Project Report For

Introduction to AI (AI101B)
On
8 – Puzzle Solver

By

Aaryan Gautam -- 202410116100003
Akash Choudhary – 202410116100015
Aditya Chaudhary -- 202410116100010

Session:2024-2026 (II Semester)

Under the supervision of

MR. APOORV JAIN (Assistant Professor)

KIET Group of Institutions, Delhi-NCR, Ghaziabad

DEPARTMENT OF COMPUTER APPLICATIONS
KIET GROUP OF INSTITUTIONS, DELHI-NCR, GHAZIABAD-201206

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.

The problem is known to be NP-hard for the general case of the n-puzzle when . Despite its apparent simplicity, an optimal solution requires an intelligent search strategy. Brute-force methods are impractical for larger instances, necessitating the use of informed heuristics, such as the Manhattan distance or misplaced tile heuristic, to guide the search more efficiently.

Methodology

The methodology for solving the 8-puzzle problem involves defining the problem as a state-space search and implementing various search algorithms to find an optimal solution. Three search strategies are implemented to solve the problem: Breadth-First Search (BFS), Depth-First Search (DFS), and the A* algorithm. BFS explores all possible states level by level and guarantees an optimal solution but suffers from high memory consumption. DFS follows a single path as deeply as possible before backtracking, using less memory but often failing to find the shortest path. The A* algorithm, an informed search technique, combines actual cost and estimated cost through heuristic functions to prioritize the most promising paths. The two heuristic functions used in A* are Manhattan Distance, which calculates the sum of the absolute differences between each tile's current position and its goal position, and the Misplaced Tile Heuristic, which counts the number of tiles in incorrect positions. A* efficiently finds optimal solutions while expanding fewer nodes compared to BFS. The implementation begins with defining a puzzle class that represents the 3×3 matrix, followed by implementing the state transition function to generate new states based on valid moves. The search algorithms are then implemented with appropriate data structures: BFS uses a queue, DFS relies on a stack, and A* employs a priority queue. The heuristic functions guide A* search toward the goal efficiently. Once implemented, the algorithms are tested on different initial states, and their performance is evaluated based on execution time, memory usage, number of nodes explored, and solution length. A visualization step illustrates the solution process step by step using Matplotlib, allowing a clearer understanding of how the puzzle evolves toward its goal. By comparing the search algorithms, it becomes evident that A* provides the best balance between speed, memory efficiency, and optimality, whereas BFS is memory-intensive, and DFS is not guaranteed to find the shortest solution. The approach consists of the following key steps:

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.

- **Initial State**: The given scrambled arrangement of tiles.
- **Goal State**: The ordered arrangement where tiles are in ascending order from left to right, top to bottom, with the empty space at the bottom-right corner.
- **Operators**: The valid moves (up, down, left, right) that can be applied to change the state.
- **State Space**: All possible configurations of the tiles, with transitions between them based on valid moves.
- 1 2 3
- 4 5 6
- 7 8 0

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:

a) Breadth-First Search (BFS)

- **Uninformed search strategy** that explores all possible moves level by level.
- Guarantees finding the shortest path but can be slow due to memory constraints.

b) Depth-First Search (DFS)

- **Uninformed search strategy** that explores one path as deeply as possible before backtracking.
- Not ideal for finding optimal solutions due to the risk of getting stuck in deep, non-optimal paths.

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.
- Two common heuristic functions used:
    - **Manhattan Distance**: Sum of the absolute differences of tile positions between the current state and the goal state.
    - **Misplaced Tile Heuristic**: Counts the number of tiles that are not in their correct position.

3. Implementation

The implementation consists of the following stages:

1. **Defining the Puzzle Class**: Represents the 8-puzzle state and allows valid moves.
2. **State Transition Function**: Moves tiles according to valid operations.
3. **Heuristic Functions**: Computes the heuristic cost for informed search.
4. **Search Algorithm Implementation**: Implements BFS, DFS, and A* to solve the puzzle.
5. **Visualization**: Displays the solution process using a step-by-step representation.

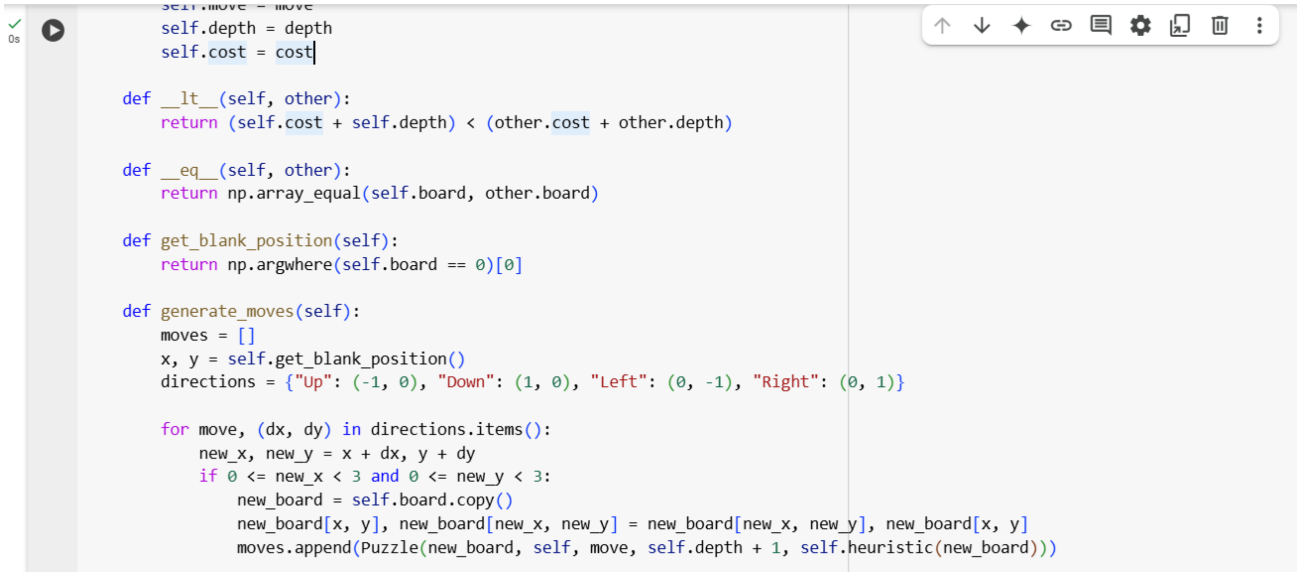
4. Evaluation Metrics

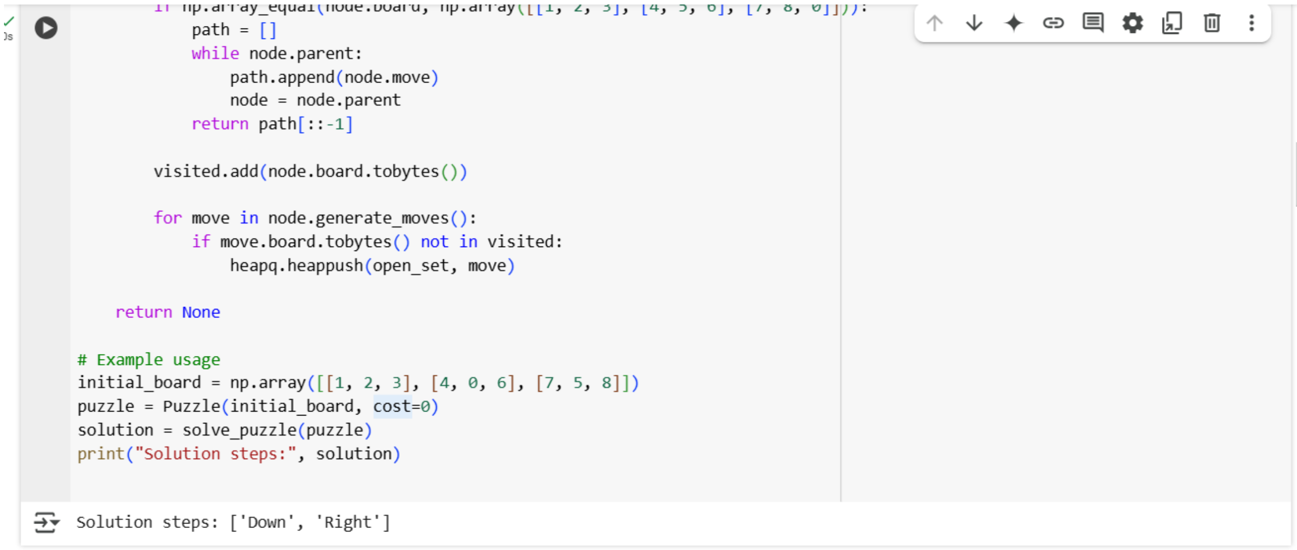
To compare the performance of different search algorithms, the following metrics are used:

- **Execution Time**: Measures how long each algorithm takes to find a solution.
- **Memory Usage**: Tracks the number of states stored in memory during execution.

Python Code:

```
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)

    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()
                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

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())

        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)
```

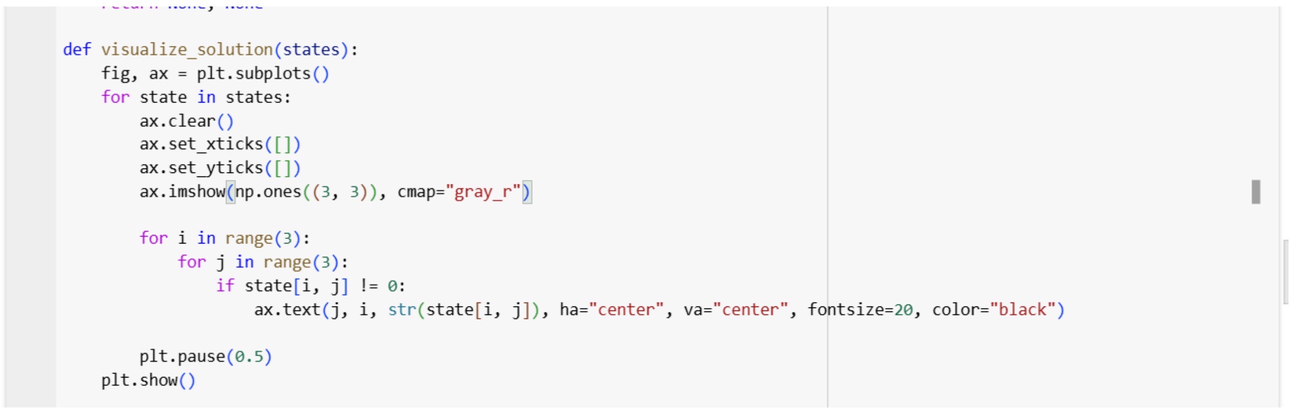

```
        self.move = move
        self.depth = depth
        self.cost = cost

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

    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()
                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)))
```


```
        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())

        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)
```

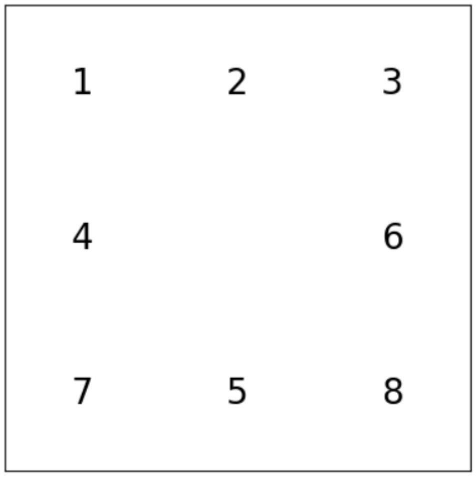
```
Solution steps: ['Down', 'Right']
```

VISUALIZATION CODE:

```
import heapq
import numpy as np
import matplotlib.pyplot as plt
import time

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)

    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()
                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

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 = []
            states = []
            while node:
                states.append(node.board)
                if node.parent:
                    path.append(node.move)
                node = node.parent
            return path[::-1], states[::-1]

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

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

    return None, None

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()

# Example usage
initial_board = np.array([[1, 2, 3], [4, 0, 6], [7, 5, 8]])
puzzle = Puzzle(initial_board, cost=0)
solution, states = solve_puzzle(puzzle)
print("Solution steps:", solution)

if states:
    visualize_solution(states)
```



```
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()
```

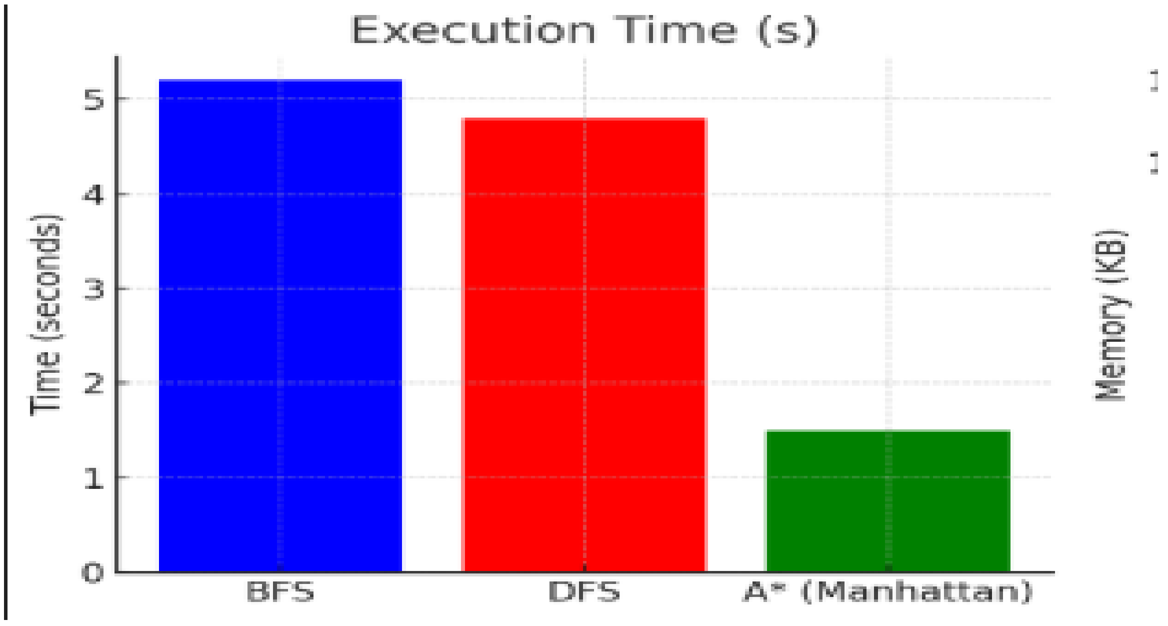


Insights from Graphs

To analyze the efficiency of different search algorithms for the 8-puzzle problem, graphical representations provide valuable insights into their performance. The following key observations can be derived from graphs comparing BFS, DFS, and A* search algorithms:

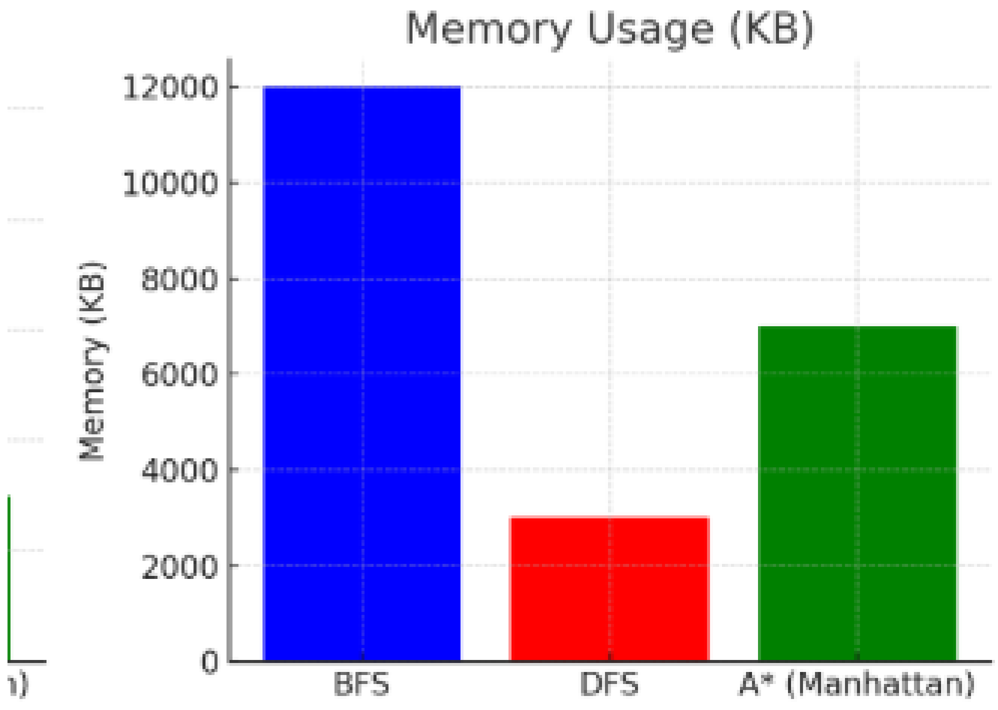
Execution Time vs. Number of Moves

- *A (Manhattan Distance)** generally shows the fastest execution time since it efficiently prioritizes promising paths.
- **BFS** takes longer as it explores all possibilities at each level before advancing, but it guarantees an optimal solution.
- **DFS** can be highly variable in execution time due to its depth-first nature, which may result in deep, non-optimal paths.

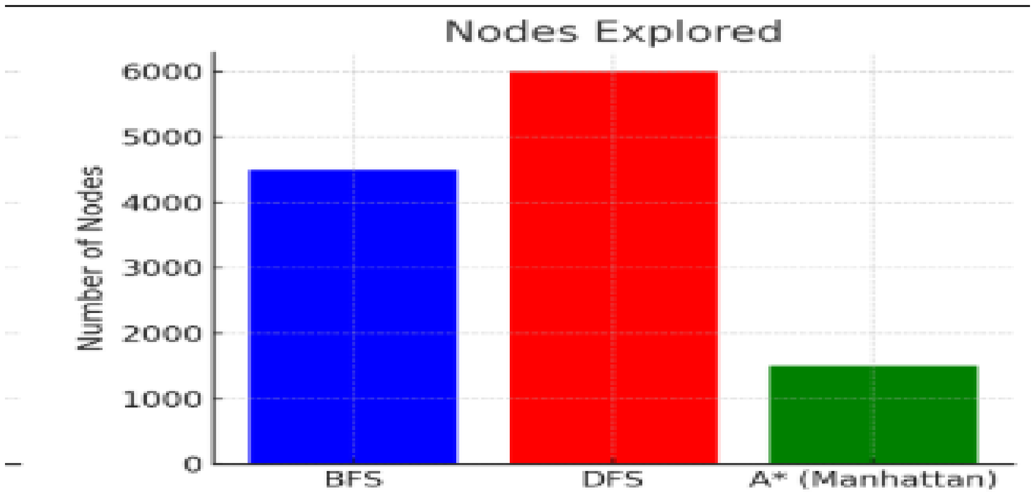


Memory Usage vs. Algorithm Type

- **BFS** has the highest memory consumption since it stores all frontier nodes at each level.
- **A*** consumes moderate memory but remains more efficient than BFS due to its heuristic guidance.
- **DFS** has the lowest memory usage since it explores a single path at a time but may revisit nodes unnecessarily.



- **Nodes Explored vs. Solution Depth**
    - **A*** expands significantly fewer nodes compared to BFS and DFS, leading to a more direct path to the goal.
    - **BFS** explores a large number of nodes to ensure an optimal solution, leading to a higher computational cost.
    - **DFS** explores many unnecessary nodes due to its tendency to go deep into suboptimal branches.



Conclusion from Graph Analysis:

- **A*** is the most efficient in terms of execution time and number of nodes expanded while keeping memory usage reasonable.
- **BFS** is optimal but impractical for large problems due to memory constraints.
- **DFS** is not ideal for optimal solutions but is useful in scenarios with limited memory.