CIS 550 Introduction to Algorithms

Spring Semester, 2022

Programming Assignment 3

Submitted By:

Mohamed Gani Mohamed Sulthan

2811619

Master’s in Computer Science

Question (10points). Using DFS and BFS for solving a maze problem. A robot needs to find a path given a start position S and an end position E.

Descriptions:

1. The maze can be represented using an 2d array,

e.g., maze = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0],
[0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0],
[0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1],
[0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0],
[0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0],
[0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
[0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
[0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0],
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

0 means the wall, 1 means an empty black can be visited by the robot.

2. In the main.py file, you are required to implement BFS and DFS functions. Please see the requirement for input and output.

3. We will check the results for the given start and end positions by looking at the output.

Implementing BFS approach:

Algorithm:

1. Get a 2D array as maze and starting point and ending point to find the dfs
2. Create an adjacent cell x and y to keep track of the direction
3. Create a Maze position class to keep track the blocks of the maze
4. Getting the length of the maze as m, n
5. Created a visited blocks and make it non visited for all the m and n
6. Assigning start position of visited_blocks as true
7. Created queue to keep track of the visited nodes while exploring other nodes
8. Start from any Node (that is called as root node)
9. Visit that Node and push into Queue
10. Explore all adjacent of visited Node in any order and push all into Queue

11. Once all adjacent pushed into Queue then pick a new vertex
12. This new vertex will be the first non explored element from the queue

Steps to do:

1. Create an adjacent cell x and y to keep track of the direction
   adjacent_cell_x = [1, 0, 0, -1]
   adjacent_cell_y = [0, 1, -1, 0]

2. To keep track of the maze position from start to end
   destination = Maze_Position(end_x, end_y)
   start = Maze_Position(start_x, start_y)

3. Getting the length of the 2D array
   m, n = (len(maze), len(maze))

4. To keep track of the blocks of maze
```
class Maze_Position:
  def __init__(self, x, y):
    self.x = x
    self.y = y
```

5. Each block will have its own position and cost of steps taken
```
class Node:
  def __init__(self, position: Maze_Position, cost):
    self.position = position
    self.cost = cost
```

6. Making all as a non visited blocks
```
visited_blocks = [[False for i in range(m)]
        for j in range(n)]
```

7. Assigning start position of visited_blocks as true
   visited_blocks[start.x][start.y] = True

8. Created queue to keep track of the visited nodes while exploring other nodes
```
while queue:
    current_block = queue.popleft()  # Dequeue the front cell
    current_position = current_block.position
    if current_position.x == destination.x and current_position.y == destination.y:
```

Code:

```
def bfs(maze, start_x, start_y, end_x, end_y):
    #return -1 if path does not exists
    # otherwise return cost the shortest path

    # To get neighbours of current node
    adjacent_cell_x = [1, 0, 0, -1]
    adjacent_cell_y = [0, 1, -1, 0]

    # To keep track of the maze position from start to end
    destination = Maze_Position(end_x, end_y)
    start = Maze_Position(start_x, start_y)

    # Getting the length of the 2D array
    r, c = (len(maze), len(maze))

    # Making all as a non visited blocks
    visited_blocks = [[False for i in range(m)]
                for j in range(n)]

    # Assigning start positionition of visited_blocks as true
    visited_blocks[start.x][start.y] = True

    # Created queue to keep track of the visited nodes while exploring other nodes
    queue = deque()
    sol = Node(start, 0)
    queue.append(sol)
    cells = 4
    cost = 0
```

```
    # Visit the Node and push into Queue
    while queue:
        current_block = queue.popleft()  # Dequeue the front cell
        current_position = current_block.position

        # If the current position matches the destination, it will print the cost taken to reach the destination
        if current_position.x == destination.x and current_position.y == destination.y:
            print("BFS")
            return current_block.cost

        if current_block not in visited_blocks:
            visited_blocks[current_position.x][current_position.y] = True
            cost = cost + 1
        x_position = current_position.x
        y_position = current_position.y
            # By using the cells count 4, It will iterate the adjacents and found the x_position and y_position
position in the maze
        for i in range(cells):
            if x_position == len(maze) - 1 and adjacent_cell_x[i] == 1:
                x_position = current_position.x
                y_position = current_position.y + adjacent_cell_y[i]
            if y_position == 0 and adjacent_cell_y[i] == -1:
                x_position = current_position.x + adjacent_cell_x[i]
                y_position = current_position.y
            else:
                x_position = current_position.x + adjacent_cell_x[i]
                y_position = current_position.y + adjacent_cell_y[i]
```

```
            # The written position are matched here and if it's equal 1 it will make as the visited block true
and increment the cost
          if x_position < 12 and y_position < 12 and x_position >= 0 and y_position >= 0:
            if maze[x_position][y_position] == 1:
              if not visited_blocks[x_position][y_position]:
                next_cell = Node(Maze_Position(x_position, y_position),
                        current_block.cost + 1)
                visited_blocks[x_position][y_position] = True
                queue.append(next_cell)
  return -1
```

Output:

BFS

Shortest cost = 18

Implementing DFS approach:

Algorithm:

1. Get a 2D array as maze and starting point and ending point to find the dfs
2. Create an adjacent cell x and y to keep track of the direction
3. Create a Maze position class to keep track the blocks of the maze
4. Getting the length of the maze as m, n
5. Created a visited blocks and make it non visited for all the m and n
6. Assigning start position of visited_blocks as true
7. Created stack to keep track of the visited nodes while exploring other nodes
8. Visit that Node and push into Stack
9. Explore its adjacent
10. Visit any one of its adjacent and push into Stack
11. Now explore that adjacent
12. Repeat 4 and 5
13. Once there is no adjacent remaining then backtrack to the parent
14. Visit another adjacent of that parent and push into stack
15. If still no any parent's adjacent then backtrack to its parent.

16. Do this until all vertex traversed.

Steps to do:

1. Create an adjacent cell x and y to keep track of the direction
   adjacent_cell_x = [1, 0, 0, -1]
   adjacent_cell_y = [0, 1, -1, 0]

2. To keep track of the maze position from start to end
   destination = Maze_Position(end_x, end_y)
   start = Maze_Position(start_x, start_y)

3. Getting the length of the 2D array
   r, c = (len(maze), len(maze))

4. To keep track of the blocks of maze
```
class Maze_Position:
  def __init__(self, x, y):
    self.x = x
    self.y = y
```

5. Each block will have its own position and cost of steps taken
```
class Node:
  def __init__(self, position: Maze_Position, cost):
    self.position = position
    self.cost = cost


def create_node(x, y, c):
  val = Maze_Position(x, y)
  return Node(val, c + 1)
```

6. Making all as a non visited blocks
```
visited_blocks = [[False for i in range(m)]
        for j in range(n)]
```

Code:

```
def dfs(maze, start_x, start_y, end_x, end_y):
    # return -1 if path does not exists
    # otherwise return steps with backtracking

    # To get neighbours of current node
    adjacent_cell_x = [1, 0, 0, -1]
    adjacent_cell_y = [0, 1, -1, 0]

    # to keep track of the maze position from start to end
    destination = Maze_Position(end_x, end_y)
    start = Maze_Position(start_x, start_y)

    # Getting the length of the 2D array
    r, c = (len(maze), len(maze))

    # Making all as a non visited blocks
    visited_blocks = [[False for i in range(m)]
            for j in range(n)]
    visited_blocks[start.x][start.y] = True

    # Created stack to keep track of the visited nodes while exploring other nodes
    stack = deque()
    sol = Node(start, 0)
    stack.append(sol)
    neigh = 4
    neighbours = []
    cost = 0
```

```
    # Visit that Node and push into Stack
    while stack:
        current_block = stack.pop()
        current_position = current_block.position
        if current_position.x == destination.x and current_position.y == destination.y:
            print("DFS")
            return current_block.cost
        x_position = current_position.x
        y_position = current_position.y

        # By using the cells count 4, It will iterate the adjacents and found the x_position and y_position
positions in the maze
        for i in range(neigh):
            if x_position == len(maze) - 1 and adjacent_cell_x[i] == 1:
                x_position = current_position.x
                y_position = current_position.y + adjacent_cell_y[i]
            if y_position == 0 and adjacent_cell_y[i] == -1:
                x_position = current_position.x + adjacent_cell_x[i]
                y_position = current_position.y
            else:
                x_position = current_position.x + adjacent_cell_x[i]
                y_position = current_position.y + adjacent_cell_y[i]

            # The written positions are matched here and if it's equal 1 it will make as the visited block true
            and increment the cost
            if x_position != 12 and x_position != -1 and y_position != 12 and y_position != -1:
                if maze[x_position][y_position] == 1:
                    if not visited_blocks[x_position][y_position]:
                        cost += 1
```

```
            visited_blocks[x_position][y_position] = True
            stack.append(create_node(x_position, y_position, current_block.cost))
    return -1
```

Output:

DFS

Shortest cost = 28