Lab Report: 09

Experiment Title: Implementation of Greedy Best First Search (GBFS) for Maze Pathfinding

Objectives:

The main objectives of this experiment are to implement the Greedy Best First Search (GBFS) algorithm for finding a path in a maze, to understand how heuristic-based search works, and to analyze the efficiency of GBFS in reaching the goal.

Theoretical Background:

Greedy Best First Search (GBFS) is an informed search algorithm that uses a heuristic to estimate the cost from a current node to the goal. It always expands the node that appears to be closest to the goal according to the heuristic function. Unlike algorithms like A*, GBFS does not consider the cost to reach the current node; it only considers the estimated cost to the goal.

Key Concepts:

Term	Meaning
Node	A position in the maze.
Open List	Modes available for expansion.
Closed List	Nodes already visited.
Heuristic	Estimated distance from the current node to the goal
Goal	The target node the search is trying to reach.

Algorithm:

1. Initialize the **open list** with the start node and an empty **closed list**.

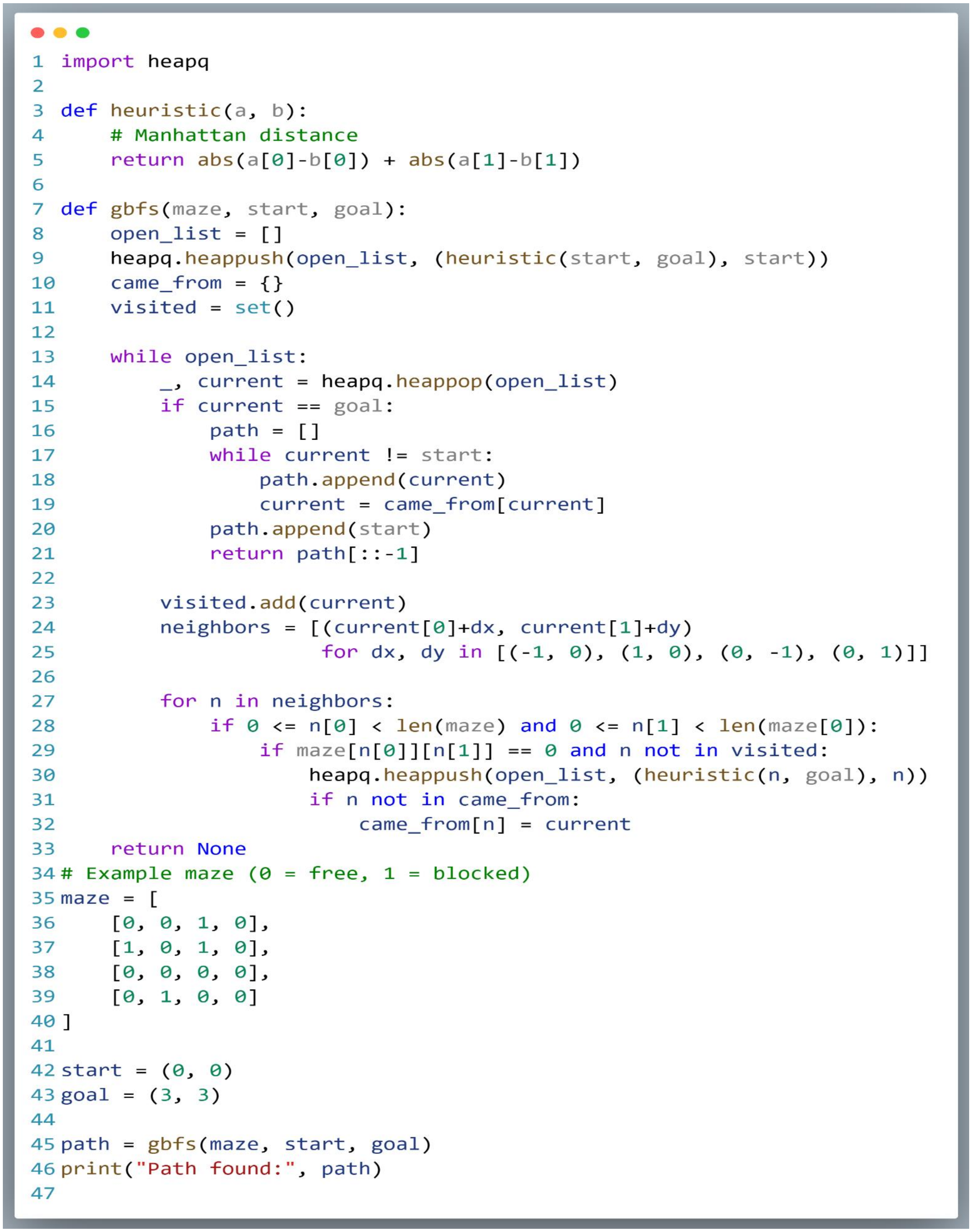
2. While the open list is not empty:

   i) Select the node with the lowest heuristic value from the open list.
   ii) If the node is the goal, return the path.
   iii) Otherwise, move the node to the closed list and expand its neighbors.
   iv) Add unvisited neighbors to the open list with their heuristic values.

3. Repeat until the goal is reached or the open list is empty.

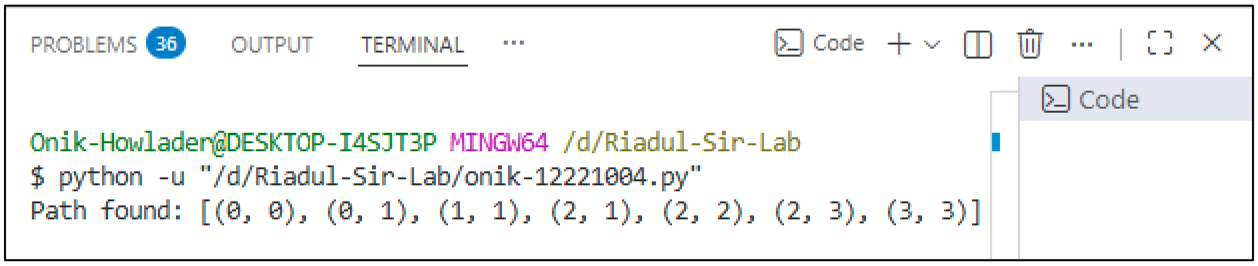
4. If the goal is unreachable, report failure.

Source Code:



```python
import heapq

def heuristic(a, b):
    # Manhattan distance
    return abs(a[0]-b[0]) + abs(a[1]-b[1])

def gbfs(maze, start, goal):
    open_list = []
    heapq.heappush(open_list, (heuristic(start, goal), start))
    came_from = {}
    visited = set()

    while open_list:
        _, current = heapq.heappop(open_list)
        if current == goal:
            path = []
            while current != start:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]

        visited.add(current)
        neighbors = [(current[0]+dx, current[1]+dy)
                     for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]]

        for n in neighbors:
            if 0 <= n[0] < len(maze) and 0 <= n[1] < len(maze[0]):
                if maze[n[0]][n[1]] == 0 and n not in visited:
                    heapq.heappush(open_list, (heuristic(n, goal), n))
                    if n not in came_from:
                        came_from[n] = current
    return None
# Example maze (0 = free, 1 = blocked)
maze = [
    [0, 0, 1, 0],
    [1, 0, 1, 0],
    [0, 0, 0, 0],
    [0, 1, 0, 0]
]

start = (0, 0)
goal = (3, 3)

path = gbfs(maze, start, goal)
print("Path found:", path)

```

<u>Screenshot Of Execution</u>:



```
Onik-Howlader@DESKTOP-I4SJT3P MINGW64 /d/Riadul-Sir-Lab
$ python -u "/d/Riadul-Sir-Lab/onik-12221004.py"
Path found: [(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (2, 3), (3, 3)]
```

<u>Observation and Result:</u>

<u>Observation:</u>

The GBFS algorithm successfully finds a path from the start to the goal using the heuristic to prioritize nodes closer to the goal. The algorithm explores fewer nodes than uninformed search strategies, but the path found may not always be the shortest. Proper management of the open and closed lists prevents revisiting nodes and ensures efficient search.

<u>Result:</u>

The implementation demonstrates that Greedy Best First Search can effectively navigate a maze by using a heuristic to guide exploration. The network quickly reaches the goal, producing a valid path while minimizing unnecessary exploration. This shows the efficiency and limitations of heuristic-based search algorithms.

Lab Report: 10

Experiment Title: Implementation of A* Search Algorithm for Maze Pathfinding.

Objectives:

The objectives of this experiment are to implement the A* Search algorithm to find the shortest path in a maze, to understand how heuristic and path cost combine in A*, and to evaluate its efficiency compared to uninformed search methods.

Theoretical Background:

The A* Search algorithm is an informed search strategy that finds the shortest path from a start node to a goal node by combining the actual cost from the start (g) and the estimated cost to the goal (h) using a heuristic function.

Key Concepts:

Term	Meaning
Node	A position in the maze.
g(n)	Cost from the start node to current node n.
h(n)	Estimated cost from node n to the goal(heuristic).
f(n)	Total estimated cost, f(n) = g(n) + h(n).
Open List	Nodes available for expansion.
Closed List	Nodes already visited.
Goal	Target node to reach.

Algorithm:

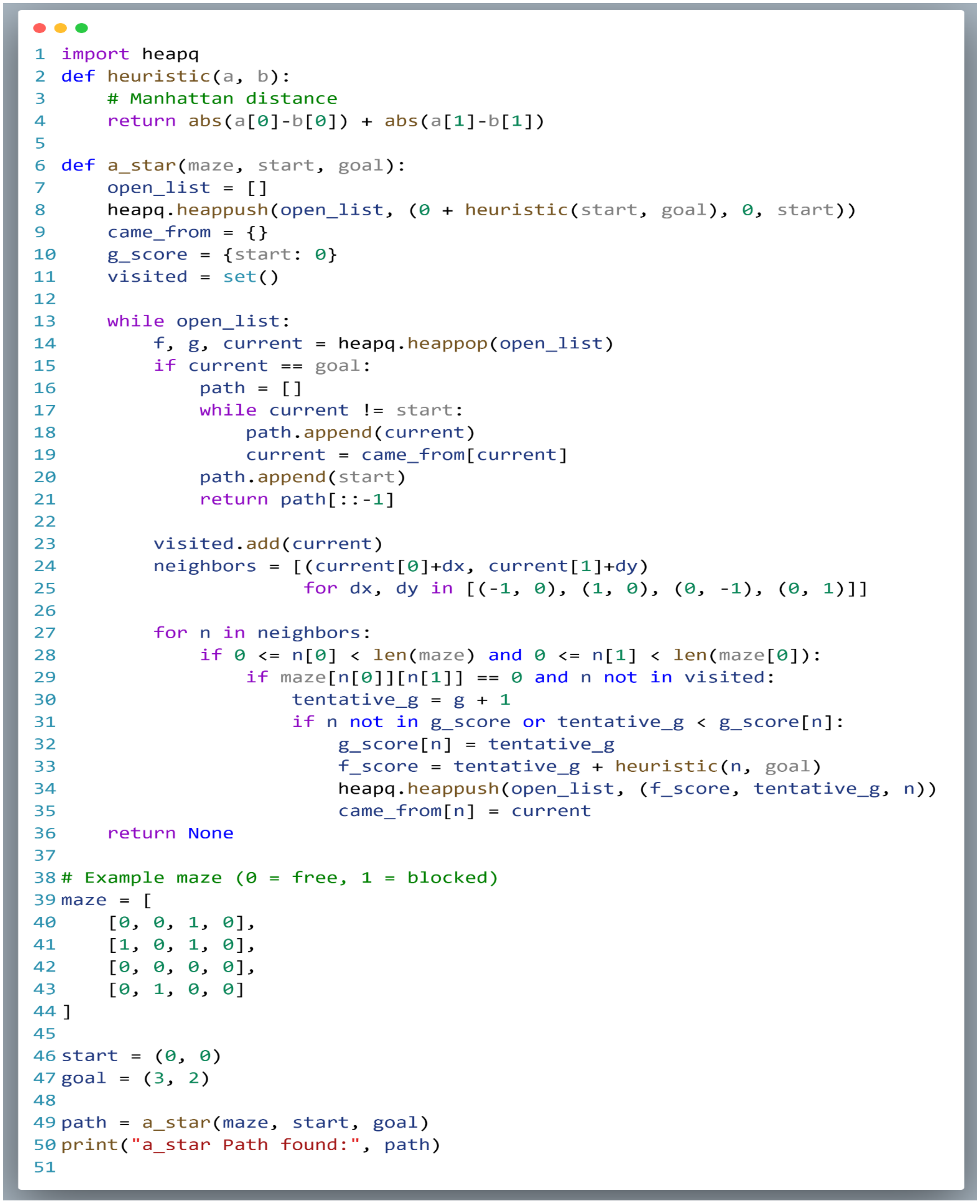
1. Initialize the **open list** with the start node and an empty **closed list**.
2. While the open list is not empty:

    i) Select the node with the lowest f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n) from the open list.
    ii) If the node is the goal, reconstruct and return the path.
    iii) Otherwise, move the node to the closed list and expand its neighbors.
    iv) For each neighbor, calculate ggg, hhh, and fff.
    v) Add unvisited neighbors or neighbors with lower f values to the open list.

3. Repeat until the goal is reached or the open list is empty.

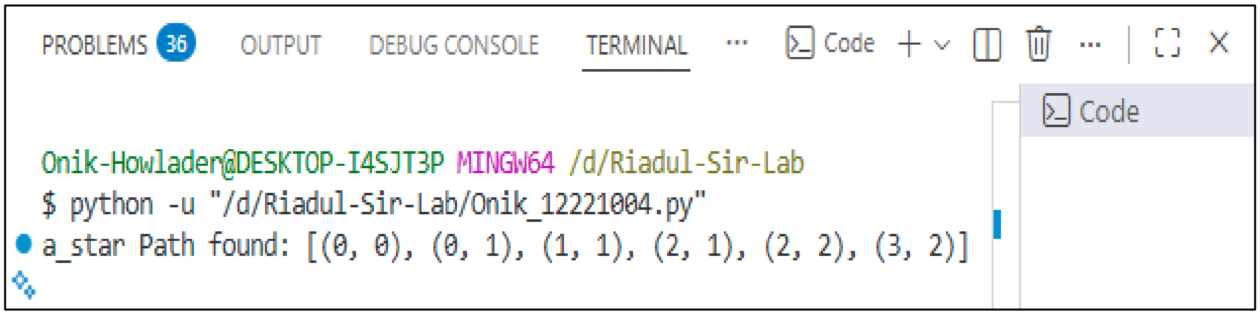
4. If no path exists, report failure.

Source Code:



```python
import heapq
def heuristic(a, b):
    # Manhattan distance
    return abs(a[0]-b[0]) + abs(a[1]-b[1])

def a_star(maze, start, goal):
    open_list = []
    heapq.heappush(open_list, (0 + heuristic(start, goal), 0, start))
    came_from = {}
    g_score = {start: 0}
    visited = set()

    while open_list:
        f, g, current = heapq.heappop(open_list)
        if current == goal:
            path = []
            while current != start:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]

        visited.add(current)
        neighbors = [(current[0]+dx, current[1]+dy)
                     for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]]

        for n in neighbors:
            if 0 <= n[0] < len(maze) and 0 <= n[1] < len(maze[0]):
                if maze[n[0]][n[1]] == 0 and n not in visited:
                    tentative_g = g + 1
                    if n not in g_score or tentative_g < g_score[n]:
                        g_score[n] = tentative_g
                        f_score = tentative_g + heuristic(n, goal)
                        heapq.heappush(open_list, (f_score, tentative_g, n))
                        came_from[n] = current
    return None

# Example maze (0 = free, 1 = blocked)
maze = [
    [0, 0, 1, 0],
    [1, 0, 1, 0],
    [0, 0, 0, 0],
    [0, 1, 0, 0]
]

start = (0, 0)
goal = (3, 2)

path = a_star(maze, start, goal)
print("a_star Path found:", path)

```

<u>Screenshot Of Execution:</u>



```
Onik-Howlader@DESKTOP-I4SJT3P MINGW64 /d/Riadul-Sir-Lab
$ python -u "/d/Riadul-Sir-Lab/Onik_12221004.py"
a_star Path found: [(0, 0), (0, 1), (1, 1), (2, 1), (2, 2), (3, 2)]
```

<u>Observation and Result:</u>

<u>Observation:</u>

The A* algorithm successfully finds the shortest path from the start to the goal by combining the cost to reach a node and the estimated cost to the goal. The heuristic guides the search efficiently, and the algorithm avoids unnecessary exploration of nodes already visited. The path returned is optimal given the admissible heuristic.

<u>Result:</u>

The implementation demonstrates that A* Search can reliably find the shortest path in a maze. By using both the actual path cost and the heuristic estimate, the algorithm efficiently navigates the maze and produces an optimal solution, showing the advantages of informed search strategies for pathfinding problems.