Surveillance Coordination and Operations Utility (SCOUt)

Keith August Cissell

June 2018

Abstract

The purpose of this project is to create an artificially intelligent "mind" to make observational decisions for a mobile robot. The mobile robot will have a set of sensors with which it can survey its surrounding environment. It will store this observed data, perform an analysis and plan its next move accordingly. The plan is to drop a robot into unfamiliar environments with a set goal to achieve and let it learn on its own the best way to approach the given situation. Usages can be as simple as mapping out an unknown area, to as difficult as searching for survivors after a natural disaster. The robot will have internal and external limitations it must work with such as battery life and terrain obstacles. The goal is to create AI that can learn how to use its observational skills to achieve a goal within a dynamic environment.

Reword/Expand

0.1 Acknowledgement

Thanks to all my peeps.

Contents

Chapter 1

Introduction

As research in the fields of autonomous systems and robotics have become more extensive, it is evident that there are a wide range of application for robots with integrated autonomy. There are rovers, drones and even aquatic robots that are capable of decision making in their own environments. The tasks that these robots carry out can greatly vary as well. This variance can cause a demand for distinct software and hardware to achieve each robot's given task. However, almost all autonomous robots operate similarly through their use of observation (typically with external sensors) and analytics of the data that is observed.

A great deal of research has been done in hybrid robots and creating hardware that is multifunctional to various tasks However, there is not an extensive amount of research on software with the capability to integrate with multiple robot compositions and tasks. Most of this is due to the fact that each robot has unique capabilities that do not overlap with many other robots. Autonomous robots seem to focus in on a certain niche and require their systems to be built from the ground up each time. This leaves the question of what pieces of autonomous control can be abstracted.

There are many evolutionary computing approaches that can be applied to decision making processes. These methods are commonly used in situations when there are a known number of controllable variables and a wide solution space to be explored. This makes them great candidates for creating a system which drives the decision-making process of autonomous robots. In particular, neural networks and deep neural networks trained in simulations seem to be a promising architecture for finding optimal control patterns in the diverse applications of autonomous robots.

This project approaches the problem from the bottom up. It looks at the very basics of autonomous robotics. This is: the collection of data from sensors, analytics of incoming data, and the output of response controls. Additionally, these three steps are repetitively being performed to achieve a given objective. I have broken this project into three phases. The first phase involves setting up a simulation environment to be used for training the autonomous system. Next, a graphical interphase will be integrated with the simulation data to allow for

easy debugging. Finally, an Artificially Intelligent system will be trained to take in various sets of environmental data as inputs, make decisions based on these inputs and its current objective, and produce a response.

The project is still a work in progress and this paper will only present phases one and two. These phases cover the procedural environment generator and the graphical interface that pairs with it. The implementation of the abstracted autonomous system will come in future work. The main topic covered looks at the representation and formulaic production of data that will be used to represent an environment. The graphical interfaces capabilities will also be touched on. For the AI component, we will look at the various evolutionary methods that hold great potential for our given problem setup.

Chapter 2

Related Work

To begin my process, I read various research papers on autonomous vehicle exploration. The key point I extracted from my readings were the various proposed tasks and situations that were being solved by autonomy. From these proposed tasks, I was able to put together a broad perspective of use cases and begin to draw parallels between them all. My end goal is to create an abstracted process for goal-oriented robots regardless of what their specific tasks were, the environment they were in, and the means in which they gathered their data. I found these were the three cornerstones of goal-oriented robotics: given task, environmental obstacles to handle, and given capabilities. This semester's work lays out the foundation for my research in the possibility and ease for this three dimensional problem to be abstracted.

Reword/Expand

The field of autonomous robotics has recently gained a high amount of attention inside and outside of the research community. As hardware capabilities and intelligent computational techniques have continued to advance, the use of robotics is being introduced to more complex tasks. Robotic agents continue to phase out humans agents for tasks that are considered mundane or dangerous, as well as for performance reasons where a machine can provide better results or adequate results for less cost. Here we will focus on the application of autonomous robots in surveillance based operations. The primary examples of surveillance based operations that we will look at are exploration based scientific research and search and rescue settings.

These two types of 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 that limit the capability of a human agent and sometimes prevent them entirely. In the majority of cases, a robotic agent is less susceptible to the same environmental hazards as a human. As well as objectively increasing agent durability and performance capabilities, use of robotics eliminates the risk of injury, disease and death of any human(s) involved in the operation. The other advantages to using autonomous robots is the diverse amount of sensors that an robotic agent can use, and their ability to analyze data quickly and without bias. Sensors, such as an infrared camera, can

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 much quicker than humans. This supports the idea that a robotic agent can perform at higher levels than a human when performing surveillance based operations.

There is an abundant amount of existing research on the use of autonomous robots for exploration.

[**?**] Multi-robots in hazardous areas; Use Bayesian prediction model to avoid hazards. [**?**] Indoor exploration in unknown environment using a Convolutional Neural Network. [**?**] Combination of autonomous exploration with localization mapping. [**?**] Multi-robot perimeter detection. [**?**] Exploring and mapping unknown environments. [**?**] Example robotics mission that requires exploration in hazardous environments. [**?**] Uses supervised learning for autonomously exploration with efficient user of a single sensor.

[**?**] Use of on board systems to model scientific data and reason path/action planning. [**?**] Exploration and sensor planning for scientific missions.

[**?**] Discusses the use of Deep Reinforcement Learning for "experience-driven autonomous learning", pairing with robotics and the challenges related to the complexity of memory, sampling and computation. [**?**] GA approach to decision tree building for intelligent action pattern building. [**?**] Another GA approach to decision tree building. [**?**] Very similar action reward system for machine learning using actor -> environment -> critique -> reward.

Proposed approach. How it differs from previous research. Thesis statement.

Paper layout?

Chapter 3

SCOUt

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

3.1 Platforms

Several coding languages and libraries are used in this project to provide simple and expandable utility. The utility is laid out in a client-server architecture 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 approach that allows objects to be extended, while keeping a single abstract component to easily maintain the parent object's behavior. This feature is crucial when it comes to long term code maintenance and encasement. For example, an Element is a data structure that defines the types of data that an agent could detect in the environment. If a future project wanted to utilize this tool with an agent that could detect Ultra Violet rays in their environment, they could create an "UltraViolet" class that extends the Element object by defining a set of predefined attributes within the class. The "UltraViolet" class can then effortlessly be integrated with all other pre-existing data structures as it will be handled as the general Element object that it extends. These generalized data structures also provide simple handling for the client portion of the utility. The client portion acts as a Graphical User Interface (GUI) for requesting certain actions to be performed on the server, and visualizing the data structures used on the server.

3.1.1 Simulation Back End

The back end is written in the Scala language. Scala is a Java based, paradigm language that combines object-oriented and functional programming methods. Object-oriented programming provides the flexibility needed to create abstracted data structures, while functional programming provides data immutability while working with large sets of diverse data. All data storage and manipulation takes place on the back end of the platform to ensures data consistency. Data is exported from the back end in two scenarios. The first scenario is when data is saved to a file for long term usage. The second is when data is passed to the client for visualization. In both cases, the data is first encoded into Json data structures. Data is imported to the back end when loading a file or when a request is received from the client. Imported data is expected as a Json object which is immediately decoded and parsed into Scala data structures before usage. The circe Scala library is used for the encoding and decoding of Json data. Circe provides encoding and decoding of Json files, and a seamless integration of Json data into the Scala language. Communication for the SCOUt server uses the library HTTP4s to create a local service to handle Http requests from the front end, and return an Http response.

Reference circe

Reference HTTP4s

3.1.2 Visualization Front End

The platform's front end is built around Electron, a framework that allows building a native desktop application with JavaScript, HTML and CSS. The GUI is written using all three of these languages in addition to SVG, an XML-Based format for vector graphics. HTML structures the page within Electron, CSS provides styling and JavaScript handles all of the logic. A few JavaScript libraries are utilized within the framework to generate visual representation for data and communicate with the back end. The SCOUt platform uses D3, node-fetch and JQuerry. D3 (Data-Driven Documents) is a visualization library that uses HTML, SVG and CSS to create graphical representation for data. Node-fetch is used for Http communication with the back end through XMLHttp request and response handling. JQuerry 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. To manage all of the dependencies between Electron, the languages and their libraries, Node Package Manager (NPM) is used. In addition to dependency management, NPM has packages of its own that simplify the process to compile all of your code into one file for Electron to handle. JavaScript is compiled into a browser friendly format using the Babel package, and the resulting JavaScript is then integrated into the HTML code structure using webpack. Our GUI can then be launched using a single NPM script which will compile all of our code into a single HTML file, launch electron and load the content. The resulting GUI then provides an interactive platform for a user to load, create and visualize data.

Reference Electron

reference Node packages

Reference NPM

3.2 Environment

Representing any environment is a tricky process. A simulation needs to balance simplicity and coverage when modeling an environment. Leave out too much from the model and it won't reflect real word scenarios. Trying to model too much can consume time and effort that could instead be spent running real world experiments. For this experiment, an environment is represented by a single, high level object that contains a few general attributes and a collection of low level objects within it to model a static, Cartesian grid layout of a real world environment. The lower level objects then contain specific details about the contents within the environment.

Each **Environment** created is represented as an n x m 2D grid of uniformly sized s x s square **Cells**, whose size is specified by a scaling factor. Along with positional data, each **Cell** contains information about the different elements and anomalies present within the s x s area it represents. An **Element** is a generalized object that represents one specific environmental attribute such as the elevation or temperature. An **Anomaly** represents some object present within the cell that could be of interest. Anomalies often have an effect on element values in their surrounding area which makes them "traceable".

class Environment name: String height: Int width: Int scale: Double grid: Array[Array[Cell]]

Cell

A Cell holds an x and y coordinate for its position within the Environment's grid attribute. These coordinates do not reflect the actual scaled size of the environment, only an index value for the order they appear within the 2-D array data structure. The scale can easily be applied to the physical location of a Cell as it is a shared global attribute within the Environment object. If an element type or an anomaly is present and falls within the area that the cell covers in the environment, it appears in their respective lists for the cell.

class Cell x: Int y: Int elements: Array[Element] anomalies: Array[Anomaly]

Element

An element is any measurable attributes within an environment. For example, temperature, elevation and decibel levels are all attributes of the environment whose values can be sampled. Different types of elements are all generalized into one abstract trait, Element. Element can be extended into classes that represent specific element types within the environment. The trait has a set of defined attributes that an extended class must define to identify the element type and how it behaves. Name and unit are used for identification and displaying the element type. The value attribute holds a numerical value for the element. This way, when a cell holds an Element, it can store the value for the given position. For example, the Elevation class would store the elevation level of the area

within the cell as an Element of name "Elevation" and the unit would indicate the measurement unit used. The radial flag, lowerBound and upperBound attributes guide and limit the values that can be set for the element type. It is mostly used when generating an environment.

reference environment gen section

trait Element name: String value: Double unit: String radial: Boolean lowerBound: Double upperBound: Double

Anomaly

An anomaly is any object that may be of significance to the robot such as a human or precious mineral. Anomalies have their own effects on elements in the environment around them. The variety of anomalies and the effects they can have are represented as data structures that follow the same extendable trait format as Element. An Anomaly class can have an area that extends multiple cells, but must exist in at least one cell in an environment. Anomalies can also have effects on multiple element types in its surrounding environment, which are in a list attribute. Each Effect class defines a "seed" Element class and a range of the effect. The seed attribute holds a specific instance of an Element class that will represent the value of that element type in the area that the attribute exists within the environment. 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 based upon how close they are to the source of the effect (where the anomaly exists). For example, Human is an anomaly that takes the area of a single cell, and effects Temperature and Decibel values in their environment. If Human is much louder than the static noise level in the environment, you will see a sharp spike in Decibel values in cells nearest the Human, and the increase in value above the environment's static level will diminish the further away you move from the Human.

trait Anomaly name: String area: Double effects: List[Effect]

trait Effect seed: Element range: Double

Layer

One last important data structure is a Layer. While Layers are not direct members of the Environment structure, they are crucial to building and analyzing the Environment. For this reason, Layers are only generated on demand through method calls. A Layer acts in a similar way to an Environment's grid, except it holds a single Element instead of a cell.

class Layer length: Int width: Int layer: Array[Array[Element]]

3.3 Agent Representation

Agents within this experiment have a core set of attributes and abilities, along with a set of sensors and a controller. The core attributes for an agent are health, energy level, a system clock, an internal map and its current position

relative to the internal map. Because SCOUt is focused on purely observational interactions with its environment, an agent only has two categories of actions that can be taken: movement and scanning. The agent can attempt to move one cell at a time in any of the cardinal directions. This allows the agent to reassess after each movement attempt. Scanning collects information about the agent's immediate environment and updates internal map. The list of scan actions that an agent can perform is based on the set of sensors the agent is equipped with. The controller is in charge of analyzing the current state of the agent and deciding the next action to be performed. This project focuses on creating a single controller (SCOUt) that is highly adaptable to wide ranges of agent configurations, environments and goals.

class Agent name: String controller: Controller sensors: List[Sensor] internalMap: Grid[Cell] xPosition: Int yPosition: Int health: Double energyLevel: Double clock: Double

3.3.1 Sensor

Sensors are created from a single trait, similar to Elements and Anomalies. A single instance of a sensor represents a scientific instrument that could be used to gather data measurements for a specific element type. Each sensor defines the element type it is able to measures, the energy and time costs to perform a "scan" action and its effective range. When performing a scan, the sensor will sweep 360 degrees around the agents location and gather data in a circular area. The circular area is calculated with the sensors range as the radius and the agents position as the center. The element type's values discovered will then be added to the agents internal map if it was not previously known.

class Sensor elementType: String range: Double energyExpense: Double runTime: Double

3.3.2 Mobility and Durability

When an agent is created, there are several attributes that can be defined to dictate how an agent will interact with different elements in the environment. The mobility of an agent will determine what level of movement the agent is capable of within an environment, how quickly an agent can move and how much energy is required. For example, if a drone was defined as the agent, it would have higher mobility, but would likely sacrifice the amount of sensors that could be carried. A wheeled robot loaded with multiple sensors would have decreased mobility, but could collect a wider variety of data. Mobility is defined by the maximum slope an agent can climb, the minimum slope an agent can traverse before taking "fall damage", and a resistance factor. The resistance factor ties into durability and scales the amount of "fall damage" that the agent receives.

movementSlopeUpperThreshHold: Double movementSlopeLowerThreshHold: Double movementDamageResistance: Double

Durability factors are also considered. This represents an agent's versatility within an environment and is directly related to specific element types. An

example is an environment with pools of water in it. Most robots would be damaged when emerged in water, but a robot could be designed amphibiously and would therefore be impervious to damage when in contact with water. Like many other data structures seen, these durability factors are defined per element type. Durability is defined by an upper and lower value threshold and a resistance factor. The thresholds define what levels of an element type that the agent can be exposed to before it begins to damage equipment. The resistance factor then influences how much damage the agent will take at levels exceeding the threshold.

damageUpperThreshold: Double damageLowerThreshold: Double damageResistance: Double

3.3.3 Actions

An agent's actions are simplified to either movement or scanning actions. These two categories cover the exploration and research aspects that would be expected from a surveillance agent. Actions are performed in a loop where the controller assesses the agent's state and then decides an appropriate action.

In simulation, movement is handled by changing the agent's current position to an adjacent cell in one of four direction. Movement to an adjacent cell is denoted as "north", "south", "west" or "east" based on the orientation of the x, y grid of cells that make up the environment. Moving a single cell at a time gives the agent the opportunity to reassess its current state before continuing movement. 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. After the attempt has been made, changes to health, energy level and the system clock are calculated and then updated. If the movement action is successfully completed, the current position is also updated.

When a sensor is used, it does a full 360 sweep of surrounding cells in the environment. This creates a search circle with radius equal to the sensor's range and the center located at the agent's current position. For each cell that fall within this search circle, 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 map out its surrounding environment. This map can be then be used by the controller to determine what actions would be most beneficial to the goal at hand.

3.3.4 State Representation

For controllers to intelligently decide when to perform what actions, they need to have sufficient data about the agent and the known surrounding environment. The agent's position, health and energy level can easily be analyzed, but the internal map containing the known environment can become a very large data

structure to analyze each time the controller has to decide upon an action. For this reason the data structure is simplified in order to reduce memory usage and computational effort required to assess a state. What we are left with is a minimal data structure that still contains all of the useful information necessary for a controller to make intelligent decisions. Instead of a 2-D array of cells, the internal map is represented as a list of element states, where each element state is a summary of the data known about a specific element type.

AgentState xPosition: Int yPosition: Int health: Double energyLevel: Double elementStates: List[ElementState]

ElementState elementType: String indicator: Boolean hazard: Boolean percentKnownInSensorRange: Double northQuadrant: QuadrantState southQuadrant: QuadrantState westQuadrant: QuadrantState eastQuadrant: QuadrantState

Element states contain useful information about how the specific element type was being studied during the operation and then technical information about known values. The indicator flag tells the controller whether this element type was being collected in order to progress the goal at hand. If the goal was to map out the elevation, the elevation ElementState would be flagged true. If the goal was to find a human, the Temperature and Decibel ElementStates would be marked true, as their values could help 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, high drops 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 the 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 limited to north, south, west and east, we can collapse known information from the internal map into four quadrants .

Image of quadrants

QuadrantState percentKnown: Double averageValueDifferential: Option[Double] immediateValueDifferential: Option[Double]

The first thing that a QuadrantState looks at is the percent of values that are already known in all the cells within the quadrant. Then, the QuadrantState stores the average and immediate known values into two "Options". These are denoted as Options because there are instances where none of the values within the quadrant are known. Average and immediate values are recorded relative to the value of the current cell. Average differential takes the difference between the current value and the average of all known values in the quadrant's cells. Immediate differential takes the difference between the current value and the cell immediately adjacent to it. So when considering elevation values within the north quadrant, the immediate differential would be the difference between the elevation at the agents current position and the elevation within the cell directly above the current position.

3.3.5 Controller

The design of an agent is created so that as it operates within an environment, it can explore and gather data from the environment while keeping track of its location and internal status. The controller is an autonomous decision making schema for moving the agent and using available sensors to update the internal map. Each action that the agent takes has an effect on its internal state. Moving in the environment requires energy usage and can result in damage (for example falling down a cliff or driving into water). Use of sensors also requires energy and the data gathered will be stored in the internal map. For intelligent controllers, data collected can be used by the controller to decide the next action that should be taken. The specific controllers used are discussed in 3.5.

3.4 Operations

To explore the usefulness and robustness of the intelligent controller, many different scenarios need to be explored. All scenarios are made up of three main components: an agent, a goal and the environment. Different configurations of each component's variables allows us to create a vast variety of scenarios. These three components are chosen and then fed into a simulation process called an operation. Each operation simulates the agent attempting to complete a goal within an environment. Interactions between the agent and its environment are played out and data will is collected to record decisions made by the agent and the outcome of each decision. Data collection takes place on a low level and a high level during each operation.

Operation Diagram

Low level data is collected each time the agent performs an action. - 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

High level data is only collected once at the simulated operation ends. - Status of internal variables (health and energy) - Number of actions taken during operation - Level of goal completion - An overall long term reward - Long term reward given to each action taken

3.4.1 Goals

An operation is run with a given end goal for the agent to achieve. SCOUt is designed for the observational and exploration portion of a task. 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 another process could take over for the actual extraction, or the location could be recorded and another agent could be sent in. Following this, the SCOUt agent could continue to search for more deposits of the mineral or return to base. Using this distinction, we classify goals into two

catagories for testing SCOUt's exploration and observation abilities, anomaly searching and element mapping. Anomaly searching requires an agent to find a given 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 readings and decibel readings to track down the person. Element mapping is fairly strait forward. The agent must map out as much readings of a certain element as possible. This gives SCOUt the opportunity to discover correlations between the element is searching for and other elements present in the environment. For example, you are move likely to find water in a valley rather than a hill.

3.4.2 Rewards

In addition to goal completion, an agent also has to be observant of its health and energy. The more efficiently an agent can complete an objective the better. The overall performance of an agent is measured by both its ability to complete the task at hand and the efficiency of the actions taken. These performance measurements are calculated on a short and long term basis, and come in the form of "rewards". In addition to measuring the performance of an agent, rewards can be used as a learning metric for an intelligent agent. When an intelligent agent finds itself in a similar state as before, it can look at the action it took in the past and the reward that was given to decide if it should perform that action again.

Short Term Rewards

Add STS Equation

Short term rewards are given each time an agent performs an action to reflect the immediate outcome of the action. 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 fail entirely to move (a hill was too steep to climb) a deduction is made. A small increase in reward is also applied if the agent moves into an unexplored area. If the agent uses a scanner, the reward is adjusted to reflect the amount of new data learned. This penalizes the agent from using a scanner twice in a row or after small movements as it is not efficient use of energy.

Long Term Rewards

Add LTS Equation

Long term rewards are calculated once the operation is over. This could mean that the agent has successfully completed its goal, or it is depleted of health or energy. To reflect these scenarios, the 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 .

LTS reward distribution equation

3.5 Controllers

Experiments in this paper compare three different types of controllers: random, heuristic and intelligent. The random controller will simply select a valid action at random until it is no longer operational, or it has completed the goal. Heuristic controllers follow a set of logical steps to decide which action is taken. Each heuristic controller has to be created specifically for the task at hand. These controllers are expected to perform at the same rate of success for each operation as they have no learning qualities about them. A single intelligent controller is created to study how it can operate across multiple environments and achieve multiple types of goals. To test the usefulness of the intelligent controller, its performance is compared against the heuristic controllers and the random controller.

3.5.1 Heuristic Controllers

Talk about Random and Heuristic Controllers.

3.5.2 Intelligent Controller

The SCOUt controller operates using memory based reinforcement learning. After each operation, the SCOUt controller will store state-action pairs into its memory for later reference. A state-action pair (SAP) contains the action that the agent took, the state that the agent was in when it decided to take this action, and the short and long term rewards that the action received. In future operations, the SCOUt controller will try to find state-action pairs where the SAP's state is similar to the agent's current state. Next, the controller will look at what action was performed and what rewards were given for the outcome, and use this to predict the best action it should perform in its current operation.

Memory

Memory is built from running simulated operations. Each time the SCOUt controller finishes an operation and attributes short and long term rewards to each action, it selects which actions to store in its memory. For experiments in this paper, the last 20 actions performed are saved, and then a uniform sampling of 5 percent of all actions early in the operation are stored. The entire memory set is not saved to cut back on memory loads and computational time when searching for similar states. The last 20 actions are always saved because they typically are the most important events leading up to success or failure of the goal at hand. The remaining actions taken prior to these last 20 are then

uniformly sampled so that the memory will also contain information related to the agent's initial searching behavior. Each action is then stored as a stat-action pair in a Json file for future use.

State Comparisons

When a SCOUt controller begins an operation, it will first load in state-action pairs from a specified memory file. The states within the existing memory is then normalized to make variances within each state's data relative to all other states in the memory set. For each state, normalization takes place for all numerical values stored within the AgentState class. This includes:

- health - energyLevel - each ElementState's: - percentKnownInSensorRange - each QuadrantState's - percentKnown - averageValueDifferential - immediateValueDifferential

format list

Normalization follows this Guassian distribution equation suggested by **??**.

normalization eq

This makes data values that are more meaningful when studied by the agent. For example, if a controller 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 in decibel readings is accurate, it needs to have an idea of what is normal variation in decibel values versus a significant increase. Gaussian distribution provides this functionality through the calculated average and standard deviation within a data set. If the agent has gathered decibel readings in its north quadrant that are well above the standard deviation of readings it has stored in its memory, it should be considered significant.

Once the internal memory has been loaded and normalized, the controller can use the states of SAPs to compare against the agent's current state and predict the best action. Each time the controller is used to decide upon an action, it will compare its current state to states within its memory. State comparisons are calculated using the following algorithm: Comparison follows this algorithm:

style algo

1. Normalize the current state (how many STDs it falls outside of the average) 2. Calculate the difference for: a. health b. energyLevel c. elementStateDifferences = for each element state: i. hazardDifference = if (current == SAP) 1 else 0 ii. indicatorDifference = if (current == SAP) 1 else 0 iii. percentKnownInSensorRangeDifference = abs(SAP - current) iv. immediateValuesKnownDifference = abs(SAP - current) / 4 d. quadrantToQuadrantStateDifferences = for each current quadrant: i. quadrantStateDifferences = for each SAP quadrant: a. quadrantElementStateDifferences = for each element type: i. hazardDifference = if (current == SAP) 1 else 0 ii. indicatorDifference = if (current == SAP) 1 else 0 iii. percentKnownDifference = abs(SAP - current) iv. averageValueDifferentialDifference = if (current known SAP known) abs(SAP - current) else (if current known == if SAP knonw) 1 else 0) v. immediateValueDifferentialDifference = if (current known SAP known) abs(SAP - current) else (if current known == if SAP knonw) 1 else 0)

The state comparison algorithm generates difference calculations with movement and scanning action types in mind. The elementStateDifferences (item

c) are used when comparing the current state againts and SAP who's action was a scan action. For each elementStateDifference, the algorithm compares the similarity between how the elemet type is being studied (for hazard detection and goal related inication), as well as the percent of the element type known in range of the sensor and how many of the four adjacent cell's values are known. The hazard and indicator differences guide the controller to determine the importance and usage of the element types data. The percent known and immediate values known difference guide the controller to decide whether usage 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 moving into one of those cells would result in taking fall damage. The quadrantStateDiffernces (item d) are used when the SAP's action was a movement action. Each quadrant for the current state is compared against every quadrant in the SAP's state. In doing so, this allows the controller to consider all orientations between the two states. Each orientation is important to consider because SOMETHING SOMETHING WORDS. Within each quadrant to quadrant comparison, the controller will consider the differences between values for each element type. These quadrantElementStateDifferences look at values related to the specific quadrant instead of the entire internal map as elementStateDifference compares.

orientation diagram

Once these specific differences have been calculated, the controller must collapse them into a single, overall state difference to help predict the reward the agent will receive for each possible actions. To collapse the collection of state differences, the controller must apply different weights to each individual difference, and average the total of them all. There are two differnt calculations for overall state differences. The first is if the SAP's action was a scanning action, and the second is if is was a movement action. The equations for this weight-based difference calculation are as follows:

equationzzz

overallStateDifference(scanning) = (health * Whealth + energyLevel * Wenergylevel + overallElementStateDifferences * WelementStates) / 3

overallElementStateDifferences = sum(for each elementStateDifference: overallElementStateDifference * WelementState) / number of elementStateDifferences

overallElementStateDifference = (hazardDifference * Whazard + indicatorDifference * Windicator + percentKnownInSensorRangeDifference * WpercentKnownInSensorRange + immediateValuesKnownDifference * WimmediateValuesKnonw) / 4

overallStateDifference(movement) = (health * Whealth + energyLevel * Wenergylevel + overallQuadrantDifferences * Wquadrants) / 3

overallQuadrantDifferences = min(for each orientation: overallOrientationDifference)

overallOrientationDifference = (overallQuadrantDifference1 * Wquadrant + overallQuadrantDifference2 * Wquadrant + overallQuadrantDifference3 * Wquadrant + overallQuadrantDifference4 * Wquadrant) / 4

overallQuadrantDifference = sum(for each quadrantElementStateDifference: overallQuadrantElementStateDifference * WquadrantState) / number of quad-

rantElementStateDifferences

overallQuadrantElementStateDifference = (hazardDifference * Whazard + indicatorDifference * Windicator + percentKnownDifference * WpercentKnown + averageValueDifferentialDifference * WaverageValue + immediateValueDifferentialDifference * WimmediateValue) / 5

Action Selection

3.6 Environment Generation

The SCOUt environment build tool captures important details that are necessary for agent-environment interactions to be simulated, while remaining simple to implement and understand. The tool is also highly abstracted so that more details can easily be added as needed while still maintaining a defined build process.

Build Process

1. User Fills out dynamically generated template.
2. Builder initializes a grid of empty cells
3. Element seeds are used to populate each present element type into the gird of cells
4. Terrain modifications 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

separate diagram?

Environment generation is guided by an environment template that holds the necessary data to build a specific environment. Use of a dynamic template to allow representation of a wide range of environments while allowing re-creation of multiple random environments based on the same template. The goal of each template is to provide influence over the values generated so that a specific environment can be modeled with minimal input.

environment template table

Element seeds require input variables to be provided in order to drive their population process within the environment.

The process to generate each environment is divided into two main phases: terrain formation and layer initialization, anomaly placement and effect radiation.

3.6.1 Element Seeds

Once I had a way to represent an environment, I developed a way to generate an environment on demand. Because an environment will be able to hold very large amounts of data, I needed way to procedurally generate an environment. Because SCOUt's AI will be training on these environments, the procedural

Reword/Integrate

generation needed to produce unique environments each time. The solution that I came up with was to build an environment one Layer of Elements at a time using Seed objects. Each Element object has a companion object called a Seed that holds parameters used to produce a Layer of its Element. The generation of each Layer is modeled on how the element may vary in a real-world scenario. Parameters within each Seed are set to a default value but can also be passed in to change how the Layer may vary throughout. An environment generator function is passed a list of these seeds along with the length and width of the 2D environment's grid. The environment generator then passes each seed to the appropriate layer generator function to create each Element layer For an example, let's look at the Seed for producing the Elevation Layer.

Class ElevationSeed elementName: String = "Elevation", dynamic: Boolean = false, average: Double = 0.0, deviation: Double = 0.15 * scale

Note: 'scale' is a global value used by all Seeds to maintain consistency.

When the environment generator finds an Elevation Seed, it passes it to the elevation layer generator. This particular generator first creates an Elevation object at Layers the (0,0) Cartesian position set to the average value held in the seed. For each proceeding, Cartesian position, the generator creates a new Elevation object and uses an algorithm to determine what value it is set to. The elevation layer's algorithm looks at the neighboring cluster of Elevation objects and calculates the mean elevation value. The new Elevation object is then set to a random value within deviation of its neighboring object's mean value. Once every Cartesian position has been given an Elevation object with an assigned value, it is returned to the environment generator. The environment generator then takes the Elevation object in the Layer, matches it to the Cell in the corresponding position in the Environment's grid and adds the Elevation object to the Cell's elements list. Most layer generators follow this same pattern.

1. Receive a Seed 2. Initialize a few positions 3. Set all remaining positions using an algorithm that looks at the initialized positions 4. Returns the Layer to be injected into the Environment's grid

Some layers are far easier to generate than others. Latitude and Longitude layers can simply be generated by linearly increasing or decreasing the values as they distance from the initialized value. Layers such as sound Decibel are much trickier. A Decibel layer requires a noise source to first be given a value, then the values of all other positions must take sound dampening into account in all directions.

Phase one is the most intensive process that relies on the element seed data and terrain modifications provided by the user to influence the procedural generation of the base environment. Following a similar process laid out by Doran and Parberry**??**, desired traits are incorporated into the environment while allowing unique formations 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 generated on top of the structural/terrain layers (Elevation and Water Depth). To begin, the element seed for Elevation is used to create a base layer which we can begin to build upon. Next terrain

modifications are applied one after the other, taking care not to overlap modifications (for example, we wouldn't want a hill to overlap with a valley and cancel each other out). Each terrain modification picks a random, unmodified cell in the environment to begin at, and updates the value of the given element type. The modifier then performs "walks" to random, unmodified neighboring cells, updating their values. These walks continue until the user specified coverage size has been met, or until there are no neighboring cells that can be modified. Smoothing is then applied with a sloping factor to the modified cells and immediate surrounding cells to reduce rigidity and give a more natural change in values between neighboring cells. Finally, once all terrain modifications have been applied, layers of all non structural element types (examples: temperature and decibel levels) are generated from their corresponding element seeds, and then added to the environment.

Phase two then places anomalies into the environment. Each user specified anomaly is randomly placed into cell(s). For anomalies that occupy more than one cell, neighboring cells are chosen at random until the anomaly coverage area is met, or there are no neighboring cells that can contain the environment. Each cell containing an anomaly is updated so that the given anomaly appears in the ${}_{a}nomalies_{l}istforreference.Afteranomalyhasbeenplaced, eachoftheanomalieseffectsareappliedtoupdatee$

3.7 Visualization Tool

The environment build tool provides a Graphical User Interface (GUI) for creating and visualizing environments. Electron is used to simulate 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. Scala library http4s is used to create a server on a localhost network for handling the http requests from the front end. This architecture allows all data creation and manipulation to be isolated in Scala on the back end, while allowing user interactions with the data to take place on the front end. Launching the app starts up the Scala server in a new terminal and opens the Electron window which will begin attempts to establish communication with the server.

Possibly cite Electron

Possibly cite http4s

3.7.1 Home Page

Once connection between the server and GUI have been established the user 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 only inputs the name and size of the environment and all other variables are selected by the server. Building a custom environment steps the user through a series of form pages to create an environment template. Loading an environment allows the user to select a previously built environment or a saved template to use.

More on operations later

Reword

Method Path Response GET /ping Responds “pong” GET /current${}_{s}tateReturnsJsonofEnvironment'scurr$

Format Server Table

Visualization is currently very minimal and does not reflect the final implementation. Once an Environment has been created, an interactive visualization will be loaded. Each Layer of Elements can be viewed one at a time. The current Layer's name will be displayed and there will be forward and backward buttons to allow the user to flip through the different layers. Each layer is currently only displayed as a heatmap. D3's Heatmap open source project is currently used to take Json data of the layer and create a visualization based on the value at each Cartesian point.

I spent the majority of the semester working on the visualizer portion of the GUI. I had already worked with D3 libraries within JavaScript to make basic contour maps. My new goal was to create a way to logically display the different layers of elements within the environment using these libraries. The visualizer also needed to incorporate a way for the user to control which layer(s) were being displayed, and present proper information corresponding to the environment. To do so, I created a "Visualizer" page to my application. This page is divided into four main sections: display, toolbar, legend, and navigation bar. The display is an SVG element that holds graphing data for the environment. It will display specified layers of the environment as well as a grid representing each "cell" within the environment. The toolbar allows the user to select the current element layer(s) that are displayed. The legend displays information on the environment in three areas: general information, information for the currently selected layer and information of a single cell in the grid. A cell's information will be displayed within the legend when selected. A cell can be selected by clicking its corresponding position in the SVG display. The selected cell will also be highlighted within the display when selected. The navigation bar will be populated with controls for the robot once it is incorporated into the application in my future work. Currently, the only item in the navigation bar is a button to return the user to the "Builder" page which allows the creation of an environment. The "Builder" page took the remaining time of the semester. It is the starting point of my application. When the app is launched, a user will be presented a screen with a few basic form fields to build an environment and two options: "Generate Random Environment" or "Build Custom Environment". The basic fields will allow the user to enter the name and dimensions of the environment to be created. These fields are required regardless of the user's build option. If the random environment build option is selected, a request will be made to the SCOUt server to create and return an environment based on the default environment parameters and will contain layers of every optional element type. If the custom environment option is selected, the user will then walk through several pages of forms to provide the "seed" data for each element layer. The first form will ask the user which elements it would like to include. Some elements (environment, latitude and longitude) are required in all environments. For each layer that the user has selected, they will then be given a form page that asks for "seed" data for each of the element types. 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 while moving through the different form pages, as well as go back and edit the data if they select "New Environment" from the Visualizer page. Form data

Reword

is also set within required bounds and checked before submission. Once the user has filled out all required form info, they can review their data and then submit it. When submitted, the front end data is gathered and sent in a request to the SCOUt server. An environment is then created with these specs and returned to the front end to be loaded into the Visualizer.

3.7.2 Visualization Page

Display

Navigation Bar

Tool Bar

Legend

Operation Tool Bar

Chapter 4

Results

What did the results yield and what can we infer from this

4.1 Experimentation

Trial setups and data interpretation

Chapter 5

Conclusion

Conclude stuff

Chapter 6

Future Work

How to improve this approach.

Sensors can scan in an arc rather than 360 degrees.

Better Memory selection/ trimming.