

**Artificial Intelligence Systems**

Lab Report # 10

**Submitted by:** Ali Hamza

**Registration No:** B23F0063AI106

**Submitted to:** Dr. Mohsin

Date: 24th April, 2025

# Lab Task 1:

# Apply Beam Search to find the shortest path in a weighted grid.

# Requirements:

# Represent the grid as a 2D array where each cell has an associated cost.

# Implement Beam Search to find the path from a start cell to a goal cell with the minimum total cost.

# At each step, consider moving to adjacent cells (up, down, left, right) and compute the cumulative cost.

# Retain only the top beam\_width paths at each level based on the cumulative cost.

# Constraints:

# Avoid revisiting the same cell in a single path.

# Handle grids with obstacles (cells that cannot be traversed).

import heapq  
  
def beam\_search(grid, start, goal, beam\_width):  
 rows, cols = len(grid), len(grid[0])  
  
 # Directions: up, down, left, right  
 directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  
  
 # Beam: (cumulative\_cost, current\_path)  
 beam = [(grid[start[0]][start[1]], [start])]  
  
 while beam:  
 next\_beam = []  
  
 for cost, path in beam:  
 x, y = path[-1]  
  
 if (x, y) == goal:  
 return path, cost  
  
 for dx, dy in directions:  
 nx, ny = x + dx, y + dy  
  
 if 0 <= nx < rows and 0 <= ny < cols and (nx, ny) not in path:  
 cell\_cost = grid[nx][ny]  
 if cell\_cost is not None: # Handle obstacle  
 new\_cost = cost + cell\_cost  
 new\_path = path + [(nx, ny)]  
 next\_beam.append((new\_cost, new\_path))  
  
 # Keep only the top `beam\_width` paths with least cost  
 beam = heapq.nsmallest(beam\_width, next\_beam, key=lambda x: x[0])  
  
 return None, float('inf') # If no path is found  
  
# Example Grid  
# 0s are obstacles (set to None), other numbers are costs  
grid = [  
 [1, 4, 1, 2],  
 [1, None, 3, 1],  
 [2, 1, None, 2],  
 [3, 2, 1, 1]  
]  
  
start = (0, 0)  
goal = (3, 3)  
beam\_width = 2  
  
path, total\_cost = beam\_search(grid, start, goal, beam\_width)  
  
print("Shortest Path:", path)  
print("Total Cost:", total\_cost)

# 

# Lab Task 2:

# Use Beam Search to solve the 8-puzzle problem.

# Requirements:

# Represent the puzzle state as a 3x3 array.

# Define the goal state and implement a heuristic function (e.g., Manhattan distance).

# Implement Beam Search to find the sequence of moves that leads from the initial state to the goal state.

# At each step, generate all possible moves (up, down, left, right) and compute their heuristic scores.

# Retain only the top beam\_width states at each level.

# Constraints:

# Ensure that the algorithm avoids cycles (repeating the same state).

# Limit the depth of the search to prevent infinite loops in unsolvable configurations.

import heapq  
  
GOAL\_STATE = ((1, 2, 3),  
 (4, 5, 6),  
 (7, 8, 0)) # 0 represents the blank tile  
  
def manhattan\_distance(state):  
 distance = 0  
 for i in range(3):  
 for j in range(3):  
 val = state[i][j]  
 if val != 0:  
 goal\_i, goal\_j = (val - 1) // 3, (val - 1) % 3  
 distance += abs(i - goal\_i) + abs(j - goal\_j)  
 return distance  
  
def find\_blank(state):  
 for i in range(3):  
 for j in range(3):  
 if state[i][j] == 0:  
 return i, j  
  
def move(state, direction):  
 i, j = find\_blank(state)  
 new\_state = [list(row) for row in state]  
 moves = {'up': (i - 1, j), 'down': (i + 1, j), 'left': (i, j - 1), 'right': (i, j + 1)}  
  
 if direction in moves:  
 ni, nj = moves[direction]  
 if 0 <= ni < 3 and 0 <= nj < 3:  
 new\_state[i][j], new\_state[ni][nj] = new\_state[ni][nj], new\_state[i][j]  
 return tuple(tuple(row) for row in new\_state)  
 return None  
  
def generate\_children(state):  
 return [(child, dir) for dir in ['up', 'down', 'left', 'right'] if (child := move(state, dir))]  
  
def beam\_search(start\_state, beam\_width=5, max\_depth=50):  
 visited = set()  
 beam = [(manhattan\_distance(start\_state), 0, start\_state, [])]  
  
 while beam:  
 next\_beam = []  
 for h, depth, state, path in beam:  
 if state == GOAL\_STATE:  
 return path  
  
 if depth >= max\_depth:  
 continue  
  
 visited.add(state)  
 for child, move\_dir in generate\_children(state):  
 if child not in visited:  
 next\_beam.append((manhattan\_distance(child), depth + 1, child, path + [move\_dir]))  
  
 beam = heapq.nsmallest(beam\_width, next\_beam, key=lambda x: x[0])  
  
 return None  
  
# Solvable initial state  
initial\_state = ((1, 2, 3),  
 (4, 5, 6),  
 (7, 0, 8))  
  
solution = beam\_search(initial\_state, beam\_width=5)  
  
if solution:  
 print(f"Solution found in {len(solution)} moves:")  
 print(" -> ".join(solution))  
else:  
 print("No solution found within limits.")

# 