

Mini-Project 1 Additional Documentation

Documentation :

The code provided is a comprehensive implementation of a maze-solving program in Python. It uses multiple search algorithms such as Depth-First Search (DFS), Breadth-First Search (BFS), Greedy Best-First Search (GBFS), and A* Search. Here's an overview of the different parts of the program:

Import Statements

This section brings in necessary Python modules such as pygame for GUI generation, csv for maze file reading, and Queue, LifoQueue, and PriorityQueue from queue for managing nodes in different search algorithms.

SearchNode Class

This class represents a node in the search space. Each node is linked to its parent, has a state which is its position in the maze, action taken to reach the state, path cost from the start to the node, heuristic cost from the node to the goal, and a list of its children.

Maze Class

This class is used to load the maze from a given CSV file and find the start and goal positions in the maze.

MazeGUI Class

This class uses the Pygame library to create a graphical interface for the maze, drawing the maze, the solution path, and managing the display refresh rate.

SearchShell Class

This class is the core of the program. It allows the user to input necessary details, such as the maze file and the search algorithm to be used, and whether to use verbose mode. It handles the execution of the selected search algorithm and displays the maze and the current state of the search in the terminal, as well as visualizing the solution using Pygame. It also manages the pausing and continuing of the program based on user input.

The SearchShell class uses a variety of search algorithms based on the user's choice:

1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)
3. Greedy Best-First Search (GBFS) using Manhattan Distance
4. Greedy Best-First Search (GBFS) using Euclidean Distance
5. A* Search using Manhattan Distance
6. A* Search using Euclidean Distance

The search algorithms are implemented using different types of queue structures: BFS uses a FIFO Queue, DFS uses a LIFO Queue (Stack), and GBFS and A* Search use a Priority Queue.

In verbose mode, the program provides periodic reports of the search progress.

Main Program

This section creates an instance of the SearchShell class and starts it. It also runs a separate thread to manage verbose status.

Please note, some of the methods seem to be missing in your code snippet such as ``SearchNode.expand_state``, ``MazeGUI.draw_solution``, ``SearchShell.expand_state`` and more. If they are defined elsewhere or in the same file but not provided in the snippet, the code should run as expected. If not, the missing methods need to be implemented for the code to work correctly.

```

195     self.maze_gui = MazeGUI(self.maze_file)
196     self.maze_gui.run(solution_path)
197     self.write_solution_to_file(solution_path)
198     print("Number of loops encountered: ", loops)
199     print("Number of unique states visited (state
200
201
202     def write_solution_to_file(self, path):
203         actions = []
204         for i in range(1, len(path)):
205             prev_row, prev_col = path[i-1]
206             curr_row, curr_col = path[i]
207             action = ""
208             if curr_row == prev_row and curr_col > prev_col:
209                 action = "Right"
210             elif curr_row == prev_row and curr_col < prev_col:
211                 action = "Left"
212             elif curr_row > prev_row and curr_col == prev_col:
213                 action = "Down"
214             elif curr_row < prev_row and curr_col == prev_col:
215                 action = "Up"
216             actions.append(action)
217         self.write_solution_to_file(path, actions)
218
219     def write_solution_to_file(self, path, actions):
220         with open(path, 'w') as f:
221             f.write(str(actions))
222
223 if __name__ == '__main__':
224     solver = Solver('maze1.csv')
225     solver.solve()
226
227 0 0 1 0 1 0 0 0 1 1 0 0 X E
228 Number of unique states visited (state space): 33
229 Current State: (14, 14)
230 S 1 1 0 0 0 0 1 0 1 0 0 0 0 1
231 X X X 1 1 0 0 0 0 0 0 0 0 0
232 1 1 0 X X 1 0 0 1 0 0 1 0 1 0
233 1 0 1 0 X X 1 0 0 1 1 0 1 0
234 0 1 1 0 1 X 1 0 0 0 0 0 0 1 1
235 0 1 0 0 0 X X X 0 1 1 1 0 0 0
236 1 1 1 0 1 1 X 1 1 0 1 0 1 0
237 0 0 1 0 0 X X X 0 1 1 0 0 0
238 1 0 0 0 0 0 1 0 X 1 0 0 0 1
239 1 0 0 1 1 0 1 0 X 1 0 X X 1 0
240 1 0 0 0 0 0 0 1 X X X 1 X 1 0
241 1 0 0 1 0 1 0 0 0 0 1 0 X 1 0
242 0 0 1 1 0 1 0 0 0 1 1 0 X X 1
243 0 1 0 0 0 1 0 0 1 0 0 0 1 X 1
244 0 0 1 1 0 1 0 0 0 1 1 0 0 X X
245 Solution Found!
246 Length of Solution: 30
247 Actions: [(1, 0), (1, 1), (1, 2), (1, 3), (2, 3), (2, 4), (3, 4), (3, 5), (4, 5), (5, 5), (5, 6), (5, 7), (6, 7), (7, 7), (7, 8), (8, 8), (9, 8), (10, 8), (10, 9), (10, 10), (9, 10), (9, 11), (9, 12), (10, 12), (11, 12), (12, 12), (12, 13), (13, 13), (14, 13), (14, 14)]
248 Number of loops encountered: 0
249 Number of unique states visited (state space): 33

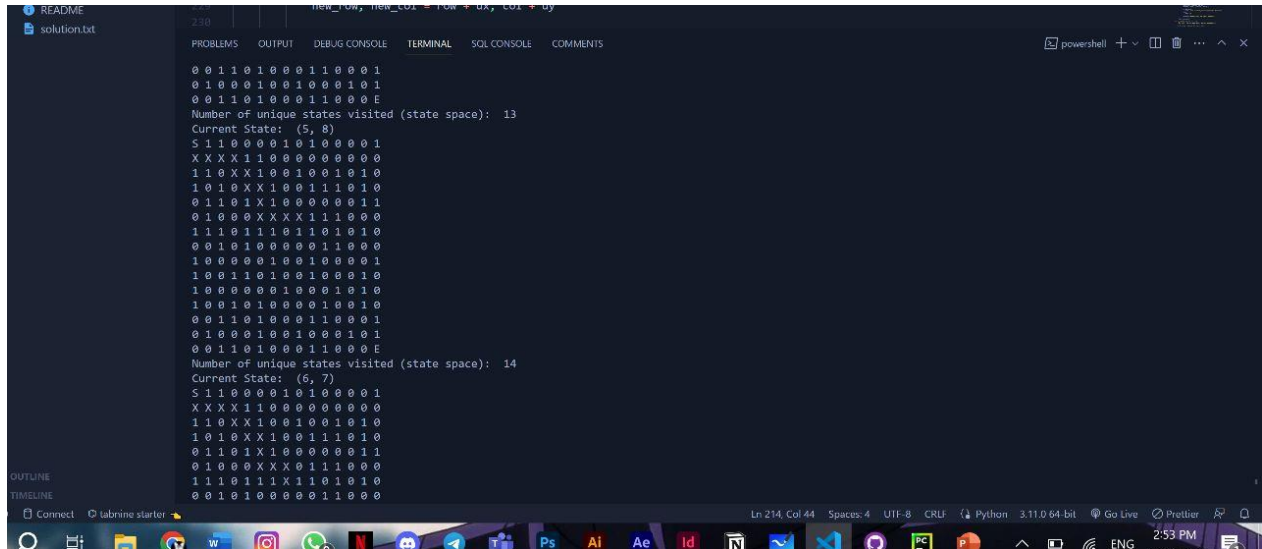
```

Solution Found!
Length of Solution: 30
Actions: [(1, 0), (1, 1), (1, 2), (1, 3), (2, 3), (2, 4), (3, 4), (3, 5), (4, 5), (5, 5), (5, 6), (5, 7), (6, 7), (7, 7), (7, 8), (8, 8), (9, 8), (10, 8), (10, 9), (10, 10), (9, 10), (9, 11), (9, 12), (10, 12), (11, 12), (12, 12), (12, 13), (13, 13), (14, 13), (14, 14)]
Number of loops encountered: 0
Number of unique states visited (state space): 33

```

1 Right
2 Left
3 Down
4 Up
5 Right
6 Left
7 Down
8 Up
9 Right
10 Left
11 Down
12 Up
13 Right
14 Left
15 Down
16 Up
17 Right
18 Left
19 Down
20 Up
21 Right
22 Left
23 Down
24 Up
25 Right
26 Left
27 Down
28 Up
29 Right
30 Left

```

A screenshot of a code editor with a dark theme. The editor shows a file named 'solution.txt' with a maze-solving application's output. The output displays a maze grid with 'S' for start and 'E' for end, followed by the number of unique states visited (13 and 14) and the current state coordinates (5, 8) and (6, 7). The maze grid is composed of 0s and 1s, with 'S' and 'E' marking the start and end points. The editor interface includes tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, SQL CONSOLE, and COMMENTS. The bottom status bar shows 'Ln 214, Col 44' and other editor settings.

The given python code is a maze-solving application that utilizes different search algorithms (DFS, BFS, GBFS, A*) to find the path from the start position to the end position in a maze.

The code provided is a maze solver that reads a maze from a user-provided file, solves it using user-selected search algorithms (Depth-First, Breadth-First, Greedy Best-First, or A*), and provides verbose mode with control over reporting and pausing intervals.

1. Classes:
 - SearchNode
 - Maze
 - MazeGUI
 - SearchShell

2. Methods:

For the class SearchNode:

- __init__
- __lt__
- expand_state
- reconstruct_path

For the class Maze:

- __init__
- load_maze
- find_start
- find_goal

For the class MazeGUI:

- __init__

- draw_maze
- draw_solution
- run

For the class SearchShell:

- __init__
- enable_verbose
- disable_verbose
- toggle_verbose
- manage_verbose_status
- read_maze_file
- read_verbose_mode
- read_report_interval
- read_pause_interval
- read_search_algorithm
- run_search_algorithm
- run_dfs
- run_bfs
- run_gbfs
- run_a_star
- run_search
- write_solution_to_file
- expand_state
- is_valid_move
- calculate_heuristic
- report_progress
- report_solution
- reconstruct_path
- start

And the imported libraries' methods and classes used are:

- sys: exit
- pygame: init, display.set_mode, display.set_caption, time.Clock, event.get, QUIT, draw.rect, quit
- csv: reader
- math: sqrt
- time: sleep
- threading: Thread
- queue: Queue, LifoQueue, PriorityQueue