Adnane El Amrani & Nabil El Maaroufi

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

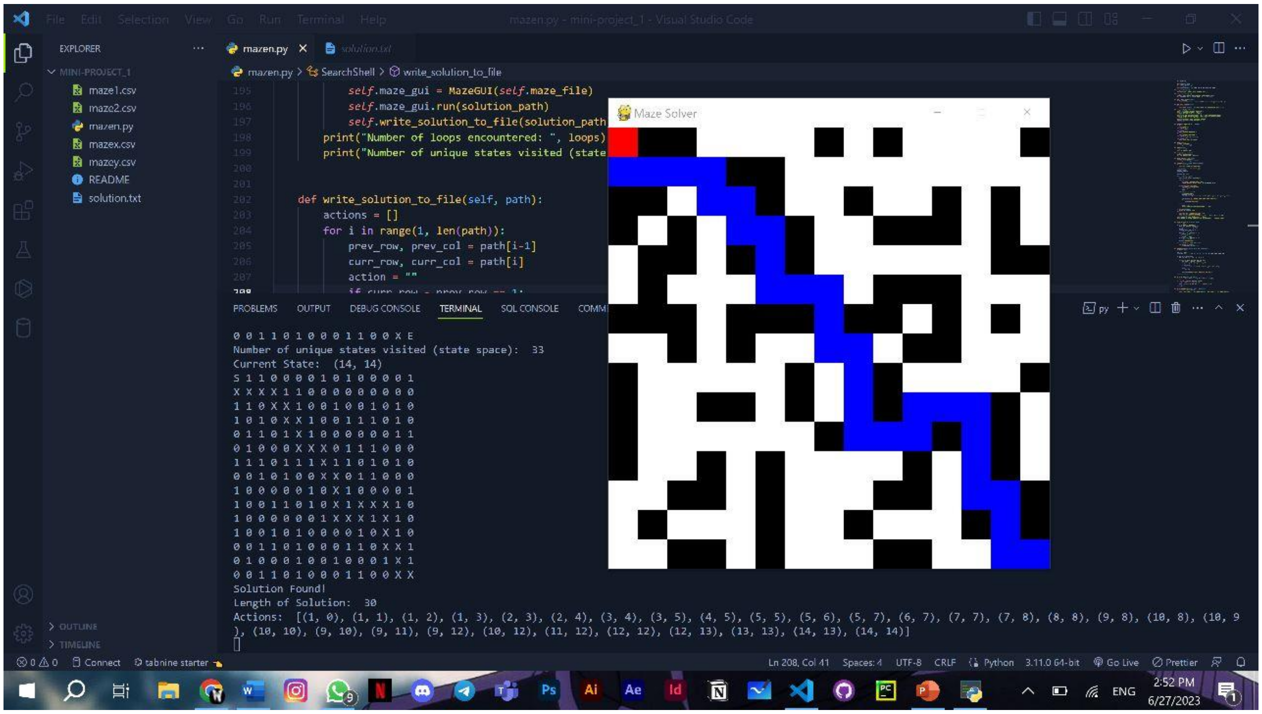
The search algorithms are implemented using different types of queue structures: BFS uses a FIFO Queue, DFS uses a LIFO Queue (Stack), and GBFS and A* Search use a Priority Queue.

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

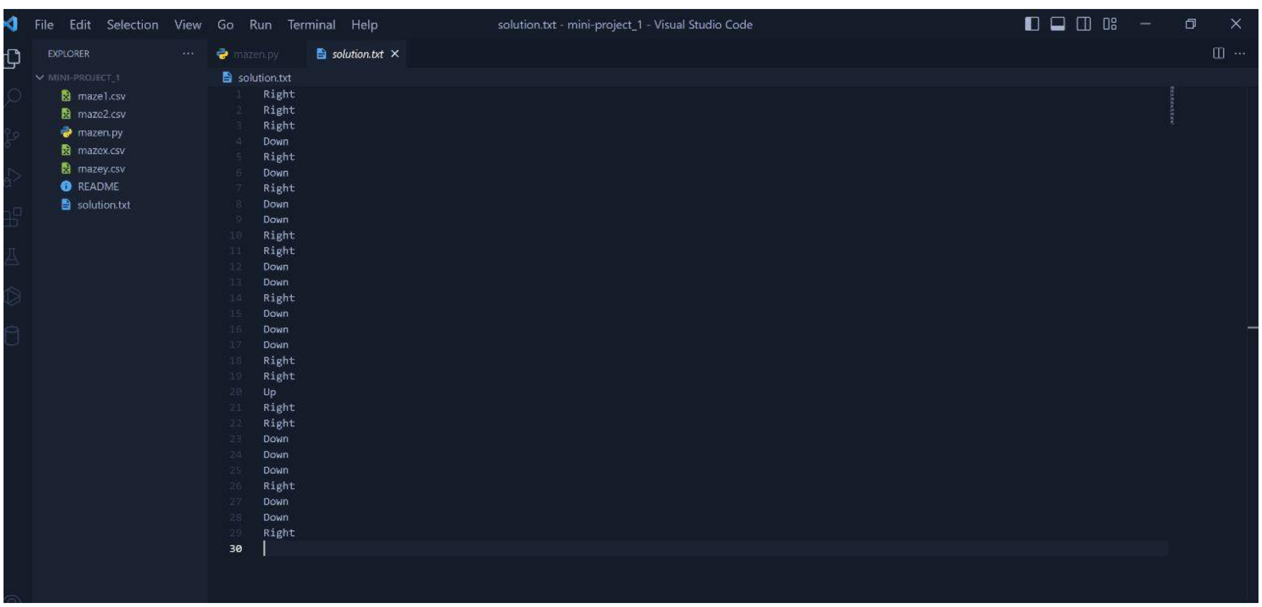
Main Program

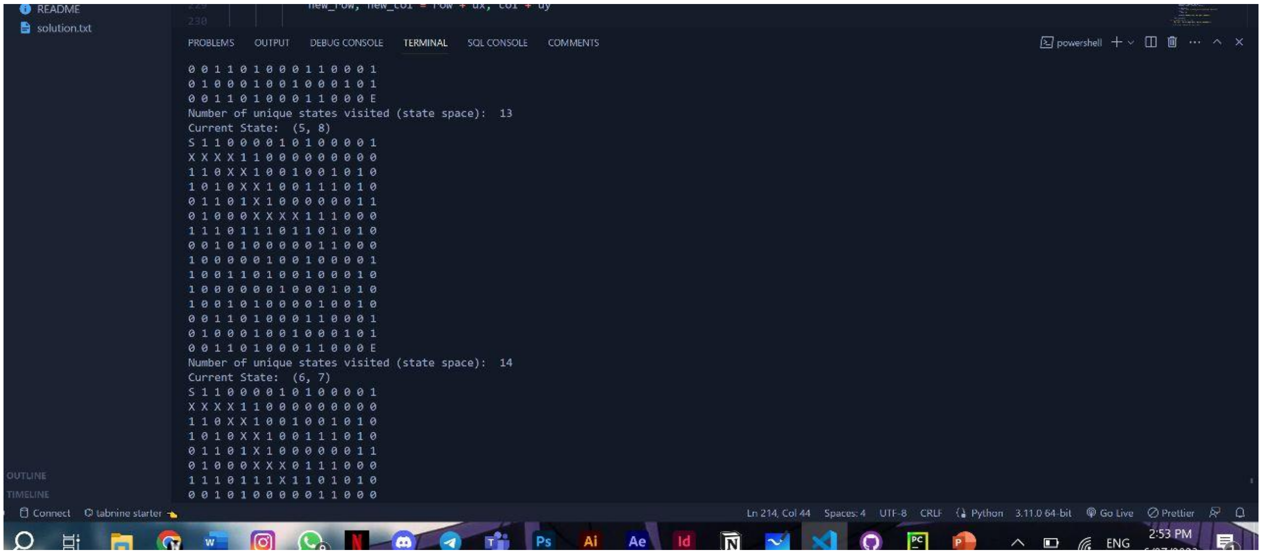
This section creates an instance of the SearchShell class and starts it. It also runs a separate thread to manage verbose status.

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.




```
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
```





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

<u>And the imported libraries' methods and classes used are:</u>

- 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