Using Informed and Uninformed Search Algorithms to Solve 8-Puzzle - AI Assignment 1

Abdelrahman Abouroumia 8368
Zeyad Hesham 8226
Kyrollos Yousef 8057

1 Introduction

The 8-puzzle is a classic problem in artificial intelligence, where the goal is to move tiles on a 3x3 board to reach a specified end configuration. This report explores the use of various search algorithms to solve the 8-puzzle problem, including both informed and uninformed search strategies.

2 Algorithms and Data Structures

The following algorithms were implemented to solve the 8-puzzle problem:

2.1 Breadth-First Search (BFS)

BFS is an uninformed search algorithm that explores all nodes at the present depth level before moving on to nodes at the next depth level. It uses a queue data structure to keep track of the frontier.

2.2 Depth-First Search (DFS)

DFS is an uninformed search algorithm that explores as far as possible along each branch before backtracking. It uses a stack data structure to manage the frontier.

2.3 Iterative Deepening Depth-First Search (IDDFS)

IDDFS combines the space efficiency of DFS with the optimality of BFS. It performs a series of depth-limited searches, increasing the depth limit with each iteration. However, for puzzles with large minimum paths, IDDFS may not perform optimally due to potential data loss from the frontier caused by system memory management.

2.4 A* Search

A* is an informed search algorithm that uses heuristics to guide its search. It combines the cost to reach the current node and the estimated cost to reach the goal. Two heuristics were implemented:

- **Manhattan Distance**: The sum of the absolute differences between the current and goal positions of each tile.
- **Euclidean Distance**: The straight-line distance between the current and goal positions of each tile.

A* uses a priority queue (heap) to manage the frontier.

3 Implementation Details

The implementation is divided into several files:

3.1 8PUZZLE.py

This is the main implementation file where the puzzle's solvability is checked, and the results of the searches are measured. It uses the `measure_search_time` function to time each search and visualize the results.

3.2 state.py

This file contains the `State` class, which represents the state of the puzzle. It includes methods to find the empty tile and generate neighboring states.

3.3 search_algorithms.py

This file contains the implementations of BFS, DFS, IDDFS, and A* search algorithms.

3.4 heuristics.py

This file contains the Euclidean and Manhattan distance calculations for the A* search. The Manhattan heuristic generally shows fewer nodes expanded to reach the solution.

3.5 data_structures.py

This file contains class implementations for the data structures used by the search algorithms:

- `Stack` for DFS
- `Queue` for BFS
- `Heap` for A*

3.6 visualize.py

This file uses the Graphviz library to visualize the search path to the goal state for each search and prints the search statistics, including the path to the goal, cost of the path, nodes expanded, search depth, and running time.

4 Assumptions and Clarifications

- The puzzle is represented as a 3x3 grid.
- The goal state is defined as `[[0, 1, 2], [3, 4, 5], [6, 7, 8]]`.
- The initial state is checked for solvability before running the search algorithms.
- The `State` class tracks the board configuration, path to the current state, cost, parent state, depth, and heuristic value.

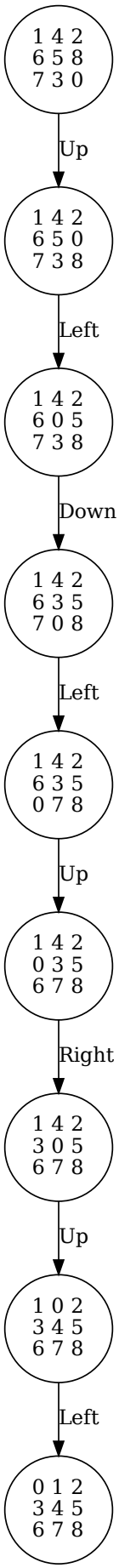
5 Sample Runs

Below are sample runs for each algorithm:

5.1 BFS

```
Initial State: [[1, 4, 2], [6, 5, 8], [7, 3, 0]]
Path: ULDLURUL
Cost of path: 8
Nodes expanded: 188
Max depth: 8
Time taken: 0.0019915103912353516 seconds
```

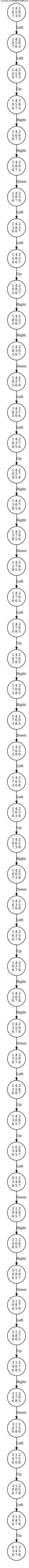
BFS:
Path: ULDLURUL
Cost of path: 8
Nodes expanded: 188
Max depth: 8
Time taken: 0.0018296241760253906 seconds



5.2 DFS

```
Initial State: [[1, 4, 2], [6, 5, 8], [7, 3, 0]]
Path: LLURRDLLURRDLLURRDLLURRDLLURDLURRDLUULDRRDLURDLULU
Cost of path: 50
Nodes expanded: 8090
Max depth: 50
Time taken: 0.054453134536743164 seconds
```

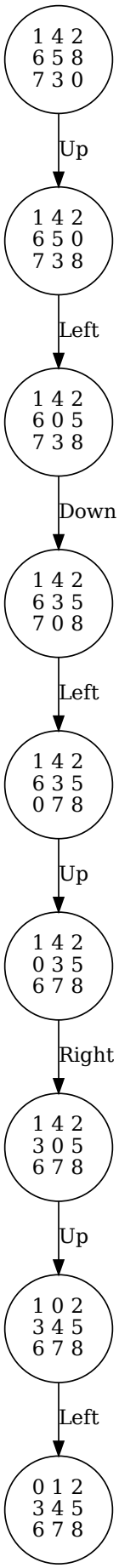
DFS:
Path: LLURRDLLURRDLLURRDLLURRDLLURDLURRDLUULDRRDLURDLULU
Cost of path: 50
Nodes expanded: 8090
Max depth: 50
Time taken: 0.053186893463134766 seconds



5.3 Iterative DFS

```
Initial State: [[1, 4, 2], [6, 5, 8], [7, 3, 0]]
Path: ULDLURUL
Cost of path: 8
Nodes expanded: 530
Max depth: 8
Time taken: 0.0055353641510009766 seconds
```

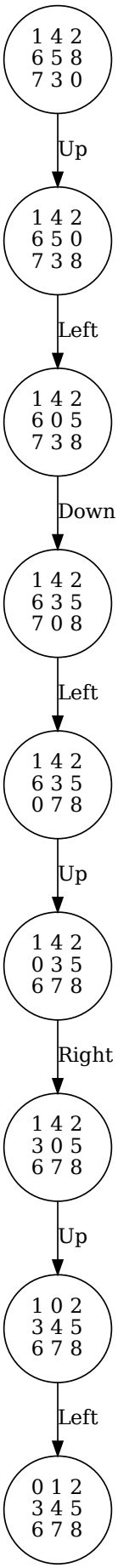
Iterative DFS:
Path: ULDLURUL
Cost of path: 8
Nodes expanded: 530
Max depth: 8
Time taken: 0.0031528472900390625 seconds



5.4 A* Euclidean Search

```
Initial State: [[1, 4, 2], [6, 5, 8], [7, 3, 0]]
Path: ULDLURUL
Cost of path: 8
Nodes expanded: 11
Max depth: 8
Time taken: 0.00042128562927246094 seconds
```

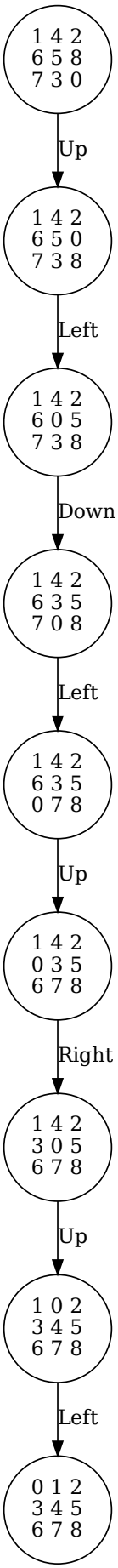
A* Euclidean Search:
Path: ULDLURUL
Cost of path: 8
Nodes expanded: 11
Max depth: 8
Time taken: 0.00025725364685058594 seconds



5.5 A* Manhattan Search

```
Initial State: [[1, 4, 2], [6, 5, 8], [7, 3, 0]]
Path: ULDLURUL
Cost of path: 8
Nodes expanded: 11
Max depth: 8
Time taken: 0.0003440380096435547 seconds
```

A* Manhattan Search:
Path: ULDLURUL
Cost of path: 8
Nodes expanded: 11
Max depth: 8
Time taken: 0.00045037269592285156 seconds



6 Conclusion

This report demonstrates the application of various search algorithms to solve the 8-puzzle problem. Each algorithm has its strengths and weaknesses, and the choice of algorithm can significantly impact the performance and efficiency of the search.

BFS and **IDDFS** always promise the lowest number of moves to reach the goal state. BFS explores all nodes at the present depth level before moving on to nodes at the next depth level, ensuring the shortest path is found. IDDFS combines the space efficiency of DFS with the optimality of BFS by performing a series of depth-limited searches.

DFS explores as far as possible along each branch before backtracking, which can lead to deep searches and higher memory usage. It does not guarantee the shortest path.

A* Search uses heuristics to guide its search, combining the cost to reach the current node and the estimated cost to reach the goal. The **Manhattan heuristic** calculates the sum of the absolute differences between the current and goal positions of each tile, generally showing fewer nodes expanded. The **Euclidean heuristic** calculates the straight-line distance between the current and goal positions of each tile. While both heuristics are effective, the Manhattan heuristic is typically more efficient for the 8-puzzle problem.

In terms of efficiency: - **A* Manhattan Search** generally performs the best in terms of nodes expanded and time taken. - **A* Euclidean Search** is also efficient but slightly less so than the Manhattan heuristic. - **BFS** and **IDDFS** ensure the shortest path but can be slower and use more memory. - **DFS** is the least efficient due to its deep search nature and higher memory usage.