VNUHCM - UNIVERSITY OF SCIENCE

Lab 1: Searching and Visualizing Algorithms in a Maze

Student's name : Phan Xuân Nam

Student's ID : 20127247

Instructors : Phạm Trọng Nghĩa, Nguyễn Thái Vũ

Subject : Introduction to Artificial Intelligence

Contents

Abstract

Algorithms are playing a key role in the development of Artificial Intelligence (AI). AI has only been developed for nearly 70 years, which is still a humble figure compared to other fields. That is the reason why in recent years, many developed countries have spent tonnes of money on different projects about algorithms to help bring out the academic outcome of experimenting. In this particular Lab exercise, we will be working with a simulation of a real-world problem: Searching Algorithms in a maze. Even though it is a small scale, we will discuss and give out explanations that each algorithm has to offer. We will also explain and visualize some of the most notable searching algorithms.

I. Lab's description

In this Lab exercise, we implement 5 algorithms:

- BFS (Breadth-first Search)
- UCS (Uniform-cost Search)
- IDS (Iterative Deepening Search)
- GBFS (Greedy Best-first Search)
- A* (A star)

In a maze, obstacles are indicated by the coordinates of each obstacle. Our task is to implement each one above and visualize them.

II. Implementation

A. Reading from a File

The file's name is "**input.txt**". The format in the file is:

- The first line: the size of the matrix (columns, rows). Save them in a list called **size**.
- The second line: the initial position and the end position (for each coordinate follows the format: rows, columns

In this report, we will assume that the start position is the first 2 numbers and the end position is the last 2 numbers. We save them in a list called **StartGoal**.

- The third line: number of obstacles. I decide not to read this because we will count the number of obstacles based on the rest of the lines in **input.txt**
- The rest of the lines: for each line, for every two numbers, we will group them as a coordinate of the obstacles. We will save each obstacle in a 2D list called **obstacles**.

B. Maze

We will generate a 2D array with ***columns + 1 * rows +2*** from the *NumPy library*. We also create a border, indicated by the number 2. For the obstacles, we will mark the coordinates of each obstacle as 4s and the lines to connect them are marked with 3s. After all that is done, we can add the position of the start and end positions on the board. Both of them are now marked with the number 9.

So how do we visualize it? The answer goes to the implementation of the *turtle library*. To make it simple: we create a function to draw a square, then loop it through the size of the matrix to create a so-called maze. Then, for each number, we will apply a different color to make the expanding and tracking algorithm visually colorful and eye-catching.

C. Declarations, Functions and Algorithms

Declarations of global values

- SIZE: the pixel that the cursor of the turtle can go
- startX: the starting position of the turtle at the x-axis
- startY: the starting position of the turtle at the y-axis
- SQUARE_SIDE: the sides of the square, which are 4 sides
- directionX, directionY: list of directions that the cursor goes, in this project is: left, down, right, up
- colorList: list of colors to be used to visualize the maze

- screen, cursor: basic attributes of the turtle library to show the graphical representation window (*Screen() and Turtle()*)

Functions

Turtle's functions:

- **drawSquare(cursor, color)**: draw a square based on the position of the cursor and color
- **drawBoard(size, StartGoal, obstacle, maze)**: visualize the matrix in graphics. Each parameter will have a separate number code that matches a color code.
- **drawPosition(x, y, cursor, color)**: draw a specific position in the maze. The cursor will draw a square.
- **setupBoard()**: basic settings to initialize the screen and the drawing speed of the cursor.

Maze's functions:

- **readFile()**: read from the **input.txt**. For each attribute, we will store them in lists: **size**, **StartGoal**, **obstacles** respectively
- **draw_line(mat, x0, y0, x1, y1, inplace=False)**: connect the coordinates of an obstacle. This is done by following *Bresenham's line algorithm*.
- **createMaze(size, StartGoal, obstacle)**: create a ***size[0]+1 * size[1]+1*** 2D matrix with the border, starting and ending position. Then, we will connect the coordinates of each obstacle to create a complete one. Then we will use the **draw_line()** to mark them with a number code.

Algorithms

Before breaking down how each algorithm works, I want to mention that each algorithm will have a similar format.

The strategy is to: work straight on the maze so that we don't have to use a visited list to check. Also, after the maze has been modified, we will use a map/dictionary to perform the backtracking to find the path from start to end. Finally, we will color the path and the expanded nodes on the graphics. The path costs and numbers of expanded nodes will be displayed on the terminal.

So, on each algorithm, we will mainly discuss the methods to convert the original matrix to the corresponding algorithm.

Uninformed Search Strategies

- **Breadth-first search (BFS):**

The idea of BFS is to use a queue to append the expanded nodes. When the node is reached, it will be popped out and put on a visited list, in this case, we don't have to use it because our maze uses an integer system to mark them which makes it easier to track and use less space.

The code for BFS:

```
    solution = {}
    solution[x, y] = (x, y) #backtrack
    costExpandedNode = 0
    goalState = False
    frontier = deque()
    frontier.append((x, y))

    while len(frontier) > 0:
        currCell = frontier.popleft()
        costExpandedNode+=1
        currX = currCell[0]
        currY = currCell[1]
        for i in range(4):
            if (goalState == True): break
            adjacentX = currX + directionX[i]
            adjacentY = currY + directionY[i]
            if (adjacentY > size[0] +1 or adjacentX > size[1]+1 or ad-
jacentX < 0 or adjacentY < 0):
                continue
            if adjacentX == startGoal[0] and adjacentY ==startGoal[1]:
                continue
            if (maze[adjacentX][adjacentY] == 0):
                frontier.append((adjacentX, adjacentY))
                solution[adjacentX, adjacentY] = (currX, currY)
                maze[adjacentX][adjacentY] = 1
                color = 'cyan'
```

```
                drawPosition(adjacentY,adjacentX, cursor, color)
            if maze[adjacentX][adjacentY] == 9:
                solution[adjacentX, adjacentY] = (currX, currY)
                goalState = True
```

I will use a dictionary called *solution* to keep track of the key-value of the explored nodes. I also initialize a queue called *frontier* and append the staring point. The searching only ends if the queue is empty. A variable for storing the cost of expanding new nodes is called *costExpandedNode*.

While the queue is not empty, we will take out the most recently added node and assign them with the *currX* and *currrY*. We check the adjacency of the current node by using the *directionX*, *directionY*.

- If our adjacent nodes are out of the maze's border or match its predecessor/parent/previous/current nodes, we will skip that.
- If the value of the adjacent nodes we're working on has a 0, we will add them to the frontier and the dictionary, increment the *costExpandedNode* by one, mark them with a number code and color code and draw that position.
- If the adjacent node matches the ending node, we stop the process and add it to the dictionary. We force the loop to break and perform the backtracking process.

<u>The code for backtracking</u> *(the backtracking applies to all algorithms performed in this lab so we will only mention this once)*:

```
    costPath = 0
    goalX, goalY = startGoal[2], startGoal[3]
    a, b = goalX, goalY
    while(goalX, goalY) != (x, y):
        color = 'red'
        maze[goalX][goalY] = -1
        costPath +=1
        drawPosition(goalY,goalX, cursor, color)
        goalX, goalY = solution[goalX, goalY]
    drawPosition(b,a, cursor, 'green')
```

We initialize the *costPath* to store the cost to get to the end from the starting point. The backtracking algorithm is fairly simple: if our current position is not the starting position, we will take the key and assign it to our current position. We also mark them with a number code and a color code to draw. It keeps performing like this until it reaches the starting point and the loop ends.

Pros and Cons:

Advantages	Disadvantages
+ The solution will be found if there is a solution + If there are multiple choices, it will find the most optimal one.	+ Require too many spaces + Quite time-consuming if the solution is far away from the starting point

- **Uniform-cost Search (UCS)**

UCS is a different version of BFS where it uses the cost system to opt for the smallest value to traverse to. In other words, if BFS combines with a different cost every time we move, it's Dijkstra's algorithm. Without the cost, it's purely BFS. In this lab, because the adjacent nodes of a node cost the same, it doesn't really affect the output if we change the position of the ending node. Overall, it's BFS with the cost system to show the least costly path.

The code for UCS:

```
    solution = {}
    x, y = startGoal[0], startGoal[1]
    solution[x, y] = (x, y) #backtrack
    costExpandedNode = 0
    goalState = False
    costList = np.ones([size[1]+1, size[0]+1], dtype = int)
    frontier = []
    frontier.append([x, y])

    while len(frontier) > 0:
        min = 100000
```

```
        index = -100000
        for i in range(len(frontier)):
            temp = frontier[i]
            if (costList[temp[1]][temp[2]] < min):
                min = costList[temp[1]][temp[2]]
                index = i
        currCell = frontier.pop(index)
        costExpandedNode+=1
        currX = currCell[0]
        currY = currCell[1]
        for i in range(4):
            adjacentX = currX + directionX[i]
            adjacentY = currY + directionY[i]
            if (adjacentY > size[0] +1 or adjacentX > size[1]+1 or ad-
jacentX < 0 or adjacentY < 0):
                continue
            if adjacentX == startGoal[0] and adjacentY ==startGoal[1]:
                continue
            if (maze[adjacentX][adjacentY] == 0):
                frontier.append((adjacentX, adjacentY))
                costList[adjacentX][adjacentY] = cost-
List[currX][currY] + 1
                solution[adjacentX, adjacentY] = (currX, currY)
                maze[adjacentX][adjacentY] = 1
                color = 'cyan'
                drawPosition(adjacentY,adjacentX, cursor, color)
            if adjacentX == startGoal[2] and adjacentY ==startGoal[3]:
                solution[adjacentX, adjacentY] = (currX, currY)
                costList[adjacentX][adjacentY] = cost-
List[currX][currY] + 1
                goalState = True
                break

        if (goalState == True): break
```

As you can see above, it's 90% similar to BFS but there's an additional thing called a cost system. In short, every time we append a new coordinate, we check if the new coordinate's value we discover is bigger or smaller than the min. We

take the minimum value's index and pop out of the frontier, then perform like normally how BFS would do for the rest of the code.

Pros and Cons:

Advantages	Disadvantages
+ It helps to find the path with the lowest sum of the cost in a weighted graph + Every time it moves to a new node, the lowest cost is always prioritized	+ Require too many spaces + Time-consuming if the solution is far away from the starting point

- **Iterative Deepening Search (IDS) with DFS**

IDS is a combination of DFS and BFS. In this case, we will apply DFS as a core component in this algorithm. The idea behind the combination is that:

We utilize the depth-limit system from the depth-limit system (DLS) to perform DFS in a BFS fashion. Or more specifically:

- We set the depth level to 0. Since the starting position is set to default at level 0, the process stops and restarts. The depth level is now incremented by 1
- Depth level = 1: we can explore adjacent nodes since the current level of the starting position is smaller than the depth level. While exploring, if the end node is found, return the path and end the process. If not, we keep exploring and give the adjacent node the level equal to the parent nodes' level + 1. Since the level of adjacent nodes is equal to the depth level, restart the process and increment the depth level to 2.
- Depth level = 2:…
- Depth level = n:…
- If not found: a solution is not available

The code for UCS:

```
    solution = {}
    x, y = startGoal[0], startGoal[1]
    solution[x, y] = (x, y) #backtrack
    expandedNode = 0
    goalState = False
    deepLevel = 0
    returnLevel = []
    while not goalState:
      expandedNode = 0
      levelList = np.zeros([size[1]+1, size[0]+1], dtype = int)
      frontierStack = []
      frontierStack.append((x, y))
      while len(frontierStack) > 0:
          currCell = frontierStack.pop()
          expandedNode +=1
          currX = currCell[0]
          currY = currCell[1]
          for i in range(4):
              adjacentX = currX + directionX[i]
              adjacentY = currY + directionY[i]
              if (levelList[adjacentX][adjacentY] >= deepLevel): break
              if (adjacentY > size[0]  or adjacentX > size[1] or adja-
centX < 1 or adjacentY < 1): continue
              if (adjacentX == startGoal[0] and adjacentY == start-
Goal[1]): continue
              if (maze[adjacentX][adjacentY] == 0 and levelList[adja-
centX][adjacentY] == 0):
                  frontierStack.append([adjacentX, adjacentY])
                  levelList[adjacentX][adjacentY] = level-
List[currX][currY] + 1
                  solution[adjacentX, adjacentY] = (currX, currY)
                  drawPosition(adjacentY,adjacentX, cursor, 'cy-
an')
              if (adjacentX == startGoal[2] and adjacentY == start-
Goal[3]):
                  solution[adjacentX, adjacentY] = (currX, currY)
                  goalState = True
                  break
      deepLevel+=1
```

As you can see above, the main difference between the three uninformed search algorithms is that it has to restart the process from scratch when the goal is not found. We will debunk how the algorithm processes and what needs to be done:

- We will initialize like what BFS does. Adding to that is a variable called *deepLevel* to assist in the limiting of going deeper into the maze and a *goalState* to mark when the goal is found. Set the *deepLevel* to 0.
- If the goal is not found, we will initialize a *levelList* with the size the same as the original maze. Each coordinate in the *levelList* has the value of 0. We also create a *frontierStack* to store the appended node. We append the starting position first to the *frontierStack*.
- While there is more than 0 component in the *frontierStack*, we will pop them out and look for the adjacent nodes. Since the starting node has a level of 0, we can't explore the adjacent nodes and will break the while-loop and increment the *deepLevel* to 1.
- We will do everything again in steps 2 and 3. In step 3, since the level of the starting node is smaller than the *deepLevel*, we can explore the adjacent nodes and set them to level 1. If the goal is found during the exploring, we set the *goalState* to True and return the path. Or else, we only append the discovered nodes to the *frontierStack* and draw the explored nodes.

  The process will keep going on until it finds the goal. Then we will perform the backtracking algorithm

Pros and Cons:

Advantages	**Disadvantages**
+ Using modest memory (obtained from DFS) + Inherited the completeness and optimality of BFS	+ The process will restart all over if, during that *deepLevel,* the goal is not found + Very time-consuming if the solution is far away from the starting point

Informed Search Strategies

(note: we will be using Manhattan distance as a heuristic for GBFS and A*)

- **Greedy Best-first search (GBFS)**

GBFS is based on the greedy algorithm and Best-first search algorithm.

The GBFS algorithm tries to explore the node that is closest to the goal. This algorithm evaluates nodes by using the heuristic function h(n), that is, the evaluation function is equal to the heuristic function, f(n) = h(n). This equivalency is what makes the search algorithm 'greedy.'

The code for GBFS:

```
    heuristicList = []
    frontier = []
    heuristic = getManhattanDist(startGoal, startGoal[0], start-
Goal[1])
    goalState = False
    solution = {}
    solution[startGoal[0], startGoal[1]] = (startGoal[0], start-
Goal[1])
    costExpandedNode = 0
    for i in range(1, size[1], 1):
        temp = []
        for j in range(1, size[0], 1):
            temp.append(getManhattanDist(startGoal, i, j))
        heuristicList.append(temp)
    frontier.append((startGoal[0], startGoal[1]))
    while len(frontier) > 0:
        poppedVal = 0
        index = -100000
        indexHeuristic = 100000
        for i in range(len(frontier)):
            temp = heuristicList[frontier[i][0]][frontier[i][1]]
            if (indexHeuristic > temp):
                indexHeuristic = temp
                index = i
        poppedVal = frontier[index]
        frontier.pop(index)
```

```
        currX, currY = poppedVal[0], poppedVal[1]
        for i in range(4):
            adjacentX = currX + directionX[i]
            adjacentY = currY + directionY[i]
            if (adjacentY > size[0] +1 or adjacentX > size[1]+1 or ad-
jacentX < 0 or adjacentY < 0): continue
            if adjacentX == startGoal[0] and adjacentY ==startGoal[1]:
                continue
            if (maze[adjacentX][adjacentY] == 0):
                frontier.append((adjacentX, adjacentY))
                costExpandedNode += 1
                solution[adjacentX, adjacentY] = (currX, currY)
                maze[adjacentX][adjacentY] = 1
                color = 'cyan'
                drawPosition(adjacentY,adjacentX, cursor, color)
           if (adjacentX == startGoal[2] and adjacentY == startGoal[3]):
                solution[adjacentX, adjacentY] = (currX, currY)
                goalState = True
        if goalState == True: break
```

The idea for this algorithm is:

- Create a 2D *heuristicList* to store heuristics. The function to return heuristics is based on the Manhattan distance

```
def getManhattanDist(startGoal, currX, currY):
    return abs(currX - startGoal[2]) + abs(currY - startGoal[3])
```

- We push the starting point to a *frontier*.
- While the length of the frontier is more than 0, we will check for the smallest heuristic in the *heuristicList* and pop it out of the *frontier,* and store it to a variable called *poppedVal*
- We will check for the adjacent nodes of *poppedVal* to see if there's any valid grid to explore. If there is, we will append the adjacent nodes to the *frontier* and draw those nodes. If we find the goal, we will set the *goalState* to True and break out of the loops.

Pros and Cons:

Advantages	Disadvantages
+ More efficient than BFS and DFS	+ Cannot guarantee to find the shortest path so it's not optimal + Bad heuristic will not work in a long term + Might get into a loop

- **A* (A-star)**

A* is used wisely in finding the shortest path from the source to the finishing point. The main difference between GBFS and A* is that A* uses another system to choose the best path which follows this formula:

$$f(n) = g(n) + h(n)$$

In this formula:

n: the current node

g(n): the cost from the starting point to node n, based on the path given from the graph or discovered before

h(n): the heuristic from node n to the final destination

f(n): the sum of g(n) and h(n)

Essentially, after we explore the adjacent nodes of the starting node, the A* algorithm chooses the node that has the smallest f(n) to go forward.

That's the key difference in A*; whereas, in GBFS, it only considers the h(n) to move forward which isn't optimal as they might run into a loop or take a long time to discover. A* is also known as a combination of GBFS and UCS which helps to find the shortest path in a shorter time.

The code for A*:

```
    heuristicList = []
    frontier = []
```

```
    frontier.append((startGoal[0], startGoal[1]))
    solution = {}
    solution[(startGoal[0], startGoal[1])] = (startGoal[0], start-
Goal[1])
    costList = np.ones([size[1]+1, size[0]+1], dtype = int)
    goalState = False
    costExpandedNode = 0

    for i in range(1, size[1], 1):
        temp = []
        for j in range(1, size[0], 1):
            temp.append(getManhattanDist(startGoal, i, j))
        heuristicList.append(temp)

    while len(frontier) > 0:
        poppedVal = 0
        index = -100000
        indexHeuristic = 100000
        for i in range(len(frontier)):
            temp = heuristicList[frontier[i][0]][frontier[i][1]] +
costList[frontier[i][0]][frontier[i][1]]
            if indexHeuristic > temp:
                indexHeuristic = temp
                index = i
        poppedVal = frontier[index]
        frontier.pop(index)

        currX, currY = poppedVal[0], poppedVal[1]
        for i in range(4):
            adjacentX = currX + directionX[i]
            adjacentY = currY + directionY[i]
            if (adjacentY > size[0] +1 or adjacentX > size[1]+1 or ad-
jacentX < 0 or adjacentY < 0):
                continue
            if adjacentX == startGoal[0] and adjacentY ==startGoal[1]:
                continue
            if (maze[adjacentX][adjacentY] == 0):
                frontier.append((adjacentX, adjacentY))
                costExpandedNode += 1
```

```
                solution[adjacentX, adjacentY] = (currX, currY)
                maze[adjacentX][adjacentY] = 1
                color = 'cyan'
                drawPosition(adjacentY,adjacentX, cursor, color)
            if (adjacentX == startGoal[2] and adjacentY == start-
Goal[3]):
                solution[adjacentX, adjacentY] = (currX, currY)
                goalState = True
        if goalState == True: break
```

In the code above, we can see that a *costList* has been initialized with a 2D array of ones. We then find the smallest value from the sum of *heuristicList* and *costList*.

After finding the smallest one, we take the index of that one and pop it out of the *frontier*. From there, we perform the exploration normally.

Pros and Cons:

Advantages	Disadvantages
+ Optimal and complete + Can solve complex problems	+ Have to store lots of values hence require memory + Time and space complexity is exponential

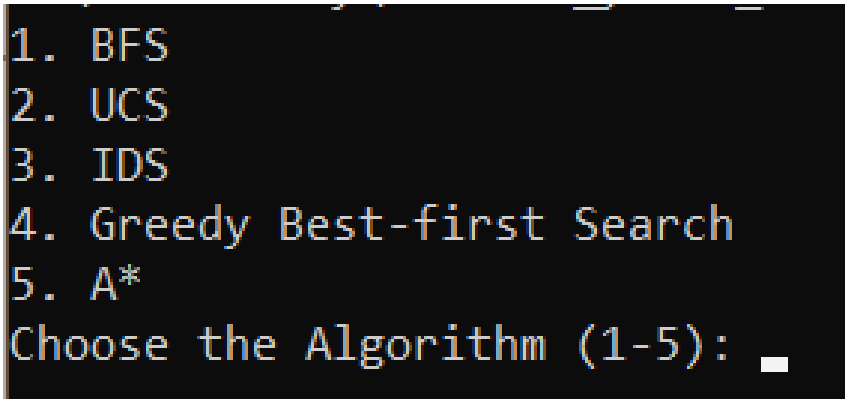
D. Examples and Visualization

For input:

22 18	# size
2 2 16 19	# start and goal
3	# number of obstacles
4 4 5 9 8 10 9 5	# coordinates of obstacle 1
8 12 8 17 13 12	# coordinates of obstacle 2
11 1 11 6 14 6 14 1	# coordinates of obstacle 3

When we run the program, the terminal will ask you to choose a number from 1 to 5, each number returns a different searching algorithm and how it runs based on the obstacles you fill in the .txt file.

The menu is shown below:

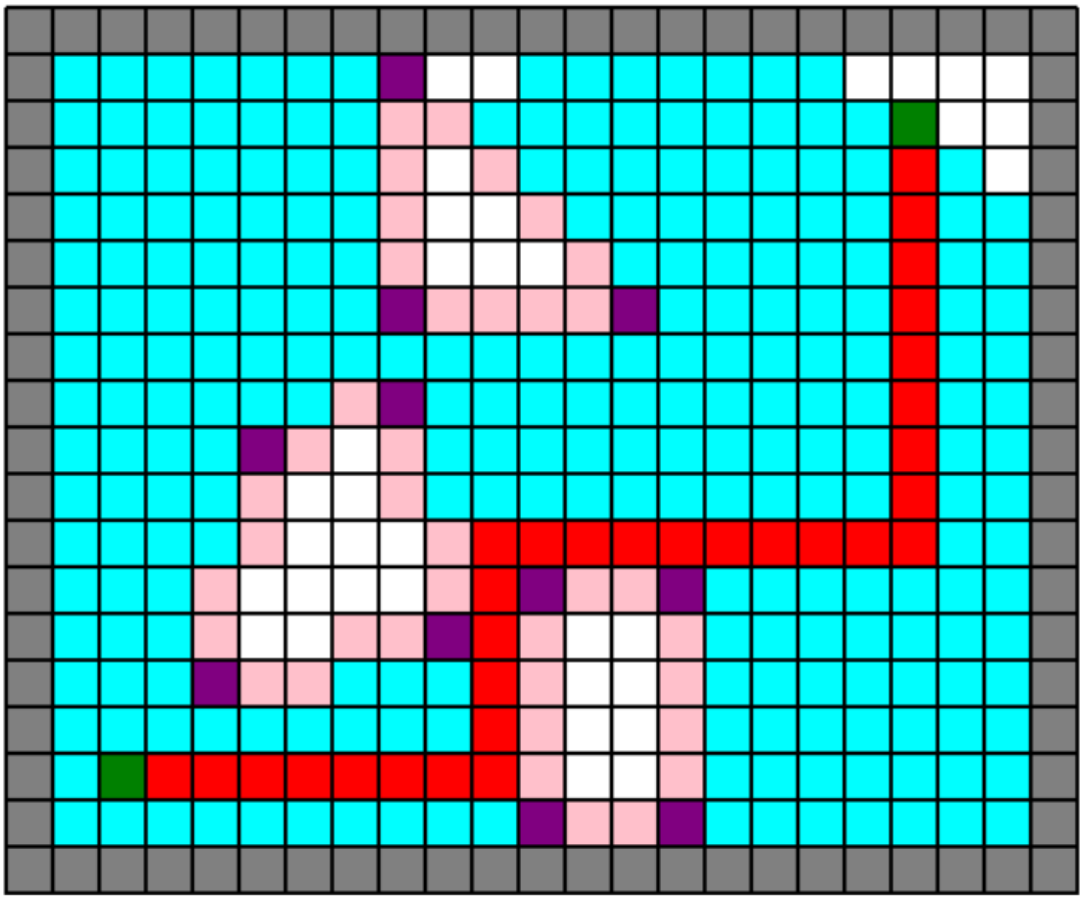


For output and visualization:

BFS

```
Cost of the path: 31
Cost of the expanded nodes: 272
```

Terminal's output

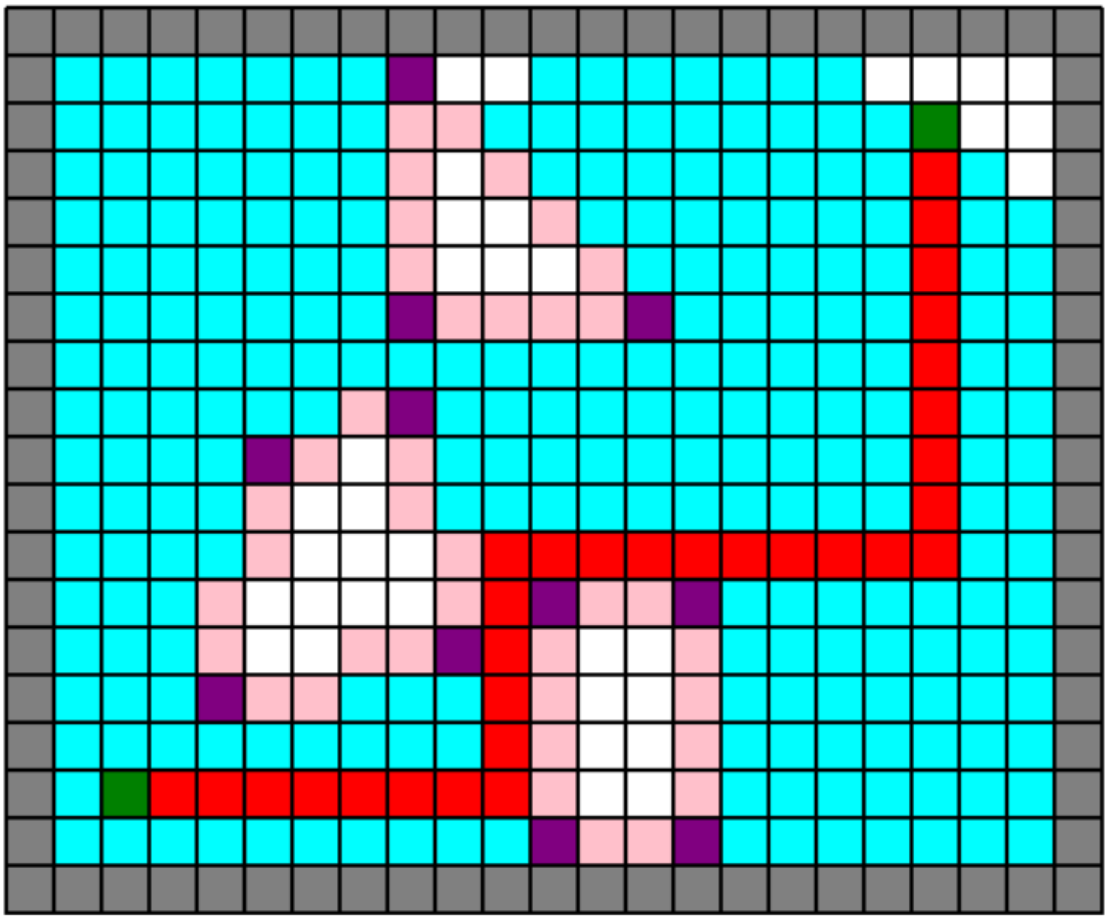


Visualization

UCS

```
Cost of the path: 31
Cost of the expanded nodes: 272
```

Terminal's output

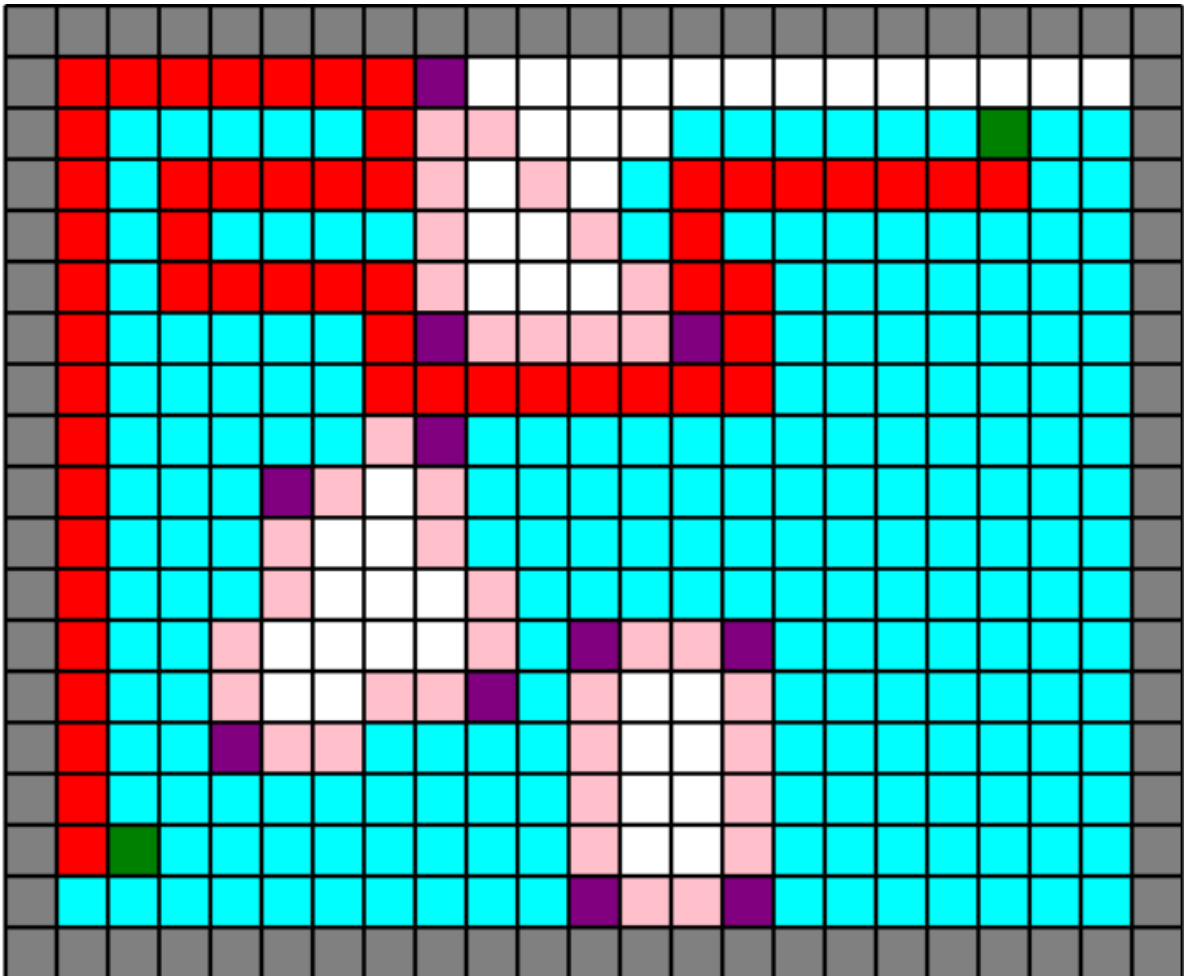


Visualization

IDS

```
Deep level: 49
Cost of expanded nodes: 264
Path cost: 55
```

Terminal's output

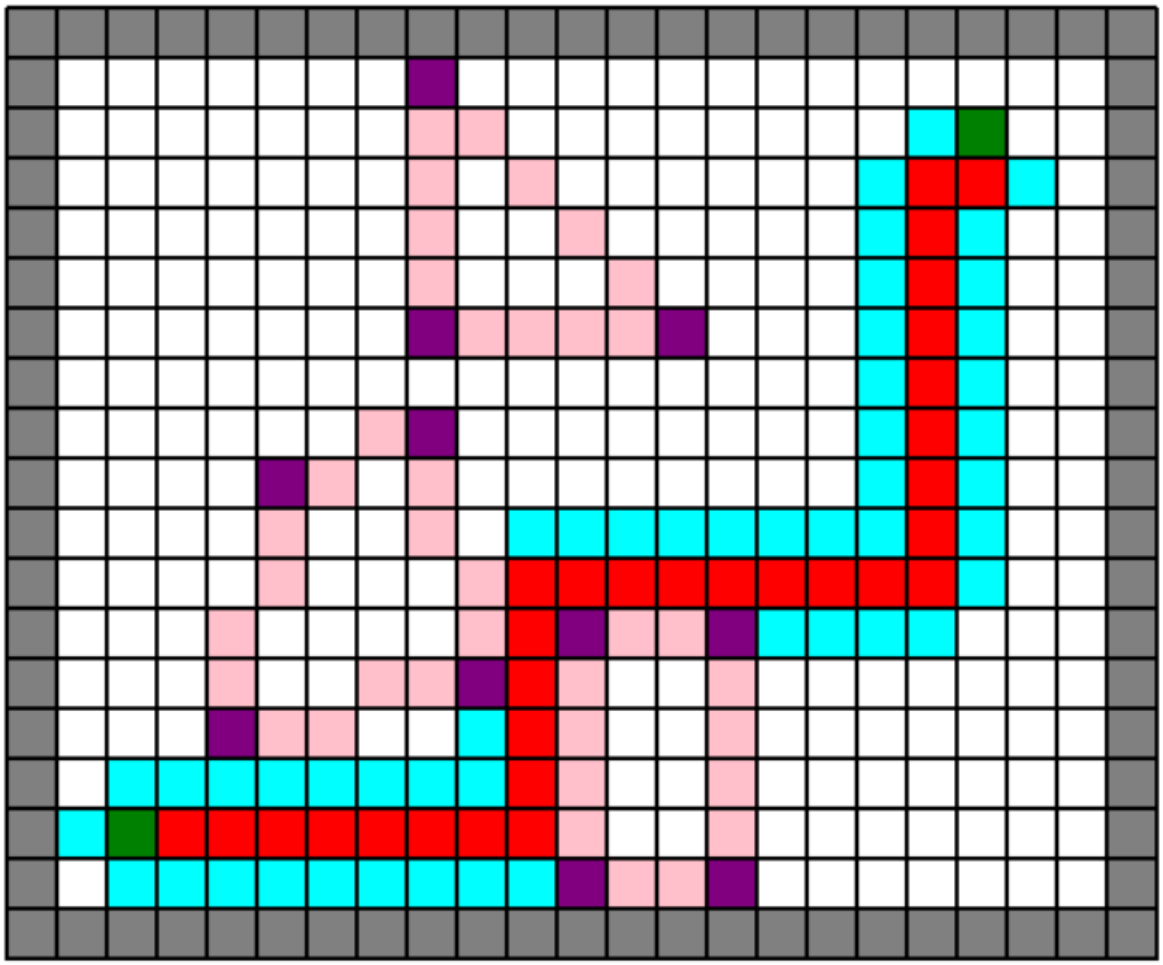


Visualization

GBFS

```
Cost of the path (included the goal): 31
Cost of the expanded nodes: 78
```

Terminal's output

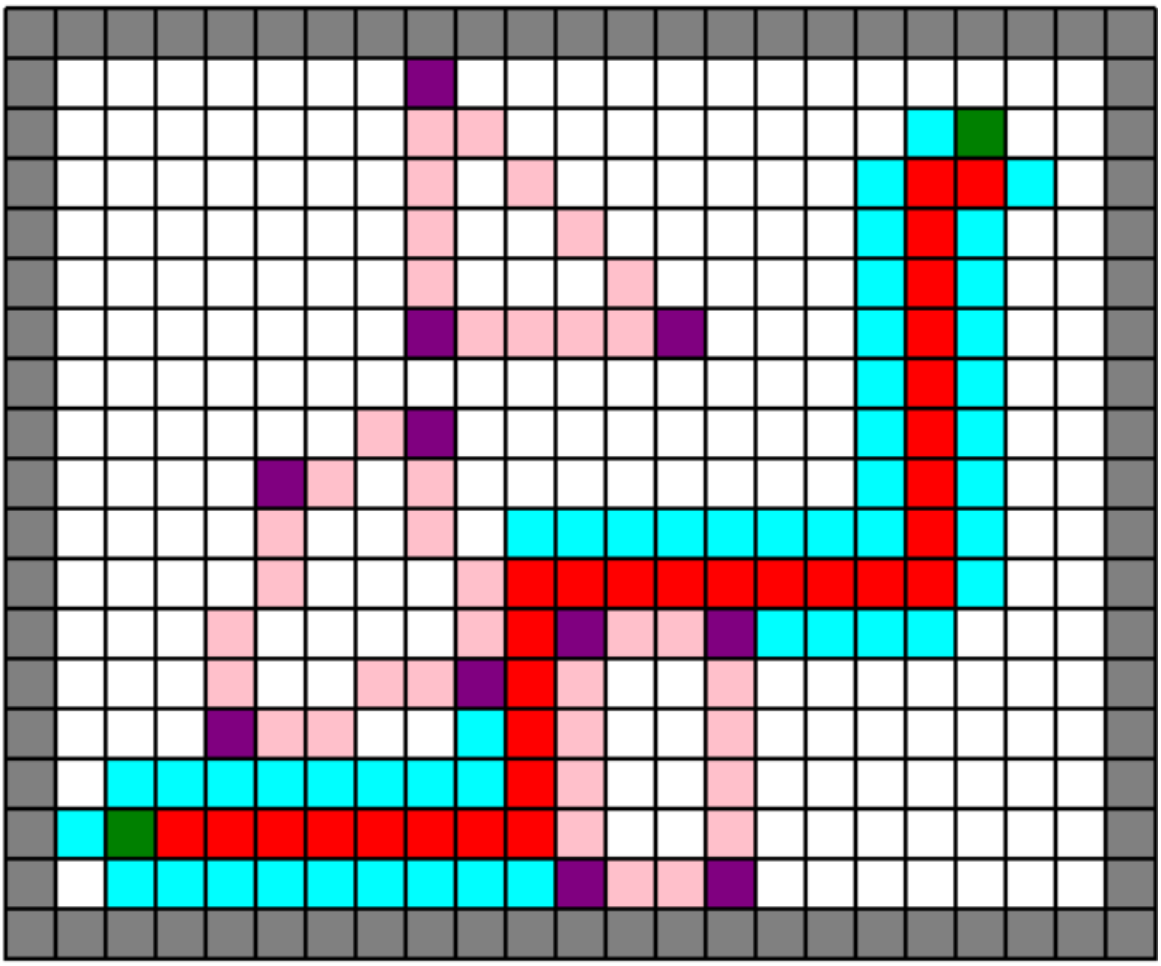


Visualization

A*

```
Cost of the path (included the goal): 31
Cost of the expanded nodes: 78
```

Terminal's output



Visualization

III. References

GeeksforGeeks. (n.d.). *GeeksforGeeks*. Retrieved from https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/

GeeksforGeeks. (n.d.). *GeeksforGeeks*. Retrieved from https://www.geeksforgeeks.org/best-first-search-informed-search/

GeeksforGeeks. (n.d.). *GeeksforGeeks*. Retrieved from https://www.geeksforgeeks.org/a-search-algorithm/

GeeksforGeeks. (n.d.). *GeeksforGeeks*. Retrieved from https://www.geeksforgeeks.org/bresenhams-line-generation-algorithm/

ramandeep8421. (n.d.). *GeeksforGeeks* . Retrieved from https://www.geeksforgeeks.org/breadth-first-traversal-bfs-on-a-2d-array/