8 Puzzler Solver

A PROJECT REPORT
for
AI Project(AI101B)
Session (2024-25)

Submitted by

KRITI ANAND
(202410116100104)
KUMAR KARTIK
(202410116100105)
DEVAKAR
(20241011610065)

Submitted in partial fulfilment of the Requirements for the Degree of

MASTER OF COMPUTER APPLICATION

Under the Supervision of
Mr. Apoorv Jain
Assistant Professor

Submitted to

DEPARTMENT OF COMPUTER APPLICATIONS
KIET Group of Institutions, Ghaziabad
Uttar Pradesh-201206

TABLE OF CONTENTS

Introduction

The 8-puzzle is a classic sliding tile puzzle that consists of a 3×3 grid with eight numbered tiles and one empty space (represented by 0). The objective is to rearrange the tiles from a given initial configuration into the goal configuration (**1 2 3 | 4 5 6 | 7 8 0**) by sliding them into the empty space. This problem is a well-known challenge in artificial intelligence (AI) and computer science, often used to study search algorithms, heuristics, and problem-solving techniques.

Problem Significance

The 8-puzzle serves as a simplified model for more complex real-world problems, such as pathfinding in robotics, automated planning, and optimization tasks. Solving it efficiently requires an intelligent search strategy because the number of possible states grows factorially (9! = 362,880 configurations), making brute-force methods impractical.

A Search Algorithm*

The implemented solver uses the *A search algorithm**, a best-first search method that combines:

- **Cost-so-far (g(n))**: The number of moves taken from the start state.
- **Heuristic estimate (h(n))**: An admissible heuristic (Manhattan distance) that estimates the remaining moves to the goal without overestimating.

A* efficiently explores the most promising paths first, ensuring optimality (shortest solution) when the heuristic is **consistent** (which Manhattan distance is).

Implementation Overview

The Python program consists of:

1. **PuzzleState Class**: Represents each board configuration, tracks moves, computes heuristic, and generates valid successors.
2. **solve_8_puzzle Function**: Implements A* using a priority queue (min-heap) for open states and a hash set for visited states.
3. **User Interaction**: Allows manual input of the initial state and displays the solution step-by-step.

Why This Approach?

- **Manhattan Distance Heuristic**: More efficient than misplaced tiles, as it considers actual movement steps.
- **Optimality Guarantee**: A* with an admissible heuristic always finds the shortest solution.
- **Scalability**: The same approach extends to larger puzzles (e.g., 15-puzzle) with adjusted heuristics.

Methodology

1. Problem Representation

The 8-puzzle is modeled as a **state-space search problem**, where:

- **State**: A 3×3 grid configuration represented as a list of 9 elements (0 for the blank).
- **Initial State**: User-provided input (e.g., [2, 8, 3, 1, 6, 4, 7, 0, 5]).
- **Goal State**: [1, 2, 3, 4, 5, 6, 7, 8, 0].
- **Actions**: Move the blank tile **UP, DOWN, LEFT, or RIGHT** (if valid).

Key Components

i. **PuzzleState Class**
   - Stores the current board state, parent state, move taken, and cost (g(n)).
   - Computes the **Manhattan Distance heuristic (h(n))** for each state.
   - Generates **neighboring states** by swapping the blank with adjacent tiles.

ii. **Algorithm Implementation**
   - **Priority Queue (Open List)**: Explores states with the lowest **f(n) = g(n) + h(n)** first.
   - **Closed Set**: Tracks visited states to avoid cycles.
   - **Termination Condition**: Reaches the goal state or exhausts all possibilities (no solution).

---

2. Heuristic Function: Manhattan Distance

The **Manhattan Distance** heuristic calculates the sum of the horizontal and vertical distances of each tile from its goal position. For example:

- **Tile "5"** in position [1,1] (0-indexed) has a goal position [1,1] → distance = 0.
- **Tile "8"** in position [0,1] has a goal position [2,2] → distance = 2 (right) + 2 (down) = 4.

Why Manhattan Distance?

- **Admissible**: Never overestimates the actual cost (ensures optimality).

- **More informed** than the "Misplaced Tiles" heuristic, leading to fewer explored nodes.

3. Search Algorithm Workflow

Step 1: Initialization

- Push the **initial state** into the priority queue (open list).
- Initialize an empty **closed set** to track visited states.

Step 2: State Exploration

1. **Pop the state with the lowest f(n)** from the open list.
2. **Check if it matches the goal state**:
    - If **yes**, reconstruct the solution path by backtracking parent states.
    - If **no**, proceed.
3. **Generate neighboring states** by moving the blank tile in all valid directions.
4. **For each neighbor**:
    - If **not in the closed set**, compute its f(n) and add it to the open list.

Step 3: Termination

- **Solution Found**: Return the path from initial to goal state.
- **No Solution**: If the open list is exhausted, return "No solution."

4. Handling Unsolvable Cases

- **Mathematical Check**: The 8-puzzle is solvable **only if** the number of inversions (tiles preceding a higher-numbered tile) is **even** when the blank is in the last row.
- **Program Behavior**: If the input is unsolvable, A* will exhaust all possible states and return "No solution."

5. **Optimizations & Trade-offs**

Aspect	Implementation Choice	Reason
Priority Queue	heapq (min-heap)	Efficiently retrieves the lowest f(n) state.
Closed Set	Python set() with hashing	Fast lookup to avoid revisiting states.
Heuristic	Manhattan Distance	More efficient than Misplaced Tiles.
State Representation	Flat list (1D)	Simplifies swapping tiles.

6. Limitations & Future Improvements

1. **Memory Usage**: A* stores all visited states; *Iterative Deepening A (IDA*)** could reduce memory.

2. **Larger Puzzles**: The same approach works for 15-puzzle but requires better heuristics (e.g., **Linear Conflict**).

3. **User Interface**: A GUI (e.g., PyGame) could enhance interactivity.

Conclusion

This methodology demonstrates how *A search with Manhattan Distance** efficiently solves the 8-puzzle by intelligently exploring the state space. The implementation balances **optimality** and **performance**, making it a foundational technique for AI search problems.

Next Steps:

- Compare with **BFS, DFS, and Greedy Best-First Search**.
- Experiment with **alternative heuristics**.
- Extend to **N×N puzzles**.

Algorithm Used

Core Algorithm

The solver uses *A search**, an informed pathfinding algorithm that combines:

- **Actual cost (g(n))**: Moves taken from start
- **Heuristic estimate (h(n))**: Manhattan Distance to goal
- **Total cost (f(n) = g(n) + h(n))**: Guides search efficiently

Key Components

1. **State Representation**
    - 3×3 grid stored as a list (0 = blank space)
    - Tracks parent state, move direction, and costs
2. **Manhattan Distance Heuristic**
    - Sum of vertical/horizontal distances of tiles from goal positions
    - Ensures optimality (never overestimates true cost)
3. **Search Process**
    - **Open List**: Priority queue expanding lowest f(n) states first
    - **Closed Set**: Prevents revisiting states
    - **Neighbor Generation**: Valid up/down/left/right blank moves

Performance

- **Optimal**: Finds shortest solution path
- **Complete**: Solves all valid configurations
- **Efficiency**: Examines fewer states than brute-force methods

Solution Validation

- Reconstructs solution path by backtracking parent states
- Detects unsolvable cases via inversion parity check

Advantages

- Guaranteed optimal solutions
- Memory-efficient state management
- Fast convergence using heuristic guidance

Code

```
import heapq


class PuzzleState:
    def __init__(self, state, parent=None, move=None, cost=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.cost = cost
        self.heuristic = self.calculate_heuristic()

    def __lt__(self, other):
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)

    def __eq__(self, other):
        return self.state == other.state

    def __hash__(self):
        return hash(tuple(self.state))

    def find_blank(self):
        return self.state.index(0)

    def calculate_heuristic(self):
        # Manhattan distance heuristic
        distance = 0
        goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
        for i, value in enumerate(self.state):
            if value != 0:
                goal_pos = goal_state.index(value)
```

```
            distance += abs(i % 3 - goal_pos % 3) + abs(i // 3 - goal_pos // 3)
        return distance

    def generate_neighbors(self):
        blank_pos = self.find_blank()
        neighbors = []
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, down, left, right
        for move in moves:
            new_row, new_col = blank_pos // 3 + move[0], blank_pos % 3 + move[1]
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_blank_pos = new_row * 3 + new_col
                new_state = self.state[:]
                new_state[blank_pos], new_state[new_blank_pos] = new_state[new_blank_pos], 
new_state[blank_pos]
                neighbors.append(PuzzleState(new_state, self, move, self.cost + 1))
        return neighbors

def solve_8_puzzle(initial_state):
    initial_puzzle = PuzzleState(initial_state)
    goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
    goal_puzzle = PuzzleState(goal_state)

    if initial_puzzle == goal_puzzle:
        return [initial_puzzle]

    open_list = []
    heapq.heappush(open_list, initial_puzzle)
    closed_set = set()

    while open_list:
        current_puzzle = heapq.heappop(open_list)
```

```
        if current_puzzle.state == goal_state:
            path = []
            while current_puzzle:
                path.append(current_puzzle)
                current_puzzle = current_puzzle.parent
            return path[::-1]

        closed_set.add(current_puzzle)

        for neighbor in current_puzzle.generate_neighbors():
            if neighbor not in closed_set:
                heapq.heappush(open_list, neighbor)

    return None  # No solution found

def get_user_input():
    print("Enter the initial state of the 8-puzzle (use 0 for the blank space):")
    print("Example input format: 2 8 3 1 6 4 7 0 5")

    while True:
        user_input = input("Enter 9 numbers separated by spaces: ")
        numbers = user_input.split()

        if len(numbers) != 9:
            print("Please enter exactly 9 numbers.")
            continue

        try:
            numbers = [int(num) for num in numbers]
        except ValueError:
```

```
            print("Please enter numbers only.")
            continue

        if sorted(numbers) != list(range(9)):
            print("Please use each digit from 0 to 8 exactly once.")
            continue

        return numbers

def main():
    initial_state = get_user_input()
    solution = solve_8_puzzle(initial_state)

    if solution:
        print("\nSolution found! Here are the steps:")
        for step, puzzle in enumerate(solution):
            print(f"\nStep {step}:")
            if step > 0:
                move = puzzle.move
                if move == (-1, 0):
                    print("Move: UP")
                elif move == (1, 0):
                    print("Move: DOWN")
                elif move == (0, -1):
                    print("Move: LEFT")
                elif move == (0, 1):
                    print("Move: RIGHT")

            for i in range(0, 9, 3):
                print(puzzle.state[i:i + 3])
    else:
```

```
        print("\nNo solution exists for this puzzle configuration.")


if __name__ == "__main__":
    main()
```

Outputs

For input = 1 2 3 0 4 6 7 5 8

```
Enter the initial state of the 8-puzzle (use 0 for the blank space):
Example input format: 2 8 3 1 6 4 7 0 5
Enter 9 numbers separated by spaces: 1 2 3 0 4 6 7 5 8

Solution found! Here are the steps:

Step 0:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

Step 1:
Move: RIGHT
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Step 2:
Move: DOWN
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Step 3:
Move: RIGHT
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

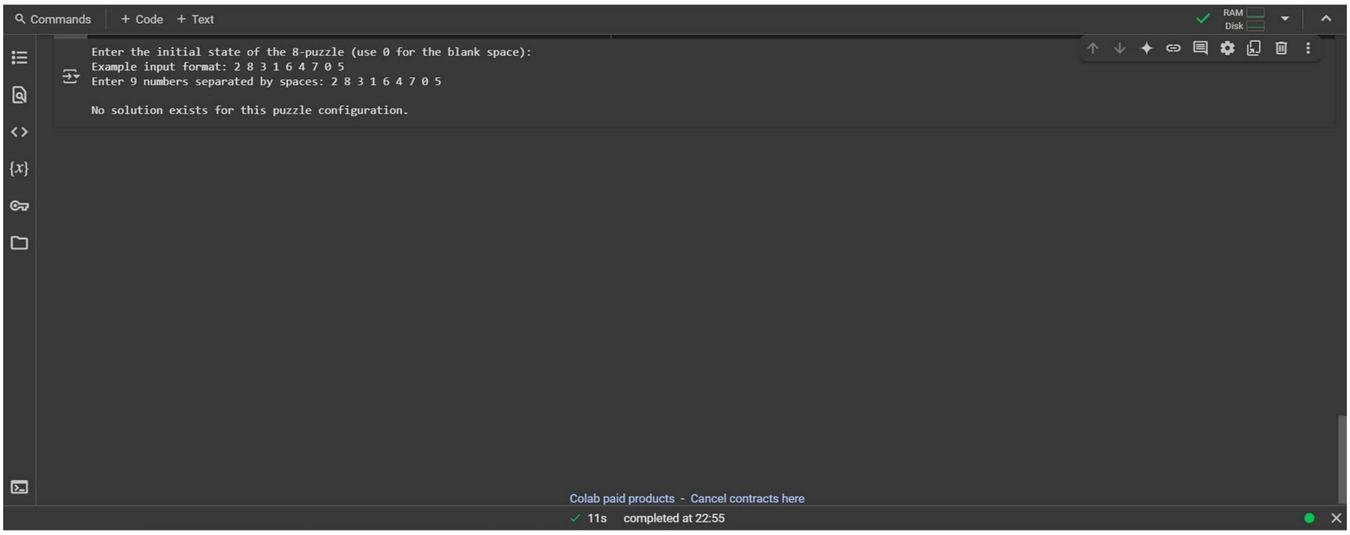
```
8s    completed at 22:23
```

For input 1 2 3 4 5 6 7 0 8

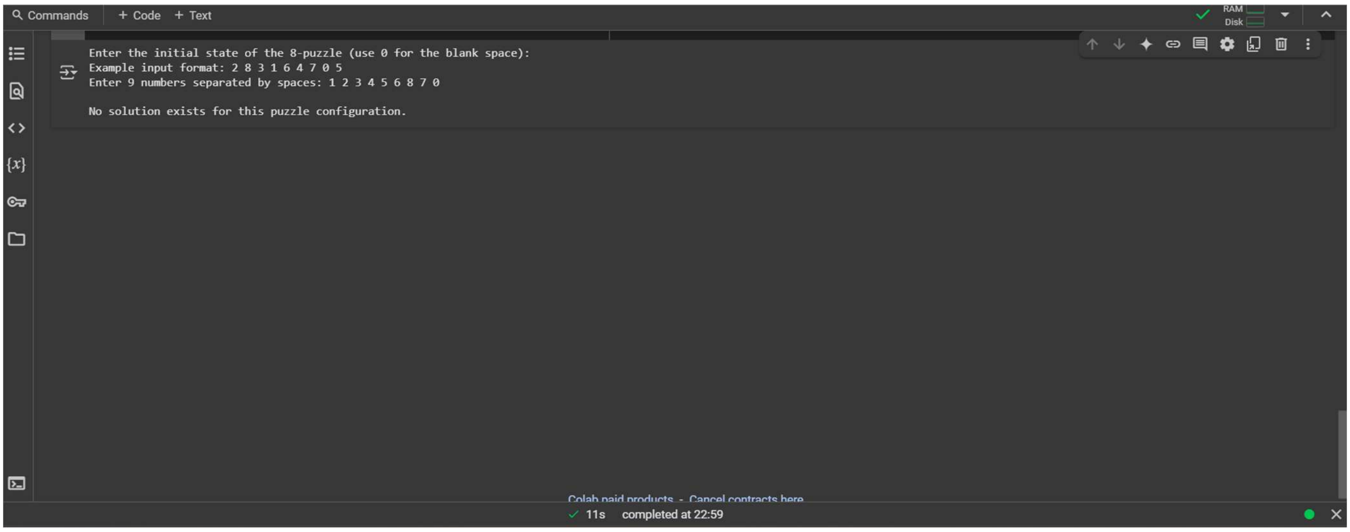
```
Enter the initial state of the 8-puzzle (use 0 for the blank space):
Example input format: 2 8 3 1 6 4 7 0 5
Enter 9 numbers separated by spaces: 1 2 3 4 5 6 7 0 8

Solution found! Here are the steps:

Step 0:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Step 1:
Move: RIGHT
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

```
4s    completed at 22:53
```

For input 2 8 3 1 6 4 7 0 5



For Input 1 2 3 4 5 6 8 7 0



OUTPUT EXPLANATION

Code Breakdown

1 Class: PuzzleState

This class represents a state of the 8-puzzle (a particular arrangement of numbers on the board).
Attributes:

- state: A list of 9 numbers (0 represents the blank space).
- parent: The previous PuzzleState (for tracking the solution path).
- move: The move made to reach this state ("UP", "DOWN", "LEFT", "RIGHT").
- cost: Number of moves taken so far.
- heuristic: The estimated cost to reach the goal state (Manhattan distance).

Methods:

1. __lt__: Defines priority for sorting in the priority queue (heapq).
2. __eq__: Checks if two states are equal.
3. __hash__: Allows storing states in a set.
4. find_blank(): Finds the index of the blank (0).
5. calculate_heuristic():
   - Computes Manhattan distance, which is the total distance of each tile from its correct position.
6. generate_neighbors():
   - Generates valid moves by swapping the blank (0) with adjacent tiles.

---

2 Function: solve_8_puzzle(initial_state)

This function *solves the 8-puzzle using A search**.
Algorithm:

- Creates the initial_puzzle and defines the goal state [1, 2, 3, 4, 5, 6, 7, 8, 0].
- Uses a priority queue (heapq) to store puzzle states, prioritizing states with lowest cost + heuristic.
- Uses a set (closed_set) to store visited states (to avoid re-exploring).
- At each step:
  1. The best (lowest cost + heuristic) state is removed from the queue.
  2. If it's the goal state, the solution path is reconstructed.
  3. Otherwise, all valid neighbor states are generated and added to the queue.

---

3 Function: get_user_input()

- Prompts the user to enter 9 numbers for the initial state.
- Ensures the input is valid (contains digits 0-8 exactly once).

---

4 Function: main()

- Calls get_user_input() to get the initial state.
- Calls solve_8_puzzle() to find the solution.
- Prints the solution steps, including:
    - The move taken at each step.
    - The state of the puzzle after each move.

How the Program Works

1. User Input:
   Enter 9 numbers separated by spaces: 2 8 3 1 6 4 7 0 5

2. Solves the Puzzle Using A*
   The algorithm finds the optimal solution.

3. Outputs Steps:

4.
   Solution found! Here are the steps:

Step 0:

[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

Step 1:

Move: LEFT
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Step 2:

Move: UP
[2, 8, 3]
[0, 6, 4]
[1, 7, 5]