

|  |  |
| --- | --- |
| **Student Name** | **Arun Adhikari** |
| **SRN No** | 202100406 |
| **Roll No** | 31 |
| **Program** | **Department of Computer Engineering** |
| **Year** | 2024-2025 |
| **Division** | G |
| **Subject** | Artificial Intelligence lab |
| **Assignment No** | 4 |



**Experiment Number - 04**

**Title /Problem Statement:** **Implement the Best First Search to solve the following problems: a. 8 puzzle b. Robot Navigation problem c. Cities Distance (shortest path) problem**

# Description: The **Best First Search (BFS) algorithm** is a heuristic-based graph traversal technique that selects the most promising node based on a heuristic function. It follows a **greedy approach**, always expanding the node that appears to be the closest to the goal. The algorithm is implemented using a **priority queue (min-heap)** where nodes are sorted based on their heuristic values.

# In this task, we aim to implement **Best First Search** to solve three different problems:

# **(a) 8-Puzzle Problem**

# The **8-puzzle problem** is a classic sliding tile puzzle consisting of a **3×3 grid** with **eight numbered tiles and one empty space**. The objective is to move the tiles strategically to match a given goal state. Best First Search will be used with **Manhattan Distance as a heuristic** to estimate the distance of the current state from the goal.

# **Challenges:**

# Managing valid tile moves within the grid.

# Implementing an efficient **heuristic function (h(n))**.

# Handling visited states to avoid infinite loops.

**(b) Robot Navigation Problem**

The **robot navigation problem** involves finding the shortest path for a robot moving in a **2D grid** (such as a warehouse or maze) from a start position to a goal position. The robot can move **up, down, left, or right**, but obstacles may be present.

**Challenges:**

* Avoiding obstacles while navigating the grid.
* Using **Manhattan Distance** or **Euclidean Distance** as the heuristic function.
* Ensuring optimal movement toward the goal.

**(c) Cities Distance (Shortest Path) Problem**

This problem represents a **road network** where cities are connected by roads of varying distances. The goal is to find the shortest path between a **start city and a destination city** using the Best First Search algorithm.

**Challenges:**

* Representing the road network as a graph.
* Using a heuristic function (e.g., estimated straight-line distance between cities).
* Handling multiple paths to avoid revisiting less optimal routes.

**Theory:**

The Best First Search (BFS) algorithm is a heuristic-based search technique used for solving pathfinding and optimization problems. It follows a greedy approach, always expanding the most promising node based on a heuristic function. This heuristic function estimates how close a given node is to the goal state, helping the algorithm make an informed decision about which node to explore next.

Best First Search is particularly useful for solving graph traversal problems, including:

* The 8-Puzzle Problem (a state-space search problem).
* The Robot Navigation Problem (finding paths in a 2D grid).
* The Cities Distance Problem (finding the shortest route between cities).

Working Principle of Best First Search

1. Heuristic Function (h(n)):
   * BFS uses a heuristic function h(n) to estimate the cost from the current node n to the goal.
   * The node with the lowest h(n) value is expanded first.
2. Priority Queue for Node Selection:
   * Nodes are stored in a priority queue (min-heap), where the node with the lowest heuristic value is given the highest priority.
3. Exploration Strategy:
   * The algorithm selects the most promising node from the queue.
   * If the selected node is the goal, the search stops.
   * Otherwise, the node is expanded, and its neighbors are added to the queue.
4. Avoiding Redundant Expansions:
   * A visited set is used to prevent unnecessary re-exploration of nodes.

Part A:

**Code:**

**import heapq**

**class Puzzle:**

**def \_\_init\_\_(self, board, goal):**

**self.board = board**

**self.goal = goal**

**self.n = len(board)**

**self.moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]**

**def heuristic(self, state):**

**""" Heuristic: Number of misplaced tiles """**

**return sum(1 for i in range(self.n) for j in range(self.n) if state[i][j] != self.goal[i][j] and state[i][j] != 0)**

**def get\_blank\_position(self, state):**

**for i in range(self.n):**

**for j in range(self.n):**

**if state[i][j] == 0:**

**return i, j**

**def generate\_successors(self, state):**

**x, y = self.get\_blank\_position(state)**

**successors = []**

**for dx, dy in self.moves:**

**nx, ny = x + dx, y + dy**

**if 0 <= nx < self.n and 0 <= ny < self.n:**

**new\_state = [row[:] for row in state]**

**new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]**

**successors.append(new\_state)**

**return successors**

**def best\_first\_search(self):**

**pq = []**

**heapq.heappush(pq, (self.heuristic(self.board), self.board))**

**visited = set()**

**while pq:**

**\_, state = heapq.heappop(pq)**

**if state == self.goal:**

**print("Goal reached!")**

**return state**

**visited.add(str(state))**

**for successor in self.generate\_successors(state):**

**if str(successor) not in visited:**

**heapq.heappush(pq, (self.heuristic(successor), successor))**

**return None**

**# Example usage**

**start\_state = [[2, 8, 3], [1, 6, 4], [7, 0, 5]]**

**goal\_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]**

**puzzle = Puzzle(start\_state, goal\_state)**

**solution = puzzle.best\_first\_search()**

**print(solution)**

import heapq

class RobotNavigation:

    def \_\_init\_\_(self, grid, start, goal):

        self.grid = grid

        self.start = start

        self.goal = goal

        self.moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]

    def heuristic(self, pos):

        """ Heuristic: Manhattan distance to goal """

        return abs(pos[0] - self.goal[0]) + abs(pos[1] - self.goal[1])

    def best\_first\_search(self):

        pq = []

        heapq.heappush(pq, (self.heuristic(self.start), self.start))

        visited = set()

        parent = {self.start: None}

        while pq:

            \_, current = heapq.heappop(pq)

            if current == self.goal:

                path = []

                while current:

                    path.append(current)

                    current = parent[current]

                return path[::-1]

            visited.add(current)

            for dx, dy in self.moves:

                nx, ny = current[0] + dx, current[1] + dy

                if 0 <= nx < len(self.grid) and 0 <= ny < len(self.grid[0]) and self.grid[nx][ny] == 0:

                    if (nx, ny) not in visited:

                        heapq.heappush(pq, (self.heuristic((nx, ny)), (nx, ny)))

                        parent[(nx, ny)] = current

        return None

# Example usage

grid = [

    [0, 0, 0, 0, 1],

    [0, 1, 1, 0, 0],

    [0, 0, 0, 1, 0],

    [1, 1, 0, 0, 0]

]

start = (0, 0)

goal = (3, 4)

robot = RobotNavigation(grid, start, goal)

path = robot.best\_first\_search()

print(path)

**import heapq**

**class Graph:**

**def \_\_init\_\_(self):**

**self.edges = {}**

**def add\_edge(self, city1, city2, distance):**

**if city1 not in self.edges:**

**self.edges[city1] = []**

**if city2 not in self.edges:**

**self.edges[city2] = []**

**self.edges[city1].append((distance, city2))**

**self.edges[city2].append((distance, city1))**

**def heuristic(self, city, goal):**

**""" Simplified heuristic assuming straight-line distance is proportional """**

**return abs(hash(city) - hash(goal)) % 100  # Simulated heuristic function**

**def best\_first\_search(self, start, goal):**

**pq = []**

**heapq.heappush(pq, (self.heuristic(start, goal), start))**

**visited = set()**

**parent = {start: None}**

**while pq:**

**\_, current = heapq.heappop(pq)**

**if current == goal:**

**path = []**

**while current:**

**path.append(current)**

**current = parent[current]**

**return path[::-1]**

**visited.add(current)**

**for cost, neighbor in self.edges.get(current, []):**

**if neighbor not in visited:**

**heapq.heappush(pq, (self.heuristic(neighbor, goal), neighbor))**

**parent[neighbor] = current**

**return None**

**# Example usage**

**graph = Graph()**

**graph.add\_edge("A", "B", 4)**

**graph.add\_edge("A", "C", 2)**

**graph.add\_edge("B", "D", 5)**

**graph.add\_edge("C", "D", 8)**

**graph.add\_edge("C", "E", 10)**

**graph.add\_edge("D", "E", 2)**

**graph.add\_edge("D", "F", 6)**

**graph.add\_edge("E", "F", 3)**

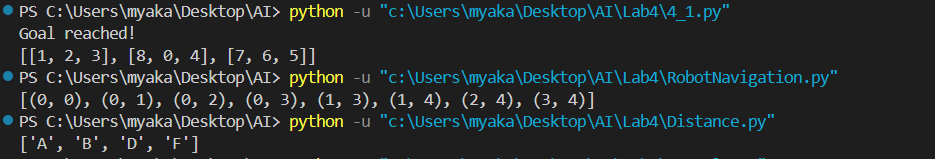
**start\_city = "A"**

**goal\_city = "F"**

**path = graph.best\_first\_search(start\_city, goal\_city)**

**print(path)**

**Output Screenshot:**



**Conclusion:**

The Best First Search (BFS) algorithm is a powerful heuristic-based search technique that prioritizes expanding the most promising node based on a heuristic function. It is widely used in graph traversal and pathfinding problems where a well-defined heuristic can efficiently guide the search toward the goal.

Through its application in the 8-puzzle problem, robot navigation, and cities distance (shortest path) problem, BFS demonstrates its capability to find quick solutions. However, due to its greedy nature, it does not always guarantee the shortest or optimal path.

While BFS is faster than exhaustive searches, it may fail in cases where the heuristic function is not well-designed. In such cases, algorithms like *A (which balances heuristic and path cost)*\* provide better accuracy.

Overall, Best First Search is an efficient approach when speed is a priority and an effective heuristic is available.