Casey Easter, Justin Roberts

Dr. Pears - CSCE 5210 - Project 1

9/21/21

INTRO: Throughout the creation of this program, we were able to gain a detailed understanding of how a simple reflex agent can interact with a fixed environment. With the use of different data structures and actuators, we were able to create a functioning agent that can, with a good degree of accuracy locate and retrieve orders within its defined "warehouse". The written program also keeps track of performance measures like most efficient and least efficient paths taken when fulfilling orders.

APPROACH: The code structure is as follows: *main.py* imports *robot_class.py* and *project_utils.py* files. The *robot_class.py* file imports *project_utils.py* and *warehouse.py*. The *project_utils.py* imports *warehouse.py*. The libraries Random and NumPy are used by *robot_class.py* and *warehouse.py*. A library called PyGame is used by *project_utils.py* for visualization. The *main.py* code initializes a Robot class object with given parameters. It takes an integer value for the number of episodes to run, a protocol (random or path), and a layout (a or b). It then starts the search procedure. At the end, it prints out the report and displays the visualizations.

The *robot_class.py* code is where the bulk of the work is done. It starts by initializing all tracked values within the **__init__** method. These are organized in the code by "topic." The **start_procedure** method calls **new_episode** the number of times passed in via the "episodes" parameter. The **new_episode** method sets all values for a new episode, initializes the warehouse layout, and starts the movement protocols. The **movement_protocol** method, like many others that will be discussed, behaves differently depending upon which value was passed in for the "protocol" parameter. In this case, it sends the necessary information to the **search_pattern** method. For the sake of brevity, the "path" protocol – being somewhat unrelated to the explicit directions of the homework prompt – will only be discussed as an interesting observation at the end of this report. Only behaviors corresponding to the "random" protocol will be discussed in this section.

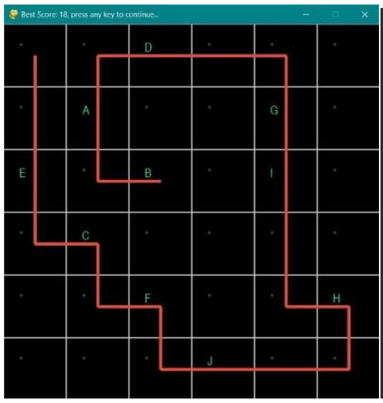
The **search_pattern** method behaves thusly for the "random" protocol: The robot "looks around" by calling the **look_around** method. Then it checks its orders with the **check_orders** method. From that, it can tell if it is next to a potential target. If it is, it moves there and gets the item (unless it was a false positive, in which case it moves there but does not get an item). If there is not a potential target, it chooses a pseudo-random direction to move by calling the **random_direction** method. The **search_pattern** method continues this loop until the robot's **check_complete** method returns True, after which it calls the **finish_episode** method.

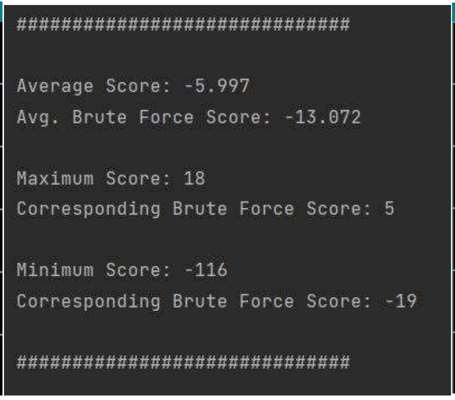
There are a variety of methods for environment navigation. The **move** method adds a directional increment to the current position of the robot, appends that move to the current path, and scores the move based on the prompt criteria. The **random_direction** method ignores invalid moves and prioritizes unvisited locations for its random selection. The **look_around** method gets the values of the cells adjacent to the robot, ignores walls, and checks for errors, overwriting readings as necessary for the given error. The **check_orders** method handles both false positive and true positive movement decisions. The **retrieve** method is only applicable to the "path" protocol.

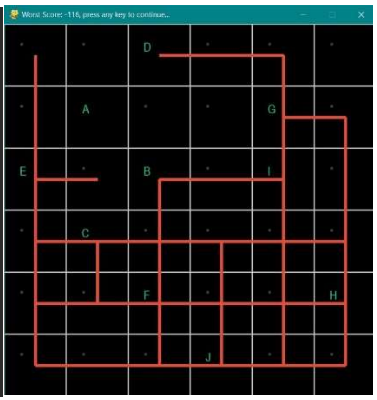
The **check_complete** method checks to see if all orders have been found. The **finish_episode** method calculates and stores average scores, baseline scores, max, and min scores, as well as stores the paths corresponding to the best and worst runs. These results are displayed with functions from *project_utils.py*. The *warehouse.py* file handles warehouse layout generation and order randomization.

RESULTS

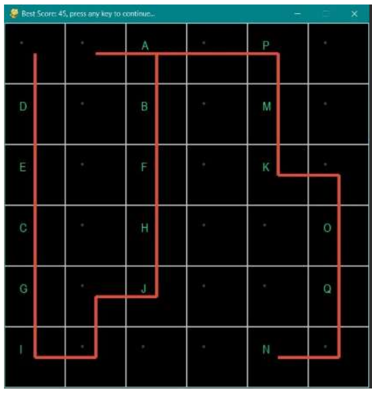
Random Protocol, Layout 'a': *(shown below in order: Best Path, Report, Worst Path)*

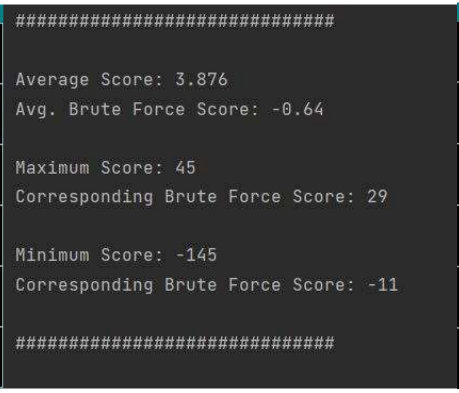

Best Score: 18, press any key to continue...

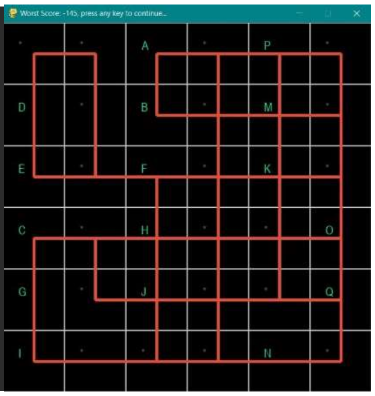

```
##############################

Average Score: -5.997
Avg. Brute Force Score: -13.072

Maximum Score: 18
Corresponding Brute Force Score: 5

Minimum Score: -116
Corresponding Brute Force Score: -19

##############################
```


Worst Score: -116, press any key to continue...

Random Protocol, Layout 'b': *(shown below in order: Best Path, Report, Worst Path)*


Best Score: 45, press any key to continue...


```
##############################

Average Score: 3.876
Avg. Brute Force Score: -0.64

Maximum Score: 45
Corresponding Brute Force Score: 29

Minimum Score: -145
Corresponding Brute Force Score: -11

##############################
```


Worst Score: -145, press any key to continue...

We feel that it is likely that the scores vary primarily due to the number of potential orders. Orders with higher counts of items result in a higher scoring potential (see brute force score bump in layout 'b'). The "random" protocol has a comparable minimum score for each layout, since that is usually driven by the robot getting "lost," but the top score and average score receive that bump as well from more frequent "advantageous" order lists. For data structures, we utilized the Robot class to remember information pertaining to its progress and coordinate dictionaries for the warehouse functions to generate environments passed into the Robot class.

CONCLUSION: A program like this has a broad array of applications in the real the world and with some modifications, can be made to be as specific as needed. Even with minimal guidance, the simple reflex agent more efficiently found orders than a program running a brute force method. So, with integration of advanced search strategies, even greater efficiency will be seen.