



|  |  |
| --- | --- |
| **Student Name** | **Arun Adhikari** |
| **SRN No** | 202100406 |
| **Roll No** | 31 |
| **Program** | **Department of Computer Engineering** |
| **Year** | 2024-2025 |
| **Division** | G |
| **Subject** | Artificial Intelligence |
| **Assignment No** | 5 |



**Experiment Number - 05**

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

**Theory:**

A\* (A-star) is one of the most efficient and widely used search algorithms for pathfinding and graph traversal. It is an informed search algorithm that finds the shortest path from a start node to a goal node using both the cost to reach the node and a heuristic estimate of the remaining cost.

***1. Characteristics of A Algorithm*\***

* **Informed Search Algorithm:** Uses heuristics to guide the search.
* **Optimized for Shortest Path:** Finds the optimal path in many scenarios.
* **Combines Uniform Cost Search (UCS) and Greedy Best-First Search (GBFS):**
  + UCS considers the cost from start to the current node.
  + GBFS considers an estimate of the cost from the current node to the goal.

**Evaluation Function:**

f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n)

Where:

* f(n)f(n)f(n) = total estimated cost of the path through node nnn.
* g(n)g(n)g(n) = cost from the start node to node nnn (known cost).
* h(n)h(n)h(n) = estimated cost from node nnn to the goal (heuristic).

A\* expands nodes with the lowest f(n)f(n)f(n), balancing actual cost and estimated future cost.

***2. Steps of A Algorithm*\***

1. **Initialize:**
   * Place the start node in the **open list** (nodes to be evaluated).
   * Keep a **closed list** (nodes already evaluated).
2. **Loop until the goal is found or open list is empty:**
   * Select the node with the lowest **f(n)f(n)f(n)** from the **open list**.
   * If this node is the **goal**, terminate the search.
   * Otherwise, move it to the **closed list**.
   * Generate its **neighboring nodes**.
   * For each neighbor:
     + Calculate **g(n)g(n)g(n) and h(n)h(n)h(n)**.
     + If the neighbor is in the closed list, ignore it.
     + If it's in the open list, update it if a better path is found.
     + Otherwise, add it to the **open list**.
3. **Reconstruct the path** from goal to start.

***3. Heuristics in A Algorithm*\***

A\* relies on heuristics to estimate the remaining cost. Two common heuristics for grid-based problems like the **8-puzzle problem** or **pathfinding** are:

**(a) Manhattan Distance**

Used in grid-based searches (e.g., 8-puzzle).

h(n)=∣xcurrent−xgoal∣+∣ycurrent−ygoal∣h(n) = |x\_{\text{current}} - x\_{\text{goal}}| + |y\_{\text{current}} - y\_{\text{goal}}|h(n)=∣xcurrent​−xgoal​∣+∣ycurrent​−ygoal​∣

* Measures horizontal and vertical movement.
* Works well in scenarios with **no diagonal movement**.

**(b) Euclidean Distance**

Used when diagonal movement is allowed.

h(n)=(xcurrent−xgoal)2+(ycurrent−ygoal)2h(n) = \sqrt{(x\_{\text{current}} - x\_{\text{goal}})^2 + (y\_{\text{current}} - y\_{\text{goal}})^2}h(n)=(xcurrent​−xgoal​)2+(ycurrent​−ygoal​)2​

**(c) Misplaced Tiles Heuristic**

Used in the **8-puzzle problem**.

h(n)=Number of misplaced tilesh(n) = \text{Number of misplaced tiles}h(n)=Number of misplaced tiles

* Simple but less accurate than Manhattan distance.

**Code:**

import heapq

# ---------- 8 PUZZLE ----------

goal\_8\_puzzle = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

def manhattan(state):

    distance = 0

    for i in range(3):

        for j in range(3):

            value = state[i][j]

            if value != 0:

                goal\_x, goal\_y = divmod(value - 1, 3)

                distance += abs(i - goal\_x) + abs(j - goal\_y)

    return distance

def get\_neighbors\_8(state):

    neighbors = []

    x, y = next((i, j) for i in range(3) for j in range(3) if state[i][j] == 0)

    directions = [(-1,0), (1,0), (0,-1), (0,1)]

    for dx, dy in directions:

        nx, ny = x + dx, y + dy

        if 0 <= nx < 3 and 0 <= ny < 3:

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

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

            neighbors.append(new\_state)

    return neighbors

def a\_star\_8\_puzzle(start):

    visited = set()

    heap = [(manhattan(start), 0, start, [])]

    while heap:

        f, g, state, path = heapq.heappop(heap)

        key = tuple(tuple(row) for row in state)

        if key in visited:

            continue

        visited.add(key)

        if state == goal\_8\_puzzle:

            return path + [state]

        for neighbor in get\_neighbors\_8(state):

            heapq.heappush(heap, (g + 1 + manhattan(neighbor), g + 1, neighbor, path + [state]))

# ---------- ROBOT NAVIGATION ----------

def heuristic\_robot(a, b):

    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a\_star\_robot(grid, start, goal):

    rows, cols = len(grid), len(grid[0])

    heap = [(0 + heuristic\_robot(start, goal), 0, start, [])]

    visited = set()

    while heap:

        f, g, current, path = heapq.heappop(heap)

        if current in visited:

            continue

        visited.add(current)

        if current == goal:

            return path + [current]

        for dx, dy in [(-1,0), (1,0), (0,-1), (0,1)]:

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

            if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] != 1:

                heapq.heappush(heap, (g + 1 + heuristic\_robot((nx, ny), goal), g + 1, (nx, ny), path + [current]))

# ---------- CITIES DISTANCE ----------

def a\_star\_cities(graph, start, goal, heuristic):

    heap = [(heuristic[start], 0, start, [])]

    visited = set()

    while heap:

        f, g, current, path = heapq.heappop(heap)

        if current in visited:

            continue

        visited.add(current)

        if current == goal:

            return path + [current]

        for neighbor, cost in graph[current].items():

            heapq.heappush(heap, (g + cost + heuristic[neighbor], g + cost, neighbor, path + [current]))

# ---------- MAIN MENU ----------

def main():

    print("A\* Algorithm Problems")

    print("1. 8 Puzzle")

    print("2. Robot Navigation")

    print("3. Cities Distance (Shortest Path)")

    choice = int(input("Enter your choice (1-3): "))

    if choice == 1:

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

        print("\nSolving 8 Puzzle...")

        solution = a\_star\_8\_puzzle(start\_state)

        for step in solution:

            for row in step:

                print(row)

            print("------")

    elif choice == 2:

        grid = [

            [0, 0, 0, 0],

            [1, 1, 0, 1],

            [0, 0, 0, 0],

            [0, 1, 1, 0]

        ]

        start = (0, 0)

        goal = (3, 3)

        print("\nSolving Robot Navigation...")

        path = a\_star\_robot(grid, start, goal)

        print("Path:", path)

    elif choice == 3:

        graph = {

            'A': {'B': 1, 'C': 4},

            'B': {'A': 1, 'C': 2, 'D': 5},

            'C': {'A': 4, 'B': 2, 'D': 1},

            'D': {'B': 5, 'C': 1}

        }

        heuristic = {

            'A': 7,

            'B': 6,

            'C': 2,

            'D': 0

        }

        start = 'A'

        goal = 'D'

        print("\nSolving Cities Distance...")

        path = a\_star\_cities(graph, start, goal, heuristic)

        print("Shortest Path:", path)

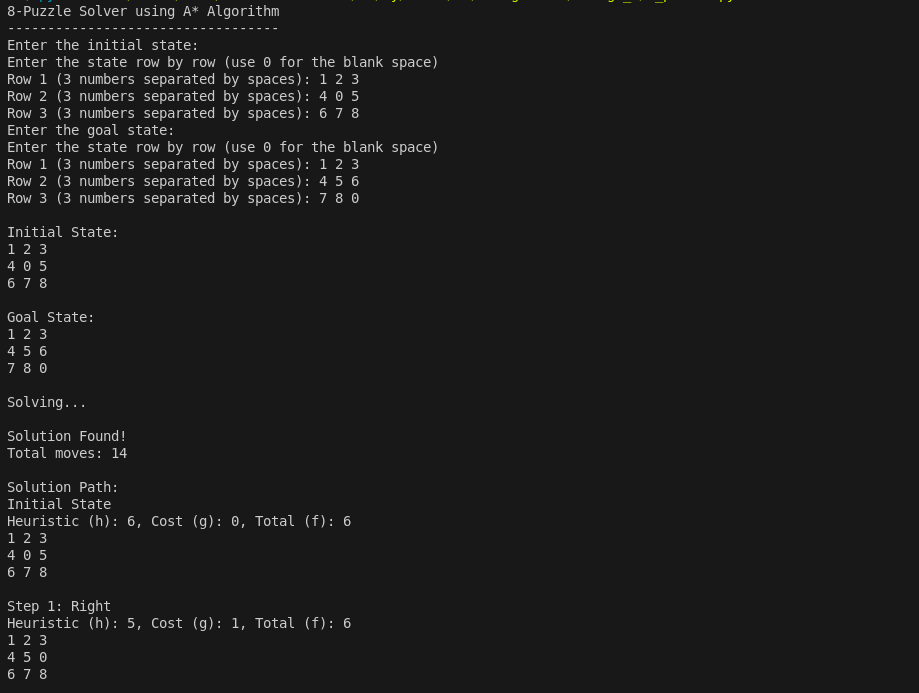
    else:

        print("Invalid choice.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Output:**



**Conclusion:**

The A\* (A-star) algorithm is a highly efficient and widely used pathfinding and graph traversal technique that combines the strengths of both Dijkstra's algorithm and Greedy Best-First Search. By using a cost function f(n) = g(n) + h(n), where g(n) represents the actual cost from the start node and h(n) is the heuristic estimated cost to the goal, A\* ensures both optimality and completeness when the heuristic is admissible. Its flexibility allows it to be applied to a variety of real-world problems such as solving the 8 Puzzle, navigating robots in grid environments, and finding the shortest path between cities. A\* intelligently prioritizes paths that appear more promising, thereby significantly reducing the search space compared to uninformed search algorithms. Overall, A\* stands out as a fundamental algorithm in the domain of artificial intelligence and operations research due to its balance of efficiency, accuracy, and applicability across domains.