github.com/sagedurivage/eight-puzzle-solver
CS170

Sage DuRivage
sduri001

Project 1

The Eight Puzzle

Trace of A* Search using the Euclidean Distance heuristic:

```
Welcome to 861285682 8 puzzle solver.
Type "1" to use a default puzzle, or "2" to enter your own puzzle.
2
Enter your puzzle, use a zero to represent the blank
Enter the first row, use space or tabs between numbers     1 0 3
Enter the second row, use space or tabs between numbers    4 2 6
Enter the third row, use space or tabs between numbers     7 5 8

Enter your choice of algorithm
Uniform Cost Search
A* with the Misplaced Tile heuristic.
A* with the Euclidean distance heuristic.

3

Expanding state

1 b 3
4 2 6
7 5 8

The best state to expand with g(n) = 1 and h(n) = 2 is...
1 2 3
4 b 6
7 5 8   Expanding this node...

The best state to expand with g(n) = 2 and h(n) = 1 is...
1 2 3
4 5 6
7 b 8   Expanding this node...


Goal!!!

To solve this problem the search algorithm expanded a total of 3 nodes.
The maximum number of nodes in the queue at any one time: 6.
The depth of the goal node was 3.
```

1. Challenges in This Project

The primary challenge for me in this project was implementing the Euclidean Distance Heuristic and A* Search functionality to produce an accurate resulting trace.

2. Design (Objects & Methods)

I organized the program around four main classes: `Node`, `Puzzle`, `Heuristic`, and `Solver`.

- The `Node` class represents each state in the puzzle, with links to parent nodes to trace the solution path.
- The `Puzzle` class maintains the initial and goal states, as well as a `getLineage()` method to identify possible successive states.
- The `Heuristic` class provides methods for the Euclidean Distance and Misplaced Tile heuristics used in A*, as well as a zero cost heuristic to turn A* into Uniform Cost Search.
- The `Solver` class includes the procedure for the A* Search algorithm, as well as the output functions for the solution traces.

3. Code Optimization

To optimize performance, I used a priority queue for the frontier so that nodes with the lowest total cost values were expanded first.

4. Graph Search

I implemented a graph search by keeping a record of explored states in a set to avoid redundant expansions and looping. I did not implement a tree search method, though I would expect that graph search requires more memory for tracking explored states.

5. Comparing Heuristic Functions

The Misplaced Tile heuristic was simple to compute but less accurate in estimating the actual cost, leading to a higher number of node expansions on average. The Euclidean Distance heuristic expanded fewer nodes while involving a more complex calculation, indicating a trade-off between accuracy and computational expense. Uniform Cost Search, expanded significantly more nodes due to the lack of heuristic guidance.

* Tables and diagrams on following page

6. Findings

This project revealed that for shallower puzzle states, the choice of heuristic has little impact on search efficiency. As puzzles become more complex, heuristics are more involved in reducing the search space and improving performance.

The Euclidean Distance heuristic was generally more effective in minimizing node expansions, while the Misplaced Tile heuristic balanced the computational cost. Uniform Cost Search proved inefficient for complex puzzles due to the lack of heuristic information.

Tables and Diagrams

Number of Nodes Expanded

	Uniform Cost	Misplaced Tile	Euclidean Distance
Trivial (0)	0	0	0
Very Easy (1)	1	1	1
Easy (2)	5	2	2
Doable (3)	15	4	4
Oh Boy (4)	141742	9182	1637
Impossible (5)	500881	382545	341706

Maximum Queue Size

	Uniform Cost	Misplaced Tile	Euclidean Distance
Trivial (0)	1	1	1
Very Easy (1)	3	3	3
Easy (2)	6	3	3
Doable (3)	16	4	4
Oh Boy (4)	59818	5453	957
Impossible (5)	71274	48879	37868