CS5011 P2 140022544 03 Nov 2017

Overview:

For this practical, we had to simulate a problem, where a "robot" should navigate through a post-disaster building. Robot has a map of the building floor and aims to locate the victim and transport him/her to a safe location on the map. Robot has to move efficiently through the floor in order to ensure victims survival. The map is represented as a 10x10 grid with obstacles, victim, exit point and robot located according to the test case. Robot is able to move only North/West/South/East directions. The aim of the practical is to evaluate a set of AI search algorithms using two main criteria: length of the robot route and efficiency represented by the number of visited search states. The practical is implemented using OOP approach. The structure and implementation decisions are explained in the following section, separated by specification parts.

Overall design:

In order to solve the problem according to the specification, I have used the guidelines from the lecture slides. I have decided not to parse the test cases from the file, but to store them directly in the program as a list of arrays. Secondly, I had written the method to parse each array and fill the Problem (I have created a separate class, that would store agent, victim and exit details in respectful data types, as well as the goal and the map). Since the initial goal of the problem is to get the robot to the victim, I had to call the search algorithms twice: once for getting the path to the victim if there is any, another to get the victim to the exit. The organisation of code execution for each of the algorithms was moved to a separate methods (*parseDFS/parseBFS*) in the main class.

The program is organised in a way, where each test map is parsed and the results of DFS and BFS algorithms(length and search steps) is printed to the console.

Part 1 is fully implemented and works both for BFS and DFS.

Part2 is attempted with best-search algorithm attempted (I have written a general structure of the *searchTreeBSA*, however didn't manage to complete it).

Algorithm design:

I have decided to use the pseudo-code from the lectures as a basis for my implementation. First of all, I have implemented common methods for both algorithms: creation of new nodes (*makeNode)*, addition of root node to the tree (*insertInFrontier*), *goalTest*, cost evaluation(*cost)*, checks whether node was in the frontier or in explored list(*isVisited*) and the successor function(*successorFn*) which returned all valid moves from an analysed state.

The difference in implementation of two algorithm in my case lies in the order of addition of new nodes to the frontier:

- BFS uses FIFO – or a queue - approach, this is why I have implemented a separate function called *insertAllEnd* which adds all the nodes,returnув by successor function to the end of the frontier.

- DFS uses LIFO – or a stack - approach, this is why I have implemented a separate function called *insertAllFront* which adds all the nodes,return by successor function to the front of the frontier.
Since both algorithms retrieve the first node of the frontier – it leads to a different behaviour.

Since every action is simply moving one cell in NSWE direction and since there is no difference in weight of any particular action, path cost is simply addition of all the moves on particular path.

Successor function works the same way for both functions: creates states for all the valid moves(move has to be in bounds of the map and the cell has to be free from an obstacle).

P1 testing:

In order to test the program, I have run it on all six test maps, provided with the practical specification. The output of my program for each case is following:

```
Map1:
BFS tree steps: 99
BFS length: 18
DFS tree steps:129
DFS length: 68

Map2:
BFS tree steps: 146
BFS length: 34
DFS tree steps:87
DFS length: 42

Map3:
BFS tree steps: 125
BFS length: 21
DFS tree steps:104
DFS length: 49

Map4:
No valid path to victim was found
No valid path to victim was found

Map5:
BFS tree steps: 115
No valid path to exit found
DFS tree steps:88
No valid path to exit found

Map6:
BFS tree steps: 76
BFS length: 29
DFS tree steps:31
DFS length: 31
```

From the output data, it can be seen, that although DFS algorithm usually performs better in terms of search states visited, BFS is generally shorter in

terms of the path length. In case of my implementation (with equal path weight) and no optimisation of DFS algorithm, DFS is performing more efficiently and provides an optimal path by the cost of larger amount of calculations.

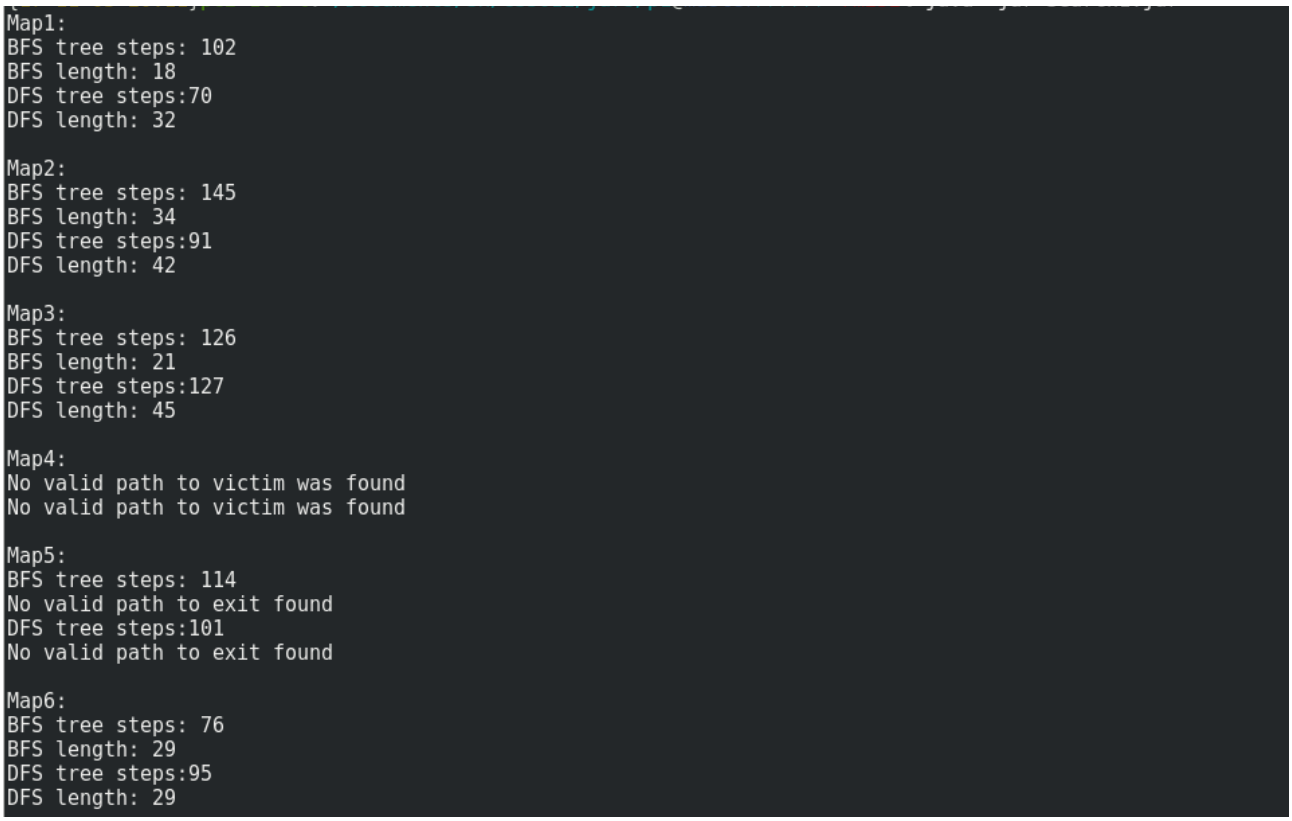
However, during my testing process, I have used debugging mode in order to track the program flow and have noticed one thing: during debugging process, the results of DFS algorithm are different from the regular run.

```
Map1:
BFS tree steps: 99
BFS length: 18
DFS tree steps:69
DFS length: 68

Map2:
BFS tree steps: 146
BFS length: 34
DFS tree steps:77
DFS length: 44

Map3:
BFS tree steps: 126
BFS length: 21
DFS tree steps:138
DFS length: 25

Map4:
No valid path to victim was found
No valid path to victim was found

Map5:
BFS tree steps: 114
No valid path to exit found
DFS tree steps:112
No valid path to exit found

Map6:
BFS tree steps: 75
BFS length: 29
DFS tree steps:57
DFS length: 41
```

I have re-run and inspected the code multiple times, but haven't found the reason for such behaviour. The output is consistent across multiple runs with the same input, but differs only for the debug mode. The only reason I have found for that could be racing condition and the way JVM processes code. I have even separated the counters and TreeSearch objects for different algorithms, which did not lead to any change in the output.

However, by the results of debug output and analysis of the maps, it can be seen that in case of large amount of obstacles and relatively limited amount of possible routes, DFS performs almost equally efficient.

The results of the jar execution differ from the execution in the IntelliJ and are following:



The results of jar execution are somewhat in between the results of an IDE program run.

Evaluation:

It is also worth noting, that since the purpose of this practical was to evaluate the algorithms, I have not implemented actual movement-instead I am simply moving the agent to the target location if the path is found. If I had more time I would implement actual movement and maybe even provide a function that would print the movement so the user can actually see the path, instead of the need of analysing debug data to ensure path existance.

Another thing to mention is the fact that due to my implementation of the new Node and State generation, I am constantly creating new instances of those classes, however all the checks for presence in frontier or explored are done by comparing the data inside those objects (particularly positions). Also I have not found any particular use for “action” in Node or “state space” in SearchTree apart from simply storing the data. I am positive, that by spending more time on analysis of the provided pseudo-code I would have found use for that data and would improve at least space complexity (lesser amount of created objects) of the project.

I have tried to adhere to the OOP paradigm and to split the logic of different parts into respectful methods with explanatory enough names.

Also I realise that the way my search works-first on victim, than on exit-is not the most efficient way and provided more time – I would implement a

solution, which would require a smooth, simple and single run of each algorithm.

Last but not least, I would try to debug the program more thoroughly in order to figure out the reasons of different behaviour for various run options of the programme:run,debug and jar.

Part2 evaluation:

At its base, Best First Search is a variation of Breadth FS with main difference in type of queues: Best First uses priority queue and heuristics (in my case is matrix proximity of one position over another to the goal) in order to structure the queue).

Running:

In order to run the programme, go to “jars” folder in the project, pick part 1 or part 2 folder and execute the following command: “ java -jar Search1.jar” or “ java -jar Search2.jar”.

Literature & bibliography:

While developing this project, I have used only lecture material and no third-party information sources.