Afia Tasnim Ria
0112231058
AI Lab

Approach:

Firstly used A* search to find the shortest path.
Manhattan distance to find the huristic.
Checked the path is blocked or not and then calculating the huristic and shortest path took decision to choose the path
During execution, the algorithm calculates both the actual cost (g_score) and the heuristic value to determine the estimated total cost (f_score) for each node. Based on these values, it dynamically chooses the next path that minimizes the overall cost, ensuring an optimal and efficient route to the goal.

Code explanation:

```
import heapq

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

# A* search function
def a_star_search(maze, start, end):
    # Check if the start or end position is invalid
    if maze[start[0]][start[1]] != 0 or maze[end[0]][end[1]] != 0:
        return "No path found"

    rows, cols = len(maze), len(maze[0])
    open_set = []  # Priority queue to store nodes to be explored.
    heapq.heappush(open_set, (0, start))
    came_from = {}  # Dictionary to reconstruct the path after finding the end.
    g_score = {start: 0}  # Dictionary to track the cost of reaching each node from the start.
    f_score = {start: heuristic(start, end)}  # Estimated total cost


    cost = {start: heuristic(start, end)}

    while open_set:
```

```
    current = heapq.heappop(open_set)[1]  # Get the node with the lowest f_score from the queue.

    if current == end:  # If the end node is reached, reconstruct the path.
        path = []  # List to store the path from start to end.
        while current in came_from:  # Trace back from the end to the start.
            path.append(current)
            current = came_from[current]  # Move to the previous node in the path.
        path.append(start)  # Add the start point to complete the path.
        return path[::-1]  # Reverse the path to get the correct order.

    # Explore neighbors (up, down, left, right)
    for direction in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
        neighbor = (current[0] + direction[0], current[1] + direction[1])  # Calculate neighbor coordinates.

        # Check if the neighbor is within bounds and not blocked
        if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols and maze[neighbor[0]][neighbor[1]] == 0:
            tentative_g_score = g_score[current] + 1  # Calculate cost to reach the neighbor.

            if neighbor not in g_score or tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current  # Record the current node as the neighbor's parent.
                g_score[neighbor] = tentative_g_score  # Update the cost to reach the neighbor.
                f_score[neighbor] = tentative_g_score + heuristic(neighbor, end)  # Update the estimated total cost.
                heapq.heappush(open_set, (f_score[neighbor], neighbor))  # Add the neighbor to the open set.

    return None

# input
rows = int(input("Enter number of rows: "))
cols = int(input("Enter number of columns: ")
maze = []

# Input the maze
print("Enter the maze row by row (0 for open cell, 1 for blocked cell):")
for i in range(rows):
    row = list(map(int, input().split()))  # Input a row of the maze.
    if len(row) != cols:  # Ensure the row has the correct number of columns.
        print("Invalid row length. Please enter the row again.")
        row = list(map(int, input().split()))
    maze.append(row)

# Input
start = tuple(map(int, input("Enter the start point (row and column): ").split()))
end = tuple(map(int, input("Enter the end point (row and column): ").split()))
```

```
# caller
path = a_star_search(maze, start, end)

# Output
if path:
    total_cost = len(path) - 1
    print("Path found:", path)
    print("Cost:", total_cost)
else:
    print("No path found")
```