VNU HCMC-UNIVERSITY OF SCIENCE
INFORMATION TECHNOLOGY FACULTY

REPORT LAB01
SEARCH ALGORITHM

Lecturer&Instructor:

- Teacher Phạm Trọng Nghĩa
- Teacher Nguyễn Thái Vũ

Subject: Cơ sở Trí Tuệ Nhân Tạo

Class: 20CLC04

Student: Trần Quang An Quốc

Student ID: 20127304

Thành phố Hồ Chí Minh, ngày 04 tháng 06 năm 2022

I. Preparation:

In this lab01, so as to visualize the algorithm, I used library called turtle and drawed a table, number of positions, some obstables, how the node expand, the source and the goal, the path found after running this algorithm.

For the preparation, I have isValid function so as to check if the node is valid or node.

```
def isValid(row, col):
    # If cell lies out of bounds
    if (row < 1 or col < 1 or row > height - 2  or  col > width - 2):
        return False

    # If cell is already visited
    if (vis[row][col]):
        return False

    # Otherwise
    return True
```

The node will be invalid if it is out of the matrix or lay on the border of the matrix and it will be available if not used before (I used another matrix call vis so as to check if it be visited or not). If all of condition above passed, the function will return true.

II. Breadth-first search

1. The idea of the algorithm

The traversal starts at the root node and checking all adjacency node. We would initialize a queue so as to contain the nodes that are not explored. According to Breadth–First search, in some cases may have cycles and it may cause the repetition. For preventing this, I used “vis” matrix so as to check if it is visited or not (as I mention before).

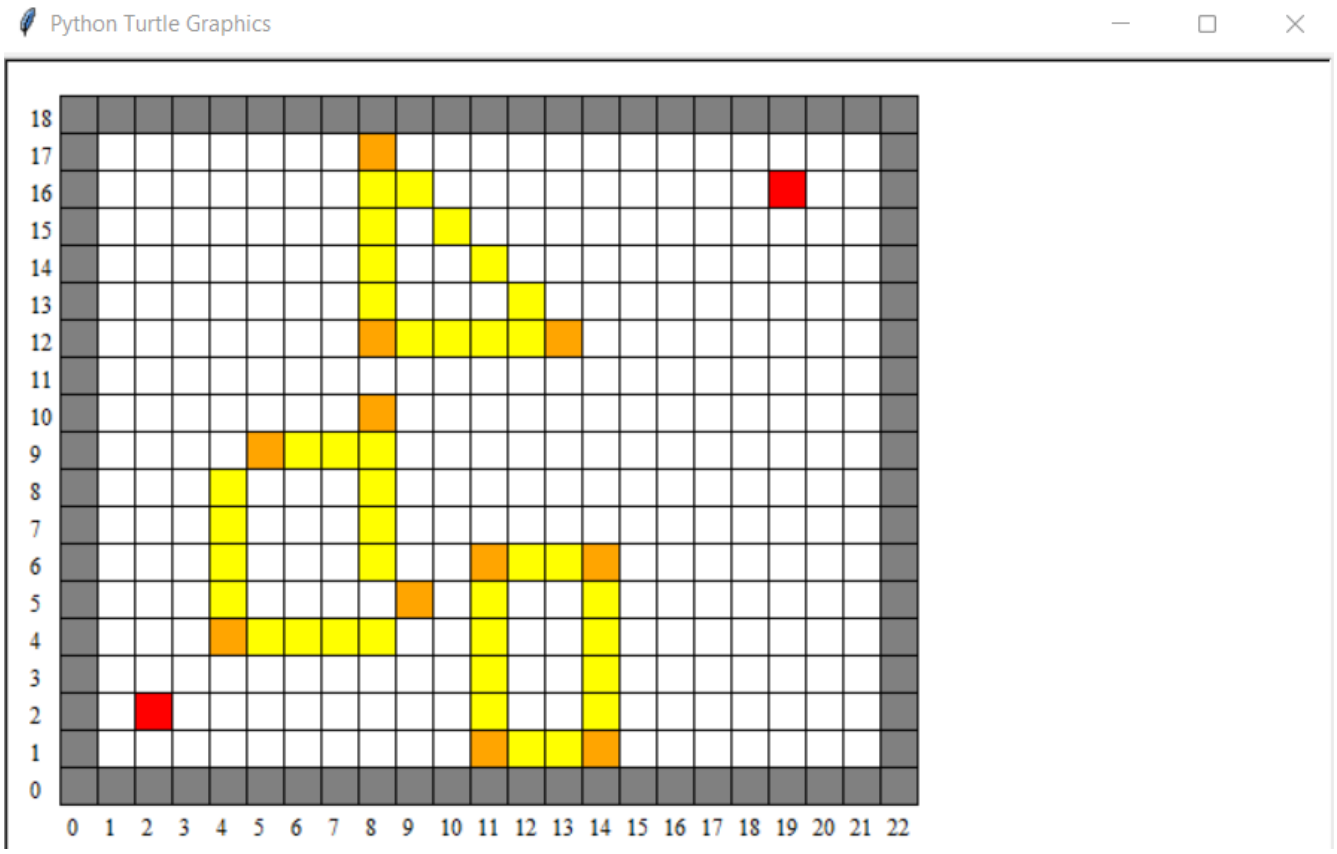
```
# Function to perform the BFS traversal
def BFS():
    # Stores indices of the matrix cells
    frontier = queue()
    #Cost of expanding
    global ex_cost
    ex_cost = 0
    # Mark the starting cell as visited
    # and push it into the queue
    frontier.append([ sourceX, sourceY ])
    vis[sourceX][sourceY] = True

    #Initialize the first element for the path
    path[(sourceX, sourceY)] =  sourceX, sourceY
    # Iterate while the queue
    # is not empty
    while (frontier):
        cell = frontier.popleft()
        x = cell[0]
        y = cell[1]

        # Go to the adjacent cells
        if [x, y] != source and [x, y] != goal: drawPos(x, y, blue)
        for i in range(4):
            adj_y = y + dRow[i]
            adj_x = x + dCol[i]
            if (isValid(adj_y, adj_x) and maze[adj_y][adj_x] == 0):
                if [adj_x, adj_y] != source and [adj_x, adj_y] != goal:
                    drawPos(adj_x, adj_y, palegreen)
                    ex_cost += 1
                frontier.append([adj_x, adj_y])
                path[(adj_x, adj_y)] = x, y
                vis[adj_y][adj_x] = True
        if [adj_x, adj_y] == goal:
            break
```

In 2D matrix, the starting for the algorithm is the source, and initialize the queue by adding the source into the container, set vis at that position is true. We use the loop until the frontier is empty, or the expanding node reach the goal, pop the first element of the queue then finding all adjacency node ands add them into the frontier. At the same time, we set the vis matrix as true to mark it. When investigating the adjacency node, I have ex_cost so as to store the expanding cost of the algorithm by summing one by one. Also at this time I have the path dictionary so as to backtrack the path from the goal back to source

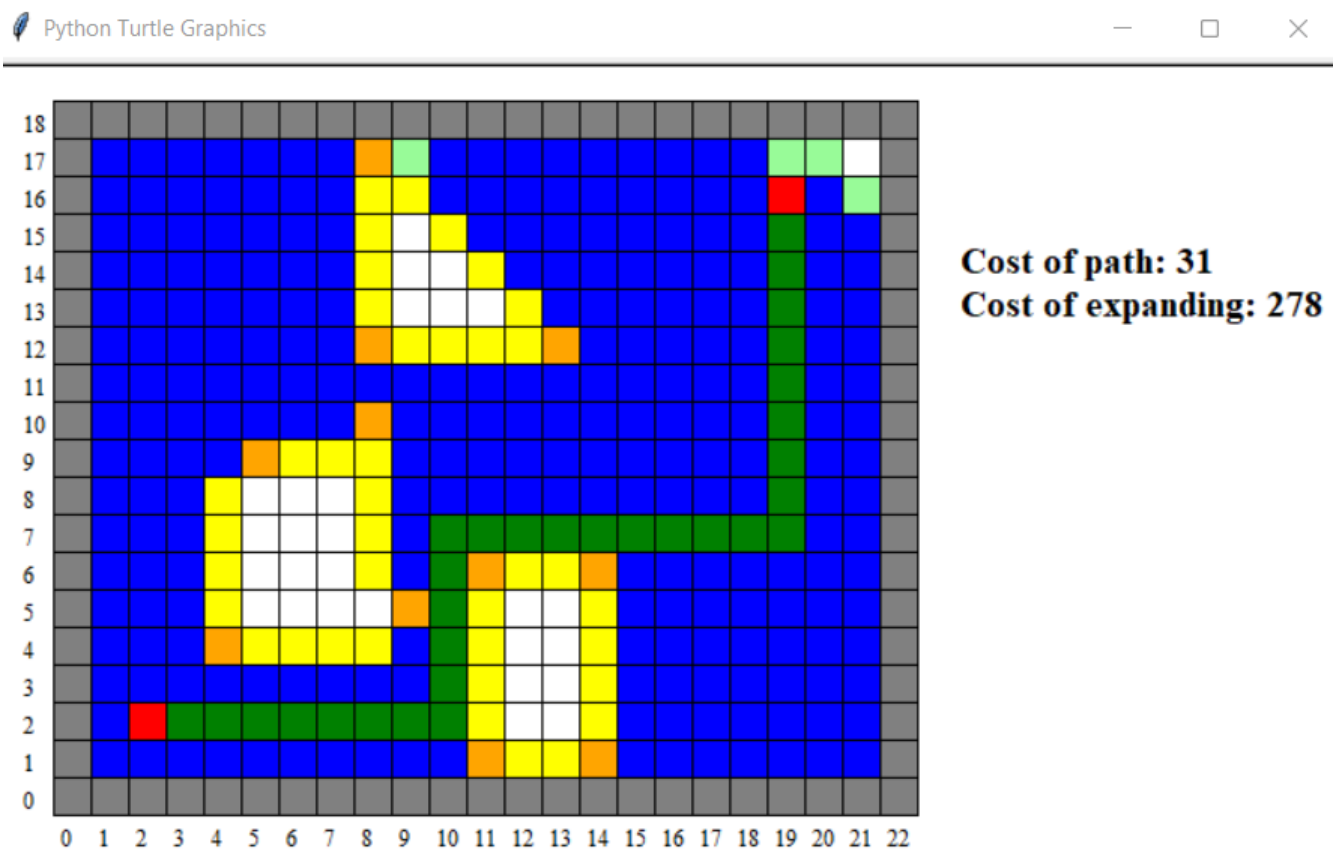
2. *Example*

Input:

```
22 18
2 2 19 16
3
4 4 5 9 8 10 9 5
8 12 8 17 13 12
11 1 11 6 14 6 14 1
```



Output:



3. Advantages and disadvantages

- Advantages:
    - Breadth-First search is easy to implement as it's simple way of thought
    - Breadth-First search can find the shortest path in any case as it expand all nodes.
- Disadvantage
    - Called as a "blind search", It need a large of space so that the performance is poor compared to other search. The space is essential to this algorithm as it used to contain all nodes of current one so as to reach the adjacency one.

III. Uniform-cost search

1. The idea of the algorithm

Uniform Cost search is a variant of Dijikstra's algorithm, as it will calculate the distance from each node to the source and choose the lowest one.

Nodes are expanded, starting from the root, according to the minimum cumulative cost, using priority queue to insert. Repeatedly until the priority queue empty,

remove the highest priority one and check if it's destination. If yes, print the total cost and stop, else enqueue 4 adjacency nodes.

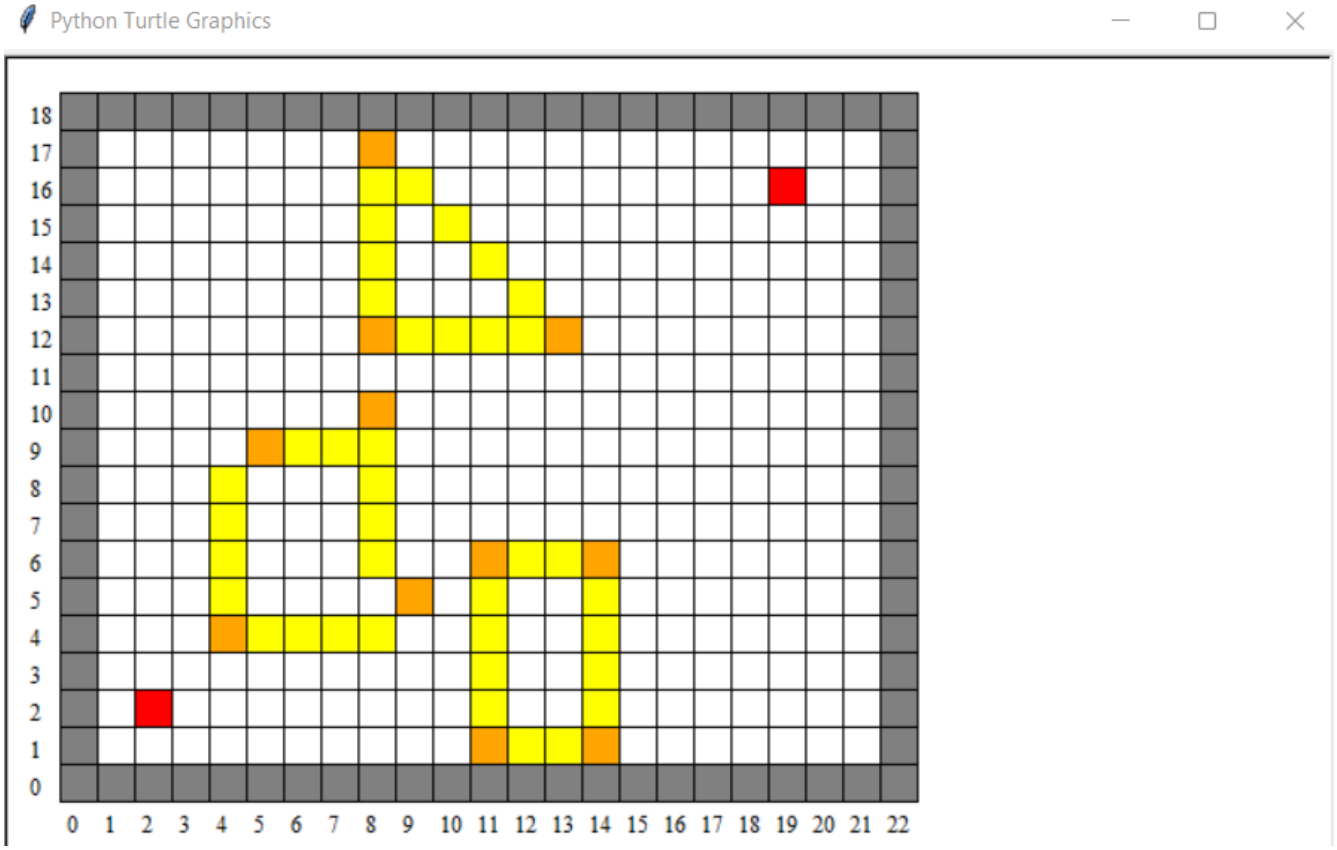
```
def UCS():
    # Stores indices of the matrix cells
    frontier = []
    #Cost of expanding
    global ex_cost
    ex_cost = 0
    # Mark the starting cell as visited
    # and push it into the queue
    frontier.append([sourceX, sourceY])
    vis[sourceX][sourceY] = True

    path[(sourceX, sourceY)] =  sourceX, sourceY
    #Matrix of cost so as to store from the source to current state
    cost = [[ 0 for i in range(width)] for j in range(height)]
    # Iterate while the queue
    # is not empty
    while (frontier):
        min = sys.maxsize
        index = 0
        # Go to the adjacent cells

        if(len(frontier) > 1):
        #find min
            for i in range(len(frontier) - 1):
                temp = frontier[i]
                if(cost[temp[1]][temp[0]] < min):
                    min = cost[temp[1]][temp[0]]
                    index = i
        cell = frontier.pop(index)
        x = cell[0]
        y = cell[1]
        if [x, y] != source and [x, y] != goal: drawPos(x, y, blue)
        if [x, y] == goal:
            break
        for i in range(4):
            adj_y = y + dRow[i]
            adj_x = x + dCol[i]
            if (isValid(adj_y, adj_x) and maze[adj_y][adj_x] == 0):
                if [adj_x, adj_y] != source and [adj_x, adj_y] != goal: drawPos(adj_x, adj_y, palegreen)
                frontier.append([adj_x, adj_y])
                cost[adj_y][adj_x] = cost[y][x] + 1
                ex_cost+=1
                vis[adj_y][adj_x] = True
                path[(adj_x, adj_y)] = x, y
```

In coding, I call an array of frontier instead of a queue for easy usage. Such as BFS, I initialize the frontier by adding source and set vis at that position true. Then I call matrix cost so as to save the distance from the source to the current state. Then starting the loop until the frontier is empty, or the goal is popped from the frontier. While repetition, if the length of the frontier is more than 1, I will find the minimum of cost in the frontier and then get the index of this one so as to pop it out of the frontier. Now I reach the adjacency node and continue adding this one to the frontier, calculate this node cost for the next loop. At this time, vis is set as true and ex_cost summing one by one.

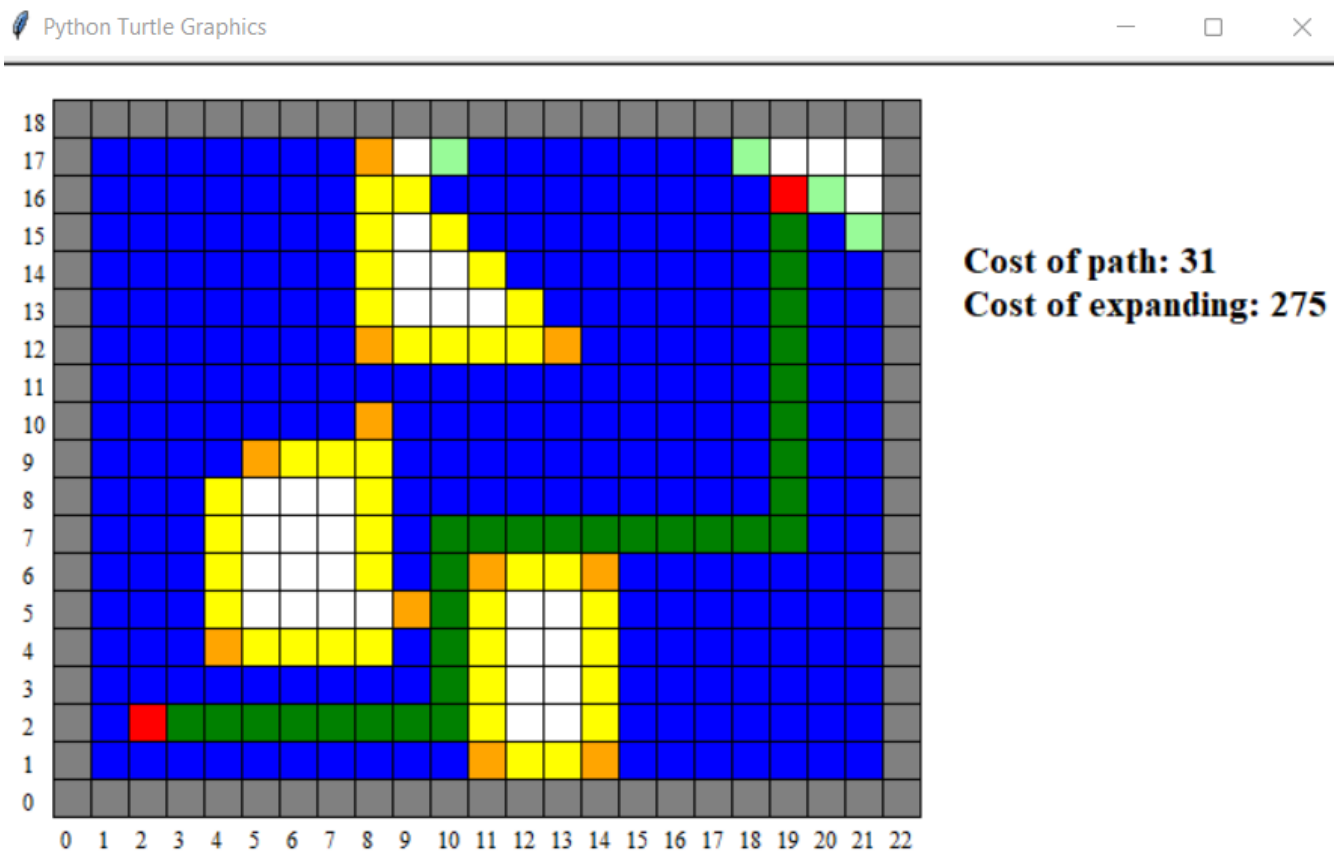
2. Example

Input:

```
22 18
2 2 19 16
3
4 4 5 9 8 10 9 5
8 12 8 17 13 12
11 1 11 6 14 6 14 1
```



Output:



3. Advantages and disadvantages

- Advantages:
  - Uniform-cost search would be optiomal more than BFS as it always choose the shortest one for each time, so it would save more space than BFS.
- Disadvantage
  - Uniform-cost search concentrate on the cost instead of step, so it would be harder for developer to control the path and more cautious for the infinite loop.

IV. Iterative deepening search

1. The idea of the algorithm

Actually, Iterative deepening search is a combination of DFS and BFS. It's mean that it base on DFS, however the way it expand look like BFS as it expand level by level.

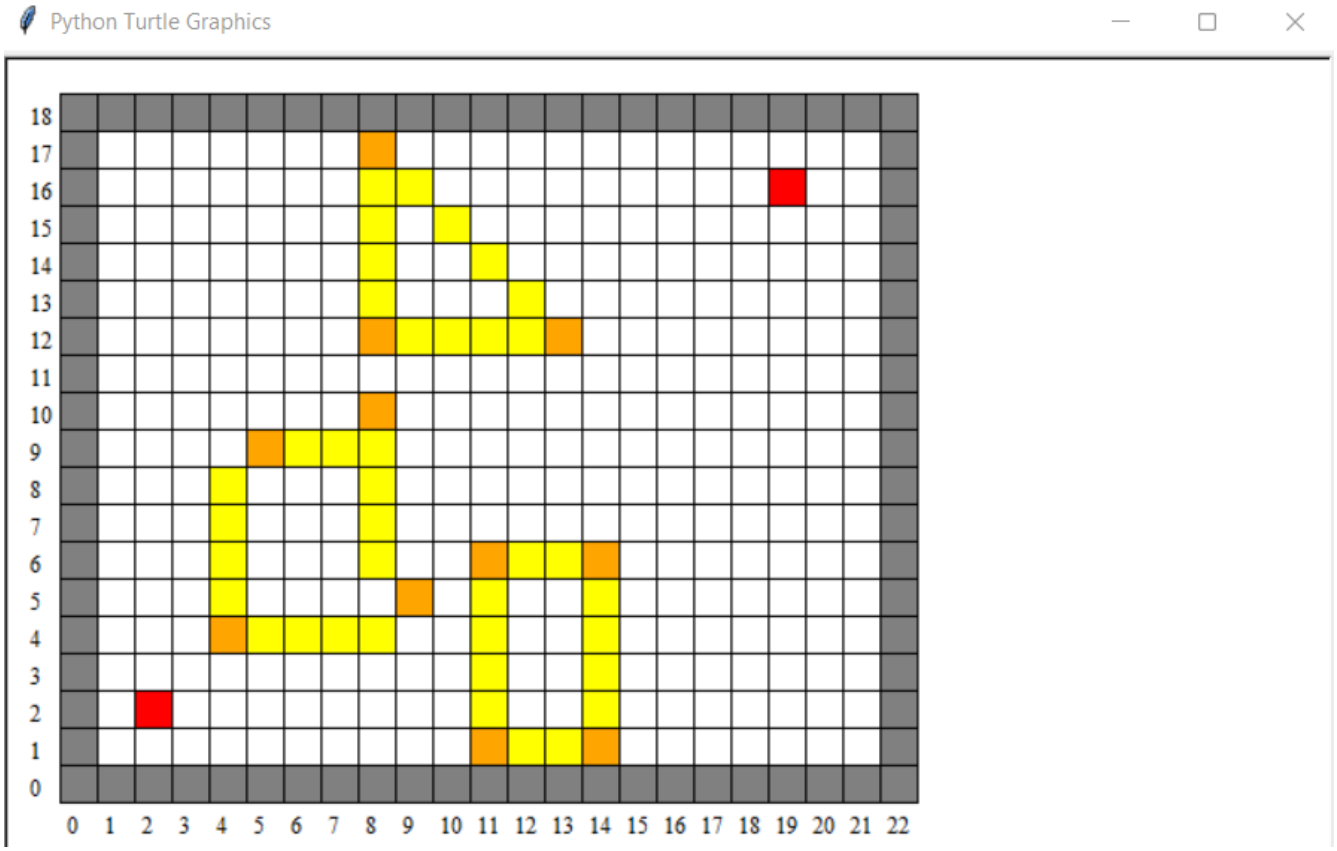
```
def IterativeDS():
    # Stores indices of the matrix cells
    frontier = queue()
    #Cost of expanding
    global ex_cost
    ex_cost = 0
    # Mark the starting cell as visited
    # and push it into the queue
    level = 0
    count = 0
    lev = [[ 0 for i in range(width)] for j in range(height)]
    check = False

    while(check == False):
        frontier.append([sourceX, sourceY])
        lev[sourceY][sourceX] = 0

        for i in range(height):
            for j in range(width):
                vis[i][j] = False
        path.clear()

        vis[sourceY][sourceX] = True
        path[(sourceX, sourceY)] =  sourceX, sourceY
        # Iterate while the queue
        # is not empty
        while(frontier):
            cell = frontier.pop()
            x = cell[0]
            y = cell[1]
            # Go to the adjacent cells
            if [x, y] == goal:
                check = True
                break
            if(lev[y][x] >= level):
                continue
            if [x, y] != source and [x, y] != goal: drawPos(x, y, blue)
            for i in range(4):
                adj_y = y + dRow[i]
                adj_x = x + dCol[i]
                if (isValid(adj_y, adj_x) and maze[adj_y][adj_x] == 0):
                    if [adj_x, adj_y] != source and [adj_x, adj_y] != goal: drawPos(adj_x, adj_y, palegreen)
                    frontier.append([adj_x, adj_y])
                    lev[adj_y][adj_x] = lev[y][x] + 1
                    ex_cost+=1
                    vis[adj_y][adj_x] = True
                    path[(adj_x, adj_y)] = x, y
        level += 1
```

For this algorithm, I call lev matrix so as to store the level of each node from the source and the source get level 0, the variable level for level limit. For the outer most loop, the loop just only stop when the level of all step can reach the goal and I use variable check to control this loop, for each loops, the path of this algorithm reset and so does the vis matrix. In the outer loop (The loop inside the outer loop), it is controlled by the frontier and only stop if the frontier is empty, we also check that if current lev equal to level, the loop will not reach the adjacency one. For each step of one node, I store the value of adjacency node equal to the current node plus one, the expanding cost plus one, vis at that position will set as true and add the coordinate into the path dictionary. After all, the level plus one and start back to the outer most loop.

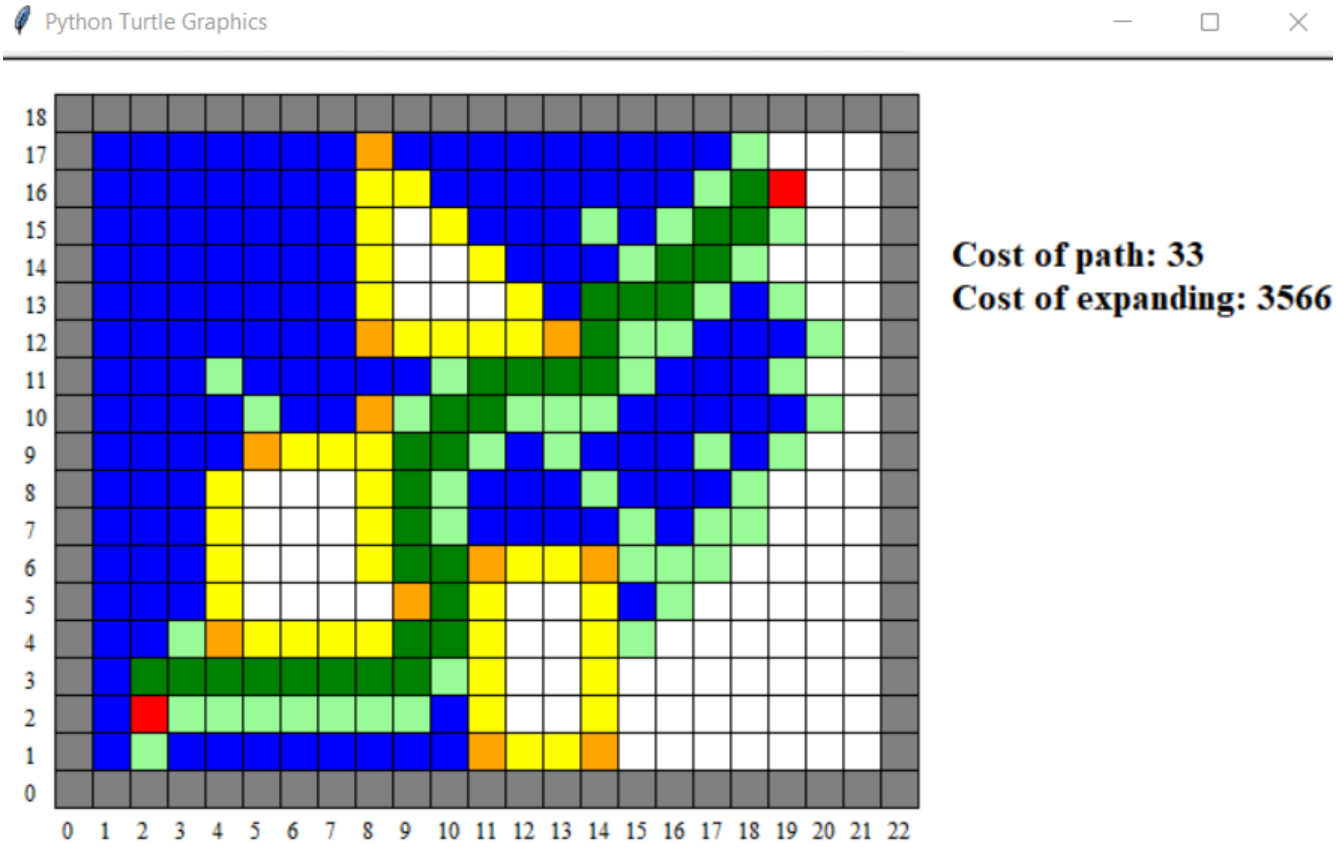
2. Example

Input:

```
22 18
2 2 19 16
3
4 4 5 9 8 10 9 5
8 12 8 17 13 12
11 1 11 6 14 6 14 1
```



Output:



3. Advantages and disadvantages

- Advantages:
  - Iterative deepening search combines the advantages of the BFS and DFS search algorithms.
  - When the goal is at the low level, it would be really efficient and also save space than other search algorithm
- Disadvantage
  - Iterative deepening search is not efficient when the goal is at high level because states are visited many time until the level reach the goal, cause space and time waste

V. Greedy-best first search

1. The idea of the algorithm

Greedy best-first search algorithm is depend mainly on BFS.

It always finds the shortest path from the source to the goal by optimizing (choosing the minimum heuristic from current node to the goal)

The heuristic will be calculated using following formula:

h= abs(i - goalY) + abs(j - goalX)

i: the row of current node

j: the column of current node
goalY: the row of goal
goalX: the column of goal

```
def GBFS():
    # Stores indices of the matrix cells
    frontier = []
    #Cost of expanding
    global ex_cost
    ex_cost = 0
    # Mark the starting cell as visited
    # and push it into the queue
    frontier.append([sourceX, sourceY])
    vis[sourceX][sourceY] = True
    cost = [[ 0 for i in range(width)] for j in range(height)]

    for i in range(height):
        for j in range(width):
            cost[i][j] = abs(i - goalY) + abs(j - goalX)

    path[(sourceX, sourceY)] =  sourceX, sourceY
    # Iterate while the queue
    # is not empty
    while (frontier):
        index = 0
        min = sys.maxsize
        if(len(frontier) > 1):
        #find min
            for i in range(len(frontier) - 1):
                temp = frontier[i]
                if(cost[temp[1]][temp[0]] < min):
                    min = cost[temp[1]][temp[0]]
                    index = i
        cell = frontier.pop(index)
        x = cell[0]
        y = cell[1]
        if [x, y] != source and [x, y] != goal: drawPos(x, y, blue)
        # Go to the adjacent cells
        for i in range(4):
            adj_y = y + dRow[i]
            adj_x = x + dCol[i]
            if (isValid(adj_y, adj_x) and maze[adj_y][adj_x] == 0):
                if [adj_x, adj_y] != source and [adj_x, adj_y] != goal: drawPos(adj_x, adj_y, palegreen)
                vis[adj_y][adj_x] = True
                ex_cost+=cost[adj_y][adj_x]
                frontier.append([adj_x, adj_y])
                path[(adj_x, adj_y)] = x, y
        if [adj_x, adj_y] == goal:
            break
```

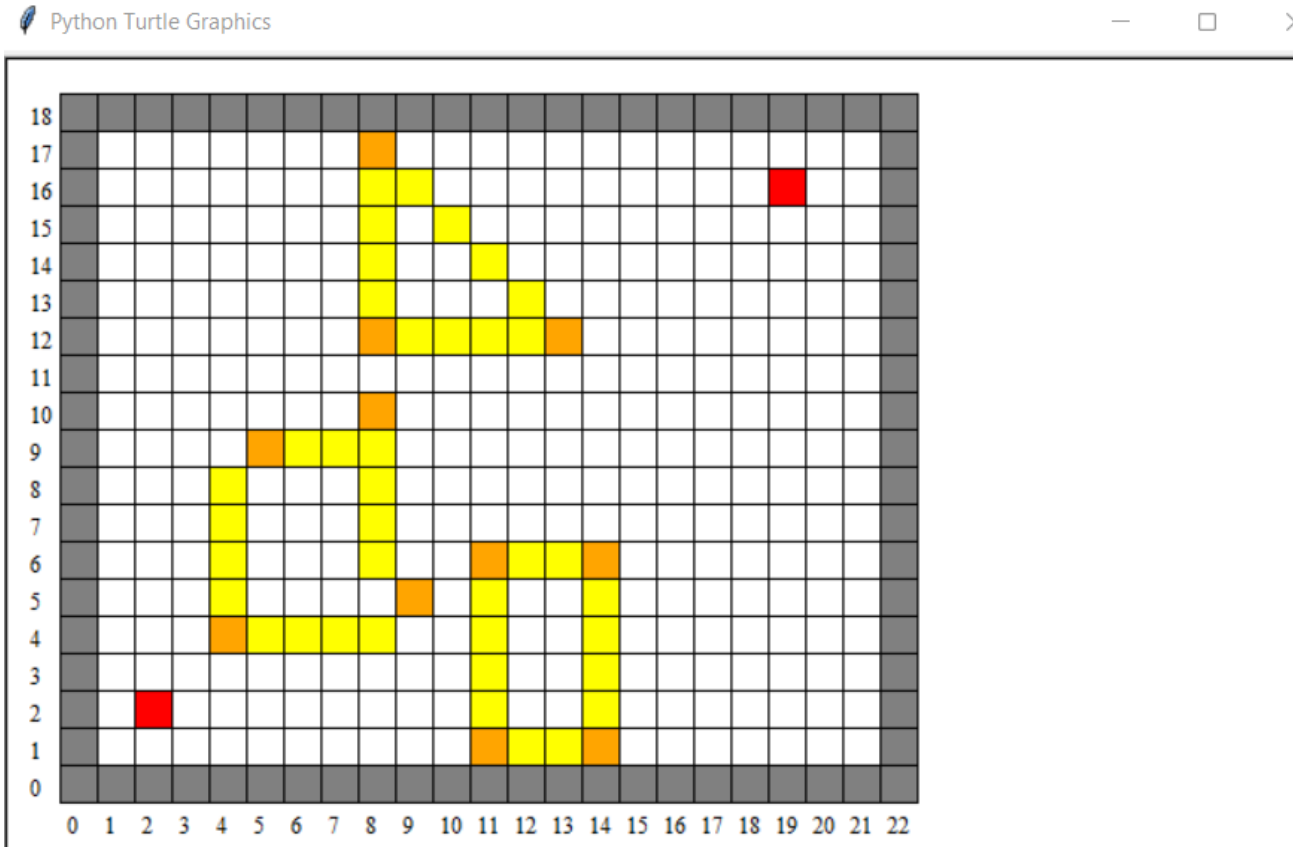
In this code, frontier will be an array that contains the coordinate of adjacency nodes, vis will be used to check whether the node is visited or not. The cost will be a matrix that contain the heuristic from the current node to the goal.

We will use path dictionary for backtracking to find the path from the source to the goal. We will then use a loop to iteratively take out element from frontier and this loop will end when there is no element left. We will find the minium cost of element and take it out. We will then use that element to figure out further node and we will use isValid() to check for validation, the vis at that coordinate will then be set true and the cost, path will be updated accordingly. The adjacency node will then be added into frontier. The algorithm will stop when the adjacency node meets the goal.

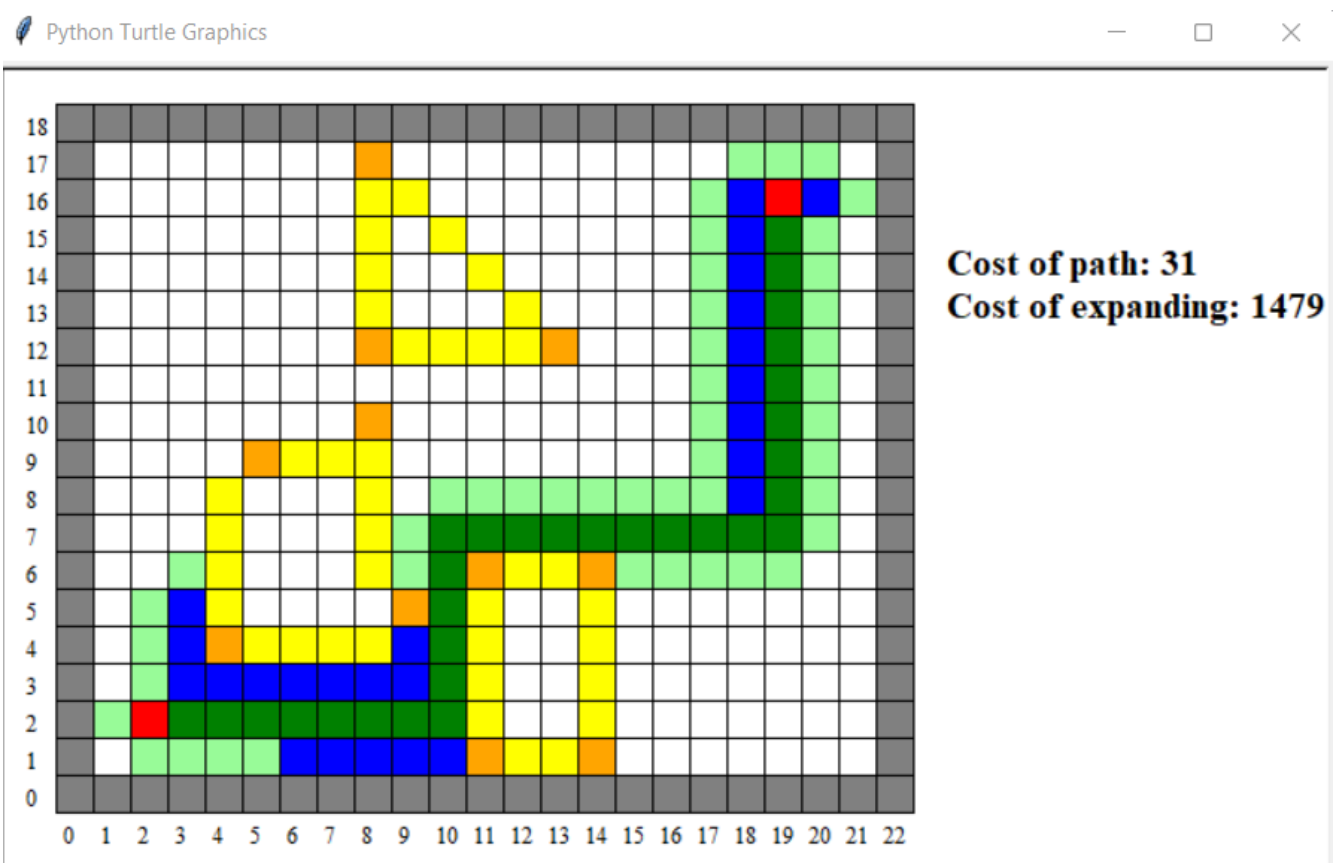
2. Example

Input:

```
22 18
2 2 19 16
3
4 4 5 9 8 10 9 5
8 12 8 17 13 12
11 1 11 6 14 6 14 1
```



Output:



3. Advantages and disadvantages

- Advantages:
  - Greedy best-first search always find the shortest path.
  - It's more efficient in space and time more than BFS and DFS algorithms.
- Disadvantage
  - Greedy best-first search need another matrix so as to contain the value of heuristic so it may not efficient in space.

VI. Graph-search A*

1. The idea of the algorithm

Graph-search A* search algorithm is depend mainly on BFS.

It always finds the shortest path from the source to the goal by optimizing (choosing the minimum of sum of heuristic and the actual cost from current node to the goal)

The heuristic will be calculated using following formula:

cost= cost_f[y][x] + cost_h[y][x]

cost_f is the matrix contains actual cost from current node to the source

cost_h is the matrix contains heuristic from current node to the goal

```
def GSA():
    frontier = []
    #Cost of expanding
    global ex_cost
    ex_cost = 0
    cost_h = [[ 0 for i in range(width)] for j in range(height)]
    cost_f = [[ 0 for i in range(width)] for j in range(height)]

    cost_f[sourceY][sourceX] = 0
    frontier.append(source)

    for i in range(height):
        for j in range(width):
            cost_h[i][j] = abs(i - goalY) + abs(j - goalX)

    path[(sourceX, sourceY)] =  sourceX, sourceY

    while(frontier):
        index = 0
        min = sys.maxsize
        if(len(frontier) > 1):
        #find min
            for i in range(len(frontier) - 1):
                temp = frontier[i]
                if(cost_f[temp[1]][temp[0]] + cost_h[temp[1]][temp[0]] < min):
                    min = cost_f[temp[1]][temp[0]] + cost_h[temp[1]][temp[0]]
                    index = i
        cell = frontier.pop(index)
        x = cell[0]
        y = cell[1]
        if [x, y] != source and [x, y] != goal: drawPos(x, y, blue)
        if ([x, y] == goal):
            break
        for i in range(4):
            adj_y = y + dRow[i]
            adj_x = x + dCol[i]
            if (isValid(adj_y, adj_x) and maze[adj_y][adj_x] == 0):
                if [adj_x, adj_y] != source and [adj_x, adj_y] != goal: drawPos(adj_x, adj_y, palegreen)
                vis[adj_y][adj_x] = True
                frontier.append([adj_x, adj_y])
                cost_f[adj_y][adj_x] = cost_f[y][x] + 1
                ex_cost = ex_cost + cost_f[adj_y][adj_x] + cost_h[adj_y][adj_x]
                path[(adj_x, adj_y)] = x, y
```

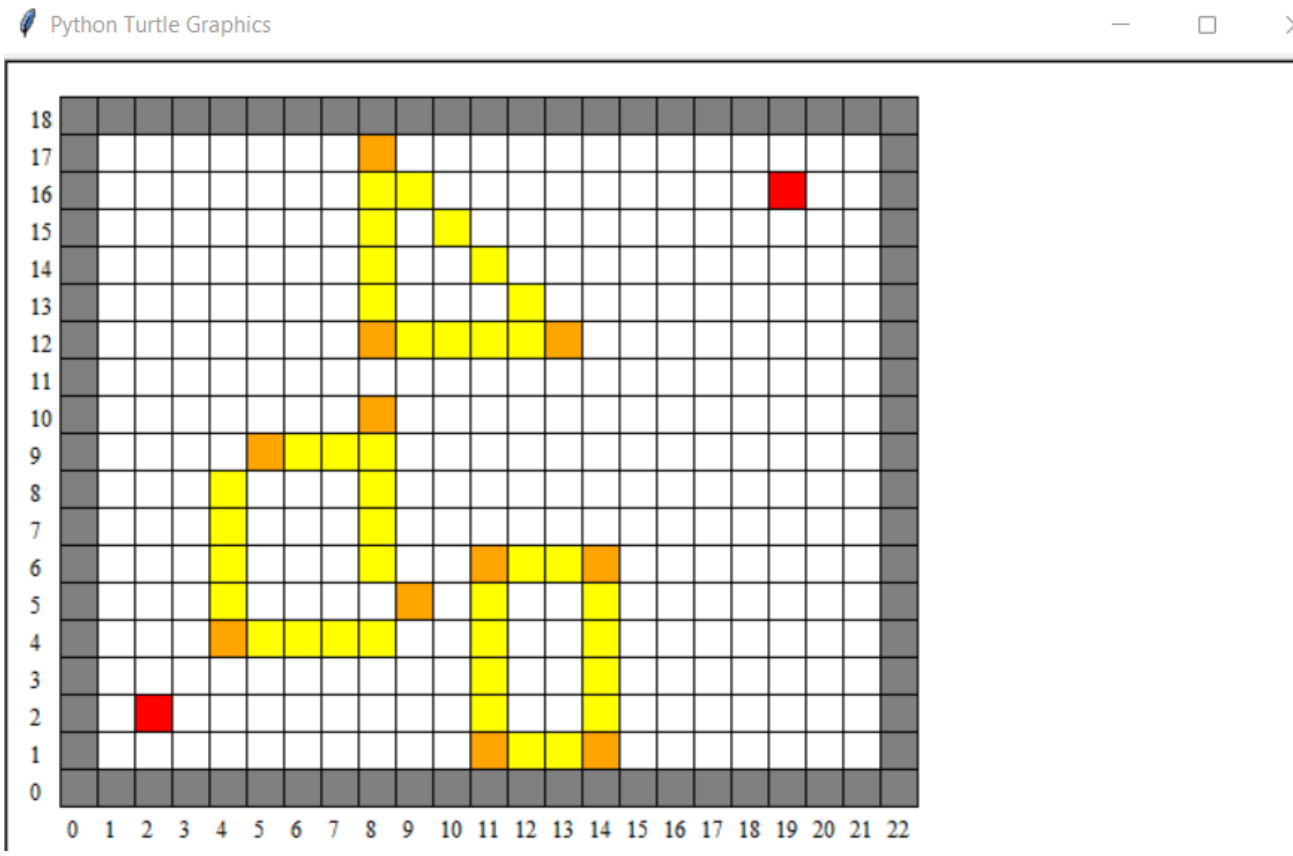
In this code, frontier will be an array that contains the coordinate of adjacency nodes, vis will be used to check whether the node is visited or not. The cost_h will be a matrix that contain the heuristic from the current node to the goal and cost_f will be a matrix that contain the actual cost from the current node to the source.

We will use path dictionary for backtracking to find the path from the source to the goal. We will then use a loop to iteratively take out element from frontier and this loop will end when there is no element left. We will find the minium sum of cost_f and cost_h of element and take it out. We will then use that element to figure out further node and we will use isValid() to check for validation, the vis at that coordinate will then be set true and the cost, path will be updated accordingly. The adjacency node will then be added into frontier. The algorithm will stop when the current node meets the goal.

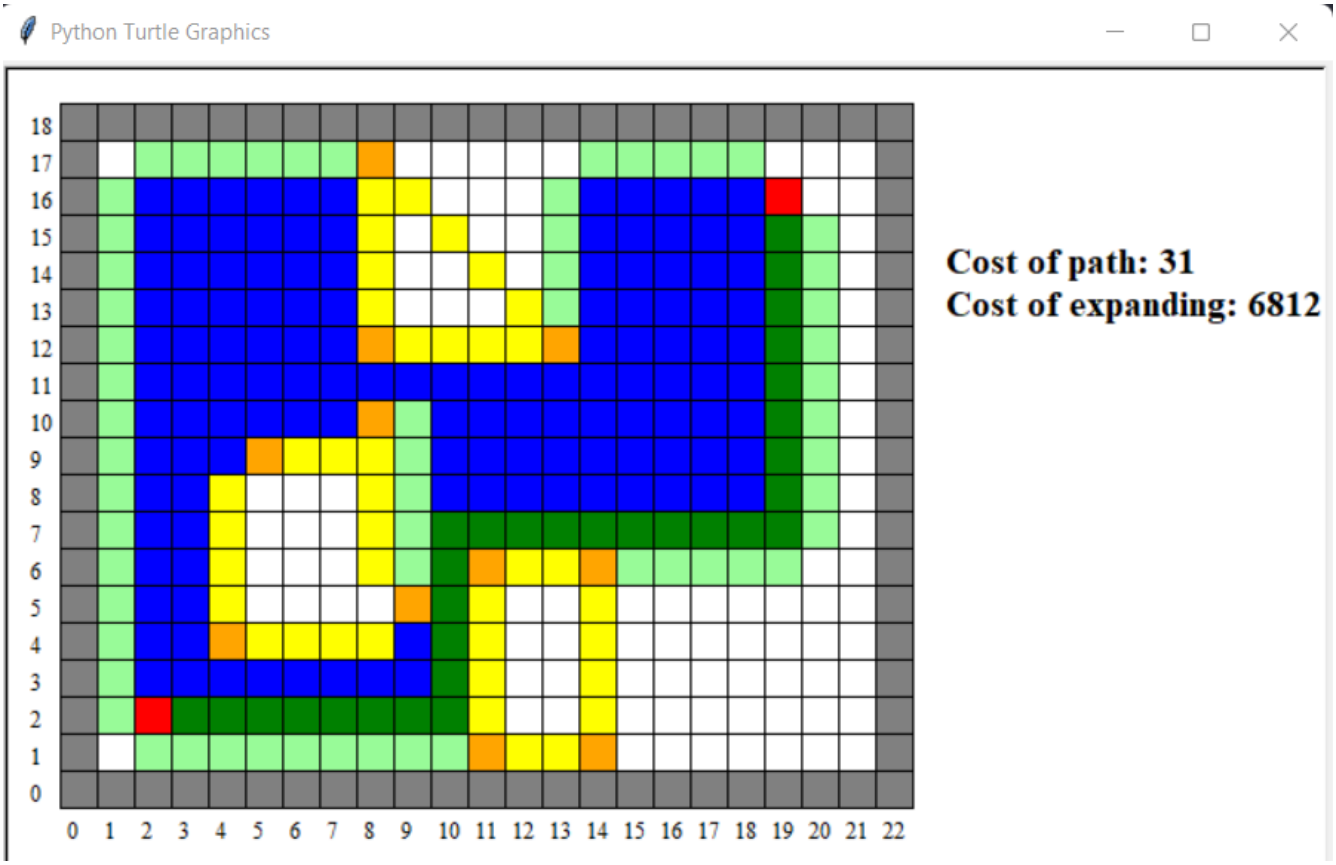
2. Example

Input:

```
22 18
2 2 19 16
3
4 4 5 9 8 10 9 5
8 12 8 17 13 12
11 1 11 6 14 6 14 1
```



Output:



3. Advantages and disadvantages

- Advantages:
  - Graph-search A* always find the shortest path.
  - It's more efficient in space and time more than BFS and DFS algorithms.
- Disadvantage
  - Greedy best-first search need two other matrixes that one contains the value of heuristic from the current node to goal and one contains the value of actual cost from source to current node so it may not efficient in space.