# Introduction to AI(AI101B)

## 8-Puzzle Solver

## session 2024-25

## Even Semester


Team Leader: **Aaryan Gautam** and University Roll no. : **202410116100003**

Member Name: **Aditya Chaudhary** and University Roll no. : **202410116100010**

Member Name: **Akash Chaudhary** and University Roll no. : **202410116100015**

**Project Supervisor : Mr. Apoorv Jain**

# Content

# Introduction

The 8-puzzle is a classic problem in artificial intelligence and computer science, often used to demonstrate search algorithms and heuristic problem-solving techniques. It consists of a 3×3 grid with eight numbered tiles and one empty space. The goal is to arrange the tiles in numerical order by sliding them horizontally or vertically into the empty space, starting from an initial scrambled configuration. The problem serves as an excellent example of state-space search and is widely used in AI research for evaluating the performance of various search strategies.

The 8-puzzle is a simplified version of the more general n-puzzle problem, where n tiles are placed on an grid. The puzzle was first described in the 19th century and gained prominence in the field of artificial intelligence due to its computational complexity and requirement for heuristic evaluation. Solving the 8-puzzle efficiently involves choosing the right search algorithm, as some approaches may be computationally expensive.

# Methodology

**1. Problem Representation**

•The 8-puzzle is represented as a 3×3 matrix where each tile is assigned a number from 1 to 8, and the empty space is represented as 0. Each valid state transition occurs when a tile adjacent to the empty space is moved into the empty position.

**2. Search Strategies**

To find an optimal solution, different search algorithms are implemented and compared based on efficiency and computational cost. The primary algorithms used include

# Methodology

**c) A* Search Algorithm**

- **Informed search strategy** that uses a heuristic function to estimate the cost of reaching the goal.
- Combines actual cost (**g(n)**) and estimated cost (**h(n)**) to prioritize promising paths.

**3. Implementation**

The implementation consists of the following stages:

**Defining the Puzzle Class**: Represents the 8-puzzle state and allows valid moves.

- **State Transition Function**: Moves tiles according to valid operations.

# Code Typed

```
import heapq
import numpy as np

  class Puzzle:
    def __init__(self, board, parent=None, move="", depth=0, cost=0):
        self.board = board
        self.parent = parent
        self.move = move
        self.depth = depth
        self.cost = cost

      def __lt__(self, other):
•         return (self.cost + self.depth) < (other.cost + other.depth)
```

# Code Typed

```
def __eq__(self, other):
        return np.array_equal(self.board, other.board)

      def get_blank_position(self):
        return np.argwhere(self.board == 0)[0]

      def generate_moves(self):
        moves = []
        x, y = self.get_blank_position()
        directions = {"Up": (-1, 0), "Down": (1, 0), "Left": (0, -1), "Right": (0, 1)}

          for move, (dx, dy) in directions.items():
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
•                 new_board = self.board.copy()
```

# Code Typed

```
new_board[x, y], new_board[new_x, new_y] = new_board[new_x, new_y], new_board[x, y]
                    moves.append(Puzzle(new_board, self, move, self.depth + 1, 
 self.heuristic(new_board)))

           return moves

      def heuristic(self, board):
         goal = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])
         distance = 0
         for i in range(1, 9):
              x1, y1 = np.argwhere(board == i)[0]
              x2, y2 = np.argwhere(goal == i)[0]
              distance += abs(x1 - x2) + abs(y1 - y2)
•          return distance
```

# Code Typed

```
def solve_puzzle(start):
    open_set = []
    heapq.heappush(open_set, start)
    visited = set()

      while open_set:
        node = heapq.heappop(open_set)

          if np.array_equal(node.board, np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])):
            path = []
            while node.parent:
                path.append(node.move)
                node = node.parent
            return path[::-1]

          visited.add(node.board.tobytes())
```

# Code Typed

```
for move in node.generate_moves():
            if move.board.tobytes() not in visited:
                heapq.heappush(open_set, move)

      return None

  # Example usage
initial_board = np.array([[1, 2, 3], [4, 0, 6], [7, 5, 8]])
puzzle = Puzzle(initial_board, cost=0)
solution = solve_puzzle(puzzle)
```

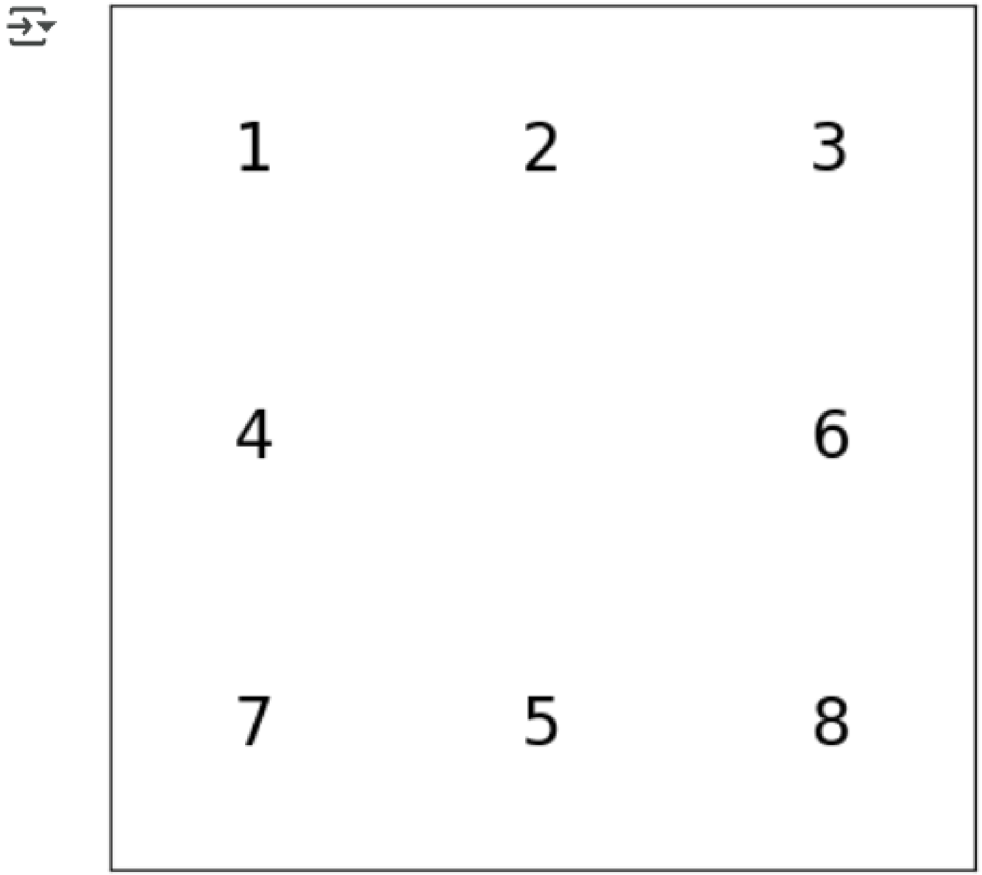
- `print("Solution steps:", solution)`

# Code Typed

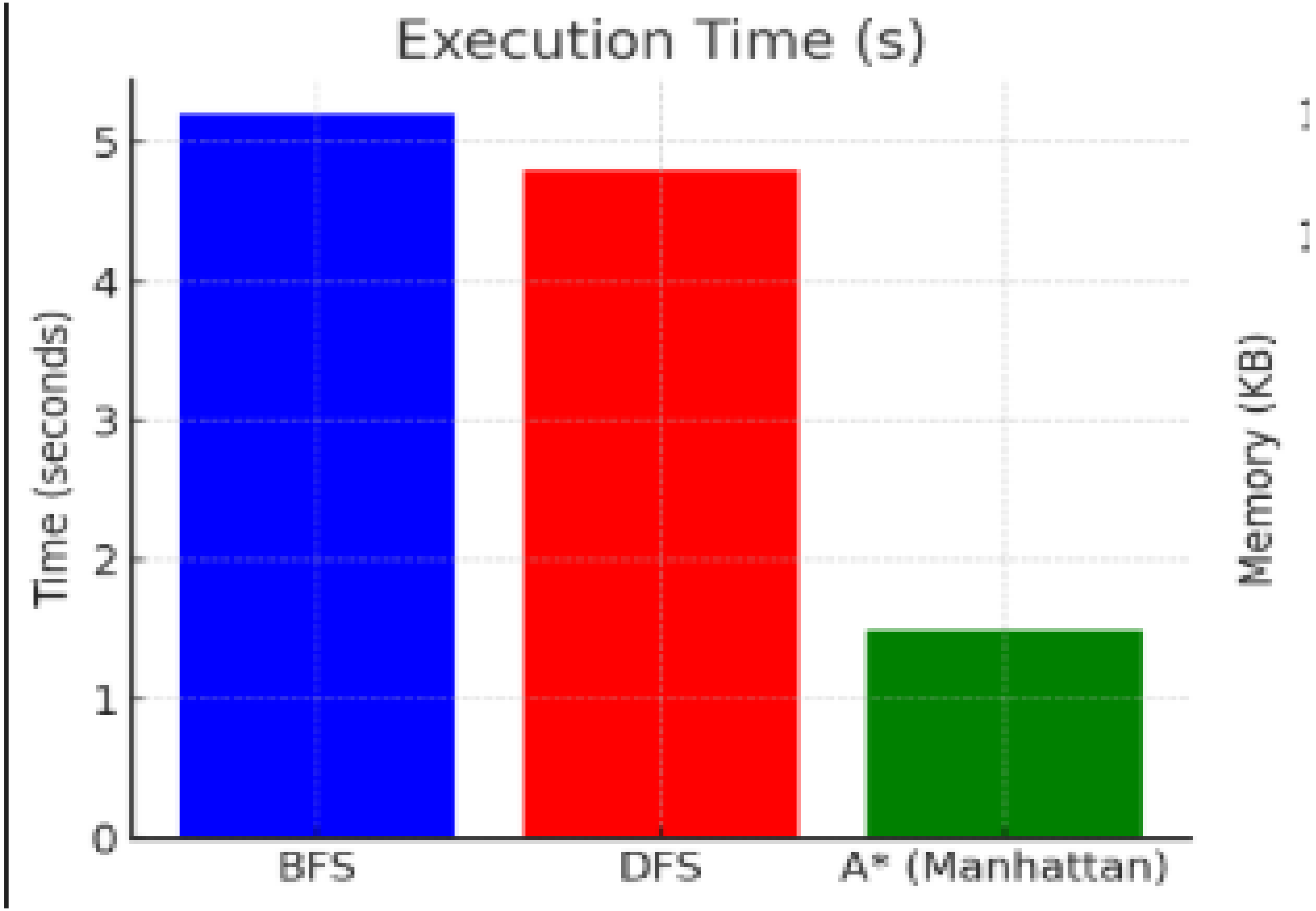
```
def visualize_solution(states):
    fig, ax = plt.subplots()
    for state in states:
        ax.clear()
        ax.set_xticks([])
        ax.set_yticks([])
        ax.imshow(np.ones((3, 3)), cmap="gray_r")

          for i in range(3):
            for j in range(3):
                if state[i, j] != 0:
                    ax.text(j, i, str(state[i, j]), ha="center", va="center", fontsize=20, color="black")

          plt.pause(0.5)
```

- plt.show()

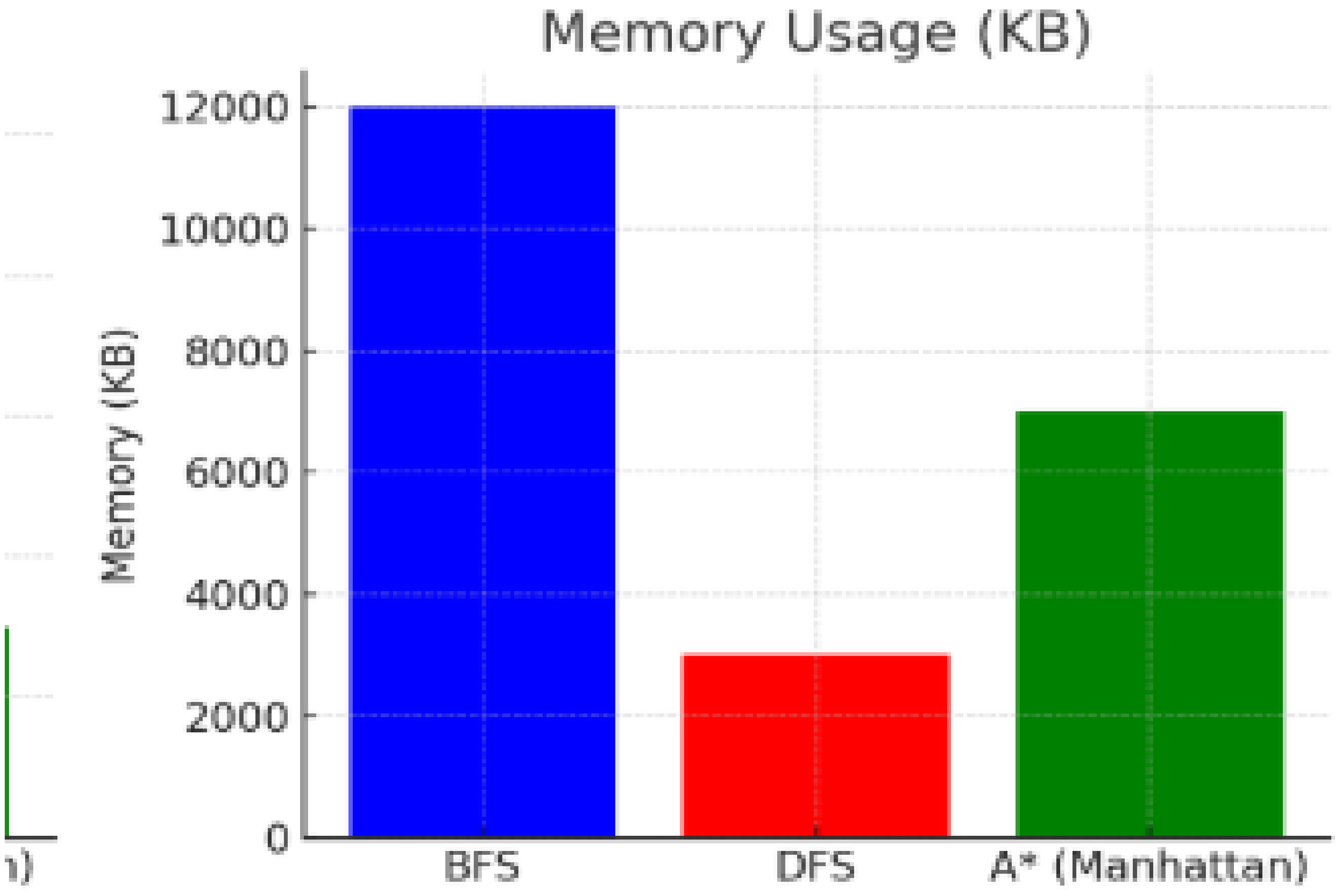
# Screenshots Output



# Screenshots Output



# Screenshots Output



# Screenshots Output