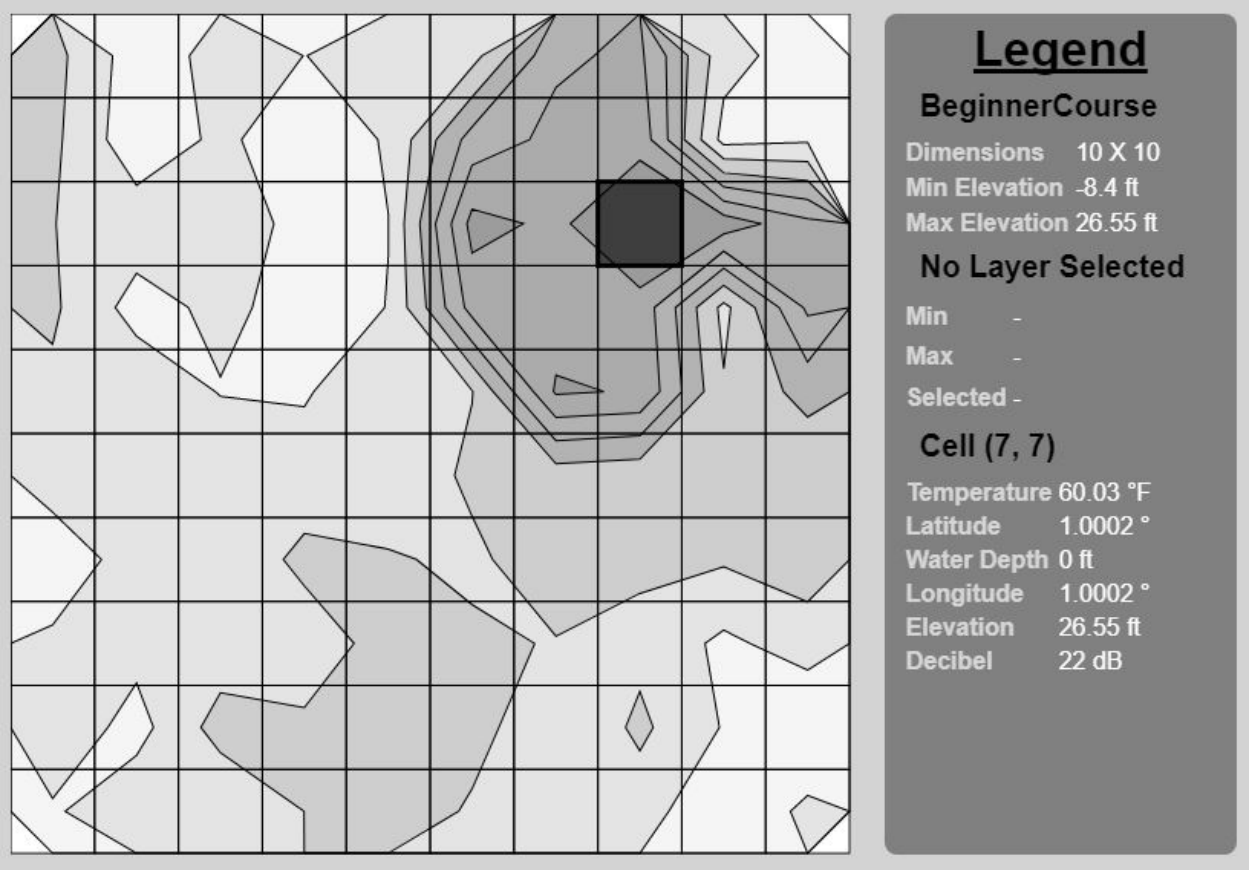

Figure 3.15: Example of the environment-visualizer's legend displaying position and element value information about cell (7, 7) selected from the environment grid (highlighted as a darker shade of grey within the display grid).

Operation Log

The operation log section is a special section that only appears in the visualizer when the user loads an operation run. This section has buttons that allow the user to step through each event that took place during the given agent's operation (see figure 3.16). The user can select to step forward or backwards by 1 or 10 events. When each event is loaded into the visualizer, the display section will update by selecting the cell where the agent is currently located. There is also a text display section that shows the index of the event that is currently being viewed, the action that was chosen, the health and energy of the agent during this event, and the long-term and short-term rewards that were received.

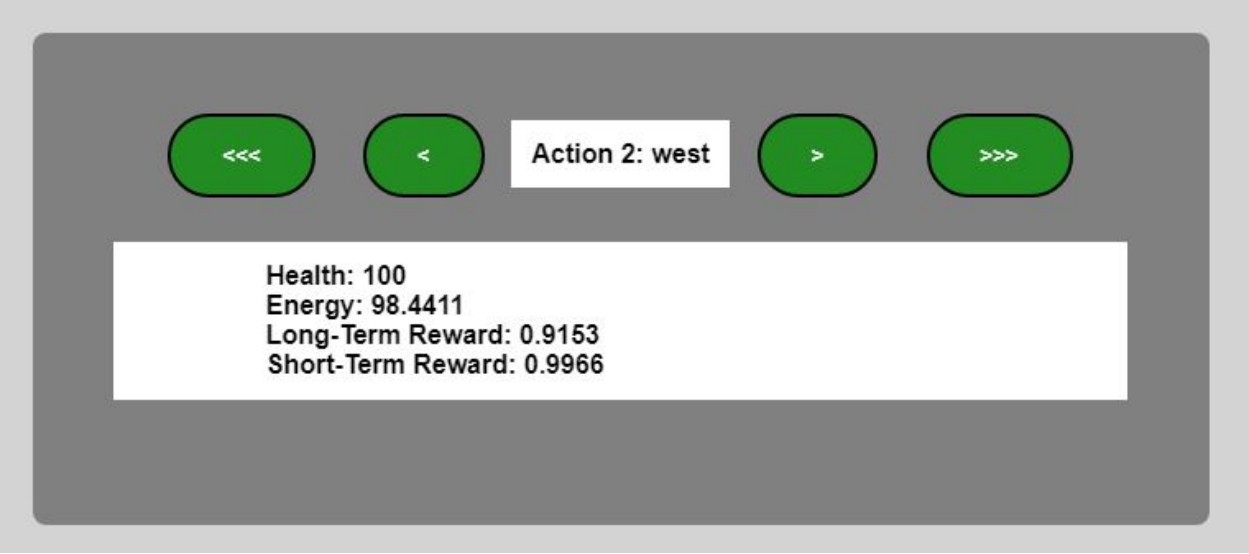

Figure 3.16: A command and dialog display tool that will supplement the environment-visualization page when viewing an operation log of an agent. The user can use the "<<<" and "<" buttons to step backwards 10 and 1 events in the log, and the ">>>" and ">" buttons to step forward 10 and 1 events in the log. The agent's health and energy are displayed during the current event being viewed from the log, along with the action's index within the log, the action selected, and the short-term and long-term rewards that were received for the event.

Chapter 4

Controllers

Three types of control schemas are compared in this project: random, heuristic, and SCOUt's memory-based learning. All three controllers are designed to operate within unknown environments using whatever sensors are available. Controllers are compared based on their ability to complete a defined goal, the number of actions that the controller had to perform before completing the goal, and the remaining health and energy levels of the agent. The random controller will select valid actions at random until the goal is completed successfully, or the agent's health or energy is depleted. This behavior provides a primary baseline for determining what levels of performance are considered *intelligent*. Intelligent controllers would need to exceed the performance of a controller that simply selects actions at random. Both the memory-based learning and heuristic approaches can be considered intelligent, as they use knowledge of their environment to select actions. It is up to each controller type to effectively use the information provided in the agent's current state to guide them towards success. Heuristic controllers perform a set of hard-coded logical analyses to choose actions. This type of approach offers practical solutions to operations but are not expected to be optimal. In addition to this, heuristic controllers are not adaptive to new situations as their logical schemas must be defined for each specific goal. Experiments in chapter 5 simulate operations for two goals: *Find Human* and *Map Water*. Separate heuristic controllers are created for

each of these and provide a secondary performance baseline. For SCOUt's memory-based learning schema to be considered both intelligent *and* adaptive, it would need to perform at the same level or better than the heuristic schemas designed specifically for each goal. The architecture of the heuristic and SCOUt control schemas are covered in section 4.1 and 4.2 respectively.

4.1 Heuristic Controllers

Two heuristic controllers are used in testing: $Heuristic_{FH}$ and $Heuristic_{MW}$.

$Heuristic_{FH}$ is designed for the *Find Human* goal, and $Heuristic_{MW}$ is designed for *Map Water*. Both use the same action decision models (figure 4.1) with slight variations. The models will consider every valid action and give each a score based on the agent's current state. The action with the highest score is then selected. Different score calculations are used for scanning and movement actions, but scores will always be a value between 0 and 1 (1 being the best possible score). The difference between the two heuristic controllers is found in the way they score movement actions. $Heuristic_{FH}$ influences movement to cells that have higher decibel and temperature differentials, as a human anomaly will likely be indicated by increased values of these element types. $Heuristic_{MW}$ encourages movement into quadrants that have fewer known element values so that it can gather new data from unexplored area. Both controllers' movement-action scores also factor in hazard avoidance. Movement into cells with the presence of water or large elevation differentials is discouraged as they could result in damage to the agent. During an operation, the heuristic controller keeps a history of actions performed at each (x, y) location in the environment. After action scores are initially calculated using their respective function, a penalty will be given to any repetitive actions. If the controller has previously selected one of the considered actions while in the same location, the calculated score will be cut in half. This will encourage the controllers to make new choices resulting in exploration of new areas, and a more efficient

use of sensors.

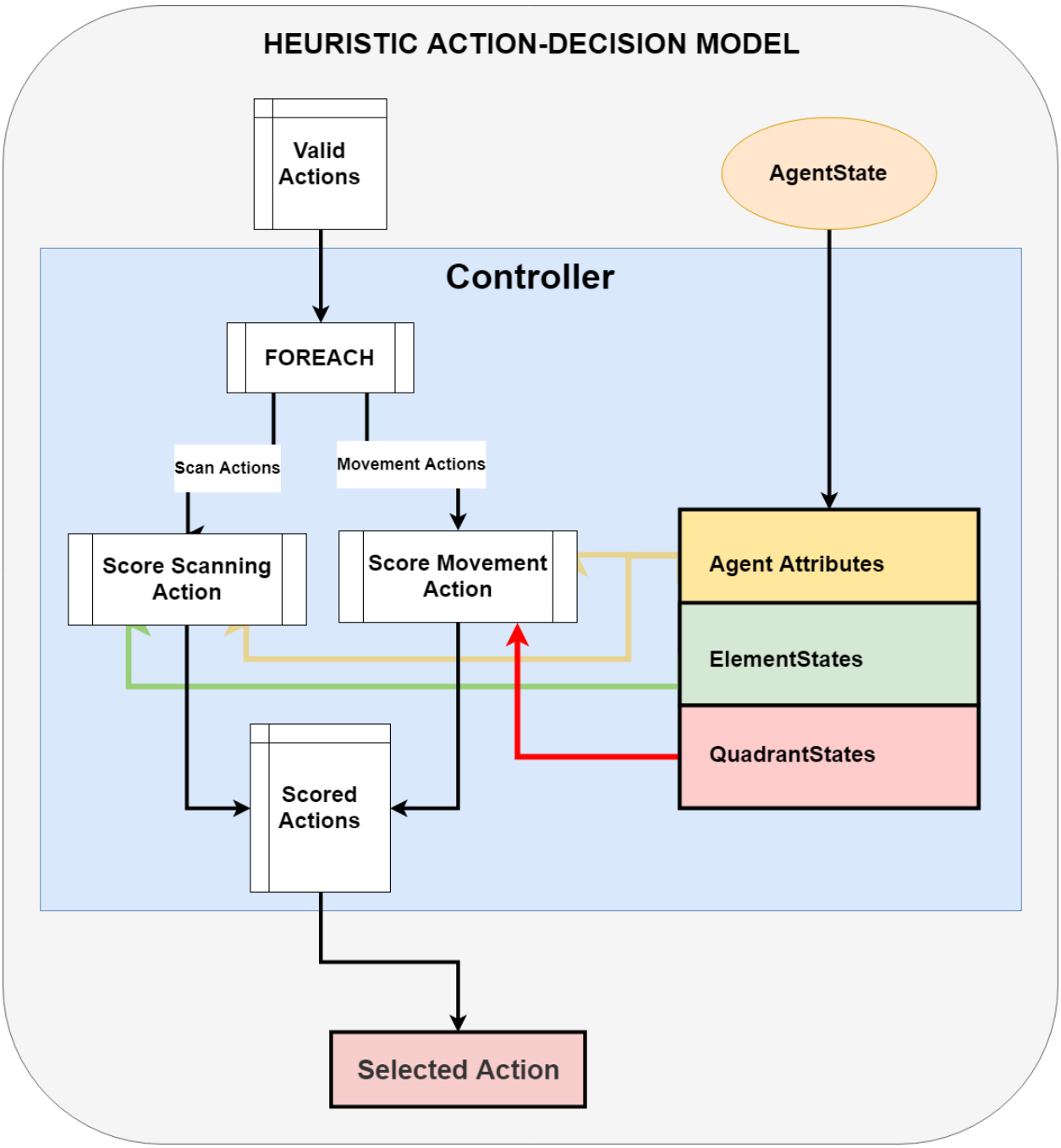

Figure 4.1: The general decision model that our heuristic controllers follow. An `AgentState` and a list of valid actions are passed to the controller. The controller then assigns a score to each action by analyzing related attributes within the `AgentState`. The highest scoring action is then returned.

Valid scanning actions are all scored using function 4. Higher scores will be given to

scanning actions for an element type that is considered more important and has fewer known values within the corresponding sensor's range. Importance of an element type is determined by whether it is flagged as hazardous and/or as an indicator. The amount of known values in the corresponding sensor's range is calculated by referencing the agent's `internalMap`. The resulting score should influence the controller to use sensors efficiently, assist with hazard avoidance, and emphasize goal completion.

Algorithm 4 Calculate a score for a considered scanning action for a specific element type based on an `ElementState`. The returned result will be used to rank the action in the decision-making process. W_{item} denotes the attributed weight for *itemReward*.

Require: $W_{indicator} \in [0, \infty)$
Require: $W_{hazard} \in [0, \infty)$
Require: $W_{pkir} \in [0, \infty)$
Require: $W_{immediates} \in [0, \infty)$
Require: $indicator \in \{true, false\}$
Require: $hazard \in \{true, false\}$
Require: $percentKnownInRange \in [0, 1]$
Require: $immediatesKnown \in [0, 4]$
Ensure: $scanActionScore \in [0, 1]$
  if indicator = true **then**
    $iScore \leftarrow W_{indicator}$
  else
    $iScore \leftarrow 0$
  end if
  if hazard = true **then**
    $hScore \leftarrow W_{hazard}$
  else
    $hScore \leftarrow 0$
  end if
  $pkirScore \leftarrow (1 - percentKnownInRange) \times W_{pkir}$
  $imdsScore \leftarrow ((4 - immediatesKnown)/4) * W_{immediates}$
  $scoresTotal \leftarrow iScore + hScore + pkirScore + imdsScore$
  $W_{total} \leftarrow W_{indicator} + W_{hazard} + W_{pkir} + W_{immediates}$
  return $scanActionScore \leftarrow scoresTotal/W_{total}$

Scoring each valid movement actions is based on the controller's specific implementation of the `scoreMovmentAction` function. These functions involve a series of sub-functions tied to each available sensor's element type. Each of the sub-functions calculate a sub-score for their element type. These sub-functions use threshold analyses on the `QuadrantState`s corresponding to the direction of movement being considered. Once each element type's sub-

score has been returned to the `scoreMovmentAction` function, an overall score is determined by a weighted average. The overall scoring functions used for $Heuristic_{FH}$ (algorithm 5) and $Heuristic_{MW}$ (algorithm 6) follow the same logic, but contain different sub-functions and related weights. For an example of how threshold analyses are conducted within a sub-function, see $Heuristic_{FH}$'s `scoreElevation` algorithm (algorithm 7).

Algorithm 5 Calculate a score for a considered movement action in a specific direction based on a set of corresponding `QuadrantState`s (QS). The returned results will be used to rank the action in the decision-making process. W_{item} denotes the attributed weight for $itemReward$. This function also uses a $score < Element - Type >$ function. Example for one such equation is algorithm 7. This equation is used specifically for the $Heuristic_{FH}$ controller's decision model.

Require: $W_{elevation} \in [0, \infty)$

Require: $W_{decibel} \in [0, \infty)$

Require: $W_{temperature} \in [0, \infty)$

Require: $W_{water} \in [0, \infty)$

Ensure: $scoreElevation \rightarrow [0, 1]$

Ensure: $scoreDecibel \rightarrow [0, 1]$

Ensure: $scoreTemperature \rightarrow [0, 1]$

Ensure: $scoreWater \rightarrow [0, 1]$

Ensure: $movementActionScore \in [0, 1]$

$eScore \leftarrow scoreElevation(QS) * W_{elevation}$

$dScore \leftarrow scoreDecibel(QS) * W_{decibel}$

$tScore \leftarrow scoreTemperature(QS) * W_{temperature}$

$wScore \leftarrow scoreWater(QS) * W_{water}$

$scoresTotal \leftarrow eScore + dScore + tScore + wScore$

$W_{total} \leftarrow W_{elevation} + W_{decibel} + W_{temperature} + W_{water}$

return $movementActionScore \leftarrow scoresTotal / W_{total}$

Algorithm 6 Calculate a score for a considered movement action in a specific direction based on a set of corresponding `QuadrantStates` (QS). The returned results will be used to rank the action in the decision-making process. W_{item} denotes the attributed weight for $itemReward$. This function also uses a $score < Element - Type >$ function. Example for one such equation is algorithm 7. This equation is used specifically for the $Heuristic_{MW}$ controller's decision model.

Require: $W_{elevation} \in [0, \infty)$

Require: $W_{water} \in [0, \infty)$

Ensure: $scoreElevation \rightarrow [0, 1]$

Ensure: $scoreWater \rightarrow [0, 1]$

Ensure: $movementActionScore \in [0, 1]$

$eScore \leftarrow scoreElevation(QS) * W_{elevation}$

$wScore \leftarrow scoreWater(QS) * W_{water}$

$scoresTotal \leftarrow eScore + wScore$

$W_{total} \leftarrow W_{elevation} + W_{water}$

return $movementActionScore \leftarrow scoresTotal/W_{total}$

Algorithm 7 Calculate a score for a `QuadrantState` (Q) of element type "elevation." The returned results will be used to rank the action in the decision-making process. W_{item} denotes the attributed weight for $itemReward$. This equation is used in both the $Heuristic_{FH}$ and $Heuristic_{FH}$ controllers' decision models.

Require: $W_{percentKnown} \in [0, \infty)$

Require: $W_{immediateValue} \in [0, \infty)$

Ensure: $percentKnown \rightarrow [0, 1]$

Ensure: $scanMovementScore \in [0, 1]$

$pkScore \leftarrow scoreElevation(QS) * W_{elevation}$

if $\exists immediateValue$ **then**

  if $|immediateValue| > 12$ **then**

    $imScore \leftarrow 1$

  else

    $imScore \leftarrow 0$

  end if

else

  $imScore \leftarrow 0$

end if

$scoresTotal \leftarrow pkScore + imScore$

$W_{total} \leftarrow W_{elevation} + W_{decibel} + W_{temperature} + W_{water}$

return $scanActionScore \leftarrow scoresTotal / W_{total}$

4.2 SCOUt Controller

The SCOUt controller uses reinforcement learning to build a memory of past actions and rewards for planning future actions. After each operation, the SCOUt controller will store state-action rewards in memory. A state-action reward (SAR) contains the action that the agent took, the state that the agent was in when it chose this action, and the short-term and long-term rewards that the agent received. In future operations, the SCOUt controller (figure 4.2) will search in memory to find SARs with states that are similar to the agent's current state. Utilizing data from the agent's current state and the controller's collection of

past action-state pairs, SCOUt will predict rewards for each possible action and select one based on these predictions.

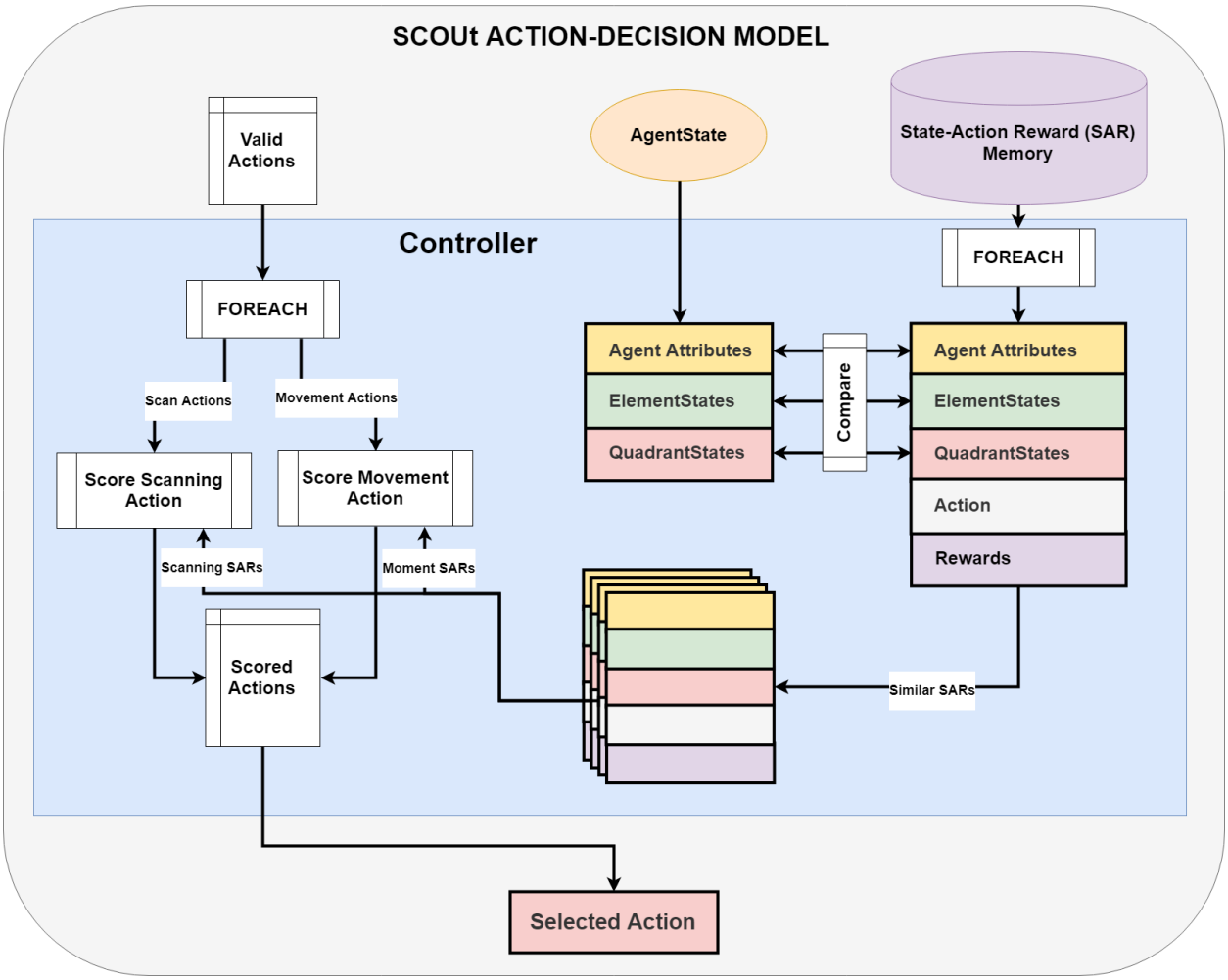

Figure 4.2: Action decision model for the SCOUt controller.

Calculations used for action decision rely on several weights and variables to assist in state comparisons and future reward prediction. Because of the large number of weights required for these calculations, a basic genetic algorithm (GA) was used to optimize these weights. The GA initialized a population of 10 weight sets and evolved them for 50 generations. Each generation creates five mutated copies and five crossover copies of individuals in the current population. The individuals that are copied for mutation or crossover are chosen using roulette selection. Fitness scores are calculated for each of the resulting 20 individuals based on their performance within a series of 50 operations. Ten survivors are then selected for the

next generation. Survivor selection keeps the two individuals with the highest fitness scores and uses roulette selection for choosing the remaining seven. The weight set with the highest fitness in the final generation was selected for use in experimentation, and its values listed in table 4.1.

4.2.1 Memory

The SCOUt controller can gather memory from every operation. When an operation has finished, and long-term rewards have been assigned to each action, the controller creates new SARs, and selects a sub-set of them to be stored in memory. The rest are discarded. Saving only a sub-set cuts back on the size of the memory file as well as the computational time that will be required to search for similar states. The current memory selection method in this project's implementation saves the last 20 SARs, and a uniformly sampled sub-set of the remaining SARs from the operation. The last 20 are always saved because they typically hold the most important events leading up to the success or failure of the operation. To also capture events that occur during the agent's initial and intermediate search process, the controller retains 5 percent of the remaining SARs generated. These SARs are uniformly sampled beginning at index 0. So, if there were 100 SARs in the remaining set, SARs indexed 0, 20, 40, 60, and 80 would be stored in memory. Each SAR is added to the controller's memory file as a JSON object. The next time that the controller is used in operation, the collection of SARs can then be decoded from the file back into `StateActionReward` class instances.

4.2.2 State Normalization

The data within each SAR's state is relative to the operation in which they were recorded. To handle variances found between all of the states stored in memory, SCOUt normalizes them using a Gaussian approach as suggested by McCaffrey [20]. Normalization helps make data values more meaningful when studied by the controller. For example, if the controller

Table 4.1: Set of variables and weights used by the SCOUt controller for action decision. These variables/weights were produced using a basic genetic algorithm.

Attribute	Description	Variable/Weight
	State Comparison Weights	
health	remaining health	0.41
energy	energy level	0.78
elementStates	overall element states	0.61
quadrantStates	overall quadrant states	0.16
	Element State Comparison Weights	
indicator	element type is indicator	0.31
hazard	element type is hazardous	0.07
percentKnownInRange	known element type values in range of sensor	1.0
immediateKnown	number of immediate cell values known	0.41
	Quadrant State Comparison Weights	
indicator	element type is indicator	0.38
hazard	element type is hazardous	0.23
percentKnown	known element type values in quadrant	0.2
averageValue	average element value in quadrant cells	0.19
immediateValue	immediate quadrant cell value	0.29
	Action Selection	
similarityThreshold	SAR comparison qualification	0.26
minimumSimilarStates	used to calculate prediction "confidence"	10
repetitionPenalty	penalty for action that would be repetitive	0.1
	Movement Action Score Weights	
predictedShortTermReward	action's predicted short-term reward	0.87
predictedLongTermReward	action's predicted long-term reward	0.45
confidence	confidence in predicted rewards	0.25
	Scanning Action Score Weights	
predictedShortTermReward	action's predicted short-term reward	0.61
predictedLongTermReward	action's predicted long-term reward	0.34
confidence	confidence in predicted rewards	1.0

$$x_{normal} = \frac{(x - m)}{sd} \tag{4.1}$$

Equation 4.1: Normalization of an attribute value, x, based on the gaussian mean, m, and gaussian standard deviation, sd, for the given attribute.

was seeking out a human, it may look for increases in decibel values. In order for the controller to determine how much of an increase is significant enough to investigate, it needs to first understand what variations are considered normal. Gaussian distribution provides this functionality through the calculation of mean and standard deviation (sd) values in a data set. If the agent has gathered decibel readings in its north quadrant that are well outside the sd found in the controller's memory, it should be encouraged to investigate. All numerical attributes within an `AgentState` are normalized using this Gaussian method. This applies to health and `energyLevel` within `AgentState`s, `percentKnownInSensorRange` within `ElementState`s, and `percentKnown`, `averageValueDifferential`, and `immediateValueDifferential` within `QuadrantState`s.

The normalization process begins by extracting each of these attributes from all SAR states within the loaded memory. Next the mean and standard deviation values are calculated and stored in an instance of a `GuassianData` class (Appendix 7.20). Once mean and standard deviation values are known, the controller will go back through every SAR's state and normalize their attributes using each corresponding `GaussianData` class instance. The normalization function (equation 4.1) will produce a "normal" value that reflects how many standard deviations the attribute falls above or below the mean. A value of 0 represents no difference between the attribute's value and the mean, values of 1 and -1 represent a difference of one standard deviation from the mean, and so on. When SCOUt searches for similar states, it will also normalize the current state using the existing `GuassianData` instances. By normalizing the current state against the states in memory, the numerical attributes compared will all be relative to the mean of the values held in memory.

$$WeightedAverage = \frac{\sum_{i=0}^{n} A_i * W_i}{\sum_{i=0}^{n} W_i} \tag{4.2}$$

Equation 4.2: A general equation that takes a list of n attribute values (V) and a list of n corresponding weights (W) and calculates a weighted average of all attribute values.

4.2.3 State Comparisons

Now that all state attributes are normalized, the controller can use a more intuitive approach for calculating the differences between two states. State comparisons are used to build a set of SARs from memory that contain states similar to an agent's current state. These SARs will later be used to assist in reward prediction. For an SAR to qualify for addition into this set, its state must have an overall difference below the *similarityThreshold* specified in table 4.1. Overall state difference is calculated using a series of difference calculations between related attributes in the two compared `AgentState`s. Results from the series of difference calculations will all be collapsed into a single *overallStateDifference* using a weighted average function (equation 4.2). By comparing each attribute separately and applying a weighted average to the resulting difference calculations, this allows the controller to assign a level of importance to each individual attribute. Importance is assigned via weight values that are between 0 and 1 (see "state comparison" weights in table 4.1). The higher the attribute's weight it, the more influence it will have in the overall state difference. An attribute with a weight of 0 will be completely ignored in a weighted average equation. Different weighted average equations are used for *overallStateDifference* calculation depending on whether the considered SAR's action is a movement or scanning action. This allows the controller to compare only the attributes that are relevant to the type of action that was selected.

Difference comparisons for each attribute in an `AgentState` are calculated based on their data type (boolean, normalized numerical value, optional normalized numerical value, or sub-class). Sub-class comparisons, such as comparing two `ElementState`s, follow the same procedure as `AgentState` does for calculating an overall difference. Difference comparisons will be made for each of the attributes within the sub-class, and a weighted average function

$$BooleanDifference = \begin{cases} x = y & 0 \\ x \neq y & 1 \end{cases} \tag{4.3}$$

Equation 4.3: Difference calculation for two boolean values, x and y.

$$GaussianDifference = |x_{normal} - y_{normal}| \tag{4.4}$$

Equation 4.4: Difference calculation for two normalized vales, x and y.

is applied to the results. Boolean differences will return 0 when the compared attributes are both true or both false and return 1 otherwise (equation 4.3). For example, *BooleanDifference* is used to calculate whether an element type in two `ElementState`s were both flagged as an indicator or not. Normalized numerical attributes follow the *GaussianDifference* equation (equation 4.4). This equation will produce values that hold the same principle as the normalization process, where the closer the difference is to 0, the more similar they are. If two values are identical, their *GaussianDifference* will be 0. Otherwise, the *GaussianDifference* will be relative to how many standard deviations away from each other the two values are. Optional values follow a unique case-based equation (equation 4.5) to calculate the *GuassianDifference* only when both are known. If one value is known and the other is not, a difference of 1 is returned. If both values are unknown a difference of 0 is returned.

Comparisons with SAR's whose chosen action was a scanning action will apply a weighted average to the health, energy, and element states of the two `AgentState`s. Each `ElementState` within the current `AgentState` will calculate their own weighted average based on the number of the immediately adjacent cells that are known, the percent of known element values in

$$optionDifference = \begin{cases} x & \text{known} \cap y & \text{known} & GaussianDifference(x,y) \\ x & \text{known} \oplus y & \text{known} & 1 \\ x & \neg\text{known} \cap y & \neg\text{known} & 0 \end{cases} \tag{4.5}$$

Equation 4.5: A difference calculation used for two values (x and y), when their values are not always known.

range of the sensor, the hazard and indicator flags. All of these *elementStateDifference*s will be averaged (non-weighted) into a single difference value, *averageElementStateDifference*. This compares the usage of the element type (hazard and/or indicator detection), and knowledge of the element type (percent known within the environment). The *hazard* and *indicator* differences can help the controller determine the usage of the element type's data being collected. The *percentKnown* and *immediateValuesKnown* differences help the controller decide whether use of an element type's sensor is efficient or necessary. For example, if an agent does not have knowledge of the elevation in adjacent cells, it couldn't confidently determine whether it is safe or possible to move into one of those cells without first scanning to find out.

If the SAR's action type is movement, overall state difference is calculated using differences in each `AgentState`s' health, energy, element states, and quadrant states. In addition to calculating *elementStateDifference*s, *quadrantToQuadrantDifferences* are calculated between every quadrant in the current state and every quadrant in the SAR state. Only one "orientation" of quadrant-to-quadrant comparisons will be used in the overall difference calculation. Four orientations are considered by rotating the SAR's quadrants in 90 degree intervals (see figure 4.3). The resulting orientation comparisons are denoted as North-to-North, North-to-West, North-to-South and North-to-East (based on the SAR's quadrant that is matched to the current state's North quadrant). The orientation that yields the lowest *quadrantToQuadrantDifferences* (*lowestQuadrantOrientationDifference*) is used in calculating $OverallDifference_m$.

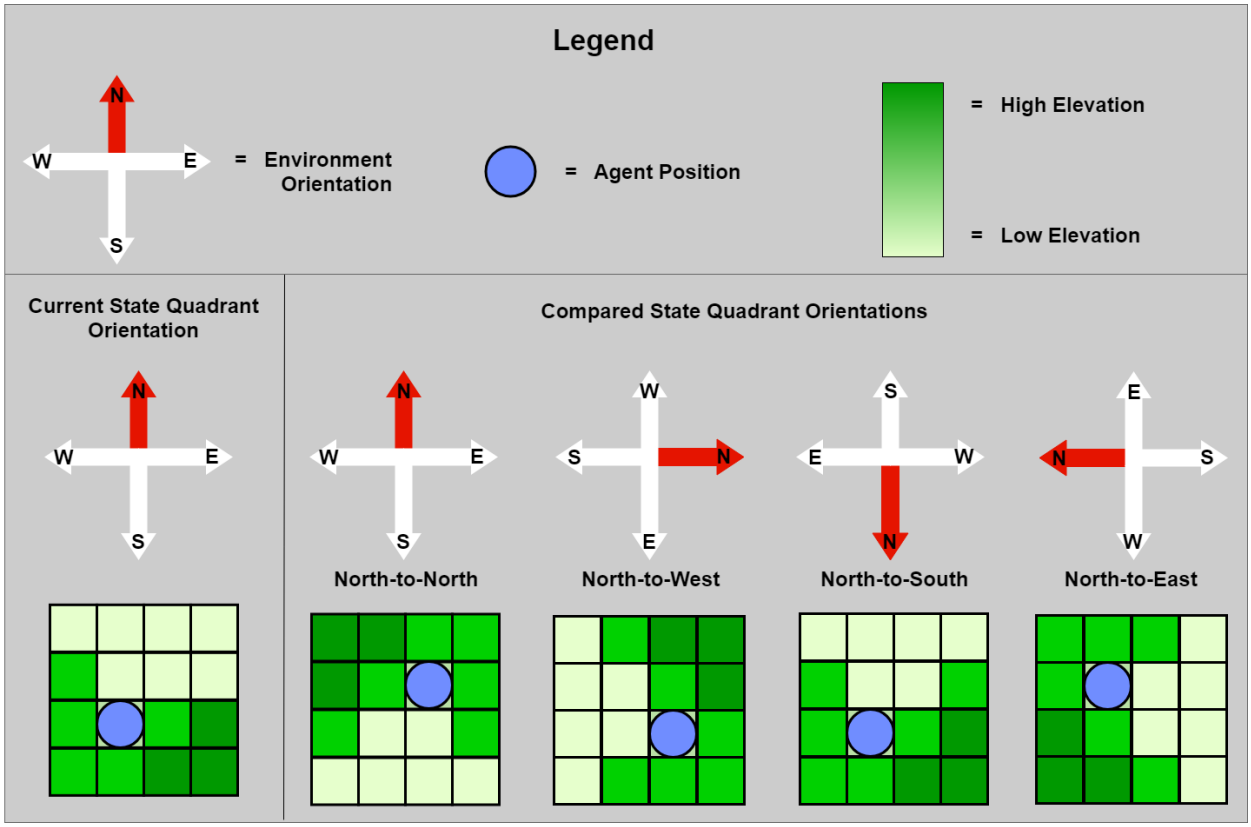

Figure 4.3: Orientation considerations between two compared states. This displays how rotating the compared environment's quadrant orientation can reveal states of higher similarity. We see that the elevation heatmaps of the two are highly similar when compared at North-to-South orientation, where as an un-altered quadrant comparison (North-to-North) would yield a very negative comparison.

Each orientation is important to consider because the controller is only concerned with moving towards interesting features in an environment, regardless of the direction. Considering the orientation with the lowest difference makes the comparison relative to the two environments instead of the cardinal direction. Consider if a highly similar SAR held information that its agent received good rewards for a particular movement action. The current agent should be encouraged to move towards the quadrant in its own environment that holds similar features (not necessarily in the same direction). Now, if the SAR's *lowestQuadrantOrientationDifference* is found when rotating its quadrants 180 degrees (North-to-South orientation) and it had chosen to move East, the current agent should choose to move West since the two states are oriented at a 180 degree difference between each other (see figure

4.4).

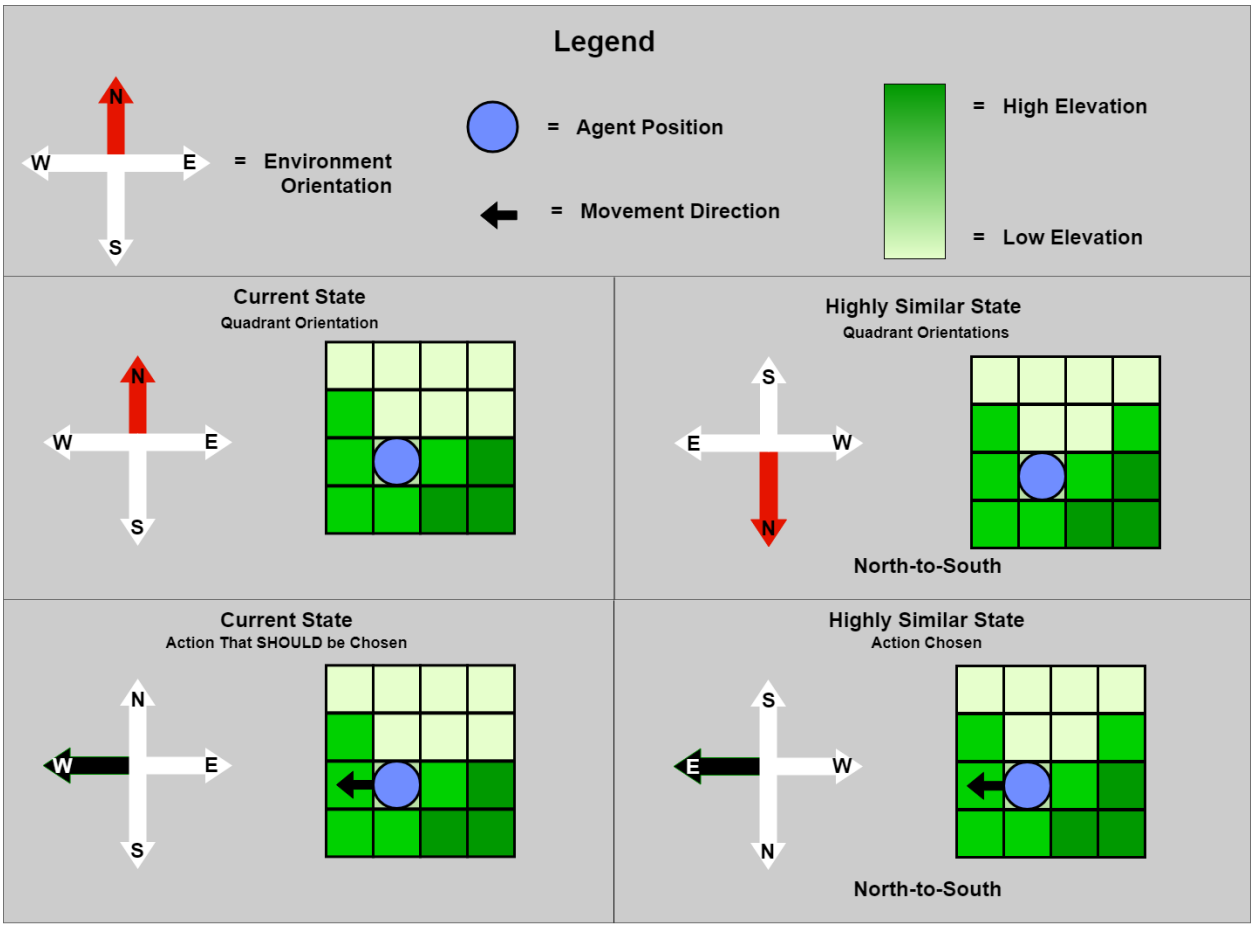

Figure 4.4: Example of two states that have a *lowestQuadrantOrientationDifference* at North-to-South orientation. The display exemplifies how after finding the most similar orientation, the action taken by the compared state must also be re-oriented to match the *lowestQuadrantOrientationDifference*. In doing so the agent will be considering movement relative to the matching features in the environment.

Quadrant-to-quadrant comparisons produce a non-weighted average of *quadrantElementStateDifferences*. A *quadrantElementStateDifference* is calculated between every `ElementState` in the current state's considered quadrant and the matching `ElementState` in the SAR state's considered quadrant (if it exists). The *quadrantElementStateDifference* uses a weighted average to compare attributes within the two `ElementStates`' considered quadrants. For example, making a North-to-South quadrant comparison would consider element types in the current state's North quadrant against element types in the SAR's South quadrant.

When making these comparisons, it is not guaranteed that the current state and SAR state will share all the same element types. For example, if the current state contains decibel data and the SAR state does not, no comparison can be made, and it will receive a *quadrantElementStateDifference* of 1. Because we are only comparing against element types in the current `AgentState`, if the SAR contains any element types not present in the current state, they are simply ignored. These comparisons examine how much information about the element type is known, and the actual values in the quadrants (if they are known). Because `averageValueDifferential` and `immediateValueDifferential` are not guaranteed to be known values for every quadrant, they use the unique option difference equation (equation 4.5).

Once all sub-differences have been calculated and either $OverallDifference_s$ or $OverallDifference_m$ is known, the controller can decide whether the SAR qualifies to be used in future reward prediction for the current agent. If the calculated overall difference is below the *similarityThreshold*, the SAR will qualify and an instance of the `StateActionDifference` class (Appendix 7.21) is created. Each instance stores the overall difference value, the SAR's action taken, and the short-term and long-term rewards. State comparison will be repeated for every SAR in the memory pool, and the resulting collection of `StateActionDifference` instances is passed to the action reward prediction algorithm.

4.2.4 Action Reward Prediction

Once the controller has generated a set of `StateActionDifference`s (SAD), it will predict a short-term and long-term reward value that each possible action might receive, along with a confidence score for the predictions. For each valid action considered, the algorithm will select a sub-set of SAD where the `StateActionDifference`'s `action` is the same as the one being considered. Predicted short-term and long-term rewards are calculated as an average of all the `shortTermScore`s and `longTermScore`s in the sub-set. Confidence is evaluated using the average of the `overallStateDifference`s in the sub-set, weighted

$$similarity = 1 - difference \tag{4.6}$$

Equation 4.6: Equation for inverting an *overallStateDifference* value to create a similarity value. The minimum *overallStateDifference* that can exist is 0. By this logic, the highest attainable similarity between two states is 1.

$$confidence = \begin{cases} n = 0 & 0 \\ n < minimumSimilarStates & \frac{\sum_{n}^{i=0} 1 - SAD_i.overallStateDifference}{minimumSimilarStates} \\ n >= minimumSimilarStates & \frac{\sum_{n}^{i=0} 1 - SAD_i.overallStateDifference}{n} \end{cases} \tag{4.7}$$

Equation 4.7: Confidence value assigned to reward prediction values based on a set of n `StateActionDifference`s (SAD), and the $minimumSimilarStates$ value from the evolved weight set (table 4.1).

by the number of `StateActionDifference`s in the sub-set (equation 4.7). The equation will invert *overallStateDifference*s when averaging them by subtracting their value from 1 (equation 4.6). This allows the prediction algorithm to look at them as "similarity" scores instead of "difference" scores. If the overall difference had been 0 (the states compared were identical), their similarity score will be 1. Because *similarityThreshold* was used to filter out SARs with high overall difference values, it can be asserted that the average of all *overallStateDifference*s will not fall below: $1 - similarityThreshold$. The prediction algorithm then computes an overall *actionScore* for each action using a weighted average of the predicted short-term reward, predicted long-term reward, and the confidence score.

4.2.5 Action Selection

Once every valid action has received an *actionScore*, there are two methods the controller may use for choosing which one the agent should perform. If the controller is being trained, roulette selection is used. Roulette selection is an integral part of training as it will give every action a chance to be selected. This will fill the controller memory with a variety of events both good and bad, giving the reward prediction algorithm more concise data to work with. When the controller is being used outside of training, the action with the highest score

is always selected. Once selected, the agent will then attempt to perform the action, and its interaction with the environment will be reflected in a new `AgentState`. If the agent is still operational after the resulting event and the goal has not yet been completed, the action decision process (figure 4.2) will begin again using the new `AgentState`. Once the agent is no longer operational, or the goal has been completed, the operation process ends, and new SARs are added to the controller's memory file.

Chapter 5

Experiments and Results

To analyze the SCOUt control schema, three instances of a SCOUt controller are trained and then tested in two experiments. The first experiment compares the performance of each SCOUt controller against a random and heuristic controller to determine if they exhibit intelligent behavior when attempting to complete a given goal. The second experiment tests adaptability of the controllers by removing sensors or changing the goal and providing additional training for new goals. This will create new situations for which the controllers have not been trained. Two different goals are used in these experiments: *Find Human* and *Map Water*. *Find Human* requires the agent to search an environment to locate a `Human` anomaly randomly placed within the environment. `Human` is an extended class of the `Anomaly` trait, and its definition can be found in Appendix 7.7. Goal completion is either 100 percent for successfully locating the human, or 0 percent for failing to find the human before health or energy has been depleted. For the goal to be successfully completed, the controller must navigate to one of the eight cells adjacent to the Human anomaly's location in the environment (figure 5.1). *Map Water* tests a controller's ability to navigate within an environment and collect as much water depth data as possible. *Map Water* operations will run until the entire area has been scanned for water depth, or the agent has depleted its health or energy. Goal completion is then reward based on the percentage of the environment

that was scanned for water depth. Training and testing are both conducted using three different `EnvironmentTemplates`. Templates differs in their difficulty to navigate due to the modifications present within them. For example, more difficult templates will generate larger environments to explore that contain more hazardous terrain modifications.

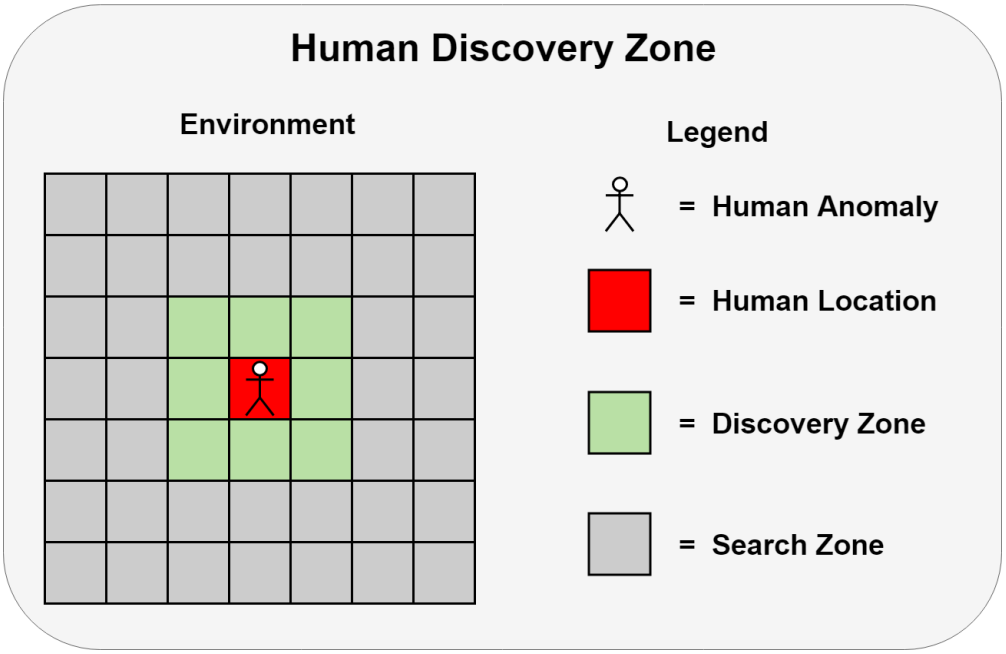

Figure 5.1: Diagram of the area in an environment that an agent must navigate to in order to "discover" a human anomaly. This applies to operations where the goal is *Find Human*.

During both training and experimentation, tests are conducted to measure the performance of each controller. A test is a series of operations that an agent will attempt to complete using a given set of sensors. Each operation in the series will be run once per controller, where each run is identical: same goal, same environment instance, same starting position, and same agent setup (aside from the controller that is used). The only exception to this is in Experiment 2 (section 5.5), when sensors are removed from the SCOUt controller's agent to test adaptability. Tests will measure the performance of several controllers in these identical operations to see how they compare. Each test will include at least one random, one heuristic and one SCOUt controller. Two different heuristic controllers are used in testing depending on the goal of the operation. $Heuristic_{FH}$ and $Heuristic_{MW}$ are

used for *Find Human* and *Map Water* operations, respectively. The random and heuristic controllers provide baselines for performance of the SCOUt controllers. These baselines are important as the difficulty of an operation is difficult to predict. Different instances of the same operation setup can yield unique environments and starting positions that will alter the difficulty of goal completion, or even prevent it entirely. Performance is measured in four categories: goal completion, the number of actions taken, remaining health and remaining energy. Additionally, the agent's starting location for each operation will be chosen in a location that does not result in damage (e.g., starting in a cell with water present) or immediate goal completion (e.g., starting next to the Human anomaly). The specific agent setups and `EnvironmentTemplates` used for training and experimentation are covered in sections 5.1 and 5.2.

5.1 Agent Setup

A similar agent setup is used for every operation in training and testing. The only variation is in the controller being used and the sensor types present. All health, energy, mobility and durability variables will be set to the same value throughout training and experimentation. This will ensure that no advantage or disadvantage is given to any controller when navigating the agent through an environment. The performance of each controller will then solely be reflected by their usage of available sensors, and analyses of the data they collect. Different sensors are available for use depending on the goal, or in the unique case where the set of equipped sensors are changed tests in Experiment 1 (section 5.5.2), depending on the test setup. Four sensors are used throughout testing: elevation, temperature, decibel and water. When sensors are available for an agent to use, the indicator flag will reflect their usage for the present goal. For *Find Human* the temperature and decibel sensors will be flagged as indicators, and for *Map Water* the water sensor will be flagged as an indicator. Water and elevation sensors are always flagged as hazard since the defined agent `Durability`

and `Mobility` instances make the agent susceptible to water and fall damage. Each Agent will always begin an operation with an empty `internalMap`, 100.0 `health` and a starting `energyLevel` of 100.0. The instances of the `Agent`, `Mobility`, `Durabilities`, and each `Sensor` used in training and testing are found in Appendix 7.22.

5.2 Environment Templates

Three `EnvironmentTemplate`s with increasing difficulty are created for use in training and experimentation (*EASY*, *MEDIUM*, and *HARD*). Change in difficulty is achieved by adjusting: environment size, presence of `TerrainModification`s, average and deviation values of each `ElementSeed`, and the values within each `Effect` of any `Anomaly` present. Increased environment size creates a wider area that an agent will have to explore. More `TerrainModification`s makes each environment potentially more hazardous. Hills and valleys can create areas with the potential of fall damage or the inability to climb slopes, and pools and streams of water will create areas that will damage an agent that enters them. Changing the average and deviation values of `ElementSeed`s used to generate the `Environment` instance has a couple of effects. As an example, by increasing the average decibel values in the environment, the distinction of the human's decibel `Effect` is dampened, and the agent will need to navigate closer to the source in order to detect any noise produced. Also, if the variance in decibel values increases, it becomes more difficult for an agent to distinguish what levels of increase are considered significant enough to investigate. Last, by adjusting the values within `Anomaly Effect`s, it can become harder for an intelligent controller to detect the `Anomaly` from a distance. For example, by reducing the decibel effect of a `Human Anomaly`, the radiation of the effect will cover less area in the environment, meaning the agent will have to search longer before it may pick up on the effect.

Each environment template has one `Human` that will be placed into it at random in a non-hazardous zone. The same templates will then be used for both the *Find Human*

and *Map Water* goals. In the case that the goal is *Map Water*, a `Human Anomaly` will still be present within the environment, but it will be ignored by the agent. Each of these `EnvironmentTemplate`s are listed in JSON format in Appendix 7.1.5. When a template is used in training or experimentation, it will be loaded from its file, converted to a Scala object, and passed to the `EnvironmentBuilder`. The resulting `Environment` instance can then be used in one or more test operations.

5.3 Training

Three separate SCOUt controllers are trained to accumulate memory pools of state-action rewards. Each of the three controllers are trained with different goal configurations but follow the same training process (figure 5.2). One is trained using the *Find Human* goal, the second using the *Map Water* goal and the third is a hybrid, trained using both goals. They are named $SCOUt_{FH}$, $SCOUt_{MW}$ and $SCOUt_{H}$ respectively. Each controller is trained for 30 iterations, where an iteration runs one operation per environment template. Once training has completed, each controller will have collected state-action rewards (SAR) from a total of 90 operations (30 on EASY, 30 on MEDIUM and 30 on HARD). Every operation that $SCOUt_{FH}$ is run in will be with the *Find Human* goal and every operation for $SCOUt_{MW}$ will be with the *Map Water* goal. $SCOUt_{H}$ will alternate goals for each iteration. In total it will have run 45 operations with the *Find Human* goal and 45 operations with the *Map Water* goal (15 on EASY, 15 on MEDIUM and 15 on HARD for each).

After each training iteration, the controller is tested with its current memory to track performance improvements. Testing at each iteration runs a series of simulated operations to collect performance data. Each series uses the controller's respective goal(s) and is run on each testing environment template 20 times (20 on EASY, 20 on MEDIUM, and 20 on HARD). The controller tested will have access to its current memory pool that it has gathered during all of the training operations it has completed so far. As the purpose of iteration testing is

to measure the *current* performance level of the controller, no SARs will be gathered during these tests and the memory will be left un-altered. For a baseline, the *Random* controller is run through the same series of tests. Results from the 60 total operations will be averaged in each of the four performance categories. The averaged results of the learning SCOUt controller will then be differenced against the averaged results of the *Random* controller. By differencing the averages, we are observing how much better or worse the SCOUt controller was able to perform than the *Random* controller in the same testing conditions. This also removes the discrepancy between each iteration test that is run, as one iteration text may have generated a series of exceptionally difficult or easy operations. It is expected that as training continues, the goal completion, remaining health and remaining energy performances of SCOUt will increase, and the number of actions performed will decrease when compared against the *Random* controller.

5.3.1 Initial Training Results

Results for $SCOUt_{FH}$ training (figure 5.3) show the desired trends of increased performance over training iterations. Average goal completion and average remaining energy begin at the same performance levels as *Random* and increase to be consistently better than random over time. The average number of actions performed begins slightly below *Random* and continue to decrease before leveling out roughly two-thirds of the way through training. This demonstrates that the controller is learning to perform more efficiently over time, as both average goal completion and average remaining health show major performance boosts, while fewer actions are being used. The average remaining health of $SCOUt_{FH}$ shows slight increase over training, but for the most part is equivalent to that of the *Random* controller.

Figure 5.3: Iteration testing performance results for $SCOUt_{FH}$ attempting *Find Human*. All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_{FH}$ average - *Random* average) VS the number of training iterations completed.

Results for $SCOUt_{MW}$ (figure 5.4) are less positive. Average goal completion and average remaining health both decrease during the first half of training but begin to show upward trends toward the end. While the average goal completion of $SCOUt_{MW}$ is consistently better than *Random*'s, the average remaining health actually performs worse throughout iteration testing. $SCOUt_{MW}$ does perform well in the average number of actions taken per operation, however this is likely due to the fact that health is depleted (agent navigating into water) and the operation is ended early. The same can be said for the remaining energy, as there will be a larger amount of energy remaining after an operation is ended due to depletion of health. The reason for these poor performance results seem to be tied with the agent's inability to avoid hazardous areas containing water. Training was repeated using different setups to see if results could be improved. These repeated training runs are discussed in subsection 5.3.2. Despite unimpressive training results, $SCOUt_{MW}$ still performs well in the

tests to follow.

Figure 5.4: Iteration testing performance results for $SCOUt_{MW}$ attempting *Map Water*. All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_{MW}$ average - *Random* average) VS the number of training iterations completed.

For $SCOUt_H$, performance was tracked for both of the *Find Human* and *Map Water* goals. The iteration test results over training for each of these are found in figure 5.5 and figure 5.6. For the majority of the results in both goal types, the average performance of $SCOUt_H$ does not show any major increasing trends. In *Find Human* testing, average goal completion rises and falls throughout training. The cause of this is unclear but is likely a side effect of simultaneous training on two goals at once. We also see a slight upward trend in the average number of actions that are being performed, but it does stay under *Random*'s performance throughout. Towards the end of training, performances in *Map Water* begin to shift in three categories. Remaining health and number of actions performed shift up, while remaining energy drops, and average goal completion remains fairly consistent. While more actions are being taken (resulting in the decrease of remaining energy), it appears that

$SCOUt_H$ is learning better hazard avoidance behaviors as remaining health has increased over time. $SCOUt_H$ training was also repeated to see if results could be improved (covered in subsection 5.3.2).

Figure 5.5: Iteration testing performance results for $SCOUt_H$ attempting *Find Human*. All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_H$ average - *Random* average) VS the number of training iterations completed.

Figure 5.6: Iteration testing performance results for $SCOUt_H$ attempting *Map Water*. All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_H$ average - *Random* average) VS the number of training iterations completed.

5.3.2 Training Variations

Two variations of SCOUt controller training were conducted to see if better iteration testing results could be achieved for $SCOUt_{MW}$ and $SCOUt_H$. While $SCOUt_{FH}$ performed well in training, this controller was also retrained using the same variations to assure that there would not be any negative effects to its performance. The repeated runs altered the way that long-term rewards were calculated at the end of operations, but the same training and iteration testing setup was used: 30 iterations, 1 test on each environment per iteration, 60 operations per iteration test, and iteration testing performance was compared relative to the *Random* controller. Variation 1 changed the way that operations factored goal completion into the long-term reward (algorithm 3). Instead of always rewarding the controller based on their percent of completion, it would only reward them if the agent had remaining health.

This essentially factors the goal completion as 0 percent if the agent had completely depleted its health. Effects of this alteration are not expected to cause changes in *Find Human* operations (goal completion is already always 0 unless the human was found), but it should lower the rewards seen in *Map Water* operations since there is typically a portion of the goal completed regardless of how the operation ended. This variation was made in hope that controllers would learn stronger hazard avoidance behaviors through the harsher long-term reward system. Variation 2 used the original goal completion equation, but altered the weight used for it. Instead of `goalRewardWeight` being set to 1, it was bumped up to 1.5. This variation was conducted to see if more emphasis was needed on the controllers' level of goal completion within the long-term reward. If a controller obtained a higher level of goal completion, it must have been able to survive in its environment long enough to do so, which would hopefully counteract the hazard avoidance issue. Results for all three controllers' in the new training variations are displayed as follows: $SCOUt_{FH}$ - figure 5.7, $SCOUt_{MW}$ - figure 5.8, $SCOUt_{H}$ - figures 5.9 and 5.10. These graphs show an overlay of the performance results for variation 1, variation 2 and the original training setups that were conducted to easily compare differences between them. All of the results for variations 1 and 2 can be viewed independently in Appendix 7.2.

Figure 5.7: Iteration testing performance results for $SCOUt_{FH}$ attempting *Find Human*. All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_{FH}$ average - *Random* average) VS the number of training iterations completed. This graph overlays the results from three different training setups: variation 1, variation 2, and original.

Figure 5.8: Iteration testing performance results for $SCOUt_{MW}$ attempting *Map Water*. All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_{MW}$ average - *Random* average) VS the number of training iterations completed. This graph overlays the results from three different training setups: variation 1, variation 2, and original.

Figure 5.9: Iteration testing performance results for $SCOUt_H$ attempting *FindHuman*. All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_H$ average - *Random* average) VS the number of training iterations completed. This graph overlays the results from three different training setups: variation 1, variation 2, and original.

Figure 5.10: Iteration testing performance results for $SCOUt_H$ attempting *Map Water*. All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_H$ average - *Random* average) VS the number of training iterations completed. This graph overlays the results from three different training setups: variation 1, variation 2, and original.

Performance results show that all of the iteration tests held the same trend lines between each variation of training setup. Looking at the actual values within each graph, we do see that both variation 1 and 2 hold slightly better results, especially in the category of average actions being performed. Difference between these two is negligible, so variation 2 was chosen as the winner since it achieved performance boosts through simply adjusting the weighting of goal calculation rather than altering the equation's logic. For this reason, the following experiments sets the `goalRewardWeight` to 1.5 in the long-term reward equation and used the resulting trained memory sets from variation 2 for the SCOUt controllers.

5.4 Experiment 1

Once training was completed, the resulting SCOUt controllers were individually tested against *Random* and their respective heuristic controller. Tests in Experiment 1 ran a series of 1000 operations per environment template. The environment template is used to generate 200 unique environments, each of which is used for 5 operations. Results are averaged for each controller's performance within each of the environment difficulties they were tested in. We expect to find that the SCOUt controllers perform better than *Random* and as good or better than the heuristic controllers. This would be reflected in higher average goal completion, average remaining health and average remaining energy, and lower average actions performed.

Results for $SCOUt_{FH}$ (figure 5.11) show clear superiority across almost every test. The only area where scout under-performed was in average remaining health. In the medium difficulty environments, $SCOUt_{FH}$ came in second in this category (right behind $Heuristic_{FH}$) but all three controllers performed within the same ~5 percent range. In the hard difficulty environments, $SCOUt_{FH}$ came in last out of the three in remaining health, but only by a margin of ~2 percent. Average goal completion in every environment difficulty was about double the performance of $Heuristic_{FH}$ and triple the performance of *Random*. $SCOUt_{FH}$ also outperformed $Heuristic_{FH}$ and *Random* in both average actions taken and average remaining health. We see in these results that the heuristic controller was able to perform better than *Random* in most tests. The margin of performance difference between $Heuristic_{FH}$ and *Random* tends to shrink as the environment difficulty is increased. While goal completion consistently remained above that of *Random*, we can see that $Heuristic_{FH}$ is having to use more actions to achieve the goal.

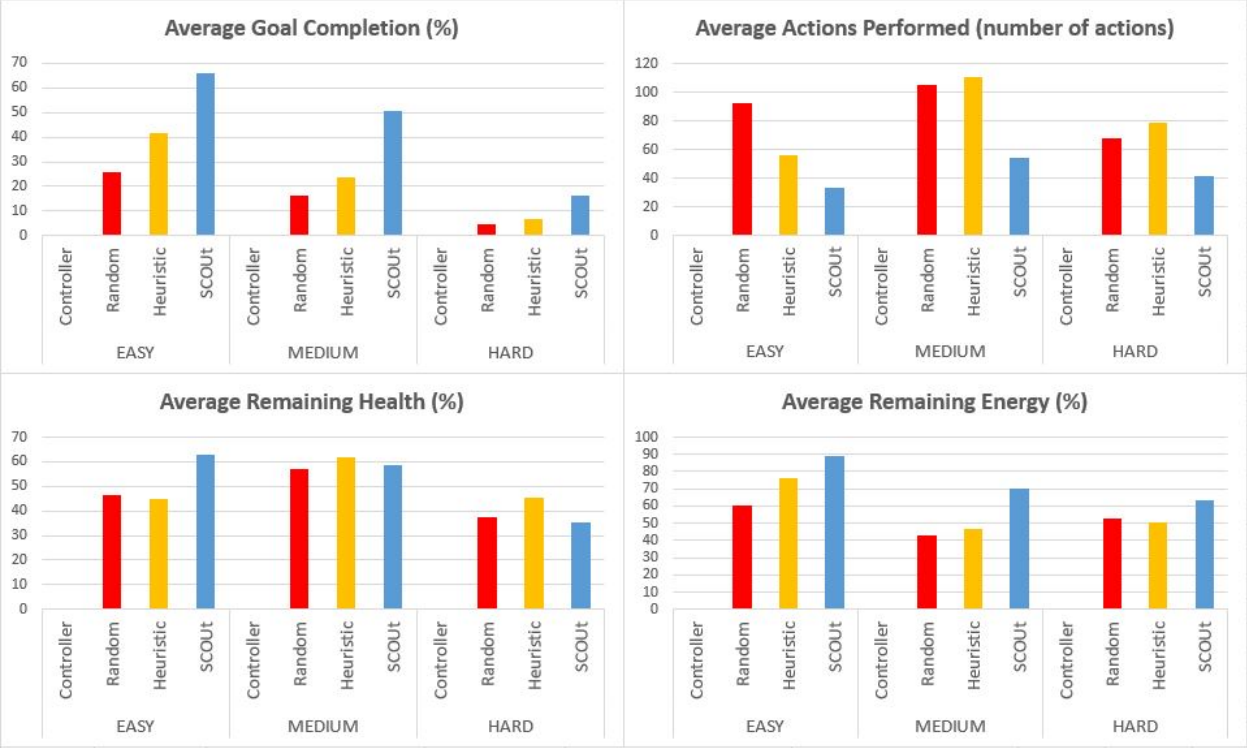

Figure 5.11: Performance results for *Random*, $Heuristic_{FH}$ and $SCOUt_{FH}$ controllers attempting *Find Human* in various environment difficulties.

$SCOUt_{MW}$'s results (figure 5.12) show performance levels that are mostly superior to $Heuristic_{MW}$ and *Random*, but it does not exhibit the same level of superiority as seen in $SCOUt_{FH}$'s results. Again, the area where the SCOUt controller is lacking in performance is the average remaining health. In the easy difficulty environment, $SCOUt_{MW}$ suffers tremendously with an average remaining health of 9 percent. Interestingly, the average remaining health increases as environment difficulty increases. Reasons behind this behavior are unclear as other performance trends increase and decrease as expected when the environment difficulty changes. $Heuristic_{MW}$ ranks as expected. The performance values are consistently better than *Random* in every category outside of average remaining energy. In this category, $Heuristic_{MW}$ only under performs *Random* with a margin of 2 percent. This shows that the heuristic model is useful, but not necessarily efficient.

Figure 5.12: Performance results for *Random*, $Heuristic_{MW}$ and $SCOUt_{MW}$ attempting *Map Water* in various environment difficulties.

$SCOUt_H$ was tested in both *Find Human* and *Map Water*. The results for each goal type are found in figures 5.13 and 5.14, respectively. Results for *Find Human* operations show trends similar to the results for the $SCOUt_{FH}$ controller, except with smaller margins of increased performance against $Heuristic_{FH}$ and *Random*. In environments of hard difficulty, $SCOUt_H$ shows performance levels that match with the heuristic controller. This is likely caused by the fact that it was only trained on *Find Human* for 15 iterations, along with a dilution effect caused by having a mixed memory set (from training on both goals).

Figure 5.13: Performance results for *Random*, $Heuristic_{FH}$ and $SCOUt_{H}$ attempting *Find Human* in various environment difficulties.

Results in *Map Water* for the hybrid controller are nearly identical to the results seen in figure 5.12. The only notable difference is that average remaining health in easy environments was ~21 percent instead of ~9 percent. Outside of this, performance scores and trends of all three controllers are roughly the same. This suggests that operations based on mapping an element type are more difficult than would be expected.

Figure 5.14: Performance results for *Random*, $Heuristic_{MW}$ and $SCOUt_H$ attempting *Map Water* in various environment difficulties.

5.5 Experiment 2

The second experiment is broken into three tests: goal changing, sensor set changing, and additional training. Goal changing, and sensor set changing are both designed to investigate the adaptability of each SCOUt controller. Heuristic and random controllers are still used as baselines, but we also use SCOUt controllers as a baseline. By comparing SCOUt controllers that were trained for the specific goal (in the case of goal changing) or un-altered (in the case of sensor set changing), we can study how performance changes when a SCOUt controller is applied in unexpected scenarios. Following this, $SCOUt_{FH}$ and $SCOUt_{MW}$ are put through additional training on the goal they were not originally trained, to build two new controllers. $SCOUt_{FH+}$ is the result of training on *Map Water*, beginning with a copy of $SCOUt_{FH}$'s existing memory. $SCOUt_{MW+}$ is the result of training on *Find Human*, beginning with a

copy of $SCOUt_{MW}$'s existing memory. The new controllers will be tested to see how well SCOUt's control model is able to learn new tasks beginning with separate memory from another task.

5.5.1 Goal Changing

Goal changing tests the performance of $SCOUt_{FH}$ on the *Map Water* goal, and the performance of $SCOUt_{MW}$ on the *Find Human* goal. The SCOUt controllers have no training on the new goal they are tested within, so behaviors are based purely on the controller's existing memory. Every controller used in experimentation thus far is compared in these tests. Tests are run in the same fashion (1000 operations per environment: 200 environments generated, each used 5 times), but in the results we display, performance is averaged for each controller across all three environment difficulties. This is done because we have already seen that each controller's performance scores show a relatively linear trend as difficulty increases (outside of the case with $SCOUt_{MW}$'s average remaining health).

Results for $SCOUt_{MW}$ in *Find Human* operations are shown in figure 5.15. Unsurprisingly, $SCOUt_{FH}$ takes the lead in every performance category, followed by $SCOUt_{H}$. What we are primarily interested in here is $SCOUt_{MW}$'s ability to outperform $Heuristic_{FH}$. We do see higher averages in the number of actions performed and remaining energy, but this is likely due to the same hazard avoidance caveat discussed in Experiment 1 (section 5.4). This is also reflected in the fact that $SCOUt_{MW}$ has the lowest average remaining health out of all controllers, followed by $SCOUt_{H}$ (again, likely suffering from the same issue). However, even though the average goal completion for $SCOUt_{MW}$ is less than $Heuristic_{FH}$, the margin of difference is only ~1.5 percent. This does lead to the idea that adaptability is present within SCOUt's control schema. Using no prior training for how to find a human within an environment, the controller was still able to perform at the same level as a heuristic model. Additionally, $SCOUt_{MW}$ also outperforms $Heuristic_{MW}$ in all categories excluding average remaining health. This supports the primary focus of this paper, which is: demonstrating

how autonomous control schemas built for one specific task are not inherently adaptable to new tasks, but there still exists underlying features of autonomous control that can be abstracted to create a unified control schema capable of achieving a variety of tasks.

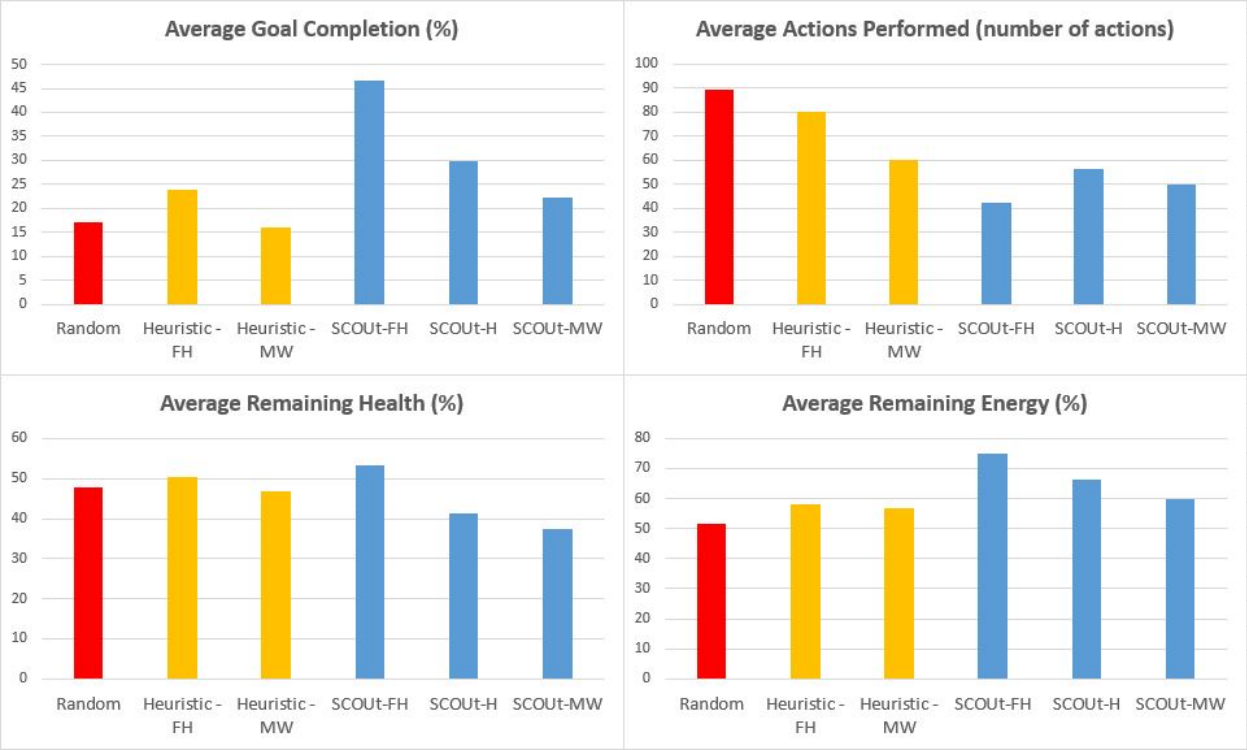

Figure 5.15: Results for $SCOUt_{MW}$ in *Find Human* operations compared against *Random*, $Heuristic_{FH}$, $Heuristic_{MW}$, $SCOUt_{FH}$, and $SCOUt_{H}$. While $SCOUt_{MW}$ did not have the best performance results out of all of the SCOUt controllers, it still performed better than *Random* and both heuristic controllers in the majority of cases. This exemplifies the SCOUt control schema's ability to adapt to new goal types.

Goal change tests for the *Map Water* goal (figure 5.16) hold some interesting results. All three SCOUt controllers outperformed *Random* and both heuristic controllers in every category except for average remaining health. The fact that even $SCOUt_{FH}$ falls victim to poor performance in hazard avoidance suggests that there is level of inevitability for this control schema to eventually choose a detrimental action. On the positive side, the goal completion rates for all three show superiority over the other controllers, with $SCOUt_{FH}$ taking the lead. This could possibly be attributed to $SCOUt_{FH}$ having training where efficient sensor usage is highly rewarded. Another interesting note is that $Heuristic_{FH}$

follows closely behind $Heuristic_{MW}$ in performance, and even takes the lead in average remaining health. This speaks more toward the goal at hand than the control schema. As stated previously in section 5.4, element type mapping is likely a trickier task than expected. The fact that the range of goal completion of heuristic and SCOUt controllers was 24 - 28 percent supports this idea.

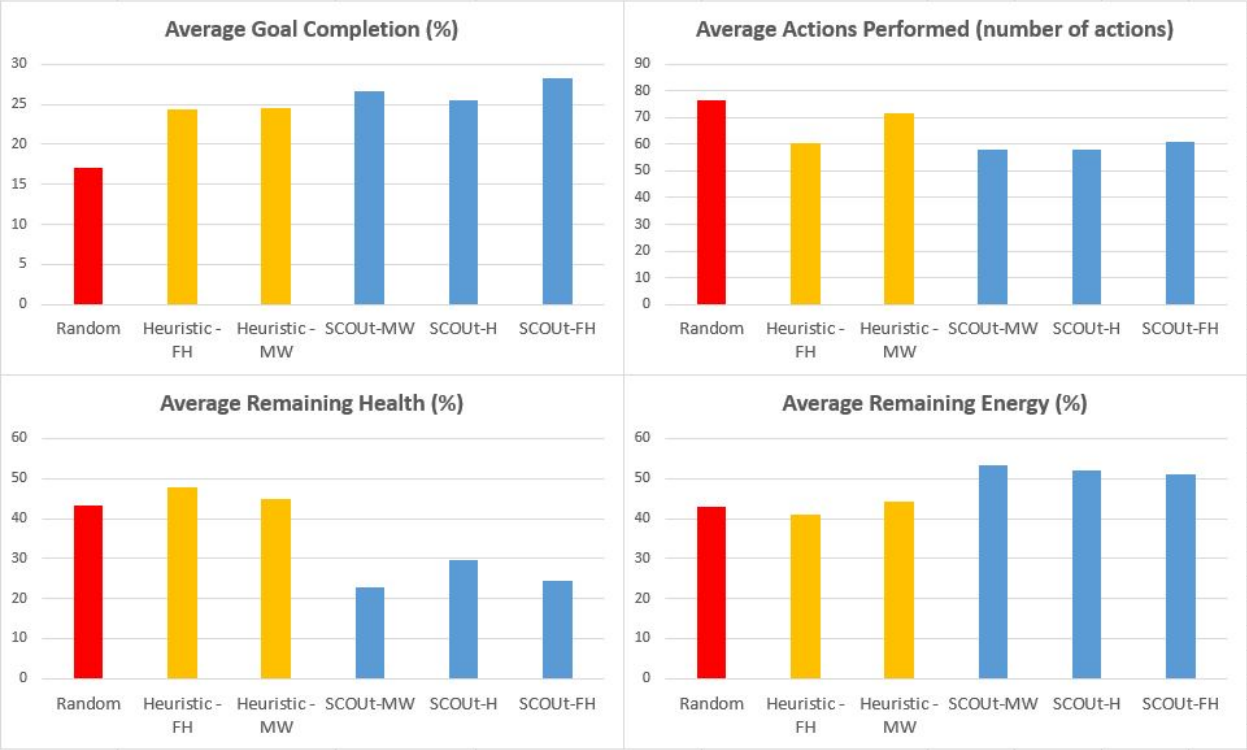

Figure 5.16: Results for $SCOUt_{FH}$ in *Map Water* operations compared against *Random*, $Heuristic_{FH}$, $Heuristic_{MW}$, $SCOUt_{MW}$, and $SCOUt_{H}$. Results show that all SCOUt controllers performed at roughly the same level in all categories. In all categories besides average remaining health, the SCOUt controllers perform best. $SCOUt_{FH}$ takes the lead in goal completion out of all controllers. This again supports the adaptability of the SCOUt control schema.

5.5.2 Sensor Set Changing

Sensor set changing analyzes how a SCOUt controller is able to perform the goal they were trained for after removing a sensor that it had learned to use for achieving the given task. Tests are conducted in the same manner as the goal change tests, where 1000 tests per environment difficulty are run, and performance results are averaged across each difficulty level. The

controllers used in each test are *Random*, the heuristic controller designed for the given goal, the original SCOUt controller trained on the goal (denoted as $< controller - name > -All$), and copies of the same SCOUt controller setup with a variation of available sensors.

First, we will examine the results for the *Map Water* goal (figure 5.17). Because only water and elevation sensors are used in *Map Water* operations (and elimination of the water sensor would result in the inability to achieve any level of goal completion), only one agent variation is considered ($SCOUt_{MW} - NoElevation$). We see $SCOUt_{MW} - All$ and $SCOUt_{MW} - NoElevation$ topping each performance category besides average remaining health. $SCOUt_{MW} - NoElevation$ outperforms $SCOUt_{MW} - All$ in each category outside of remaining health. This is likely due to the fact that $SCOUt_{MW} - NoElevation$ only has a water sensor, and therefore can only choose to scan for water, or move. The drop in average remaining health could then be attributed to the controller's lack of ability to detect hazardous drops in elevation. If this is true, it would suggest that SCOUt's ability to learn hazard avoidance is not entirely flawed.

Figure 5.17: Performance results for $SCOUt_{MW}$ when its elevation sensor was removed ($SCOUt_{MW} - NoElevation$) in *Map Water* operations. It was also compared against an un-altered version of $SCOUt_{MW}$ ($SCOUt_{MW} - All$), $Heuristic_{MW}$, and *Random*. The results show that the SCOUt control schema was able to perform adaptively without the availability of the elevation sensor.

Moving on to testing with the $SCOUt_{FH}$, we see positive results in figure 5.18. Four variations of sensor sets are examined by removing the elevation, water, temperature and then decibel sensors. The four respective controllers are denoted as $SCOUt_{FH} - NoElevation$, $SCOUt_{FH} - NoWater$, $SCOUt_{FH} - NoTemp$ and $SCOUt_{FH} - NoDecibel$. In three out of the four performance categories, the majority of SCOUt controllers perform best. The only SCOUt controller with a large performance difference among the five is $SCOUt_{FH} - NoDecibel$. Surveillance of decibel values within the environment is a critical behavior to tracking down the location of the human anomaly. The fact that $SCOUt_{FH} - NoDecibel$ shows performance drops demonstrates the ability of SCOUt's control schema to learn pattern recognition behaviors for goal completion. Despite the handicap of having no decibel sensor, the controller still outperformed *Random* and $Heuristic_{FH}$ in goal completion, number of

actions performed, and remaining energy. This further demonstrates the adaptability of the SCOUt control schema. $SCOUt_{FH} - NoDecibel$ likely was still able to pick up on the agent using its temperature sensor, which would require the agent to be moved into closer proximity to the human to detect their heat signature. Removal of each other sensor surprisingly had little to no effect on performance compared to $SCOUt_{FH} - All$. Performance in all categories aside from remaining health for all of these controllers were roughly double $Heuristic_{FH}$'s and triple *Random*'s. These results reveal a strong level of adaptability within SCOUt's memory-based reinforcement learning schema when dealing with new agent setups.

Figure 5.18: Performance results for several variations of SCOUt controllers with a sensor removed in *Find Human* operations. These variations were tested against the un-altered SCOUt controller ($SCOUt_{FH}$), $Heuristic_{FH}$, and *Random*. Results show that SCOUt's control schema was able to adapt to each situation where a sensor was removed. The SCOUt controllers outperform $Heuristic_{FH}$ and *Random* in the majority of performance categories. The only noticeable decline in SCOUt's performance is with the removal of the decibel sensor ($SCOUt_{FH} - NoDecibel$). This is likely due to decibel readings being a key indicator of a

5.5.3 Additional Training

The final task of Experiment 2 was to observe how quickly a controller is able to improve and learn new behaviors related to a different goal. Two new controllers are retrained and then tested for completing a new goal. $SCOUt_{FH+}$ begins with a copy of $SCOUt_{FH}$'s trained memory and is then retrained for completing *Map Water*. $SCOUt_{MW+}$ begins with a copy of $SCOUt_{MW}$'s trained memory and is then retrained for completing *Find Human*. The retraining for each new controller follows the same setup seen in section 5.3. Once completed, each controller will be tested in the new goal they were trained against $SCOUt_{FH}$, $SCOUt_{MW}$, the goal's respective heuristic controller, and the *Random* controller. We expect to see that the retrained controllers will perform better than the original SCOUt controller whose memory they use. Additionally, we would like to see measures of performance that are better than the heuristic controller and near the levels of the original SCOUt controller that was trained for the given goal. Tests are conducted in the same fashion as seen in Change Goal and Change Water testing. One thousand operations are conducted per environment difficulty and the results are averaged from the total 3000 operations.

Retraining for both new controllers show trends that are similar to all training seen in our original controllers. $SCOUt_{FH+}$ retraining on *Map Water* (figure 5.19) does not show any increase in goal completion and we see a decline in the average health. Remaining energy and number of actions performed climb and fall for the same reason discussed with the original $SCOUt_{MW}$ controller, where short operations caused by health depletion reflect in less overall activity. $SCOUt_{MW+}$ shows major improvement in goal completion while maintaining a low average number of actions taken (seen in figure 5.20). Additionally, we see minor climbs in the average remaining health and energy of the agent as it learns to operate more efficiently.

Figure 5.19: Iteration testing performance results for $SCOUt_{FH+}$ attempting *Map Water*. All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_{FH+}$ average - *Random* average) VS the number of training iterations completed.

Figure 5.20: Iteration testing performance results for $SCOUt_{MW+}$ attempting *Map Water*. All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_{MW+}$ average - *Random* average) VS the number of training iterations completed.

After training, $SCOUt_{FH+}$ was tested for performance in *Map Water* operations against the original $SCOUt_{FH}$, $SCOUt_{MW}$, $Heuristic_{MW}$, and *Random*. The results seen in figure 5.21 show minor improvements over $SCOUt_{FH}$ for goal completion. Remaining health is slightly lower, but the number actions taken and remaining energy where still equivalent to that of $SCOUt_{FH}$. Again, we see another SCOUt controller outranking the $Heuristic_{MW}$ controller designed specifically for the goal at hand in all categories except remaining health. While hazard avoidance is still an issue, it can still be argued that this controller is operating efficiently as it shows better performance averages in all other categories.

Figure 5.21: Iteration testing performance results for $SCOUt_{FH+}$ attempting *Map Water*. All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_{FH+}$ average - *Random* average) VS the number of training iterations completed.

Figure 5.22 shows the performance of $SCOUt_{MW+}$ against the original $SCOUt_{MW}$, $SCOUt_{FH}$, $Heuristic_{FH}$, and *Random* in *Find Human* operations. Here we see that $SCOUt_{MW+}$ does outrank $SCOUt_{MW}$ and the heuristic controller as expected. $SCOUt_{FH}$ still takes the lead in every category, but the new controller does show major overall improve-

ments due to its additional training. This further supports the argument that a memory-based reinforcement learning model holds many adaptive attributes for facing new situations.

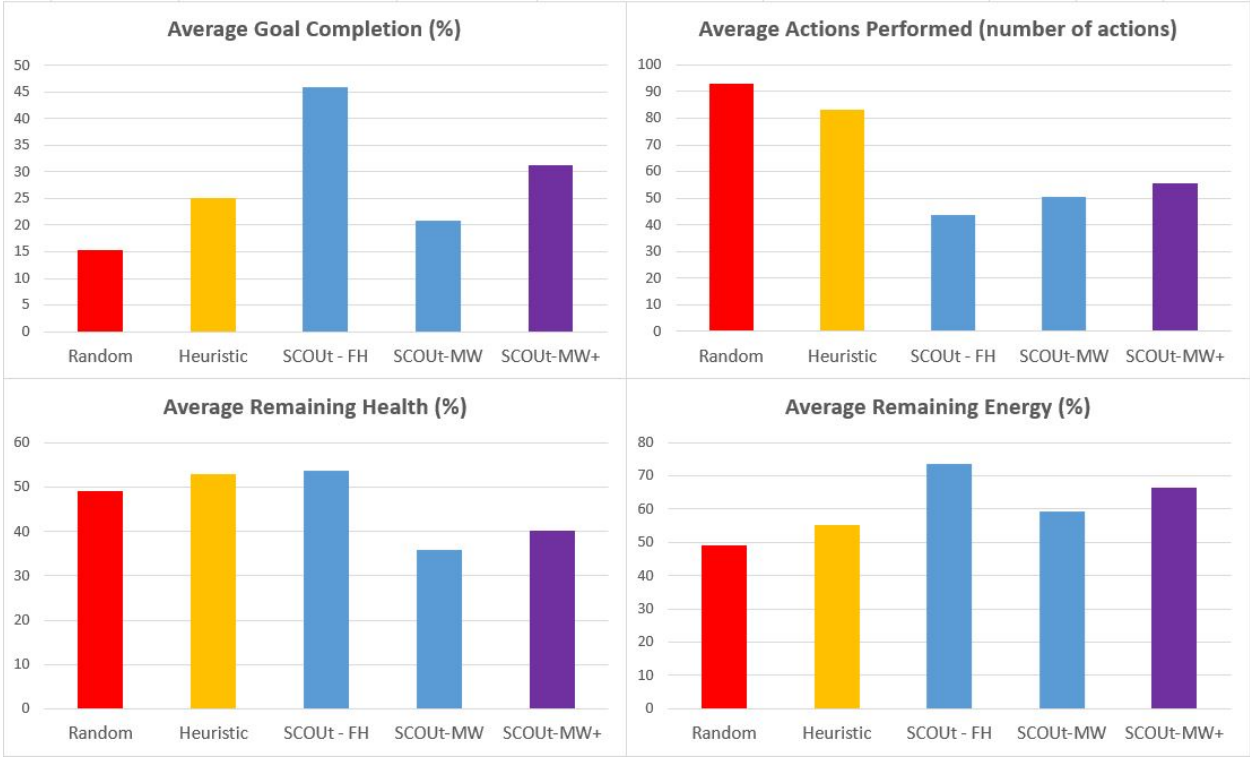

Figure 5.22: Iteration testing performance results for $SCOUt_{MW+}$ attempting *Map Water*. All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_{MW+}$ average - *Random* average) VS the number of training iterations completed.

5.6 Discussion

Our testing revealed that SCOUt's control schema was in fact adaptive to new tasks. In Experiment 2, we see SCOUt controllers performing better than random and heuristic approaches, even when faced with disadvantages, such as lack of sensors or no knowledge of how to complete a goal. These are features that we don't see in our heuristic models. The heuristic approaches were able to perform adequately in their respective goals, but due to the goal-specific logic that was used in their control models, they did not perform well when presented to a new goal type. We do see an issue with SCOUt's hazard avoidance, as results in all testing showed poor ranking in the category of average remaining health. Despite

this, SCOUt still showed superior results in the other three categories for the majority of tests. It is suspected that if the hazard avoidance issue were to be fixed, we would see even better results in all performance categories. Another notable feature within our results was $SCOUt_{FH}$'s performance in the goal change testing (figure 5.15). Here we saw that the SCOUt controller without a decibel sensor equipped showed noticeable performance drops, but still performed better than the random and heuristic controllers. This signifies two things. First, SCOUt is learning behaviors related to goal completion. Changes in decibel levels within the environment are the strongest indicator to finding a human's location. The fact that $SCOUt_{FH} - NoDecibel$ showed lower performance compared to all other SCOUt controllers in the test suggests that the controller has learned search behaviors related to analyzing the environment. Second, because SCOUt was still able to outperform the random and heuristic controllers despite its disadvantage in this situation, we argue that SCOUt's control model is in fact adaptive. While $SCOUt_{FH} - NoDecibel$ would not be able to pick up on the human's sound effect, it was able to use other learned information to still guide it to successfully goal completions.

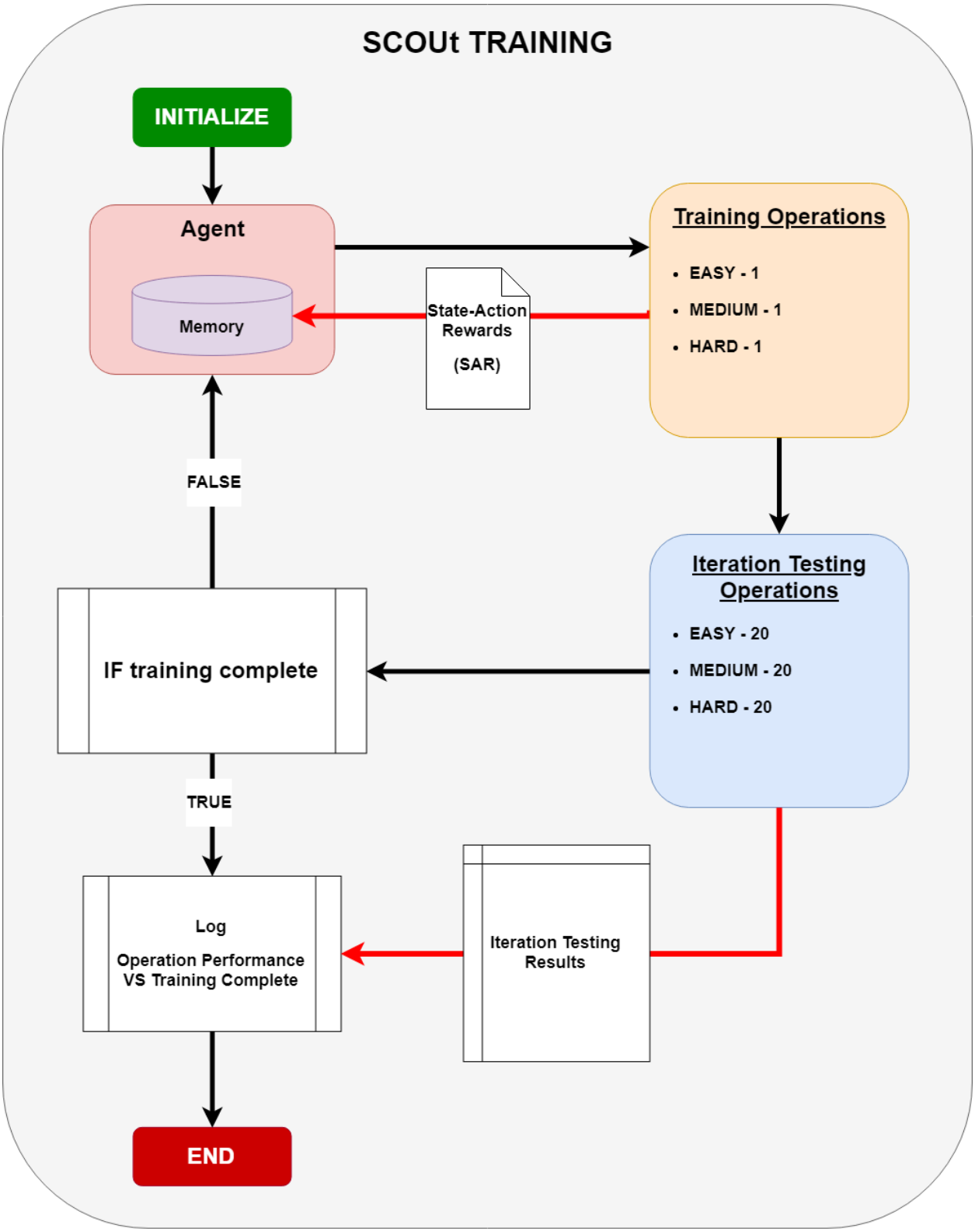

Figure 5.2: This diagram shows the training process that each SCOUt controller will follow. Training is completed after 30 iterations of the process. The number of operations for each environment difficulty in training and in iteration testing are shown. The goal for the operation will depend on the specific SCOUt controller that is being trained.

Chapter 6

Conclusion

The wide variety of use cases for autonomous robotics in the field of exploration suggests the need of a unified solution for the setup and execution of related operations. Unique environments and tasks are found throughout the problem spaces of exploration that often require a robotic agent to be constructed from the ground up. SCOUt provides a platform for both modeling and simulating wide varieties of goal driven tasks within an environment, as well as an adaptive control solution for robotic agents. The abstracted data structures used by SCOUt offer a framework that can easily be utilized and expanded upon by the growing communities of robotics and exploration.

Simulation testing is a valuable tool for planning out exploration-based operations. It offers both cost and risk avoidance solutions for building, training, and testing new ideas. SCOUt's simulation platform touches all of these features in the form of a user friendly setup tool. New testing scenarios can be created with minimal input allowing a user to direct their focus towards the primary tasks at hand. In this project, thousands of operations were simulated with a variety agents, environments, goals, and the interactions between them. Each operation holds the opportunity to present a unique scenario for each controller to be tested within. Features of the controlled agent can be adjusted to reflect its available sensors, maneuverability, and durability to environmental factors. Environments are procedurally

generated to produce unique features within similar settings, and agent starting positions are chosen randomly within. Data collected from all of these operations could then be averaged, charted, and analyzed to track the performance of different controllers across the vast problem space. The simulation platform also allowed multiple controllers to be tested and compared in identical conditions, removing discrepancies between each unique operations faced.

The SCOUt project aimed to uncover the concept of a generalized work flow that lies within exploration in unknown environments. Through research in a wide variety of studies, we found the core components of such operations to be a continuous cycle of gathering and analyzing data to draw new beliefs and conclusions that help to progress a goal. Our memory-based learning control schema presented in this paper demonstrates a unified solution built on this premise. Data from both the environment and the agent is condensed into a single "state" that is passed as input to the decision model. The model will then decide if actions should be performed to gather more data to analyze, or if the controller should navigate the agent towards new, potentially interesting areas within the environment. Both types of actions will work together in the cycle of state analysis and decision making to progress goal completion. Two types of exploration based goals were examined in varying environmental conditions: anomaly searching and element type mapping. SCOUt's unique control schema was tested for its performance and adaptability in these two types of scenarios. Heuristic control schemas were also built for each of the specific goals, and were tested in tandem to the SCOUt's control model. The heuristic controllers follow the same process of state analysis and action decisions, but apply logical analyses rather than a prediction model based on memory of past events. Heuristic controllers created are focused on completing one specific goal and reflect a specialized, non-adaptive control solution that might be applied in these scenarios.

We see in our results that SCOUt's model was in fact able to perform adaptively across a variety of situations. SCOUt demonstrated the ability to learn task related behaviors that could be applied to multiple goals. When changing the goal, available sensors, and environ-

mental settings, SCOUt was able to maintain an efficient level of performance. In scenarios such as the removal of sensors, some controllers showed little to no drop in performance at all. In the majority of performance categories, the memory-based learning controllers showed superior results compared to that of the heuristic controllers. With better average rates of goal completion, number of actions taken, and remaining energy, we see how SCOUt's model is both adaptive *and* efficient. Even in some cases where SCOUt is placed in scenarios in which it was not trained, we still see results that are more efficient than the heuristic approaches. On these grounds, it is strongly believed that autonomous control can be abstracted into a unified solution for exploration related operations.

One area of the memory-based control schema that could use improvement is hazard avoidance. In the *Map Water* operations especially, we see SCOUt controllers suffer in their ability to maintain the agent's health. SCOUt's analytical ability relied heavily on a dense network of weights tied to examining past rewards it had received for performing actions. Both the weight and reward systems create a grey area that likely caused poor performance results in hazard avoidance. A proper level of importance in the agent's remaining health was not being reflected in the decision model. This is a common caveat seen when artificial intelligence (AI) is left to independently control every aspect of a task. When AI is introduced to real-world environments, there are certain aspects of problems that typically have desired behaviors which seem trivial from a human's instinctual base of knowledge. However, it cannot be guaranteed that an AI will pick up on these since they can only "think" analytically and not instinctually. When facing undesirable behavior, AI solutions are often enhanced using sets of "rules" that they must follow. For example, when building autonomous self-driving vehicles, rules are often embedded into the vehicle's control schema to assist with safety (stay within a designated lane, always drive at a safe distance behind the vehicle in front of you, etc.). Applying similar sets of rules to SCOUt's control schema could have greatly improved its performance in hazard avoidance and subsequently in all other areas of performance that were measured. This project elected to forgo any hard coded rules into SCOUt's control

schema for two reasons. First, all testing and training was done in simulation so there were no real-world risks involved in a "rogue" AI approach to control. Second, we wanted to test the memory-based learning model to the fullest of its capabilities without the assistance of any external knowledge.

This project opens research potentials into similar abstracted approaches within the fields of autonomous robotics and exploration. Results suggest the potential for other categories of autonomics that could be abstracted into their own unified control models. Top-down approaches could be applied by finding the underlying work flow of each sub-field to generalize the process and reduce the amount of repetitive work required in finding solutions on a case-by-case basis.

6.1 Future Work

This section covers a few ideas for future work that could improve upon the current state of the SCOUt project.

Behavior Rules Adding a set of hard-coded rules to SCOUt's decision model would likely lead to even better results. Ideally, we would want to prevent the controller from selecting actions that would damage the agent or have no benefit to the operation (e.g., using sensors in areas that have already been mapped or moving into quadrants that are already mapped). Each time the controller is given a set of valid actions to choose from, they could be passed through the set of rules and any undesirable actions could be removed from consideration. Using a rule set would additionally provide a tool for investigating suspected causes of poor performance. In our results, we saw a poor performance in remaining health and we suspected that the controller was not properly learning to avoid hazards. Rules for avoiding movement into cells with water or large drops in elevation could each be implemented and tested independently. Testing each rule in isolation would help to determine if poor health performances were being caused by one or the other, by neither, or by both. Rules which

show obvious performance improvements could then be implemented by the controller to prevent undesired behaviors.

Artificial Neural Network Integration. Original ideas for the adaptive control schema included the use of an artificial neural network (ANN). The ANN would take an agent state and a list of valid actions as input and output the selected action. Due to the dynamic nature of agent states, use of an ANN was ruled out. If an agent changes the sensors it is equipped with, the ANN would need to account for a new set of input values due to the different set of `ElementState`s that would be found in the `AgentState`. However, the memory-based approach required the use of several weights to guide each state comparison and action scoring equation. Our solution was to optimize the weight set using a genetic algorithm (GA). Use of an ANN in place of the state comparison equations would eliminate the need for external weights to be provided. This could further enhance the adaptability of SCOUt's decision model. State-action rewards (SARs) could be compared against the current state by passing attributes through the ANN to output difference scores. The ANN could be trained with backpropagation using the existing simulation platform and the short-term and long term-reward systems. This model would likely require the ANN to remain "open" in the sense that its weights are constantly being trained during every operation.

Improved Memory Management. Improvements to both the process of saving and loading the SCOUt controller's memory could be made. For saving memory, the application of cluster would cut down on memory storage requirements. Currently, SCOUt saves the last 20 SARs from each operation, and uniformly samples 5 percent of the remaining SARs. Instead of only saving a sub-set, all SARs could be saved in memory and a data clustering algorithm could later be applied to "clean-up" the memory. The clusters could then be averaged and given a weight based on the number of SARs that fell within the cluster. This would eliminate redundancies within the memory set, as well as reduce the amount of computational time required for the controller to conduct state comparisons. As for loading memory, an adaptive approach could be taken to select only a sub-set of the SARs to use

in each operation. The controller could begin by analyzing the given goal and agent setup to choose SARs from memory that correlate to the operation at hand. We expect that this would have two positive effects: less memory means less computational time for the action decision model, and a more concise memory set would yield higher performance results.

Integration of Goals into Agent States. Currently, the only place that we see the reflection of the current goal in SCOUt's decision model is with the `indicator` flag found in each `ElementState`. A higher-level approach could be taken so that the decision model analyzes an *OperationState* rather than just an `AgentState`. This could be achieved by classifying the goal type and goal instance. For example, a *Find Human OperationState* would also include a current goal type of "anomaly searching" and a goal instance of "human" since this is the specific anomaly that the agent is searching for. These attributes could then be weighted into the state comparison system to produce a more concise *OverallStateDifference* score.

Bibliography

[1] Akash Arora, Robert Fitch, and Salah Sukkarieh. An Approach to Autonomous Science by Modeling Geological Knowledge in a Bayesian Framework. *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3803–3810, September 2017. arXiv: 1703.03146.

[2] Akash Arora, P. Michael Furlong, Robert Fitch, Terry Fong, Salah Sukkarieh, and Richard Elphic. Online Multi-modal Learning and Adaptive Informative Trajectory Planning for Autonomous Exploration. In Marco Hutter and Roland Siegwart, editors, *Field and Service Robotics*, volume 5, pages 239–254. Springer International Publishing, Cham, 2018.

[3] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A Brief Survey of Deep Reinforcement Learning. *IEEE Signal Processing Magazine*, 34(6):26–38, November 2017. arXiv: 1708.05866.

[4] Shi Bai, Fanfei Chen, and Brendan Englot. Toward autonomous mapping and exploration for mobile robots through deep supervised learning. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2379–2384, Vancouver, BC, September 2017. IEEE.

[5] Mike Bostock. Data-Driven Documents (D3).

[6] Travis Brown. circe.

[7] A. J. Clark. Evolving adabot: A mobile robot with adjustable wheel extensions. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, November 2017.

[8] Jonathan E. Clark, Daniel I. Goldman, Tao Chen, Robert J. Full, and Daniel E. Koditschek. Toward a Dynamic Vertical Climbing Robot. 2006.

[9] Justin Clark and Rafael Fierro. Mobile robotic sensors for perimeter detection and tracking. *ISA Transactions*, 46(1):3–13, February 2007.

[10] Jonathon Doran and Ian Parberry. Controlled Procedural Terrain Generation Using Software Agents. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(2):111–119, June 2010.

[11] Wolfgang Fink, James M. Dohm, Mark A. Tarbell, Trent M. Hare, Victor R. Baker, Dirk Schulze-Makuch, Roberto Furfaro, Alberto G. Fairen, Ty P.A. Ferre, Hideaki Miyamoto, Goro Komatsu, and William C. Mahaney. Tier-Scalable Reconnaissance Missions For The Autonomous Exploration Of Planetary Bodies. In *2007 IEEE Aerospace Conference*, pages 1–10, Big Sky, MT, USA, 2007. IEEE.

[12] Zhiwei Fu, Bruce L. Golden, Shreevardhan Lele, S. Raghavan, and Edward A. Wasil. A Genetic Algorithm-Based Approach for Building Accurate Decision Trees. *INFORMS Journal on Computing*, 15(1):3–22, February 2003.

[13] Duncan W. Haldane, Kevin C. Peterson, Fernando Garcia Bermudez, and Ronald S. Fearing. Animal-inspired design and aerodynamic stabilization of a hexapedal millirobot. *2013 IEEE International Conference on Robotics and Automation*, pages 3279–3286, 2013.

[14] Aaron M. Hoover, Samuel Burden, Xiao-Yu Fu, S. Shankar Sastry, and Ronald S. Fearing. Bio-inspired design and dynamic maneuverability of a minimally actuated six-legged

robot. *2010 3rd IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics*, pages 869–876, 2010.

[15] James K. Hopkins, Brent W. Spranklin, and Satyandra K. Gupta. A survey of snake-inspired robot designs. *Bioinspiration & biomimetics*, 4(2):021001, 2009.

[16] Bahare Kiumarsi, Kyriakos G. Vamvoudakis, Hamidreza Modares, and Frank L. Lewis. Optimal and Autonomous Control Using Reinforcement Learning: A Survey. *IEEE Transactions on Neural Networks and Learning Systems*, 29(6):2042–2062, June 2018.

[17] Alex Kossett and Nikolaos Papanikolopoulos. A robust miniature robot design for land/air hybrid locomotion. *2011 IEEE International Conference on Robotics and Automation*, pages 4595–4600, 2011.

[18] Stella Latscha, Michael Kofron, Anthony Stroffolino, Lauren Davis, Gabrielle Merritt, Matthew Piccoli, and Mark Yim. Design of a Hybrid Exploration Robot for Air and Land Deployment (H.E.R.A.L.D) for urban search and rescue applications. *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1868–1873, 2014.

[19] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, 2006.

[20] James McCaffrey. How To Standardize Data for Neural Networks, 2014.

[21] npm, Inc. node-fetch.

[22] npm, Inc. Node Package Manager.

[23] Open-Scource Software. Babel.

[24] Open-Scource Software. webpack.

[25] Open-Source Software. Electron.

[26] Open-Source Software. http4s.

[27] Daniel Perea Ström, Igor Bogoslavskyi, and Cyrill Stachniss. Robust exploration and homing for autonomous robots. *Robotics and Autonomous Systems*, 90:125–135, April 2017.

[28] David L. Poole and Alan K. Mackworth. *Artificial intelligence: foundations of computational agents*. Cambridge University Press, New York, 2010.

[29] Anthony R Cassandra. A survey of POMDP applications. *Working Notes of AAAI 1998 Fall Symposium on Planning with Partially Observable Markov Decision Processes*, January 1998.

[30] Mac Schwager, Philip Dames, Daniela Rus, and Vijay Kumar. A Multi-robot Control Policy for Information Gathering in the Presence of Unknown Hazards. In Henrik I. Christensen and Oussama Khatib, editors, *Robotics Research*, volume 100, pages 455–472. Springer International Publishing, Cham, 2017.

[31] Guy Shani, Joelle Pineau, and Robert Kaplow. A survey of point-based POMDP solvers. *Autonomous Agents and Multi-Agent Systems*, 27(1):1–51, July 2013.

[32] Olivier Sigaud and Stewart W. Wilson. Learning classifier systems: a survey. *Soft Computing*, 11(11):1065–1078, September 2007.

[33] Lauren M. Smith, Roger D. Quinn, Kyle A. Johnson, and William R. Tuck. The Tri-Wheel: A novel wheel-leg mobility concept. *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4146–4152, 2015.

[34] C. Stachniss, D. Hahnel, and W. Burgard. Exploration with active loop-closing for FastSLAM. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 2, pages 1505–1510, Sendai, Japan, 2004. IEEE.

[35] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning. MIT Press, Cambridge, Mass, 1998.

[36] Lei Tai, Shaohua Li, and Ming Liu. Autonomous exploration of mobile robots through deep neural networks. *International Journal of Advanced Robotic Systems*, 14(4):172988141770357, July 2017.

[37] The jQuery Foundation. jQuery.

[38] Li Yi and Kang Wanli. A New Genetic Programming Algorithm for Building Decision Tree. *Procedia Engineering*, 15:3658–3662, 2011.

Chapter 7

APPENDIX

7.1 Appendix A: Code Listings

This appendix contains all code listings referenced within the paper.

7.1.1 Environment Data Structures

This section covers all data structures related to an environment. Together these traits, classes, and instances can be combined in different ways to create unique models of real-world environments.

Appendix 7.1: A class for representing a real-world environment. It is laid out in a grid of cells that contain information related to each of their respective areas within the grid.

```
1 class Environment (
2   name: String,
3   height: Int,
4   width: Int,
5   scale: Double,
6   grid: Array[Array[Cell]]
7 )
```

Appendix 7.2: A class that represents a sub-section of an environment. The data stored in each instance represents the features that are found within the `Cell`'s area within the environment.

```
1 class Cell (
2   x: Int,
3   y: Int,
4   elements: Array[Element],
5   anomalies: Array[Anomaly]
6 )
```

Appendix 7.3: An `Element` trait is a generalized representation of a measurable feature type within an environment. Specific element types can be created by extending this trait. Each instance defines specific information about what values can be represented for the specific element type.

```
1 trait Element {
2   val name: String
3   val value: Double
4   val unit: String
5   val radial: Boolean
6   val lowerBound: Double
7   val upperBound: Double
8 }
```

Appendix 7.4: `Elevation` is a class which extends the `Element` trait. This class models elevation levels within an environment. Different instances can be created to specify a level of elevation for each cell in an environment.

```
1 class Elevation(var value: Option[Double]) extends Element {
2   val name = "Elevation"
3   val unit = "ft"
4   val constant = true
5   val radial = false
6   val lowerBound = -500.0
7   val upperBound = 500.0
8   def this(d: Double) = this(Some(d))
9   def this()          = this(None)
10 }
```

Appendix 7.5: An `Anomaly` is any object within an environment that may be of interest. Anomalies often have a set of effects that will alter the environment around them. This trait can be extended to define specific types of anomalies that can be represented in an environment.

```
trait Anomaly {
  val name: String
  val area: Double
  val effects: List[Effect]
}
```

Appendix 7.6: The `Effect` trait is a generalized description of an alteration that an `Anomaly` has on the environment. Effects will alter a single element type in an area of the environment that the anomaly is located within.

```
trait Effect {
  val seed: Element
  val range: Double
}
```

Appendix 7.7: `Human` is a specific `Anomaly` class. This class represents a peron that could be found in an environment. Human's have two defined `Effects`: sound and heat. These effects will alter the decibel and temperature element values in the human's general area within the environment.

```
class Human(
  val name: String = "Human",
  val area: Double = 6.0,
  val effects: List[Effect] = List(
    new Sound(seed = new Decibel(40.0)),
    new Heat(seed = new Temperature(98.6))
  )
) extends Anomaly {
  def this(formData: Map[String, String]) = this(
    area = formData("Area").toDouble,
    effects = List(
      new Sound(seed = new Decibel(formData("Sound").toDouble))
        ,
      new Heat(seed = new Temperature(formData("Heat").toDouble
        ))
    )
  )
}
```

Appendix 7.8: A `Layer` holds a collection of instances of a specific `Element` class. The collection is represented as a 2-dimentional grid that is relative to an `Environment` grid. They can be thought of as the same structure as an environment, but only containing information about a single element type.

```
class Layer (
  length: Int,
  width: Int,
  layer: Array[Array[Element]]
)
```

Appendix 7.9: An `Agent` represents a physical member capable of acting within an environment. The class defines a controller for selecting actions, sensors that the agent is equipped with, mobility and durability features of the agent for modeling interactions with an environment, and several internal status variables.

```
class Agent (
  name: String,
  controller: Controller,
  sensors: List[Sensor],
  internalMap: Array[Array[Cell]],
  xPosition: Int,
  yPosition: Int,
  health: Double,
  energyLevel: Double,
  mobility: Mobility,
  durabilities: List[Durability]
)
```

7.1.2 Agent Data Structures

This section contains data structures that are representative of an agent. These structures model the different capabilities an agent has to interact with an environment.

7.1.3 Environment Generation Data Structures

This section includes the data structures used to guide the procedural generation of unique environments.

Appendix 7.10: A `Sensor` is a tool that an agent can utilize to collect data about a specific element type within an environment. Sensors have a set range and energy cost related to using them.

```
1 trait Sensor {
2   val elementType: String
3   val range: Double
4   val energyExpense: Double
5   val indicator: Boolean
6   val hazard: Boolean
7 }
```

Appendix 7.11: `Mobility` contains a set of variables related to how an agent will be able to safely move within an environment.

```
1 class Mobility (
2   movementSlopeUpperThreshHold: Double,
3   movementSlopeLowerThreshHold: Double,
4   movementDamageResistance: Double,
5   movementCost: Double,
6   slopeCost: Double
7 )
```

Appendix 7.12: `Durability` defines how an agent will interact with an element type in an environment. Different agents will each have strengths and weaknesses defined by how they will react when in contact with certain elements in an environment.

```
1 class Durability (
2   elementType: String,
3   damageUpperThreshold: Double,
4   damageLowerThreshold: Double,
5   damageResistance: Double
6 )
```

Appendix 7.13: An `AgentState` represents an instance of the internal status of an agent and the information that the agent knows about its environment. An `Agent`'s internal map is condensed into a set of sub-states within the entire agent state for each element type that the agent has knowledge of.

```
class AgentState (
  xPosition: Int,
  yPosition: Int,
  health: Double,
  energyLevel: Double,
  elementStates: List[ElementState]
)
```

Appendix 7.14: `ElementState`s are representative of an agent's knowledge of a specific element type in an environment. The information contained is directly related to what the agent has gathered into its internal map through the use a sensor. Specific known values of the element type are divided into four quadrants relative to the agent's current position.

```
class ElementState (
  elementType: String,
  indicator: Boolean,
  hazard: Boolean,
  percentKnownInSensorRange: Double,
  northQuadrant: QuadrantState,
  southQuadrant: QuadrantState,
  westQuadrant: QuadrantState,
  eastQuadrant: QuadrantState
)
```

Appendix 7.15: A `QuadrantState` represents a collection of known information about a specific element type in a sub-set of cells within an environment. The data is condensed to an average known value differential and immediate known value differential relative to the value in the cell that the agent currently occupies.

```
class QuadrantState (
  percentKnown: Double,
  averageValueDifferential: Option[Double],
  immediateValueDifferential: Option[Double]
)
```

Appendix 7.16: `Controller`s are the decision-making models that are used to select actions for an agent to take. The process that each controller uses to select an action vary, but must choose from a set of valid actions and can use information that the agent has gathered while exploring the environment.

```
1 trait Controller {
2   def setup(mapHeight: Int, mapWidth: Int): Unit
3   def selectAction(actions: List[String], state: AgentState):
      String
4   def shutDown(stateActionRewards: List[StateActionReward]):
      Unit
5 }
```

Appendix 7.17: `EnvironmentTemplate`s hold the entire collection of attributes that are used to guide the process of generating an environment.

```
1 class EnvironmentTemplate (
2   name: String,
3   height: Int,
4   width: Int,
5   scale: Double,
6   elementSeeds: List[ElementSeed],
7   terrainModification: List[TerrainModification],
8   anomalies: List[Anomaly]
9 )
```

Appendix 7.18: `ElementSeed` is a trait used to define helper classes for specific `Element` classes. They have a set of attributes that can be set to guide how the element type will be initialized in an environment and functions related to the actual process in which the element type will be procedurally generated within the environment.

```
1 trait ElementSeed {
2   val elementType: String
3   def buildLayer(height, width, scale)
4 }
```

Appendix 7.19: A `TerrainModification` is a trait used for defining processes to alter element types within the environment. These help to add unique features within element types found in the environment.

```
trait TerrainModification (
  val name: String
  val elementTypes: List[String]
  def modify(layers: List[Layer])
}
```

Appendix 7.20: `GaussianData` holds the mean and standard deviation of a collection of values.

```
  class GaussianData(
    mean: Double,
    std: Double
  )
```

7.1.4 State Comparison Data Structures

This section includes data structures related to comparing multiple `AgentState`s to each other.

Appendix 7.21: `StateActionDifference` holds values related to a comparison that was made between a current `AgentState` and an `AgentState` within SCOUt's memory of state-action rewards. Each instance defines the differences that were calculated between two states, an overall state difference, the action that was taken by the agent in the state-action reward, and then the short-term and long-term rewards that the were received for the action.

```
class StateActionDifference(
  overallStateDifference: Double,
  action: String,
  shortTermScore: Double,
  longTermScore: Double
)
```

7.1.5 Experimentation Setup

This section includes the code listings for the `Agent` configuration and `EnvironmentTemplates` used in experimentation.

7.2 Appendix B: Additional Figures

This appendix holds all figures that were not placed in the main body of the paper.

7.2.1 Training Variation 1

This section contains results for our first variation of training the three SCOUt controller memories ($SCOUt_{FH}$, $SCOUt_{MW}$, and $SCOUt_{H}$). During training and iteration testing, the long-term reward (algorithm 3) was adjusted to only factor in a `goalReward` if the agent had remaining health at the end of an operation. This was done to observe how SCOUt controllers would change the agent's behavior when their memory of state-action rewards reflected smaller long-term rewards from operations where the agent's health was depleted. It was hoped that we would see better performance in average remaining energy and subsequently all other areas. However, we found no significant improvements in performance. Appendix 7.1 shows the results for $SCOUt_{FH}$, Appendix 7.2 shows the results for $SCOUt_{MW}$, and Appendix 7.3 and 7.4 show results for $SCOUt_{H}$ in *Find Human* and *Map Water* operations, respectively.

7.2.2 Training Variation 2

This section contains results for our second variation of training the three SCOUt controller memories ($SCOUt_{FH}$, $SCOUt_{MW}$, and $SCOUt_{H}$). During training and iteration testing, the *goalRewardWeight* used for calculating long-term reward (algorithm 3) was set to 1.5. This was done to observe how SCOUt controllers would change the agent's behavior when their memory of state-action rewards reflected a stronger emphasis on the level of goal completion

Appendix 7.22: Instances of the `Agent` class and `Sensor` classes that are used in experimentation. Some attributes are set per operation during experiments. These are marked with a comment "Defined Per Operation."

```
// AGENT

class Agent (
  name: String,                          // Defimed Per Operation
  controller: Controller,                // Defimed Per Operation
  sensors: List[Sensor],                 // Defimed Per Operation
  internalMap: Array[Array[Cell]],       // Defimed Per Operation
  xPosition: Int,                        // Defimed Per Operation
  yPosition: Int,                        // Defimed Per Operation
  health: Double = 100.0,
  energyLevel: Double = 100.0,
  mobility: Mobility = new Mobility (
   movementSlopeUpperThreshHold = 1.0,
   movementSlopeLowerThreshHold = -1.0,
   movementDamageResistance = 0.0,
   movementCost = 0.5,
   slopeCost = 0.2
  ),
  durabilities: List[Durability] = List(
   new Duribility (
     elementType = "Water␣Depth",
     damageUpperThreshold = 0.25,
     damageLowerThreshold = Double.MaxValue,
     damageResistance = 0.0
   ),
   new Duribility (
     elementType = "Temperature",
     damageUpperThreshold = 150.0,
     damageLowerThreshold = -50.0,
     damageResistance = 0.0
   )
  )
)

// SENSORS

val elevationSensor = new Sensor (
  elementType = "Elevation",
  range = 30.0,
  energyExpense = 0.5,
  hazard = true,
  indicator: Boolean  // Defined Per Operation
)

val waterSensor = new Sensor (
  elementType = "Water␣Depth",
```

Appendix 7.23: JSON data storing the an environment template used in experimentation. This template is title *EASY* as it represents a relatively easy environment for an operation to take place in. The environment is 8×8 cells and contains only one modification for a pool of water to be generated within the environment.

```
{
  "name" : "EASY",
  "height" : 8,
  "width" : 8,
  "elements" : [
    "Elevation",
    "Decibel",
    "Water␣Depth",
    "Temperature"
  ],
  "seeds" : {
    "Elevation" : {
      "Average" : {
        "value" : 0,
        "unit" : "ft"
      },
      "Deviation" : {
        "value" : 1,
        "unit" : "ft"
      }
    },
    "Decibel" : {
      "Average" : {
        "value" : 10,
        "unit" : "db"
      }
    },
    "Water␣Depth" : {
    },
    "Temperature" : {
      "Average" : {
        "value" : 50,
        "unit" : "degrees␣F"
      },
      "Deviation" : {
        "value" : 0.5,
        "unit" : "degrees␣F"
      }
    }
  },
  "terrain-modifications" : [
    {
      "terrain-modification" : "Water␣Pool␣Modification",
      "Max␣Depth" : {
        "value" : 30,
```

Appendix 7.24: JSON data storing the an environment template used in experimentation. This template is title *MEDIUM* as it presents a few challenging features that an agent may face. The environment is 10×10 cells and contains both a "hill" elevation modification and a water pool modification. Compared to the *EASY* environment template, *MEDIUM* has higher ambient levels of decibel and temperature values, making it slightly more difficult to identify the effects of a human anomaly within the environment.

```
{
  "name" : "MEDIUM",
  "height" : 10,
  "width" : 10,
  "elements" : [
    "Elevation",
    "Decibel",
    "Latitude",
    "Water␣Depth",
    "Longitude",
    "Temperature"
  ],
  "seeds" : {
    "Elevation" : {
      "Average" : {
        "value" : 0,
        "unit" : "ft"
      },
      "Deviation" : {
        "value" : 3,
        "unit" : "ft"
      }
    },
    "Decibel" : {
      "Average" : {
        "value" : 15,
        "unit" : "db"
      }
    },
    "Water␣Depth" : {
    },
    "Temperature" : {
      "Average" : {
        "value" : 60,
        "unit" : "degrees␣F"
      },
      "Deviation" : {
        "value" : 1,
        "unit" : "degrees␣F"
      }
    }
  },
  "terrain-modifications" : [
```

Appendix 7.25: JSON data storing the an environment template used in experimentation. This template is title *HARD* as it presents many challenging features that an agent may face. The environment is 12×12 cells and contains a "hill" elevation modification, a "valley" elevation modification, a water pool modification, and a water stream modification. Additionally, the ambient levels of decibel values and temperature values are raised and the sound and heat effects of the human anomaly are suppressed even more than seen in the *MEDIUM* environment template. The combination of all of these factors create environments that are both highly difficult to safely navigate and difficult to identify anomaly effects within.

```
{
  "name" : "HARD",
  "height" : 12,
  "width" : 12,
  "elements" : [
    "Elevation",
    "Decibel",
    "Latitude",
    "Water␣Depth",
    "Longitude",
    "Temperature"
  ],
  "seeds" : {
    "Elevation" : {
      "Average" : {
        "value" : 0,
        "unit" : "ft"
      },
      "Deviation" : {
        "value" : 3,
        "unit" : "ft"
      }
    },
    "Decibel" : {
      "Average" : {
        "value" : 20,
        "unit" : "db"
      }
    },
    "Water␣Depth" : {
    },
    "Temperature" : {
      "Average" : {
        "value" : 70,
        "unit" : "degrees␣F"
      },
      "Deviation" : {
        "value" : 2,
        "unit" : "degrees␣F"
      }
    }
```

Figure 7.1: Iteration testing performance results for $SCOUt_{FH}$ attempting *Find Human* using setup variation 1 (see subsection 5.3.2). All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_{FH}$ average - *Random* average) VS the number of training iterations completed.

Figure 7.2: Iteration testing performance results for $SCOUt_{MW}$ attempting *Map Water* using setup variation 1 (see subsection 5.3.2). All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_{MW}$ average - *Random* average) VS the number of training iterations completed.

Figure 7.3: Iteration testing performance results for $SCOUt_H$ attempting *Find Human* using setup variation 1 (see subsection 5.3.2). All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_H$ average - *Random* average) VS the number of training iterations completed.

Figure 7.4: Iteration testing performance results for $SCOUt_H$ attempting *Map Water* using setup variation 1 (see subsection 5.3.2). All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_H$ average - *Random* average) VS the number of training iterations completed.

Figure 7.5: Iteration testing performance results for $SCOUt_{FH}$ attempting *Find Human* using setup variation 2 (see subsection 5.3.2). All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_{FH}$ average - *Random* average) VS the number of training iterations completed.

attained. It was hoped that we would see better performance in all categories. While only slight improvements were observed, this method was chosen for all testing conducted in our experimentation. Appendix 7.5 shows the results for $SCOUt_{FH}$, Appendix 7.6 shows the results for $SCOUt_{MW}$, and Appendix 7.7 and 7.8 show results for $SCOUt_H$ in *Find Human* and *Map Water* operations, respectively.

Figure 7.6: Iteration testing performance results for $SCOUt_{MW}$ attempting *Map Water* using setup variation 2 (see subsection 5.3.2). All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_{MW}$ average - *Random* average) VS the number of training iterations completed.

Figure 7.7: Iteration testing performance results for $SCOUt_H$ attempting *Find Human* using setup variation 2 (see subsection 5.3.2). All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_H$ average - *Random* average) VS the number of training iterations completed.

Figure 7.8: Iteration testing performance results for $SCOUt_H$ attempting *Map Water* using setup variation 2 (see subsection 5.3.2). All graphs show the controller's average difference in performance compared to *Random* ($SCOUt_H$ average - *Random* average) VS the number of training iterations completed.