

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN – ĐHQG TP HCM

KHOA CÔNG NGHỆ THÔNG TIN

Lớp 22CLC07



PROJECT 1: SEARCHING

Delivery System

Lecturer: Nguyễn Tiến Huy

Nguyễn Trần Duy Minh

Bùi Duy Đăng

Table of Contents

1.	Member Introduction.....	3
2.	Words of Appreciation	3
3.	Work division and progress percentage	3
4.	Graphical User Interface	4
5.	Functions that link the GUI to algorithms and support functions	9
6.	Compenents	10
a.	Node.py	10
b.	helper.py.....	11
7.	Level modes.....	14
a.	Level 1	14
b.	Level 2	18
c.	Level 3	19
8.	Test cases and results.....	20
a.	Level 1:.....	20
b.	Level 2:.....	28
c.	Level 3:.....	31
d.	Output file.....	36
9.	References	37
10.	Demo.....	38

1. Member Introduction

MSSV	Name
22127479	Lê Hoàng Lĩnh
22127103	Lê Thị Hồng Hạnh
22127278	Vũ Thu Minh

2. Words of Appreciation

We would like to thank the lecturers for their dedicated guidance, because this is the first time to do a project like this, so there may be mistakes, we hope that the lecturers can consider ignoring and commenting. Respect

3. Work division and progress percentage

Tasks	MSSV	Progress percentage (100%)
Code		
Algorithm: BFS, DFS, UCS, A*, GBFS	22127103	100%
Level 1,2,3	22127103	100%
Reading input file, Writing output file	22127278	100%
UI		
Background, buttons to enter the Algorithm/Input/Back, Interface Alignment, Insert input test case	22127479	100%
Buttons to Enter the input Maps drawing: level1, level2, level3 Insert input test case	22127278	100%
Report		
Align report Table of contents parts 1,2,3, parts 4:	22127479	100%

background, a half part 8 Video demo.9		
Table of contents parts 4: map, 5, a half part 8, 9	22127278	100%
Table of contents parts 6,7,9	22127103	100%

4. Graphical User Interface

Level modes	Progress percentage (100%)
Level 1	100%
Level 2	100%
Level 3	100%
Level 4	0%

❖ **Libraries used:** Tkinter and pillow.

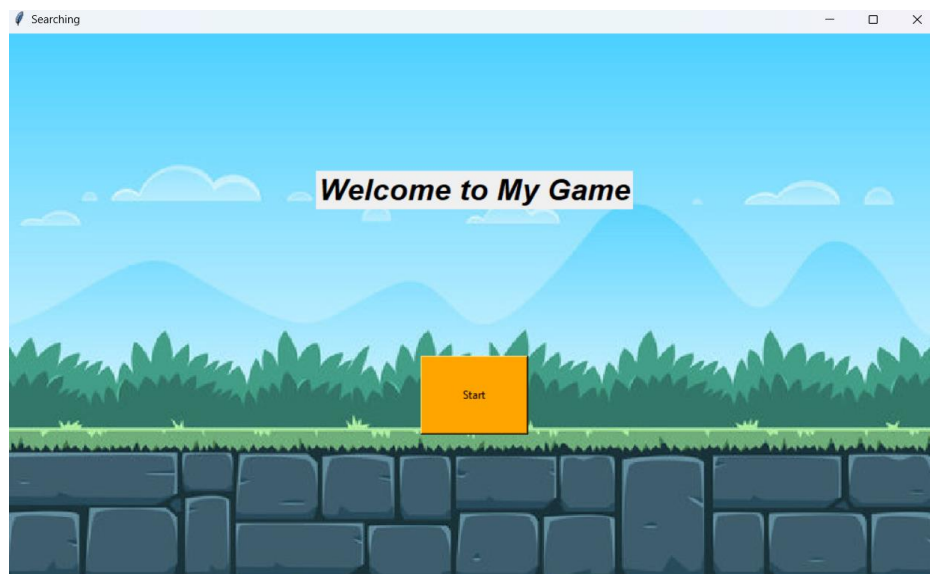
❖ **Start background**

- Set window width and height which are 1000 and 600.
- Create and a basic Tkinter root window with “Searching” title.
- Use the Pillow library to read the image from the background.jpg file, then resize it to fit the screen dimensions use `resize()` function, and finally convert it to a format that Tkinter can use with the `PhotoImage()` function.
- Create a widget label “Welcome to my game” with font Aria, font size 24 and style bolditalic, center the label, use `grid()` function to put the label into desired position.
- Start button with width 15 and height 5, and linked to the `start_game(start_button, title_label, root)` function. The “lambda” in command argument help we keep the function stop to wait we click at “Start”. Use `grid()` function to put the label into desired position.

- Run the main loop to keep Tkinter running.



4.1 Background.jpg.

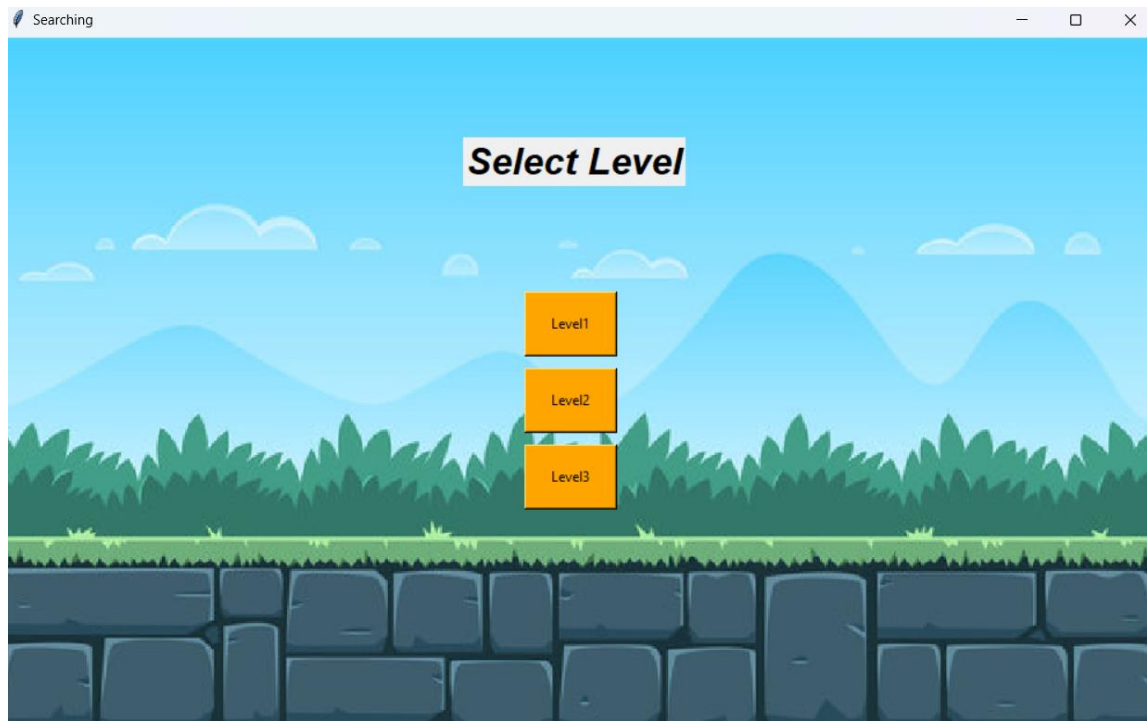


4.2 Background of game.

❖ Level background

- When we click the Start button, we turn to level background.
- Before starting, we need to hide the previous buttons and labels using the `grid_forget()` method include `start_button` and `title_label`.

- Create a widget label “Select level” with font Aria, font size 24 and style bolditalic, center the label, use grid() function to put the label into desired position..
- Level1 button with width 10 and height 3, and linked to the algorithm(start_label, level1_button, level2_button, level3_button, level4_button, root) function. The “lambda” in command argument help we keep the function stop to wait we click at “level1”. Use grid() function to put the label into desired position.
- Level2 button with width 10 and height 3, and linked to the level2() function. The “lambda” in command argument help we keep the function stop to wait we click at “level2”. Use grid() function to put the label into desired position.
- Level3 button with width 10 and height 3, and linked to the level3() function. The “lambda” in command argument help we keep the function stop to wait we click at “level3”. Use grid() function to put the label into desired position.

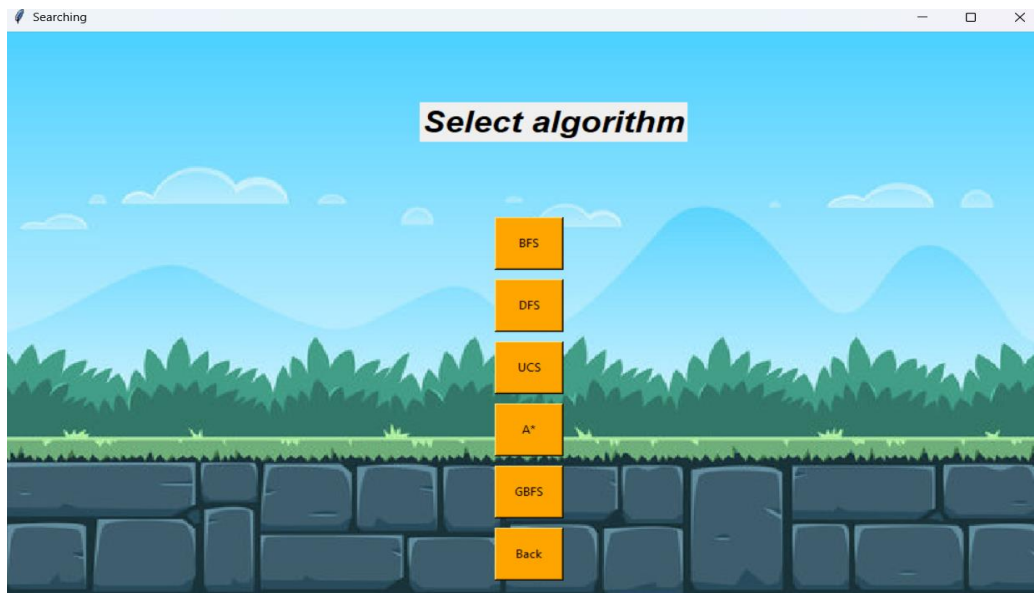


4.3 Select level background.

❖ Algorithm background

- When we click the Level1 button, we turn to algorithm background.
- Before starting, we need to hide the previous buttons and labels using the grid_forget() method include start_label, level1_button, level2_button, level3_button and level4_button.

- Create a widget label “Select algorithm” with font Aria, font size 24 and style bolditalic, center the label, use grid() function to put the label into desired position..
- BFS button with width 8 and height 3, and linked to the level1_bfs() function. The “lambda” in command argument help we keep the function stop to wait we click at “BFS”. Use grid() function to put the label into desired position.
- The remaining buttons should be set up in the same way as the button above: DFS button linked to the level1_dfs() function, UCS button linked to the level1_ucs() function, A* button linked to the level1_astar() function and GBFS link to the level1_gbfs() function.
- Return button with width 8 and height 3, and linked to the return_start() function. The “lambda” in command argument help we keep the function stop to wait we click at “BFS”. Use grid() function to put the label into desired position.
- Return_start(start1_label,BFS_button,UCS_button,Astar_button, GBFS_button, return_button, root) function: hide the previous button or label include start1_label, BFS_button, DFS_button, UCS_button, A*_button, GBFS_button, return_button. Use grid() to show the buttons on the “Select Level” background include: start_label, level1_button, level2_button, level3_button, level4_button.



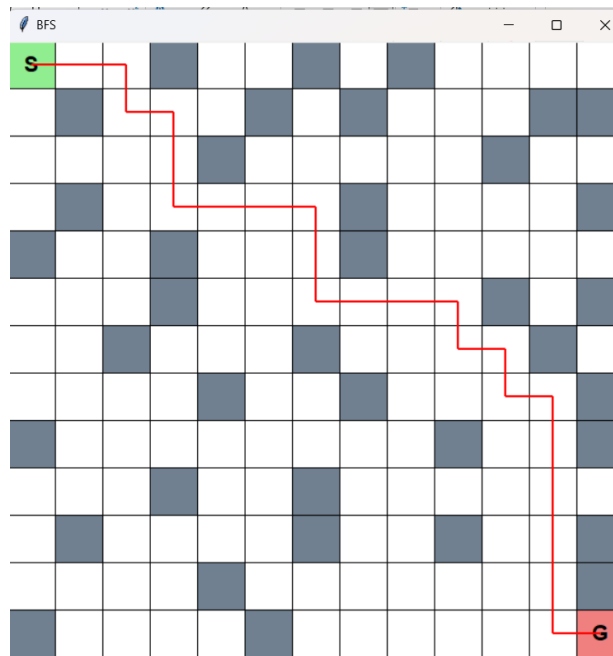
4.4 Select algorithm background.

❖ Map

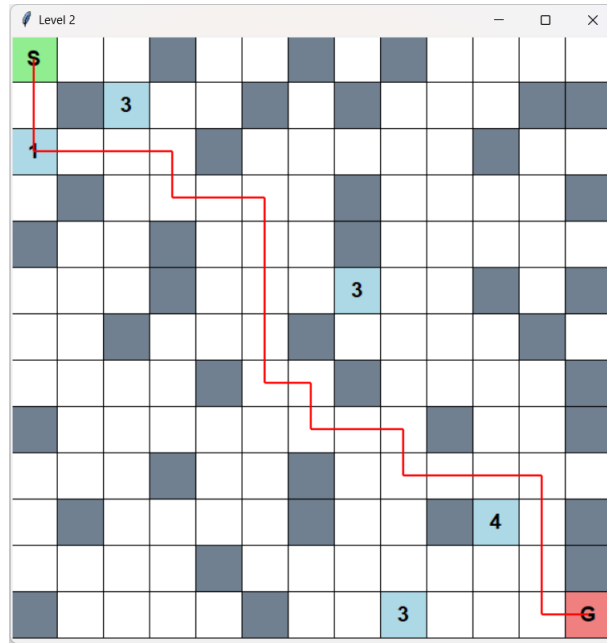
- The `__init__` function with parameters `self`, `root`, `mpa_matrix`, `n`, `m` is used to initialize the map. Call `tk.canvas` and `canvas.pack()` to create a 'Canvas' object of size 600 x 600 pixels and display it on the interface. Initialize the size of each square on the map by taking the smallest value of the interface window length divided by the number of map columns and the interface window height divided by the number of

map rows, this ensures that each tile in the map is a square. Then call the `draw_map` function to draw the map.

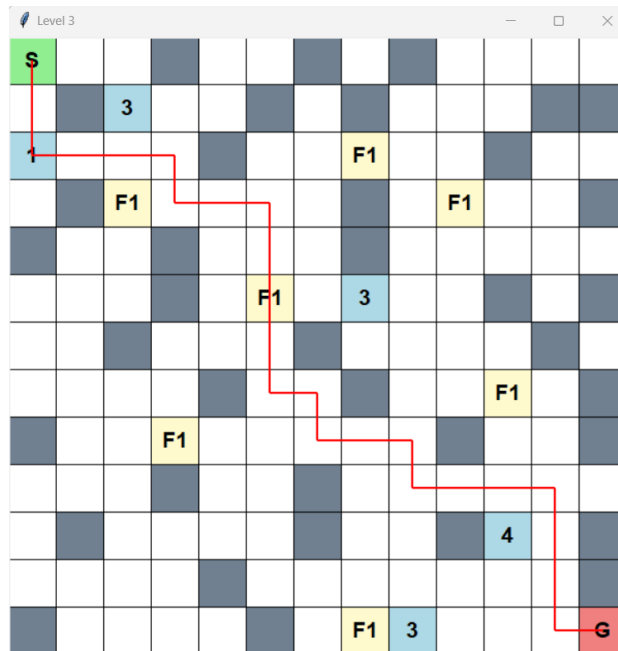
- The `draw_map` function takes `self`, `n`, `m` as arguments. Where `n` is the number of rows and `m` is the number of columns. Use the `canvas.delete('all')` command to delete all existing objects on the 'Canvas' to ensure that the new map will be drawn from scratch without overlapping the old objects. Iterate through each element of the map to determine the color for each square, with the first character 'S' colored light green, the 'G' colored light red, the 'F' colored lemon chiffon, the cells containing numbers greater than 0 colored light blue, the '-1' colored slate gray, the '0' colored white. Use the `canvas.create_rectangle` method to draw the squares on the map. Finally, check to see if the element starts with 'S', 'G', 'F' or a number greater than 0, then write text to the corresponding square using the `canvas.create_text` method.
- The `draw_path` function draws a path from S to G, it uses the list of coordinates in 'path' and the `creat_line` method to connect consecutive points in 'path' with red straight lines. Each straight line is drawn from the center of the square containing the starting point to the center of the square containing the ending point. This creates a continuous straight line from the starting point to the ending point.



4.5 Map for level 1



4.6 Map for level 2



4.7 Map for level 3

5. Functions that link the GUI to algorithms and support functions

❖ Level1_bfs(n, m, map_matrix, pathList):

The `level1_bfs` function passes the arguments `n`, `m`, `map_matrix`, `pathList`. Where `n`, `m` are the number of rows and columns of the matrix, `map_matrix` is a numpy array of matrix values and `pathList` is a list of paths. Use the `creat_root('BFS')` function to create a Tkinter window with the title "BFS". Initialize `app = PathFinderApp(root, map_matrix, n, m)`, where `PathFinderApp` is the class responsible for the map and path. Check if `pathList[0]` (corresponding to the path of the BFS algorithm) is empty, if empty, call the `showNoPathFound()` function to display a message that the path was not found on the UI, otherwise, call the `draw_path` function to draw the path on the map.

❖ **Level1_dfs, level1_ucs, level1_astar, level1_gbfs functions** are implemented similarly to `level1_bfs` function.

❖ **Level2():**

The function uses the command `file_path = filedialog.askopenfilename(filetypes = [("Text files", "*.txt")])` to open a dialog box that allows the user to select an input file with the .txt format and save the path to the `file_path` variable. Call the function `readInput(file_path, 2)` to read data from the file and return the number of rows, columns, map matrix and time limit. Call the function `pathFinding_level2` to perform the path finding algorithm for level 2 and return a list of path coordinates 'path'. Initialize a Tkinter window with the label 'level 2' and `PathFinderApp(root, map_matrix, n, m)`. Check if the path is empty or not, if empty, call the function `showNoPathFound()` to display a dialog box notifying the path could not be found on the UI, otherwise, save the path to a file using the function `writeOutput` and draw the path on the map.

❖ **Level3():**

`level3()` implements the same as `level2()` but instead of calling the `pathFinding_level2` function, it calls the `pathFinding_level3` function to find the path for level 3.

❖ **ShowNoPathFound():**

The function uses '`messagebox.showinfo`' from the Tkinter library to display a message box with the title "No Path Found" and the text "No Path Found!" to notify the user when the path cannot be found.

6. Components

a. Node.py

Declare class `Node`, this class stores information about a traversable position on the map.

Attributes:

- position (tuple): the location of a node in map, it is also the index of this element in matrix
- parent (tuple): the location of its parent node
- g_cost (int): the path cost from the start node to this node
- h_cost (int): the heuristic of this node is sum of Manhattan distance from this node to goal node
- f_cost (int): $g_cost + h_cost$
- delivery_time (int): the delivery time it takes to reach this node
- remain_fuel (int): the amount of fuel remaining in the vehicle at this node

Methods:

- def __init__(self, position, parent, g_cost, h_cost=0, delivery_time=0, remain_fuel=0)

This is the function that initializes a node with the attributes mentioned above. By default, some attributes are initialized to zero because these attributes will not be used in some algorithms.

- def __lt__(self, other)

This function is used to sort the list of nodes in order of f_cost

- def updateCost(self, g_cost, h_cost=0)

This method updates g_cost and f_cost when this node can be reached from other node with lower cost. By default, h_cost is zero because h_cost will not change

- def getNeighborPos(self, map)

This method returns all locations (tuple) that can be reached from the current node.

b. helper.py

This file contains some sub functions for all project

def readInput(filepath, level)

Input:

- filepath (string): the path of input file
- level (int): the function will return different number of values in different level.

Output:

- In level 1, it will return only row, col, map

- In level 2, it will return row, col, map and time_limit
- In level 3, it will return row, col, map, time_limit and fuel_limit

Description: This function is used to read input file and return some values based on the level

def createOutputFilepath(inputFilepath)

Input:

- inputFilepath (string): the path of input file

Output:

- outputFilepath (string): the path of output file

Description: The functions will find 'input' in the path of input file and replace it with 'output'. For example, inputFilepath is 'input1_level1.txt' so outputFilepath is 'output1_level1.txt'

writeOutput_level1(filepath, pathList)

Input:

- filepath (string): the path of input file
- pathList (list): the list 5 paths corresponding to 5 algorithms in level 1

Description: This function takes input filepath and calls createOutputFilepath(filepath) to get outputFilepath. Then, it opens or creates an output file to write 5 paths to this file. The writeOutput is separated for level 1 because it must write the algorithm name and the path of this algorithm.

def writeOutput(filepath, path)

Input:

- Filepath (string): the path of input file
- path (list): the path is returned from algorithm of level 2 or level 3

Description: This function takes input filepath and calls createOutputFilepath(filepath) to get outputFilepath. Then, it opens or creates an output file to write this path.

def findPosition(map, value)

Input:

- map (numpy.ndarray): a matrix represents all the components of the map

- value (string): the value of the position in map

Output:

- pos (tuple): index of map has the value

Description: The function uses enumerate to find the index of map that has value. It is used for finding position of value 'S' (start position) or 'G' (goal position).

def reconstructPath(exploredSet, startPos, goalPos, isGinE)

Input:

- exploredSet (list): list of all nodes that was traversed
- startPos (tuple): the position (index) of 'S'
- goalPos (tuple): the position (index) of 'G'
- isGinE (boolean): determine whether 'G' in exploredSet

Output:

- path (list): path from position of 'S' to position of 'G'

Description: From exploredSet and map, we can find the path by querying the parent nodes. We start querying parent nodes from the last nodes in exploredSet, isGinE is used to determine whether the goal node should be added to the end of the path. If isGinE is True, the last node in exploredSet is goal node and it is naturally added to path. Otherwise, we have to add goal node to path.

def getWaitTime(map, position)

Input:

- map (numpy.ndarray): a matrix represents all the components of the map
- position (tuple): the index of element that needs to get wait time

Output:

- t (int): the time the vehicle spends at that node

Description: When the vehicle is in Toll Booths or Fuel Station, it spends time. This function is used to get time that it spent.

def getDeliveryTime(map, path)

Input:

- map (numpy.ndarray): a matrix represents all the components of the map
- path (list): path from position of 'S' to position of 'G'

Output:

- t (int): the time it takes the vehicle to travel from 'S' to 'G' including. waiting time at Toll Booths and Fuel Station

Description: The total time to go from S to G is the total number of nodes in the path minus 1 and the waiting time is calculated by getWaitTime() mentioned above. Therefore, the delivery time is the sum of travel time and waiting time.

7. Level modes

a. Level 1

pathFinding_level1(map)

Input:

- map (numpy.ndarray): a matrix represents all the components of the map

Output:

- pathList (list): includes 5 paths corresponding to 5 algorithms (BFS, DFS, UCS, GBFS, A*), if any algorithm cannot find a path to goal, its path is empty

Description: When a user chooses level 1 in GUI, this function is called. The target of this function is to find the path of 5 algorithms so it will call BFS, DFS, UCS, GBFS, A*. In this level, graph search is used. Two important terms related to graph search are Frontier and Explored Set

BFS(map, startPos, goalPos)

Input:

- map (numpy.ndarray): a matrix represents all the components of the map
- startPos (tuple): the position in map where its value is 'S'
- goalPos (tuple): the position in map where its value is 'G'

Output:

- Boolean value: determine whether a path can be found or not
- frontier (list): list of all nodes that will be traversed if the goal was not reached
- exploredSet (list): list of all nodes that was traversed, a path can be created based on exploredSet

Description: BFS algorithm explores all the nodes in a graph at the current depth before moving on to the nodes at the next depth level. It starts at 'S' node and traverses all its neighbors before moving on to the next level of neighbors.

Step-by-step:

- Step 1: Create a Frontier queue and add start node to queue, create empty exploredSet
- Step 2: Pop the first node in Frontier and add it to exploredSet
- Step 3: Find all neighbor nodes of this node, if any neighbor node was not in Frontier and exploredSet, it will be pushed to Frontier
- Step 4: Keep repeating step 2 and step 3 until any neighbor node found is goal node or Frontier is empty

UCS(map, startPos, goalPos)

Input:

- map (numpy.ndarray): a matrix represents all the components of the map
- startPos (tuple): the position in map where its value is 'S'
- goalPos (tuple): the position in map where its value is 'G'

Output:

- Boolean value: determine whether a path can be found or not
- frontier (list): list of all nodes that will be traversed if the goal was not reached
- exploredSet (list): list of all nodes that was traversed, a path can be created based on exploredSet

Description: UCS algorithm explores the nodes in graph based on the lowest path cost in order to find the path from S to G with the minimum path cost. Instead of using queue in BFS, UCS uses a Priority queue for Frontier.

Step-by-step:

- Step 1: Create a Frontier priority queue and add start node to queue, create empty exploredSet
- Step 2: Pop the node that has lowest g_cost in Frontier and add it to exploredSet. In our algorithm, type of Frontier is list in python so we will sort the Frontier in order of increasing g_cost, then the first node in Frontier will be popped
- Step 3: Find all neighbor nodes of this node, if any neighbor node was not in Frontier and exploredSet, it will be pushed to Frontier. If neighbor node was in Frontier, determine whether g_cost of neighbor is less than g_cost of this node in Frontier, if so, update cost, parent node of this node.

- Step 4: Keep repeating step 2 and step 3 until goal node is in exploredSet or Frontier is empty

DFS(map, startPos, goalPos)

Input:

- map (numpy.ndarray): a matrix represents all the components of the map
- startPos (tuple): the position in map where its value is 'S'
- goalPos (tuple): the position in map where its value is 'G'

Output:

- Boolean value: determine whether a path can be found or not
- frontier (list): list of all nodes that will be traversed if the goal was not reached
- exploredSet (list): list of all nodes that was traversed, a path can be created based on exploredSet

Description: DFS algorithm explores the nodes in graph as far as possible along each branch before backtracking. It uses a stack to keep track of the nodes explored so far along a specified branch which help in backtracking in the graph.

Step-by-step:

- Step 1: Create a Frontier stack and push start node on the top of stack, create empty exploredSet
- Step 2: Pop the top node of Frontier and add it to exploredSet. In our algorithm, type of Frontier is list in python so the last node in Frontier will be popped
- Step 3: Find all neighbor nodes of this node and push them to Frontier
- Step 4: Keep repeating until goal node is found or Frontier is empty.

GBFS(map, startPos, goalPos)

Input:

- map (numpy.ndarray): a matrix represents all the components of the map
- startPos (tuple): the position in map where its value is 'S'
- goalPos (tuple): the position in map where its value is 'G'

Output:

- Boolean value: determine whether a path can be found or not
- frontier (list): list of all nodes that will be traversed if the goal was not reached

- exploredSet (list): list of all nodes that was traversed, a path can be created based on exploredSet

Description: GBFS algorithm attempts to find the most promising path cost from S to G. Instead of prioritizing path cost from S to current node in UCS, it prioritizes paths that appear to be the most promising, regardless of whether or not they are actually the shortest path. The algorithm uses a heuristic function to determine which part is the most promising. In our algorithm, heuristic is sum of Manhattan distance from current node to goal node.

Step-by-step:

- Step 1: Create a Frontier priority queue and add start node to queue, create empty exploredSet
- Step 2: Pop the node that has lowest h_cost in Frontier and add it to exploredSet. In our algorithm, type of Frontier is list in python so we will sort the Frontier in order of increasing h_cost, then the first node in Frontier will be popped
- Step 3: Find all neighbor nodes of this node, if any neighbor node was not in Frontier and exploredSet, it will be pushed to Frontier.
- Step 4: Keep repeating step 2 and step 3 until any neighbor node is goal node or Frontier is empty

Astar(map, startPos, goalPos)

Input:

- map (numpy.ndarray): a matrix represents all the components of the map
- startPos (tuple): the position in map where its value is 'S'
- goalPos (tuple): the position in map where its value is 'G'

Output:

- Boolean value: determine whether a path can be found or not
- frontier (list): list of all nodes that will be traversed if the goal was not reached
- exploredSet (list): list of all nodes that was traversed, a path can be created based on exploredSet

Description: A* algorithm is a combination of UCS and GBFS. It not only uses heuristic in GBFS to guide search, but also ensure to compute a path with minimum cost in UCS. The Frontier in this algorithm is prioritized $f_cost = g_cost + h_cost$.

Step-by-step:

- Step 1: Create a Frontier priority queue and add start node to queue, create empty exploredSet

- Step 2: Pop the node that has lowest f_cost in Frontier and add it to exploredSet. In our algorithm, type of Frontier is list in python so we will sort the Frontier in order of increasing f_cost , then the first node in Frontier will be popped
- Step 3: Find all neighbor nodes of this node, if any neighbor node was not in Frontier or exploredSet, it will be pushed to Frontier. If neighbor node was in Frontier, determine whether g_cost of neighbor is less than g_cost of this node in Frontier, if so, update cost g_cost and f_cost , parent node of this node.
- Step 4: Keep repeating step 2 and step 3 until goal node is in exploredSet or Frontier is empty

b. Level 2

def pathFinding_level2(map, time_limit)

Input:

- map (numpy.ndarray): a matrix represents all the components of the map
- time_limit (int): the time that the vehicle must reach goal node within

Output:

- path (list): the path from S to G, it will be empty if no found path
- Description: When a user chooses level 2 in GUI, this function is called. The target of this function is to find the path that has lowest path cost and reach goal node within time_limit. This function will call searchAlgorithm_level2 to find the path

def searchAlgorithm_level2(map, startPos, goalPos, time_limit)

Input:

- map (numpy.ndarray): a matrix represents all the components of the map
- startPos (tuple): the position in map where its value is 'S'
- goalPos (tuple): the position in map where its value is 'G'
- time_limit (int): the time that the vehicle must reach goal node within

Output:

- Boolean value: determine whether a path can be found or not
- frontier (list): list of all nodes that will be traversed if the goal was not reached
- exploredSet (list): list of all nodes that was traversed

Description: This algorithm is based on A* algorithm, but we replace g_cost with $delivery_time$ (the time it takes from S to current node). Therefore, Frontier is prioritized $delivery_time + h_cost$.

Step-by-step:

- Step 1: Create a Frontier priority queue and add start node to queue, create empty exploredSet
- Step 2: Pop the node that has lowest delivery_time + h_cost in Frontier and add it to exploredSet. In our algorithm, type of Frontier is list in python so we will sort the Frontier in order of increasing delivery_time + h_cost, then the first node in Frontier will be popped
- Step 3: Find all neighbor nodes of this node and new_delivery_time of each neighbor, new_delivery_time = delivery_time of current node + 1 + waiting time at current node. If any neighbor node was not in Frontier or exploredSet, it will be pushed to Frontier. If neighbor node was in Frontier, determine whether new_delivery_time of neighbor is less than delivery_time of this node in Frontier, if so, update cost delivery_time and parent node of this node.
- Step 4: Keep repeating step 2 and step 3 until goal node is in exploredSet or Frontier is empty

c. Level 3

def pathFinding_level3(map, time_limit, fuel_limit)

Input:

- map (numpy.ndarray): a matrix represents all the components of the map
- time_limit (int): the time that the vehicle must reach goal node within
- fuel_limit (int): initial capacity of fuel

Output:

- path (list): the path from S to G, it will be empty if no found path

Description: When a user chooses level 3 in GUI, this function is called. The target of this function is to find the path that has lowest path cost and reach goal node within time_limit and capacity of fuel_limit. This function will call searchAlgorithm_level3 to find the path

def searchAlgorithm_level3(map, startPos, goalPos, time_limit, fuel_limit)

Input:

- map (numpy.ndarray): a matrix represents all the components of the map
- start (tuple): the position starts to find 'G'. **Note: At position start in map, its value does not have to be 'S', its value can be '0', 'F1', 'F2',...**
- goalPos (tuple): the position in map where its value is 'G'
- time_limit (int): the time that the vehicle must reach goal node within

- fuel_limit (int): initial capacity of fuel

Output:

- Boolean value: determine whether a path can be found or not
- path (list): the path from S to G within time_limit and capacity of fuel_limit

Description: There are 2 main parts of this algorithm. Firstly, it tries to get the path from **start** to G within time_limit and capacity of fuel_limit, the algorithm will end at this part if a path can be found. Otherwise, the second part will be executed to find **all Fuel Stations** (call findFuelStation() function mentioned below) that can be reached from **current start** within time_limit and fuel_limit, then we get a list of position of Fuel Station f_list. We consider each element of f_list is a **new_start** with **new time_limit** equals to **current time_limit** minus time cost spent to reach this fuel station from current start and new fuel_limit is current fuel_limit (fuel is refilled at Fuel Station). Then, we call searchAlgorithm_level3 (map, new start, goalPos, new time_limit, fuel_limit) function recursively. This function returns a boolean variable that determines whether a path is found or not and the path from **start** to G (if no path found, path is empty: []). The recursive call will end if out of time_limit occurs. The path returned from recursive functions is added to the current path (from start to Fuel Station) and determines whether it is a valid path or not. A valid path is a path that has the last element is the position of 'G'. There can be so many paths, but we just get the lowest time cost path.

Step-by-step:

- Step 1: Use searchAlgorithm_level2() to find the path from start to 'G' within time_limit and fuel_limit.
- Step 2: If a path can be found, return True, path. Otherwise, use findFuelStation() to find all Fuel Stations that can be reached from start within time_limit and fuel_limit.
- Step 3: Consider each fuel station as a new start, new time_limit equals to current time_limit minus time cost spent to reach this fuel station, new fuel_limit is fuel_limit. Call recursively searchAlgorithm_level3() with new start.
- Step 4: The path returned from recursive functions is added to the current path (from start to Fuel Station). We need to filter all valid paths – the paths can reach 'G' and return the path that has the lowest time cost. If there is no valid path, return False and empty path.

8. Test cases and results

Note: The red path is the result found, the remaining paths are added to explain the result.

a. Level 1:

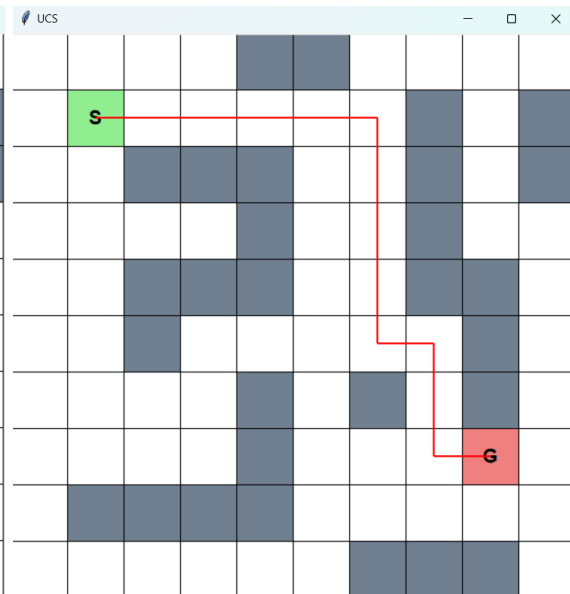
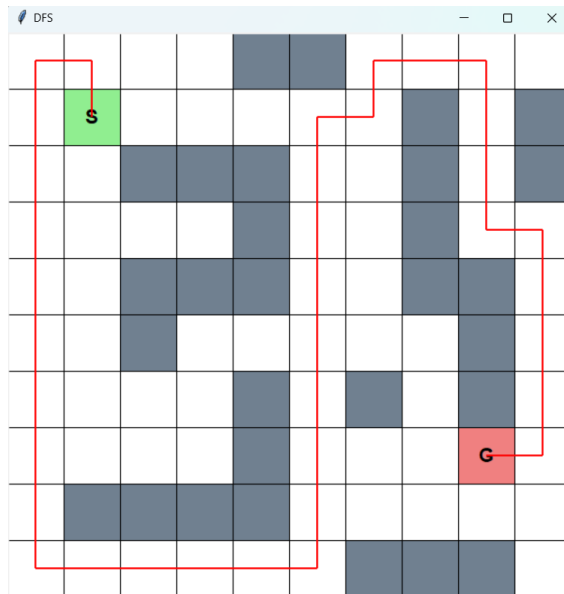
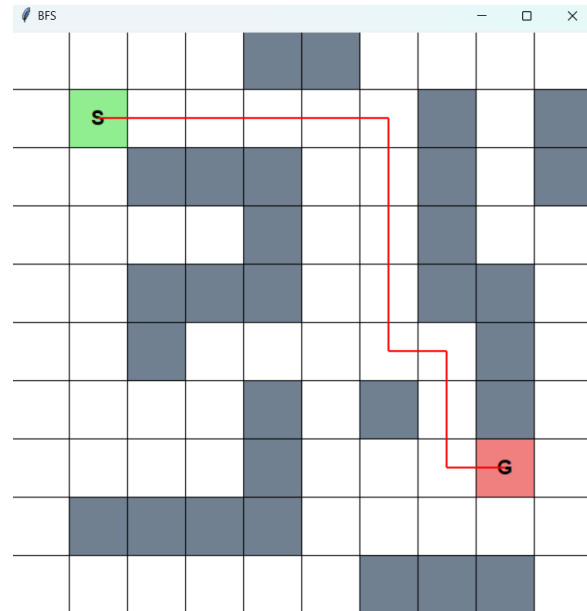
- The input file consists of the first line containing 2 values corresponding to the number of rows and columns. The following lines are the elements of the map matrix. In which, '0' is the cells that can be passed through, '-1' is the obstacles, 'S' is the starting point and 'G' is the destination.

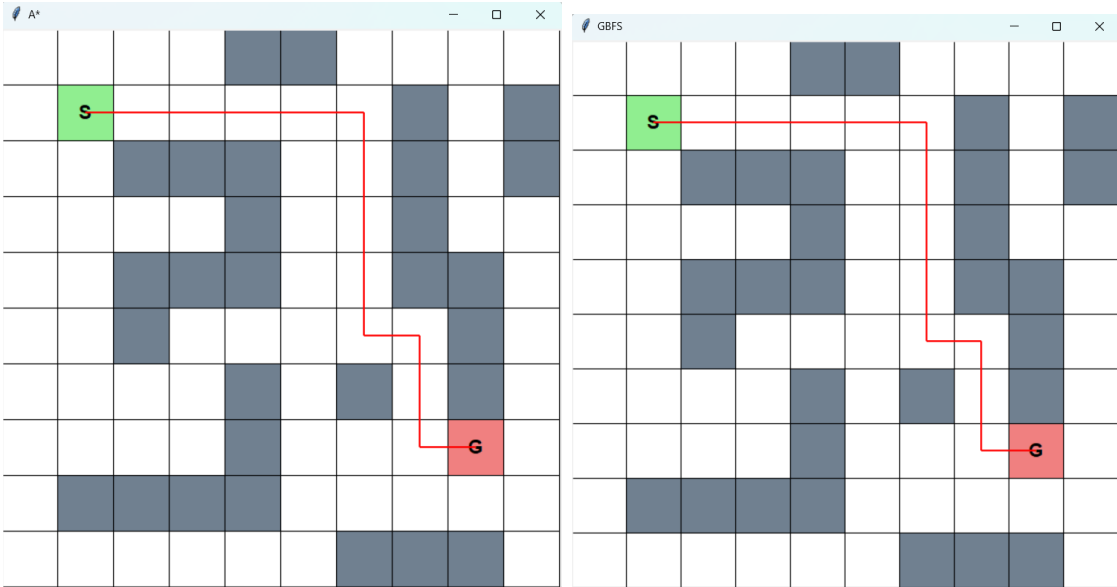
Input1_level1.txt

```

1      10 10
2      0 0 0 0 -1 -1 0 0 0 0
3      0 S 0 0 0 0 0 -1 0 -1
4      0 0 -1 -1 -1 0 0 -1 0 -1
5      0 0 0 0 -1 0 0 -1 0 0
6      0 0 -1 -1 -1 0 0 -1 -1 0
7      0 0 -1 0 0 0 0 0 -1 0
8      0 0 0 0 -1 0 -1 0 -1 0
9      0 0 0 0 -1 0 0 0 G 0
10     0 -1 -1 -1 -1 0 0 0 0 0
11     0 0 0 0 0 0 -1 -1 -1 0

```





In this test case, the BFS, UCS, A*, and GBFS algorithms all find the same path. This happens due to the following reasons:

- BFS searches level by level, meaning it checks the neighboring cells before moving further away. Since all cells have the same cost, BFS will find the shortest path (in terms of the number of cells) from the start point S to the goal G.
- UCS is a version of BFS when all steps have the same cost. UCS also searches for the lowest-cost path, but since the cost of traversing each cell is the same, UCS will find the shortest path just like BFS.
- A* uses an evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost from the starting point to the current cell, and $h(n)$ is a heuristic function that estimates the cost from the current cell to the goal. In this case, $h(n)$ is the Manhattan distance. However, when all the costs across cells are the same and the heuristic is correct, A* behaves like UCS, finding the shortest path.
- GBFS only uses the heuristic function $h(n)$ for routing. It searches for the cells with the lowest $h(n)$ value first. However, with the same cost per cell and $h(n)$ chosen as the Manhattan distance, GBFS is just as likely to find a path as BFS, UCS, and A* in this case.

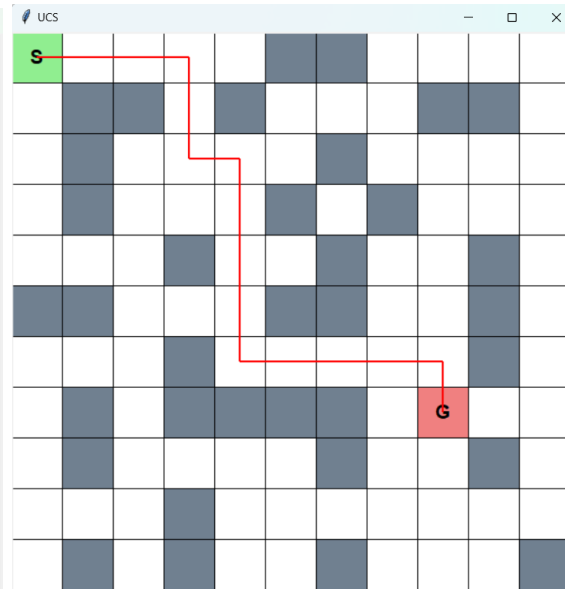
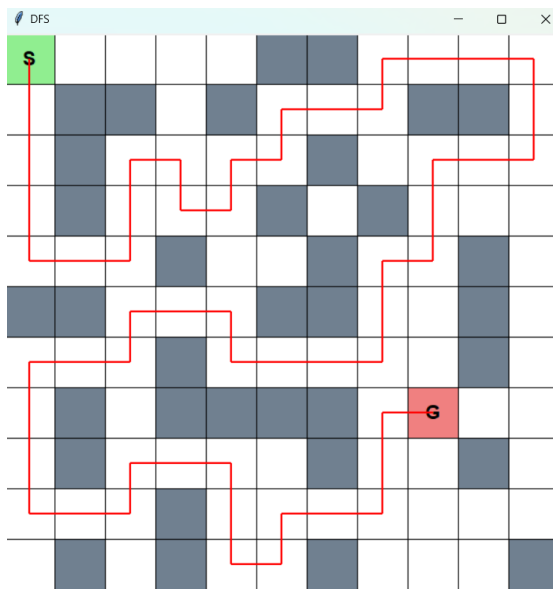
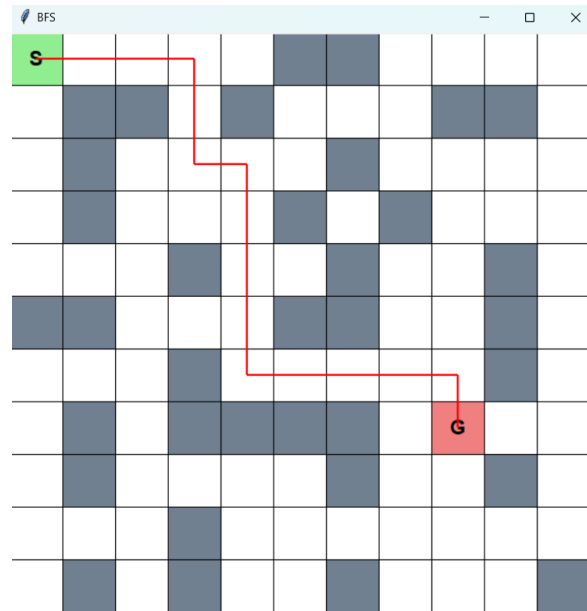
The path of the DFS algorithm is different from the other algorithms because DFS explores depth first, that is, it goes as far as it can along each branch before returning, which can lead to suboptimal paths in many cases. However, in some specific cases, it can still find a path from the starting point to the destination, but it does not guarantee that it is the shortest path.

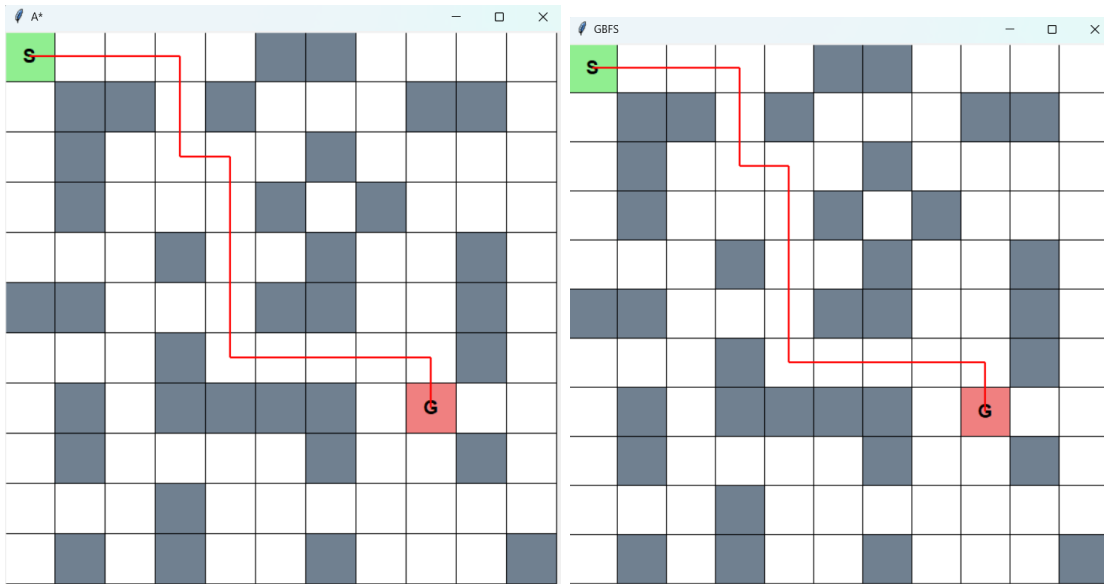
Input2_level1.txt

```

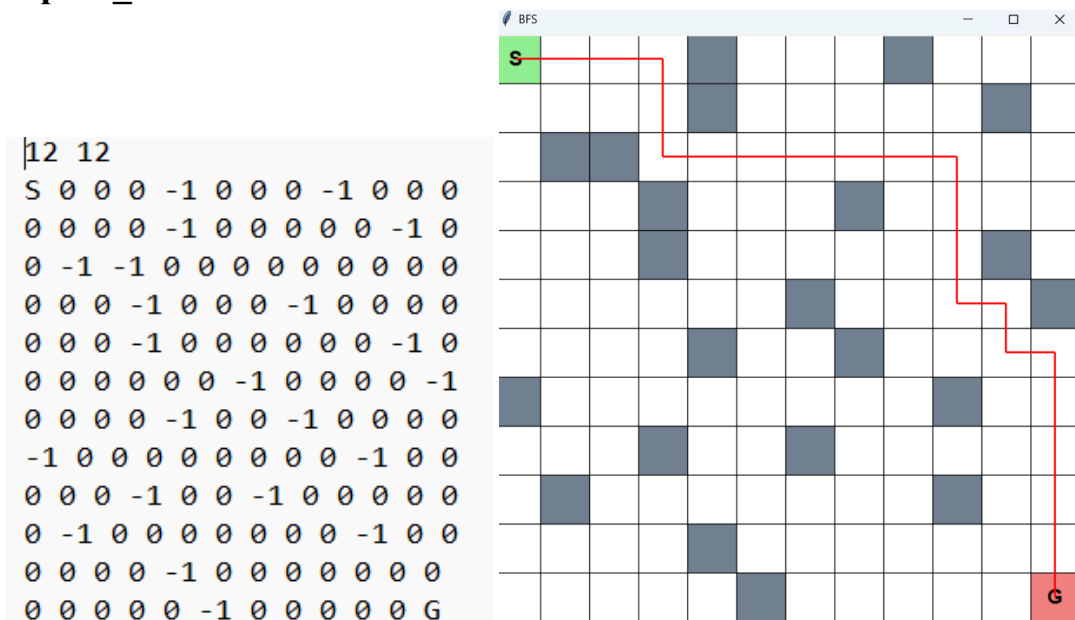
1      11 11
2      S 0 0 0 0 -1 -1 0 0 0 0
3      0 -1 -1 0 -1 0 0 0 -1 -1 0
4      0 -1 0 0 0 0 -1 0 0 0 0
5      0 -1 0 0 0 -1 0 -1 0 0 0
6      0 0 0 -1 0 0 -1 0 0 -1 0
7      -1 -1 0 0 0 -1 -1 0 0 -1 0
8      0 0 0 -1 0 0 0 0 0 -1 0
9      0 -1 0 -1 -1 -1 -1 0 G 0 0
10     0 -1 0 0 0 0 -1 0 0 -1 0
11     0 0 0 -1 0 0 0 0 0 0 0
12     0 -1 0 -1 0 0 -1 0 0 0 -1

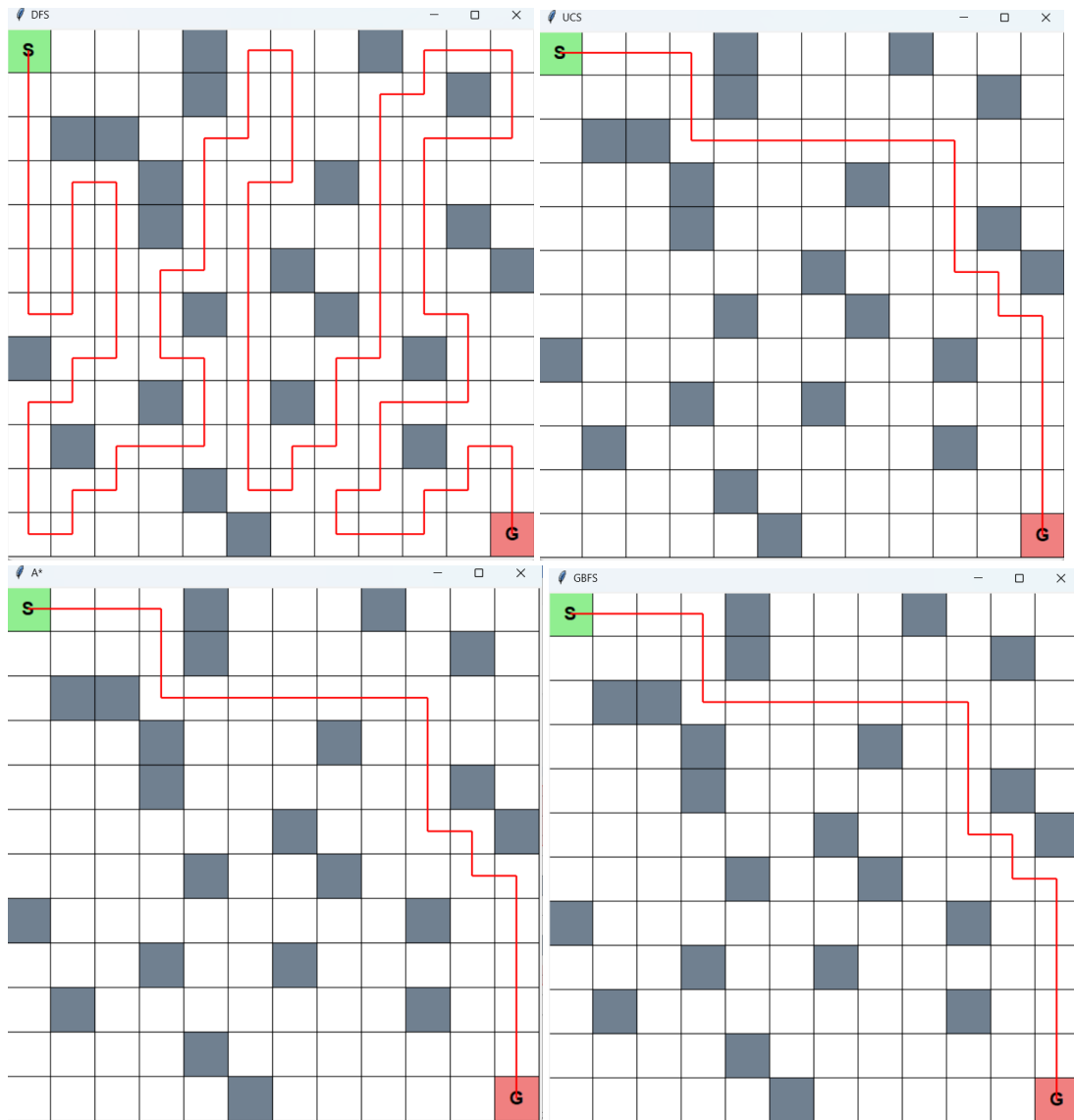
```



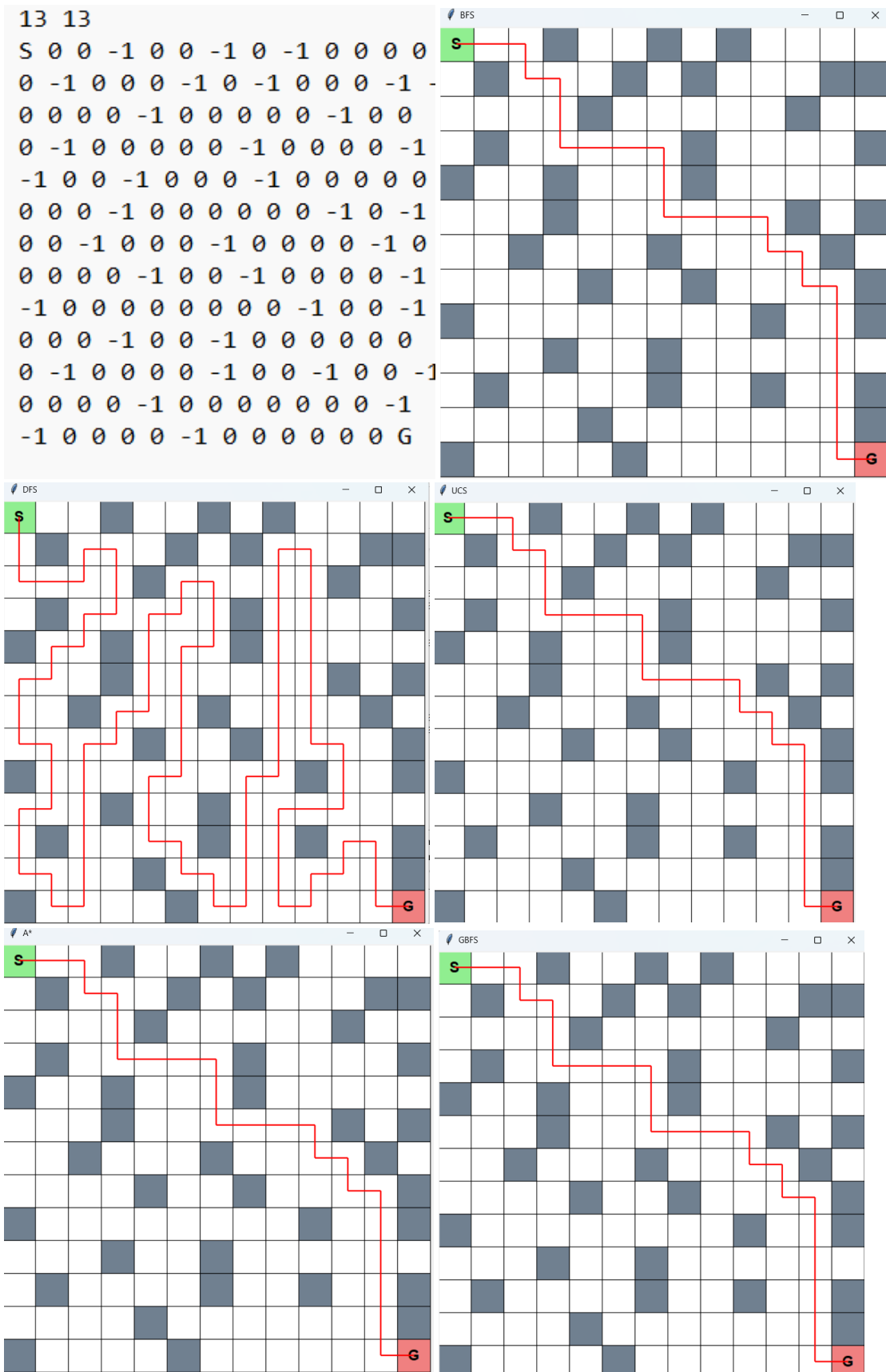


Input3_level1.txt

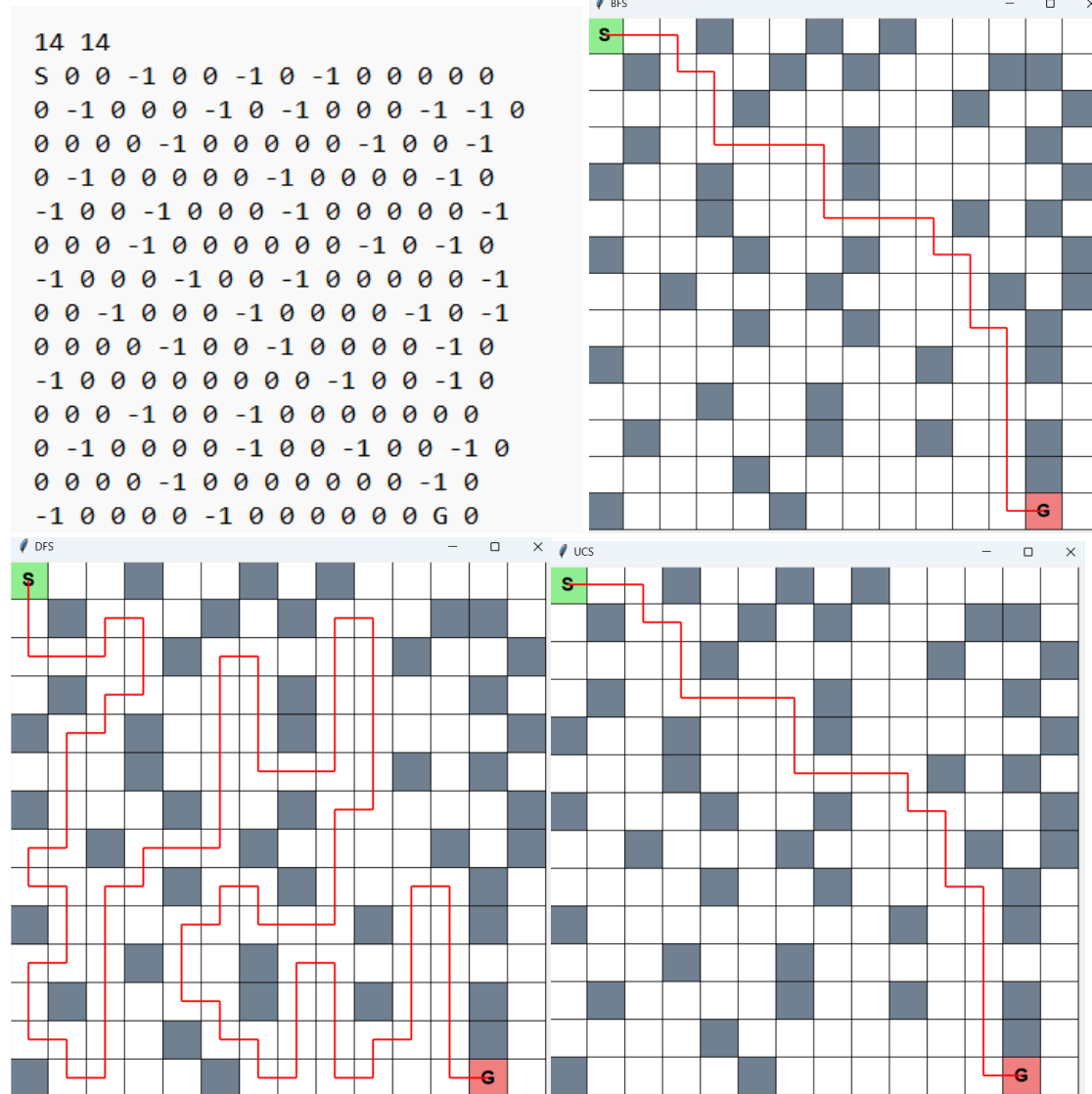


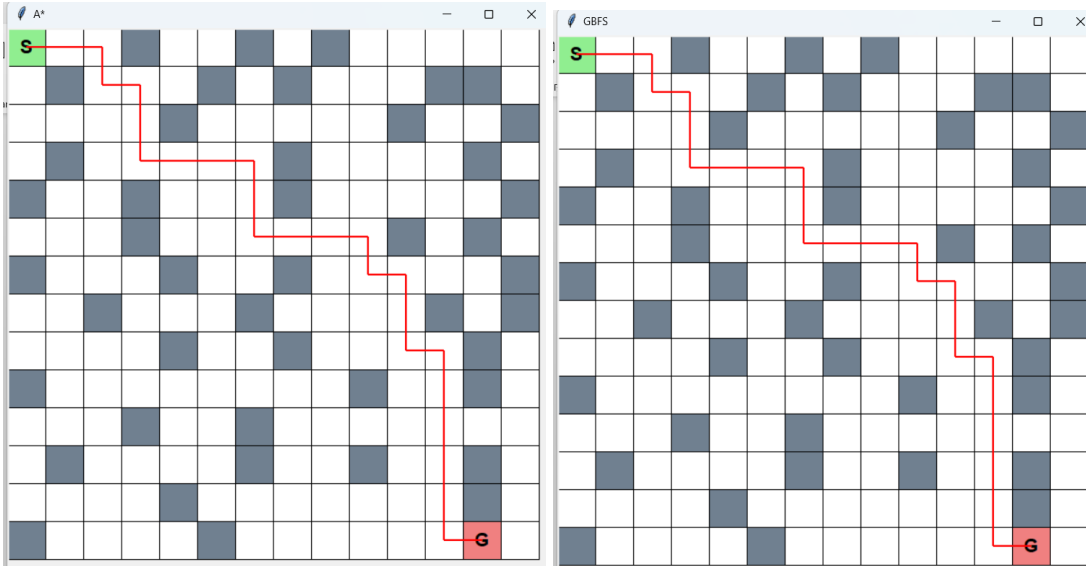


Input4_level1.txt



Input5_level1.txt



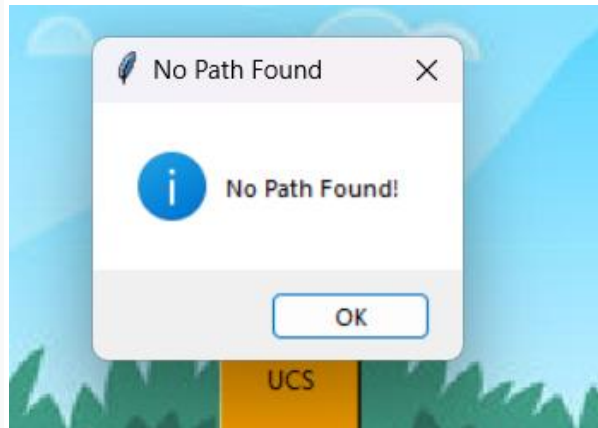


Input6_level1.txt

```

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

```



In this test case because around point G is blocked so there will be no path.

b. Level 2:

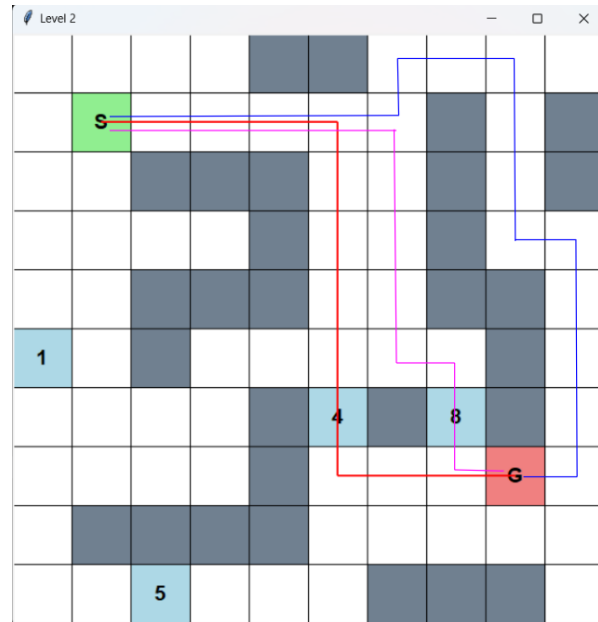
- The input file consists of the first line containing 3 values corresponding to the number of rows, the number of columns and the time limit. The following lines are the elements of the map matrix. In which, '0' are the cells that can be passed, '-1' are the obstacles, cells with values greater than 0 ($t > 0$) are the cells that the delivery vehicle takes t minutes to pass, 'S' is the starting point and 'G' is the destination point.

Input1_level2.txt

```

1      10 10 20
2      0 0 0 0 -1 -1 0 0 0 0
3      0 5 0 0 0 0 0 -1 0 -1
4      0 0 -1 -1 -1 0 0 -1 0 -1
5      0 0 0 0 -1 0 0 -1 0 0
6      0 0 -1 -1 -1 0 0 -1 -1 0
7      1 0 -1 0 0 0 0 0 -1 0
8      0 0 0 0 -1 4 -1 8 -1 0
9      0 0 0 0 -1 0 0 0 G 0
10     0 -1 -1 -1 -1 0 0 0 0 0
11     0 0 5 0 0 0 -1 -1 -1 0

```



In this test case, with $t = 20$, it can be seen that:

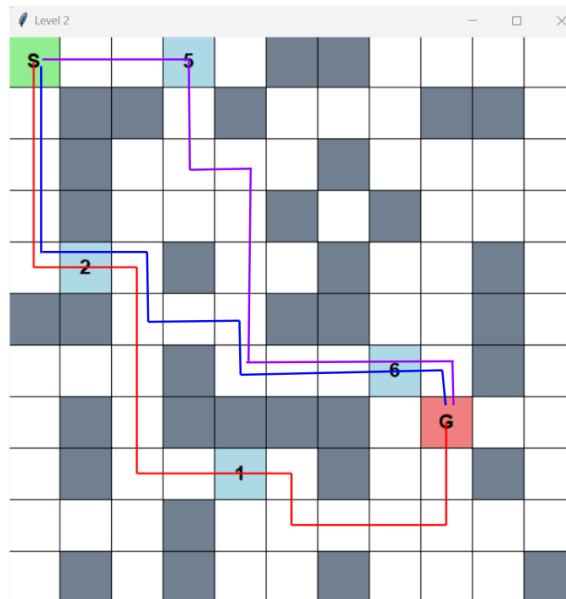
- The purple path is the shortest path (passing 13 cells), but it is not chosen because of overtime committed (21 minutes of delivery).
- The blue path and red path have the same delivery time (17 minutes of delivery), but the red path is chosen because it is shorter (passing through 13 cells).

Input2_level2.txt

```

1      11 11 25
2      5 0 0 5 0 -1 -1 0 0 0 0
3      0 -1 -1 0 -1 0 0 0 -1 -1 0
4      0 -1 0 0 0 0 -1 0 0 0 0
5      0 -1 0 0 0 -1 0 -1 0 0 0
6      0 2 0 -1 0 0 -1 0 0 -1 0
7      -1 -1 0 0 0 -1 -1 0 0 -1 0
8      0 0 0 -1 0 0 0 6 0 -1 0
9      0 -1 0 -1 -1 -1 -1 0 G 0 0
10     0 -1 0 0 1 0 -1 0 0 -1 0
11     0 0 0 -1 0 0 0 0 0 0 0
12     0 -1 0 -1 0 0 -1 0 0 0 -1

```



In this test case, with $t = 25$, it can be seen that:

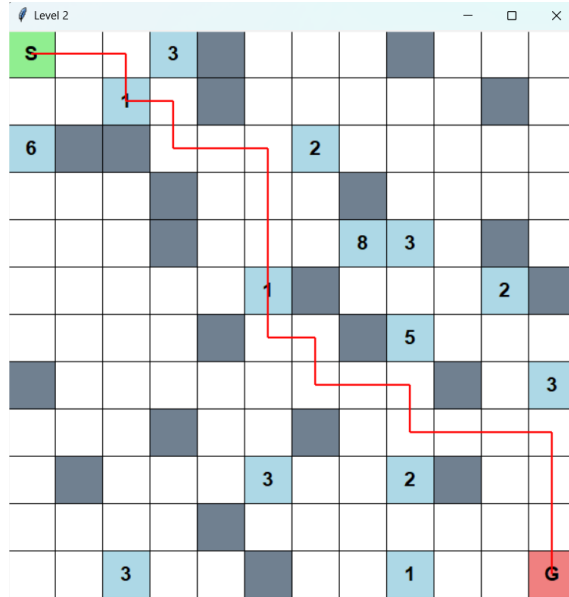
- The purple path and the blue path have the same number of cells to pass through (15 cells) but purple path has a longer delivery time than the blue path.
- The red path passes through more cells than the blue path (19 cells), but the red path is chosen because it has a faster delivery time (22 minutes of delivery).

Input3_level2.txt

```

1      12 12 25
2      5 0 0 3 -1 0 0 0 -1 0 0 0 0
3      0 0 1 0 -1 0 0 0 0 0 -1 0
4      6 -1 -1 0 0 0 2 0 0 0 0 0
5      0 0 0 -1 0 0 0 -1 0 0 0 0
6      0 0 0 -1 0 0 0 8 3 0 -1 0
7      0 0 0 0 0 1 -1 0 0 0 2 -1
8      0 0 0 0 -1 0 0 -1 5 0 0 0
9      -1 0 0 0 0 0 0 0 0 -1 0 3
10     0 0 0 -1 0 0 -1 0 0 0 0 0
11     0 -1 0 0 0 3 0 0 2 -1 0 0
12     0 0 0 0 -1 0 0 0 0 0 0 0
13     0 0 3 0 0 -1 0 0 1 0 0 G

```

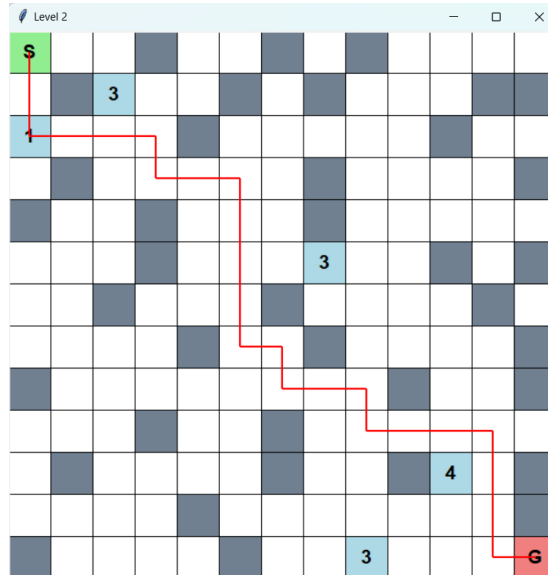


Input4_level2.txt

```

1      13 13 26
2      5 0 0 -1 0 0 -1 0 -1 0 0 0 0
3      0 -1 3 0 0 -1 0 -1 0 0 0 -1 -1
4      1 0 0 0 -1 0 0 0 0 0 -1 0 0
5      0 -1 0 0 0 0 0 -1 0 0 0 0 -1
6      -1 0 0 -1 0 0 0 -1 0 0 0 0 0
7      0 0 0 -1 0 0 0 3 0 0 -1 0 -1
8      0 0 -1 0 0 0 -1 0 0 0 0 -1 0
9      0 0 0 0 -1 0 0 -1 0 0 0 0 -1
10     -1 0 0 0 0 0 0 0 0 -1 0 0 -1
11     0 0 0 -1 0 0 -1 0 0 0 0 0 0
12     0 -1 0 0 0 0 -1 0 0 -1 4 0 -1
13     0 0 0 0 -1 0 0 0 0 0 0 0 -1
14     -1 0 0 0 0 -1 0 0 3 0 0 0 G

```

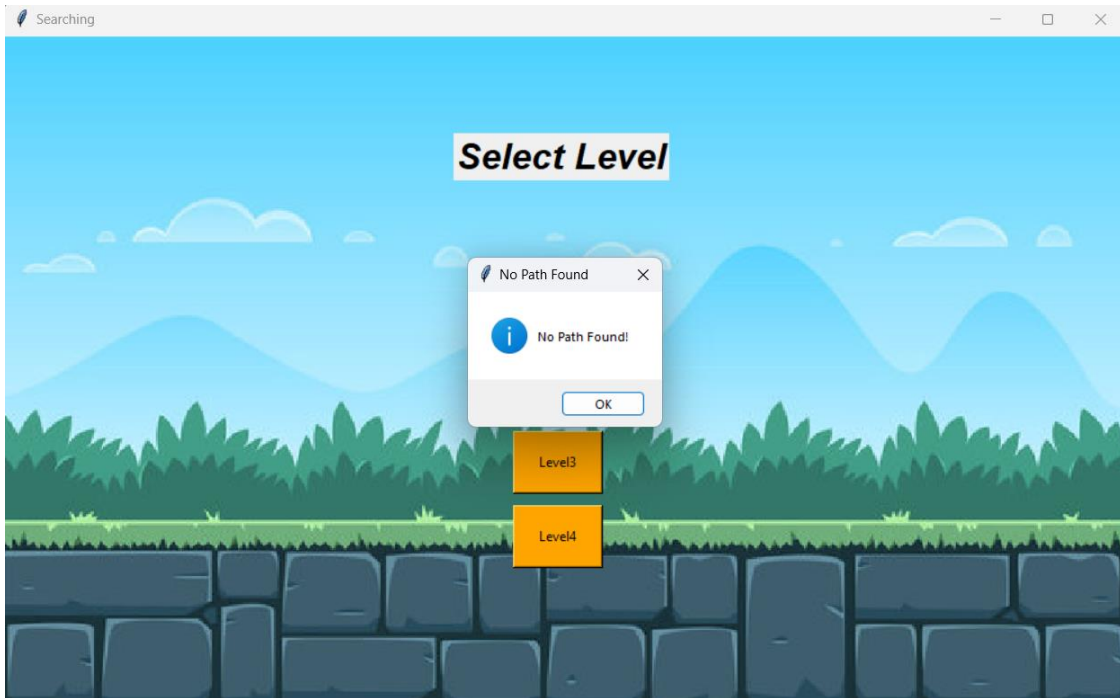
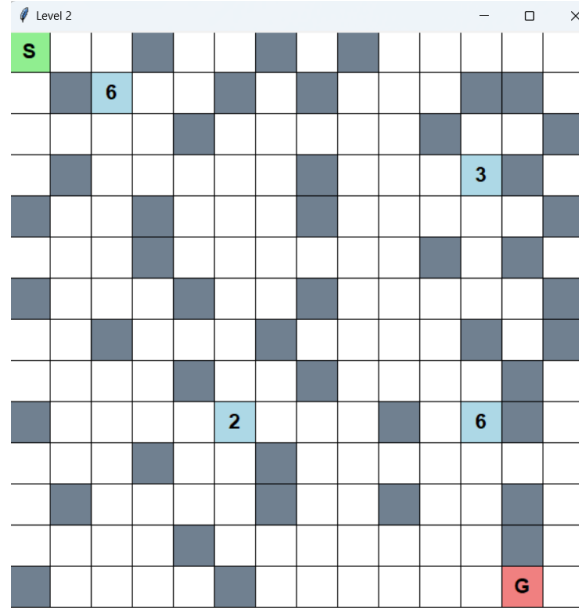


Input5_level2.txt

```

1    14 14 24
2    5 0 0 -1 0 0 -1 0 -1 0 0 0 0 0
3    0 -1 6 0 0 -1 0 -1 0 0 0 -1 -1 0
4    0 0 0 0 -1 0 0 0 0 0 -1 0 0 -1
5    0 -1 0 0 0 0 0 -1 0 0 0 3 -1 0
6    -1 0 0 -1 0 0 0 -1 0 0 0 0 0 -1
7    0 0 0 -1 0 0 0 0 0 0 -1 0 -1 0
8    -1 0 0 0 -1 0 0 -1 0 0 0 0 0 -1
9    0 0 -1 0 0 0 -1 0 0 0 0 -1 0 -1
10   0 0 0 0 -1 0 0 -1 0 0 0 0 -1 0
11   -1 0 0 0 0 2 0 0 0 -1 0 6 -1 0
12   0 0 0 -1 0 0 -1 0 0 0 0 0 0 0
13   0 -1 0 0 0 0 -1 0 0 -1 0 0 -1 0
14   0 0 0 0 -1 0 0 0 0 0 0 0 -1 0
15   -1 0 0 0 0 -1 0 0 0 0 0 0 0 0

```



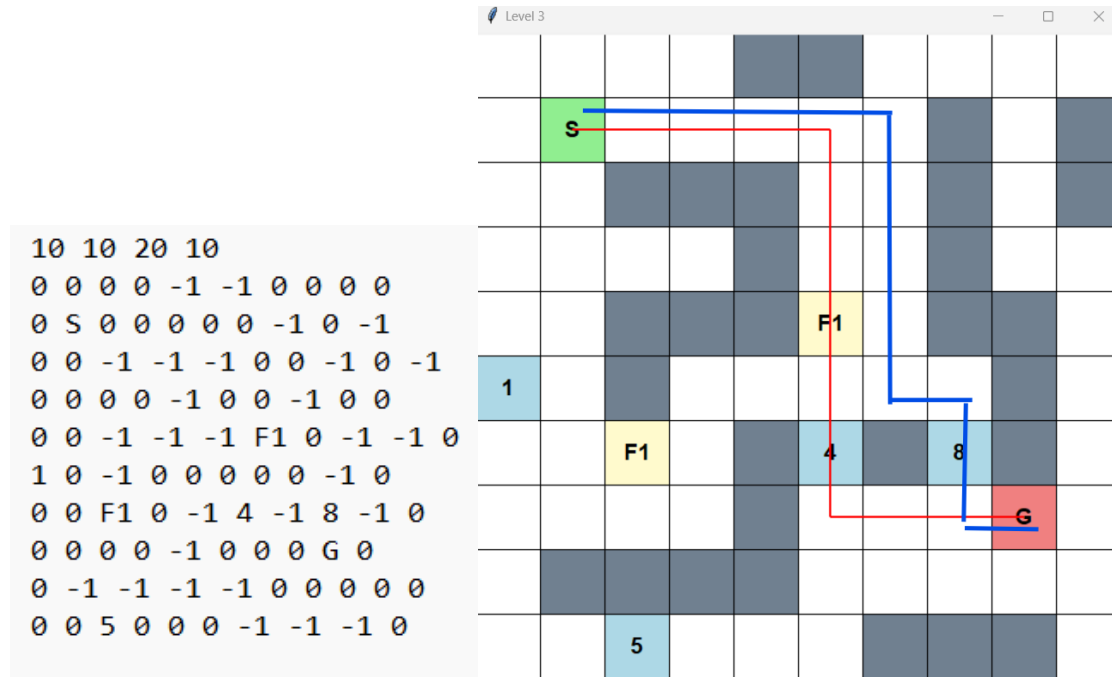
In this test case, the algorithm cannot find a path because the time limit is 24 while the shortest path found has a delivery time of 25.

c. Level 3:

- The input file consists of the first line containing 4 values corresponding to the number of rows, the number of columns, the time limit, and the fuel tank capacity. The following lines are the elements of the map matrix. In which, '0' are the cells that can be passed, '-1' are the obstacles, cells with values greater than 0 ($t > 0$) are the cells that the delivery

vehicle takes t minutes to pass, 'Fx' cells are the gas stations that the delivery vehicle will fill up its tank and take x minutes to pass, 'S' is the starting point, and 'G' is the destination point.

Input1_level3.txt



In this test case, with $t = 20$ and $f = 10$, it can be seen that:

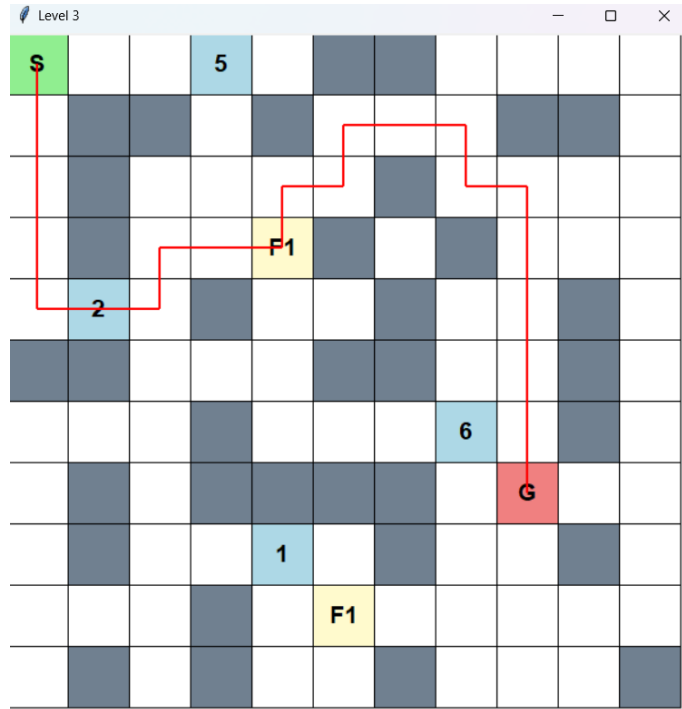
- The blue path and the red path have the same number of cells to pass through (13 cells) but blue path has a longer delivery time than the red path (21 minutes > 18 minutes).
- The red path must go through the fuel station to refill because $f = 10$ while it must pass through 13 cells.

Input2_level3.txt


```

11 11 25 12
S 0 0 5 0 -1 -1 0 0 0 0
0 -1 -1 0 -1 0 0 0 -1 -1 0
0 -1 0 0 0 0 -1 0 0 0 0
0 -1 0 0 F1 -1 0 -1 0 0 0
0 2 0 -1 0 0 -1 0 0 -1 0
-1 -1 0 0 0 -1 -1 0 0 -1 0
0 0 0 -1 0 0 0 6 0 -1 0
0 -1 0 -1 -1 -1 -1 0 G 0 0
0 -1 0 0 1 0 -1 0 0 -1 0
0 0 0 -1 0 F1 0 0 0 0 0
0 -1 0 -1 0 0 -1 0 0 0 -1

```

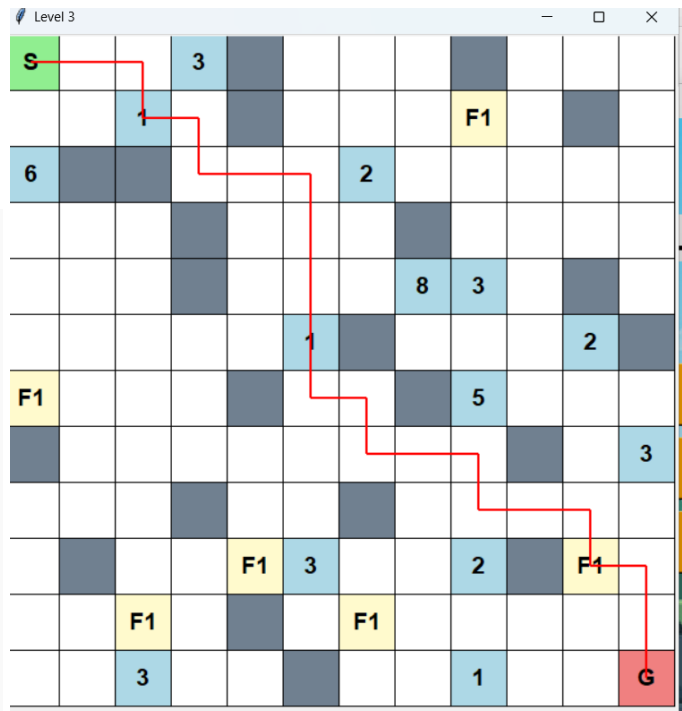


Input3_level3.txt

```

12 12 25 19
S 0 0 3 -1 0 0 0 -1 0 0 0
0 0 1 0 -1 0 0 0 F1 0 -1 0
0 -1 -1 0 0 0 2 0 0 0 0
0 0 0 -1 0 0 0 -1 0 0 0
0 0 0 -1 0 0 0 8 3 0 -1 0
0 0 0 0 0 1 -1 0 0 0 2 -1
F1 0 0 0 -1 0 0 -1 5 0 0 0
-1 0 0 0 0 0 0 0 0 -1 0 3
0 0 0 -1 0 0 -1 0 0 0 0 0
0 -1 0 0 F1 3 0 0 2 -1 F1 0
0 0 F1 0 -1 0 F1 0 0 0 0 0
0 0 3 0 0 -1 0 0 1 0 0 G

```

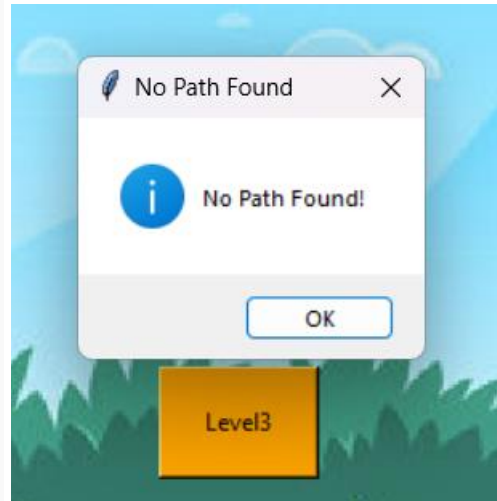


Input4_level3.txt

```

13 13 26 20
S 0 0 -1 0 0 -1 0 -1 0 0 0 0
0 -1 3 0 0 -1 0 -1 0 0 0 -1 -1
1 0 0 0 -1 0 0 F1 0 0 -1 0 0
0 -1 0 0 0 0 0 -1 0 F1 0 0 -1
-1 0 0 -1 0 0 0 -1 0 0 0 0 0
0 0 0 -1 0 0 0 3 0 0 -1 0 -1
0 0 -1 0 0 0 -1 0 0 0 0 -1 0
0 0 0 0 -1 0 0 -1 0 0 F1 0 -1
-1 0 0 F1 0 0 0 0 0 -1 0 0 -1
0 0 0 -1 0 0 -1 0 0 0 0 0 0
0 -1 0 0 0 0 -1 0 0 -1 4 0 -1
0 0 0 0 -1 0 0 0 0 0 0 0 -1
-1 0 0 0 0 -1 0 0 3 0 0 0 G

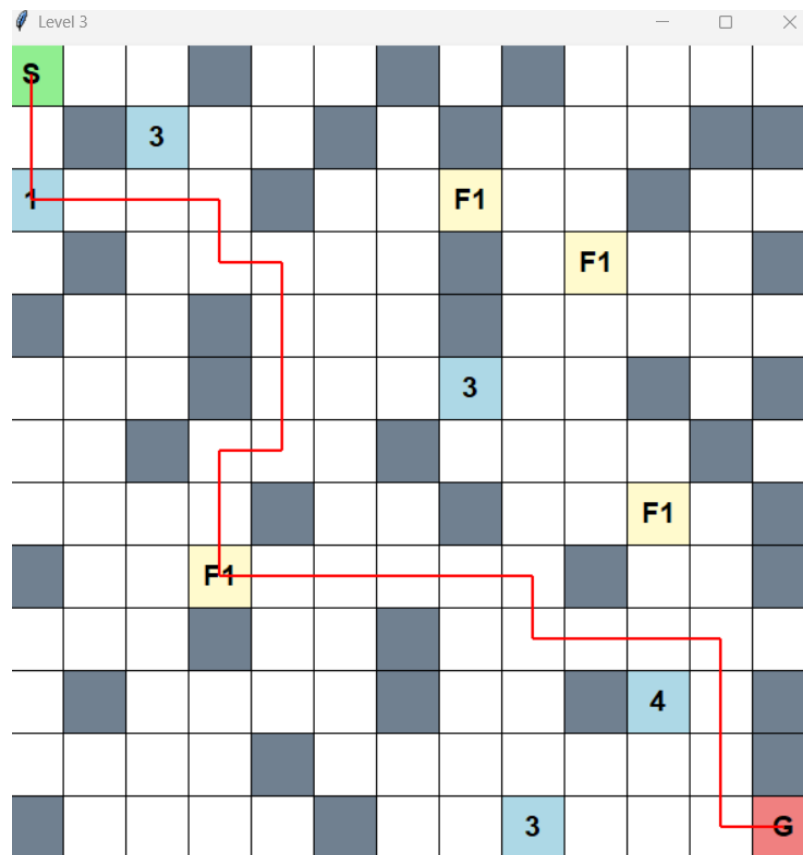
```



In this test case with $t = 26$ and $f = 20$, we can't find the path.

If we change the time $t = 28$, we have just found the path. Because the shortest path cost 28 minutes delivery

The image below illustrates the path when changing the time limit:



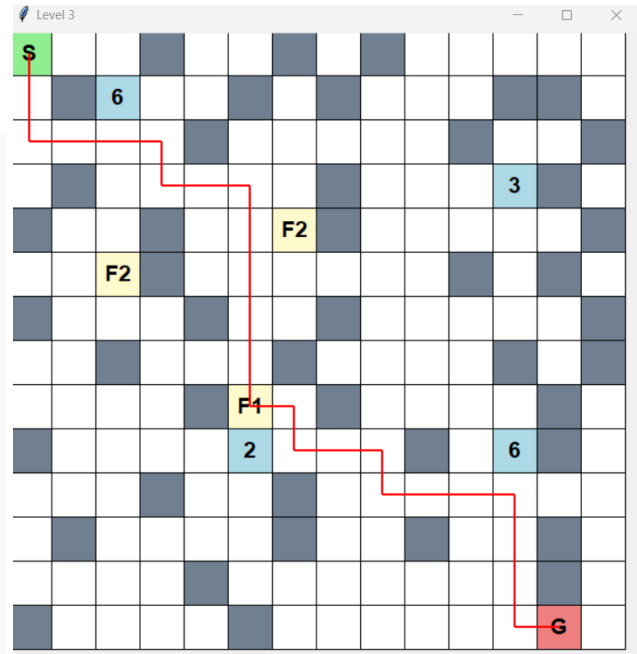
Input5_level3.txt

14 14 28 23

```

S 0 0 0 -1 0 0 0 -1 0 -1 0 0 0 0 0 0
0 -1 6 0 0 0 -1 0 -1 0 0 0 0 -1 -1 0
0 0 0 0 0 -1 0 0 0 0 0 0 -1 0 0 -1
0 -1 0 0 0 0 0 0 -1 0 0 0 3 -1 0
-1 0 0 -1 0 0 F2 -1 0 0 0 0 0 0 -1
0 0 F2 -1 0 0 0 0 0 0 0 -1 0 -1 0
-1 0 0 0 -1 0 0 -1 0 0 0 0 0 0 -1
0 0 -1 0 0 0 -1 0 0 0 0 -1 0 -1
0 0 0 0 -1 F1 0 -1 0 0 0 0 -1 0
-1 0 0 0 0 2 0 0 0 -1 0 6 -1 0
0 0 0 -1 0 0 -1 0 0 0 0 0 0 0 0
0 -1 0 0 0 0 -1 0 0 -1 0 0 -1 0
0 0 0 0 -1 0 0 0 0 0 0 0 -1 0
-1 0 0 0 0 -1 0 0 0 0 0 0 0 G 0

```



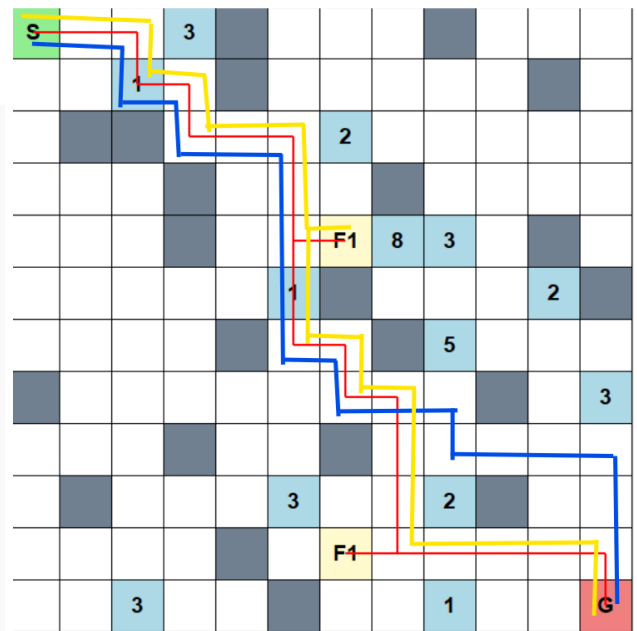
Input6_level3.txt

12 12 30 12

```

S 0 0 3 -1 0 0 0 -1 0 0 0
0 0 1 0 -1 0 0 0 0 0 -1 0
0 -1 -1 0 0 0 2 0 0 0 0 0
0 0 0 -1 0 0 0 -1 0 0 0 0
0 0 0 -1 0 0 F1 8 3 0 -1 0
0 0 0 0 0 1 -1 0 0 0 2 -1
0 0 0 0 -1 0 0 -1 5 0 0 0
-1 0 0 0 0 0 0 0 0 0 -1 0 3
0 0 0 -1 0 0 -1 0 0 0 0 0
0 -1 0 0 0 3 0 0 2 -1 0 0
0 0 0 0 -1 0 F1 0 0 0 0 0
0 0 3 0 0 -1 0 0 1 0 0 G

```



In this test case with $t=30$ and $f=12$, it can be seen that:

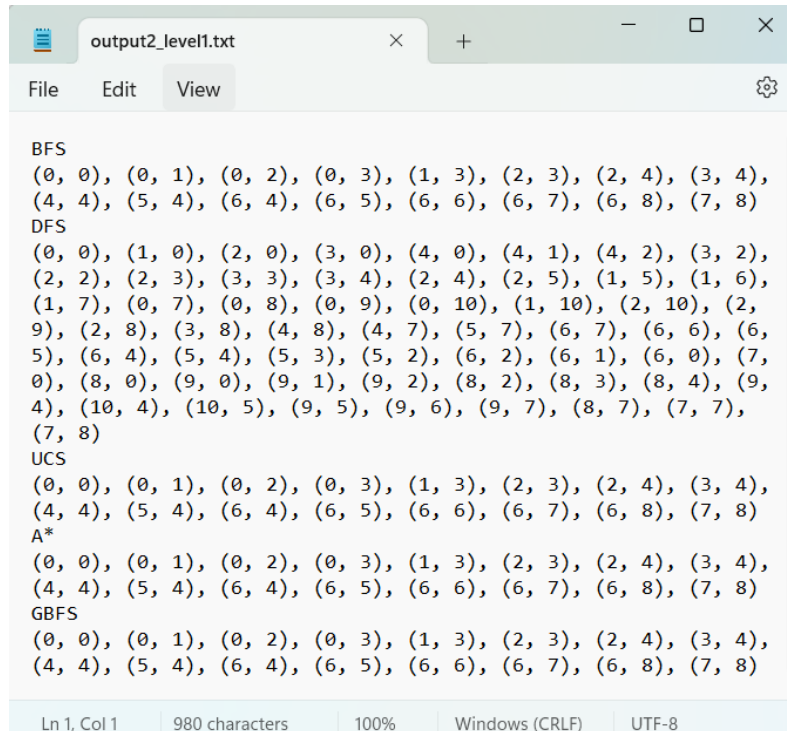
- The blue path is the shortest path without fuel limit that cost 24 minutes.
- The yellow path is the path with $f=15$ and that cost 26 minutes. Because there is a fuel limit, before reaching the destination, the car will run out of gas (need 24 fuel to go to G) so it must go through a fuel station.

- The red path is the path with $f=12$ in input and that cost 28 minutes. Similar to the yellow path, the red path will go to the first fuel station but going from the first fuel station to the destination will cost 15 (similar to the blue path) fuel and it will run out of gas, so we must go to the second fuel station to go to the G.

d. Output file

After running each algorithm, output files will appear

Example: file output2_level1.txt







```

BFS
(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (2, 4), (3, 4),
(4, 4), (5, 4), (6, 4), (6, 5), (6, 6), (6, 7), (6, 8), (7, 8)
DFS
(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (3, 2),
(2, 2), (2, 3), (3, 3), (3, 4), (2, 4), (2, 5), (1, 5), (1, 6),
(1, 7), (0, 7), (0, 8), (0, 9), (0, 10), (1, 10), (2, 10), (2,
9), (2, 8), (3, 8), (4, 8), (4, 7), (5, 7), (6, 7), (6, 6), (6,
5), (6, 4), (5, 4), (5, 3), (5, 2), (6, 2), (6, 1), (6, 0), (7,
0), (8, 0), (9, 0), (9, 1), (9, 2), (8, 2), (8, 3), (8, 4), (9,
4), (10, 4), (10, 5), (9, 5), (9, 6), (9, 7), (8, 7), (7, 7),
(7, 8)
UCS
(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (2, 4), (3, 4),
(4, 4), (5, 4), (6, 4), (6, 5), (6, 6), (6, 7), (6, 8), (7, 8)
A*
(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (2, 4), (3, 4),
(4, 4), (5, 4), (6, 4), (6, 5), (6, 6), (6, 7), (6, 8), (7, 8)
GBFS
(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (2, 4), (3, 4),
(4, 4), (5, 4), (6, 4), (6, 5), (6, 6), (6, 7), (6, 8), (7, 8)

```

The output will be in the same place as the file's input

 output1_level1.txt
 output1_level2.txt
 output1_level3.txt
 output2_level1.txt
 output2_level2.txt
 output2_level3.txt
 output3_level2.txt
 output3_level3.txt
 output4_level2.txt
 output4_level3.txt
 output5_level2.txt
 output5_level3.txt
 output6_level3.txt

9. References

- <https://viettuts.vn/python-tkinter>
- <https://www.codingal.com/coding-for-kids/blog/build-gui-games-in-python/>
- <https://towardsdatascience.com/making-simple-games-in-python-f35f3ae6f31a>
- <https://www.pythontutorial.net/tkinter/tkinter-canvas/>
- <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>
- <https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/>
- <https://codereview.stackexchange.com/questions/247368/depth-first-search-using-stack-in-python>
- <https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/>
- <https://www.geeksforgeeks.org/a-search-algorithm/>

- Material lectures from Mr.Nguyen Tien Huy

10. Demo

Link youtube demo:

https://www.youtube.com/watch?v=EUaIVNYpWec&ab_channel=L%C4%A9nh