TCSS 435
Programming Assignment 1

Objective: Expressing the problem as a search problem and identifying proper solving methods. Specifying, designing and implementing uninformed & informed search methods.

Instructions: In this assignment you will implement a set of search algorithms (BFS, DFS, Greedy, A*) to find solution to the **path planning problem** in a Maze. Theoretically solving the problems as a search process includes progressive construction of a solution:

- Establish the problem's components
  - Initial state
  - Final state
  - Operators (successor functions)
  - Solution
- Defining the search space
- Establish the strategy for search a solution into the search space.

Step 1: Defining our Maze & Agent

For a maze of size M x N find the best path returned by the search strategies to navigate an agent from its initial state to the goal state.

Update your `MazeRunner.py` source code (from Assignment 0) to create a Maze using following configurations *(Note: The `MazeRunner.py` will be our tester file)*:

- Maze size will M x N. We will test for following two Maze sizes: 20 x 30 & 50 x 50
- You can choose the theme to be light or dark and pattern to be vertical or horizontal.
- Goal State will be generated randomly using random number generator.
- Set loop percent to be 100.

Place an agent in Maze at cell position (i.e. row & column) generated using a random number generator. Make sure that goal cell and agent cell are not the same. You can use the default shape of the agent (i.e. square), or you can change the shape to be 'arrow' but set the footprints to be True since we want to visualize the final path.

Step 2: Implementing Search Algorithms

Create a class file `SearchSolution.py` to implement the following search strategies:

- Breadth-first search (BFS)
- Depth-first search (DFS) – depth-first search needs to check for cycles, or a solution will most likely never be found.
- Greedy search (GS), using Manhattan Distance as the heuristic.
- A* (AStar) using Manhattan Distance as the heuristic.

Notes:

- *Step cost of the agent = 1.*
- *You can implement these search strategies using either tree or graph search.*
- *Use one 'queue' for all algorithms as mentioned in the lecture notes.*

- *Your search algorithms will need the `maze_map` to know open spaces vs. closed walls.*
- *Agent can only navigate through open spaces (i.e. agent cannot jump over the wall)*
- *Planning is part of the simulation process, i.e. you will only show the agent's final solution path from its initial state to goal state on the GUI Maze.*

Step 3: Passing inputs

Your program *`MazeRunner.py`* will accept input arguments from the command line in the following format: `MazeRunner.py [M] [N] [searchmethod]`

- `[M]` refers to the total number of Maze rows
- `[N]` refers to the total number of Maze columns
- `[searchmethod]` can be: BFS, DFS, GS, AStar.

Examples:

- `MazeRunner.py 20 30 DFS`
- `MazeRunner.py 50 50 AStar`

Step 4: Generating Outputs

Your program will generate 2 different outputs – GUI & and a readme file.

1. GUI Output: Trace the agent's solution path from its initial state to the goal state as returned by each search method.
2. `Readme.txt`: In your readme file you will write (file writing) the evaluation results in the following format:
   `[size][searchmethod]: [depth], [numCreated], [numExpanded], [maxFringe]`
   - `[size]` represents the Maze size.
   - `[searchmethod]` refers to BFS, DFS, GS, AStar
   - `[depth]` represents the depth in the search tree where the solution is found. The integer will be zero if the solution is at the root and it will be "-1" if a solution was not found.
   - `[numCreated]` is the counter that is incremented every time a node of the search tree is created (output 0 if depth == -1).
   - `[numExpanded]` is the counter that will be incremented every time the search algorithm acquires the successor states to the current state, i.e., every time a node is pulled off the fringe and found not to be the solution (output 0 if depth == -1).
   - `[maxFringe]` is the maximum size of the fringe at any point during the search (output 0 if depth == -1).

Submission Guideline: Zip & upload the following files to the assignment submission link:

- `SearchSolution.py`: Class file that implements BFS, DFS, GS, AStar search strategies.
- *`MazeRunner.py`*: The main source code file that accepts the input and connects your pymaze to the search strategies.
- `Readme.txt`: Output for each of the algorithm and problem size in the format explained in the output section.