**CMPT 310. Winter 2025, Assignment #1**

**Introduction:**

The goal in this project is to practice design and implementation of the common search algorithms used in planning for intelligent agents in a fully visible environment. The environment is a cleaning robot which moved from room to room in a grid and clean the rooms/cells if dirty. Dirty cells are specified with grey color. There are wall cells which restrict the free movement, and the robot must find its ways around the wall blocks. Each move from a cell to its neighboring cell costs 1.

The robot finds its way around the grid by pre-planning its path to the next dirty room. For planning the robot has access to 3 uninformed (DFS, BFS, UCS) and 2 informed (Greedy, A\*) search algorithms. There is also a reflex agent style planning in which robot moves around in a random move, if no neighboring dirty room exists, otherwise moves to the dirty room and clean it.

For each path to the next dirty room, the ***explored*** and *path* cells are displayer with orange and pink colors. There are 2 counters at the top of the GUI that display the accumulated *explored count* and the total *path count* so far. Both these numbers are accumulative.

This assignment is provided in the form of a shell module, and you are going to fill in the specified missing parts. These parts currently have print messages saying: "For students to implement". Once you have done a part, remove the printed message. The GUI and mechanics of the application should work fine as it is, and the above functionalities work only after you implement the corresponding algorithms.

The scripts in this project contain lots of comments and instruction. Read them carefully. They are part of the assignment description.

The way implementation works is that the *path* and *explored* list are computed when you select one of the search algorithms from the menu and click on **run** button.

The agent starts from the middle of the grid. The next search is performed automatically for the next dirty room and so on, till all grid is clean.

**Cost**:

The basic *cost* function is the ***Step*** count, which is the number of step (moving from a cell to a neighboring cell). There is a cost dropdown menu to choose other *costing* options. Available options are: ***StepTurn***, ***StayLeft***, ***StayUp***. The *StepTurn* charges extra for each turn using a formula described below.

A turn of 90’ should costs 3 compare to cost of 1 for each step. For example for currently heading north state, a 90 turn clockwise to head east and then move to the right cell , the cost should be 3 + 1=4. This should result in the agent preferring the path with smaller number of turns.

The **StayLeft** should be designed such that encourages the agent to clean the left half of the grid cells first before moving to the right half. You must come up with a simple cost function which embodies such tendency. Similarly, **StayUp** should cause the agent to first clean the rooms in top half of the grid. For both of the last 2 cost function we only are looking for the tendency and not matching any exact cost numbers. Both StayLeft and StayUp are used with **UCS** algorithm.

**Heuristic:**

For heuristic we have 2 choices: Manhattan and Euclid distance.

**Rubric:**

1. The breakdown of the rubric is given in the table below, which displays the result of the test for the above functionalities. The numbers for each test can be off by +/-1.
2. **StayLeft** and **StayUp** each worth **10 points**.

**How to Test**

You should use PyCharm to load the project and test your solutions. Once you load the folder in PyCharm, at the top of the IDE, it gives you various run configurations to test all the required search cases:

*BFS, DFS, UCS, A\*\_Euclid, A\*\_Manhattan, Greedy\_Euclid, Greedy\_Manhattan*.

The last 4 correspond to informed searches. There are also “*corner*” version of the informed searches. In these cases, there are 4 dirty cells in the 4 corners of the grid. The corner configurations are just for testing to verify your implementations work well and won’t be used for grading and nothing needs to be done.

**Commandline Arguments:**

You can also run the script using command line format bellow, in a python console in the same folder as the project.

The agent can be launched from command line as following:

>python VacuumSearch.py -s searchAlgorithm -c costFunction -r heuristic

Look at the function readCommand() in vacuum\_search.py for detail of the above commandline options.

PyCharm’s run configurations are to make it easier to test your solution. They are using the command line options to pre-set the various app’s options.

Grading staff will be running the corresponding configuration listed in the below table to test your implementation and verify that it produces the numbers given (with +/-1 leeway)

For *StayUp* and *StayLeft* we only check the behavior for correctness, meaning when *StayLeft* is chosen, all the left half of the grid’s dirty cells must be cleaned before the right ones, but the order does not matter.

**Algorithm part**:

This part worth **10 points**. Answer the following question in a text file named **WeightedDFS.txt**:

As was explained in class, the vanilla DFS follows blindly the leftmost branch as it goes down the search tree, counting each step as one unit of cost. Suppose the cost per action is not one any more but is a positive number. How would you modify your DFS implementation/algorithm to take advantage of the variations of the costs for actions at each node to find a shortest path (lowest cost) for the DFS solution. Remember the solution would still be considered DFS, but a more optimum solution than the vanilla version. Provide your answer in the form of pseudo-code, similar to the BestFirstSearch algorithm or BFS algorithm given in class and in the search ppt. Your pseudo-code should be detailed enough that is implementable without further explanation needed by a software engineer.

**Expected Counter values table:**

Each algorithm has total Path Count and total Explored Count numbers. Your implementation must meet these number within +/-1.

The numbers are the followings:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Path\_Count** | **Explored\_Count** | **Cost** | **Points** |
| BFS | 59 | 478 | Step | 10 |
| DFS | 350 | 647 | Step | 10 |
| UCS | 59 | 545 | Step | 10 |
| Greedy\_Manhattan | 78 | 113 | Step | 5 |
| A\*\_Manhattan | 59 | 154 | Step | 5 |
| A\*\_Euclid | 72 | 101 | Step | 5 |
| BFS | 59 | 478 | StepTurn | 5 |
| DFS | 350 | 647 | StepTurn | 5 |
| UCS | 59 | 607 | StepTurn | 5 |
| Greedy\_Euclid | 76 | 100 | StepTurn | 5 |
| A\*\_Euclid | 72 | 103 | StepTurn | 5 |
| UCS |  |  | Stay\_Left | 10 |
| UCS |  |  | Stay\_Up | 10 |
| Weighted DFS Algo |  |  |  | 10 |

**Important**: You should not modify any code outside of the area specified. It makes the grading difficult since graders will look at your code as verifying you have used the class material and code base for the solution. Graders will drop your search.py into the assignment folder and try to run it. So any change done to other files will not be taken into account.

**Hand in:**

To hand in you must put the following 2 files in a folder, zipped it and upload it to Canvas:

1-search.py

2-WeightedDFS.txt

**Happy Coding.**