Mitchell Doan
May 6th, 2020

CS 170 Project 1: 8-Puzzle Solver

Introduction:

This assignment aims at utilizing A* graph search with 3 different heuristic functions to solve the 8-Puzzle problem. The 3 heuristic functions I implemented were Uniform Cost, Misplaced Tile Heuristic, and Euclidean Distance Heuristic. In order to see the differences in the various heuristic functions' performances, I solved the 8-Puzzle problem using various initial states (Trivial - Difficult). As you will see later in the report, different initial states displayed the importance of effective heuristic functions in A* search. This report will thoroughly analyze the methods in which I implemented my 8-Puzzle solver, along with my findings in this project. Code will be attached in a folder outside this report.

Design:

I implemented my 8-Puzzle solver in C++ using a graph search algorithm. I split my program into various files deemed appropriate. In more detail:

- Main.cpp
  - UI menu
  - General Search Function
    - Main driver of code
  - Priority Queue used as frontier
  - Vector used to store explored nordes
  - Expand function which expanded current node based on possible operations
  - Handles success state by printing out necessary output
  - Traceback which traces steps to solve puzzle from initial state to goal state
- Node.h
  - Node struct
  - Holds matrix, parent, f(n), g(n), h(n)
- Operator.h/cpp
  - Hold many necessary functions which were needed to expand nodes
  - Handles creating matrices for left, right, up, and down operator
  - Other important functions such as creating a matrix from a vector and find function which was used by euclidean heuristic and various operators
- Heuristic.h/cpp
  - Implements functions which calculate h(n) for misplaced tiles heuristic and euclidean distance heuristic

Splitting up this program into smaller files allowed for general/more reusable code which was easily improved and modified to account for the different heuristic functions. Additionally, splitting code allowed project visualization and testing to be much easier.

Comparing Heuristic Functions:

This program implements 3 different heuristic functions to solve any 8-Puzzle initial state: Uniform Cost Search, A* with Misplaced Tile Heuristic, and A* with Euclidean Distance Heuristic.

Uniform Cost Search:

Uniform Cost was a simple implementation of A* search where h(n) was hard-coded to 0. Because each operation (left, right, up, down) has a cost of 1, this implementation was simply a BFS algorithm which expanded each leaf of the frontier until the goal state was reached. Overall, this algorithm solves puzzles at a slow pace and the amount of leaves in the priority queue grows exponentially with tree depth.

A* with Misplaced Tile Heuristic

This algorithm utilized A* search where h(n) was assigned to a node based on how many tiles were misplaced in comparison to the goal state. g(n) remained the same as uniform cost search which assigned g(n) to the depth of the node where the initial node has a depth 0. f(n) computed the sum of g(n) and h(n) and this is what the priority queue used to determine priority of nodes in the frontier. Lower f(n) had a higher priority.
The example below would return 3 as {3, 6, b} are all in the wrong position.

```
Puzzle        Goal
| 1 2 b |     | 1 2 3 |
| 4 5 3 |     | 4 5 6 |
| 7 8 6 |     | 7 8 b |
```

**'b' represents the blank space in the puzzle

A* with Euclidean Distance Heuristic

This algorithm is similar to the algorithm described above, however, calculates the heuristic function using a Euclidean Distance formula. h(n) is calculated by taking the sum of the distances of each tile's current position, and its intended goal position. The formula to calculate each tile's distance goes:
sqrt((curr_row - goal_row)^2 + (curr_column - goal_column)^2)
This h(n) was added to the g(n) to get the f(n) of each node, and the priority queue was ordered by the lowest f(n).
This example has an h(n) = sqrt((0-2)^2 + (2-0)^2) = 3.

```
Puzzle        Goal
| 1 2 7 |     | 1 2 3 |
| 4 5 6 |     | 4 5 6 |
| 3 8 b |     | 7 8 b |
```

Sample Tests:
In this project, I used 6 different sample tests to display the overall performance of the 3 different algorithms. The sample tests varied from trivial to impossible. Below will illustrate the different sample tests.

Different Sample Puzzles Tested

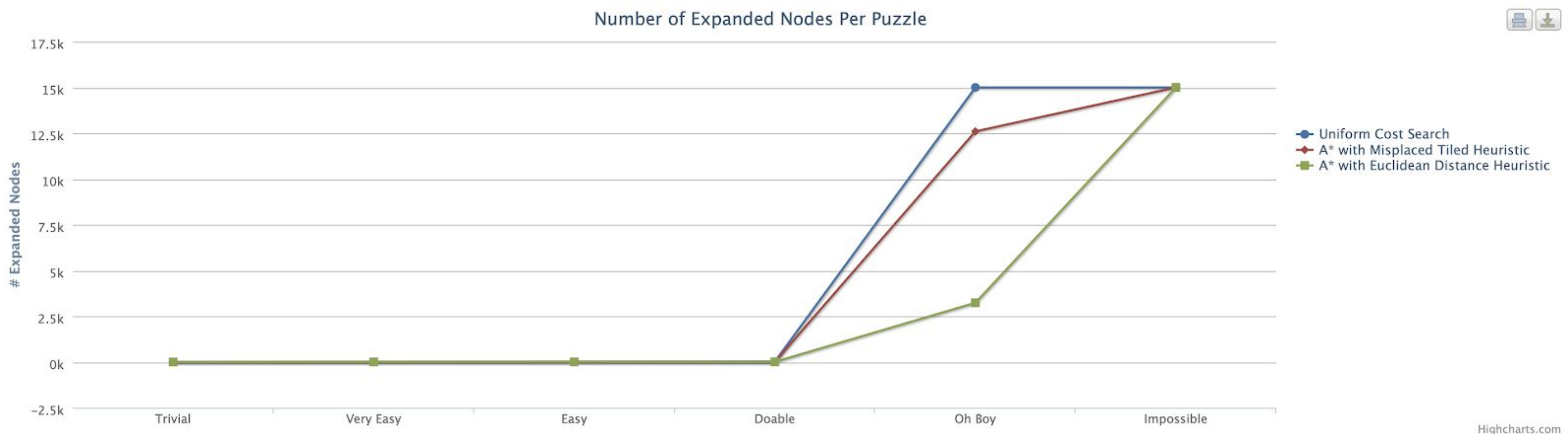
Trivial	Very Easy	Easy	Doable	Oh Boy	Impossible
| 1 2 3 |	| 1 2 3 |	| 1 2 0 |	| b 1 2 |	| 8 7 1 |	| 1 2 3 |
| 4 5 6 |	| 4 5 6 |	| 4 5 3 |	| 4 5 3 |	| 6 b 2 |	| 4 5 6 |
| 7 8 b |	| 7 b 8 |	| 7 8 6 |	| 8 7 b |	| 5 4 3 |	| 8 7 b |

Number of Nodes Expanded Per Puzzle

Level/Algorithm	Trivial	Very Easy	Eays	Doable	Oh Boy	Impossible
Uniform Cost Search	0	3	6	17	NAN	NAN
A* with Misplaced Tile	0	4	5	8	12,549	NAN
A* with Euclidean Distance	0	4	5	8	3,235	NAN

Figure 1. Data points of number of nodes expanded for different puzzle levels, for 3 different algorithms

Figure 2. Graph to display growth of expanded nodes for the 3 different algorithms
Note: Graph caps at 15,000



Max Number of Nodes in Frontier (Priority Queue)

Level/Algorithm	Trivial	Very Easy	Eays	Doable	Oh Boy	Impossible
Uniform Cost Search	1	3	6	18	NAN	NAN
A* with Misplaced Tile	1	3	3	4	4,570	NAN
A* with Euclidean Distance	1	3	3	4	1,256	NAN

Figure 3. Data points of max number of nodes for different puzzle levels, for 3 different algorithms

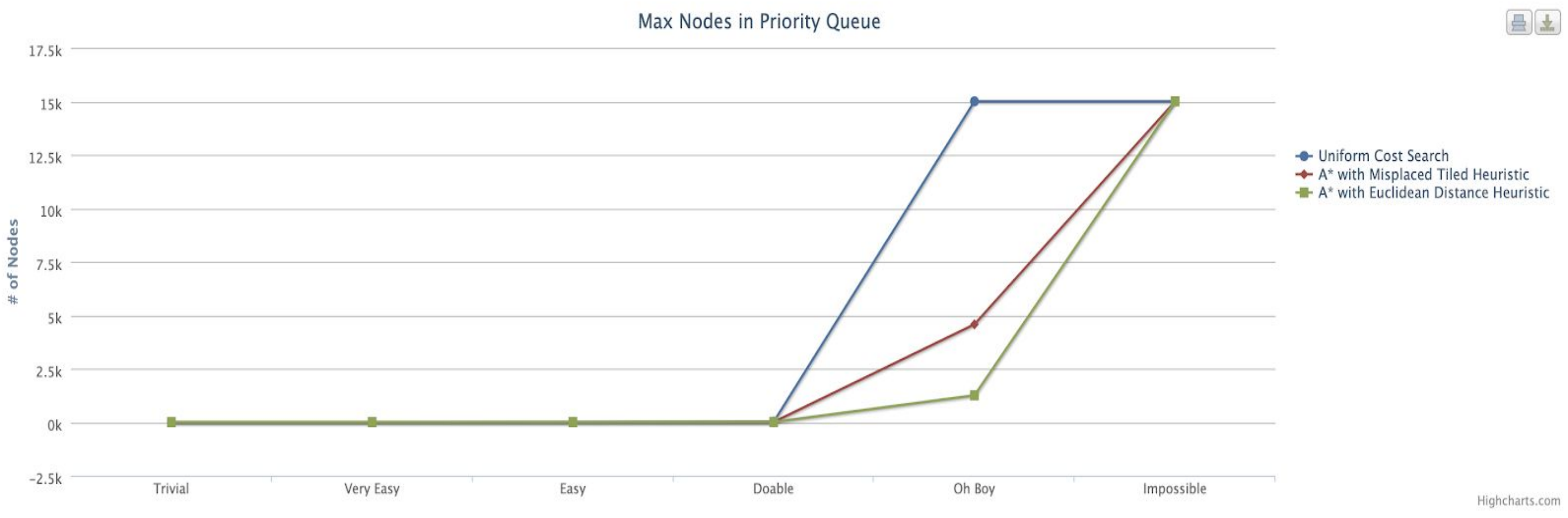


Figure 4. Displays growth of Max number of nodes in frontier
Note: Graph caps at 15,000

Search Time in Microseconds

Level/Algorithm	Trivial	Very Easy	Eays	Doable	Oh Boy	Impossible
Uniform Cost Search	162	455	110	337	NAN	NAN
A* with Misplaced Tile	165	325	630	1030	1429340	NAN
A* with Euclidean Distance	181	507	704	1341	112617	NAN

Figure 5. Data points of search time (in microseconds) for different puzzle levels, for 3 different algorithms

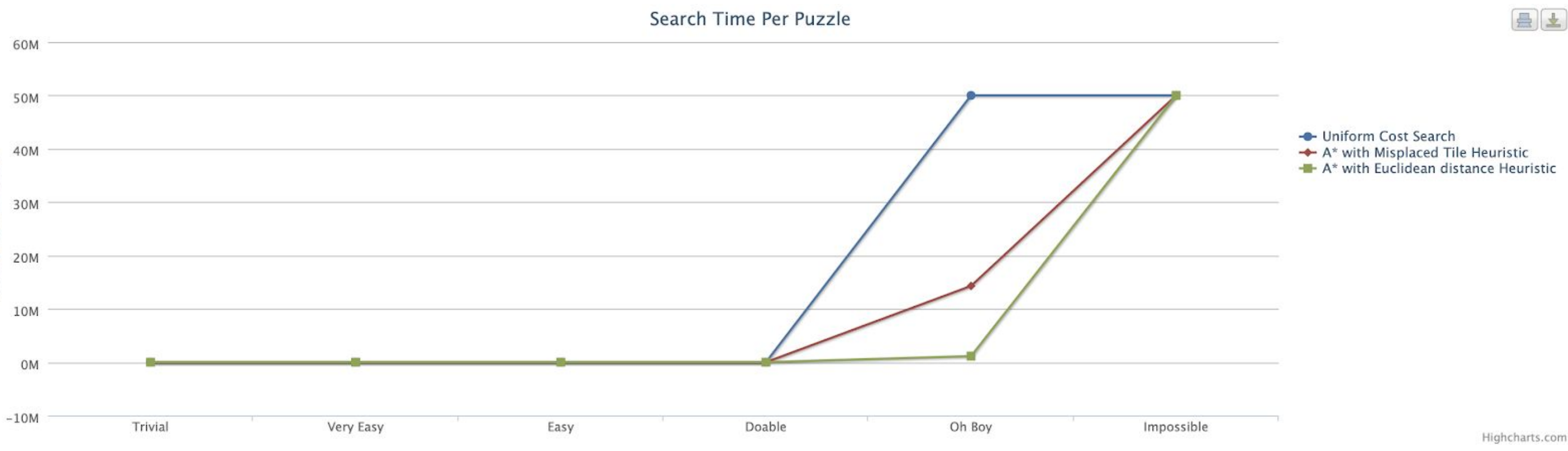


Figure 6. Data points of search times for different puzzle levels, for 3 different algorithms
Note: Graph caps at 60M Microseconds

Analysis on Sample Tests

Number of Nodes Expanded:

We can see how the different algorithms dramatically impact the number of nodes expanded. In figure 1. we see that as the puzzles become increasingly more difficult, Uniform Cost becomes a wildly inefficient algorithm compared to the other two. In figure 2. we also see that while A* in general is able to find the optimal solution, using the Euclidean distance heuristic proved to be the superior algorithm with much more difficult puzzles.

Max number of Nodes in Frontier:

The Max number of nodes in the frontier showed similar data compared to the expanded nodes data. In figure 3. We see that uniform cost showed similar data to the other two algorithms, however quickly became inefficient as the puzzles became harder. In figure 4. We see a similar outcome to figure 3, however with the heuristic functions having an even bigger impact in the number of nodes. A* search with Euclidean Heuristic proved to be the superior algorithm again.

Search Time:

When looking at figure 5. we can see how time increases with puzzle difficulty. Note that time is generally the same for all 4 algorithms until reaching the “Oh Boy” puzzle. In figure 6. we get a visualization that shows how much faster A* search with Euclidean Distance heuristic solves the puzzle in comparison to A* search with Misplaced Tile Heuristic. Similarly to space required for each algorithm, the difference in time to solve each puzzle becomes apparent as the puzzles become harder.

Conclusion

In the end my biggest take away from this project is the effectiveness of a good heuristic function in A* graph search. As stated in the beginning, Uniform Cost Search (which simply hard codes 0 as the heuristic function) was capable of finding a solution for 8-Puzzle problems. However as the puzzles became increasingly difficult, the time and space required to solve the puzzles grew at an exponential rate.

When comparing A* with Misplaced Tile Heuristic and A* with Euclidean Distance Heuristic, we can see how utilizing an effective heuristic function can improve graph search immensely. The Euclidean Distance heuristic further demonstrates how a better heuristic function vastly improves an algorithm with a decent heuristic function.

In conclusion, utilizing effective heuristic functions will decrease time and space required to solve graph search problems. While other algorithms may be able to solve shallow problems at a similar rate, a good heuristic function will solve difficult problems much faster while requiring less space.

Traceback of a Sample Puzzle

```
Welcome to Mitchell Doan's (862022288) 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 between numbers: 1 0 3
Enter the second row, use space between numbers: 4 2 6
Enter the third row, use space between numbers: 7 5 8

Your Matrix:
1 b 3
4 2 6
7 5 8

Enter your choice of algorithm
1 Uniform Cost Search
2 A* with the Misplaced Tile heuristic.
3 A* with the Euclidian distance heuristic.
3
A* with Euclidian distacnce heuristic

Expanding State
1 b 3
4 2 6
7 5 8     Expanding...

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

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

GOAL!!!

To solve this problem, the search algorithm expanded a total of 9 nodes.
The maximum number of nodes in the queue at any one time: 6

Input 1 to view trace back, any other key to exit:
1
```

```
Tracing Back:

Initial Matrix:
1 b 3
4 2 6
7 5 8

1 2 3
4 b 6
7 5 8

1 2 3
4 5 6
7 b 8

1 2 3
4 5 6
7 8 b

Total of 3 moves.

Trace back complete.
```

*Note: Red represents User input