Full name	Kiều Công Hậu
ID	**18127259**
Class	18CLC1

Class 18CLC – Term III/2019-2020

Course: CSC14003 - Artificial Intelligence

Lab01: Search Strategies

REPORT

A. Check list

No.	Task	Progress
1	Implement all of the searh strategies: - *Breadth-first search* - *Uniform-cost search* - *Iterative deepening search* that uses depth-first tree search as core component and avoids loops by checking a new node against the current path - *Greedy-best first search* using the Manhattan distance as heuristic - *Tree-search A** using the same heuristic as above	100%
2	For each search strategy, print to the console the following information: - The time to escape the maze - The list of explored nodes (in correct order) - The list of nodes on the path found (in correct order).	100%
3	Create some example mazes as input text files in the INPUT folder and corresponding search results to them as output text files in the OUTPUT folder.	100%

B. Brief description of main functions

The usage, the input parameters, and the returned result of each function are noted carefully as comments in my source code. You can read the below brief description along with those comments for better understanding.

1. ***Main (main.py)***

In *'main.py'*, the input text file of the maze is read first and stored into a graph data structure, then all of search-strategy functions are invoked to solve this maze and the corresponding output is printed onto the console.

By default, the input text file's path is *"../INPUT/maze_0.txt"*. If you want to test on another maze, change the value of *input_file_name* (line 10). I also prepare 3 more input text files which are located at the INPUT folder (*maze_1.txt*, *maze_2.txt* and *maze_3.txt*).

2. ***Input/Output handle (IOHandle.py)***

'IOHandle.py' is created in order to implement some functions related to Input/Output Handle such as:

- Reading the input file text of a maze *(read_maze)*
- Printing the maze's solution onto the console *(print_solution, print_solution_ids)*

3. ***Breadth-first search (BFS.py)***

The *breadth-first search* algorithm (BFS for short) is implemented in *'BFS.py'* with the main function is named *breadth_first_search*. This function returns 3 values: (*the time to escape the maze, the list of explored nodes, the list of nodes on the path found)*. If there is no solution, this function return *(None, None, None)* instead.

The function *get_path* is used to get a path from the list of explored nodes.

4. ***Uniform-cost search (UCS.py)***

The *Uniform-cost search* algorithm (UCS for short) is implemented in *'UCS.py'* with the main function is named *uniform_cost_search*. This function returns 3 values: (*the time to escape the maze, the list of explored nodes, the list of nodes on the path found)*. If there is no solution, this function return *(None, None, None)* instead.

The function *update* is used to update a node in the frontier with the lower cost.

The function *get_path* is used to get a path from the list of explored nodes.

5. Iterative deepening search (IDS.py)

The *Iterative deepening search* algorithms (IDS for short) is implemented in *'IDS.py'* with the main function is named *iterative_deepening_search*. This function returns 3 values: (*the time to escape the maze, the list of explored nodes, the list of nodes on the path found)*. If there is no solution, this function return *(None, None, None)* instead.

The *depth_limited_tree_search* is used to find the maze's solution with the limit depth. This function return:

- *(the time to escape the maze, the list of explored nodes, the list of nodes on the path found)* if there is a solution
- *(the time to try to escape the maze within the limit depth, the list of explored nodes, None)* if there is no solution within the depth limit.
- *(None, None, None)* if it cannot find the solution.

The function *recursive_depth_limited_tree_search* is just a part of the function the above function (*depth_limited_tree_search)*, which helps the algorithm to explore the nodes recursively.

To avoid loops by checking a new node againts the current_path, I create a dictionary *on_current_path* and a list *current_path* to track which nodes are on the current path.

6. Greedy-best first search (GBFS.py)

The Greedy-best first search algorithm (GBFS for short) is implemented in 'GBFS.py' with the main function is named greedy_best_first_search. This function returns 3 values: (*the time to escape the maze, the list of explored nodes, the list of nodes on the path found)*. If there is no solution, this function return *(None, None, None)* instead.

The function *update* is used to update a node in the frontier with the lower cost.

The function *get_path* is used to get a path from the list of explored nodes.

The function *manhattan_distance* is used to calculate the Manhattan distance (heuristic) of a state.

7. Tree-search A (TSA.py)*

The Tree-search A* algorithm (TSA for short) is implemented in ‘TSA.py’ with the main function is named tree_search_a. This function returns 3 values: (*the time to escape the maze, the list of explored nodes, the list of nodes on the path found)*. If there is no solution, this function return *(None, None, None)* instead.

The function *get_path* is used to get a path from the list of explored nodes.

The function *manhattan_distance* is used to calculate the Manhattan distance (heuristic) of a state.

THE END