Lab 1. EARIN

Variant 1

Oleg Kim

Seniv Volodymyr

26.03.2023

Task description:

Write a program that solves a maze using two search algorithms: breadth-first search (BFS) and a depth-first search (DFS). The maze is a 2D grid with empty spaces, walls, a start, and an end position. The objective is to find a path from start to end position. The maze should be loaded from file. A step-by-step visualization of the algorithm is required.

Launch instruction:

To run the code, open the directory consisting of the script itself and map and compile/run script.

The file is not pre-compiled since it can be open on different OS.

In the directory you can find 3 maps, the map is a .txt file with 0,1 separated by comas.

it can be modified according to representation:

'0'-represents wall,

'1'-represents path we can access,

'2'-start,

'3'-finish.

There is no limit on size of map.

IMPORTANT:

For the correctness, please place outer walls for the maze. The algorithm will stop when it goes outside of map.

To specify map, you need to change file name of map on line 36 in the script 'bfs_and_dfs.py'.

Example: maze = extract_map('map.txt') to maze = extract_map('map2.txt').

The program itself is a console application, when the script is running you will be asked to type in name of algorithm you want to use (all options displayed.).

Type the name of algorithm you want to test or 'end' to stop the script.

Theoretical background:

Breadth-first search (BFS) is an algorithm used for traversing or searching a tree or graph data structure. BFS explores the neighboring vertices of the current node before moving on to the next level of vertices. This makes it a good choice for finding the shortest path in a maze.

When applied to a maze, BFS can be used to find a path from a starting point to a goal point. The maze can be represented as a grid of cells, where each cell is either a wall or a passage. The algorithm works by traversing the passages in the maze, marking each visited cell, and keeping track of the current path.

The algorithm can be implemented using a queue data structure to keep track of the cells to visit, and a set to keep track of the visited cells. The time complexity of BFS is O(V+E), where V is the number of vertices in the graph and E is the number of edges. In a maze with n cells, the time complexity of BFS is also O(n).

BFS is a popular algorithm for solving mazes because it guarantees to find the shortest path and can be easily adapted to other problems such as finding the minimum number of moves to solve a puzzle.

Depth-first search (DFS) can be used to find a path from a starting point to a goal point. The maze can be represented as a grid of cells, where each cell is either a wall or a passage. The algorithm works by traversing the passages in the maze, marking each visited cell, and keeping track of the current path.

The algorithm can be implemented using a stack data structure to keep track of the path, and a set to keep track of the visited cells. The time complexity of DFS is O(V+E), where V is the number of vertices in the graph and E is the number of edges. In a maze with n cells, the time complexity of DFS is O(n).

DFS is a popular algorithm for solving mazes because it is simple to implement and does not require much memory. However, it may not always find the shortest path, and can get stuck in loops or dead ends if not implemented carefully.

Solution:

Map loading:

Firstly, we wanted to solve the problem with map loading. We did by opening the file by “open(file_name)” python function and started loading inside of the two-

dimensional list line by line the content of the file and then return this list. Since our map consists of 1's, 0's, 2 and 3 separated by the commas we used the line.split(',') function. All of this was done in "extract_map(file_name)" function.

After that we made two functions which find the staring position and end position: "find_start(maze)" and "find_end(maze)" respectively. They operate in the way that they use a nested loops which iterate line by line, column by column finding the elements which equal '2' – for start position and '3' – for end position. At the end both functions return two pair of coordinates inside the list with start and end positions.

Print the maze:

With the help of nested loops we traverse through the two dimensional list line by line and replacing each symbol in the following way:

1. 'S' – start
2. 'E' – end
3. '*' - followed path
4. ' ' – free path
5. 'X' – walls

Printing the map.

BFS algorithm:

Solution of the BFS algorithm is the "bfs(maze,start,end)" function. It takes a two-dimensional array "maze", starting position "start" and end position "end". Firstly, since the BFS uses the queue solution principle we create a queue FIFO (first in first out) with start position to maintain a list of cells to be visited next. The next positions are calculated inside the for loop which adds to the coordinates of current position the coordinates of the possible movements from move list "moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]". If we meet the wall, it returns false, and no step is done. If it is not the wall the "next_move" added to the visited list and queue. After each step the map is printed with the visited cells passed as argument to print the path. This algorithm lasts till current position is not equal to the end position and the queue is not empty. In case of finding the end position function returns true, in case of failure it returns false. Also we added a time() function with small interval in order to make the steps of algorithm better visible.

DFS algorithm:

Solution of the DFS algorithm is the "dfs(maze,start,end,visited=set(), path=[])" function. . It takes a two dimensional array "maze", starting position "start", end position "end", set of visited cells "visited=set ()" and "path=[]" in order

to call function recursively on neighboring cells. The principle of work is quite similar to BFS but here we utilize stack principle LIFO(last in first out) instead of queue FIFO(first in first out). The principle as follows:

1. We choose a starting cell in the maze.
2. Mark the current cell as visited.
3. If the current cell is the end cell, stop and return the path.
4. Otherwise, for each neighbor of the current cell that has not been visited, mark it as visited and add it to the current path.
5. Recursively apply the algorithm to the current cell and the new path.
6. If the recursive search returns a path to the goal cell, stop, and return the path.
7. If all neighbors have been visited and the goal has not been found, backtrack to the previous cell in the path and continue the search from there.

Result:

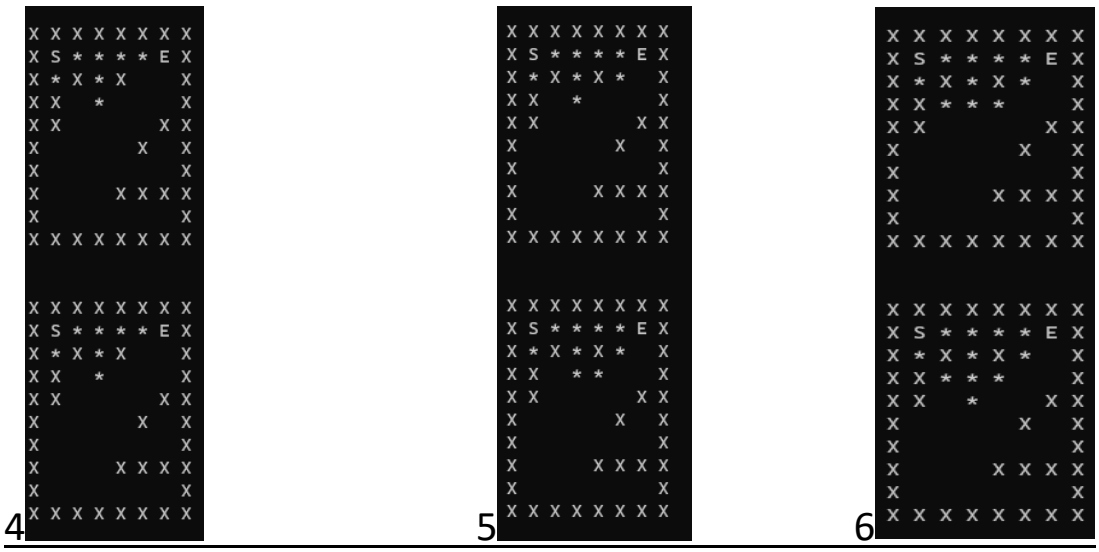
For map.txt:

BFS:

1

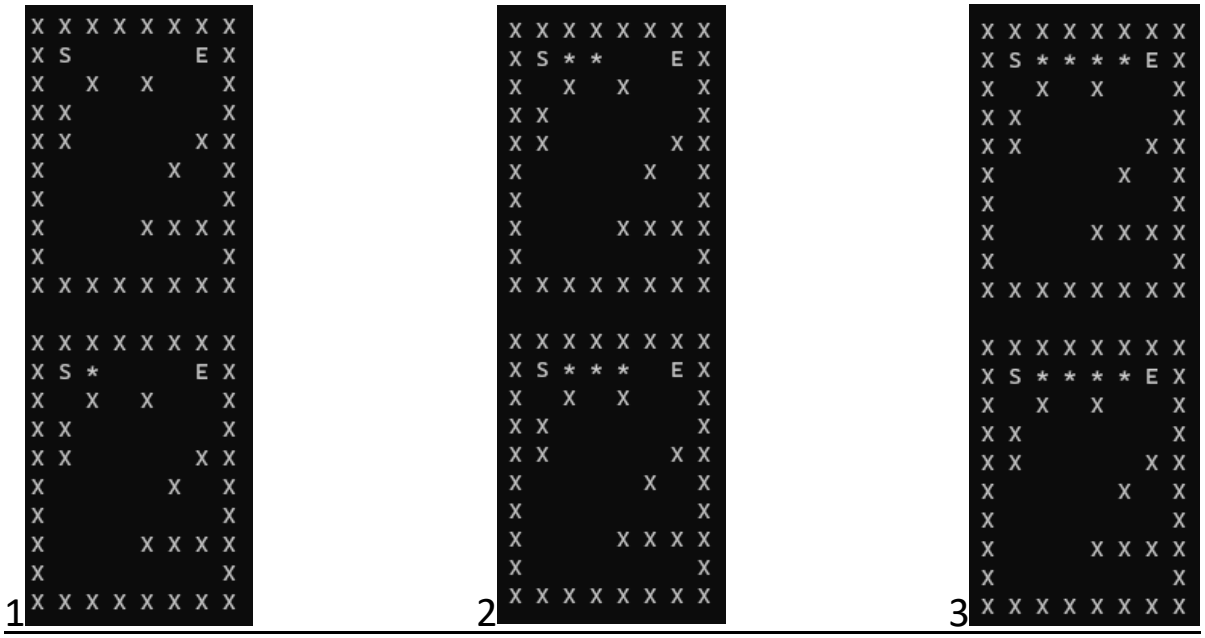
2

3



number of steps: 12

DFS:



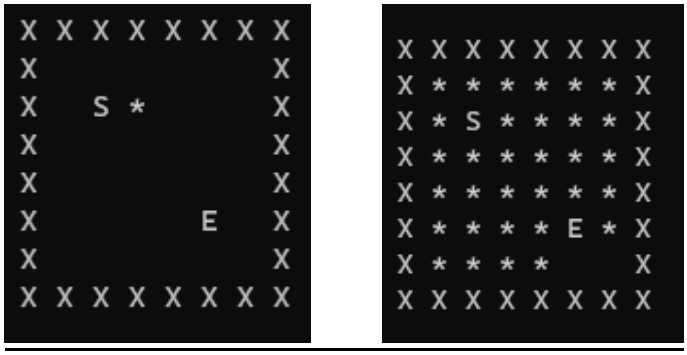
number of steps: 5

In this example DFS is more affective since it has only one direction leading to end while bfs searches in every possible way making more steps.

For map2.txt:

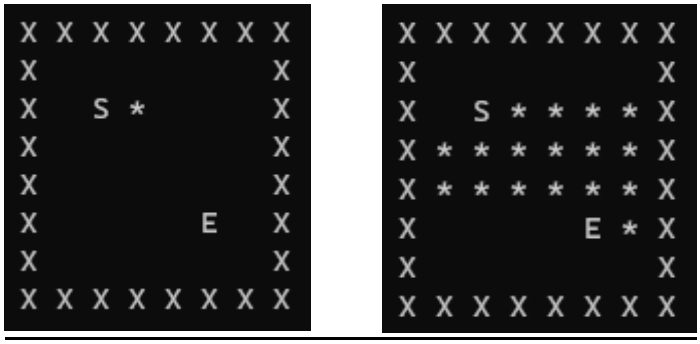
Since there were done too many steps it was irrelevant to make all the screenshots, we just added the results of step counting.

BFS:



number of steps: 34

DFS:



number of steps: 18

For map3.txt:

For the last map DFS also appeared to be more effective than BFS.

What could be improved:

The program stops working when going out of borders, the script can be rewritten so that it won't go out of borders or will treat the element on border as 0 regardless of its value. visualization can be made using pygame or turtle graphics for better visualization. Also, we can apply backtracking to track the route from start to finish.

Conclusion:

DFS (Depth First Search) algorithm can be faster than BFS in finding a solution when the maze has a lot of long and winding paths. This is because DFS explores one path to its end before backtracking and trying another path, which means it can quickly reach the end of long winding paths. However, DFS does not guarantee the shortest path to the solution, and theoretically can get stuck in an infinite loop if the maze contains cycles.

BFS (Breadth First Search) algorithm, on the other hand, guarantees the shortest path to the solution if the maze does not contain cycles. BFS explores all the neighboring cells of the current cell before moving to the next level, which means it can find the shortest path to the solution. However, BFS can take a long time to find the solution if the maze has a lot of dead-ends or long paths, as it explores all possible paths in a breadth-first manner.

DFS can be faster than BFS but very big factor the maze because if there are lots of dead ends and cycles both algorithms can get stuck. BFS can require more memory than DFS and may not always be the most efficient algorithm for very large mazes or graphs.