Simplified Rover Simulation using JAM Agent

Rithin Chalumuri

1 Introduction

This paper presents a simulation of a simplified rover whose primary goal is to explore the rocks placed in a an two dimensional grid. The rover has a number of capabilities i.e. moving around the area, analyzing a rock to check if it has traces of water and taking it back to the base if it does. The problem can be expressed as follows; given a $m \times n$ grid with total area α and r rocks placed in certain coordinates, we need to ensure the rover travels to the locations where the rocks are, analyzes them and brings them back to base (x_0,y_0) if they have traces of water w. The rover is controlled by an Intelligent Agent to direct the movements and actions. This agent was developed using JAM Intelligent Agent Framework and several tests have been conducted to evaluate it's performance in different conditions. All the source code and the instructions video to run the simulations can be found in the **GitHub** repository link below.

1.1 GitHub Repository

https://github.com/rithinch/Rover-JAM-Agent-Simulation

2 Design Specification

The overall simulation system is composed of three components; Dynamic World Model Generation (Facts), JAM Agent and Simulation Visualization. The entire system can be run solely using the **run.py** python script; *main()* method handles the execution of all components.

2.1 Dynamic World Model Generation

This component allows the user to dynamically generate different world models file that the JAM agent can later be run with. Both user-entered input and random generated samples options are supported. When entering the data the user can specify state variables such as base position, starting position and rock locations. The generated model data file is stored in the **experiments** folder with the filename of format; **facts_n_m_n.txt**. This component is used to setup different conditions for running experiments and evaluating the agent behaviour in those conditions. The following methods in the **run.py** file make up this component; *getInput(), genRandom(), CreateJAMFactsFile().*

2.2 JAM Agent

This component allows the user to run the jam agent with the specified world model. The *runJAMAgent(facts_filename)* method in **run.py** file takes in the the dynamically generated world model filename and invokes **mars_rover_agent.jam** with correct jam.jar java package distribution in the command prompt. The jam agent logs each action to the command prompt stdout. Once agent finishes, this stdout stream is then returned by the method for further use in the overall system and a file with same world model filename + '_output' is created in the same directory where the world model text files exists. The agent has been designed in a way that it works for different rock locations and different grid sizes (including negative axis) i.e entire 2d space. The core JAM agent design includes the following facts, goals and plans:

2.2.1 Facts

These are passed in from the facts file and not directly included in the agent code, this is so that we can run same agent code for multiple world models. Currently the agent interacts with the following facts during execution:

- **Base Location** - this store the coordinates of the rover's base location in the grid.
- **Number of Rocks** - this tells the agent how many rocks there are in the world model so that it can go to analyze them.

Table 1: Simulation Results

Scenario	Rocks	Grid Size	Goals Established	APL's Generated	Time Taken (s)
1	10	10×10	226	1385	0.076
2	25	50×50	1642	6779	0.171
3	50	100×100	8470	50976	3.891
4	100	500×500	70914	284474	6.389
5	500	1000×1000	645642	2586562	163.416

- **Rocks** - this stores information about the coordinates of the the rocks and it's features.
- **Rover Position** - this stores information about the rover's current position coordinates as the rover is moved around the area α.

2.2.2 Goals and Plans

There are two types of goals set; top-level goals and sub-goals. Each goal also has a plan included i.e the steps the agents needs to take in order to successfully achieve the goal. The following are the details:

- **Explore Rocks** - this is our top-level goal telling the agent to explore all the rocks from the world model.
- **Go to Position** - this a sub-goal in top-level goal and is achieved when the agent successfully goes from position a to b. It has further subgoals such as moving forward, backward, left and right.
- **Analyze Rock** - this goal is used when it finds a rock and then it looks to for traces of water.
- **Pick Up Rock** - this goal is used when the agent finds a rock with water, it removes the rock from that location and marks the rock as found.

2.3 Simulation Visualization

In order to be able to monitor the agent better, a visualization component has been added. This component is written in python and uses *matplotlib* package for plotting the grid, rock locations and rover movements. The idea is that, once the agent finishes running the outputs from the stdout is parsed into appropriate data structures that we can further use for plotting. The methods *parseJAMOutput* and *vizualize* in **run.py** are responsible for handling the parsing and visualizing of the agent output. Moreover, there is also a *replay* method, which can take in the agent output file and run the visualization of that particular simulation. It is helpful when certain simulations take too long to run from scratch, with this in place we can just run it once and visualize it multiple time to evaluate the agent behaviour.

3 Experiments and Results

Several experiments were conducted to evaluate the agent performance. Three factors were kept in mind when designing experiments; how the agent performance is when area size largely scales, does the agent carry the tasks successfully as planned, and how the agent behaves in edge cases example when the rock coordinates are in negative quadrants. It is noted that agent behaves correctly in all the tested scenario's and is successfully able to cover entire 2d space. The results from the experiments are shown in **Table 1**. We can see that as the scale increases the number of applicable plan life cycles (APL) and goals established exponentially increase, and the time to operate has also largely increased. The generated world model data for all the experiments is placed in the **experiments** folder so in future when advanced algorithm's are implemented we can fairly compare both old and new agents' behaviour. Having a visualization of the simulation has helped ease the testing since the behaviour can easily be observed as opposed to text based outputs. Although the algorithm implemented in this agent is much faster than check each coordinate for rocks algorithm, there is still plenty of scope for optimization such that it performs efficiently in large scale situations. It would also be interesting to implement multiple agents to exploring the grid to find the rocks since that can also cover the area quicker.