Uninformed Search

Implement a grid-maze solving program that uses uninformed search algorithms to solve grids. The agent's actions are moving in one of four directions: up, down, left, and right. **Each action has a step cost of 1.**

Assignment requirements

1. Load the grid from a text file.
2. Allow for different start and goal pairs.
3. After running the algorithm to solve the grid, display the final path on the grid visually and write the new grid to a separate file.
4. Record the number of expanded nodes when running an algorithm, and the path cost of the solution.
5. Implement **both** Breadth First Search (BFS) and Depth First Search (DFS). **Do not use separate functions for these.**

A grid is formatted like below where 1's represent locations the agent cannot traverse:

```
1 1 1 1 1 1 1 1
1 0 0 0 1 1 1 1
1 0 0 0 0 0 0 1
1 1 1 0 0 1 0 1
1 0 1 0 0 1 0 1
1 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1
```

The final path can be displayed with 'S' for the initial state, 'G' for the goal state, and '*' symbols for the path:

```
1 1 1 1 1 1 1 1
1 S 0 0 1 1 1 1
1 * * * 0 0 0 1
1 1 1 * 0 1 0 1
1 0 1 * G 1 0 1
1 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1
```

Get code to read a text file into a 2d list, and code to write a path to a file: https://pastebin.com/S5yzfJPf

Output:

1. The path in the file "path.txt" or print to console that no solution exists.
2. The path cost of the solution
3. The number of nodes expanded for the search

Recommended Steps:

Step: Main function

Create a main function and set up your script to call it when executed. In the main function, call readGrid to read a file with a grid into a 2D list of 1's and 0's. Create list variables to hold the start and goal locations. Then call the function that implements the searching algorithm.

Step: Create Node class

Create a class called Node that has members 'value' and 'parent'. The 'value' member should be a 2-element list representing a location. The 'parent' member should be a reference variable to a Node object.

Step: Breadth First Search main loop skeleton

Create a function called uninformedSearch: def uninformedSearch(grid, start, goal).
Initialize a Node object called 'current' to track which node the agent is currently at.
Initialize a data structure to hold the open list. Initialize a data structure to hold the closed list.
Create a while loop with a proper termination condition. This while loop will contain the searching actions. You can set up the basic skeleton of this now by using comments as placeholders for functions you have not implemented yet.

Step: getNeighbors function

Create a function called getNeighbors that takes in a location and a grid:
def getNeighbors(location, grid)
The 'location' parameter should be a 2-element list that describes a location, such as [1,1], [1,2], etc. The 'grid' parameter is the 2D list of 1's and 0's representing a grid.
The function should return a list of positions (e.g., [[1,3], [2,1], [2,2]]) that 1) are adjacent to the 'location' parameter, 2) have a value of 0 on the grid, and 3) are within the grid boundaries.

Step: expandNode function

Create a function called expandNode that takes in at least a Node object (it may take in more parameters depending on your preference). The purpose of this function is to get all the neighbors of the Node object passed in and add those neighbors to the open list if and only if the necessary conditions are passed.
Note: do not use the "in" operator to check if a Node object is in a list, and do not use the == operator to compare two Node objects. Remember that object variables are references so the == operator will return false if the objects do not have the same memory address. The "in" operator uses the == operator. My suggestion is to create a function to compare two Node objects and/or create a function to check if a Node object is in a list of Node objects.

Step: Finish Breadth-First Search main loop

With the expandNode function finished, you can continue to work on and finish the BFS main loop. Make sure that your loop terminates on the correct conditions.

Step: setPath function

This function should be called once the goal has been found and the final path must be obtained. Create a function called setPath that takes in two parameters: ‘current’ and ‘path’. The ‘current’ parameter should be the Node object whose value is the goal location. The ‘path’ parameter should be an empty list that gets filled in with the final path by the setPath function. Use this function to find your final path and then return the path. Outside of your search algorithm function, call the `outputGrid` function to write a file showing your final path. Confirm that your path is correct.

Step: Implement Depth-First Search

After BFS is complete, copy the function into a separate function called depthFirstSearch. Make the necessary changes to implement DFS instead of BFS. Only 1 or 2 lines of code should change.

Step: Combine BFS and DFS functions into a single function called uninformedSearch

Create a new function called uninformedSearch that takes an additional parameter specifying whether to run BFS or DFS. Look at the changes you made to move from BFS to DFS. Integrate those changes into this function. Delete both function breadthFirstSearch and depthFirstSearch. Note: You may write more functions than the ones listed in the above steps. As you work through the steps, you may find it convenient or necessary to write several utility functions.

Bonus: Allow the user to input the initial and goal states.

Prompt the user to enter values for the initial and goal states. Pass those values to your searching algorithm functions.