Assignment 1: Tree-based search

Introduction to Artificial Intelligence (Hanoi)

Student name: Hoang Minh Le

Student id: 104656973

|  |
| --- |

**Table of contents**

1. **Abstract**
2. **Introduction**
3. **Instruction**
4. **Command line**
5. **Search Algorithms**
6. **Uninformed search methods**

* **Depth first search(DFS)**
* **Breadth first search(BFS)**
* **Custom search 1**

1. **Informed search methods**

* **A\* search**
* **Greedy best first search(GBFS)**

1. **Implementation**
2. **Features/Bugs/Missing**
3. **Features**
4. **Bugs**
5. **Missing**
6. **Research**
   1. **GUI implementation**
7. **Conclusion**
8. **Acknowledge**
9. **References**
10. **Abstract**

This project focuses on creating an AI agent that helps the robot to navigate inside a map using various search algorithms. Depth First Search(DFS) and Breadth First Search(BFS) are two searching methods from uniformed methods that are being used in this project. A\* Search(AS) and Greedy Best First Search(GBFS) are from informed methods. Also there will be two custom methods, Custom Search 1(CUS1) and Custom Search 2(CUS2)

For this project, Python was the chosen coding language due to its adaptability which offers libraries aiding in data manipulation algorithm execution and result representation Pythons clarity and expressiveness also contribute to its suitability, for both development and streamlined progress

1. **Introduction**

In uninformed searching methods, the AI will explore the map without having any information about the goal location and the best path to reach it. Depth First Search goes as far as possible until it reaches the end of the branch before backtracking. While Breadth FIrst Search considers all neighboring nodes at the current depth level before moving on to the nodes at the next level down.

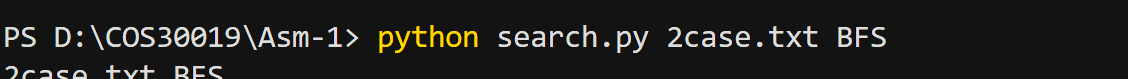
Informed search methods use heuristic functions to calculate the cost of reaching the goal from the starting point. A\* Search looking for nodes by calculating the actual cost to reach them from the starting point to the estimated cost of reaching the goal. While Greedy Best First Search(GBFS) focuses on expanding nodes that are closest to the goal based on the heuristic value of the nodes.

Moreover, in this project, there will be a GUI display interface that will allow users to visualize how the robot agent navigates through obstacles to reach the goal node on the map

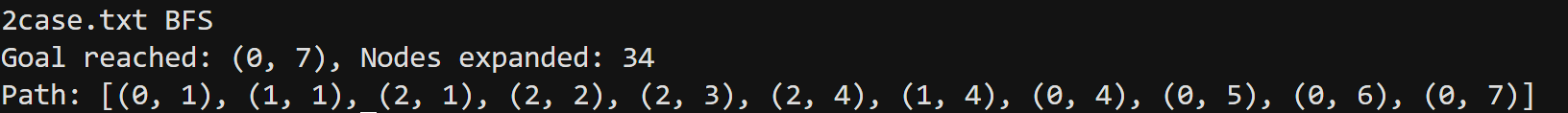
1. **Instruction**

There is only one searching code file, search.py. This allows users to test different search algorithms on a given input map file, txt file. To execute the file in command, please follow this format:

python search.py <txt file> method(in capital)



If the AI agent successfully reach the goal path, the output will be like the following



[filename] will the the input txt map file, in the case which is 2case.txt

[method] is the chosen search method which is next to the filename

[goal] is the goal node of the map

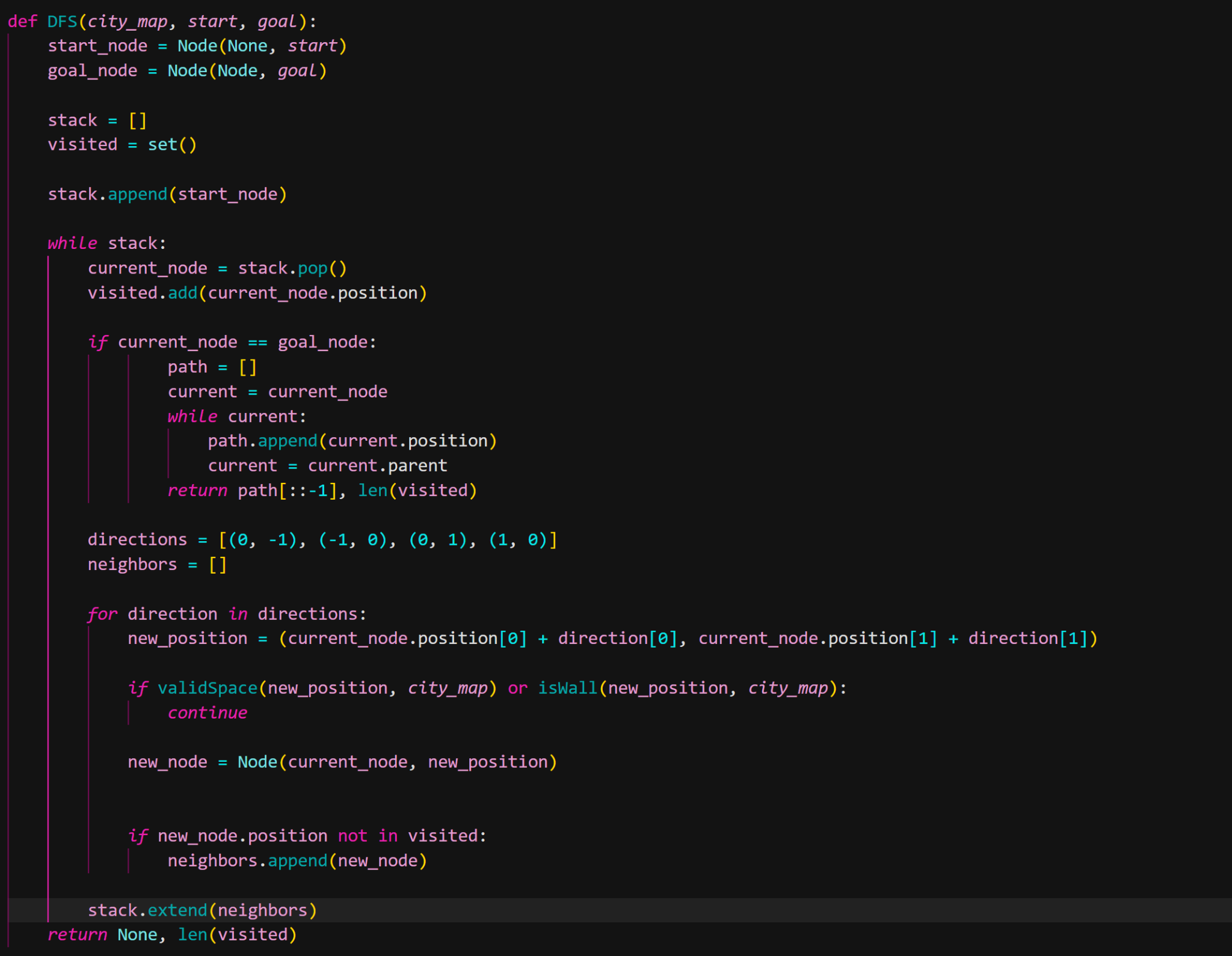
[number\_of\_nodes] is the number of nodes the program has searched through during the searching

[path] is a sequence of moves in the solution that brings the robot from the start to the end configuration.

1. **Search Algorithms**
2. *Uninformed Search Methods:*
   1. Depth First Search(DFS) is a recursive algorithm that explores a graph or tree by starting at a given starting point, it then explores as far as possible until it reaches the end of each branch before backtracking. It stores nodes in a stack and backtracks when it reaches the end.

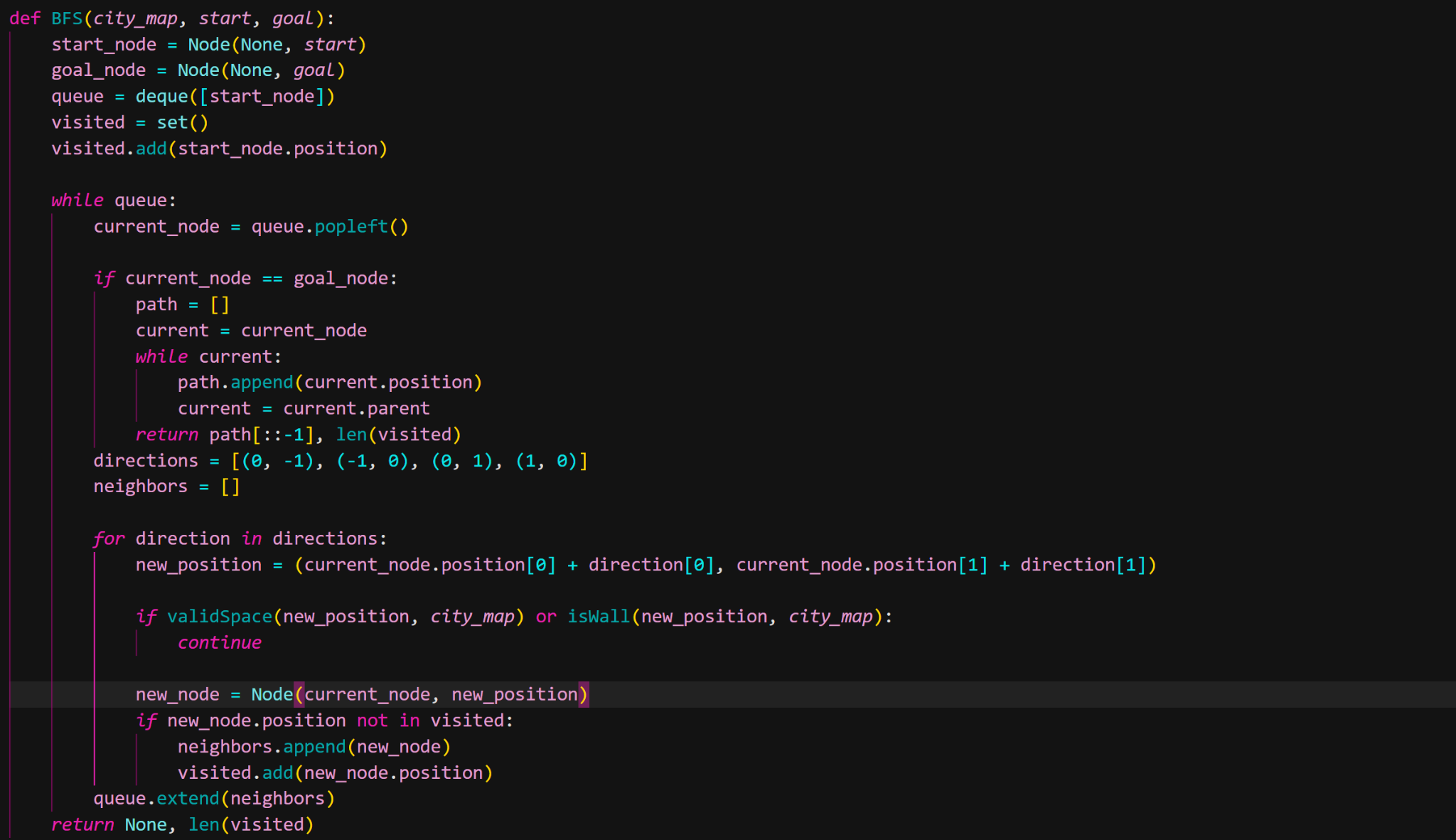
* Function

1. DFS function explores as far as possible until each reaches the end of that branch then backtracks.
2. DFS algorithm stack data structure and LIFO principle
3. DFS uses a stack to track nodes
4. DFS visits a node, then marks it, then searches to find the neighbors of that node, then explores all of its neighbors, going as deep as possible before backtracking. If that node doesn’t have any neighbor, it backtracks to explore other paths
5. The DFS algorithms aim to explore the depth of each brand until it reaches the goal node, it may be leading to a long path



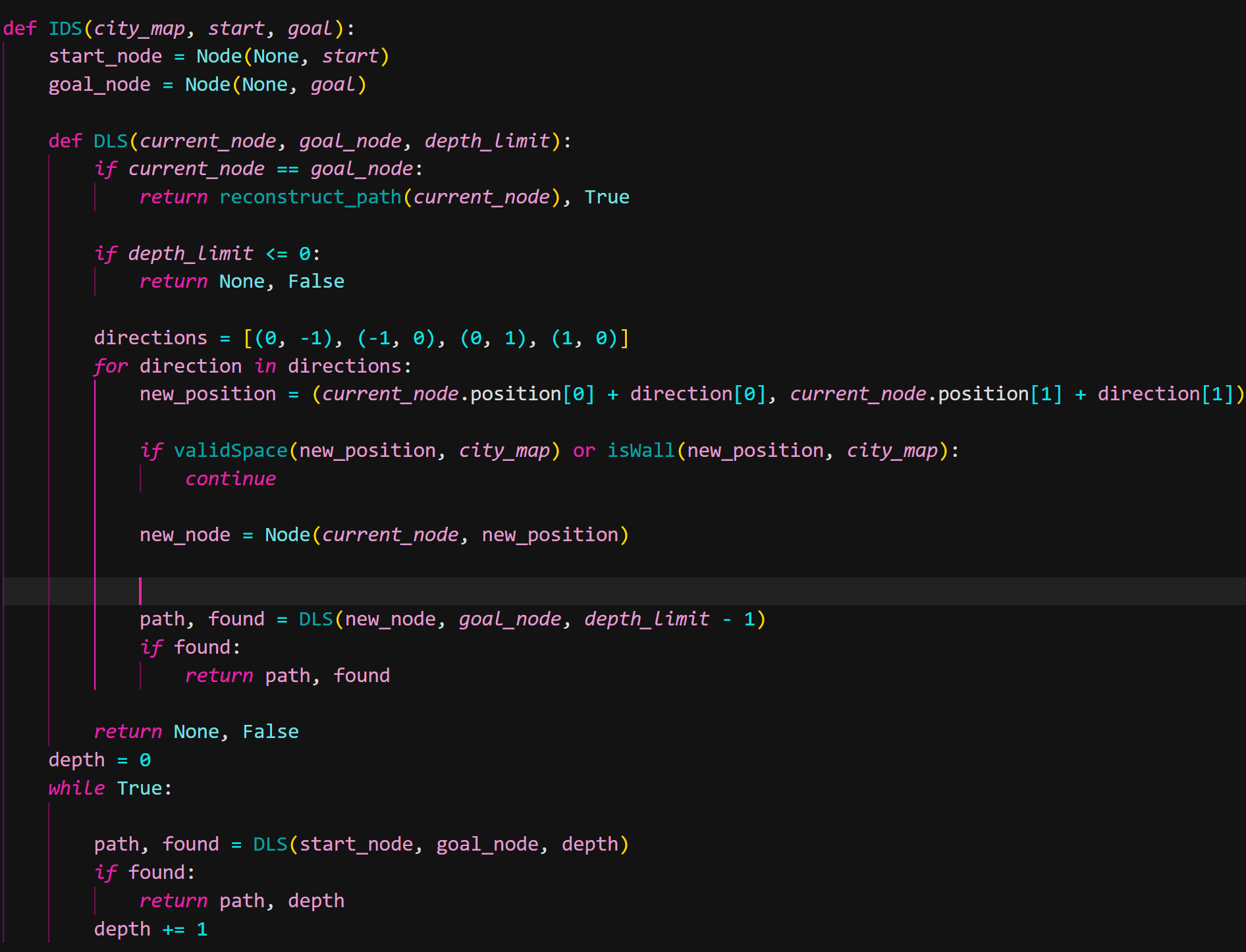
* 1. Breadth First Search(BFS) explores all nodes at the current depth level before moving down to the next level. It stores visited nodes inside a queue. The first node to enter the queue is the first to be processed (FIFO order).
* Function

1. The BFS function explores all neighbor nodes at the current depth level before moving on to nodes at the next depth level
2. BFS requires larger memory than the DFS because it stores all the nodes at the current level in the queue.
3. BFS aims to find the shortest path in the map because it explores all nodes at the current level before going deeper.
4. The algorithms use a deque in order to manage the queue of paths, make sure that all nodes at the current depth level are visited before going to the next level



* 1. Iterative Deepening Search (IDS) combines the search strategy of BFS and DFS. It use the depth searching of DFS with the complete search capability of BFS
* Function:

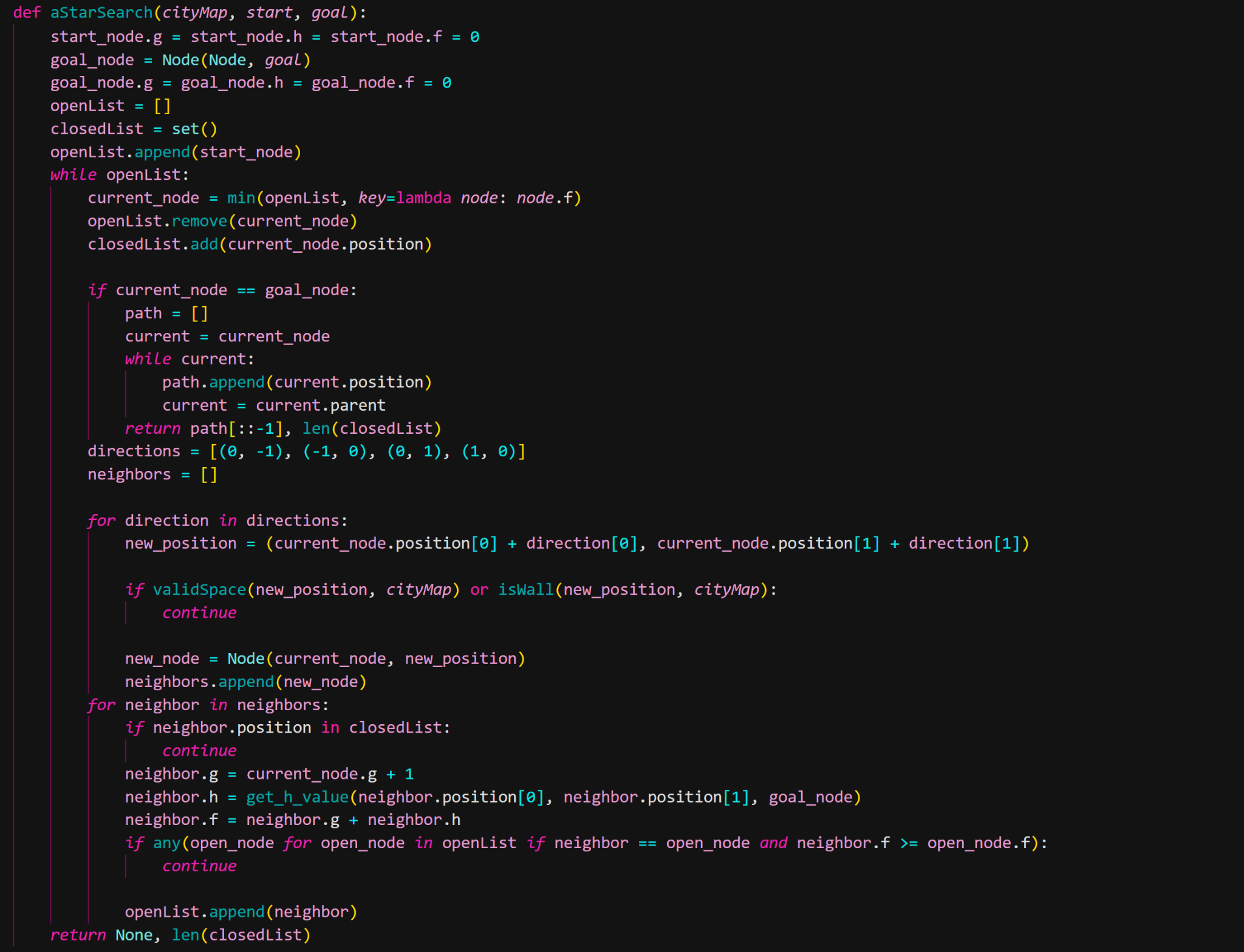
1. Unlike DFS search until reaching the maximum depth level of a branch, IDS performs a series of depth-limited DFS searches. Starting from depth level 0 , which is the root node(start node), and increase the depth limit by 1 in each iteration until the goal state is found
2. In other words, IDS is an upgraded version of DFS, which can guarantee a shorter path compared with DFS search.
3. IDS will always find the goal node if the goal is exists



1. *Informed Search Methods:*
2. A\* Search(AS) expands nodes with the lowest cost, by combining the actual cost to reach them from the start and the estimated cost to reach the goal.

* Function:

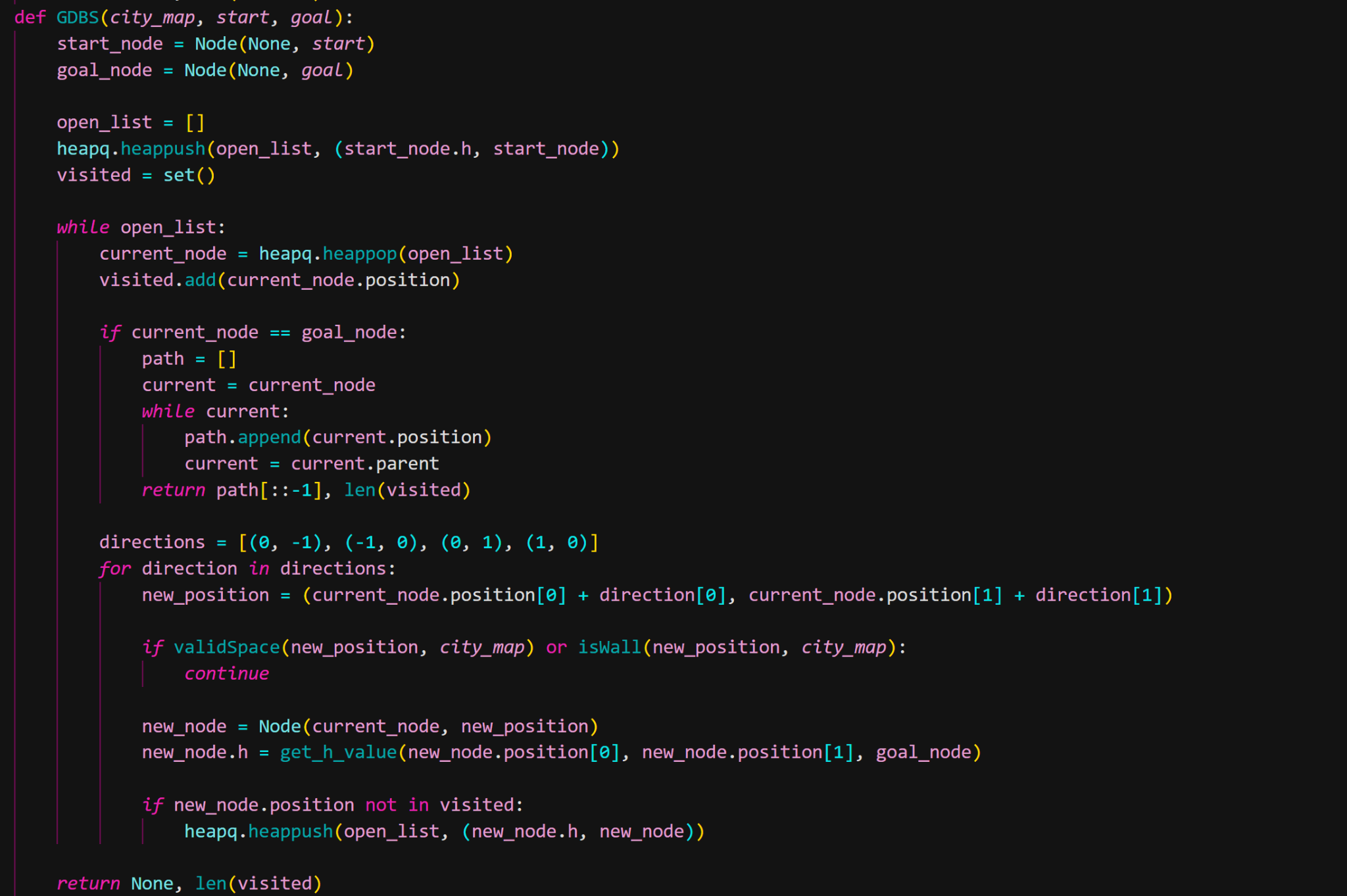
1. A\* evaluates nodes by calculating a value f(n) by adding up two components value
   1. g(n): The actual cost of the path from the start node to the current node n
   2. h(n): The estimated cost (heuristic) to reach the goal from node n
   3. A\* explores node with the lowest f(n)
2. A\* make sure to find the shortest path if the heuristic h(n) is admissible
3. The open list tracks nodes to be explored, and the closed list stores nodes that have already been visited.
4. The A\* search function evaluates nodes by combining the actual cost to reach them from the start and the estimated cost to reach the goal.



1. Greedy Best First Search(GDFS) focuses on expanding nodes that are closest to the goal based on the heuristic value, same like the other methods, it stores nodes into a list.

* Function:

1. GBFS uses a heuristic function to calculate the cost from the current node n to the goal node.
2. The method will start at the open list, Select the node with the lowest h(n) value from the open list
3. That node will be added to the closed list. And then it will explore the neighbor of that node, add the neighbors to the open list
4. Again, find the node with the lowest h(n)
5. Repeat the loop until it reaches the goal node.



1. **Implementation**
   1. *Node class*

Description: The node class represents a single node inside the map. Include the node position, parent node, heuristic value.

class Node:

def \_\_init\_\_(*self*, *parent* =None, *position*=None):

*self*.parent = *parent*;

*self*.position = *position*

*self*.g = 0

*self*.h = 0

*self*.f = 0

def \_\_eq\_\_(*self*, *other*):

*return* *self*.position == *other*.position

def \_\_lt\_\_(*self*, *other*):

*return* *self*.g < *other*.g

Component:

* \_\_init\_\_(self, parent =None, position=None): Initializes a node with its coordinates, parent node and path cost.
* \_\_eq\_\_(self, other): compare nodes in grid-based search algorithms, particularly in algorithms like A\*
* \_\_lt\_\_(self, other): Use to compare the g value of different nodes
  1. *Get\_h\_value*

def get\_h\_value(*x*, *y*, *goal*):

*return* abs(*x* - *goal*.position[0]) + abs(*y* - *goal*.position[1])

* This function calculates the Manhattan distance heuristic between the current node's position and the goal node’s position
* The Manhattan distance is the sum of the absolute differences in the x and y coordinates

* 1. *validSpace(newNodePos, cityMap)*

def validSpace(*newNodePos*, *cityMap*):

*return* *newNodePos*[0] < 0 or *newNodePos*[0] > (len(*cityMap*) - 1) or *newNodePos*[1] < 0 or *newNodePos*[1] > len(*cityMap*[len(*cityMap*)-1]) - 1

* This function checks if a new position is out of bounds in the map
* It returns True if the newNodePos is invalid, meaning the position is outside of the map
* The grid boundaries are checked by comparing newNodePos[0] (row index) and newNodePos[1] (column index) with the dimensions of the grid.
  1. *isWall(newNodePos, cityMap)*

def isWall(*newNodePos*, *cityMap*):

*return* *cityMap*[*newNodePos*[0]][*newNodePos*[1]] != 0

* This function checks whether a position in the map is a wall or an obstacle.
  1. *reconstruct\_path(current\_node)*

def reconstruct\_path(*current\_node*):

path = []

*while* *current\_node*:

path.append(*current\_node*.position)

*current\_node* = *current\_node*.parent

*return* path[::-1]

* The reconstruct\_path function is use to backtrack the path when the agent reach the goal node, it will traceback to the root
* It starts from the goal node (current\_node) and traces back through each node’s parent until it reaches the start node
  1. *read\_file(filename)*

def read\_file(*filename*):

*with* open(*filename*, 'r') *as* file:

lines = file.readlines()

grid\_size = eval(lines[0].strip())

start = eval(lines[1].strip())

goal = eval(lines[2].split('|')[0].strip()) *# For simplicity, take the first goal*

city\_map = [[0 *for* \_ *in* range(grid\_size[1])] *for* \_ *in* range(grid\_size[0])]

*for* line *in* lines[3:]:

x, y, w, h = eval(line.strip())

*for* i *in* range(h):

*for* j *in* range(w):

city\_map[y + i][x + j] = 1

*return* city\_map, start, goal

* This function reads an input text file that specifies the grid, the start and goal positions, and walls. It read this information to create the map, start position, and goal position.
* Components:
* grid\_size: The first line of the input file contains the grid size
* Eval: use to convert string to Python list
* start: The second line contains the starting coordinates
* goal: The third line contains the goal position
* city\_map: Initializes a grid of size grid\_size filled with 0s
* walls: Remaining lines describe walls in the format (x, y, w, h) where x, y are the coordinates of the nodes, and w, h represent the width and height. These cells are marked as 1 in the map
  1. *save\_results:*

def save\_results(*city\_map*, *start*, *goal*, *path*, *output\_file*='path\_data.txt'):

*with* open(*output\_file*, 'w') *as* f:

f.write(f"{len(*city\_map*)}, {len(*city\_map*[0])}\n")

*for* row *in* *city\_map*:

f.write(' '.join(map(str, row)) + '\n')

f.write(f"{*start*}\n")

f.write(f"{*goal*}\n")

*for* step *in* *path*:

f.write(f"{step}\n")

* This function saves the results of the output from the search.py(path, start, goal, map) to an output file, which will be used by the GUI (gui.py) for drawing out the map.
* It will convert from the input file into the map with white space(0), wall(1), start position and in red and goal in green
* Save the path from the search method into that file and the GUI,py will draw
  1. *Search method*

def main():

*if* len(sys.argv) != 3:

print("Usage: python search.py <filename> <method>")

*return*

filename = sys.argv[1]

method = sys.argv[2]

city\_map, start, goal = read\_file(filename)

*if* method == 'A\*':

path, num\_nodes = aStarSearch(city\_map, start, goal)

*elif* method == 'DFS':

path, num\_nodes = DFS(city\_map, start, goal)

*elif* method == 'BFS':

path, num\_nodes = BFS(city\_map, start, goal)

*elif* method == 'GBFS':

path, num\_nodes = GBFS(city\_map, start, goal)

*elif* method == 'IDS':

path, num\_nodes = IDS(city\_map, start, goal)

*else*:

print("Unknown search method. Use 'A\*', 'DFS', or 'BFS', 'GBFS.")

*return*

*if* path:

print(f"{filename} {method}")

print(f"Goal reached: {goal}, Nodes expanded: {num\_nodes}")

print(f"Path: {path}")

save\_results(city\_map, start, goal, path)

subprocess.run(['python', 'gui.py', 'path\_data.txt'])

*else*:

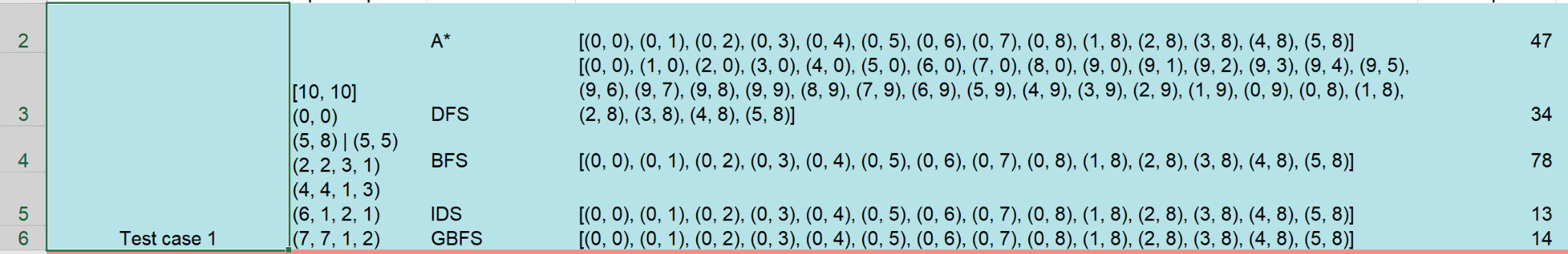
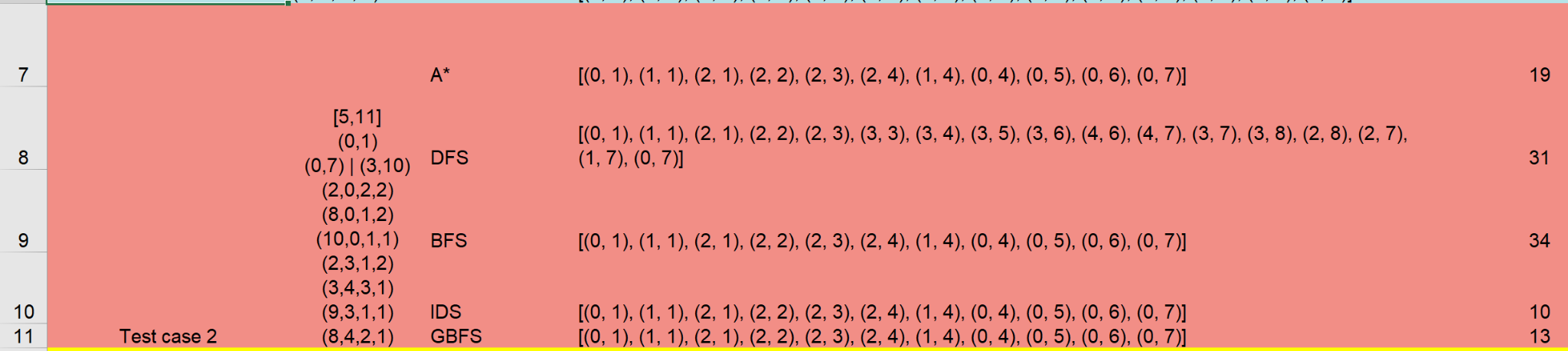
print(f"{filename} {method}")

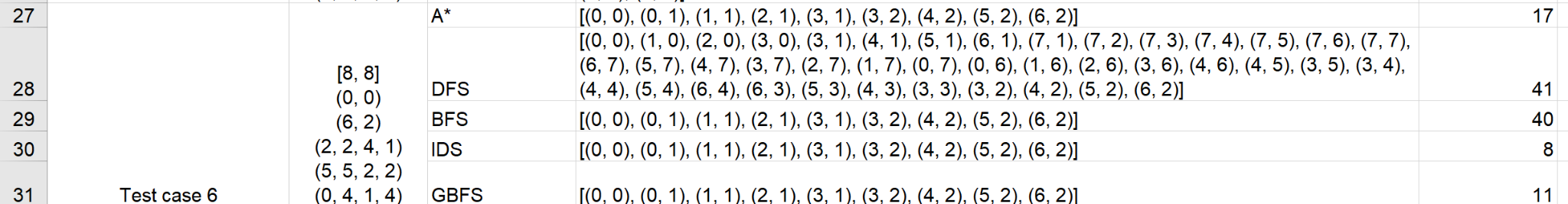
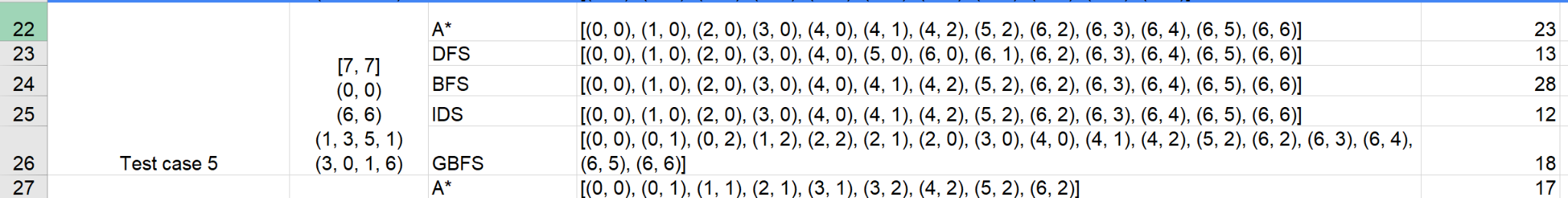
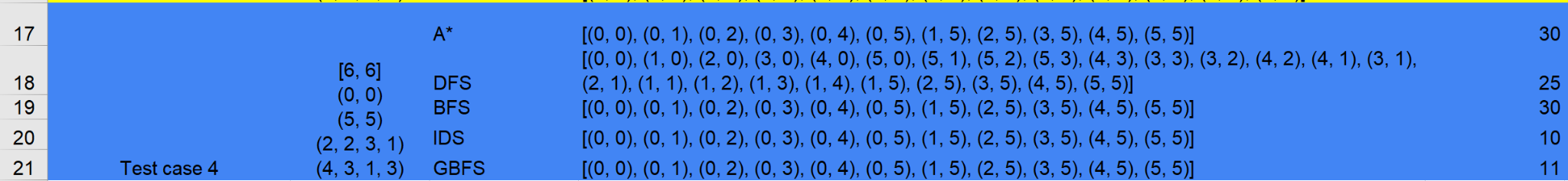
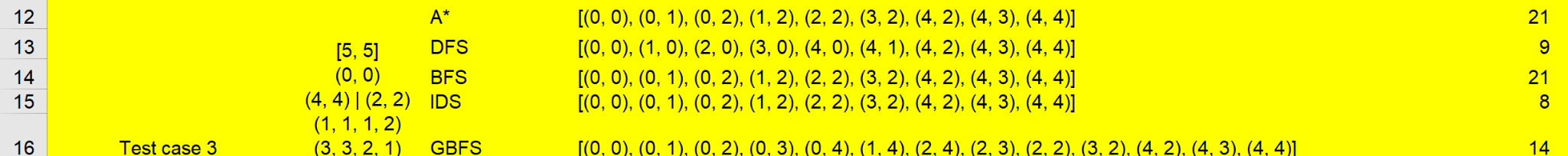
print(f"No goal is reachable; Nodes expanded: {num\_nodes}")

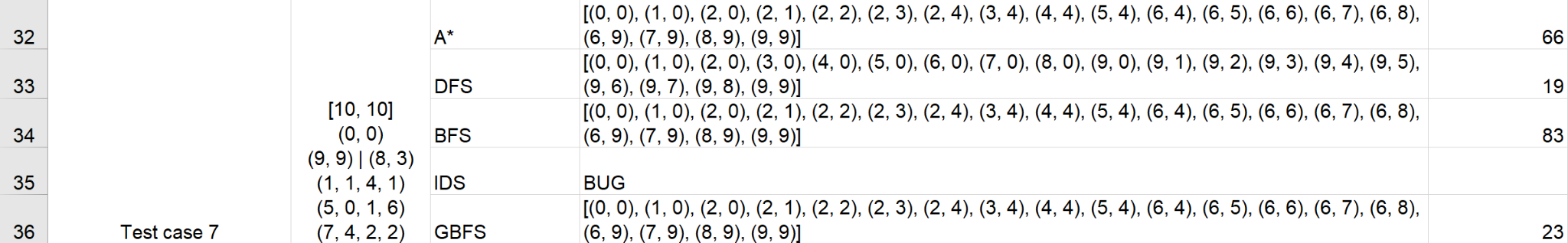
* This function is the main entry point of the program.
* It reads the input file using parse\_input\_file.
* Selects a search algorithm based on the command-line arguments.
* Runs the selected search algorithm.
* If a path is found, it saves the results and calls the GUI to visualize the path.

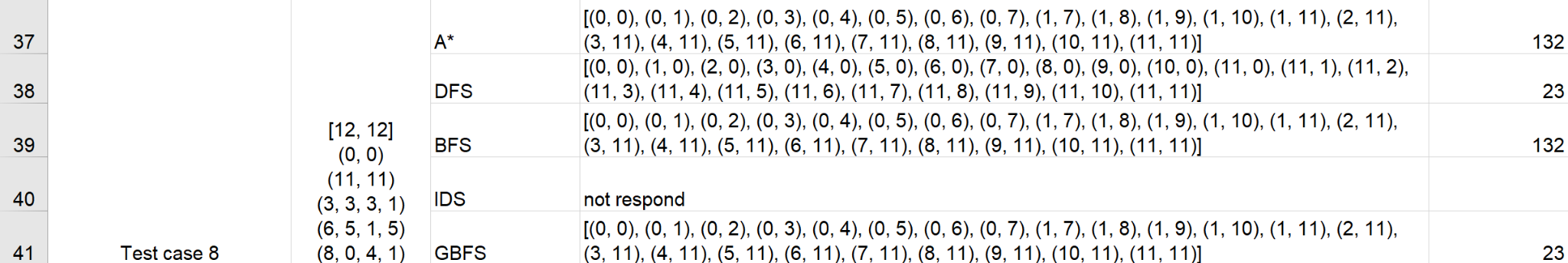
1. **Testing**

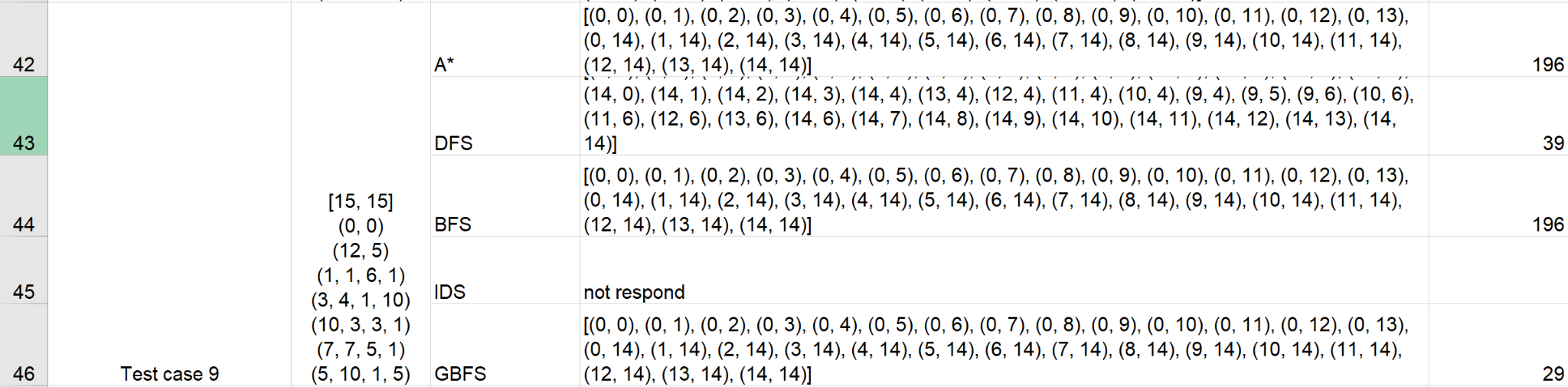
Below are the screenshots of 10 test cases that I tested. There are 5 columns, the first one is the test number, second one is the input map txt file, the third column is the searching methods, the fourth one is the path and the last one is the node expanded.











1. **Analysis**
2. *DFS search*

* Suitable for small map and less obstacles
* Memory efficient
* Not always provide the shortest path to goal

1. *BFS search*

* Attempts to provide the shortest path
* But require large memory space due to its searching method

1. *CUS1(IDS search)*

* Suitable for medium size map
* In a larger grid with more complex obstacles like from test case 7 8 and 9, IDS might take too long to reach deeper nodes because it explores the search space repeatedly at increasing depths
* IDS could waste time and calculations that take place at each depth.

1. *A\* Search*

* Guarantees the shortest path, balances memory usage and path efficiency
* A\* works best in larger map size and complicate map
* For some small size map, A\* required more memory than the other searching method like DFS or IDS
* A\* is depend on the quality of the heuristic function

1. *Greedy Best First Search*

* Efficient memory usage
* Could work well in small , medium and even large map size with different complexity
* Not always provide the shortest path

1. **Features/Bugs/Missing**
2. *Feature*
3. Tree-based search algorithms

* Can implement four different searching methods from uninformed and informed methods. Depth-first search (DFS), Breadth-first search (BFS) and Greedy-best-first search (GBFS), A star (AS).
* Moreover, include one custom search for uninformed methods, IIterative Deepening Search(IDS)

1. Save searching result

* After running the search method with the test case, transvert the input text map file into a grid using 0 for white space and 1 for wall. Furthermore, save the path into that file for the grid.py to use

1. GUI

* Build a GUI display interface that accurately draws the path that the AI agents use to reach the goal node from the root node through obstacles.
  1. *Bugs*

1. IDS method cannot work in large map size

* Start from test case 7, where the map has the grid of 10x10, the method did not respond when running the search.py

1. Goal state

* Can only read one goal node.
  1. *Missing*

1. CUS2

* Only implement the custom search 1 for uninformed method, missing the custom 2 for informed method

1. Goal state

* Currently, the AI can only reach one goal, the goal coordinates that stand before the second one. I have not implemented the code for the AI to read and can reach two goals at once.

1. **Research**
2. *Idea*

* For the research part, I built a GUI display interface which can be found in file gui.py. The GUI displays a window that represents the navigation of a robot agent through obstacles step-by-step to reach the goal node on the map based on different searching methods.

1. *Components:*
2. Import

* The script imports the necessary modules, tkinter for the GUI and sys for command-line arguments.

1. Draw\_map function(city\_map, path, start, goal)

rows = len(*city\_map*)

cols = len(*city\_map*[0])

canvas = tk.Canvas(root, *width*=cols \* 40, *height*=rows \* 40)

canvas.pack()

colors = {

'empty': 'white',

'wall': 'black',

'start': 'red',

'goal': 'green',

'path': 'blue'

}

*for* row *in* range(rows):

*for* col *in* range(cols):

x1, y1 = col \* 40, row \* 40

x2, y2 = x1 + 40, y1 + 40

color = colors['empty']

*if* *city\_map*[row][col] == 1:

color = colors['wall']

*if* (row, col) in *path*:

color = colors['path']

*if* (row, col) == *start*:

color = colors['start']

*if* (row, col) == *goal*:

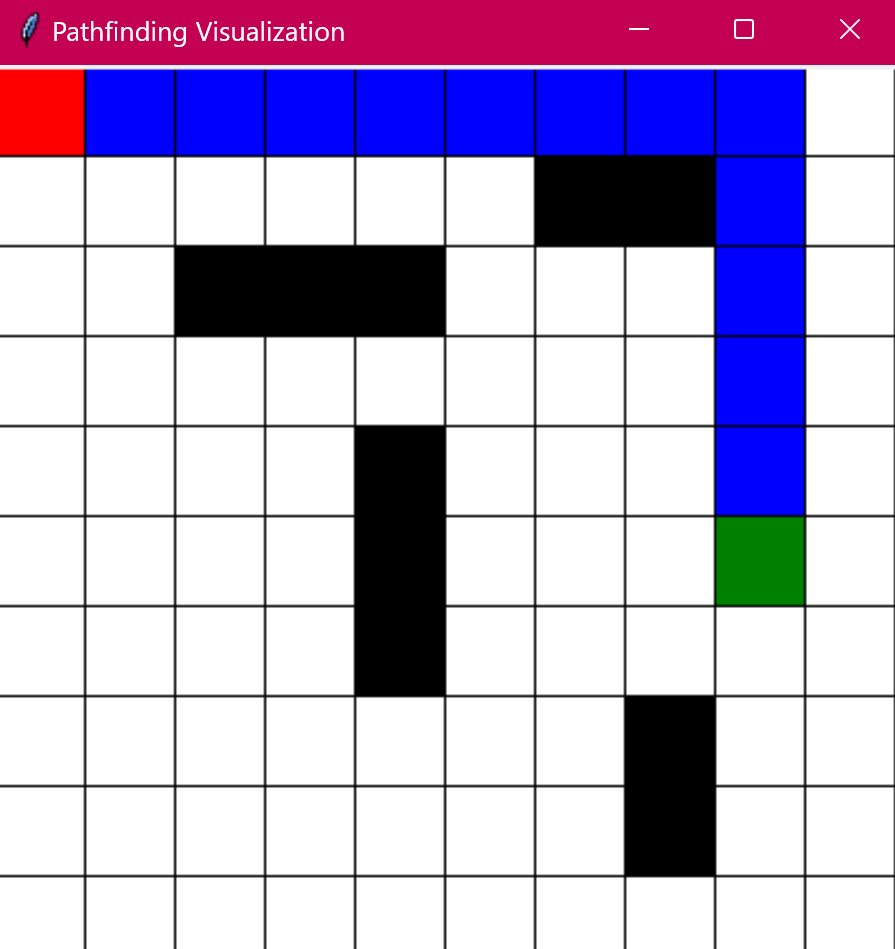
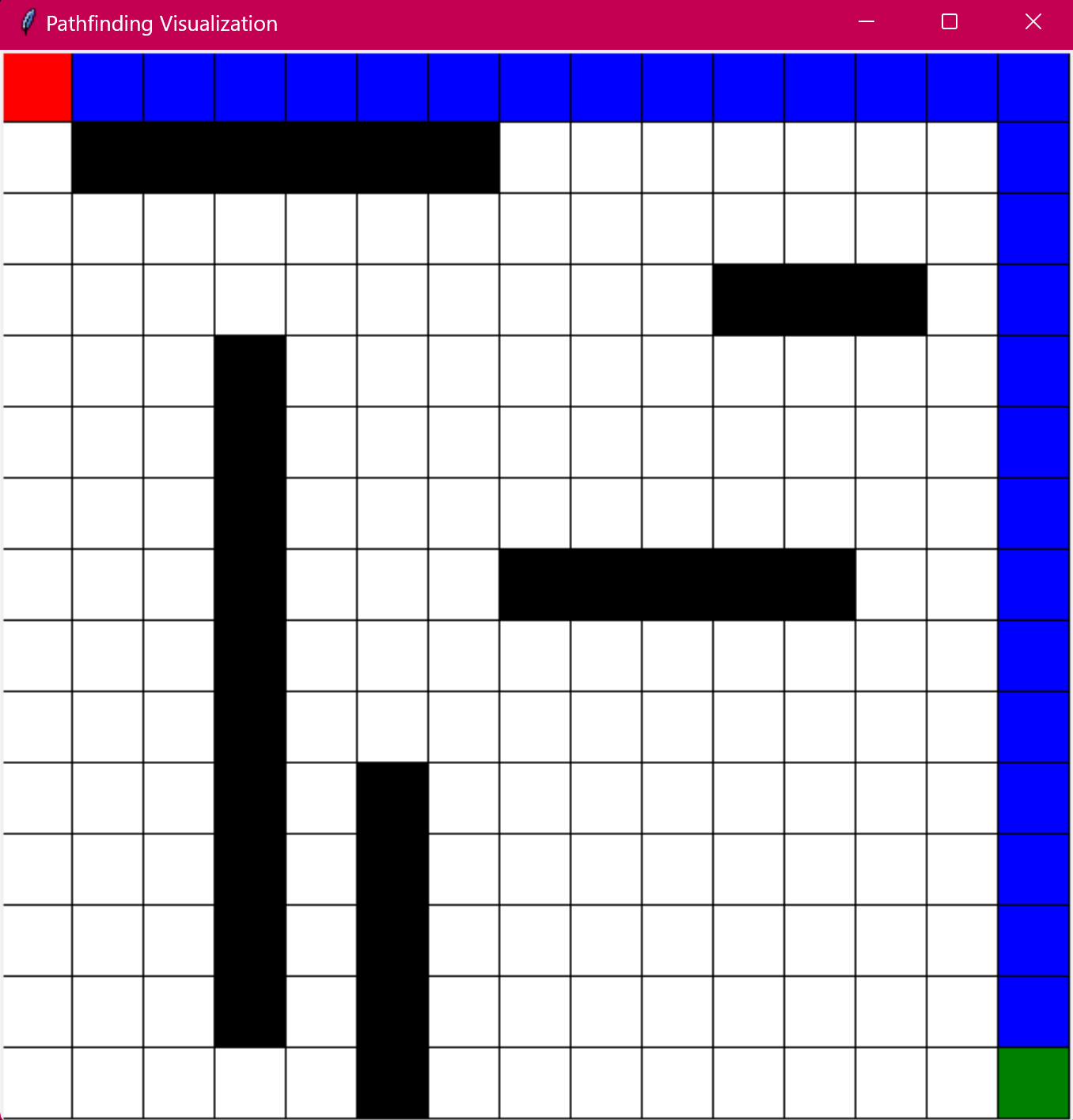
color = colors['goal']

canvas.create\_rectangle(x1, y1, x2, y2, *fill*=color)

root.mainloop()

* The draw\_map function is used to build a graphical representation of the map. It uses a Tkinter window and creates a canvas where the grid will be drawn based on the size of the map.The colors are defined in a dictionary for later usage. White for empty spaces, red for root node, green for goal node, path is blue and black represents the wall.
* The for loop is used to draw the map. The x1 and y1 coordinates represent the top-left corner of the rectangle, while x2 and y2 represent the bottom-right corner. The rectangle is calculated by multiplying the column index and row index by the size of each cell. The gui.py will read data from a text file where the map will be written in number 1 and 0, 1 for wall and 0 for spaces.
* Finally, the code draws the rectangles on a Tkinter canvas using the calculated coordinates and determined colors, visually representing the grid's layout. Once all rectangles are drawn, the root.mainloop() function is called to initiate the Tkinter event loop, keeping the window open for user interaction

1. Sample file

1. **Conclusion**

* To wrap up this report, based on the data extract from 10 different test cases, among the 5 searching methods:
* For uninformed search, DFS could be better for this problem if the aim is to minimize the memory usage. BFS would likely work well if the goal is to find the shortest path. IDS could be the best because it is a combination of DFS and BFS; however, I am currently facing a bug that IDS cannot respond to when searching in a large size map. With uninformed search, it can be efficient when searching inside a small to medium map.
* For informed search, A\* is guaranteed to find the best path if the heuristic used is admissible. GBFS can provide a faster solution but not guaranteed the best option.
* In conclusion, for this problem, informed search methods should be the most optimal choice to choose, especially A\* search.

1. **Acknowledge**
2. Pedamkar, P. (2023, July 28). Iterative Deepening Depth-First Search | Advantages and Disadvantages. EDUCBA.
3. <https://www.youtube.com/watch?v=BK8cEWKHCkY>. Explanation of Iterative Deepening Search
4. <https://realpython.com/python-gui-tkinter/>. Guide in using python Tkinter for GUI visualization

**References**

1. Pedamkar, P. (2023, July 28). *Iterative Deepening Depth-First Search | Advantages and Disadvantages*. EDUCBA. Retrieved October 20, 2024, from <https://www.educba.com/iterative-deepening-depth-first-search/>
2. Mahesh, H. (2023, December 31). Iterative Deepening Search | IDS Search | DFS Algorithm in Artificial Intelligence [Video]. YouTube. <https://www.youtube.com/watch?v=BK8cEWKHCkY>
3. David, A. (2023, April 25). Python GUI programming with Tkinter. Real Python. Retrieved October 20, 2024, from

<https://realpython.com/python-gui-tkinter/>