De La Salle University - Manila

Term 2 A.Y. 2022-2023

In partial fulfillment

of the course

INTRODUCTION TO INTELLIGENT SYSTEMS

MCO1: MazeBot

CSINTSY - S12

SUBMITTED BY:
DELA CRUZ, Frances Julianne R.
GAMBOA, Mikkel Dominic
VERANO, Carl Matthew
YU, Hanz Patrick

SUBMITTED TO:
Mr. Thomas James Tiam-Lee, Ph.D.

I. Introduction

The bot would aim to find the specified goal given a text file with symbols that correspond to the maze. The '#' symbolizes a wall, the '.' symbolizes a space, the 'S' symbolizes the starting point and the 'G' symbolizes the goal. In this paper, we will cover how the bot works in finding the goal, its analysis, and the recommendations that could improve the bot's performance.

The bot follows the Breadth-First Search algorithm or BFS. This would mean that it would search all nodes at the current depth level first before moving onto the next. In this case, the bot would use a queue to implement BFS to find which nodes would be traversed first. It would start at the root node and then explore its children nodes. One by one, it would explore each node and if the goal has not been reached, the bot would enqueue its children nodes leaving it to be explored later. If the node is the destination goal, it would then print that the goal has already been found. (*Breadth First Search or BFS for a Graph* , 2012)

II. Program

Text file Input

The program requires a text file input named *"maze.txt"*, and the first row must contain an integer *n* that should be the size of the *n* x *n* maze. The following rows each contain a symbol that makes up the maze. This file must be located in the same folder as the .jar file. An incorrect text file would either result in a compilation error or the maze being smaller than intended.

Running the Program

You must first locate where you have extracted the given zip file, and the program must be ran through there. The program requires Java 19.0.2 to be ran, so it has been decided to add the Java installer in the zip file as well, in the case it has not been installed. The command to execute the program is simply "java -jar <filename>.jar". And then, the output of the program is then displayed through the command prompt.

Exploration

On execution, The maze size and starting index are displayed. Following that, the program will display the index of the state it is currently exploring, the list of indexes of already explored states, the index of the state it will explore next, and the current state number. Moreover, for the list of explored states, the order of exploration (which was first explored) will be from left to right. When counting the states, it has been decided that the first state is the first move of the bot. The state where the bot has not moved is what was defined as state 0. Furthermore, the maze is also visually shown similar to how it was input from the text file. If an area has been explored, the bot would leave the symbol 'O' once an index has been passed by. This section will continuously iterate until it reaches an end state.

End State Results

For the end result, the program may either reach a dead-end or goal state. When it reaches a dead-end state, it will display a list of all the explored states and how many states it had explored before reaching the dead-end conclusion. On the other hand, the goal state has the same displays, but it has 2 differing aspects. It will indicate that the search has reached a goal state, and a final path for the bot will also be displayed.

III. Algorithm

The basic pseudocode of the algorithm is as follows:

```
Explore(starting state)
Check if the current state is the goal
    if it is return
    if not disregard
Check if the tile to the North, South, East, and West are
          valid states to traverse
    if it is add the state to the list of moves in that respective order
    if not disregard
Add the current state to the list of explored states
Try to remove the first state in the list of moves
Explore(the removed state)
```

To describe the algorithm in further detail: a function called *explore* was created which passes specific states as its parameters. However, its first call passes the starting state or the “S” found in the *maze.txt* file. Although in the implementation there are other parameters such as a 2D array of the whole maze, the size of the maze, the list of moves, and the list of explored states, for demonstrative purposes, the pseudocode of the function will focus on the state being passed.

After passing the current state to explore into the function parameter, a series of checks will occur. The first check is to see if the current state is the goal state or not. If it is, the number of explored states, the list of explored states, and the final path taken will be displayed and the program will end. If it is not the goal state, the program will continue with the succeeding checks. The next check is to see if the state to the North, South, East, and West (in that particular order) are valid states to traverse. In this case, a state is valid if it is within the maze, if it is not a wall/”#”, and if it has not been explored. If the state passes all these checks, then that state will be added to the list of moves to be done. If the state does not pass even just one of these checks, then the condition will be ignored and the program will continue on. After all of these checks, the current state will be added to the list of explored states. Then, the program will try to remove the first move in the list of moves if there are still moves. If there are no more moves available then the program will display that it is unable to reach the Goal and the program will end. Finally, if there still are moves, then the function will recursively call itself and pass the state that was removed as its parameter as that will be the next state to be explored.

Based on the implementation, the algorithm that was followed is called Breadth-First Search as it explored all the nearby states in a certain order of the current state before moving on to the next states. This is due to the implementation of queues for our list of moves which always adds new moves based on the North, South, East, and West order to the back of the line, and always explores the first in line first.

IV. Results and Analysis

Time Complexity:

The time complexity of the program is O(n^2). Because the BFS algorithm of the program will search through the entire *n* x *n* maze one by one (recursively), this implies that at worst case, the whole search to find the goal state will be O(n^2).

What the bot can handle and how:

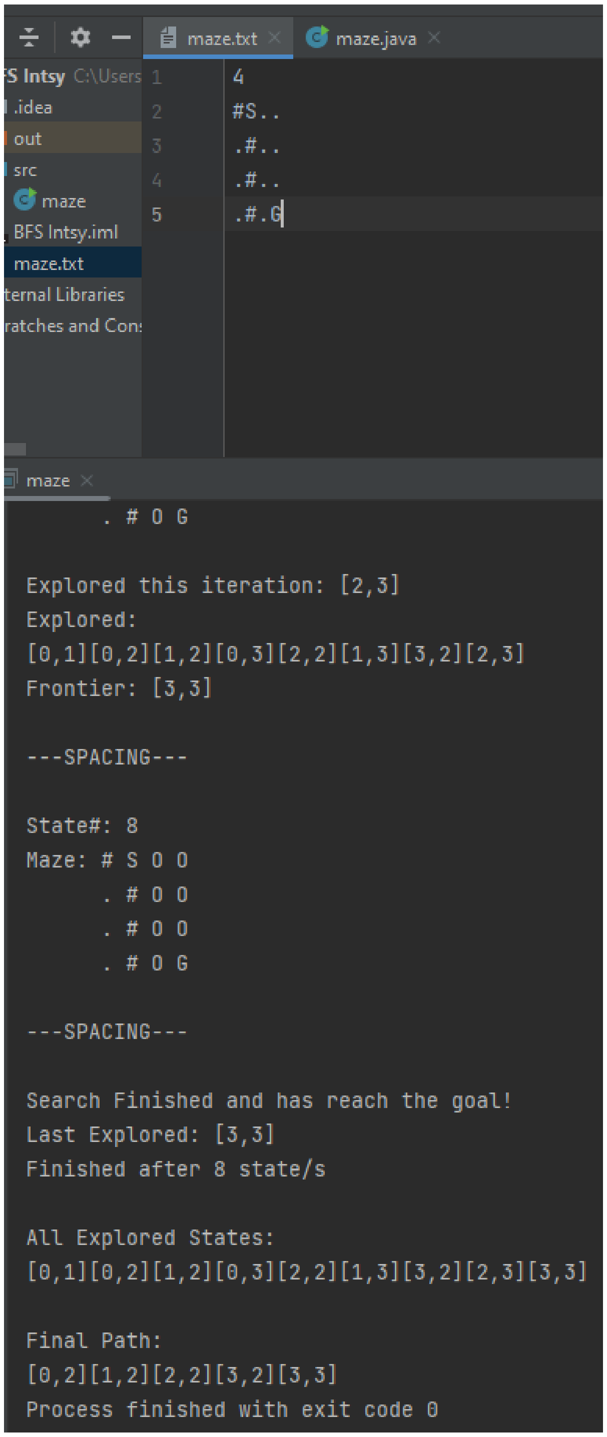
The purpose or use of BFS is to find the solution regardless of the cost
since BFS can find the solution as long as it's reachable and it exists. In this case for the maze, as long as it's reachable (no walls block the path between the Start and the Goal) then the MazeBot can handle the search for the goal. When there are multiple solutions (meaning there are multiple ways of reaching the goals in the context of the maze problem), the search always returns the minimal solution (the least in terms of length/depth). Moreover, according to Vidyashri (n.d.), this search is optimal when all the costs are equal, in this case, the distance between the points, regardless of their positions and how far they are from the goal, have the same costs, if not then it's not optimal. In the context of the maze program, with reference to the aforementioned statement by Vidyashri, the program's BFS is optimal since it disregards any costs when it searches the maze by breadth-first in a queue.

Moreover, BFS can handle small *n* x *n* maze size dimensions because although the program implementation is recursively going through the whole maze itself in the worst case, the small-sized mazes are still manageable by the program as the program can execute in a timely manner.

What it can't handle and why:

As mentioned earlier, BFS aims to return the shortest path that leads to the goal, however, if the graph, or in this case, the maze points are weighted, then it will no longer work. By definition, the search algorithm works when all points have equal weight so the shortest path would be the one where the least number of points were explored, however, adding weights to these points may make the costs differ from point to point thus implying that the fewest points explored might no longer be the shortest if the cost to travel to a point is expensive.

The size of the *n* x *n* matrix in this case is only at most 64 x 64, but when the size of *n* is a lot larger, then it's inefficiency will be more evident since it will take a very long time to finish the search (since worst case it'll go through all the points) and requires a lot of memory (Simic, 2022). Moreover, since the program recursively iterates through the points of the maze, the time complexity would become O(n^2) in the worst case thus implying that it may even take forever and the program might not even finish.

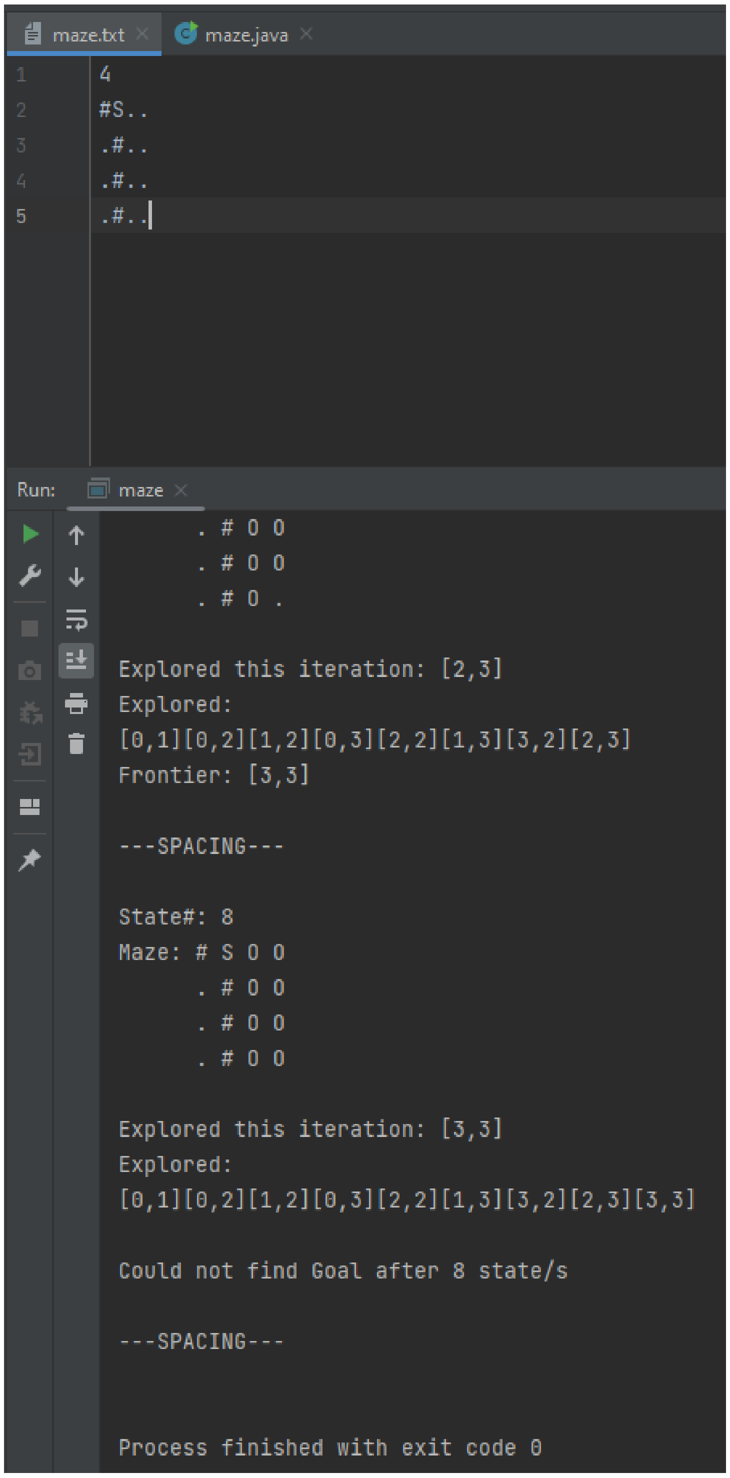


maze.txt maze.java

```
4
#S..
.#..
.#..
.#.G
```

```
       . # 0 G

 Explored this iteration: [2,3]
 Explored:
 [0,1][0,2][1,2][0,3][2,2][1,3][3,2][2,3]
 Frontier: [3,3]

 ---SPACING---

 State#: 8
 Maze: # S 0 0
       . # 0 0
       . # 0 0
       . # 0 G

 ---SPACING---

 Search Finished and has reach the goal!
 Last Explored: [3,3]
 Finished after 8 state/s

 All Explored States:
 [0,1][0,2][1,2][0,3][2,2][1,3][3,2][2,3][3,3]

 Final Path:
 [0,2][1,2][2,2][3,2][3,3]
 Process finished with exit code 0
```

Program prints out that it has reached the goal successfully when the goal is reachable.

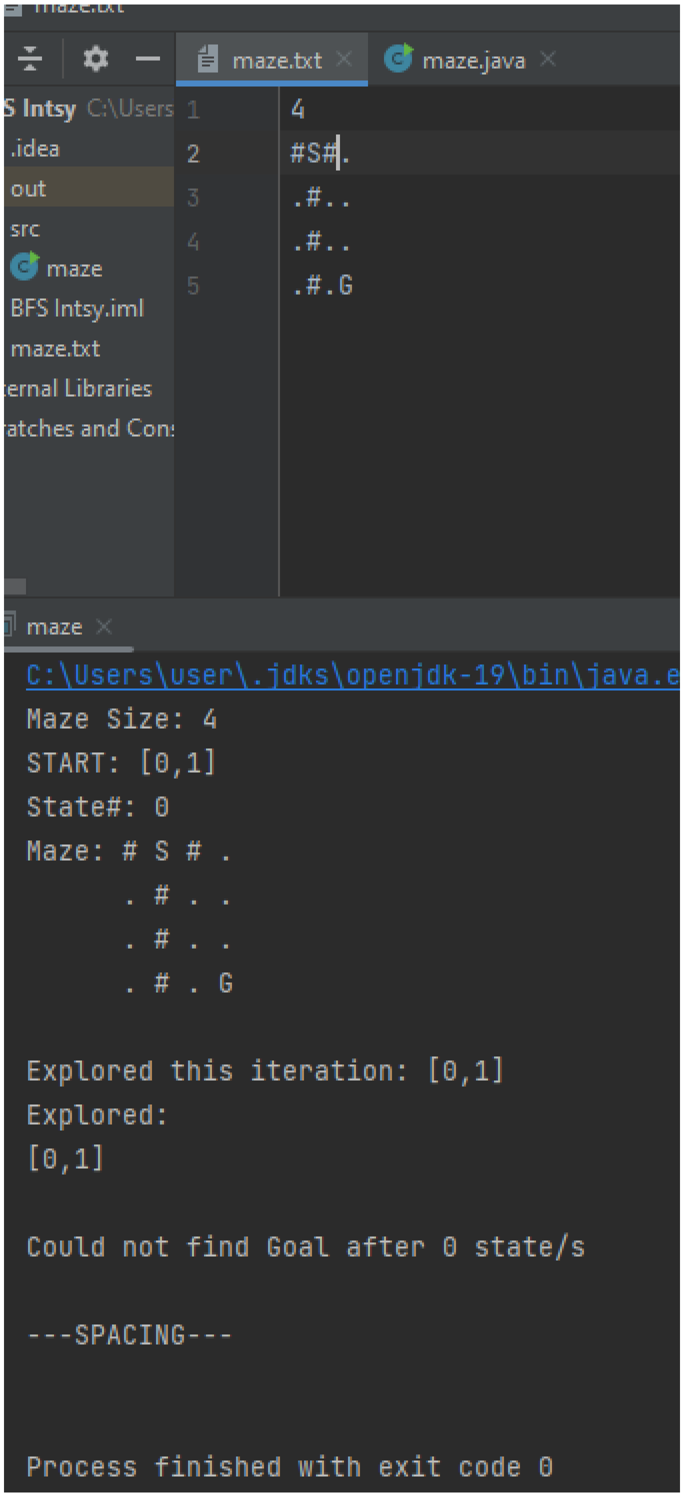

maze.txt   maze.java

```
4
#S..
.#..
.#..
.#..
```

Run: maze

```
      . # 0 0
      . # 0 0
      . # 0 .

Explored this iteration: [2,3]
Explored:
[0,1][0,2][1,2][0,3][2,2][1,3][3,2][2,3]
Frontier: [3,3]

---SPACING---

State#: 8
Maze: # S 0 0
      . # 0 0
      . # 0 0
      . # 0 0

Explored this iteration: [3,3]
Explored:
[0,1][0,2][1,2][0,3][2,2][1,3][3,2][2,3][3,3]

Could not find Goal after 8 state/s

---SPACING---


Process finished with exit code 0
```

The program will keep searching until the whole maze is explored when there is no goal.



```
4
#S#.
.#..
.#..
.#.G
```

```
C:\Users\user\.jdks\openjdk-19\bin\java.e
Maze Size: 4
START: [0,1]
State#: 0
Maze: # S # .
      . # . .
      . # . .
      . # . G

Explored this iteration: [0,1]
Explored:
[0,1]

Could not find Goal after 0 state/s

---SPACING---


Process finished with exit code 0
```

Program will print out that it did not reach the goal when it's impossible to reach, such as when it's blocked by walls.

V. Recommendations

Breadth-First Search works for small inputs, and given the constraints that the maximum size of the maze would be 64x64, it would still optimally work. However, if this were to be increased, it would not be wise to keep the bot as it is.

The traversal path of the bot would consume a lot of time and memory as it would explore every node at the current depth level until the destination node is found. Hence, this would mean that it would be inefficient to use it in large maze sizes. Furthermore, as pointed out in the analysis portion, when there are different costs for each node, the BFS would also not return the optimal path as the bot only accounts for the situation in which all actions cost the same.

To address this, the bot would have to be updated to follow other algorithms that are much more efficient when handling bigger sizes of mazes and would have a better time and space complexity. An example of this would be A*, which takes into account the cost and heuristic of the node. The A* algorithm would always return the most optimal path, given that the heuristic is consistent, which would affect and optimize the time it would take the bot to find the goal. The heuristic would be the one responsible for assisting the bot to find the optimal path to the goal by guiding the node by guiding it to nodes that are closer to the set goal. It would guide the bot to the node that is closer to our goal. Thus, reducing the nodes that are explored by the bot.

VI. References

Breadth First Search Algorithm - GATE CSE Notes. (2022, July 18). BYJUS; BYJU'S. https://byjus.com/gate/breadth-first-search-algorithm-notes/#:~:text=Disadvantages%20of%20Breadth%20First%20Search%20Algorithm&text=It%20can%20be%20slow%20since,paths%20through%20the%20search%20tree.

Breadth First Search or BFS for a Graph. (2012, March 20). GeeksforGeeks; GeeksforGeeks. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

H, V. M. (2022, January 24). *Breadth-first search Example Advantages and Disadvantages*. VTUPulse. https://www.vtupulse.com/artificial-intelligence/breadth-first-search-example-advantages-and-disadvantages/#:~:text=Advantages%20of%20Breadth%2Dfirst%20search%20are%3A&text=If%20there%20is%20a%20solution%2C%20BFS%20is%20guaranteed%20to%20find%20it.&text=The%20algorithm%20is%20optimal%20(i.e.

Moore, K., & Gujrati, A. (n.d.). *Breadth-First Search (BFS) | Brilliant Math & Science Wiki*. Brilliant.org. Retrieved March 6, 2023, from https://brilliant.org/wiki/breadth-first-search-bfs/#:~:text=BFS%20is%20good%20to%20use

Simic, M. (2021, October 20). *Depth-First Search vs. Breadth-First Search | Baeldung on Computer Science*. Www.baeldung.com. https://www.baeldung.com/cs/dfs-vs-bfs

VII. Contributions of Each Group Member

MEMBER	CONTRIBUTIONS
DELA CRUZ, Frances Julianne	Introduction and Recommendations (write-up), contributed in coding
GAMBOA, Mikkel Dominic	Algorithm (write-up), Program algorithm + code
VERANO, Carl Matthew	Results and Analysis (write-up), contributed in coding, Leader
YU, Hanz Patrick	Program (write-up), contributed in coding