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

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

Chapter 3

Methodology

The project's simulation consists of three main components: environments, agents and the interactions between the two. A wide range of diversity is required for both the environments and agents represented in simulation.

After finalizing my project idea, I began back end development on the environment's data structure. I chose to use Scala as my programming language for storage and manipulation of data on the back end. Scala is a multiparadigm language that combines both functional and object oriented programming styles. This made it ideal for the back end as I can limit the mutability of my data structures using functional approaches, while still taking advantage of abstraction used in OO programming styles. The flexibility of Scala also plays a role in creating a concurrent form of communication to the GUI, but more on that later. Reword/Integrate

3.1 Environments

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 used to run real world experiments. 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. separate diagram?
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

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

Each **Environment** created is represented as an n x m 2D grid of uniformly sized s x s square **Cell**, whose size is specified by a scaling factor. Along with positional data, each **Cell** contains information about the different elements and anomalies present within their s x s area. 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".

To start, I came up with a simplistic design for representing dynamic environments and broke it down into its core pieces using an object oriented approach. The main structure is the Environment object which contains a 2D rectangular, Cartesian grid. The x and y coordinates of the grid act as the latitude and longitude coordinate of evenly spaced points in the Environment. The 2D data grid holds a Cell object at each of these points. The Cell is in charge of holding their x and y position and the values of various environmental elements at this position in the Environment. Each of these environmental elements are stored in their own specific object that inherits from an Element Trait. Traits are a Scala specific structure that acts as a parent object that can be extended from, without the ability for it to be implemented by itself. Temperature, Elevation, Longitude and Latitude are examples of objects that extend the Element trait. This extension approach allows each of the children to hold values specific to the environmental element they represent. Together, all of these objects form a representation of an environment and its various elemental features. Here is an overview of each of these data structures.

Reword/Integrate

class Environment name: String grid: Array[Array[Cell]]

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

trait Element name: String value: Double unit: String constant: Boolean

. . .

One last important object that I created was a Layer. A Layer acts in a similar way to an Environment's grid, except it holds a single Element instead of a cell. This data structure will mainly be used for visualization and analytical analysis once the AI is implemented. For this reason, Layers are only generated on demand through method calls.

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

Element and Anomaly are abstract data structures that each provide a basic object that is then extended to create specific instances. They all share core members that allow handling to be generalized when being manipulated or interacted with. Each instance also has an associated seeding process to automate their population within the environment. Seeding processes are specific to each instance of an Element or Anomaly.

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

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 the environment around them and this then leads to phase three.

3.1.1 Environment Generation

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

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 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.

Reword/Integrate

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 $_{a}nomalies_{l}istforreference.Afterananomalyhasbeenplaced, eachoftheanomalieseffectsareappliedtoupdatee$

3.1.2 Environment Build 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

Possibly cite Electron

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 http4s

Once connection between the server and GUI have been established the user can choose to generate a random environment or build a custom environment. 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.

Load environment or template from file

2.1.3 SCOUt Server The SCOUt Server was created with the assistance of HTTP4S, a Scala open source library. HTTP4S allows easy setup and use of an HTTP request handler. For my project, the server runs on localhost:8080. The localhost is then accessible by any applications running locally on the computer via HTTP requests. A service can then be setup on the back-end to handle specific incoming requests. If a request matches the proper path and request method type, the service calls Scala functions to complete the requested action, and then returns an HTTP response. At this time, the service only handles a few basic requests.

reword/integrate

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

2.1.4 Data Communication The SCOUt server must both receive parameters as well as send data structures. To allow easy communication between of data between multiple languages, the server expects to receive JSON data in request bodies and sends JSON data in its response bodies. In cases that parameters are required to be passed in with a request, the JSON must properly be constructed for SCOUt's service to handle it.

2.2 GUI (Front-End) 2.2.1 GUI Structure This GUI is handled with JavaScript, HTML and CSS. In order to encapsulate SCOUt into its own application, I used Electron which emulates a browser window as a standalone desktop application. All of these components are handled together using Node Package Manager (NPM). NPM allows simple commands to be run in a terminal to launch the entire SCOUt application, or just specific parts. JavaScript handles HTTP communication with the SCOUt server, as well as all data storage, HTML manipulation and user input handling for the GUI.

reword/integrate

2.2.2 Communication JavaScript uses the “fetch” command to make HTTP requests to SCOUt's local server and handle the responses.

2.2.3 Environment Data Structure When data is received from the SCOUt server, the JSON is parse and a data structure is created on the JavaScript side. The data structures strongly resemble the same exact layout as the Scala data structures. The only difference is the restriction to edit data. The only time that the data on this front-end will be edited, is when handling an HTTP response that confirms a change was made to the data structure on the back-end.

2.2.4 Environment Visualization 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.

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 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. Reword

3.1.3 Visualizer

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 Reword 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.

3.2 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. 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.

An agent is considered operational as long as it has remaining health and energy. Health is effected by.... Energy is effected by....

3.2.1 Movement

An agent's movement is simplified to stepping from its current cell to one of its four neighboring cells. Movement to a neighboring 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 scale to the size of the cells within the environment representation. 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.

3.2.2 Sensors

A sensor is an object that allows an agent to gather data about one specific Element Type within an environment. Each sensor has set attributes for the element type that it can detect, the range, the energy required to run and the time required to run. Additionally two flags are passed to a sensor when it is first setup. One of these flags denote if the sensor is being used to detect "hazard" elements in the environment. Whether an element is hazardous typically depends on the setup of the agent. For example, water could be considered hazardous to most robots, however if the agent was designed for aqueous missions, water could no longer be a concern. The other flag denotes if the sensor is being used to detect "indicator" elements in the environment. Indicator elements are any element that are associated to the agent's goal. An example is if the agent is searching for a human within the environment, their heat signature might be significantly different than the surrounding environment, so temperature would be considered an indicator.

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.

3.2.3 State Representation

For controllers to intelligently select when to perform what actions, they need to have sufficient data about the agent and the known surrounding environment. The agent's health and energy level can easily be stored, but the internal map containing the known environment can become a very large data structure to work with. For this reason the data structure is stripped down in order to reduce memory usage and computational effort required to analyze a state. Instead of a 2-D array of cells, the internal map is represented as a list of element states for each element type that there is a sensor present for.

reference cell structure diagram?

Each element state is comprised of the value at the agents current position (if known), a flag for whether the element type is considered a hazard element, a flag for whether the element type is considered an indicator element, and then four quadrant states for each cardinal direction. Because agent movement is split into north, south, west and east, we can collapse information about the element type into four quadrants . A quadrant state contains the percent of cells where the value for the element type is known, the neighboring cell's value (if known) and the average of all known cell values in the quadrant. These metrics allow us to generate useful information that ties the current state with consequences that may come from each action. Particularly...

Image of quadrants

add state comparison equations?

talk about normalization

3.2.4 Controllers

In this experiment three different controllers are created and compared for their performance.

3.2.5 Goals

3.3 Simulation

To explore the usefulness and robustness of our intelligent controllers, 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 where interactions between them will be played out and data will be collected. Data collection takes place on a low level and a high level during each simulation.

Simulation Process Diagram

Low level data is collected each time the agent takes action throughout the main loop of the simulation process.

High level data is only collected once at the end state of the simulation process.

Chapter 4

Experimentation

Trial setups and data interpretation

Chapter 5

Results

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

Chapter 6

Conclusion

Conclude stuff