

**Artificial Intelligence Systems**

Lab Report # 09

**Submitted by:** Ali Hamza

**Registration No:** B23F0063AI106

**Submitted to:** Dr. Mohsin

Date: 12th April, 2025

# Lab Task 1:

# For the given graph , imagine node A as starting node and your goal is to reach Y. Apply hill climbing and see how closer you can get to your destination.

# 

import heapq  
  
def hill\_climbing(start, goal, obstacles, rows, cols):  
 *"""  
 Hill Climbing algorithm to find a path in the maze.  
 """* def heuristic(node):  
 # Manhattan distance  
 return abs(node[0] - goal[0]) + abs(node[1] - goal[1])  
  
 def get\_neighbors(node):  
 # Possible moves: up, down, left, right  
 moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  
 neighbors = []  
 for move in moves:  
 new\_node = (node[0] + move[0], node[1] + move[1])  
 if 0 <= new\_node[0] < rows and 0 <= new\_node[1] < cols and new\_node not in obstacles:  
 neighbors.append(new\_node)  
 return neighbors  
  
 path = [start]  
 current = start  
 visited = set()  
  
 print(f"Starting Hill Climbing from {start} to {goal}...\n")  
  
 while current != goal:  
 visited.add(current)  
 neighbors = get\_neighbors(current)  
  
 if not neighbors:  
 print("No path found! Stuck at local maxima.")  
 return path  
  
 # Use a priority queue to select the best neighbor  
 priority\_queue = []  
 for neighbor in neighbors:  
 if neighbor not in visited:  
 heapq.heappush(priority\_queue, (heuristic(neighbor), neighbor))  
  
 if not priority\_queue:  
 print("No path found! Stuck at local maxima.")  
 return path  
  
 # Get the neighbor with the best heuristic value  
 \_, next\_node = heapq.heappop(priority\_queue)  
  
 if heuristic(next\_node) >= heuristic(current):  
 print("No path found! Stuck at local maxima.")  
 return path  
  
 print(f"Moving from {current} to {next\_node} (Heuristic: {heuristic(next\_node)})")  
 path.append(next\_node)  
 current = next\_node  
  
 print("\nReached the goal!")  
 print(f"Path length: {len(path)}")  
 return path  
  
  
# Maze dimensions  
rows, cols = 6, 6  
  
# Obstacles (blue cells)  
obstacles = {(0, 2), (0, 3), (0, 4), (1, 2), (1, 4), (2, 0), (2, 4),  
 (3, 0), (3, 4), (4, 0), (4, 1), (4, 2), (4, 4)}  
  
# Start and goal positions  
start = (0, 0) # A  
goal = (5, 4) # Y  
  
# Run the Hill Climbing algorithm  
path = hill\_climbing(start, goal, obstacles, rows, cols)  
  
# Output the path  
print("\nPath taken:")  
for step in path:  
 print(step)

# 

# Lab Task 2:

# Write a program that implements Hill Climbing algorithms to solve this maze. Write the path followed (in the form of coordinates) and the cost of the path.

# 

import numpy as np  
  
# Define the maze  
maze = [  
 [1, 1, 1, 1, 1],  
 [1, 0, 0, 0, 1],  
 [1, 1, 0, 1, 1],  
 [0, 1, 0, 1, 0],  
 [1, 1, 1, 1, 2]  
]  
  
# Define starting point (green) and goal point (red)  
start = (0, 0) # Replace with green's coordinates  
goal = (4, 4) # Replace with red's coordinates  
  
# Hill climbing algorithm with random restarts  
def hill\_climbing\_with\_restarts(maze, start, goal, max\_restarts=10):  
 def heuristic(coord):  
 *"""Calculate Manhattan distance to the goal."""* return abs(coord[0] - goal[0]) + abs(coord[1] - goal[1])  
  
 # Possible moves (down, up, right, left)  
 moves = [(1, 0), (-1, 0), (0, 1), (0, -1)]  
  
 def print\_maze(maze, path):  
 *"""Print the maze with the path marked."""* maze\_copy = np.array(maze, dtype=str)  
 for r in range(len(maze\_copy)):  
 for c in range(len(maze\_copy[0])):  
 if maze\_copy[r][c] == '1':  
 maze\_copy[r][c] = '.'  
 elif maze\_copy[r][c] == '2':  
 maze\_copy[r][c] = 'G'  
 elif maze\_copy[r][c] == '0':  
 maze\_copy[r][c] = '#'  
  
 for step in path:  
 maze\_copy[step[0]][step[1]] = 'P'  
  
 for row in maze\_copy:  
 print(" ".join(row))  
 print("\n")  
  
 for restart in range(max\_restarts):  
 print(f"Restart {restart + 1}/{max\_restarts}")  
 current = start  
 path = [current]  
 cost = 0  
 visited = set()  
  
 while current != goal:  
 visited.add(current)  
 neighbors = []  
 for move in moves:  
 neighbor = (current[0] + move[0], current[1] + move[1])  
 # Check if neighbor is within bounds, not an obstacle, and not visited  
 if (0 <= neighbor[0] < len(maze) and 0 <= neighbor[1] < len(maze[0])  
 and maze[neighbor[0]][neighbor[1]] != 0 and neighbor not in visited):  
 neighbors.append(neighbor)  
  
 # Sort neighbors by heuristic (ascending)  
 neighbors.sort(key=heuristic)  
  
 # Check if there's a better neighbor to move to  
 if neighbors and heuristic(neighbors[0]) < heuristic(current):  
 current = neighbors[0]  
 path.append(current)  
 cost += 1  
 print(f"Moving to {current} (Heuristic: {heuristic(current)})")  
 print\_maze(maze, path)  
 else:  
 print("Stuck in a local minimum, restarting...\n")  
 break  
  
 if current == goal:  
 print("\nReached the goal!")  
 print(f"Path length: {len(path)}")  
 return path, cost  
  
 print("Failed to find a solution after maximum restarts.")  
 return None, None  
  
# Solve the maze  
path, cost = hill\_climbing\_with\_restarts(maze, start, goal)  
  
# Print the results  
if path:  
 print("Path followed:", path)  
 print("Cost of the path:", cost)  
else:  
 print("No path found.")

# 