

**VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF SCIENCE
ADVANCED INFORMATION TECHNOLOGY FACULTY**



**PRACTICAL PROJECT: SEARCHING
INTRODUCTION TO ARTIFICIAL INTELLIGENCE – CSC14003**

PROJECT #1: DELIVERY SYSTEM

—o0o—

INSTRUCTOR(S)

MS. NGUYỄN NGỌC THẢO
MR. NGUYỄN THANH TÌNH
MS. HỒ THỊ THANH TUYẾN

—o0o—

GROUP'S INFORMATION

<u>NO.</u>	<u>STUDENT ID</u>	<u>FULL NAME</u>	<u>EMAIL</u>
1	22127174	NGÔ VĂN KHẢI	nvkhai22@clc.fitus.edu.vn
2	22127322	LÊ PHƯỚC PHÁT	lpphat22@clc.fitus.edu.vn
3	22127388	TÔ QUỐC THANH	tqthanh22@clc.fitus.edu.vn
4	22127441	THÁI HUYỀN TÙNG	thtung22@clc.fitus.edu.vn

HO CHI MINH CITY, JULY 2024

TABLE OF CONTENTS

I. Project evaluation	5
1. Work assignment table	5
2. Self-evaluation of the completion rate at each level of the project and other requirements	6
II. Detailed program system description.....	6
1. Requirements.....	6
2. Settings.....	6
3. Supporting classes and functions.....	6
4. Graphical User Interface (GUI) and gameplay.....	11
III. Detailed algorithm description and implementation	24
1. Level 1	24
a. Ideas	24
b. Detailed algorithm description and implementation	24
2. Level 2	28
a. Ideas	28
b. Detailed algorithm description and implementation	28
3. Level 3	29
a. Ideas	29
b. Detailed algorithm description and implementation	29
4. Level 4	31
a. Ideas	31
b. Detailed algorithm description and implementation	31
IV. Test case description and evaluation	32
1. Test case description	32
2. Test case evaluation.....	43

TEACHERS' COMMENTS

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Date: July, ..., 2024

Graded and commented Teacher(s)

PLEDGE

Our team declares that this research was conducted by the team, under the supervision and guidance of the teachers of the **Introduction to Artificial Intelligence – CSC14003** subject: **Ms. Nguyen Ngoc Thao, Mr. Nguyen Thanh Tinh, and Ms. Ho Thi Thanh Tuyen**. The results of this study are legal and have not been published in any form before. All documents used in this study were collected by the team and from various sources and are fully listed in the references section. In addition, we also use the results of several other authors and organizations. All are properly cited. In case of copyright infringement, we are responsible for such action. Therefore, **Ho Chi Minh City University of Science (HCMUS)** is not responsible for any copyright violations committed in our research.

RESEARCH PROJECT

I. Project evaluation

1. Work assignment table

StudentID	Full Name	General Tasks	Detailed Tasks	Completion
22127174	Ngô Văn Khải	GUI	Gameplay Graphic User Interface (Design)	100%
			Gameplay Graphic User Interface (Coding)	100%
			Describe GUI	100%
22127322	Lê Phước Phát	Level 1	Describe detailed Level 1 algorithms and implementation (Report)	100 %
			Implement Level 1 (Coding)	100 %
		Level 2	Describe detailed Level 2 algorithms and implementation (Report)	100 %
			Implement Level 2 (Coding)	100 %
		Test cases	Generate for 5 test cases (Coding)	100 %
			Describe the test cases and the results when run on each of those test cases. (Report)	100 %
22127388	Tô Quốc Thanh	Level 3	Describe detailed Level 3 algorithms and implementation (Report)	100 %
			Implement Level 3 (Coding)	100 %
		Level 4	Describe detailed Level 4 algorithms and implementation (Report)	100 %
			Implement Level 4 (Coding)	100 %
		Test cases	Describe the test cases and the results when run on each of those test cases. (Report)	100 %
22127441	Thái Huyền Tùng	GUI	Menu Graphic User Interface (Design)	100%
			Menu Graphic User Interface (Coding)	100%
		Video	Video demos	100%

2. Self-evaluation of the completion rate at each level of the project and other requirements

No.	Details	Completion Rate
1	Finish Level 1 successfully	100 %
2	Finish Level 2 successfully	100 %
3	Finish Level 3 successfully	100 %
4	Finish Level 4 successfully	100 %
5	Graphical User Interface (GUI)	100 %
6	Generate 5 test cases for each level with different attributes. Describe them in the report.	100 %
7	Demo videos for demonstrating each test case.	100 %
8	Report all algorithms, and experiment with some reflections or comments.	100 %

II. Detailed program system description

1. Requirements

- Python version: 3.10+ with Pygame graphical module.
- Operating System: The command may vary across platforms, so you need run this program on Windows.

2. Settings

- Firstly, you need to set up the virtual environment in Python by using this command line below:

```
pip install virtualenv
python -m venv venv.
```

- Secondly, you need to activate the virtual environment in Python by using this command line below:

```
.venv\Scripts\activate
```

- Thirdly, we will install all libraries in the file requirements.txt by using this command line below. Note that we need “cd” into the folder “Source”.

```
pip install -r requirements.txt
```

- Finally, you need to run the application by executing the command “python main.py” in the console or terminal.

3. Supporting classes and functions

In our program, we designed some supporting classes and functions to prepare for each level to be implemented easily in each level.

- *Class cell (...)*

Implementation file: *cell.py*

This “**cell**” class is designed to represent a single cell on a map grid used for a search algorithm in delivering the system. Each “**cell**” object is initialized by

the **constructor method** `__init__` with the **y-coordinate** and **x-coordinate** of the cell and a **“raw_value”** from the map data. The **“raw_value”** is converted to a floating-point number and stored as **“value”**, with a default value of 0 if the conversion fails. This conversion is useful for handling numeric map data such as distances or costs. This constructor also maintains several dictionaries to track attributes related to multiple vehicles navigating through the map. These attributes include:

- **“visited”**: tracks whether a cell has been visited by a specific vehicle.
- **“parent”**: stores the parent cell for each vehicle, useful for path reconstruction.
- **“cost”**: holds the cumulative cost for a vehicle to reach this cell.
- **“heuristic”**: contains heuristic values for the cell, aiding in informed search algorithms like A* or GBFS.
- **“fuel”**: keeps track of the fuel level of each vehicle when it reaches this cell.
- **“time”**: stores the time taken for each vehicle to reach this cell.
- **“current_vehicle”**: identifies the current vehicle occupying the cell.

The class also includes a less-than operator **“lt”**, which is implemented to enable the less-than comparison between the cells. This method first compares cells based on the time it takes for the current vehicle to reach them, using the **“time”** attribute. If the times are equal, it then compares the cumulative cost for the vehicle to reach each cell using the **“cost”** attribute, which means the number of cells visited by the current vehicle. If the costs are also equal, the method proceeds to compare the fuel levels at each cell using the **“fuel”** attribute. This comparison is wrapped in a try block to handle any potential exceptions. If the fuel levels are equal, the method finally compares the heuristic values of the cells, which estimate the cost to reach the goal from the current cell. The method returns True if the heuristic value of the current cell is less than that of the other cell.

- *Class `vehicle_base` (...)*

Implementation file: *vehicle_base.py*

The **“vehicle_base”** class is designed to serve as a foundational class for different types of vehicles within the system. This class encapsulates common attributes and methods that are shared among all vehicle types.

The **“__init__”** method initializes a vehicle with attributes such as its name, starting coordinates (`start_y` and `start_x`), delivery time limit, initial fuel amount, and the algorithm used for pathfinding, which defaults to "algo" for level 1. These attributes include the vehicle's initial and current fuel, starting and current positions, as well as goal coordinates which are set to -1 initially to indicate they will be determined later. Additionally, temporary start and goal coordinates (`tmp_start_x`, `tmp_start_y`, `tmp_goal_x`, `tmp_goal_y`) are used for

intermediate calculations, ensuring the vehicle can navigate effectively between various points on the map. This method also includes several lists: `path` to store the path taken by the vehicle, `blocked_opposite` to keep track of cells blocked by an opposite vehicle, and `blocked_temp` to record temporarily blocked cells where the vehicle might need to wait.

The method **“get_algorithm_name”** returns the name of the algorithm class used by the vehicle, providing insight into the pathfinding strategy being employed.

This structure allows for the creation of specialized vehicle types by inheriting from **“vehicle_base”**, ensuring they share these common functionalities and attributes.

- *Class Board (...)*
Implementation file: *board.py*

The **“Board”** class represents the game board for a vehicle navigation simulation, where various types of vehicles navigate a grid-based map to reach their goals while managing fuel and time constraints. This class includes attributes to store the dimensions of the board (`“n”` is the number of rows, `“m”` is the number of columns), initial fuel amount (`“f”`), time limit (`“t”`), and map data (`“map_data”`). It also maintains lists of cells, vehicles, goals, and fuel stations.

The **“constructor__init__ method”** initializes these attributes, creates cell objects for each map cell, and generates vehicle objects based on the level of the game. The goals and fuel stations are identified and stored during initialization. Each vehicle's goal coordinates are set, and several auxiliary methods (**“generate_visited”**, **“generate_parent”**, **“generate_cost”**, **“generate_heuristic”**, **“generate_time”**, **“generate_fuel”**) initialize the status, parent pointers, cost values, heuristic values, time values, and fuel values for each cell, tailored to each vehicle.

In this class, we will design some auxiliary methods to initialize the respective values for all cells for a specific vehicle. These methods ensure that all necessary statuses and metrics are properly set up for the vehicles to navigate the board.

- **“generate_visited”**: this method will initialize the visited status by a `“False”` value for all cells for a specific vehicle through its name.
- **“generate_parent”**: this method will initialize the parent pointers by the initial pointer `(-1, -1)` for all cells for a specific vehicle through its name.
- **“generate_heuristic”**: this method will initialize the heuristic values for all cells for a specific vehicle.

- **“generate_cost”**: this method will set the cost values to infinity for all cells in the game board for a specific vehicle through its name, which this cell is initially considered unreachable or that the travel cost has not been determined. The cost represents the cumulative effort or resources needed for the vehicle to travel through the cells.
- **“generate_fuel”**: this method will set fuel values to infinity for all cells in the game board for a specific vehicle through its name.
- **“generate_time”**: this method will set the time values to infinity for all cells in the game board for a specific vehicle through its name, which this cell is initially considered unreachable or that the time has not been determined.

The **“get_vehicle”** method returns a sorted list of vehicles by their names.

The **“get_distance”** method calculates the Manhattan distance between two points, used for heuristic calculations.

The **“can_visit”** method checks if a cell can be visited by a specific vehicle based on its coordinates, visitation status, and obstacle status.

The **“tracepath”** method traces the path from the goal to the start for a specific vehicle, returning a list of tuples representing the path.

The **“unique_path”** method removes duplicate coordinates from the path to ensure a unique set of coordinates.

The **“path_time_fuel”** method returns the path with time and fuel information for a specific vehicle, including the time the vehicle leaves each cell and the remaining fuel.

- *Function `fought_cells (...)`*

Implementation file: *board.py*

The `fought_cells` function is designed to determine if a given cell, specified by its coordinates (y, x), is part of any path from a list of paths. Each path is represented as a list of tuples containing coordinates. The function takes three arguments: y, the y-coordinate of the cell; x, the x-coordinate of the cell; and paths, a list of paths where each path is a list of tuples representing coordinates. The function returns an integer representing the index of the path that contains the cell (y, x), or -1 if no path contains the cell.

The function iterates over all paths using `enumerate(paths)`, which provides both the index i and the path itself. For each path, it extracts the coordinates (py, px) from the tuples in the path using a list comprehension. It then checks if the cell (y, x) is present in the extracted coordinates using the `in` operator. If the cell (y, x) is found in the current path's coordinates, the function returns the index i of that path. If the cell (y, x) is not found in any of the paths, the function returns -1.

- *Function `read_input_file (...)`*

Implementation file: *read_input.py*

The `read_input_file` function is designed to read a file containing the city map and relevant parameters for a delivery vehicle routing problem. The function accepts a single argument, `filename`, which is the name of the input file to be read. The file contains parameters such as the number of rows (`n`), columns (`m`), committed delivery time (`t`), fuel tank capacity (`f`), and the map data representing the city layout.

The function starts by opening the specified file in read mode. It then reads the first line of the file, which contains four integers separated by spaces: `n` (number of rows), `m` (number of columns), `t` (committed delivery time), and `f` (fuel tank capacity). These values are extracted by stripping any leading or trailing whitespace from the first line, splitting the line into individual components, and converting them to integers using the `map` function.

Next, the function initializes an empty list called `map_data` to store the city map data. It then reads the next `n` lines from the file, where each line represents a row of the city map. For each line, the function strips any leading or trailing whitespace, splits the line into individual elements, and appends the resulting list to `map_data`. Each element in `map_data` can be one of the following:

- 0: Space that the delivery vehicle can move into.
- -1: Impassable space (e.g., buildings, walls, and objects).
- 'S': The starting location of the delivery vehicle.
- 'G': The goal location (customer's location).
- Positive integers representing the toll booth times.

Finally, the function returns a tuple containing the extracted values: `n`, `m`, `t`, `f`, and `map_data`. This tuple provides all the necessary information for initializing the city map and configuring the delivery vehicle routing problem.

- *Function `write_paths_to_file (...)`*

Implementation file: *write_out.py*

The `write_paths_to_file` function is responsible for writing the paths taken by vehicles to a specified file. This function accepts three arguments: `file_path`, which is the path to the file where the paths will be written; `vehicles`, which is a list of vehicle objects; and `level`, which indicates the level of the algorithm being used. The `level` parameter is utilized to differentiate the output format based on the algorithm level.

The function begins by opening the specified file in append mode using the `with open(file_path, "a") as file:` statement. This ensures that existing content in the file is preserved and new content is added to the end.

The function then checks the value of the `level` parameter. If the level is 1, the function iterates over each vehicle in the `vehicles` list and performs the following actions:

1. Retrieves the algorithm name used by the vehicle by calling the `get_algorithm_name` method on the vehicle object.
2. Write the algorithm name to the file, followed by the vehicle's name.
3. Check if the vehicle has a valid path. If a valid path exists, it formats the path as a string of coordinates (y, x) and writes this formatted path to the file. If no valid path is found, it writes a message indicating this.

For levels other than 1, the function follows a similar process but omits writing the algorithm name. It iterates over each vehicle in the vehicles list and performs the following actions:

1. Write the vehicle's name on the file.
2. Check if the vehicle has a valid path. If a valid path exists, it formats the path as a string of coordinates (y, x) and writes this formatted path to the file. If no valid path is found, it writes a message indicating this.

4. Graphical User Interface (GUI) and gameplay

4.1 Graphic User Interface (GUI) description:

When the program is run, the menu is displayed first. Users can choose which level, input file (and algorithm for level 1) to run, see the credit or exit program.

Then the game board is shown. All GUI programs are stored in GUI folder. There are many constant variables stored in GUI/*constants.py* folder.

- *Class LeveList (...)*
Implementation file: level_list.py

The “**LeveList**” class is programed to get level.

- *Class Image_UI (...)*
Implementation file: image.py

The “**Image_UI**” class is designed to store all images used in the program.

The “**__init__**” method initializes the displayed screen from pygame and the cell side's length. The class stores all images in GUI/assets folder: background image, cell side's length, cell's size, empty cell, wall cell, gas station cell (fuel station cell), toll booth cell (time cell), start cell for of vehicles, goal cell of 10 vehicles, 10 vehicles, different line shape of 10 vehicles. Class's attribute:

- “**screen**”: stores window's screen.
- “**background**”: stores background image.
- “**cell_side**”: stores cell side's length.
- “**cell_size**”: stores cell's size.
- “**empty_cell_img**”: stores empty cell image.
- “**wall_cell_img**”: stores wall cell image.
- “**fuel_cell_img**”: stores gas station cell (fuel station cell) image.
- “**time_cell_img**”: stores toll booth cell (time cell) image.

- “**start_cell_img**”: stores 10 vehicles’ Starting location image.
- “**goal_cell_img**”: stores 10 vehicles’ Goal location image.
- “**vehicle_right_img**”: stores 10 vehicles’ image (go to the right).
- “**vehicle_up_img**”: stores 10 vehicles’ image (go up).
- “**vehicle_left_img**”: stores 10 vehicles’ image (go to the left).
- “**vehicle_down_img**”: stores 10 vehicles’ image (go down).
- “**line_h_img**”: stores horizontal line image in 10 different colors.
- “**line_v_img**”: stores vertical line image in 10 different colors.
- “**left_down_img**”: stores left_down line image in 10 different colors.
- “**left_up_img**”: stores left_up line image in 10 different colors.
- “**right_down_img**”: stores right_down line image in 10 different colors.
- “**right_up_img**”: stores right_up line image in 10 different colors.

The “**writeNumber**” method writes number of gas station(s) and toll booth(s) on the map.

The “**showX**” method shows the ‘X’ image in the given position. If this method shows any image relative to vehicle, the number of that vehicle is given to the method.

The “**drawX**” method shows the ‘X’ line in the given position with the given color.

- *Class Board_UI (...)*
*Implementation file: **image.py***

The “**Board_UI**” class shows the game board from the input file.

The “**__init__**” method initializes pygame screen, the board’s size, committed delivery time, fuel tank capacity and level. The class finds the cell side’s length by the map’s size. The cell side’s length is 120, 60, 45 or 30 depending on the board’s size. Class's attribute:

- “**n**”: stores window’s screen.
- “**m**”: stores window’s screen.
- “**t**”: stores window’s screen.
- “**f**”: stores window’s screen.
- “**map_data**”: stores window’s screen.
- “**cells**”: stores window’s screen.
- “**level**”: stores window’s screen.

The “**readMapData**” method reads and stores the input map to the class.

The “**returnCellSide**” method returns the cell side’s length.

The “**showBoard**” method displays all items in the map.

- *Function map_UI (...)*

Implementation file: *gui.py*

This “**map_UI**” function shows the game map by initializing Class Board_UI. Press ‘Enter’ to trigger the program to run the chosen search algorithm.

- *Function path_UI (...)*

Implementation file: *gui.py*

This “**path_UI**” function shows the vehicles’ movement after the search algorithm is complete, time index, fuel index of each vehicle and algorithm name (level 1). After pressing the Enter key when viewing the game map, the program starts to run the search algorithm and returns the path for the vehicle(s). This function receives the “paths” list and copy to another list called “line_list”. The “line_list” list does store the shape of the line of each vehicle and stores a new goal position. “Paths” list shows new goal positions when there are 2 paths[][] same tuples stand next to each other. Since “paths” only stores vehicles’ positions, time index and fuel index, it does not consist of the level 4 requirement: goal changing.

- *Function menu_UI (...)*

Implementation file: *gui.py*

This “**menu_UI**” function displays the menu and lets users choose a level, input file, and algorithm for level 1. Users can use arrow keys and enter key to choose what they want.

In addition, the user can choose level, input file from 1 to 5 (and algorithm if level 1). Then, the window shows the map from the input file. Press the Enter key then the vehicle(s) move to their goal. After finishing running, the user can press the Enter key again, the program ends showing the map and shows the menu.

- *Class BackButton (...)*

Implementation file: *common.py*

The initialization of this class requires the pygame screen. In addition, we can pass the x position, y position and the content. This class works as a back button so that we can return to the previous page.

Function get_back_button_rect() helps us get the size of the back button.

Function back_to() returns the current page or the previous page if we clicked it.

- *Class ChoiceList (...)*

Implementation file: *common.py*

The initialization of this class only requires the pygame screen so that it can render the views to players.

Function `choose_options()` requires three compulsive arguments: `choice_list`, `is_up`, `is_down` and one optional argument is the `letter_spacing`. The argument `choice_list` is the list that we want to display to the screen, `is_up` and `is_down` are variables that manage the top arrow keyboard pressing and bottom arrow keyboard pressing respectively so that we can move up and move down to choose the option.

- *Class Credit (...)*

Implementation file: *credit.py*

The initialization of this class requires one argument that is the pygame screen to render the view.

Function `write_text_content()` to write and display the text to the view. In this situation we use this function to write down our member's name of our student's id so that everyone can know the authors of this project.

Function `get_back_to()` performs when we click back button, if we did not do that it will return the current view (credit view).

Function `display_credit()` is the core part of this class. It will display the title and the content of the credit page. It requires one compulsive argument which is the `is_left` so that it can keep track of the back button pressing process.

- *Class Menu (...)*

Implementation file: *menu.py*

The initialization of this class only one compulsive argument that is the pygame screen. There are also some private variables such as `option_result` which are used to manage the option of players, background to display the surrounding image and `menu_screen_choice` which stores a 2d array and each element of that array is 1d array containing a boolean value and the text content.

Function `get_choice()` which is used to get the current option in the menu and highlight it.

Function `get_choose_option()` which is used for getting the selected option in the menu when the players press enter button.

Function `display_menu()` is used for displaying the title and the content (the list of choices) in the menu. Moreover, it is also used for checking the enter button pressing from the players.

- *Class Text_Display (...)*

Implementation file: *text.py*

The initialization of this class requires one compulsive argument that is the content of the text. The rest of arguments such as `font_type`, `font_size` and `text_color` have already had a default value. However, we can change them for specific purposes.

Function `get_text_position()` helps us to get the text position in the screen which returns the position `x` and position `y` in Oxy surface.

Function `show_text()` has one optional argument which shows the color of the text. The default value of that color is white color which has already been declared in `constant.py`.

Function `center_text()` which will display the text in the middle depends on the text position.

- *Class `UI_Level_1` (...)*

Implementation file: *ui_level_1.py*

Although its name shows that it might support the level 1 UI, we decided to use it to display all levels of the project. The “level_1” we can understand as the version of this class. This class only needs one argument which is pygame screen to render views. These variables such as `is_click_back` used for checking back action, `option_back_to` used for getting to the previous page and `option_result` used for checking the selected option when players press enter key. Two lists which show input files and the name of search algorithm respectively.

Function `get_choice()` is used for getting the current option.

Function `get_back_to()` performs when we click back button, if we did not do that it will return the current view.

Function `get_option_result()` if we click the enter key on the keyboard, it will return the selected option for the program.

Function `write_text_content()` is used for writing text on the screen.

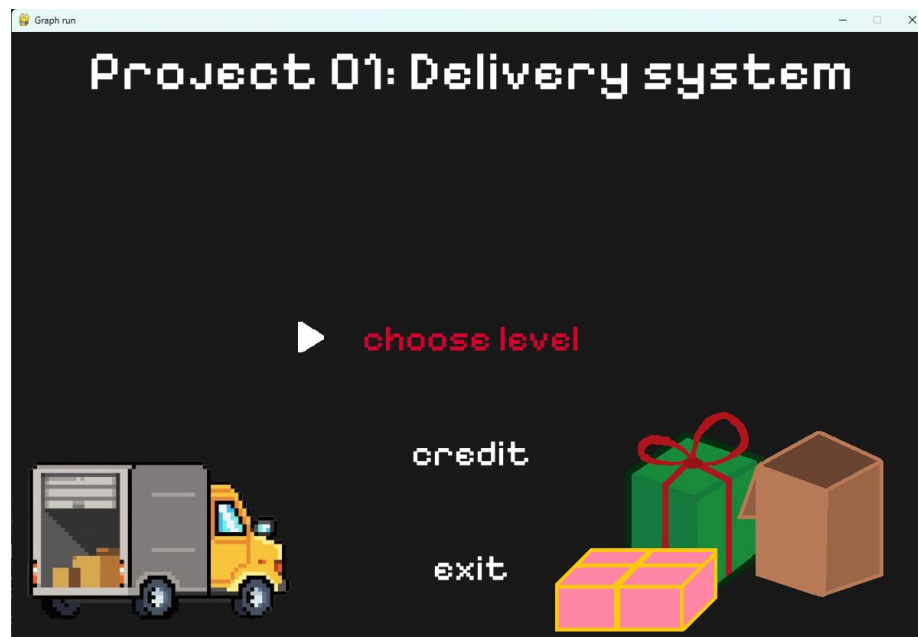
Function `draw_ui()` uses the function `write_text_content` to display text on the specific purpose.

Function `show_level_inputs()` is used for showing the input files of a specific level in the program. There are 4 levels in total and each level has 5 input files. This function also shows the title of this page and checks the key input from the keyboard, left arrow button and enter button which performs back the previous page and choose the selected input file respectively.

Function `show_level_list()` performs almost the same as the function `show_level_inputs()`. Instead of showing the input files it will show the search algorithm names. However, because of the different current state, we must split it out from the function `show_level_inputs()`.

4.2 Gameplay description:

First, we open the program by using the command: `python main.py`.



The main menu of the game.

Here we can see there are three categories in the Menu of the game. First is the “choose level”, then the “credit” and finally the “exit button”.

First, we will describe the exit button. This selection helps players to stop and close the program.

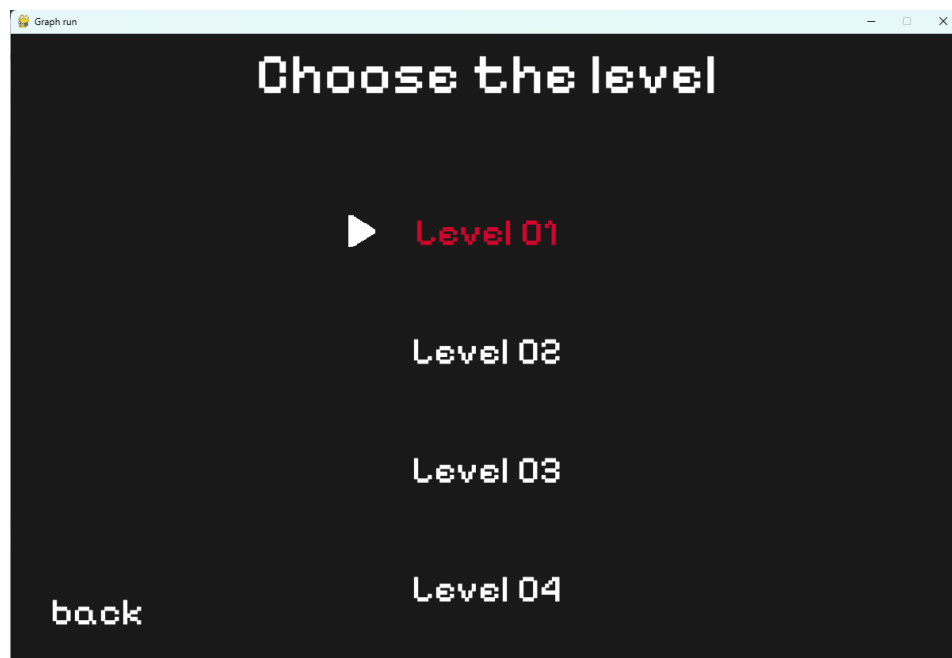
Second, the credit page will show our group members who contribute to this project.



The credit page.

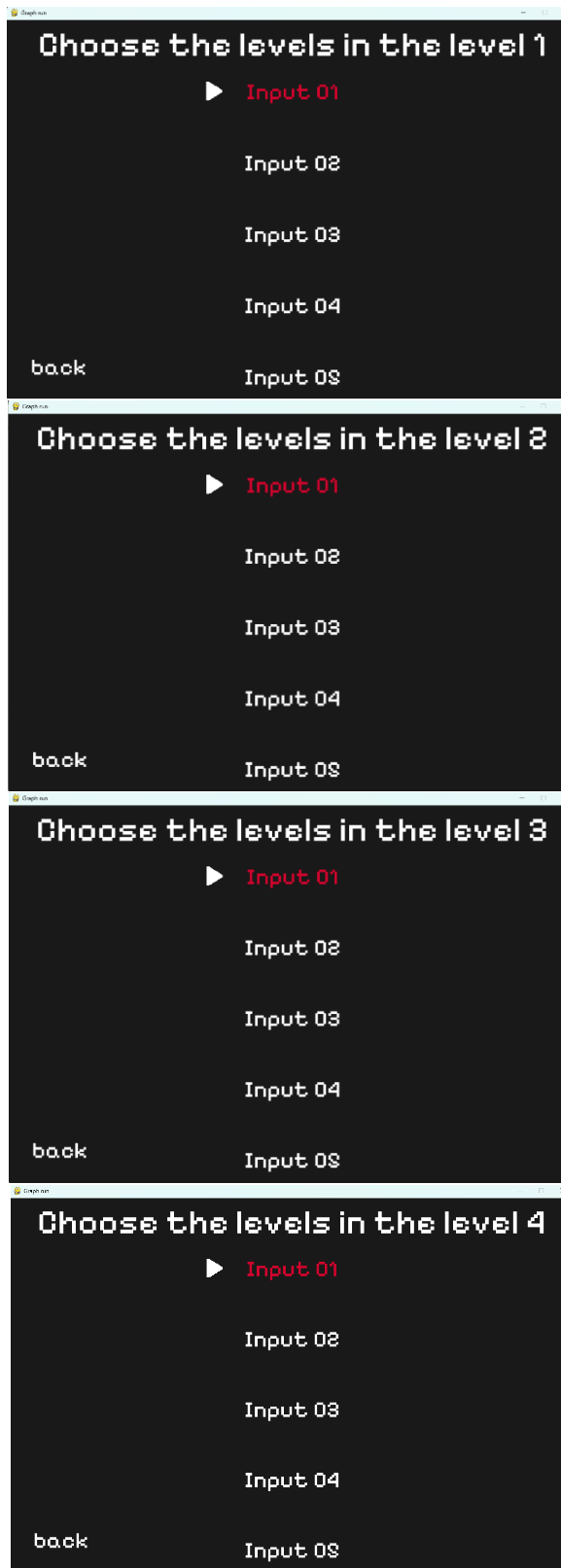
Next, we move to the main part of the game. The choose level category.

The figure below is the place where players can choose the level, including 4 different levels.



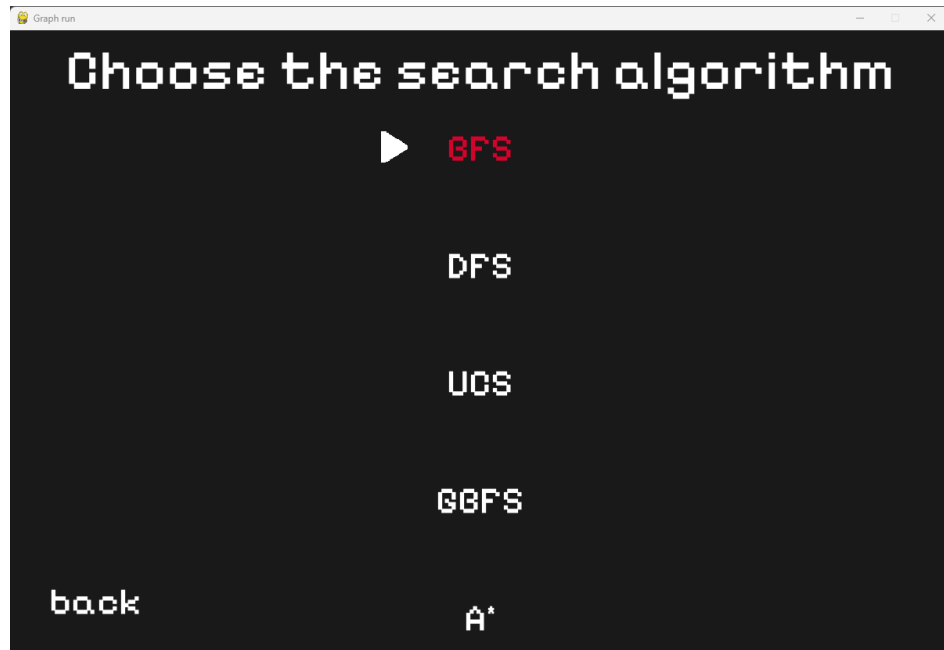
Choose level menu.

If we choose one of these options, the screen will show that there are 5 different inputs at each level.



Each level has 5 different input files.

First, we move to Level 01 in the choose level menu. In the Level 01, each input file will execute 5 search algorithms, including BFS, DFS, UCS, GBFS, A*



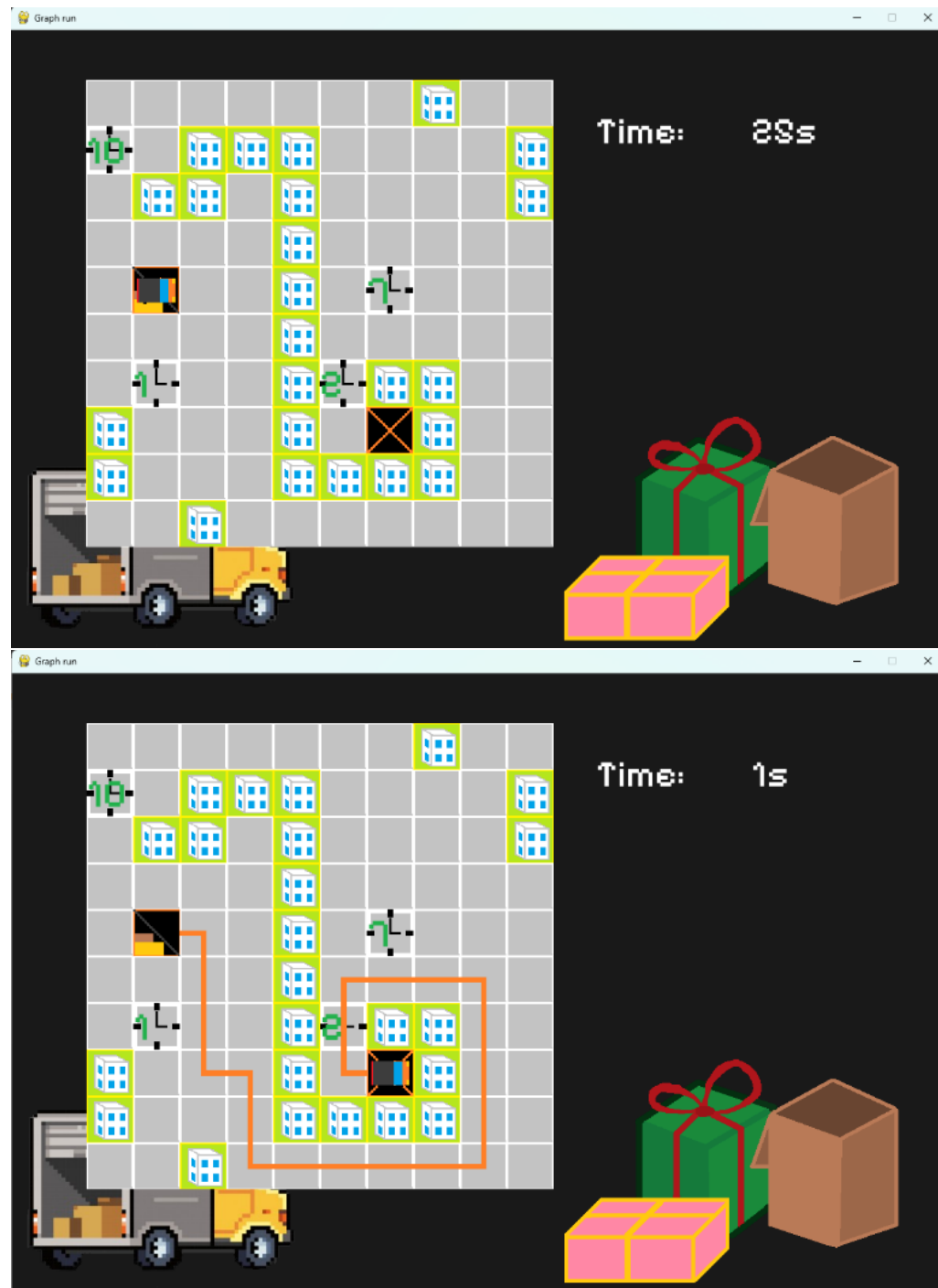
5 search algorithms in each input file in Level 01.

When players click one of these search algorithms, the screen will show the players the board that executes the selected algorithm.



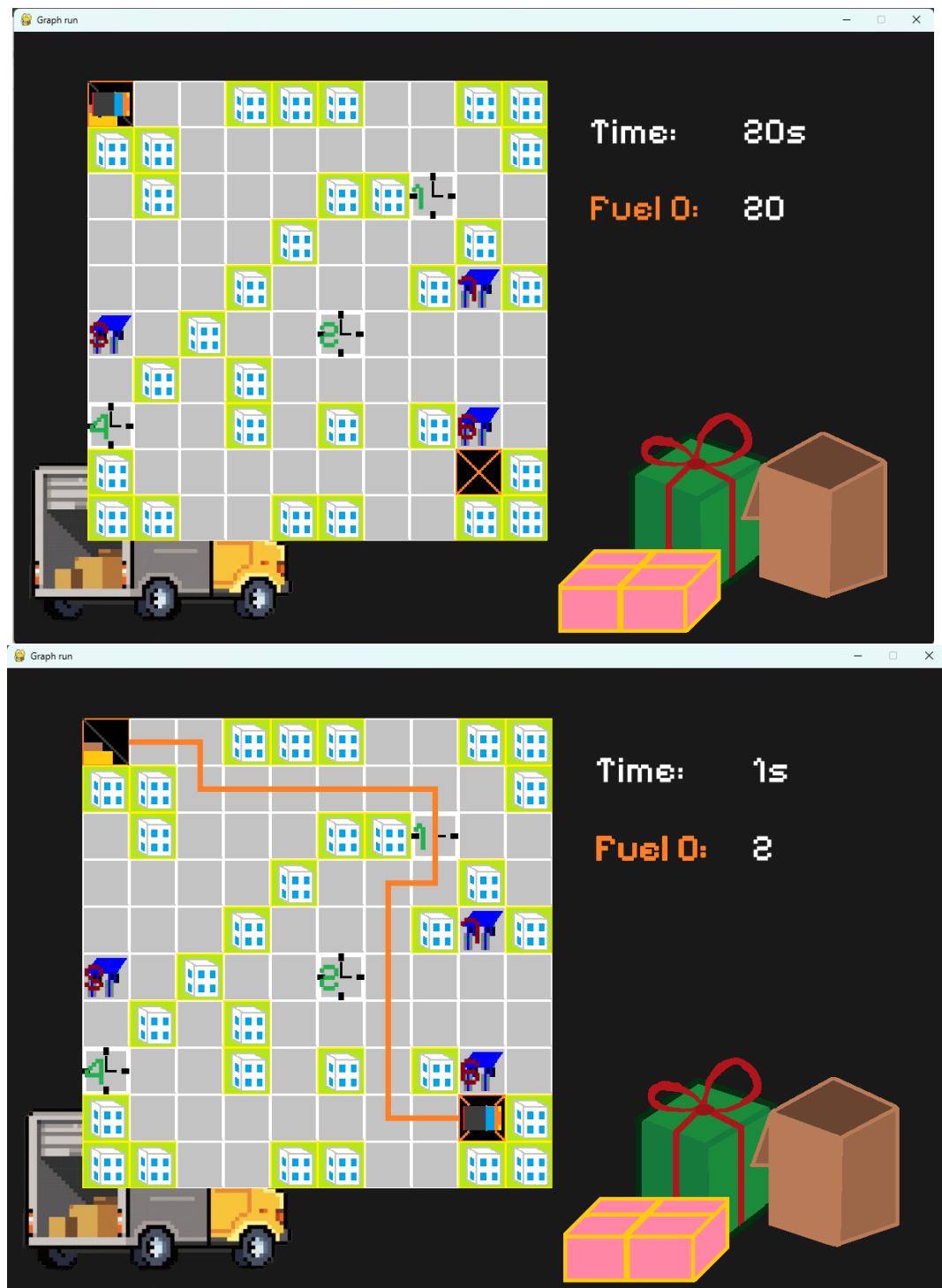
E.g. The board of the BFS algorithm.

Next, we move to Level 02, in this level, each input file will execute the A* search algorithm to find the shortest path from start point to the destination within a given time.



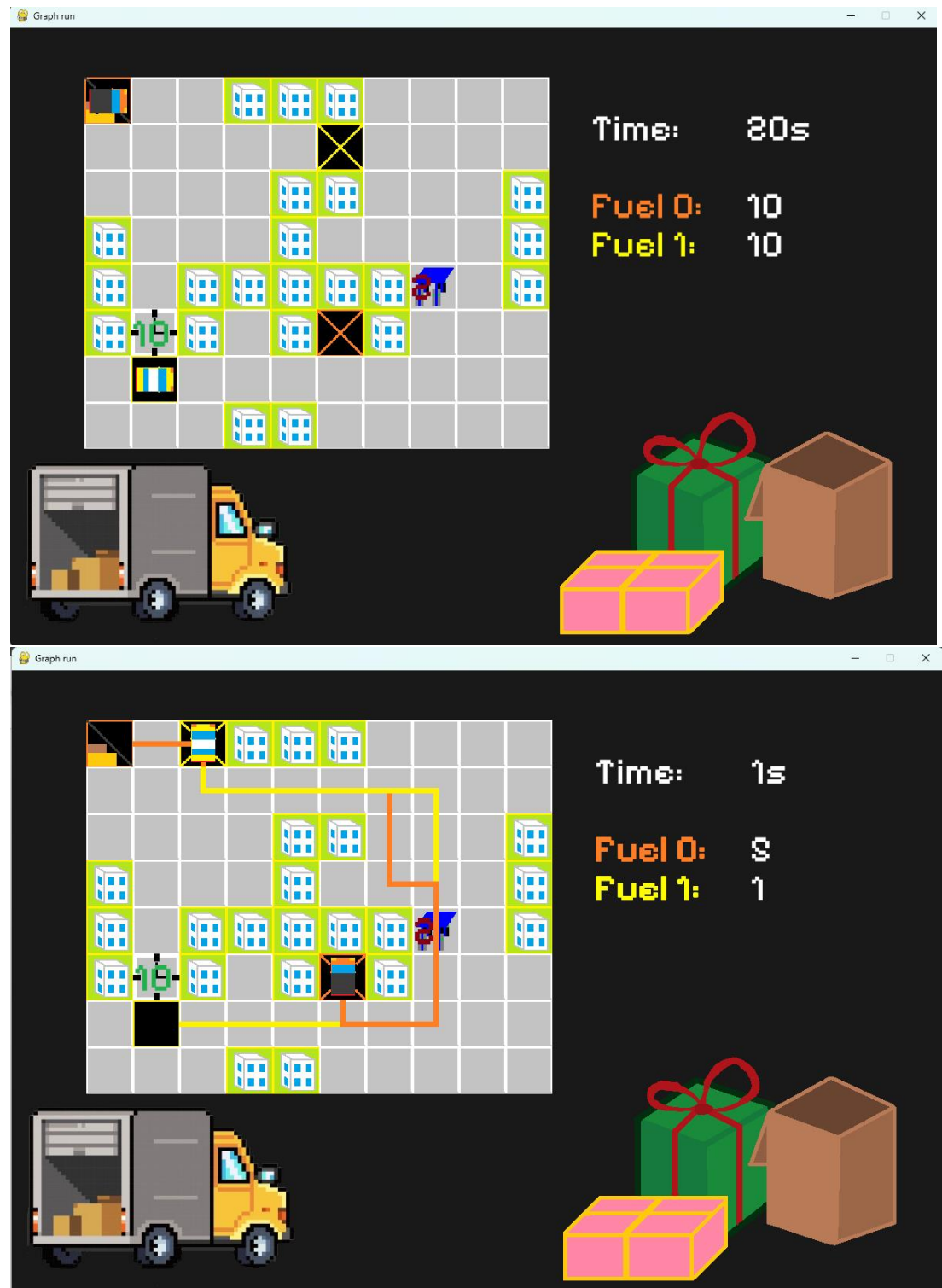
E.g. The first input file of Level 02 before and after execution.

Next, we move to Level 03, in this level, each input file will execute the A* search algorithm to find the shortest path from start point to the destination within a given time and a given fuel.



E.g. The fourth input file of Level 03 before and after execution.

Finally, we move to Level 04 which performs with multiple agents within a given time and fuel to find the path.



E.g. The fifth input file in Level 04 which shows a rectangle map.

III. Detailed algorithm description and implementation

1. Level 1

a. Ideas

Initially, we will design the `vehicle_level1` class that inherits from a **“vehicle_base”** class, which provides foundational functionalities for vehicles, and represents a level 1 vehicle. This vehicle can use various search algorithms:

- **Blind search algorithms:** Breadth-First Search (BFS), Depth-First Search (DFS), and Uniform-Cost Search (UCS).
- **Heuristic search algorithms:** Greedy Best First Search, A*

The class constructor **“__init__”** will initialize a **“vehicle_level1”** instance with specific attributes: vehicle’s name, starting coordinates (**“start_y”** for y-coordinate, and **“start_x”** for x-coordinate), delivery time limit (**“delivery_time”**), fuel capacity (**“fuel”**), and the algorithm to be used (**“algorithm”**). It invokes the constructor of the base class using `super()`, passing these attributes and then calls the `set_algorithm` method to assign the appropriate algorithm to the vehicle.

The **“set_algorithm”** method determines which algorithm to associate with the vehicle based on the provided algorithm name. It checks the value of the algorithm and assigns the corresponding algorithm class instance to the vehicle's algorithm attribute. If an unrecognized algorithm name is provided, it raises a **ValueError**.

Finally, the `process` method allows the vehicle to execute its assigned algorithm on a given board. This method calls the `execute` method of the algorithm instance, passing the board as an argument. The result, which includes the path along with time and fuel information, is returned.

For each specific algorithm, we will describe it in detail in the next parts of the document.

Note that at this level, there is only one vehicle (agent) and it will travel directly from the initial state, which means the start cell of this vehicle in the board game to the final state, which means the end cell of it.

b. Detailed algorithm description and implementation

- *Breadth First Search (BFS)*

Initially, we will initialize the **BFSAlgorithm** instance with a given vehicle object. This vehicle is stored as an attribute of the instance, which inherits the **“vehicle_level1”** class, allowing the algorithm to access and modify the vehicle’s state during execution.

Then, we will initialize all attributes for the board cells with the initial values and states, which are **“visited”**, **“parent”**, **“cost”**, **“heuristic”**, **“fuel”** and **“time”**, specifically to the vehicle’s name. This ensures that each vehicle can maintain its state independently. Next, this

algorithm will start by identifying the starting cell of the vehicle (the first state-delivery vehicle (S)) and marking it as visited. It then initializes a frontier, implemented as a deque, containing the starting cell.

The algorithm enters a loop where it processes each cell in the frontier. Within this loop, the BFS algorithm will remove a cell from the frontier (following the **First-In-First-Out**, or **FIFO principle**) or it can be said that BFS will choose the shallowest cell in the frontier because the frontier is a queue. Then, it checks if the goal cell has been reached. If the goal is reached, the algorithm marks it as visited and exits the loop. If not, the algorithm explores the neighboring cells (successors) in four possible directions: right, left, up, and down. For each valid neighbor (successor), it marks the cell as visited, sets its parent to the current cell, and adds it to the frontier for further exploration.

Once the BFS loop completes, the algorithm checks if the goal cell was reached and, if so, traces the path from the goal cell back to the start cell using the parent references. If no valid path is found, it ensures the goal cell's visited status is reset.

- *Depth First Search (DFS)*

This algorithm is like the BFS algorithm, but it will use the stack for the frontier, which follows the Last-In-First-Out (LIFO) principle.

Initially, we will initialize the **DFSAlgorithm** instance with a given vehicle object. This vehicle is stored as an attribute of the instance, which inherits the “**vehicle_level1**” class, allowing the algorithm to access and modify the vehicle’s state during execution.

Then we will initialize all attributes for the board cells with the initial values and states by the vehicle's name.

The algorithm starts by identifying the starting point of the vehicle, marking it as visited, and initializing a stack containing the starting cell, which is used to manage the frontier.

Next, the algorithm enters a loop where it will choose the deepest cell in the frontier by removing the cell in the stack. It will check if the current cell is the goal cell, which means that the goal cell has been reached. If the goal cell is reached, the algorithm marks it as visited and exits the loop. If not, it expands the current cell to generate its child cells by moving from the current cell in four directions: right, left, up, and down. For each valid child cell, it marks it as visited, sets its parent to the current cell, and adds it to the stack for further exploration.

Once the DFS loop completes, the algorithm checks if the goal cell was reached and, if so, traces the path from the goal cell back to the

start cell using the parent references. If no valid path is found, it ensures the goal cell's visited status is reset.

- *Uniform Cost Search (UCS)*

Initially, it is likely above algorithms that we will initialize the **UCSAlgorithm** instance with a given vehicle object. This vehicle is stored as an attribute of the instance, allowing the algorithm to access and modify the vehicle's state during execution.

Then, it initializes all attributes for the board cells with the initial values and states by the vehicle's name.

The algorithm begins by identifying the starting point of the vehicle, marking it as visited, and initializing the initial path cost default by 0 and the frontier (priority queue) containing the starting cell and its cost.

This algorithm enters a loop where it chooses the lowest-cost cell in the priority queue, which means that it will remove the node with the lowest path cost. Then, it will check the goal cell has been reached. If the goal is reached, the algorithm marks it as visited and exits the loop. If not, the algorithm will expand the current cell to generate its successors in four directions: left, right, up, and down. Because the vehicle will travel through each cell, we can consider that the cost of each cell is one unit. It uses the “can_visit” method with the new coordinate of the new cell and the specific vehicle’s name to check if this coordinate of this cell is out of bounds, this cell is a block (such as buildings, walls,...) or this cell has been visited by other vehicles. If it returns true, this cell is a valid successor. Otherwise, this cell will be ignored. For each valid successor, it calculates the new cost of reaching that cell by adding 1 unit into the current path cost of this vehicle. If the new cost is lower than the previously recorded cost for that cell, it updates the cell’s properties and adds it to the priority queue with its new cost.

Once the UCS loop completes, the algorithm checks if the goal cell was reached and, if so, traces the path from the goal cell back to the start cell using the parent references. If no valid path is found, it ensures the goal cell's visited status is reset.

- *Greedy Best First Search (GBFS)*

Initially, it is likely above algorithms that we will initialize the **GBFSAlgorithm** instance with a given vehicle object. This vehicle is stored as an attribute of the instance, allowing the algorithm to access and modify the vehicle's state during execution.

Then, it initializes all attributes for the board cells with the initial values and states by the vehicle's name.

The algorithm begins by identifying the starting point of the vehicle, marking it as visited, and initializing the frontier (priority queue) containing the starting cell and its heuristic value.

This algorithm enters a loop where it chooses the lowest heuristic values of the current cell in the priority queue, which means that it will pop the cell from the frontier with the lowest heuristic value. Then, it checks if the goal cell has been reached. If the goal cell is reached, the algorithms mark it as visited and exit the loop. If not, the algorithm will expand the current cell to generate its successors in four directions: left, right, up, and down. Because the vehicle will travel through each cell, we can consider that the cost of each cell is one unit. It uses the “can_visit” method with the new coordinate of the new cell and the specific vehicle’s name to check if this coordinate of this cell is out of bounds, this cell is a block (such as buildings, walls,...) or this cell has been visited by other vehicles. If it returns true, this cell is a valid successor. Otherwise, this cell will be ignored. For each valid neighbor, it updates the cell's properties and adds it to the priority queue with its heuristic value.

After the GBFS loop completes, the algorithm checks if the goal cell was reached and, if so, traces the path from the goal cell back to the start cell using the parent references. If no valid path is found, it ensures the goal cell's visited status is reset.

- *A* Search*

Initially, we will initialize the AStarAlgorithm instance with a given vehicle object. This vehicle is stored as an attribute of the instance, which inherits the “vehicle_level1” class, allowing the algorithm to access and modify the vehicle’s state during execution.

Then, we will initialize all attributes for the board cells with the initial values and states, which are “visited”, “parent”, “cost”, “heuristic”, “fuel” and “time”, specifically to the vehicle’s name. This ensures that each vehicle can maintain its state independently.

The method begins by initializing these properties for the vehicle. It then sets the starting cell's properties, marking it as visited, setting its cost to zero, and associating it with the current vehicle. The method uses a priority queue (implemented with heapq) to manage the frontier, starting with the initial cell and its heuristic value. The main loop processes the cell with the lowest total cost (sum of the path cost and heuristic), and if the goal cell is reached, the loop exits. For each cell, the algorithm explores its neighbors (right, left, down, up) and updates their properties if a lower-cost path is found. Neighbor cells are then added to the priority queue with their total cost. After exiting the loop, if the goal cell is visited, the method traces the path from the

goal back to the start using the parent references. The execute method sets temporary start and goal coordinates for the vehicle, calls the `a_star` method to perform the search, and returns the path's time and fuel consumption.

2. Level 2

a. Ideas

At this level, we will use the A* algorithm to find the best path when the vehicle has a constraint about its time capacity. Because the time to move between two adjacent cells is 1 minute, for each cell traversed, we will calculate the time values of this vehicle by adding it to the total values. In addition, there are cells where a vehicle is required to stop for $a[i, j]$ minutes to pay a road toll, so we will calculate the more time fee into the previous time path values. If the time capacity of this vehicle is less than the delivery time limit (provided in the input file), meaning that the vehicle can also move, we continue to find the best path based on the path cost (sum of the cost value and heuristic value).

b. Detailed algorithm description and implementation

The “**vehicle_level2**” class extends the “**vehicle_base**” class, representing a vehicle in level 2 that utilizes the A* algorithm for pathfinding. This class initializes with a name, starting coordinates, delivery time, and fuel. It assigns an instance of the **AStarAlgorithm** to the vehicle, enabling it to execute the A* algorithm for navigating the board.

The “**AStarAlgorithm**” class is designed to find the shortest path from a vehicle’s start position to its goal position on a given board within the delivery time limit.

The constructor initializes the class with the specified vehicle. The main method, “**a_star_execution**”, performs the A* search algorithm on the board. It starts by generating necessary properties for the board's cells, including visited status, parent references, cost, and heuristic values. If the vehicle is starting at its initial coordinates, the time for the vehicle is also initialized to 1, otherwise, it is initialized to the current time of this vehicle. Then, it identifies the start cell based on the vehicle’s start coordinates. The start cell is marked as visited, and its cost is set to zero. The algorithm uses a priority queue (frontier) to manage cells to explore, starting with the initial cell.

The main loop processes the cell with the lowest f-score, calculated as the sum of the path cost and heuristic value. For each cell, the algorithm explores its neighbors (right, left, down, up) and updates their properties if a lower-cost path is found. Neighbor cells are added to the priority queue with their updated costs. The loop exits if the goal cell is reached. The algorithm then traces the path from the goal back to the start using parent references, returning the path if the goal cell is visited. If no path is found, the goal cell is marked as unvisited.

3. Level 3

a. Ideas

Use the A* algorithm to find the best path. Additionally, for each cell traversed, we save the remaining fuel of the vehicle. If the fuel is less than 0, meaning the vehicle cannot move to that cell, we do not continue to expand that branch.

If the vehicle cannot directly move from the start position to the goal, we will find the best fuel station the vehicle needs to move to and use the A* algorithm to find the path to that station.

Assuming the vehicle successfully reaches the fuel station, we will reset the vehicle's current fuel by the input fuel value. With the fuel station position as the new starting point, we use A* to find the best path to the goal. If the fuel is not enough, we again find the best fuel station and move to it.

In the case where the vehicle does not have enough fuel to move to the goal and cannot find a suitable fuel station (too far from the goal, not enough fuel to reach, etc.), the algorithm will immediately stop and show that no valid path was found from the start position to the goal.

b. Detailed algorithm description and implementation

Create `Vehicle_level3` class inherits from `vehicle_base` class. This class should include the vehicle's name, start position, goal position, current position, and fuel amount.

Implement the A* algorithm based on its original concept, with the frontier, is a priority queue, it stores cells in the map, which waiting to be expanded. The priority should be determined by the following principles: (cost, time, fuel, heuristic), where:

- **cost** represents the number of cells to traverse to reach the current cell
- **time** indicates the time of leaving the current cell
- **fuel** represents the amount of fuel remaining when reaching the current cell
- **heuristic** is the Euclidean distance (L2 distance) from the current cell to the goal. If the cell is a fuel station, it will have a little bit more priority.

Cells in the frontier will be given priority to expand in the above order, meaning the cell with the lowest cost will be expanded first. If it has the same cost, we will consider which cell has less time. If it has the same cost and time, we will prioritize the fuel needed to reach each cell, and finally based on the heuristic value.

By this way we will find the shortest path, taking the least amount of time and consuming the least fuel.

When initializing the frontier, the first element stored will be the cell at the starting position of the vehicle, then follow these steps:

1. Check if the frontier still has any elements. If so, take the first element to expand and pop that element out of the frontier. Otherwise, we end the algorithm.
2. Expand the element that just popped out: check 4 cells around the current cell (not including the diagonal), if it is possible to move to those cells (not exceeding the map limit, not a wall, not yet expanded, not yet added to the frontier,), we proceed to check that cell.
3. A new cell meets the conditions to be added to the frontier if: the cost from the currently expanded cell plus 1 is less than the original cost of that cell (default is $+\infty$), and the amount of fuel when arriving at the new cell is greater than or equal to 0 and time is allowed. We change the new values for that cell (cost, time, fuel), mark the path for tracing, mark it as added to the frontier, and then add it to the frontier. Because the frontier is a priority queue, it will arrange cell positions according to the rules presented above.
4. Check if the newly expanded cell is the cell at the vehicle's goal position. If it is correct, we stop the algorithm, otherwise, we back to step 1.

Create a function to find the best gas station: the closer the gas station is to the goal (with a smaller L2 distance value), the more priority it will have for the vehicle to move to. After choosing the best gas station, we assign the new destination location to the vehicle temporarily is that gas station instead of the original goal.

Create a function to combine the above functions:

- 1 Check whether the goal position has been visited. If so, stop the algorithm. If not, continue to step 2.
2. We call the A* algorithm, in the first-time call, the starting position and goal position will be the initial positions when we initialize the vehicle, but the next calls will be the new positions (usually starting at a gas station, or trying to go to a gas station, which will be explained in step 3).
3. In step 2, if we cannot find a suitable path and it is the first time we reach this step 3, it means from the start position cannot go directly to the goal position, so we need to find the best gas station and let the vehicle move to there. If the starting location in step 2 is a gas station (second-time call onwards), but cannot find a suitable path, meaning the vehicle will forever be unable to find its way

to the goal, we stop the algorithm immediately and display that the vehicle is not able to find the way to the goal.

4. Return to step 1

4. Level 4

a. Ideas

Because at level 4 there are many agents, we need to use an array called vehicles to store the vehicles

Like level 3, we use the A* algorithm to find the best path (including going to fuel stations) for each vehicle.

However, because many vehicles are moving, vehicles can appear on the same cell at the same time, and this is not valid. Therefore, when a collision occurs, vehicles need to wait for that vehicle to pass or find another path

The vehicle with the symbol 'S' is the vehicle with the highest priority, other vehicles need to give way to 'S' so that it reaches its goal as soon as possible. For other vehicles to give way to another vehicle, we just need to mark that cell not to go through → find another path, or increase the time needed to get to that cell → increase the time then that path may no longer be optimal, a new path will be searched, which may be different from the original path (cell indicators are changed for the blocked vehicle's perspective, other vehicles remain the same, Therefore, some values in the cell will be saved as a direct, to store cell values for each vehicle's perspective)

To complete this level, we need to store the time the vehicle leaves each cell in its path. This makes it easy to check at each corresponding time, which cell the vehicles are in on the map. If two vehicles collide, we mark the collision cells in each vehicle and call A* again to find the best path (maybe the current path is still the best then).

When a vehicle has reached its goal, we take the goal position as the starting point and create a new goal for it. Here, because there is no special requirement for a new destination, we will create a goal close to the starting point, to save computer memory when calling new A* and ensure that in most cases there is a path that exists. This is quite similar in reality; each shipper's delivery destination is usually not too far from each other. After delivering an order, he often moves to neighboring orders.

The algorithm will repeat until time runs out or the 'S' vehicle reaches its goal.

b. Detailed algorithm description and implementation

Create Vehicle_level4 class inherits from vehicle_base class. This class should include the vehicle's name, start position, goal position, current position, fuel amount, cells that are blocked or cells that need to wait.

Implement A* as level 3, but at each self, we need to check if it is in a blocked cell (vehicle cannot visit that cell) or waiting cell (vehicle needs to wait to move to that cell) of the current vehicle, the remaining is the same to A* of level 3.

Combine A* with find best Fuel station as level 3, call it the find_best_path function

To complete this level, we follow these steps:

1. With each vehicle, call find_best_path to find the path to the goal (including fuel station) (if the 'S' vehicle doesn't have the path, the function stops immediately).
2. For any single minute, we find the current position of each vehicle. If it's the time the vehicle leaves the current self, we check that the next cell in the path is possible: If there is a vehicle currently parked on it, add that cell to the blocked cells or waiting cells of the current vehicle. If two vehicles are opposite, it is a blocked cell, otherwise, it is a waiting cell (or temporary block for some minutes). If there isn't any vehicle on the next cell, move your vehicle to that cell and delete it on the old cell
3. If a vehicle reaches its goal, regenerate a new goal for it. If it's an 'S' vehicle, stop the function immediately.
4. If any vehicle is blocked, increase the current time by 1 minute and go back to step 1, else stop the function (no vehicle is blocked, all can go to its goal, now we just need to display it through GUI).

IV. Test case description and evaluation

1. Test case description

With each specific case, our team ran our experiment on about 5 test cases for each level.

Note: In test case 1, I will describe in detail the way how to find out the path solution for each search algorithm. For each remaining test case, I just show you the result of each level for analyzing and comparing their performance and efficiency.

- Level 1:
 - Input

input1_level1.txt	Level1 Image
<pre> 10 10 25 10 0 0 0 0 0 0 -1 0 0 0 0 -1 -1 -1 0 0 0 0 -1 0 -1 -1 0 -1 0 0 0 0 -1 0 0 0 0 -1 0 0 0 0 0 0 S 0 0 -1 0 0 0 0 0 0 0 0 0 -1 0 0 0 0 0 0 0 0 0 -1 0 -1 -1 0 0 -1 0 0 0 -1 0 G -1 0 0 -1 0 0 0 -1 -1 -1 -1 0 0 0 0 -1 0 0 0 0 0 0 0 </pre>	

In this map, we designed it with the size 10x10, the total delivery time limit which is 25 minutes, and the fuel amount which is 10.

At this level, the map just contains the vehicle at the starting point, the goal cell, and all the buildings, walls, ... (blocks).

○ Output

output1_level1.txt	GUI for output1_level1
<pre> BFSAlgorithm: S (4, 1) (4, 0) (3, 0) (2, 0) (1, 0) (1, 1) (0, 1) (0, 2) (0, 3) (0, 4) (0, 5) (1, 5) (2, 5) (3, 5) (4, 5) (5, 5) (6, 5) (7, 5) (7, 6) </pre>	

When you choose the BFS algorithm, this algorithm will find the best path that this vehicle can reach to the goal.

Firstly, this algorithm will initialize the cell (4, 1) as the starting point of the vehicle and mark it as visited. Then it will create a frontier, implemented as a deque, containing the starting cell.

The algorithm enters a loop where it processes each cell in the frontier. Within this loop, the BFS algorithm will remove a cell from the frontier (following the **First-In-First-Out**, or **FIFO principle**) or it can be said

that BFS will choose the shallowest cell in the frontier because the frontier is a queue.

In this case, the cell (4, 1) is the current cell will be chosen. Then, it checks if the goal cell has been reached. It can be seen that the cell (4, 1) is not the goal cell so the goal cell has not been reached. If not, it expands the current cell to generate its child cells by moving from the current cell in four directions: right, left, up, and down.

In this case, the cell (4, 1) will generate 4 child cells in four directions with order are right, left, down, and up, which are (4, 2) – right, (4, 0) – left, (5, 1) – down, and (3,1) – up. Then, it will check if each child cell is visited by the current node, or is the building block, or is out of bounds, it will return false. If not, the vehicle will move in this cell and the algorithm marks the new cell as visited and sets up its parent as the current cell. Finally, it will add the new cell to the frontier to adds it to the frontier for further exploration.

Once the BFS loop completes, the algorithm checks if the goal cell was reached and, if so, traces the path from the goal cell back to the start cell using the parent references. If no valid path is found, it ensures the goal cell's visited status is reset.

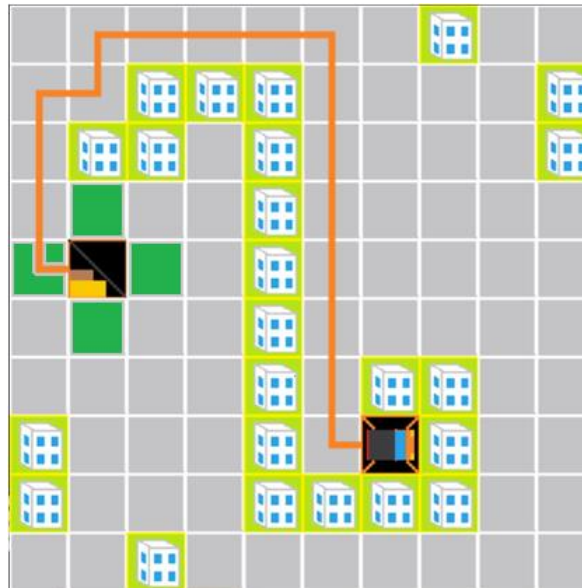
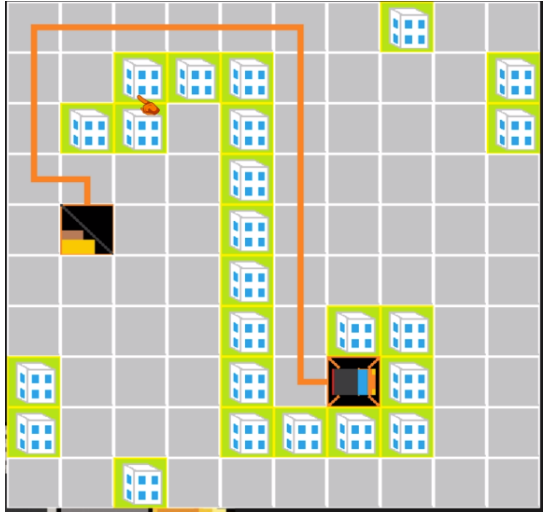
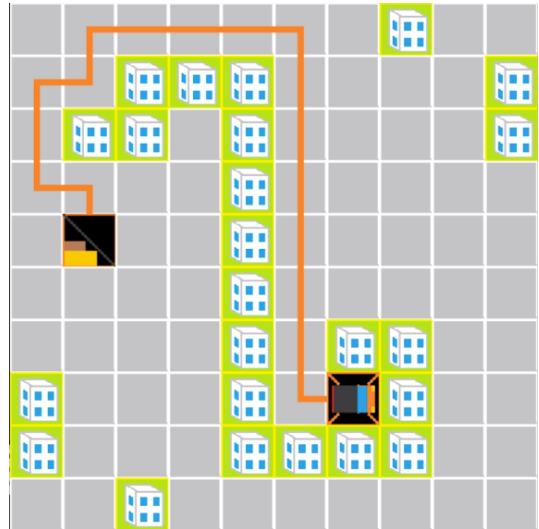


Figure: the BFS algorithm expands the current node to generate its child node.

output1_level1.txt	GUI for output1_level1
<p>DFSAlgorithm:</p> <p>S</p> <p>(4, 1) (3, 1) (3, 0) (2, 0)</p> <p>(1, 0) (0, 0) (0, 1) (0, 2)</p> <p>(0, 3) (0, 4) (0, 5) (1, 5)</p> <p>(2, 5) (3, 5) (4, 5) (5, 5)</p> <p>(6, 5) (7, 5) (7, 6)</p>	

In this algorithm, it will trace the path like the BFS algorithm, but the frontier is managed as the stack, which follows the Last-In-First-Out (LIFO) principle.

output1_level1.txt	GUI for output1_level1
<p>UCSAlgorithm:</p> <p>S</p> <p>(4, 1) (3, 1) (3, 0) (2, 0)</p> <p>(1, 0) (1, 1) (0, 1) (0, 2)</p> <p>(0, 3) (0, 4) (0, 5) (1, 5)</p> <p>(2, 5) (3, 5) (4, 5) (5, 5)</p> <p>(6, 5) (7, 5) (7, 6)</p>	

The algorithm begins by identifying the starting point of the vehicle which is cell (4,1), marking it as visited, and initializing the initial path cost default by 0 and the frontier (priority queue) containing the starting cell and its cost.

This algorithm enters a loop where it chooses the lowest-cost cell in the priority queue, which means that it will remove the node with the lowest path cost. Then, it will check the goal cell has been reached.

In this case, the cell (4, 1) is not the goal cell, so the goal cell has been reached. The algorithm expands the current cell to generate its child cells by moving from the current cell in four directions: right, left, up, and down.

In this case, the cell (4, 1) will generate 4 child cells in four directions with order are right, left, down, and up, which are (4, 2) – right, (4, 0) – left, (5, 1) – down, and (3,1) – up. Then, it will check if each child cell is visited by the current node, or is the building block, or is out of bounds, it will return false. For each valid successor, it calculates the new cost of reaching that cell by adding 1 unit into the current path cost of this vehicle. If the new cost is lower than the previously recorded cost for that cell, it updates the cell's properties and adds it to the priority queue with its new cost.

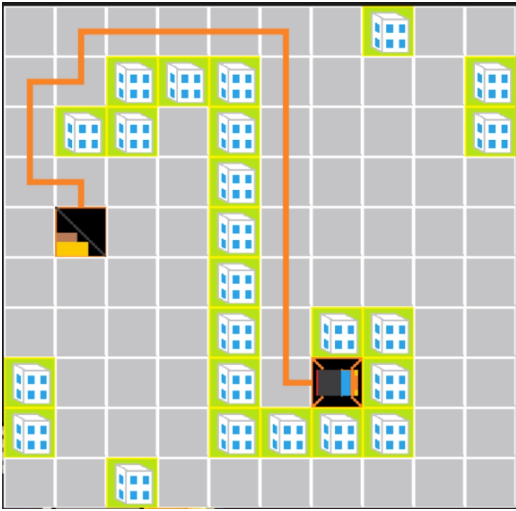
output1_level1.txt	GUI for output1_level1
<p>GBFSAlgorithm:</p> <p>S</p> <p>(4, 1) (4, 2) (4, 3) (5, 3)</p> <p>(6, 3) (7, 3) (8, 3) (9, 3)</p> <p>(9, 4) (9, 5) (9, 6) (9, 7)</p> <p>(9, 8) (8, 8) (7, 8) (6, 8)</p> <p>(5, 8) (5, 7) (5, 6) (5, 5)</p> <p>(6, 5) (7, 5) (7, 6)</p>	

The algorithm begins by identifying the starting point of the vehicle, which is (4,1), marking it as visited, and initializing the frontier (priority queue) containing the starting cell and its heuristic value.

The algorithm will enter a loop where it will choose the lowest heuristic value cell in the frontier, which means that it will pop the cell from the frontier with the lowest heuristic value. The heuristic value is calculated by the L1 distance from the current coordinate cell to the goal coordinate cell. Then, it checks if the goal cell has been reached. In this case, the cell (4, 1) is not the goal coordinate cell so the goal cell has not been reached. Then, it expands the current cell to generate its child cells in four directions with order: right, left, down, and up, which are (4, 2) – right, (4, 0) – left, (5, 1) – down, and (3,1) – up. Then, it will check if each child cell is visited by the current node, or

is the building block, or is out of bounds, it will return false. In this case, all the new cells are blank spaces so they can be visited by the current cell. For each valid neighbor, it updates the cell's properties and adds it to the priority queue with its heuristic value. In this case, we will choose cell (4,2) because it has the lowest heuristic value from the coordinate (4,2) to the goal coordinate (7, 6). And then it will continue to enter this loop.

After the GBFS loop completes, the algorithm checks if the goal cell was reached and, if so, traces the path from the goal cell back to the start cell using the parent references. If no valid path is found, it ensures the goal cell's visited status is reset.

output1_level1.txt	GUI for output1_level1
<p>AStarAlgorithm:</p> <p>S</p> <p>(4, 1) (3, 1) (3, 0) (2, 0)</p> <p>(1, 0) (1, 1) (0, 1) (0, 2)</p> <p>(0, 3) (0, 4) (0, 5) (1, 5)</p> <p>(2, 5) (3, 5) (4, 5) (5, 5)</p> <p>(6, 5) (7, 5) (7, 6)</p>	

It initializes the starting cell's properties, which is cell (4, 1), marking it as visited, setting its cost to zero, and associating it with the current vehicle. The method uses a priority queue (implemented with heapq) to manage the frontier, starting with the initial cell and its heuristic value.

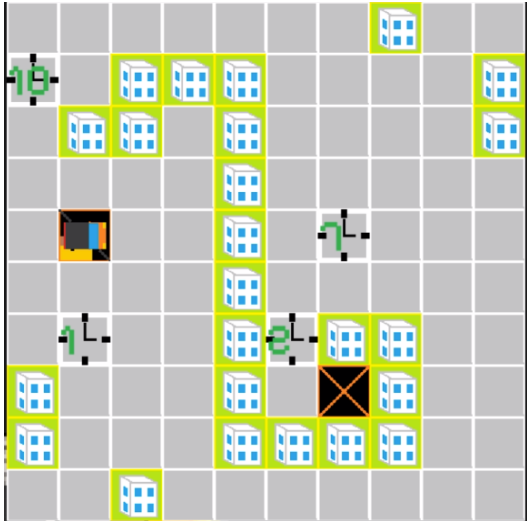
The algorithm will enter a loop where it will choose the lowest total cost cell in the frontier, which means that it will pop the cell from the frontier with the lowest total cost by the sum of path cost (true cost) and the heuristic value of this cell. The heuristic value is calculated by the L1 distance from the current coordinate cell to the goal coordinate cell. Then, it will check the current cell is the goal cell, which means that the goal cell has been reached. In this case, the cell (4, 1) is not the goal cell, so the algorithm will

For each cell, the algorithm explores its neighbors (right, left, down, up) and updates their properties if a lower-cost path is found.

Neighbor cells are then added to the priority queue with their total cost.

After exiting the loop, if the goal cell is visited, the method traces the path from the goal back to the start using the parent references. The execute method sets temporary start and goal coordinates for the vehicle, calls the a_star method to perform the search, and returns the path's time and fuel consumption.

- Level 2:
 - Input

input1_level2.txt	Level2 Image
<pre> 10 10 25 10 0 0 0 0 0 0 -1 0 0 10 0 -1 -1 -1 0 0 0 0 -1 0 -1 -1 0 -1 0 0 0 0 -1 0 0 0 0 -1 0 0 0 0 0 0 S 0 0 -1 0 7 0 0 0 0 0 0 0 -1 0 0 0 0 0 0 1 0 0 -1 2 -1 -1 0 0 -1 0 0 0 -1 0 G -1 0 0 -1 0 0 0 -1 -1 -1 -1 0 0 0 0 -1 0 0 0 0 0 0 0 </pre>	

In this map, there are cells where a vehicle is required to stop for $a[i, j]$ minutes to pay a road toll (a positive integer that is calculated according to the average waiting time of vehicles). Besides that, it contains many building blocks, walls, ... The delivery time limit is 25 minutes, which means this vehicle cannot move when the current time is greater than this time limit.

- Output

output1_level2.txt	GUI for output1_level2
--------------------	------------------------

(No GUI)	
<p>S</p> <p>(4, 1) (4, 2) (5, 2) (6, 2) (7, 2) (7, 3) (8, 3) (9, 3) (9, 4) (9, 5) (9, 6) (9, 7) (9, 8) (8, 8) (7, 8) (6, 8) (5, 8) (5, 7) (5, 6) (5, 5) (6, 5) (7, 5) (7, 6)</p>	

In this test case at level 2, we will use the A* algorithm, as described in the above part. The actual time when the vehicle travels is 22 minutes but this vehicle must pay 2 minutes when passing the toll booths with fee as 2 minutes.

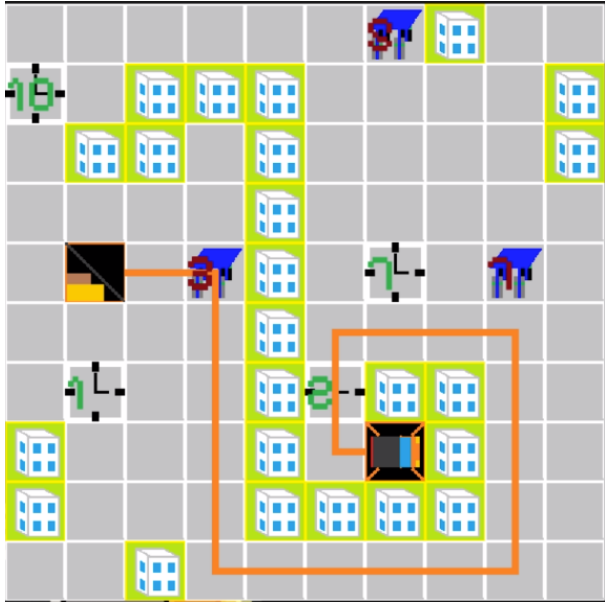
- Level3:
 - Input

input1 level3.txt	Level3 Image
<p>10 10 27 20</p> <p>0 0 0 0 0 F5 -1 0 0</p> <p>10 0 -1 -1 -1 0 0 0 0 -1</p> <p>0 -1 -1 0 -1 0 0 0 0 -1</p> <p>0 0 0 0 -1 0 0 0 0 0</p> <p>0 S 0 F3 -1 0 7 0 F7 0</p> <p>0 0 0 0 -1 0 0 0 0 0</p> <p>0 1 0 0 -1 2 -1 -1 0 0</p> <p>-1 0 0 0 -1 0 G -1 0 0</p> <p>-1 0 0 0 -1 -1 -1 -1 0 0</p> <p>0 0 -1 0 0 0 0 0 0 0</p>	

At this level, we need to pay attention to the limit of fuel and current fuel of the vehicle whenever it passes a cell because if the vehicle doesn't have enough vehicles, it will not be able to reach its goal.

So, we built some fuel stations on the map to refill his fuel, and the time needed to do that action (F3 for 3 minutes, F7 for 7 minutes). Now he needs to find the shortest path to the goal, in a short time with limited fuel.

○ Output

output1_level3.txt	GUI for output1_level3
<p>S</p> <p>(4, 1) (4, 2) (4, 3) (5, 3) (6, 3) (7, 3) (8, 3) (9, 3) (9, 4) (9, 5) (9, 6) (9, 7) (9, 8) (8, 8) (7, 8) (6, 8) (5, 8) (5, 7) (5, 6) (5, 5) (6, 5) (7, 5) (7, 6)</p>	

In this level, we use a start algorithm to find the path, it prioritizes the cells with the fewest cost, then the least of time and enough fuel. By this Order, it will find the shortest path first, if it is out of limit time, vehicle will find another path, if not enough fuel, it will go to and fuel station and continue to find path to the goal.

In this test case, if it doesn't go to the fuel station at (4, 3) he will never reach the goal (because not enough fuel, the shortest path is up to 22 cells, so it will need 22 liters of fuel, but the start and limit fuel is 20 liters).

If he moves to the F7 fuel station at (4, 8), which is nearer to the goal than the fuel station at (4, 3) he will make a longer path with a longer time waiting to refill his fuel, so he must go to (4, 3) fuel station to able to reach his goal.

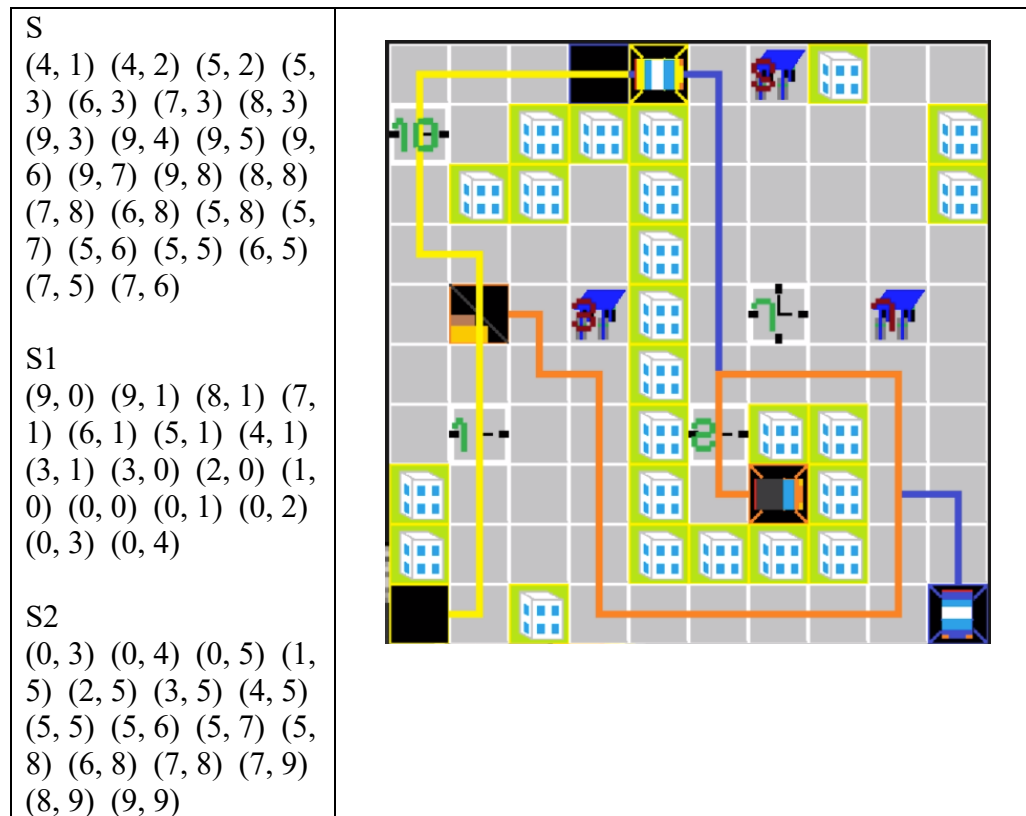
- Level4:
 - Input

input1_level4.txt	Level4 Image
10 10 100 100 G1 0 0 S2 0 0 F5 -1 0 0 10 0 -1 -1 -1 0 0 0 0 -1 0 -1 -1 0 -1 0 0 0 0 -1 0 0 0 0 -1 0 0 0 0 0 0 S 0 F3 -1 0 7 0 F7 0 0 0 0 0 -1 0 0 0 0 0 0 1 0 0 -1 2 -1 -1 0 0 -1 0 0 0 -1 0 G -1 0 0 -1 0 0 0 -1 -1 -1 -1 0 0 S1 0 -1 0 0 0 0 0 0 G2	

At this level, many vehicles are moving at the same time, so we need to check if the collisions happened. Then vehicles will need to wait or find another path to avoid collision. All vehicles will need to give way for the 'S' vehicle, because when he reaches the goal, the system will stop, or it will loop until out of limit time and the other vehicle will need to generate a new goal whenever they reach their goal.

- Output

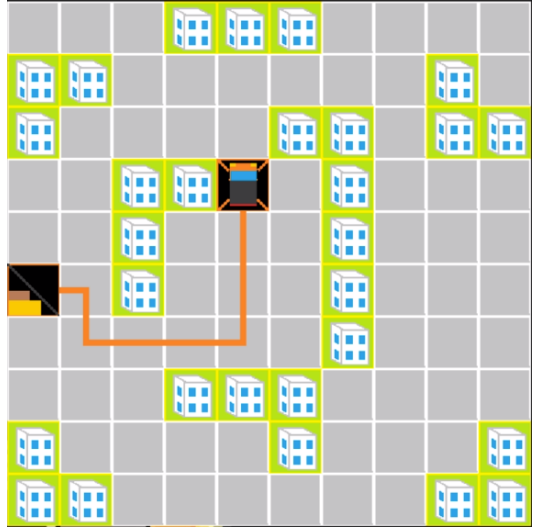
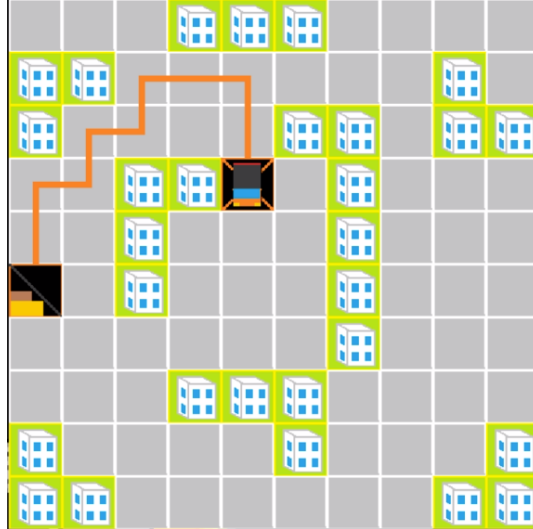
output1_level4.txt	GUI for output1_level4
--------------------	------------------------

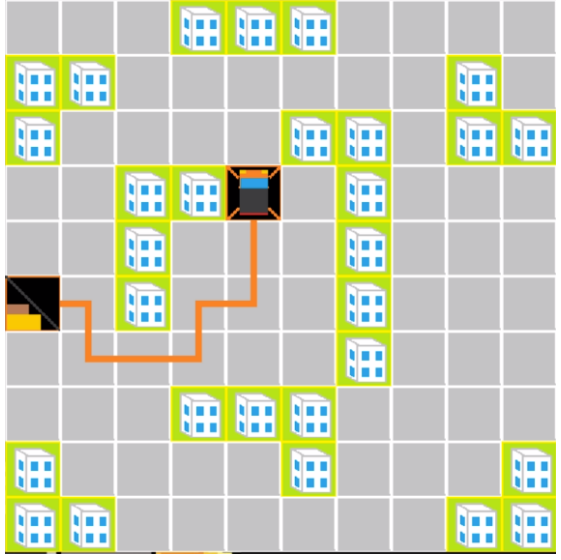
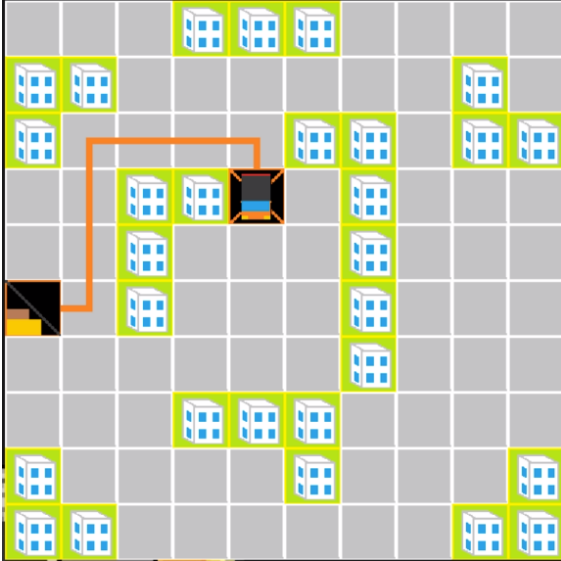


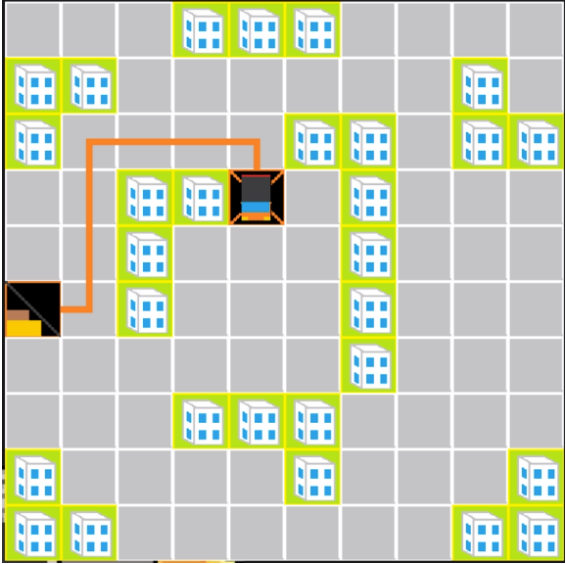
Blue vehicle (S2) makes a strange move at (7,8) to (7,9), but usually, he will go to (8,8) instead of (7,9) because (8,8) is nearer to the goal (small L2 distance, so it will have small heuristic value, it will have higher priority). But at the same time, the 'S' vehicle also moves to that cell, and the 'S' vehicle is the most priority object, so 'S2' will need to wait or find another path to reach his goal. But if he waits, at the next move 'S' will move to his current cell (because he is waiting, he will not move at the previous turn), and this will make a collision. So 'S2' needs to find another path, it leads to the strange move of 'S2'.

2. Test case evaluation

- Test case 2
 - Level 1

input2_level1.txt	output2_level1.txt	GUI
<pre> 10 10 15 8 0 0 0 -1 -1 -1 0 0 0 0 -1 -1 0 0 0 0 0 0 -1 0 -1 0 0 0 0 -1 -1 0 -1 -1 0 0 -1 -1 G 0 -1 0 0 0 0 0 -1 0 0 0 -1 0 0 0 S 0 -1 0 0 0 -1 0 0 0 0 0 0 0 0 0 -1 0 0 0 0 0 0 -1 -1 -1 0 0 0 0 -1 0 0 0 0 -1 0 0 0 -1 -1 -1 0 0 0 0 0 0 -1 -1 </pre>	<p>BFSAlgorithm:</p> <p>S</p> <p>(5, 0) (5, 1) (6, 1) (6, 2) (6, 3) (6, 4) (5, 4) (4, 4) (3, 4)</p>	
	<p>DFSAlgorithm:</p> <p>S</p> <p>(5, 0) (4, 0) (3, 0) (3, 1) (2, 1) (2, 2) (1, 2) (1, 3) (1, 4) (2, 4) (3, 4)</p>	

	<p>UCSAlgorithm:</p> <p>S</p> <p>(5, 0) (5, 1) (6, 1) (6, 2) (6, 3) (5, 3) (5, 4) (4, 4) (3, 4)</p>	
	<p>GBFSAlgorithm:</p> <p>S</p> <p>(5, 0) (5, 1) (4, 1) (3, 1) (2, 1) (2, 2) (2, 3) (2, 4) (3, 4)</p>	

	<p>AStarAlgorithm: S (5, 0) (5, 1) (4, 1) (3, 1) (2, 1) (2, 2) (2, 3) (2, 4) (3, 4)</p>	
--	---	--

○ Level 2

input2_level2.txt	output2_level2.txt	GUI
<pre> 10 10 15 8 0 0 0 -1 -1 -1 0 0 1 0 -1 -1 0 0 0 0 0 0 -1 0 -1 0 0 1 0 0 -1 -1 0 -1 -1 0 0 -1 -1 G 0 -1 0 0 0 0 0 -1 0 8 0 -1 0 0 0 S 0 -1 0 0 0 -1 0 2 0 0 0 0 0 0 0 -1 0 0 0 0 0 0 -1 -1 -1 0 0 0 0 -1 0 0 0 0 -1 0 0 0 -1 -1 -1 0 0 9 0 0 0 -1 -1 </pre>	<pre> S (5, 0) (5, 1) (6, 1) (6, 2) (6, 3) (5, 3) (5, 4) (5, 5) (4, 5) (3, 5) (3, 4) </pre>	

○ Level 3

input2_level3.txt	output2_level3.txt	GUI
<pre> 10 10 15 8 0 0 0 -1 -1 -1 0 0 0 0 -1 -1 0 0 0 0 0 F 8 -1 0 -1 0 0 1 0 F 2 -1 -1 0 -1 -1 0 0 -1 -1 G 0 -1 0 0 0 0 0 -1 0 8 0 -1 0 0 0 S 0 -1 0 0 0 -1 0 2 0 0 0 0 0 0 0 -1 0 0 0 0 F 3 0 -1 -1 -1 0 0 0 0 -1 0 0 0 0 -1 0 F 7 0 -1 -1 -1 0 0 9 0 0 0 -1 -1 </pre>	<pre> S (5, 0) (5, 1) (4, 1) (3, 1) (2, 1) (2, 2) (1, 2) (1, 3) (1, 4) (2, 4) (3, 4) </pre>	

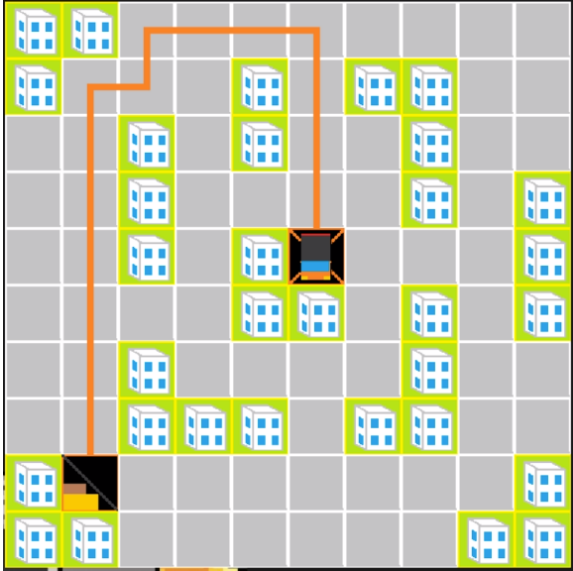
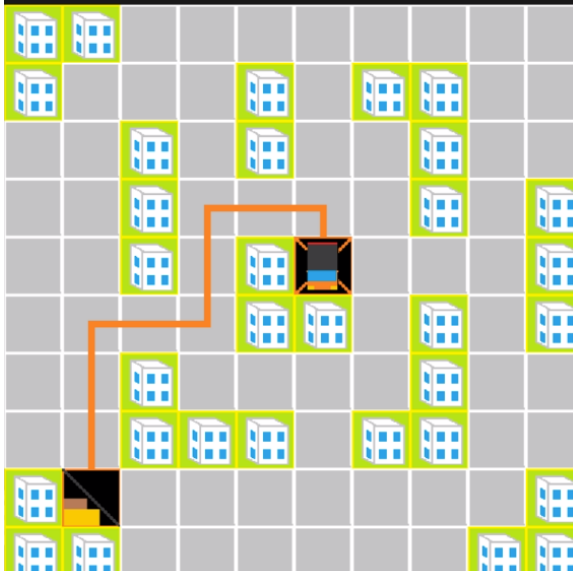
○ Level 4

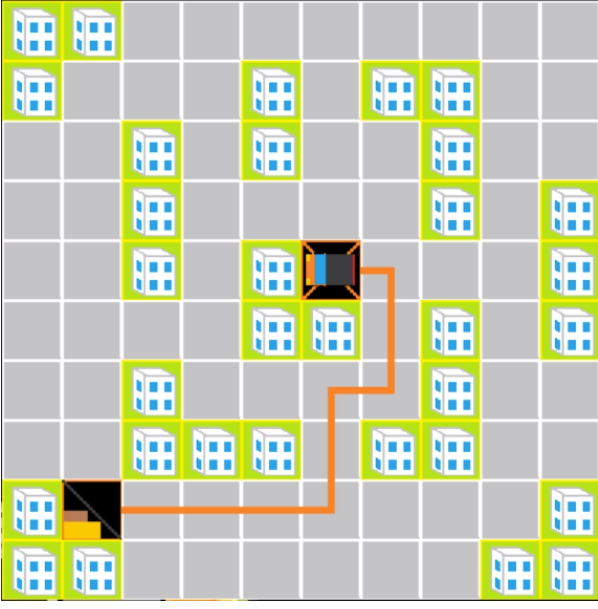
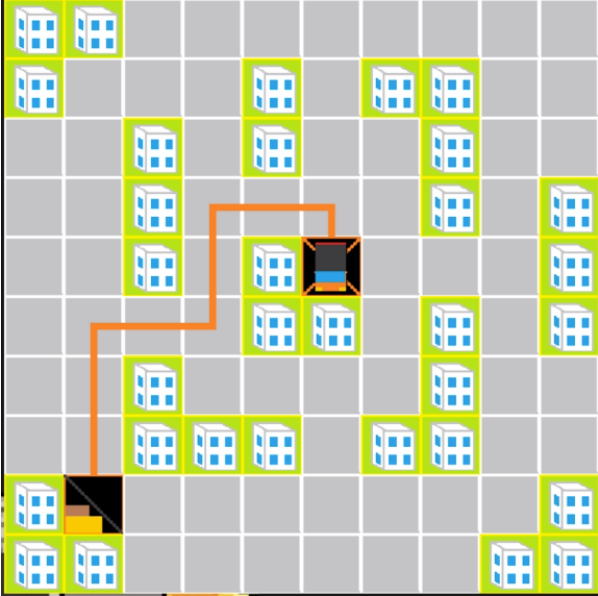
input2_level4.txt	output2_level4.txt	GUI
<pre> 10 10 100 100 S1 0 0 -1 -1 -1 S3 1 0 0 -1 -1 0 0 0 0 0 F8 -1 0 -1 G2 0 10 F2 -1 -1 0 -1 -1 0 0 -1 -1 G 0 -1 0 0 S2 0 0 -1 0 8 0 -1 0 0 0 S 0 -1 0 0 0 -1 0 2 0 0 0 0 0 0 0 -1 0 0 0 G3 F3 0 -1 -1 -1 0 0 0 0 -1 0 0 0 0 -1 0 F7 G1 -1 -1 -1 0 0 9 0 0 0 -1 -1 </pre>	<pre> S (5, 0) (5, 1) (6, 1) (6, 2) (6, 3) (6, 4) (5, 4) (5, 5) (4, 5) (3, 5) (3, 4) S1 No valid path found S2 No valid path found S3 No valid path found </pre>	

• Test case 3

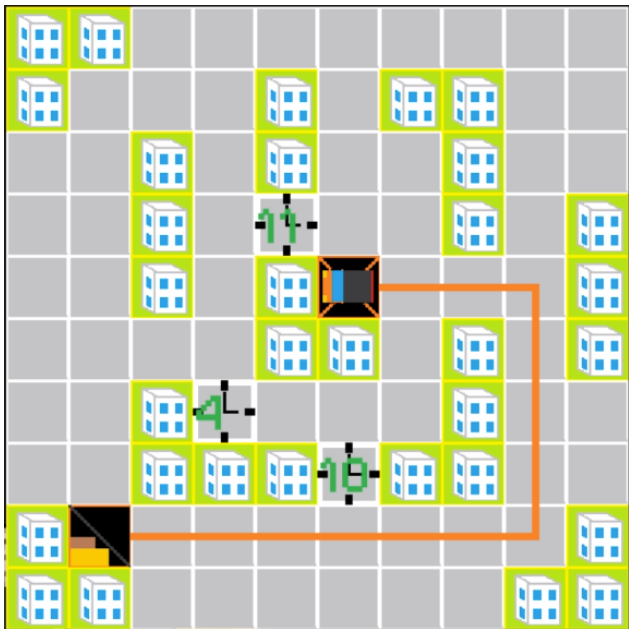
○ Level 1

input3_level1.txt	output3_level1.txt	GUI
<pre> 10 10 20 10 -1 -1 0 0 0 0 0 0 0 0 -1 0 0 0 -1 0 -1 -1 0 0 0 0 -1 0 -1 0 0 -1 0 0 0 0 -1 0 0 0 0 -1 0 -1 0 0 -1 0 -1 G 0 0 0 -1 0 0 0 0 -1 -1 0 -1 0 -1 0 0 -1 0 0 0 0 -1 0 0 0 0 -1 -1 -1 0 -1 -1 0 0 -1 S 0 0 0 0 0 0 0 -1 -1 -1 0 0 0 0 0 0 -1 -1 </pre>	<pre> BFSAlgorithm: S (8, 1) (8, 2) (8, 3) (8, 4) (8, 5) (7, 5) (6, 5) (6, 6) (5, 6) (4, 6) (4, 5) </pre>	

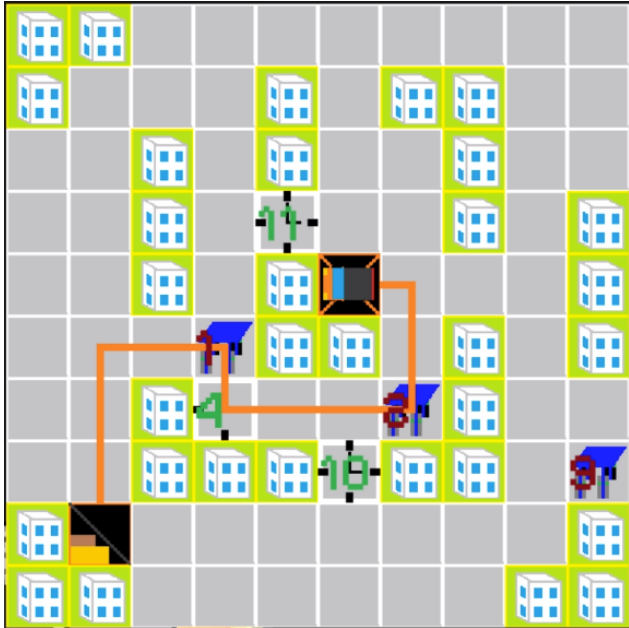
	<p>DFSAlgorithm:</p> <p>S</p> <p>(8, 1) (7, 1) (6, 1) (5, 1) (4, 1) (3, 1) (2, 1) (1, 1) (1, 2) (0, 2) (0, 3) (0, 4) (0, 5) (1, 5) (2, 5) (3, 5) (4, 5)</p>	
	<p>UCSAlgorithm:</p> <p>S</p> <p>(8, 1) (7, 1) (6, 1) (5, 1) (5, 2) (5, 3) (4, 3) (3, 3) (3, 4) (3, 5) (4, 5)</p>	

	<p>GBFSAlgorithm:</p> <p>S</p> <p>(8, 1) (8, 2) (8, 3) (8, 4) (8, 5) (7, 5) (6, 5) (6, 6) (5, 6) (4, 6) (4, 5)</p>	
	<p>AStarAlgorithm:</p> <p>S</p> <p>(8, 1) (7, 1) (6, 1) (5, 1) (5, 2) (5, 3) (4, 3) (3, 3) (3, 4) (3, 5) (4, 5)</p>	

○ Level 2

input3_level2.txt	output3_level2.txt	GUI
<pre> 10 10 20 10 -1 -1 0 0 0 0 0 0 0 -1 0 0 0 -1 0 -1 -1 0 0 0 0 -1 0 -1 0 0 -1 0 0 0 0 -1 0 1 1 0 0 -1 0 -1 0 0 -1 0 -1 0 0 0 -1 0 0 0 0 -1 -1 0 -1 0 -1 0 0 -1 4 0 0 0 -1 0 0 0 0 -1 -1 -1 1 0 -1 -1 0 0 -1 S 0 0 0 0 0 0 0 -1 -1 -1 0 0 0 0 0 0 -1 -1 </pre>	<pre> S (8, 1) (8, 2) (8, 3) (8, 4) (8, 5) (8, 6) (8, 7) (8, 8) (7, 8) (6, 8) (5, 8) (4, 8) (4, 7) (4, 6) (4, 5) </pre>	

○ Level 3

input3_level3.txt	output3_level3.txt	GUI
<pre> 10 10 20 10 -1 -1 0 0 0 0 0 0 0 -1 0 0 0 -1 0 -1 -1 0 0 0 0 -1 0 -1 0 0 -1 0 0 0 0 -1 0 1 1 0 0 -1 0 -1 0 0 -1 0 -1 0 0 0 -1 0 0 0 F1 -1 -1 0 -1 0 -1 0 0 -1 4 0 0 F2 -1 0 0 0 0 -1 -1 -1 1 0 -1 -1 0 F9 -1 S 0 0 0 0 0 0 0 -1 -1 -1 0 0 0 0 0 0 -1 -1 </pre>	<pre> S (8, 1) (7, 1) (6, 1) (5, 1) (5, 2) (5, 3) (6, 3) (6, 4) (6, 5) (6, 6) (5, 6) (4, 6) (4, 5) </pre>	

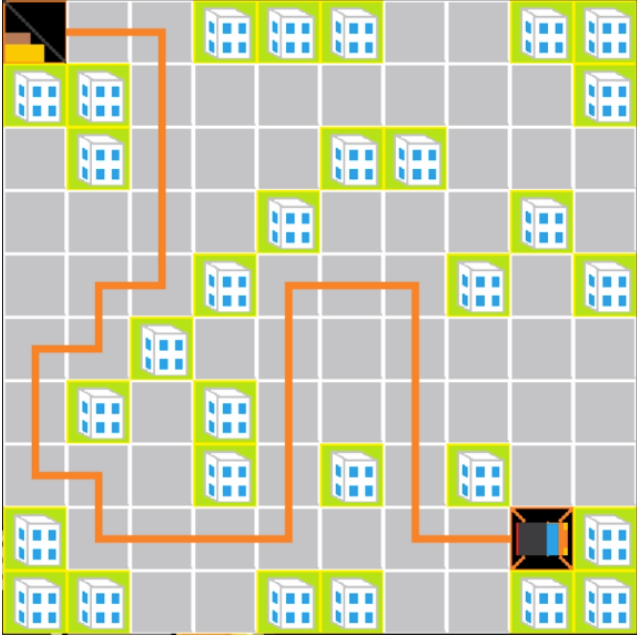
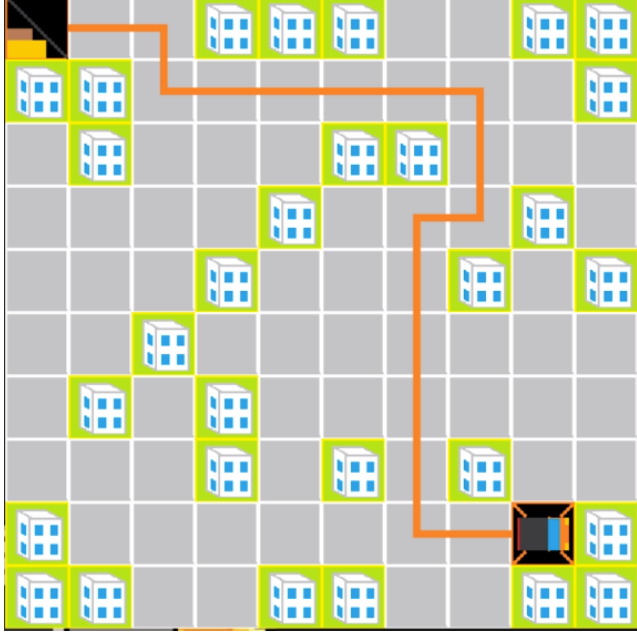
○ Level 4

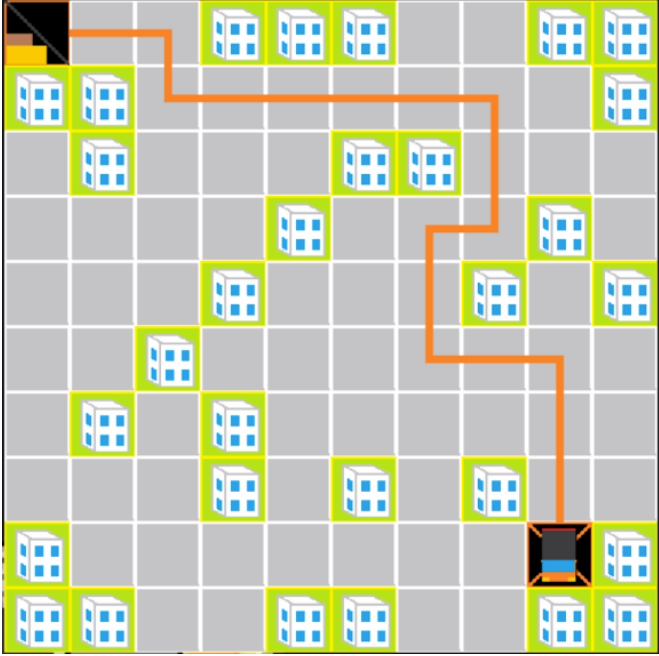
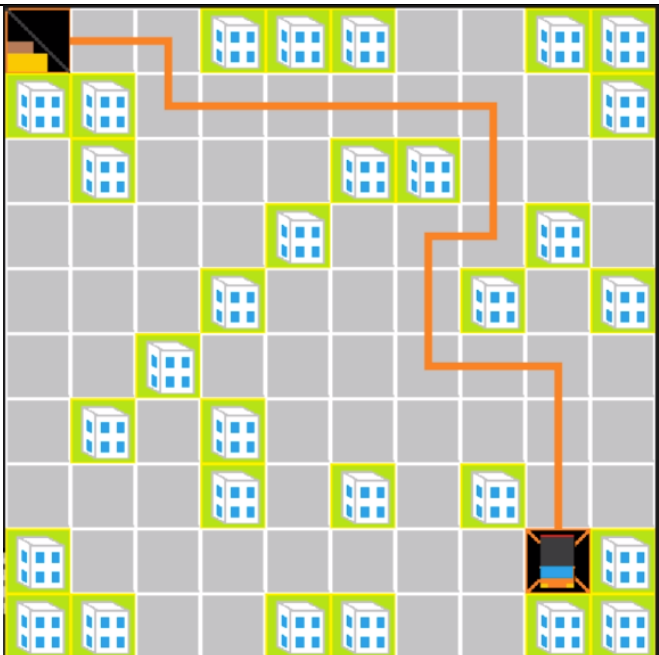
input3_level4.txt	output3_level4.txt	GUI
<pre> 10 10 20 10 -1 -1 0 0 0 0 0 0 S2 -1 S1 0 0 -1 0 -1 -1 0 0 0 0 -1 0 -1 0 0 -1 0 0 0 0 -1 0 1 1 0 0 -1 0 -1 0 0 -1 0 -1 G 0 0 0 -1 0 G1 G2 F1 -1 -1 0 -1 0 -1 0 0 -1 4 0 0 F2 -1 0 0 0 0 -1 -1 -1 10 -1 -1 0 F9 -1 S 0 0 0 0 0 0 0 -1 -1 -1 0 0 0 0 0 0 -1 -1 </pre>	<pre> S (0, 0) (0, 1) (0, 2) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (2, 7) (3, 7) (3, 6) (4, 6) (5, 6) (6, 6) (7, 6) (8, 6) (8, 7) (8, 8) </pre>	

• Test case 4

○ Level 1

input4_level1.txt	output4_level1.txt	GUI
<pre> 10 10 20 20 S 0 0 -1 -1 -1 0 0 -1 -1 -1 -1 0 0 0 0 0 0 -1 1 0 -1 0 0 0 -1 -1 0 0 0 0 0 0 0 -1 0 0 0 -1 0 0 0 0 -1 0 0 0 -1 0 -1 1 0 0 -1 0 0 0 0 0 0 0 0 -1 0 -1 0 0 0 0 0 0 0 0 0 -1 0 -1 0 -1 0 0 -1 0 0 0 0 0 0 0 G - 1 </pre>	<pre> BFSAlgorithm: S (0, 0) (0, 1) (0, 2) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (2, 7) (3, 7) (3, 6) (4, 6) (5, 6) (5, 7) (5, 8) (6, 8) (7, 8) (8, 8) </pre>	

<p>-1 -1 0 0 -1 -1 0 0 - 1 -1</p>	<p>DFSAlgorithm: S (0, 0) (0, 1) (0, 2) (1, 2) (2, 2) (3, 2) (4, 2) (4, 1) (5, 1) (5, 0) (6, 0) (7, 0) (7, 1) (8, 1) (8, 2) (8, 3) (8, 4) (7, 4) (6, 4) (5, 4) (4, 4) (4, 5) (4, 6) (5, 6) (6, 6) (7, 6) (8, 6) (8, 7) (8, 8)</p>	
	<p>UCSAlgorithm: S (0, 0) (0, 1) (0, 2) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (2, 7) (3, 7) (3, 6) (4, 6) (5, 6) (6, 6) (7, 6) (8, 6) (8, 7) (8, 8)</p>	

	<p>GBFSAlgorithm:</p> <p>S</p> <p>(0, 0) (0, 1) (0, 2) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (2, 7) (3, 7) (3, 6) (4, 6) (5, 6) (5, 7) (5, 8) (6, 8) (7, 8) (8, 8)</p>	
	<p>AStarAlgorithm:</p> <p>S</p> <p>(0, 0) (0, 1) (0, 2) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (2, 7) (3, 7) (3, 6) (4, 6) (5, 6) (5, 7) (5, 8) (6, 8) (7, 8) (8, 8)</p>	

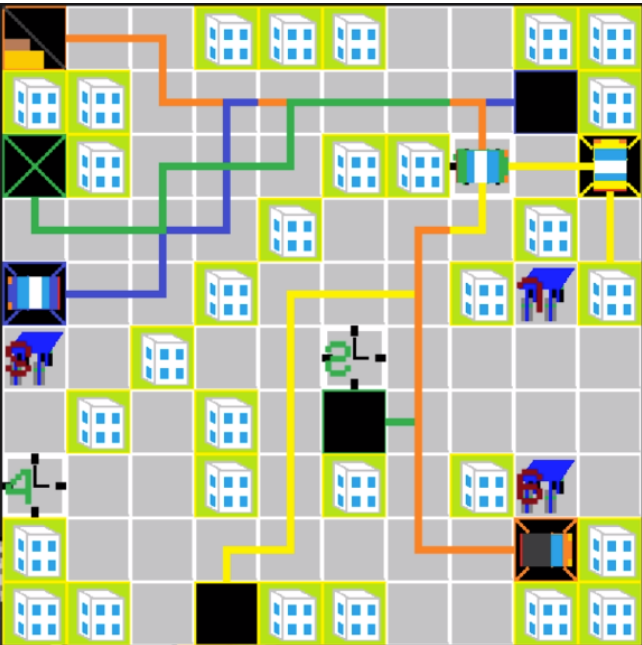
○ Level 2

input4_level2.txt	output4_level2.txt	GUI
<pre> 10 10 20 20 S 0 0 -1 -1 -1 0 0 -1 -1 -1 -1 0 0 0 0 0 0 0 -1 0 -1 0 0 0 -1 -1 1 0 0 0 0 0 0 -1 0 0 0 -1 0 0 0 0 -1 0 0 0 -1 0 -1 0 0 -1 0 0 2 0 0 0 0 0 -1 0 -1 0 0 0 0 0 0 4 0 0 -1 0 -1 0 -1 0 0 -1 0 0 0 0 0 0 0 G -1 -1 -1 0 0 -1 -1 0 0 -1 -1 </pre>	<pre> S (0, 0) (0, 1) (0, 2) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (2, 7) (3, 7) (3, 6) (4, 6) (5, 6) (5, 7) (6, 7) (6, 8) (7, 8) (8, 8) </pre>	<p>The GUI displays a 10x10 grid. Obstacles are represented by blue cubes. A start point 'S' is at (0,0) and a goal point 'G' is at (8,8). An orange line shows the path from S to G, passing through (0,1), (1,2), (1,3), (1,5), (1,6), (2,7), (3,7), (4,6), (5,6), (6,7), and (6,8).</p>

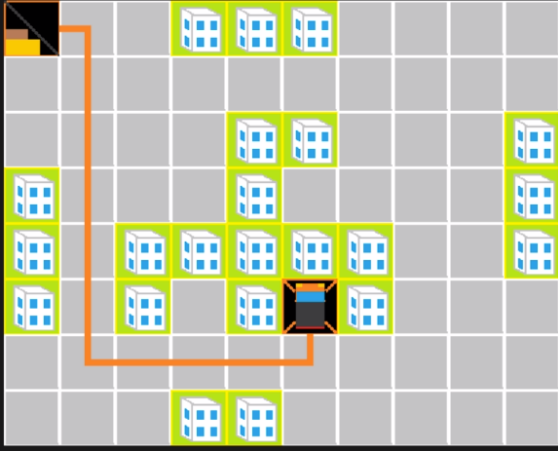
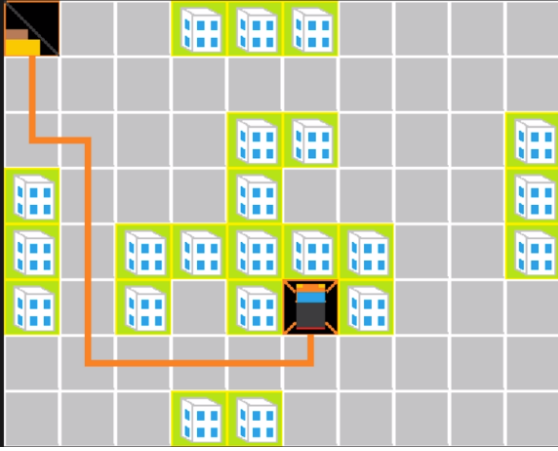
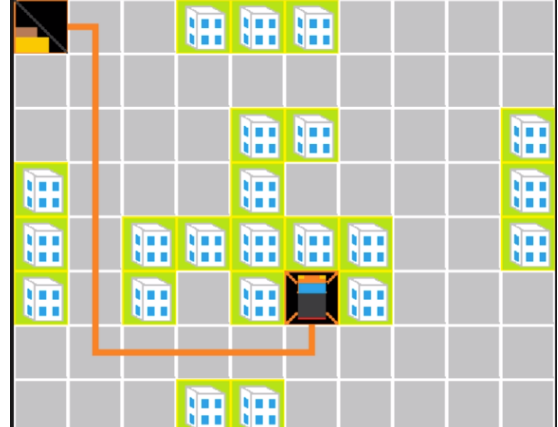
○ Level 3

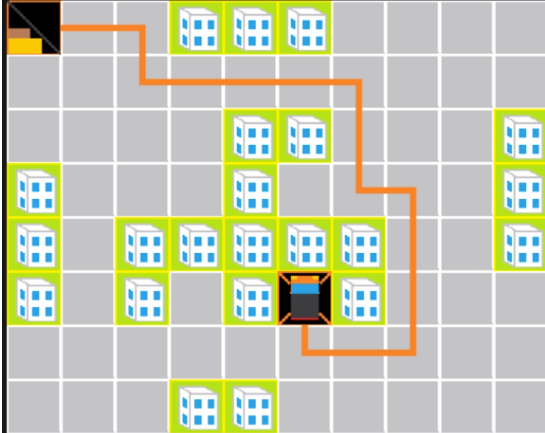
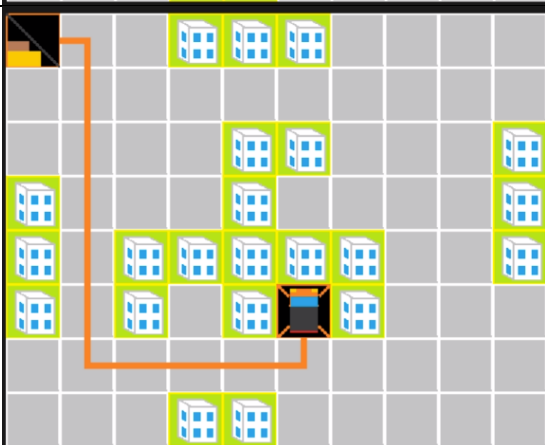
input4_level3.txt	output4_level3.txt	GUI
<pre> 10 10 20 20 S 0 0 -1 -1 -1 0 0 -1 -1 -1 -1 0 0 0 0 0 0 0 -1 0 -1 0 0 0 -1 -1 1 0 0 0 0 0 0 -1 0 0 0 -1 0 0 0 0 -1 0 0 0 -1 F7 -1 F5 0 -1 0 0 2 0 0 0 0 0 -1 0 -1 0 0 0 0 0 0 4 0 0 -1 0 -1 0 -1 F6 0 -1 0 0 0 0 0 0 0 G -1 -1 -1 0 0 -1 -1 0 0 -1 -1 </pre>	<pre> S (0, 0) (0, 1) (0, 2) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (2, 7) (3, 7) (3, 6) (4, 6) (5, 6) (6, 6) (7, 6) (8, 6) (8, 7) (8, 8) </pre>	<p>The GUI displays a 10x10 grid. Obstacles are represented by blue cubes. A start point 'S' is at (0,0) and a goal point 'G' is at (8,8). An orange line shows the path from S to G, passing through (0,1), (1,2), (1,3), (1,5), (1,6), (2,7), (3,7), (4,6), (5,6), (6,6), (7,6), and (8,6). There are additional markers 'F7' at (8,1) and 'F6' at (8,7).</p>

○ Level 4

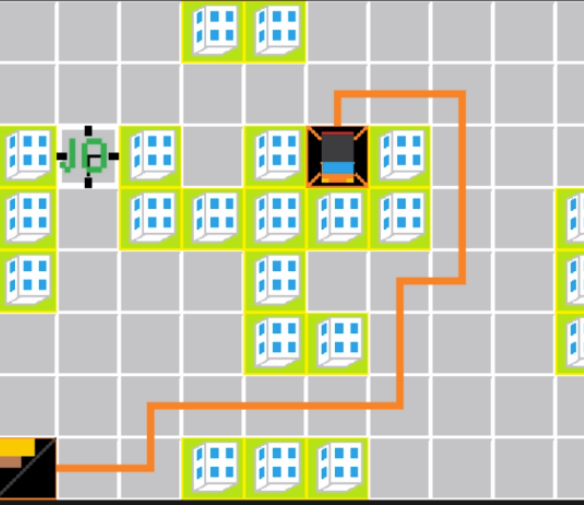
input4_level4.txt	output4_level4.txt	GUI
10 10 20 20 S 0 0 -1 -1 -1 0 0 -1 -1 -1 -1 0 0 0 0 0 0 S2 -1 G3 -1 0 0 0 -1 -1 1 0 0 0 0 0 0 -1 0 0 0 -1 G1 0 0 G2 -1 0 0 0 -1 F7 -1 F5 0 -1 0 0 2 0 0 0 0 0 -1 0 -1 0 S3 0 0 0 0 4 0 0 -1 0 -1 0 -1 F6 0 -1 0 0 0 0 0 0 0 G -1 -1 -1 0 S1 -1 -1 0 0 -1 -1	S (0, 0) (0, 1) (0, 2) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (1, 7) (2, 7) (3, 7) (3, 6) (4, 6) (5, 6) (6, 6) (7, 6) (8, 6) (8, 7) (8, 8) S1 (9, 3) (8, 3) (8, 4) (7, 4) (6, 4) (5, 4) (4, 4) (4, 5) (4, 6) (3, 6) (3, 7) (2, 7) (2, 8) (2, 9) (3, 9) (2, 9) S2 (1, 8) (1, 7) (1, 6) (1, 5) (1, 4) (1, 3) (2, 3) (3, 3) (3, 2) (4, 2) (4, 1) (4, 0) S3 (6, 5) (6, 6) (5, 6) (4, 6) (3, 6) (3, 7) (2, 7) (1, 7) (1, 6) (1, 5) (1, 4) (2, 4) (2, 3) (2, 2) (3, 2) (3, 1) (3, 0) (2, 0) (2, 7)	

- Test case 5
 - Level 1

input5_level1.txt	output5_level1.txt	GUI
	BFSAlgorithm: S (0, 0) (0, 1) (1, 1) (2, 1) (3, 1) (4, 1) (5, 1) (6, 1) (6, 2) (6, 3) (6, 4) (6, 5) (5, 5)	
8 10 20 10 S 0 0 -1 -1 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 -1 0 0 0 -1 -1 0 0 0 -1 0 0 0 0 -1 -1 0 -1 -1 -1 -1 -1 0 0 -1 -1 0 -1 0 -1 G -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 -1 0 0 0 0 0	DFSAlgorithm: S (0, 0) (1, 0) (2, 0) (2, 1) (3, 1) (4, 1) (5, 1) (6, 1) (6, 2) (6, 3) (6, 4) (6, 5) (5, 5)	
	UCSAlgorithm: S (0, 0) (0, 1) (1, 1) (2, 1) (3, 1) (4, 1) (5, 1) (6, 1) (6, 2) (6, 3) (6, 4) (6, 5) (5, 5)	

	<p>GBFSAlgorithm:</p> <p>S</p> <p>(0, 0) (0, 1) (0, 2) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (2, 6) (3, 6) (3, 7) (4, 7) (5, 7) (6, 7) (6, 6) (6, 5) (5, 5)</p>	
	<p>AStarAlgorithm:</p> <p>S</p> <p>(0, 0) (0, 1) (1, 1) (2, 1) (3, 1) (4, 1) (5, 1) (6, 1) (6, 2) (6, 3) (6, 4) (6, 5) (5, 5)</p>	

○ Level 2

input5_level2.txt	output5_level2.txt	GUI
<pre> 8 10 20 10 S 0 0 -1 -1 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 -1 0 0 0 -1 -1 0 0 0 -1 0 0 0 0 -1 -1 0 -1 -1 -1 -1 -1 0 0 -1 -1 10 -1 0 -1 G -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 -1 0 0 0 0 0 </pre>	<p>S</p> <p>(0, 0) (0, 1) (0, 2) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (2, 6) (3, 6) (3, 7) (4, 7) (5, 7) (6, 7) (6, 6) (6, 5) (5, 5)</p>	

○ Level 3

input5_level3.txt	output5_level3.txt	GUI
<pre> 8 10 20 10 S 0 0 -1 -1 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 -1 0 0 0 -1 -1 0 0 0 -1 0 0 0 0 -1 -1 0 -1 -1 -1 -1 -1 F2 0 -1 -1 10 -1 0 -1 G -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1 -1 0 0 0 0 </pre>	<pre> S (0, 0) (0, 1) (0, 2) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (2, 6) (3, 6) (3, 7) (4, 7) (5, 7) (6, 7) (6, 6) (6, 5) (5, 5) </pre>	

○ Level 4

input5_level4.txt	output5_level4.txt	GUI
<pre> 8 10 20 10 S 0 0 -1 -1 -1 0 0 0 0 0 0 0 0 0 G1 0 0 0 0 0 0 0 0 -1 -1 0 0 0 -1 -1 0 0 0 -1 0 0 0 0 -1 -1 0 -1 -1 -1 -1 -1 F2 0 -1 -1 10 -1 0 -1 G -1 0 0 0 0 S1 0 0 0 0 0 0 0 0 0 0 -1 -1 0 0 0 0 </pre>	<pre> S (0, 0) (0, 1) (0, 2) (1, 2) (1, 3) (1, 4) (1, 5) (1, 6) (2, 6) (3, 6) (3, 7) (4, 7) (5, 7) (6, 7) (6, 6) (6, 5) (5, 5) S1 (6, 1) (6, 2) (6, 3) (6, 4) (6, 5) (6, 6) (6, 7) (5, 7) (4, 7) (3, 7) (2, 7) (1, 7) (1, 6) (1, 5) (1, 6) (2, 6) (3, 6) (3, 7) (3, 8) </pre>	

• Comments

- If the map is very large, it takes a very long time to reach the goal point of the vehicles.

- If the algorithm can not find the path for this vehicle, it will return the empty path, and the system program will return the message “No Path Found”.
- At level 1, we can see that:
 - A* and UCS can find the best path with the lowest cost values of these cells. Otherwise,
 - Sometimes, GBFS can find the best path with the lowest heuristic values.
 - BFS and DFS cannot find an efficient path for this vehicle.
- At level 2, the A* algorithm is the best way to find the path with the time constraint. If the time constraint is lower than the time limit, the vehicle can find the path. If not, the algorithm will return the empty path and the system will return the message “No Path Found”.
- At level 3, the A* algorithm is the best way to find the path with the time and fuel constraints.
- At level 4, the A* algorithm is the best way to find the path with the time, fuel constraints, and vehicle constraints.

REFERENCES

During the process of researching and implementing the **project #1: Searching – Delivery System**, our team used and referenced some of the following open and electronic documents:

Programming

- [1] [Breadth First Search or BFS for a Graph - Geeksforgeeks](#) *(last updated: 20/07/2024)*
- [2] [Depth First Search or DFS for a Graph - Geeksforgeeks](#) *(last updated: 20/07/2024)*
- [3] [Uniform-Cost Search \(Dijkstra for large Graphs\) - Geeksforgeeks](#) *(last updated: 20/07/2024)*
- [4] [A* Search Algorithm – Geeksforgeeks](#) *(last updated: 20/07/2024)*

Report

- [5] Slides CSC14003 – Introduction to Artificial Intelligence 2024 – Nguyen Ngoc Thao – Nguyen Hai Minh (Access date: 14/07/2024 *(last updated: 27/07/2024)*)

In addition, the demo video about recording the running process of our program for each test case and our source code should be mentioned in this part:

Demo Video

- [6] Youtube: [Link here](#)

Source Code

- [7] Github: [Link here](#)