

**University of Science - VNUHCM
Faculty of Information Technology**

---o0o---



Project 1: Searching DELIVERY SYSTEM

Course : Introduction to Artificial Intelligence

**Instructors : Nguyen Ngoc Thao
Nguyen Tran Duy Minh
Nguyen Thanh Tinh**

Class : 22CLC06

**Students : Le Duy Anh – 22127012
Huynh Cao Tuan Kiet – 22127219
Ly Dinh Minh Man – 22127255
Vo Nguyen Phuong Quynh - 22127360**

Ho Chi Minh City, July 2024

TABLES OF CONTENTS

TABLES OF CONTENTS.....	2
1 Information	4
1.1. About students:.....	4
a) Group's information	4
b) Member evaluation.....	4
1.2. About project.....	5
a) Project result:.....	5
b) Completion assessment.....	5
c) Tools	5
1.3. Folder tree.....	5
2 Algorithm Implementations	7
2.1. Level 1: Basic Level.....	7
a) BFS.....	7
b) DFS	7
c) UCS	8
d) GBFS.....	8
e) A*	9
2.2. Level 2: Time-limit Level	10
a) Idea:	10
b) Algorithm implementation	10
2.3. Level 3: Fuel & Time limit Level.....	11
a) Idea	11
b) Algorithm implementation	11
2.4. Level 4: Multi-agent Level.....	12
a) Idea	12
b) Environment.....	13

c) <i>Algorithm Implementation: Class BfsBot</i>	13
3 Test cases & Results	15
3.1. Test generation	15
a) <i>MapGenerator class</i>	15
b) <i>Test generate</i>	15
3.2. Results checking	16
a) <i>Level 1</i>	16
b) <i>Level 2</i>	19
c) <i>Level 3</i>	20
d) <i>Level 4</i>	21
4 GUI & Code base	22
4.1. GUI Visualize:	22
4.2. Code base:	25
a) <i>Folder structure:</i>	25
b) <i>Program flow:</i>	26
4.3. GUI implementation:	27
a) <i>Overview of the GUI Structure</i>	27
b) <i>GUI Structure</i>	27
c) <i>Detailed Components</i>	27
d) <i>Detailed Description of Components</i>	27
e) <i>Interaction and Navigation</i>	28
References	29

1

Information

1.1. About students:

a) Group's information

- Group: 03
- Class ID: 22CLC06

No.	Fullname	Students'ID
1	Lê Duy Anh	22127012
2	Huỳnh Cao Tuấn Kiệt	22127219
3	Lý Đình Minh Mẫn	22127255
4	Võ Nguyễn Phương Quỳnh	22127360

b) Member evaluation

No.	Student	ID	Contribution	Evaluation
1	Le Duy Anh	22127012	<ul style="list-style-type: none"> • Code base design • Level 4 implementation • GUI json system • Menu GUI 	100%
2	Huynh Cao Tuan Kiet	22127219	<ul style="list-style-type: none"> • GUI Visualize system • Data generation • Video demo 	100%
3	Ly Dinh Minh Man	22127255	<ul style="list-style-type: none"> • Level 3 implementation • Level 4 implementation • Report 	100%
4	Vo Nguyen Phuong Quynh	22127360	<ul style="list-style-type: none"> • Level 1 implementation • Level 2 implementation • Report 	100%

1.2. About project

a) Project result:

- Source code: check out our source code on [GitHub](#).
- Demo video: [Youtube](#).

b) Completion assessment

No	Details	Elavalution
1	Finish Level 1 successfully.	15%
2	Finish Level 2 successfully.	15%
3	Finish Level 3 successfully.	15%
4	Finish Level 4 successfully.	10%
5	Graphical User Interface (GUI).	10%
6	Generate at least 5 test cases for all algorithm with different attributes. Describe them in the experiment section.	15%
7	Report algorithms, experiment with some reflection or comments.	20%
TOTAL		100%

c) Tools

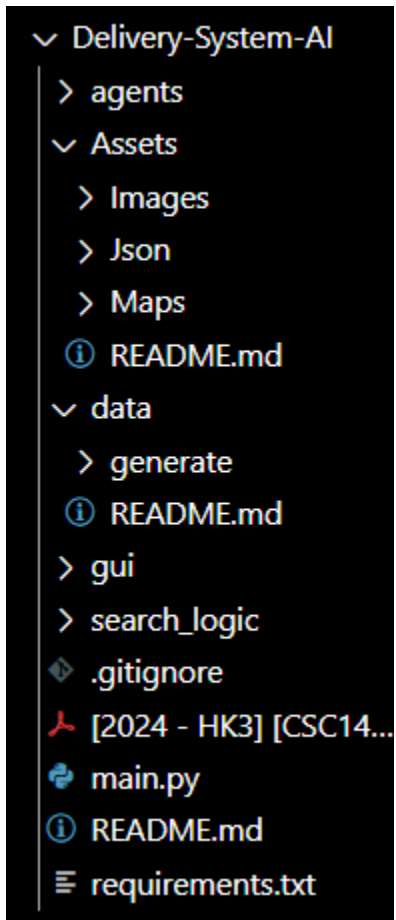
- Language: Python 3.11
- IDE: Visual Studio Code
- Support libraries:
 - `queue.PriorityQueue`: Provides a priority queue data structure, essential for certain graph search algorithms like UCS, A*.
 - `pygame`, `json`, `PIL.image`: Provides environment to implementate a graphical interface to show the process step by step.

1.3. Folder tree

Deliver-System-AI: include main code file and folders:

- agent
- Assets:
 - Images: include images for GUI display.
 - Json: include json files for GUI display.

- Maps: include input txt files as map data.
- data: data generation folder
 - generate: include map generation python program.
- gui: include python code files for GUI display.
- search_logic: include python code files of search strategies.



2

Algorithm Implementations

2.1. Level 1: Basic Level

Find the path from S to G. Include 5 algorithms: BFS, DFS, UCS, GBFS and A*

a) *BFS*

“Expand the root node first, then all the neighbors of the root node are next, then their neighbors, and so on.” It uses a **FIFO queue** frontier to manage the nodes to be explored, ensuring that all nodes at the present depth level are explored before moving on to nodes at the next depth level.

Algorithm implementation:

- Initialization:
 - queue: frontier, a FIFO queue of tuple of node, path from S to node, (initialized with the start node).
 - visited: A list to keep track of whether a node is visited.
- BFS loop:
 - Dequeue Node: The first node in the frontier queue is dequeued for exploration.
 - Goal check: If current node is goal, return path and break out of the loop.
 - Check Visited: If the node has already been visited, continue to the next iteration. Else: Mark the node as visited.
 - Explore neighbors: For each neighbor of the current node, if the neighbor is not visited, then add the neighbor to the queue.

b) *DFS*

Depth-First Search (DFS) algorithm that explores **as far as possible** along each branch before backtracking: “Always expand the deepest node in the frontier first.” Using a **LIFO stack** to manage the nodes to be explored, ensuring that the deepest nodes are explored first.

Algorithm implementation:

- Initialization:
 - stack: frontier, a LIFO stack of tuple of node, path from S to node,

(initialized with the start node).

- visited: A list to keep track of whether a node is visited.
- DFS loop:
 - Pop Node: The first node in the frontier stack is dequeued for exploration.
 - Goal check: If current node is goal, return path and break out of the loop.
 - Check Visited: If the node has already been visited, continue to the next iteration. Else: Mark the node as visited.
 - Explore neighbors: For each neighbor of the current node, if the neighbor is not visited, then add the neighbor to the stack.

c) UCS

Uniform-Cost Search (UCS) is a graph traversal algorithm that extends Breadth-First Search by **considering edge costs**: “UCS resembles the mechanism of **Dijkstra's algorithm**.” It uses a **priority queue** to explore the node with the lowest path cost from the start node, ensuring that the shortest path is found in terms of cost.

Algorithm implementation:

- Initialization:
 - queue: frontier, a priority queue with priority value is path cost from start to node, and tuple of node, path from S to node, (initialized with the start node).
 - visited: A list to keep track of whether a node is visited.
- UCS loop:
 - Get Node: The node with the lowest path cost is dequeued from the priority queue.
 - Goal check: If current node is goal, return path and break out of the loop.
 - Check Visited: If the node has already been visited, continue to the next iteration. Else: Mark the node as visited.
 - Explore neighbors: For each neighbor of the current node, if the neighbor is not visited, then add the neighbor to the queue with priority value is **cost + weight (current, neighbor)**.

d) GBFS

Greedy Best-First Search (GBFS) algorithm that selects the node that appears to be closest to the goal based on **heuristic value**: “Expand first the node with the **lowest**

$h(n)$ value – the node that appears to be closest to the goal.” It uses a **priority queue** to explore nodes with the lowest heuristic value (Manhattan distance) first, aiming to reach the goal as quickly as possible.

Algorithm implementation:

- Initialization:
 - queue: frontier, a priority queue with priority value is weight from start to node, and tuple of node, path from S to node, (initialized with the start node).
 - visited: A list to keep track of whether a node is visited.
- GBFS loop:
 - Get Node: The node with the lowest heuristic value is dequeued from the priority queue.
 - Goal check: If current node is goal, return path and break out of the loop.
 - Check Visited: If the node has already been visited, continue to the next iteration. Else: Mark the node as visited.
 - Explore neighbors: For each neighbor of the current node, if the neighbor is not visited, then add the neighbor to the queue with priority value is *heuristic of neighbor*.

e) A*

A* (A-star) search algorithm extends UCS algorithm by **using heuristics** to guide the search. It combines the cost to reach the node and an estimate of the cost from the node to the goal (heuristic value - Manhattan distance) to prioritize nodes in the frontier.

Algorithm implementation:

- Initialization:
 - queue: frontier, a priority queue with priority value is weight from start to node, and tuple of node, path from S to node, (initialized with the start node).
 - visited: A list to keep track of whether a node is visited.
- A* loop:
 - Get Node: The node with the lowest path cost is dequeued from the priority queue.

- Goal check: If current node is goal, return path and break out of the loop.
- Check Visited: If the node has already been visited, continue to the next iteration. Else: Mark the node as visited.
- Explore neighbors: For each neighbor of the current node, if the neighbor is not visited, then add the neighbor to the queue with priority value is $cost - heuristic(current) + weight(current, neighbor) + heuristic(neighbor)$.

2.2. Level 2: Time-limit Level

a) Idea:

To find the most efficient path within a given time constraint, we employ a modified Uniform Cost Search (UCS) algorithm.

This enhanced version prioritizes nodes based on:

- Position
- Time remaining

A state in this context is fully defined by its current position and the amount of time left. To prevent redundant computations, the algorithm explores a node only if it represents a unique combination of position and remaining time.

b) Algorithm implementation

Initialization:

- Priority queue (pq): A frontier, priority queue with a tuple containing cost, time, current node, and path as priority values. Initilized with $(0, time\ limit, start, [start])$.
- marked: A set to keep track of whether a node with time value is in frontier.

UCS loop

- Dequeue Node: The first node in the frontier queue is dequeued for exploration, time as the time left.
- Time check: If time is over ($time < 0$), skip and move to the next node in frontier.
- Goal check: If current node is goal, return path and break out of the loop.
- Explore neighbors: For each neighbor of the current node:
 - Calculate the time left values.
 - Mark the neighbor with new time values and add it to the priority queue.

2.3. Level 3: Fuel & Time limit Level

a) Idea

To determine the optimal path while considering time and fuel constraints, an enhanced version of the Uniform Cost Search (UCS) algorithm is utilized. This algorithm prioritizes nodes based on a combination of path cost, remaining fuel, and available time.

A state in this context is defined by its:

- Position
- Remaining time
- Remaining fuel

To avoid redundant exploration, the algorithm expands a node only if a state with the same position, fuel level, and time has not been previously encountered.

b) Algorithm implementation

Initialization:

- Priority queue (pq): A frontier, priority queue with a tuple containing cost, gas, time, current node, and path as priority values. Initialized with (0, gas, time, start, [start]).
- marked: A set to keep track of whether a node with specific gas and time values is in frontier.

UCS loop

- Dequeue Node: The first node in the frontier queue is dequeued for exploration, time as the time left.
- Time check: If time is over (time < 0), skip and move to the next node in frontier.
- Goal check: If current node is goal, return path and break out of the loop.
- Continue conditions: If the path length exceeds the product of the graph's rows and columns or if the time is negative, continue to the next node.
- Explore neighbors: For each neighbor of the current node:
 - Calculate the new gas and time values.
 - If the neighbor is a gas station (indicated by a negative `ctime`), refill the gas and adjust the time accordingly.
 - If the new gas value is negative or the neighbor with the new gas and

time values has already been marked, continue to the next neighbor.

- Mark the neighbor with the new gas and time values and add it to the priority queue.

2.4. Level 4: Multi-agent Level

a) Idea

Overall Strategy

- The agent will initially plan its path without considering other agents. This path is determined using a Uniform Cost Search (UCS) algorithm based on the agent's current state (time, gas, position). Once the path is determined, the agent will execute it, adapting its behavior based on the actions of other agents.

Path Planning

- State Representation: Each possible state of the agent is defined by a tuple of:
 - **(Current Time, Remaining Gas, Current Position).**
- Pathfinding Algorithm: Uniform Cost Search (UCS) is used to find the path with the least cost (in terms of cells traversed) from the current state to the target position, while considering the constraints of time and gas.

Agent Interactions

- **Scenario 1: Next Move Occupied by Another Agent**
 - Wait Time Exists: The agent will randomly respond by either:
 - Assuming the other agent is waiting and remaining in the current position.
 - Assuming the other agent is blocked and finding an alternative move using Breadth-First Search (BFS).
 - No Wait Time: The agent will randomly respond by either:
 - Assuming the other agent is blocked and finding an alternative path using BFS.
 - Remaining in the current position to wait for the other agent to move.
- **Scenario 2: Next Move Not Occupied by Another Agent** - The agent performs the planned move according to the UCS path.
- **Scenario 3: No Valid Next Move** - The agent randomly selects a neighboring

cell or opts to stay in its current position as a potential solution to the blockage (if there's any blockage on the map).

b) Environment

Initialization

- Load map data including grid, starting positions, time limit, and fuel.
- Create individual agents with their initial conditions.
- Initialize files for tracking agent and goal positions.

Simulation Loop

- Continuously iterate until the time limit expires or a primary agent reaches its goal.
- For each agent:
 - Handle waiting periods if applicable.
 - Determine the agent's next move based on its bot's strategy.
 - Update the agent's position and internal state on the map.
 - If the agent is a sub-agent and reaches its goal, update its state and goal position.
 - Modify the map's state to reflect agent actions.

Output

Generate a JSON file encapsulating simulation results, including the map, agent trajectories, final goals, and initial setup.

c) Algorithm Implementation: Class BfsBot

This class is inherited from the BotBase class and represents a bot that uses Breadth-First Search (BFS) to navigate in an environment.

init(self, agent_list, map, time, gas):

- Initializes the bot object with:
 - agent_list: A list of agents containing information like position and goal.
 - map: A copy of the environment map.
 - time: The initial time available to the agent.
 - gas: The initial gas available to the agent.

goals:

This is a list that stores the goal positions for each agent extracted from agent_list.

`bfsToGoal(self, map, state, current_pos):`

- This function is responsible for finding the next move towards the goal using BFS. It takes the following arguments:
 - `map`: The environment map.
 - `state`: A dictionary containing information about the current agent state like position, time, and gas.
 - `current_pos`: The current position of the agent.
- It performs a BFS search starting from the current position and explores neighboring cells.
- It prioritizes cells with lower movement cost, remaining time, and gas cost.
- It keeps track of visited states to avoid revisiting the same configuration.
- If the goal is found, it returns the first move in the path towards the goal.
- Otherwise, it returns `None`.

`get_move(self, map, state, current_pos):`

- This function determines the next move for the agent. It takes the same arguments as `bfsToGoal`.
- It first tries to find the next move using the pre-computed path from `bfsToGoal`.
- If no path is found or the next move is already occupied by another agent (except the agent's current position), it considers alternative strategies:
 - *Random exploration (with exploration rate):*
 - If the agent has no time or gas remaining, it returns `None`.
 - Otherwise, it checks a random number between 0 and 100.
 - If the random number is greater than or equal to a predefined exploration rate (`RATE`), it returns `None` (assuming the agent waits).
 - If the random number is less than the exploration rate, it moves to a random valid neighboring cell.
 - *Move towards a closer cell (if exploration fails):*
 - It prioritizes moving towards the neighboring cell with the smallest distance to the goal using a breadth-first search (similar to `bfsToGoal` but simpler).
 - It returns the first move in the path towards the closer cell.

3

Test cases & Results

3.1. Test generation

The generated map includes obstacles, a starting point, and multiple goal points, representing customers. The map ensures connectivity and favoring of corner positions when required.

a) MapGenerator class

The MapGenerator class generates a map grid with the following key features:

Parameters

- row: Number of rows in the grid.
- col: Number of columns in the grid.
- clusterSize: Maximum size of obstacle clusters.
- numClusters: Number of obstacle clusters.
- favorCorners: Boolean to decide if starting positions should favor corners.

Methods

- generateMap(): Generates the map ensuring all spaces are connected.
- _addCluster(): Adds clusters of obstacles to the map.
- _isValid(): Checks if a cell is valid (within bounds and empty).
- _neighbors(): Generates neighboring cells.
- _checkOpponent(): Checks for opponents near a given cell.
- _favorCornerPositions(): Returns a corner position if favoring corners.
- _randomPosition(): Generates a random position, optionally favoring corners.
- addStartGoal(): Adds starting and goal points ensuring connectivity.
- setFavorCorners(): Sets the favoring of corner positions.
- _bfs(): Performs Breadth-First Search to find paths between points.
- _markReachable(): Marks reachable cells from a given cell.
- isConnected(): Checks if all empty cells are connected.
- addPlayers(): Adds multiple players with starting and goal positions.

b) Test generate

For each level (1 to 4), we have generated 6 maps:

- 5 square maps with increasing size: 5*5, 10*10, 15*15, 20*20, and 25*25

cells.

- 1 rectangle map: 28*40 cells.
- Time limit and fuel tank: ups to map data.

3.2. Results checking

a) Level 1

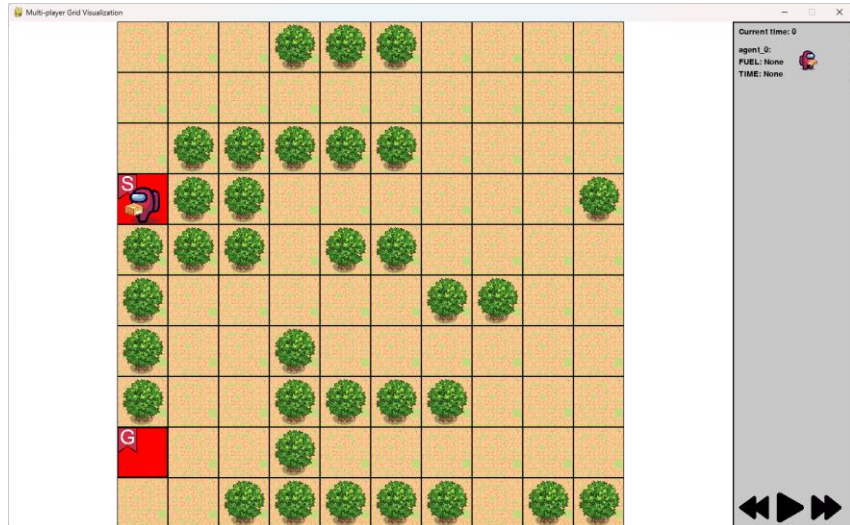
Map 2:

```

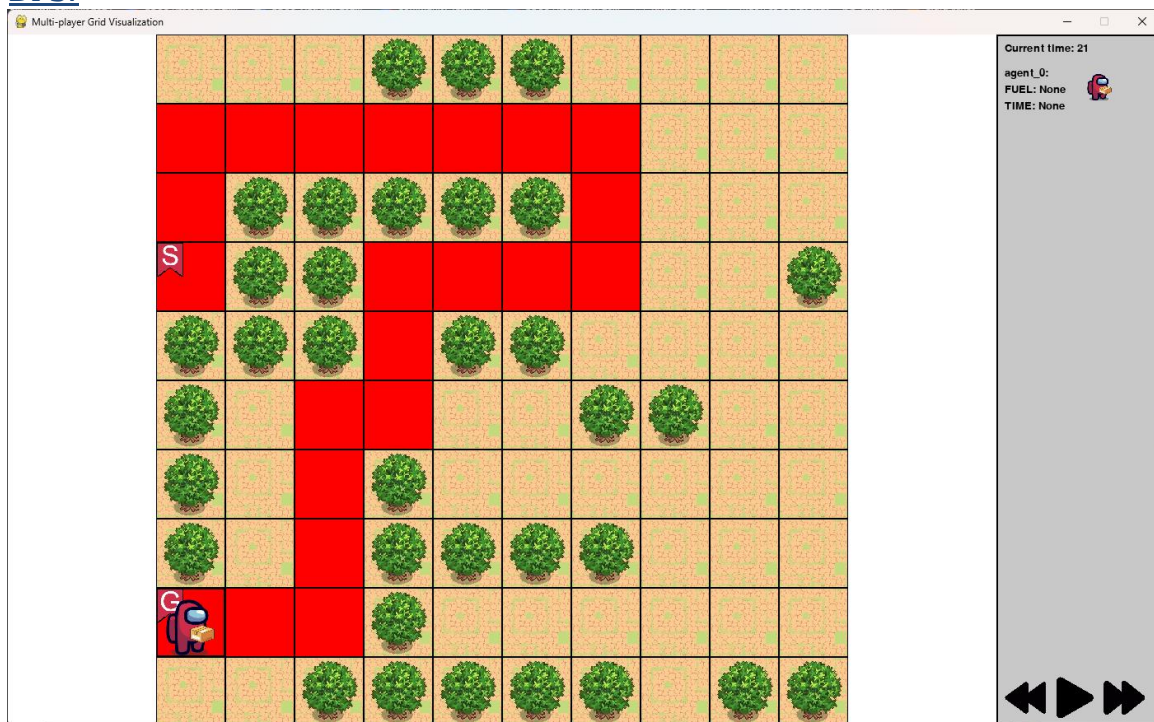
10 10 100 100
0 0 0 -1 -1 -1 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 -1 -1 -1 -1 -1 0 0 0 0
S -1 -1 0 0 0 0 0 0 -1
-1 -1 -1 0 -1 -1 0 0 0 0
-1 0 0 0 0 0 -1 -1 0 0
-1 0 0 -1 0 0 0 0 0 0
-1 0 0 -1 -1 -1 -1 0 0 0
G 0 0 -1 0 0 0 0 0 0
0 0 -1 -1 -1 -1 -1 0 -1 -1

```

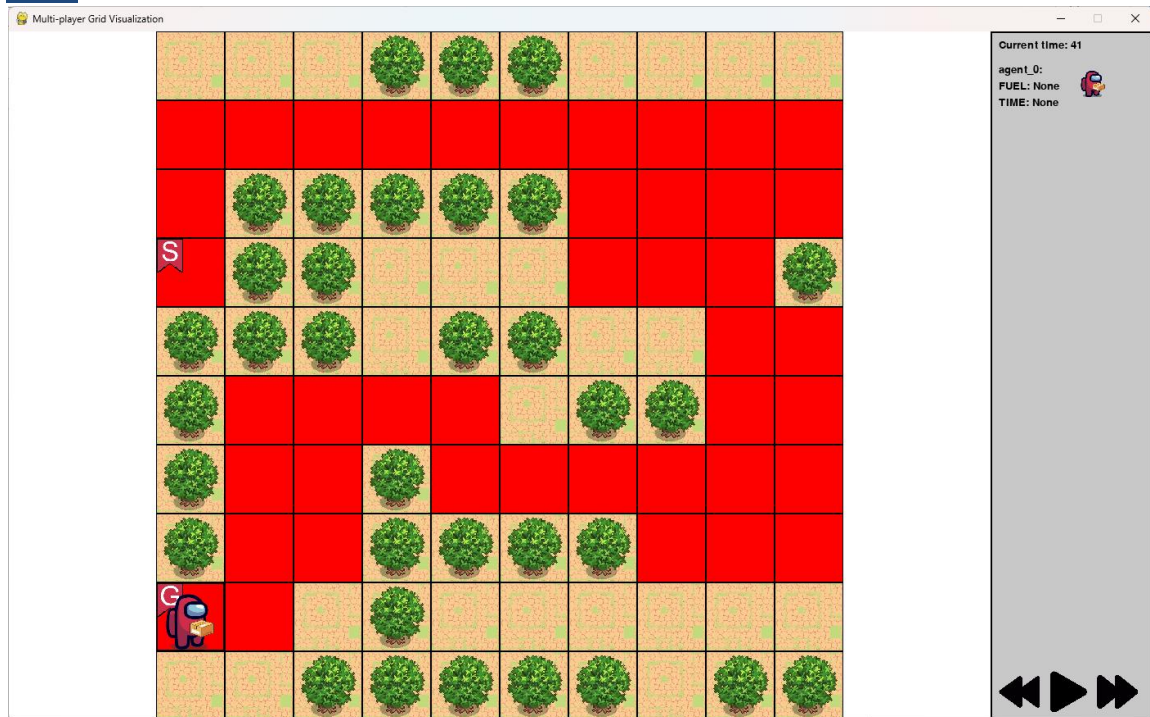
- Size: 10 x 10
- Start: (3, 0)
- Goal: (8, 0)



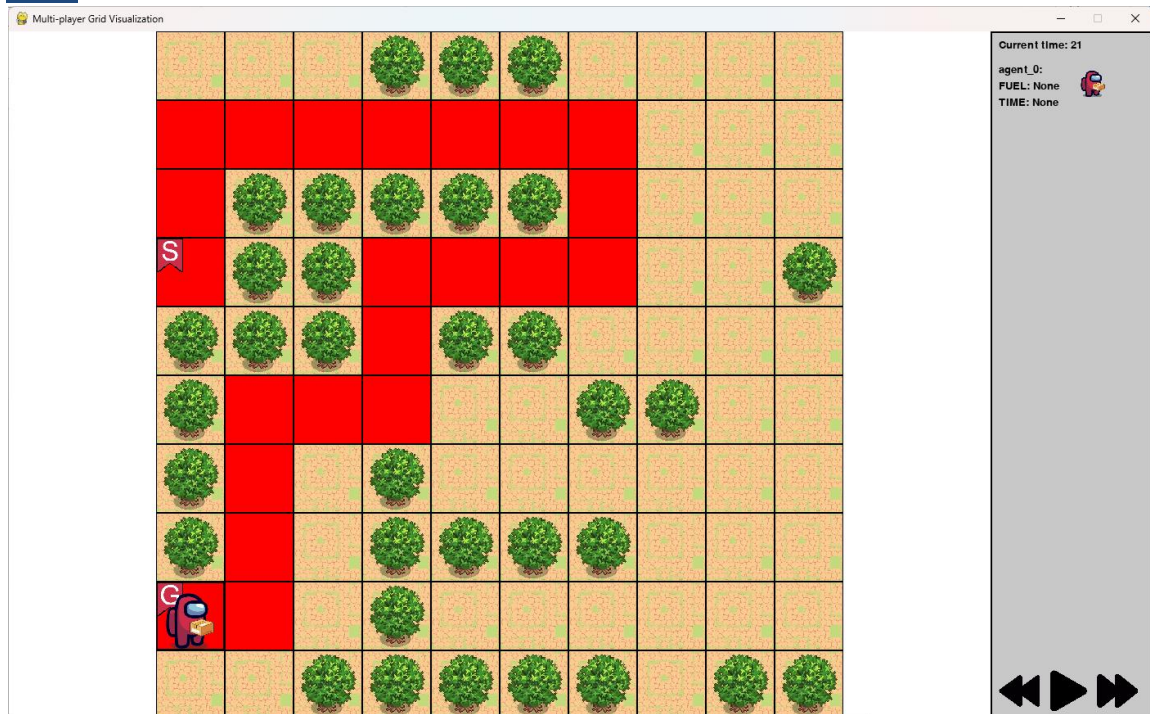
BFS:

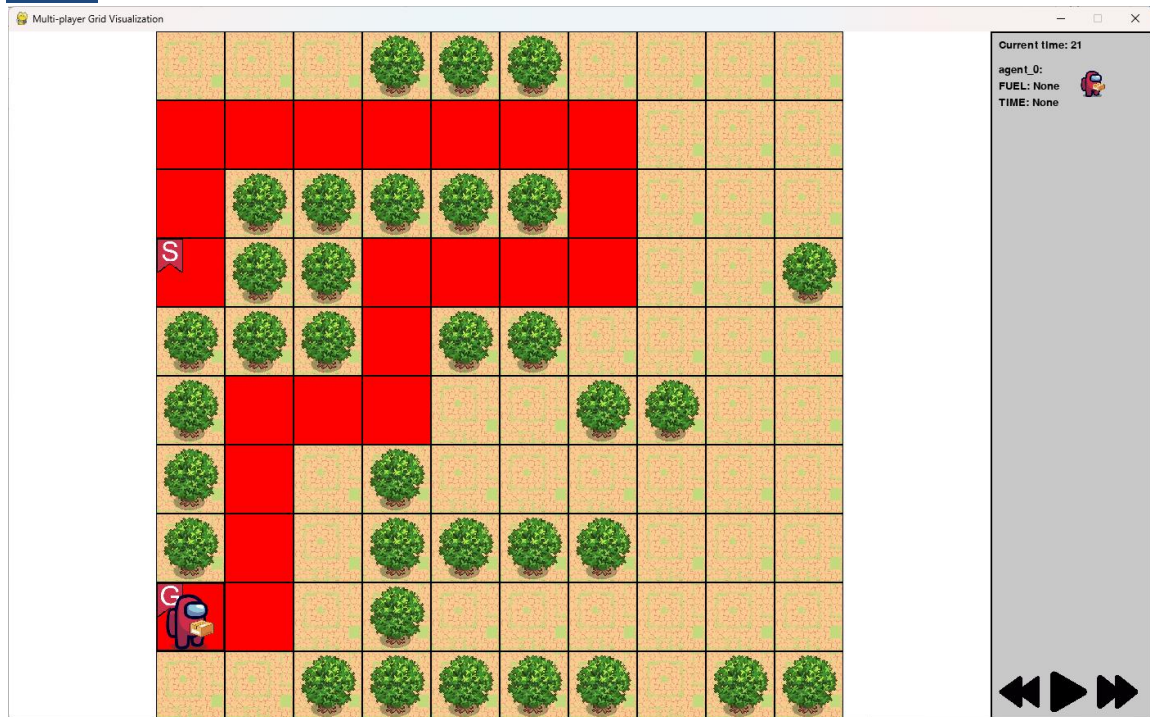
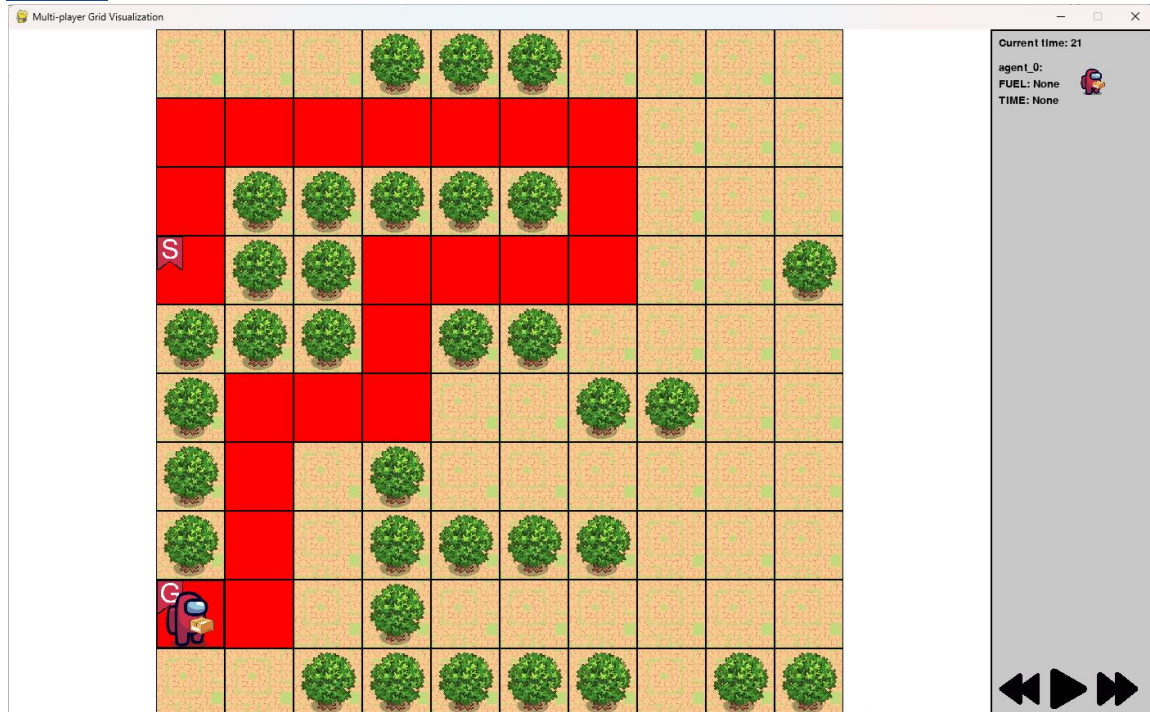


DFS:



UCS:



GBFS:A star:

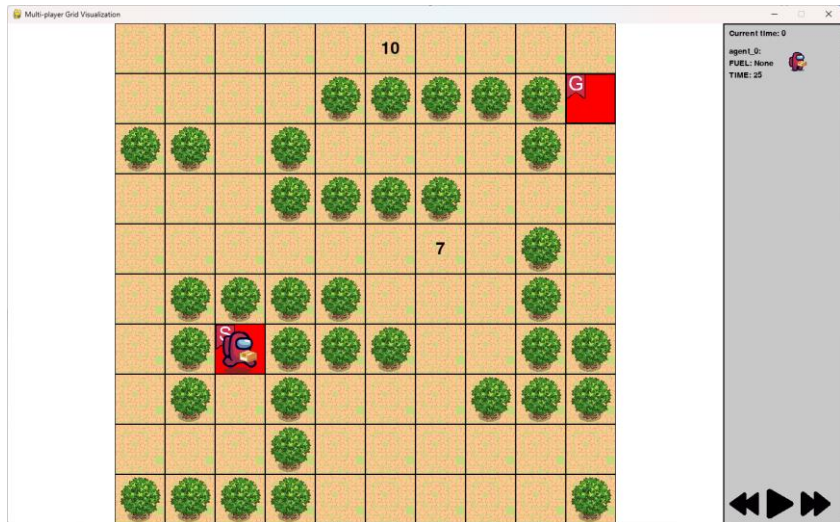
b) Level 2

Map 2:

```

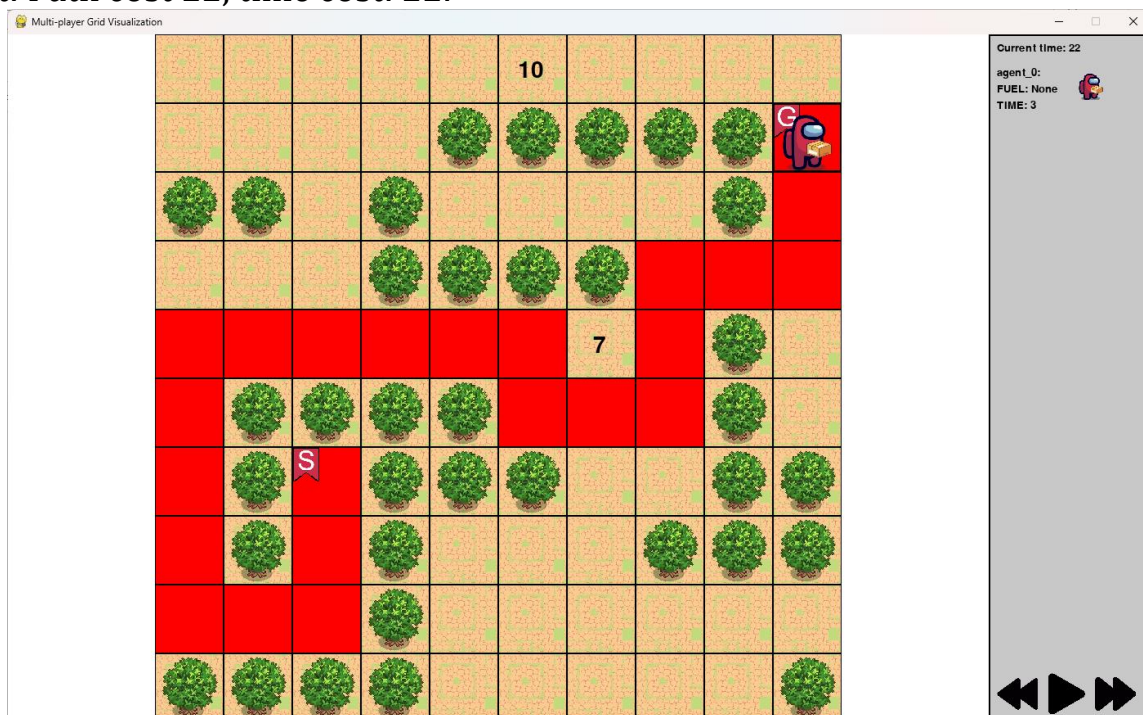
10 10 25 25
0 0 0 0 0 10 0 0 0 0
0 0 0 0 -1 -1 -1 -1 -1 G
-1 -1 0 -1 0 0 0 0 -1 0
0 0 0 -1 -1 -1 -1 0 0 0
0 0 0 0 0 0 7 0 -1 0
0 -1 -1 -1 -1 0 0 0 -1 0
0 -1 5 -1 -1 -1 0 0 -1 -1
0 -1 0 -1 0 0 0 -1 -1 -1
0 0 0 -1 0 0 0 0 0 0
-1 -1 -1 -1 0 0 0 0 0 -1

```



- Size: 10 x 10
- Start: (6, 2)
- Goal: (1, 9)
- Time limit: 25

Result: Path cost 22, time cost: 22.



c) Level 3

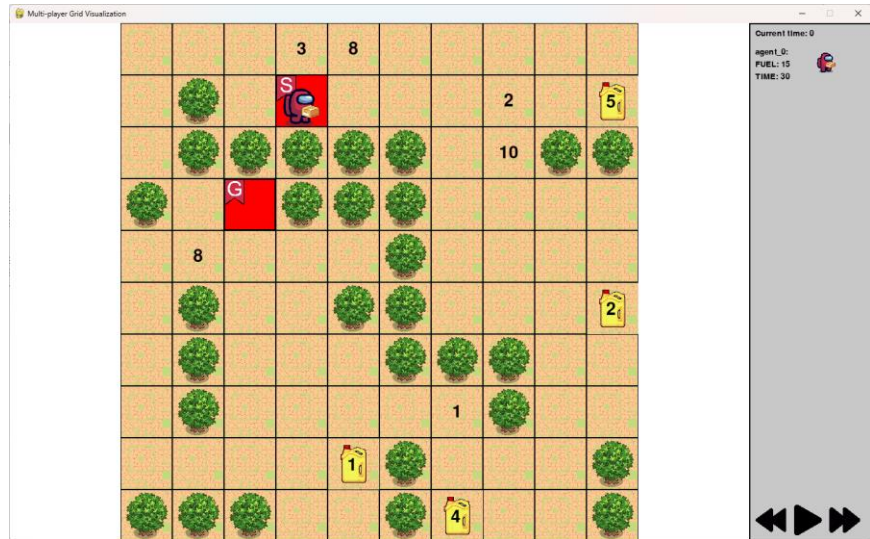
Map 2:

```

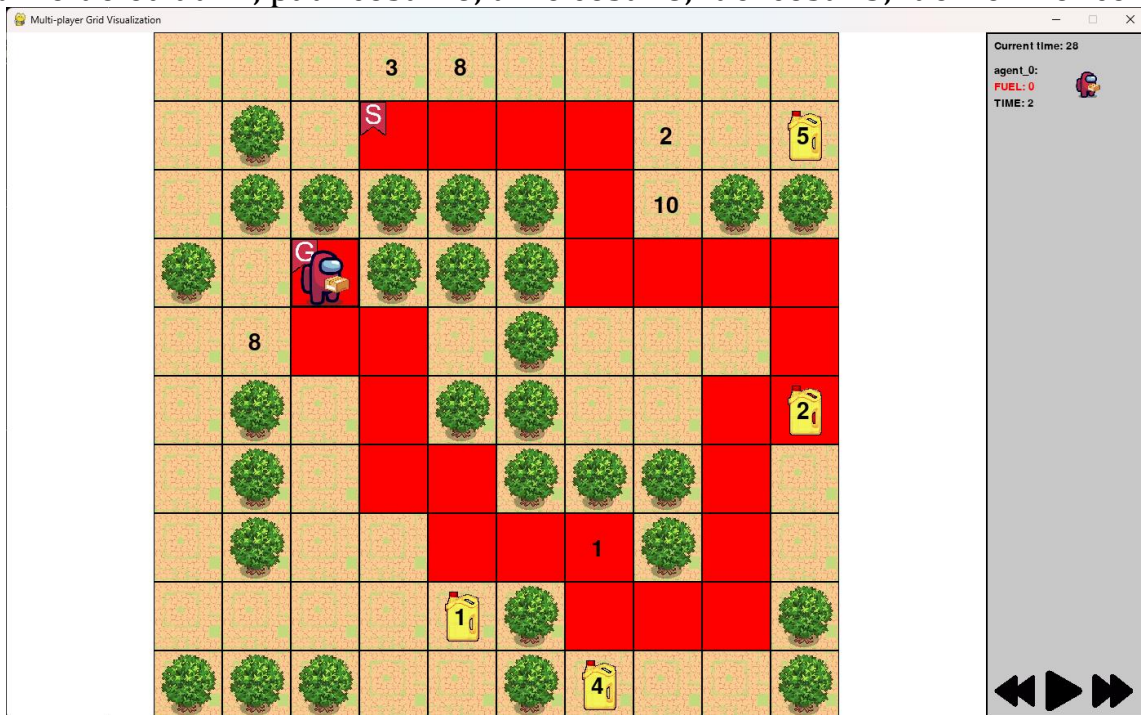
10 10 30 15
0 0 0 3 8 0 0 0 0 0
0 -1 0 S 0 0 0 2 0 F5
0 -1 -1 -1 -1 -1 0 10 -1 -1
-1 0 G -1 -1 -1 0 0 0 0
0 8 0 0 0 -1 0 0 0 0
0 -1 0 0 -1 -1 0 0 0 F2
0 -1 0 0 0 -1 -1 -1 0 0
0 -1 0 0 0 0 1 -1 0 0
0 0 0 0 F1 -1 0 0 0 -1
-1 -1 -1 0 0 -1 F4 0 0 -1

```

- Size: 10 x 10
- Start: (1, 3)
- Goal: (3, 2)
- Time limit: 30
- Fuel tank: 15
- Fuel station: F5(1, 9); F2(5, 9); F1(8, 4); F4(9, 6)



Result: Refueled at F2, path cost: 26, time cost 28, fuel cost 25, fuel refill once.



d) Level 4

Map 2:

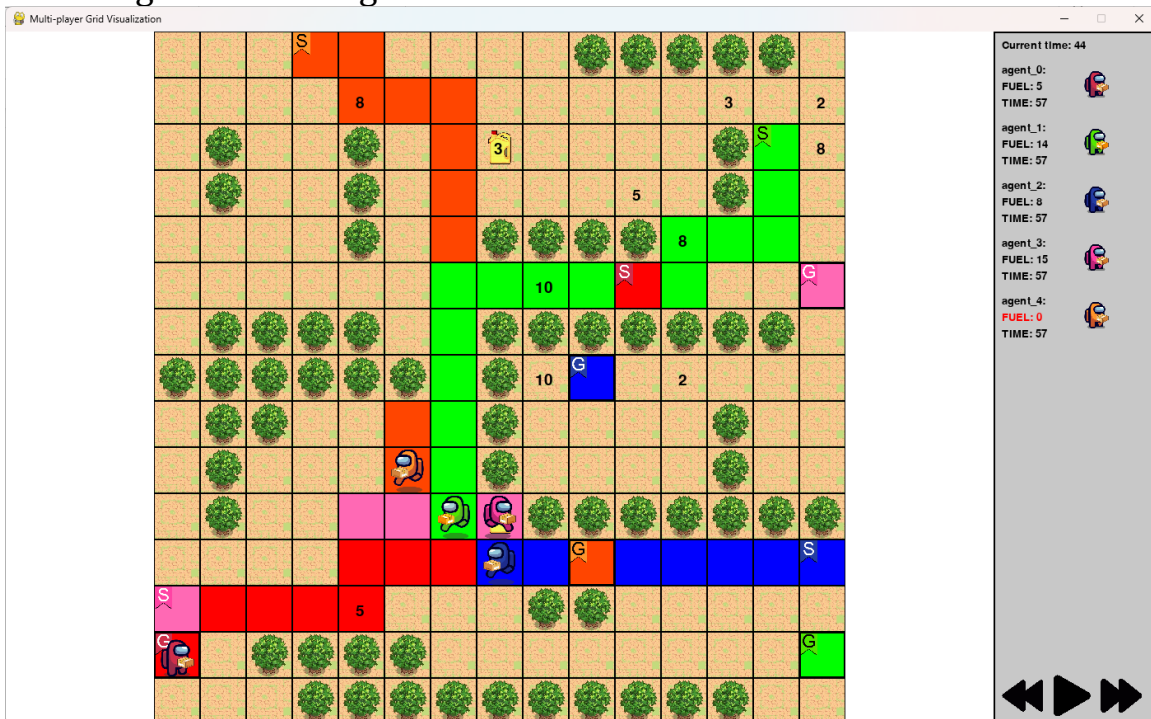
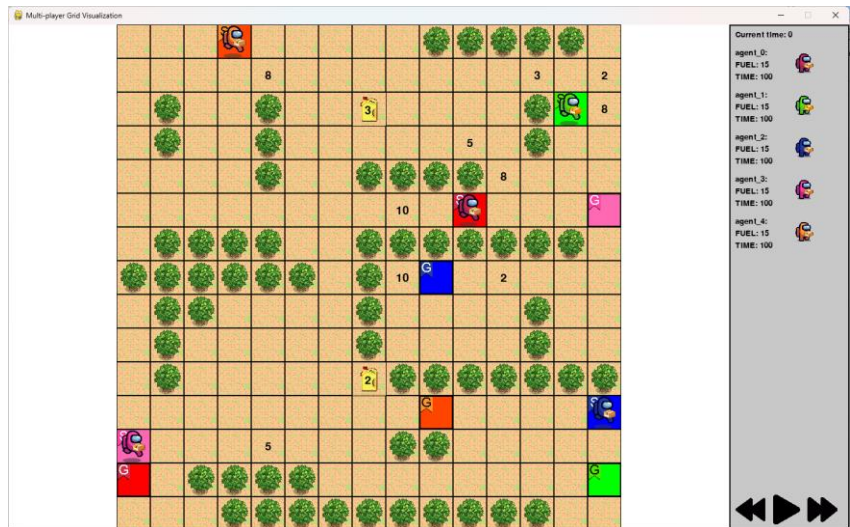
```

15 15 100 15
0 0 0 5 4 0 0 0 0 0 -1 -1 -1 -1 -1 0
0 0 0 0 8 0 0 0 0 0 0 0 3 0 2
0 -1 0 0 -1 0 0 F3 0 0 0 0 -1 S1 8
0 -1 0 0 -1 0 0 0 0 0 5 0 -1 0 0
0 0 0 0 -1 0 0 -1 -1 -1 -1 8 0 0 0
0 0 0 0 0 0 0 0 10 0 S 0 0 0 G3
0 -1 -1 -1 -1 0 0 -1 -1 -1 -1 -1 -1 0
-1 -1 -1 -1 -1 -1 0 -1 10 G2 0 2 0 0 0
0 -1 -1 0 0 0 0 -1 0 0 0 0 -1 0 0
0 -1 0 0 0 0 0 -1 0 0 0 0 -1 0 0
0 -1 0 0 0 0 0 F2 -1 -1 -1 -1 -1 -1 -1
0 0 0 0 0 0 0 0 0 G4 0 0 0 0 S2
S3 0 0 0 5 0 0 0 -1 -1 0 0 0 0 0
G 0 -1 -1 -1 -1 0 0 0 0 0 0 0 0 G1
0 0 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 0

```

- Time limit: 100
- Fuel tank: 15
- Gas station: F3(2, 7), F2(10, 7)
- Main agent (agent_0): Start(5, 10) – Goal(11, 9)
- 4 side agents

Result: Main agent arrive to goal:

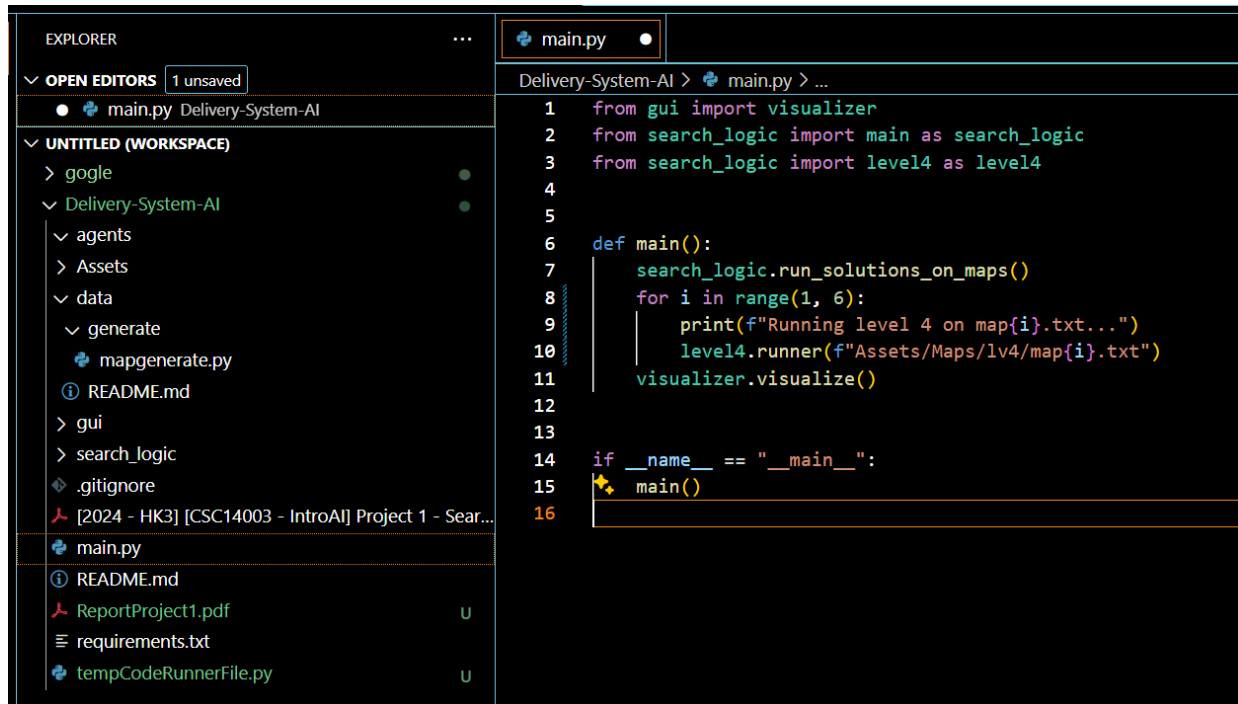


4

GUI & Code base

4.1. GUI Visualize:

To start running our system, user can call run command at python file `main.py` in the main folder:



```
1 from gui import visualizer
2 from search_logic import main as search_logic
3 from search_logic import level4 as level4
4
5
6 def main():
7     search_logic.run_solutions_on_maps()
8     for i in range(1, 6):
9         print(f"Running level 4 on map{i}.txt...")
10        level4.runner(f"Assets/Maps/lv4/map{i}.txt")
11    visualizer.visualize()
12
13
14 if __name__ == "__main__":
15     main()
16
```

After that, visualize system will pop out.



Menu screen

This is our homepage, as there you can choose the Level to visualize.

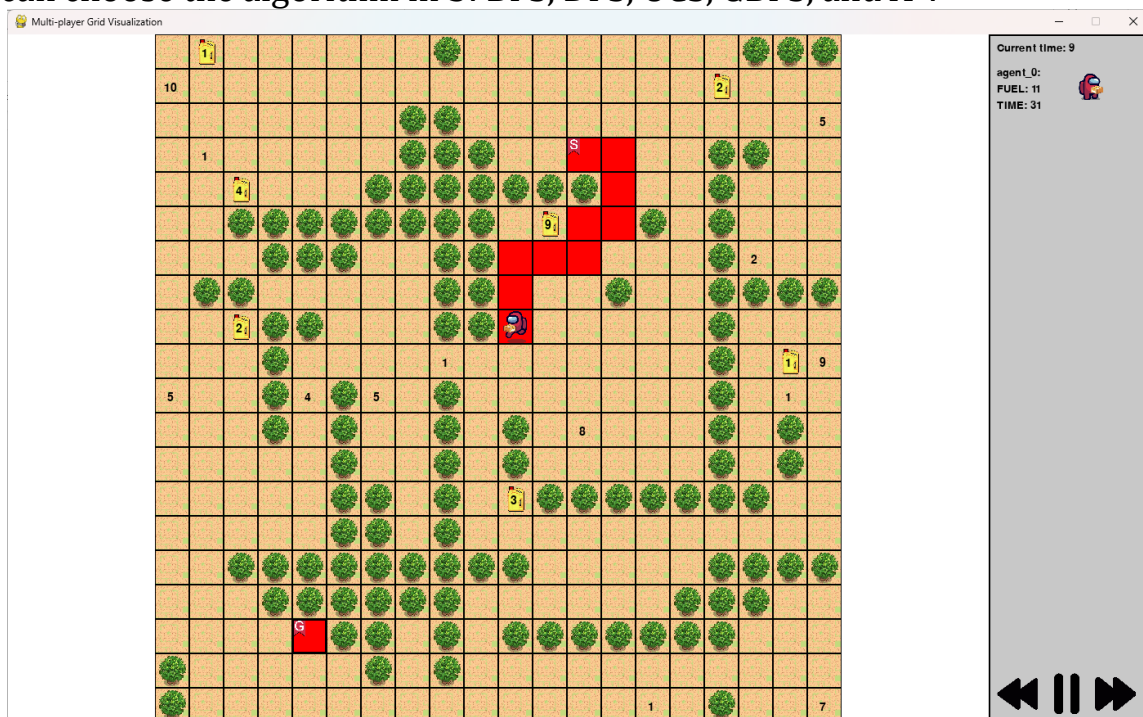


Level 1 screen



Level 2, 3, and 4 screen

In the Level screen, system ask users to choose map to visualize. By the way, in Level 1, we can choose the algorithm in 5: BFS, DFS, UCS, GBFS, and A*.



Visualize screen

After choosing map, visualize screen starts to display the path from start to goal. To get back, press **Esc** to escape to the Menu screen.



The screenshot shows a side bar with the following information:

Current time: 14	
agent_0:	
FUEL: 11	
TIME: 87	
agent_1:	
FUEL: 9	
TIME: 87	
agent_2:	
FUEL: 11	
TIME: 87	
agent_3:	
FUEL: 7	
TIME: 87	
agent_4:	
FUEL: 10	
TIME: 87	

At the side bar:

- User can track time, fuel of each agent.
- You can navigate the visualization by buttons in left conner.



4.2. Code base:

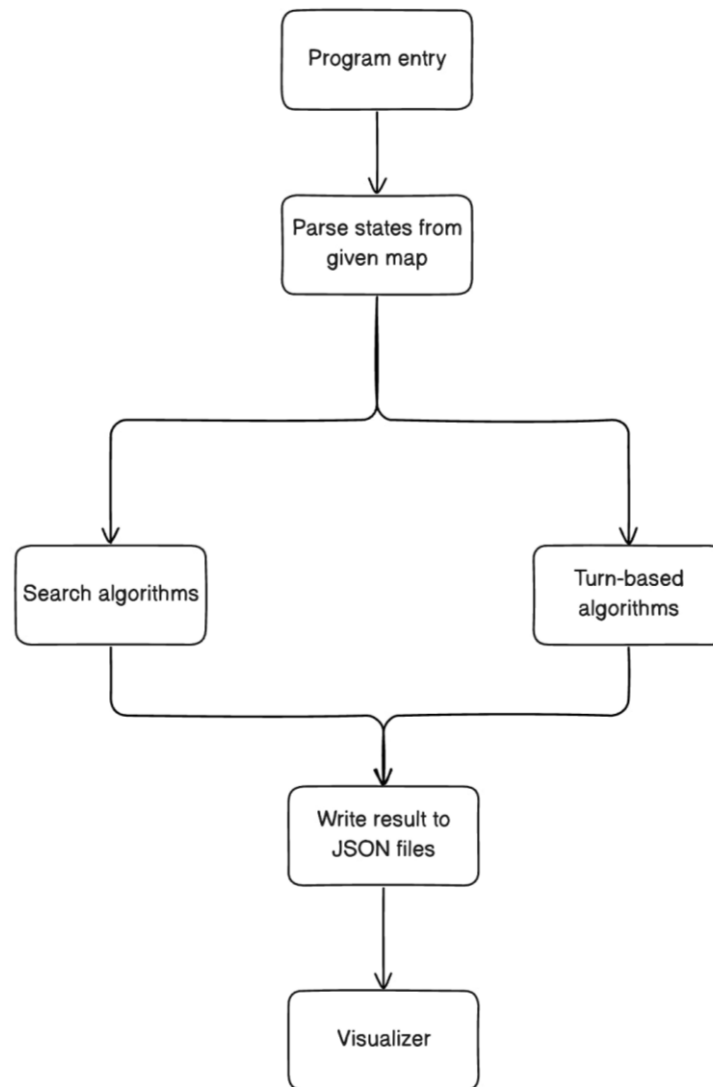
a) Folder structure:

All the core logic of our project is placed inside `search_logic` folder. For level 1, 2, and 3, we have a base `SolutionBase` that contains necessary properties and methods for a solution to find result for any given map.

In level 4, we created a `BotBase` that supports creating bots easily. New bot can be added with respect to methods of its base so that it can be integrated into the function flow seamlessly.

There are also some utility functions that help working with inputs and outputs.

With these functions, the development process went smoothly, and we were able to implement new algorithms faster.

b) Program flow:

Program flow

When initializing the project, we considered several critical components, including the main system for running algorithms, different ways to visualize the result for better observations. After some discussions, we concluded that it is best to separate the two tasks, as it is better for them to transfer data through files than through in-memory.

This design makes it easy to implement a visualizer as we only need to parse necessary content from given JSON source files, then display it with nice looking UI for end users.

4.3. GUI implementation:

a) Overview of the GUI Structure

Layout: The application features a multi-window layout with a main window for visualizing grid-based multi-player movements and auxiliary windows for menu navigation and level selection

b) GUI Structure

Layout:

- **Main Window:** Displays the grid visualization and player interactions.
- **Sidebar:** Contains controls and player information.
- **Menu:** Allows level selection.
- **Level Selection:** Dropdown menus for file and algorithm selection.

c) Detailed Components

- **Main Window:**
 - Grid Visualization: Shows roads, obstacles, flags, fuel stations, and player positions.
 - Images and Colors: Uses images from specified folders and color codes for players.
- **Sidebar:**
 - Control Buttons: Previous, next, play, and pause buttons.
 - Player Information: Displays player fuel, time, and state.
- **Menu:**
 - Buttons: Level selection (lv1 to lv4) and exit button with hover effects.
 - Background: Main menu background image.
- **Level Selection:**
 - Dropdown Menus: For selecting files and algorithms.
 - Background: Level-specific background image.

d) Detailed Description of Components

Main Window:

The main window is the core of the application, where the grid visualization and player interactions occur. It is initialized with specific dimensions defined in config.py.

Grid Visualization (grid.py):

- **Roads:** Displayed as the background grid.
- **Obstacles:** Randomly assigned images from the OBSTACLE_FOLDER.
- **Flags:** Represent start and goal positions using images from the

FLAG_FOLDER.

- **Fuel Stations:** Displayed using images from the FUEL_FOLDER.
- **Player Positions:** Indicated using color-coded rectangles and player-specific images.

[Sidebar \(sidebar.py\):](#)

- **Control Buttons:** Includes previous, next, play, and pause buttons for navigating through the turns.
- **Player Information:** Displays each player's fuel, time, and current state.

[Menu \(menu.py\):](#)

- **Background:** Displays the main menu background image.
- **Buttons:** Includes level selection buttons (lv1 to lv4) and an exit button, with hover effects.

[Level Selection \(level_page.py\):](#)

- **Dropdown Menus:** Allows the selection of files and algorithms (for lv1) using animated dropdowns.
- **Background:** Displays a level-specific background image.

e) Interaction and Navigation

[User Flow:](#)

- **Main Menu:** Select a level.
- **Level Selection:** Choose an algorithm and file for lv1, or directly select a file for other levels.
- **Grid Visualization:** Navigate turns using control buttons.

[Interactive Elements:](#)

- **Buttons and Dropdowns:** Provide visual feedback through hover effects.

References

- 1) [Course study document](#) of Ms. Nguyen Ngoc Thao
- 2) “*Cơ sở AI*” lecture book from VNH-HCM University of Science
- 3) “*Artificial Intelligence: A modern Approach (Fourth Edition)*” book

__END.__