Report on 8-Puzzle Solver Using AI

Introduction

The 8-puzzle is a classic problem in artificial intelligence, often used to illustrate search algorithms, heuristic methods, and problem-solving techniques. The puzzle consists of a 3x3 grid with 8 numbered tiles (1 to 8) and one empty space (0). The objective of the puzzle is to reach a goal state from a given start state by sliding the tiles into the empty space. The puzzle is commonly used to demonstrate AI algorithms such as *A search**, **breadth-first search**, and **depth-first search**.

This report discusses the implementation of an 8-puzzle solver using *A search** and the **Manhattan distance heuristic**. Additionally, it provides insights into the solution and the potential of AI to solve combinatorial puzzles efficiently.

Problem Description

In the 8-puzzle, the goal is to arrange the tiles into a specific configuration, typically represented as:

```
1 2 3
4 5 6
7 8 0
```

Where the number `0` represents the empty space. The puzzle starts in a random configuration, and the task is to move the tiles around until the puzzle matches the goal state. The valid moves involve sliding a tile into the empty space (up, down, left, or right).

Solution Approach

1. Search Algorithms: A* Search

The *A algorithm** is used in the solution of the 8-puzzle problem. A* is a best-first search algorithm that uses a heuristic to determine the most promising paths to explore. The main idea of A* is to maintain a set of paths that are most likely to lead to the goal.

For the 8-puzzle, A* combines two values:

- **g(n)**: The cost to reach a node from the start.
- **h(n)**: The heuristic estimate of the cost to reach the goal from the current node.

The algorithm explores the node with the smallest value of **f(n) = g(n) + h(n)**. A* guarantees an optimal solution if the heuristic is admissible (it never overestimates the cost to reach the goal).

2. Heuristic Function: Manhattan Distance

For the 8-puzzle, we use the **Manhattan distance** as the heuristic function. The Manhattan distance for each tile is the sum of the absolute differences between the current position of the tile and its goal position. This heuristic is admissible because it always underestimates or equals the true cost of reaching the goal.

The Manhattan distance for a tile is computed as:

```
|current_row - goal_row| + |current_column - goal_column|
```

3. Data Structure: Priority Queue (Min-Heap)

To implement A*, we use a **priority queue** (min-heap) to explore the nodes with the smallest **f(n)** value first. The priority queue stores states and their corresponding costs (f, g, h). This ensures that the algorithm explores the most promising states first, improving its efficiency.

4. Visited States: Tracking States

To avoid revisiting states and to ensure the algorithm terminates, we maintain a list of visited states. If a state is encountered that has already been explored, it is skipped.

5. Solving the Puzzle: AI Solution

The solver automatically uses the A* search to find the solution path from the start state to the goal state. Once the puzzle is solved, it outputs the sequence of moves to solve the puzzle.

Key Components of the 8-Puzzle Solver

1. State Representation:

The state of the puzzle is represented as a 3x3 tuple, where each tuple represents a row in the grid. This allows for easy manipulation and comparison of states.

Example:

```
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)
```

2. Movement Logic:

The empty space (0) can be moved up, down, left, or right. This is handled by checking the boundaries of the grid and swapping the empty space with the adjacent tile in the chosen direction.

3. User Interaction:

The user has the option to either play the puzzle manually by making moves or use the AI to solve the puzzle automatically. In case the user decides to quit, the AI takes over and solves the puzzle from the current configuration.

4. Random Start State:

The start state is randomly generated using a shuffle of the numbers 0 through 8. To ensure the puzzle is solvable, we check for the number of inversions in the state. If the state is unsolvable, we regenerate the state until we find one that is solvable.

Code Implementation

The implementation is done in Python, using libraries like `heapq` for the priority queue and `pandas` for tracking visited states. The code is structured to handle both user interaction and AI solving, allowing for an engaging experience where the user can either play the puzzle manually or have it solved by the AI.

The following is a simplified structure of the main components:

```
import random
import heapq
import pandas as pd

# Goal state and Manhattan distance heuristic
goal_state = ((1, 2, 3), (4, 5, 6), (7, 8, 0))

# A* solver function, Manhattan distance calculation, and puzzle
manipulation
# ...

def a_star_solver(start_state):
    # Implementation of the A* search algorithm
    pass

def manhattan_distance(state):
    # Calculate Manhattan distance for the current state
    pass

# Puzzle interaction functions: user_play, solve_puzzle, etc.
# ...
```

Performance Analysis

The performance of the A* algorithm in the 8-puzzle problem is highly dependent on the quality of the heuristic used. The Manhattan distance is a relatively efficient heuristic and ensures that the A* algorithm performs well for solving the puzzle in a reasonable amount of time.

Time Complexity:

The time complexity of the A* algorithm is influenced by the number of states explored. In the worst case, the algorithm may explore all possible configurations of the puzzle. There are a maximum of **9! = 362,880** possible states (though many are unreachable). In practice, the A* algorithm will find a solution much more quickly due to the heuristic guiding the search.

Space Complexity:

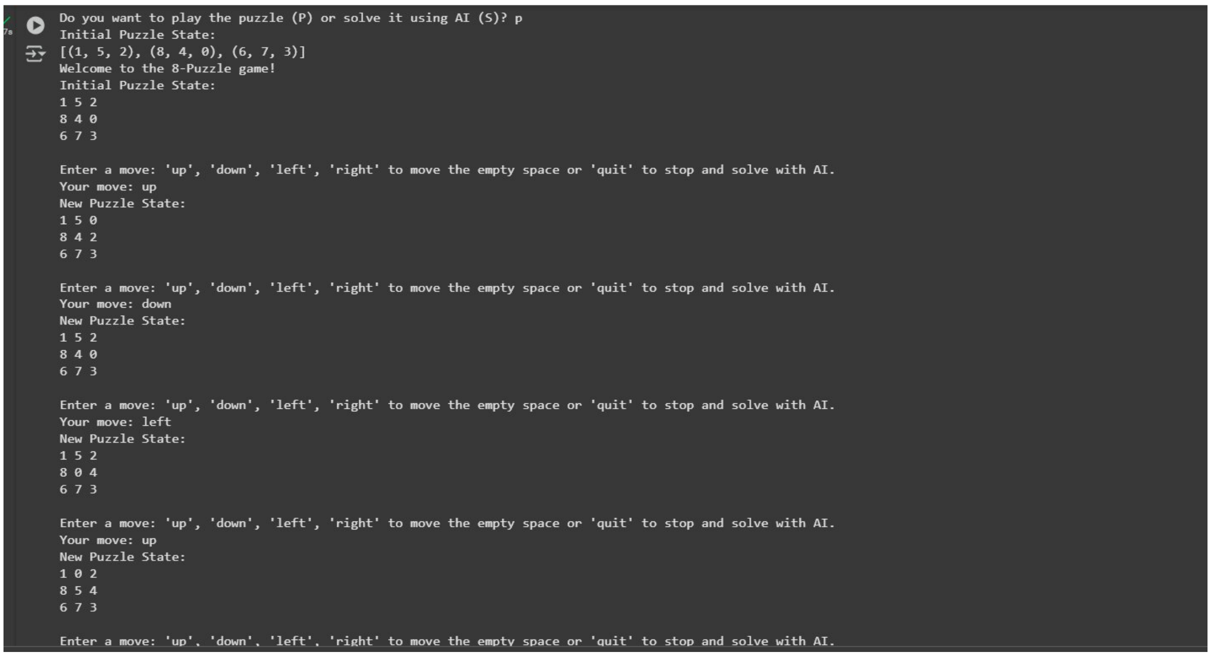
The space complexity is dominated by the storage of the open and closed lists (priority queue and visited states). The number of states stored at any given time is proportional to the number of states the algorithm needs to explore.

Conclusion

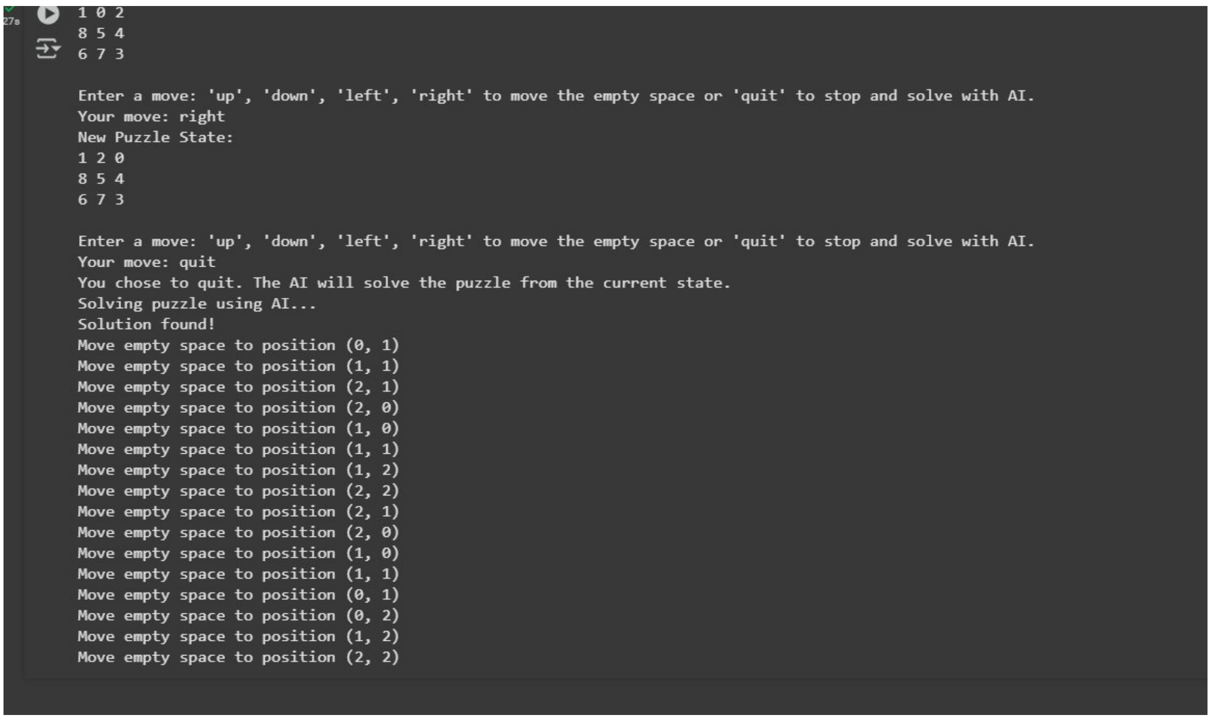
The 8-puzzle solver using A* search and the Manhattan distance heuristic is an efficient and elegant solution to a classical AI problem. The AI can solve the puzzle from any solvable start state, and the user can interact with the puzzle either by playing it manually or letting the AI solve it. The solution illustrates the power of heuristic search algorithms like A* in solving combinatorial puzzles efficiently.

Future improvements could include experimenting with different heuristics, such as **misplaced tiles**, or implementing more advanced search techniques such as **Iterative Deepening A*** or **ID-A*** for even larger puzzle sizes.

Here are some pictures of the output:



```
Do you want to play the puzzle (P) or solve it using AI (S)? p
Initial Puzzle State:
[(1, 5, 2), (8, 4, 0), (6, 7, 3)]
Welcome to the 8-Puzzle game!
Initial Puzzle State:
1 5 2
8 4 0
6 7 3

Enter a move: 'up', 'down', 'left', 'right' to move the empty space or 'quit' to stop and solve with AI.
Your move: up
New Puzzle State:
1 5 0
8 4 2
6 7 3

Enter a move: 'up', 'down', 'left', 'right' to move the empty space or 'quit' to stop and solve with AI.
Your move: down
New Puzzle State:
1 5 2
8 4 0
6 7 3

Enter a move: 'up', 'down', 'left', 'right' to move the empty space or 'quit' to stop and solve with AI.
Your move: left
New Puzzle State:
1 5 2
8 0 4
6 7 3

Enter a move: 'up', 'down', 'left', 'right' to move the empty space or 'quit' to stop and solve with AI.
Your move: up
New Puzzle State:
1 0 2
8 5 4
6 7 3

Enter a move: 'up', 'down', 'left', 'right' to move the empty space or 'quit' to stop and solve with AI.
```


```
1 0 2
8 5 4
6 7 3

Enter a move: 'up', 'down', 'left', 'right' to move the empty space or 'quit' to stop and solve with AI.
Your move: right
New Puzzle State:
1 2 0
8 5 4
6 7 3

Enter a move: 'up', 'down', 'left', 'right' to move the empty space or 'quit' to stop and solve with AI.
Your move: quit
You chose to quit. The AI will solve the puzzle from the current state.
Solving puzzle using AI...
Solution found!
Move empty space to position (0, 1)
Move empty space to position (1, 1)
Move empty space to position (2, 1)
Move empty space to position (2, 0)
Move empty space to position (1, 0)
Move empty space to position (1, 1)
Move empty space to position (1, 2)
Move empty space to position (2, 2)
Move empty space to position (2, 1)
Move empty space to position (2, 0)
Move empty space to position (1, 0)
Move empty space to position (1, 1)
Move empty space to position (0, 1)
Move empty space to position (0, 2)
Move empty space to position (1, 2)
Move empty space to position (2, 2)
```