AI Lab 1: 8-Puzzle Game

Mohamed Tarek Hussein - 21011161
Asser Ossama Elzeki - 21010241
Youssef Tarek Hussien - 21011595

Introduction

In this report, we explore various search algorithms for solving the 8-puzzle game, including:

- A* Search Algorithm
- BFS (Breadth-First Search) Algorithm
- DFS Search Algorithm
- Iterative DFS Search Algorithm
- Algorithm Comparison
- Algorithm Analysis
- User Manual

Searching Algorithms

---

1 A* Search Algorithm

This module implements the **A* (A-star)** search algorithm for solving the 8-puzzle problem (sliding tile puzzle). The algorithm explores states by minimizing the total cost function $f(n) = g(n) + h(n)$, where:

- $g(n)$: Cost to reach the current state from the start state.
- $h(n)$: Heuristic cost (estimated cost) to reach the goal state from the current state.

Currently, this module supports **Manhattan** and **Euclidean** distance heuristics. Additional search algorithms (such as Breadth-First Search, Depth-First Search, and Uniform-Cost Search) will be added soon.

Class: `AStar`

Parameters:
- `start_state` (str): The starting state of the 8-puzzle, represented as a string (e.g., `'123456780'` where '0' is the blank space).
- `goal_state` (str): The target goal state to reach (default is `'012345678'`).
- `heuristic` (str): The heuristic function to use. Options are `'Manhattan'` or `'Euclidean'`. Default is `'Manhattan'`.

Attributes:
- `explored_nodes` (int): Tracks the number of nodes (states) explored during the search.
- `search_depth` (int): The maximum depth reached during the search.
- `total_time` (float): Total time taken by the algorithm in seconds.

Methods

`run() -> tuple`

Executes the A* search algorithm and returns the path to the goal and the total cost.

- **Returns:** A tuple containing:
  - `path` (list): The sequence of states leading to the goal (or `None` if no path is found).
  - `cost` (float): The total cost to reach the goal (or `float('inf')` if no path is found).

`get_neighbors(state: str) -> list`

Generates the neighboring states by moving the blank space ('0') up, down, left, or right.

- **Returns:** A list of tuples where each tuple contains:
  - `neighbor_state` (str): A neighboring state of the puzzle.
  - `cost` (int/float): The cost to reach this neighbor.

`swap(state: str, i: int, j: int) -> str`

Swaps two characters in the puzzle state to simulate tile movement.

- **Returns:** A new string with the characters at indices `i` and `j` swapped.

`get_cost(state: str) -> float`

Calculates the heuristic cost (Manhattan or Euclidean distance) of a given state.

- **Returns:** The heuristic cost to reach the goal from the given state.

`manhattan_distance(state: str) -> int`

Calculates the Manhattan distance between the current state and the goal state.

- **Returns:** The Manhattan distance as an integer.

`euclidean_distance(state: str) -> float`

Calculates the Euclidean distance between the current state and the goal state.

- **Returns:** The Euclidean distance as a floating-point number.

`get_path(came_from: dict, current: str) -> list`

Reconstructs the path from the start state to the goal state.

- **Returns:** A list of states representing the path to the goal.

`get_info() -> dict`

Returns statistics about the search process, such as the number of explored nodes, total execution time, and the maximum search depth.

- **Returns:** A dictionary with the following keys:
  - `'explored_nodes'`: Number of nodes explored during the search.
  - `'total_time'`: Total time taken for the search (in seconds).
  - `'search_depth'`: The maximum depth reached during the search.

Example Using Euclidean heuristic

This is an example that demonstrates how to use the A* search algorithm. The puzzle starts with a state `'123456780'` and aims to reach the goal state `'012345678'` using the **Euclidean distance** heuristic.

```
# Initialize the A* algorithm with start and goal states
, and the Euclidean heuristic
astar = AStar('041586732', '012345678', 'Euclidean')

# Run the algorithm
path, cost = astar.run()

# Print the resulting path and cost
print(path)  # Outputs: ['401586732', '410586732', '415086732',...]
print(cost)  # Outputs: 147.631

# Print additional information about the search process
print(astar.get_info())
```

```
# Outputs: {'explored nodes': 11961, 'total time': 0.178,
'max-search-depth': 24, 'cost': 147.631}
```

Example Using Manhattan heuristic

If we used manhattan heuristic instead we will get:

```
# Initialize the A* algorithm with start and goal states
, and the Manhattan heuristic
astar = AStar('041586732', '012345678', 'Manhattan')

# Run the algorithm
path, cost = astar.run()

# Print the resulting path and cost
print(path)  # Outputs: ['541086732', '541786032', '541786302',...]
print(cost)  # Outputs: 171

# Print additional information about the search process
print(astar.get_info())
# Outputs: {'explored nodes': 8892, 'total time': 0.079,
'max-search-depth': 22, 'cost': 171}
```

Class: `BFS`

Parameters:

- `start_state` (str): The starting state of the 8-puzzle, represented as a string (e.g., `'123456780'` where '0' is the blank space).
- `goal_state` (str): The target goal state to reach (default is `'012345678'`).
- `heuristic` (str): Not used in BFS, but included for compatibility with other algorithms.

2 BFS (Breadth-First Search) Algorithm

Class: `BFS`

Parameters:

- `start_state` (str): The starting state of the 8-puzzle, represented as a string (e.g., `"123456780"`, where `"0"` is the blank space).
- `goal_state` (str): The target goal state to reach (default is `"012345678"`).
- `heuristic` (str): Not used in BFS but included for compatibility with other algorithms.

Attributes:

- `explored_nodes` (int): Tracks the number of nodes (states) explored during the search.
- `search_depth` (int): The maximum depth reached during the search.
- `total_time` (float): Total time taken by the algorithm in seconds.

Methods:

`run()`

Executes the BFS search algorithm and returns the path to the goal and the total cost.

- **Returns:**
  - A tuple containing:
    - `path` (list): The sequence of states leading to the goal (or `None` if no path is found).
    - `cost` (float): BFS always returns a cost of 0, as it focuses on finding the shortest path without a heuristic cost.

`get_neighbors(state: str)`

Generates the neighboring states by moving the blank space ('0') up, down, left, or right.

`swap(state: str, i: int, j: int)`

Swaps two characters in the puzzle state to simulate tile movement.

`get_path(came_from: dict, current: str)`

Reconstructs the path from the start state to the goal state.

`get_info()`

Returns statistics about the search process, such as the number of explored nodes, total execution time, and the maximum search depth.

Example Usage:

```
# Initialize the BFS algorithm with start and goal states
bfs = BFS('041586732', '012345678')

# Run the algorithm
path = bfs.run()

# Print the resulting path and cost
print(path)
```

```
print(bfs.get_info())
outputs: {'explored nodes': 94467, 'total time': 0.391,
'max search depth': 22}
```

—

3 DFS Search Algorithm

Class: `DFS`

Parameters:

- `start_state` (str): The initial state of the 8-puzzle, represented as a string (e.g., `"123456780"` where `"0"` is the blank space).
- `goal_state` (str): The goal state of the puzzle (default is `"012345678"`).
- `heuristic` (str): Not used in DFS but included for consistency with other algorithms.

Attributes:

- `explored_nodes` (int): Tracks the number of explored nodes.
- `search_depth` (int): Tracks the maximum depth reached during the search.
- `total_time` (float): Total time taken for the search in seconds.

Methods:

`run()`

Executes the DFS algorithm and returns the path to the goal if found, or `None` if there is no solution.

`get_neighbors(state: str)`

Generates neighboring states by moving the blank tile up, down, left, or right.

`swap(state: str, i: int, j: int)`

Swaps two characters in the puzzle state to simulate tile movement.

`get_path(came_from: dict, current: str)`

Reconstructs the path from the start state to the goal state using a dictionary of predecessors.

`get_info()`

Returns a dictionary with statistics about the search, including the number of explored nodes, total execution time, and the maximum search depth.

Example Usage:

```
# Initialize the DFS algorithm with start and goal states
dfs = DFS('041586732', '012345678')

# Run the algorithm
path = dfs.run()

# Print the resulting path and search statistics
print(path)
print(dfs.get_info())
outputs: {'explored nodes': 107390, 'total time': 0.201,
'max search depth': 70180}
```

—

4 Iterative DFS Search Algorithm

Class: `IT_DFS`

Parameters:

- `start_state` (str): The initial state of the 8-puzzle, represented as a string (e.g., `"123456780"` where `"0"` is the blank space).
- `goal_state` (str): The goal state of the puzzle (default is `"012345678"`).
- `heuristic` (str): Not used in IDDFS but included for consistency with other algorithms.

Attributes:

- `explored_nodes` (int): Tracks the number of explored nodes.
- `search_depth` (int): Tracks the maximum depth reached during the search.
- `total_time` (float): Total time taken for the search in seconds.

Methods:

`run()`

Executes the IDDFS algorithm and returns the path to the goal if found, or `None` if there is no solution.

`get_neighbors(state: str)`

Generates neighboring states by moving the blank tile up, down, left, or right.

`swap(state: str, i: int, j: int)`

Swaps two characters in the puzzle state to simulate tile movement.

`get_path(came_from: dict, current: str)`

Reconstructs the path from the start state to the goal state using a dictionary of predecessors.

`get_info()`

Returns a dictionary with statistics about the search, including the number of explored nodes, total execution time, and the maximum search depth.

Example Usage:

```
# Initialize the IDDFS algorithm with start and goal states
it_dfs = IT_DFS('041586732', '012345678')

# Run the algorithm
path = it_dfs.run()

# Print the resulting path and search statistics
print(path)
print(it_dfs.get_info())
outputs: {'explored nodes': 167450, 'total time': 0.16,
'max search depth': 28}
```

Algorithm Comparison

The table below compares the five algorithms based on their performance in solving the 8-puzzle problem for the test case with start state `'041586732'` and goal state `'012345678'`.

Algorithm	Explored Nodes	Total Time (s)	Max Search Depth	Total Cost
A* (Euclidean)	11961	0.178	24	147.631
A* (Manhattan)	8892	0.079	22	171
BFS	94467	0.391	22	0
DFS	107390	0.201	70180	0
IDDFS	167450	0.16	28	0

Table 1: Comparison of 8-Puzzle Solving Algorithms

Algorithm Analysis

The table in Table 1 provides an overview of five algorithms used to solve the 8-puzzle problem, based on their performance in terms of nodes explored, execution time, maximum search depth, and path cost. Here's a detailed comparison:

- **A* with Manhattan Heuristic:** Among the algorithms, A* with the Manhattan heuristic is the most efficient, exploring only 8,892 nodes and taking 0.079 seconds to reach the solution with a search depth of 22. This is due to the heuristic's accuracy in estimating the cost to the goal, making it computationally efficient for this problem. A* has a theoretical time complexity of $O(b^d)$ and memory complexity of $O(b^d)$, where b is the branching factor, and d is the solution depth. A*'s efficiency largely depends on the quality of its heuristic; with the Manhattan distance, it reaches the goal quickly with relatively low memory usage.

- **A* with Euclidean Heuristic:** While still efficient, A* with the Euclidean heuristic explores slightly more nodes (11,961) and takes 0.178 seconds, making it slower than the Manhattan heuristic. This is likely due to the Euclidean distance being less accurate in estimating the actual moves required in a grid-based puzzle. Similar to the Manhattan heuristic, A* with the Euclidean heuristic also has a time and memory complexity of $O(b^d)$.

- **Breadth-First Search (BFS):** BFS guarantees the shortest path but is memory-intensive and has a high time complexity of $O(b^d)$, making it impractical for large state spaces. In this example, BFS explored 94,467 nodes and took 0.391 seconds. Its memory complexity is also $O(b^d)$, as it stores every node at each level. While BFS ensures an optimal solution, its large memory footprint and time make it less efficient compared to A* for this problem.

- **Depth-First Search (DFS):** DFS has a lower memory requirement than BFS, with memory complexity $O(b \times m)$, where m is the maximum depth of the search tree. However, it is not optimal and can get trapped in deep branches, especially in problems with a high branching factor. Here, DFS explored 107,390 nodes, and although its memory usage was lower, it reached a maximum search depth of 70,180, making it highly inefficient. Its time complexity is $O(b^m)$, where m is the maximum depth, which can be problematic in deep or expansive search spaces like the 8-puzzle.

- **Iterative Deepening Depth-First Search (IDDFS):** IDDFS combines the space efficiency of DFS with the completeness of BFS, with time complexity $O(b^d)$ and memory complexity $O(b \times d)$. Although it explores nodes repeatedly at each depth, it ultimately finds a solution using less memory than BFS. In this case, IDDFS explored 167,450 nodes, took 0.16 seconds, and reached a maximum depth of 28. While slower than BFS,

it's more memory-efficient, making it a reasonable choice for large search spaces without strong heuristic guidance.

Summary: Based on the analysis, A* with the Manhattan heuristic is the most effective algorithm for solving the 8-puzzle due to its balance of speed, optimality, and memory usage. BFS, while optimal, is less practical due to its high memory usage, and DFS is less suitable because of its inefficiency in large, deep search spaces. IDDFS is a good compromise when optimality is not critical but memory efficiency is needed.

User Manual

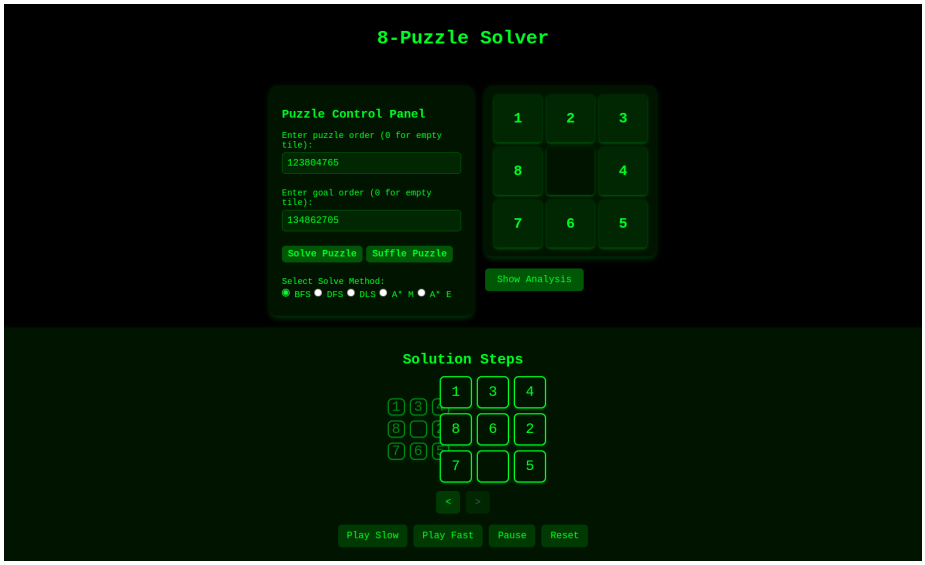

Figure 1: User Manual Overview

Puzzle Control Panel

Initial State

- Enter the starting configuration of the puzzle in row-wise order from left to right.
- Validation ensures each number from 0 to 8 is present exactly once.

Goal State

- Enter the desired final configuration of the puzzle in row-wise order from left to right.

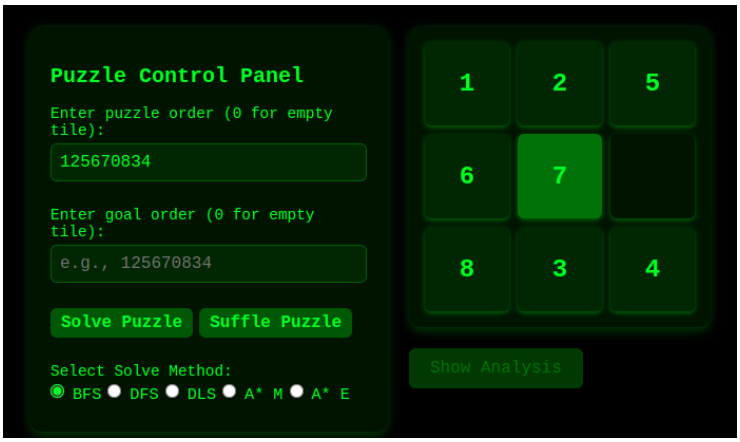

Figure 2: Puzzle Control Panel

- Validation ensures each number from 0 to 8 appears only once.
- The default goal state is 123456780.

Shuffle Puzzle

- Shuffles the initial input, creating a randomized, solvable configuration.

Interactive Puzzle

- Allows you to manually rearrange the puzzle tiles, setting a new initial state for solving.

Method Selection

- Choose a solving algorithm for the puzzle from the following options:
  - BFS (Breadth-First Search)
  - FS (Forward Search)
  - IDS (Iterative Deepening Search)
  - A* with Manhattan Distance
  - A* with Euclidean Distance
- Select your preferred method using the radio buttons.

Solve Puzzle

- Sends the selected algorithm, initial state, and goal state to the backend server for processing and retrieves the solution path.

Solution Area

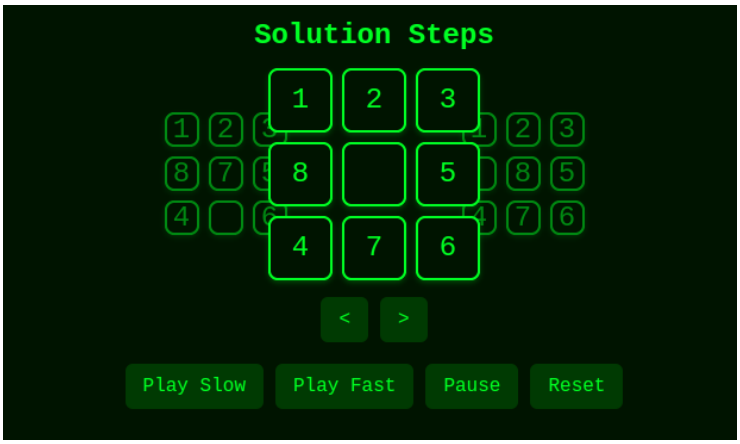

Figure 3: Solution Area

Pause

- Pauses and resumes the solution playback, providing a controlled view of each solving step.

Next Step / Previous Step

- Step through the solution one move at a time, either forward or backward.

Reset

- Resets the puzzle to its initial configuration, allowing you to restart the solution process or choose a new method.

Play Fast / Play Slow

- **Play Fast:** Quickly displays solution steps for a rapid view of the solving sequence.
- **Play Slow:** Plays the solution steps at a slower pace, making it easier to follow each move.

Analysis Tab

Displays detailed metrics on the solution once the algorithm has finished running. Information provided includes:

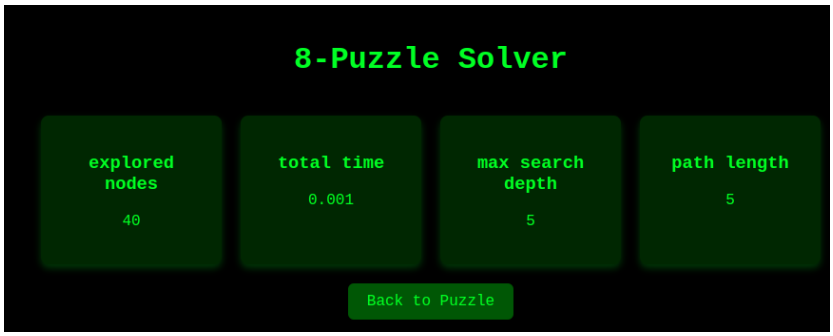

Figure 4: Analysis Tab

- **Maximum Depth:** The deepest level of the puzzle explored during solving.
- **Nodes Explored:** The total number of nodes analyzed by the algorithm.
- **Path Length:** The number of steps in the final solution path.

This section helps to evaluate the algorithm's efficiency and performance for the given puzzle.