****

**NAME – Ashish Gupta**

**BRANCH- CSEAI**

**UNIVERSITY ROLL NO – 202401100300073**

**CLASS ROLL NO. 73**

**Title- *"Solving the 8-Puzzle Problem Using the A Search Algorithm"*\***

**Introduction**

The 8-puzzle is a sliding puzzle consisting of a 3x3 grid with 8 numbered tiles and one blank space. The objective is to arrange the tiles in a specific goal configuration, usually:

Copy

1 2 3

4 5 6

7 8 0

where the blank space is represented by 0. The challenge lies in sliding the tiles into the correct positions by moving one tile at a time into the blank space. The goal of this project is to develop a solution for the 8-puzzle problem using the *A search algorithm*\*, a popular method for finding the optimal path in search problems.

**Problem Representation**

* **State Representation**: The 8-puzzle consists of a 3x3 grid. The grid contains 8 numbered tiles and one blank space, represented by 0. A state of the puzzle is the configuration of these tiles and the blank space.
  + Each state is represented as a 3x3 matrix, such as:

Copy

1 2 3

4 5 6

7 8 0

Here, the goal is to move the tiles such that they reach the goal configuration.

* **Goal State**: The goal state for the 8-puzzle is predefined and is usually represented as:

Copy

1 2 3

4 5 6

7 8 0

The task is to rearrange the tiles from a given start state to this goal configuration.

**2. *A Search Algorithm*\***

The A\* algorithm is a best-first search algorithm that uses both the cost to reach the current node (g(n)) and the estimated cost from the current node to the goal node (h(n)). The total cost function used by A\* is:

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

Where:

* g(n) is the number of moves it took to reach the current state from the start state.
* h(n) is the heuristic value that estimates the remaining cost to reach the goal state.

***A Search Process:*\***

The general process for applying the A\* algorithm to the 8-puzzle is as follows:

1. **Initialization**:
   * Begin with the initial state of the puzzle.
   * Calculate its Manhattan Distance heuristic, which measures how far each tile is from its goal position.
   * Push the initial state into a priority queue (using a heap for efficient retrieval).
2. **Exploration**:
   * While the priority queue is not empty:
     + Pop the state with the lowest value of f(n) = g(n) + h(n) from the queue.
     + If this state matches the goal state, the algorithm terminates and returns the solution path.
     + Otherwise, generate all possible valid successor states by sliding the tiles into the blank space.
     + For each successor, calculate g(n) (incremented by 1 from the parent state) and h(n) (Manhattan Distance for that state).
     + Add each valid successor to the priority queue if it hasn't been visited.
3. **Termination**:
   * If the goal state is found, the algorithm traces back the path from the goal state to the start state, which forms the solution.
   * If no solution exists (all possible states have been explored), the algorithm terminates and indicates failure.

**3. Heuristic Function: Manhattan Distance**

The **Manhattan Distance** heuristic is used in this implementation of the A\* algorithm. The Manhattan Distance for a tile is the sum of the absolute horizontal and vertical distances between its current position and its goal position.

For example:

* The tile 1 in the start state might be at position (1, 1) but needs to be at position (0, 0). The Manhattan distance for tile 1 is |1-0| + |1-0| = 2.

The heuristic for the entire puzzle is the sum of the Manhattan distances of all the tiles from their current positions to their goal positions.

**4. State Space Exploration**

The A\* algorithm explores the state space by repeatedly selecting the state with the lowest total cost (f(n)) from the priority queue. The steps involved in exploring the state space are:

1. **Generate Successors**:
   * For each state, the algorithm generates all possible states that can be reached by sliding a tile into the blank space. The valid directions are up, down, left, and right (if the move is within the bounds of the puzzle).
2. **Check for Goal**:
   * After generating all successor states, the algorithm checks whether any of them match the goal state. If the goal state is found, the search ends.
3. **Avoid Redundancy**:
   * States that have already been visited are not revisited to avoid loops and redundant computations. This is tracked using a set that stores visited states as immutable tuples of rows.

**5. Solution Path Reconstruction**

Once the goal state is reached, the solution path can be reconstructed by tracing back from the goal state to the initial state. Each state contains a reference to its parent state (the state from which it was reached). The path is then reversed to display the sequence of states from the initial configuration to the goal configuration.

**6. Time and Space Complexity**

* **Time Complexity**: The worst-case time complexity of the A\* algorithm is O(bd)O(b^d)O(bd), where:
  + bbb is the branching factor (the average number of valid moves from each state).
  + ddd is the depth of the solution (the number of moves from the start to the goal).

Since the 8-puzzle has a relatively small state space, the search typically explores fewer states compared to larger puzzles.

* **Space Complexity**: The space complexity is also O(bd)O(b^d)O(bd), as the algorithm needs to store all the generated states in memory. This includes the states in the priority queue and the set of visited states.

**7. Algorithm Efficiency**

The efficiency of the A\* algorithm in solving the 8-puzzle problem depends on:

* The **quality of the heuristic**: Manhattan Distance is a very efficient heuristic for the 8-puzzle, guiding the search towards the goal more effectively.
* The **size of the state space**: For the 8-puzzle, the state space has 9! (362,880) possible configurations, but the search is usually much smaller due to the effective heuristic and pruning of redundant states.

The algorithm ensures that the solution is optimal by always exploring the most promising states first.

**Summary of the Methodology:**

1. **Problem Definition**: Represent the puzzle as a 3x3 grid and define the goal state.
2. **Search Strategy**: Apply the A\* algorithm using the Manhattan Distance heuristic to guide the search efficiently.
3. **State Exploration**: Generate valid successor states by sliding tiles and use a priority queue to explore the state space.
4. **Solution Path**: Reconstruct and return the solution path once the goal state is reached.
5. **Complexity**: A\* provides an optimal and efficient solution with manageable time and space complexity for the 8-puzzle.

This methodology provides an effective framework for solving the 8-puzzle problem and can be extended to larger sliding puzzles.

**Code**

**import heapq**

**# The goal state**

**GOAL\_STATE = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]**

**# Direction vectors for moving blank space**

**MOVES = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right**

**class Puzzle:**

**def \_\_init\_\_(self, state, moves=0, prev=None):**

**self.state = state**

**self.moves = moves**

**self.prev = prev**

**self.blank\_pos = self.find\_blank()**

**self.manhattan\_distance = self.calculate\_manhattan\_distance()**

**# Calculate Manhattan distance (sum of distances for all tiles)**

**def calculate\_manhattan\_distance(self):**

**distance = 0**

**for r in range(3):**

**for c in range(3):**

**val = self.state[r][c]**

**if val == 0:**

**continue**

**goal\_r, goal\_c = divmod(val - 1, 3)**

**distance += abs(goal\_r - r) + abs(goal\_c - c)**

**return distance**

**# Find the position of the blank space (0)**

**def find\_blank(self):**

**for r in range(3):**

**for c in range(3):**

**if self.state[r][c] == 0:**

**return r, c**

**return None**

**# Generate all possible next states by moving the blank**

**def get\_next\_states(self):**

**next\_states = []**

**r, c = self.blank\_pos**

**for dr, dc in MOVES:**

**nr, nc = r + dr, c + dc**

**if 0 <= nr < 3 and 0 <= nc < 3:**

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

**new\_state[r][c], new\_state[nr][nc] = new\_state[nr][nc], new\_state[r][c]**

**next\_states.append(Puzzle(new\_state, self.moves + 1, self))**

**return next\_states**

**# A\* priority queue comparison**

**def \_\_lt\_\_(self, other):**

**return (self.moves + self.manhattan\_distance) < (other.moves + other.manhattan\_distance)**

**def a\_star\_solver(start\_state):**

**start\_puzzle = Puzzle(start\_state)**

**if start\_puzzle.state == GOAL\_STATE:**

**return []**

**pq = []**

**heapq.heappush(pq, start\_puzzle)**

**visited = set()**

**while pq:**

**current = heapq.heappop(pq)**

**visited.add(tuple(tuple(row) for row in current.state)) # Immutable representation for visited states**

**# Check if we've reached the goal**

**if current.state == GOAL\_STATE:**

**solution = []**

**while current:**

**solution.append(current.state)**

**current = current.prev**

**return solution[::-1]**

**# Generate next states**

**for next\_state in current.get\_next\_states():**

**state\_tuple = tuple(tuple(row) for row in next\_state.state)**

**if state\_tuple not in visited:**

**heapq.heappush(pq, next\_state)**

**return None # No solution found**

**def print\_solution(solution):**

**if solution is None:**

**print("No solution found")**

**else:**

**for step in solution:**

**for row in step:**

**print(row)**

**print()**

**# Example 8-puzzle start state**

**start\_state = [**

**[1, 2, 3],**

**[5, 6, 0],**

**[4, 7, 8]**

**]**

**solution = a\_star\_solver(start\_state)**

**print\_solution(solution)**