Project 1

CS 170: Introduction to Artificial Intelligence

Saturday, February 20, 2021

Prepared and Written for Dr. Eamonn Keogh

Written by Andre Tran

Email: atran112@ucr.edu

SID: 862051183

Table of Contents

Resources

In completing this assignment I consulted:

- Blind Search_part1.pptx
- Blind Search_part2.pptx
- Trace of blind search.pptx
- Heuristic Search.pptx
- Search_to_failure.pptx
- Diameter_Puzzle.pptx
- Eight-Puzzle_briefing_and_review of search.pptx
- Project_1_The_Eight_Puzzle.pdf
- Project_1_sample_report.pdf

All important code is original. Unimportant subroutines that are not completely original are in the following:

- Priority queue:
  - https://stackoverflow.com/questions/19535644/how-to-use-the-priority-queue-stl-for-objects
  - https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/
- For timing out functions with time.h:
  - https://www.cplusplus.com/reference/ctime/
- For information on the set class:
  - https://www.cplusplus.com/reference/set/set/

Introduction

The 8-puzzle is 3 by 3 grid that consists of 8 numbers and one missing tile, and it is a smaller version of the 4 by 4 grid 15-puzzle. The solution to either the 8-puzzle or 15-puzzle is to have the tiles ordered numerically in each row with the bottom right tile empty. This is done by sliding tiles adjacent to the blank space into the blank space. Eventually, if slid in the correct directions, the puzzle will be solved. An image of a solved 15-puzzle can be seen below in figure 1. So, the goal of this project is to solve an 8-puzzle tile through a program. This program should find the solution in the least number of steps, and, depending on the algorithm implemented, it should solve it in the shortest amount of time and the least amount of space possible.

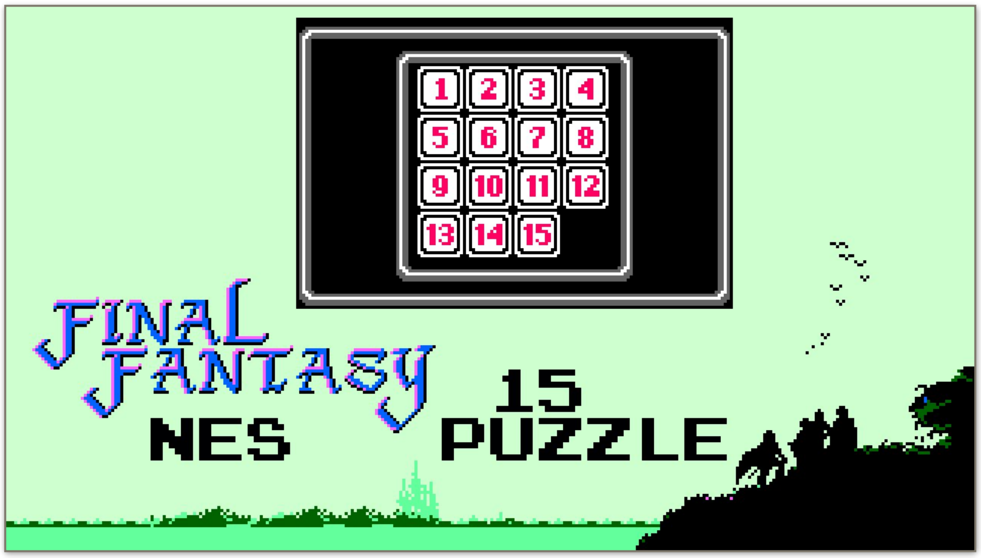


FIGURE 1
A complete 15-Puzzle from a mini-game in Final Fantasy

This assignment is the first project in Dr. Eamonn Keogh's Introduction to AI course at the University of California, Riverside during the quarter of Winter 2021. This paper explores the usage of the Uniform Cost Search, and the Misplaced Tile and Manhattan Distance heuristics applied to A*. My language of choice was C++, and a portion of my code is included as well as a link to my Github Repository.

Comparison of Algorithms

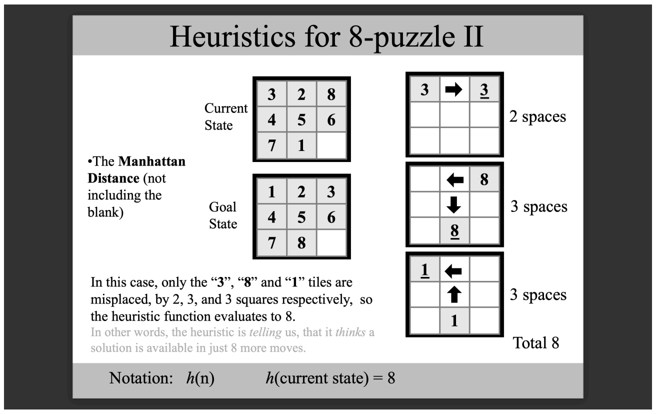
There are three algorithms implemented in this project. These include the following: Uniform Cost Search, A* using the Misplaced Tile heuristic, and A* using the Manhattan Distance Heuristic.

Uniform Cost Search

This type of search is A* search with h(n) set to 0. Thus, every move to a different state will increase the cost/depth/g(n) by 1, thereby traversing every possibility until it reaches either the solution or fails. It is the slowest of the three algorithms.

The Misplace Tile Heuristic

This type of search finds the number of tiles that are "misplaced" in a puzzle and assigns the h(n) to that value. Below in figure 2 is an image from Dr. Keogh's slides that depicts how the Misplaced Tile Heuristic is calculated. This number is then summed with g(n) to generate a total cost f(n). Thus, the queue will traverse the list based on the lowest total costs.



Heuristics for 8-puzzle II

•The **Manhattan Distance** (not including the blank)

In this case, only the "**3**", "**8**" and "**1**" tiles are misplaced, by 2, 3, and 3 squares respectively, so the heuristic function evaluates to 8.
In other words, the heuristic is *telling* us, that it *thinks* a solution is available in just 8 more moves.

Notation: *h*(n) *h*(current state) = 8

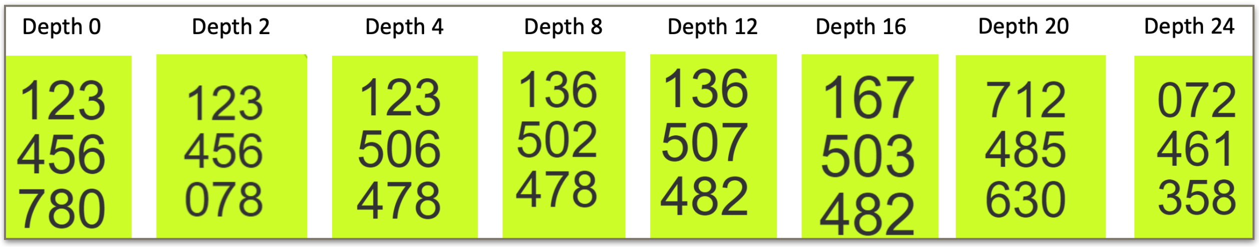
FIGURE 2
Screenshot from Dr. Keogh's slides that depict how the Misplaced Tile Heuristic is Calculated

The Manhattan Distance Heuristic

This type of search finds the distance of the misplaced tiles from their goal state. All of these distances (except for the blank tile) are summed up to get the value of h(n). As with the Misplaced Tile Heuristic, this h(n) is then added to the g(n) to find the total costs f(n). This will ensure that puzzles with a total lower cost are prioritized in the queue.

Comparison of Algorithms on Sample Puzzles

As shown in Figure 3, Dr. Keogh provided us with the following test cases, sorted by the depth of the optimal solution. So, to compare the different algorithms, I have created two graphs below (figure 4 and figure 5) that compare the depth to the time (the number of nodes expanded) and space (the maximum size of the queue). Please note that my implementation checked for duplicates, so my results may differ from those who did not.



Depth 0	Depth 2	Depth 4	Depth 8	Depth 12	Depth 16	Depth 20	Depth 24
123	123	123	136	136	167	712	072
456	456	506	502	507	503	485	461
780	078	478	478	482	482	630	358

FIGURE 3
A screenshot of Dr. Keogh's slides depicting test cases for the 8-puzzle in order of depth

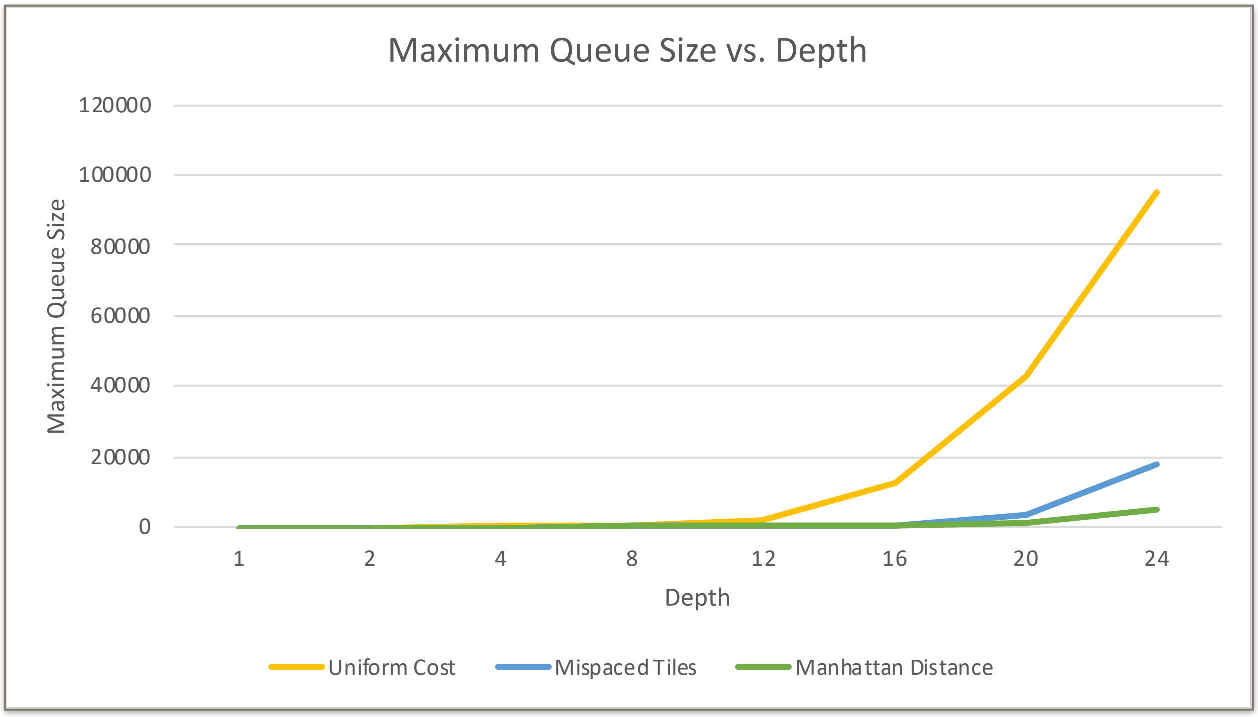


FIGURE 4
Graph showing the Maximum Queue Size in relation to Depth

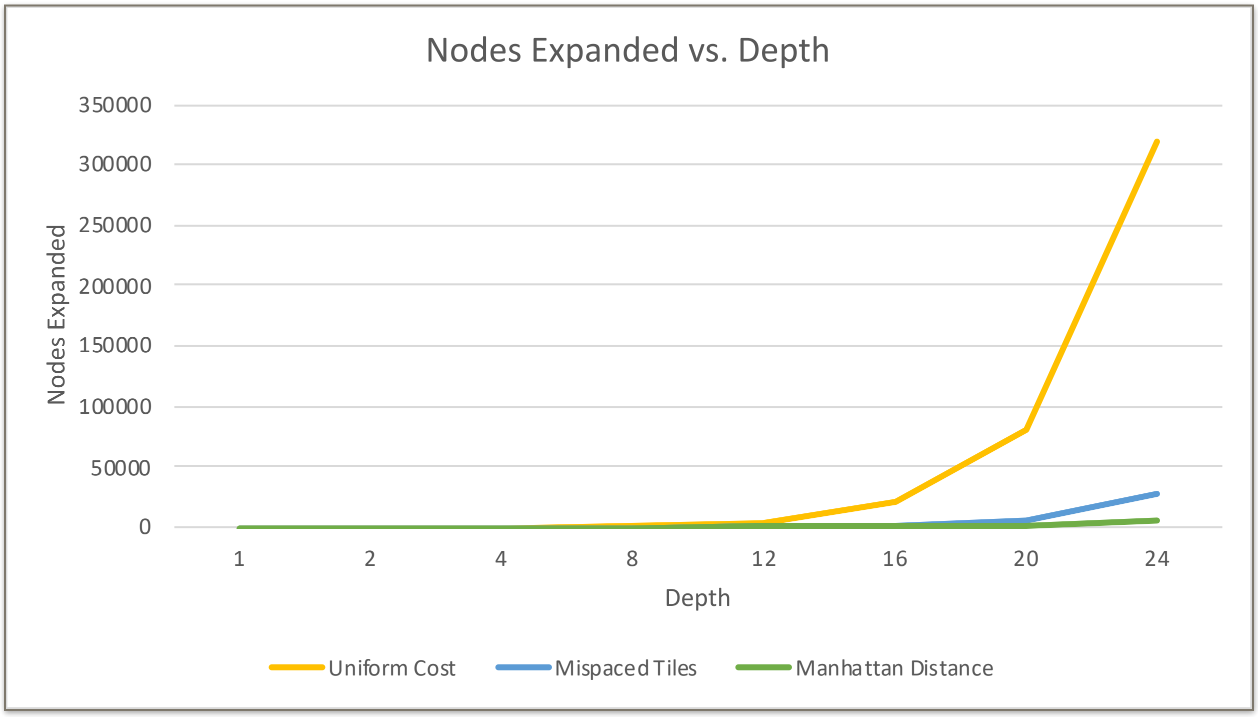


FIGURE 5
Graph showing the Number of Nodes Expanded in relation to Depth

In figures 4 and 5 above, we can see the trends of the algorithm's depth in relation to their maximum queue size (figure 4) and nodes expanded (figure 5). From the figures, we can see that all three algorithms take up a similar amount of space (Maximum Queue Size) and time (Nodes Expanded) up until depth 12. After that, the Uniform Cost algorithm takes up the most space and most time while the Misplaced Tiles and Manhattan Distance take up significantly less space and time. However, the Manhattan Distance is slightly quicker and takes up less space in comparison to Misplaced Tiles. Thus, Manhattan Distance is the best algorithm, then Misplaced Tiles, and, lastly, Uniform Cost.

Conclusion

Each algorithm is slightly different in the way it solves each problem. The Uniform Costs only considers g(n), while the Misplaced Tiles and Manhattan Distance algorithms consider both the g(n) and h(n). However, the best way to solve the 8-puzzle problem is with the Manhattan Distance algorithm as it takes up the least amount of space and takes the least amount of time.

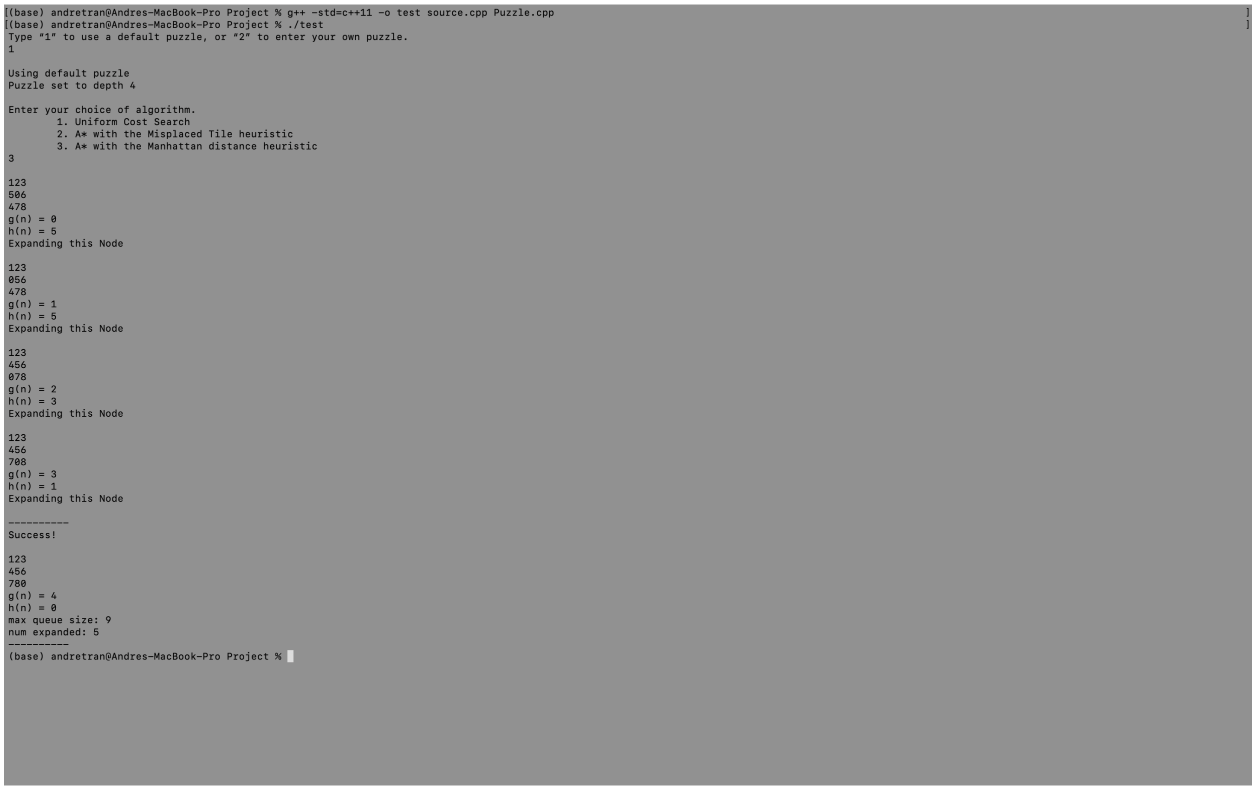
In the future, I hope to use these concepts in other areas. For example, I have a strong interest in the urban environment and transportation which heavily involve maps. So, I hope I can use this knowledge (such as the Manhattan Distance algorithm) to find locations that are underserved by public transportation or other public amenities. Thus, my opportunities to utilize this knowledge are limitless, and implementing a small project like this, though small, can help me grow my AI fundamentals to eventually fulfill my career goals.

Trace

Below are screenshots of the trace of an easy puzzle and a hard puzzle.

Easy Puzzle

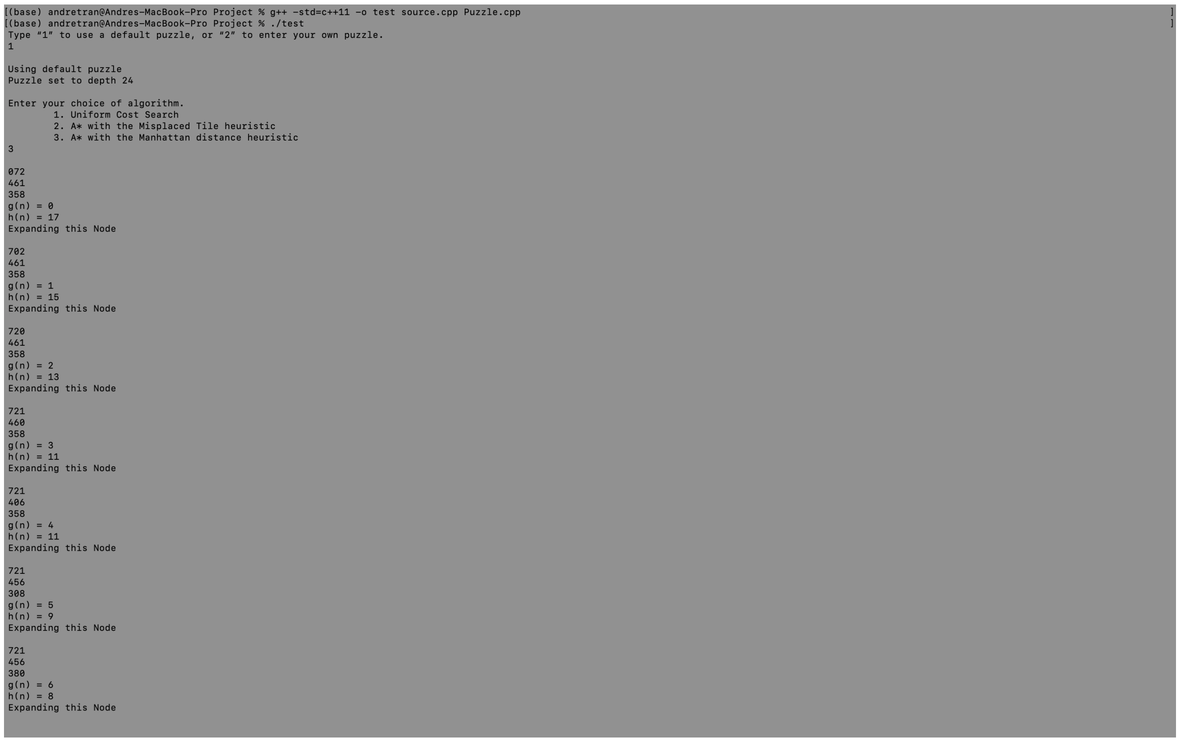
Below is the trace of an easy puzzle at depth 4.



```
[(base) andretran@Andres-MacBook-Pro Project % g++ -std=c++11 -o test source.cpp Puzzle.cpp      ]
[(base) andretran@Andres-MacBook-Pro Project % ./test                                            ]
 Type “1” to use a default puzzle, or “2” to enter your own puzzle.
 1

 Using default puzzle
 Puzzle set to depth 4

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

 123
 506
 478
 g(n) = 0
 h(n) = 5
 Expanding this Node

 123
 056
 478
 g(n) = 1
 h(n) = 5
 Expanding this Node

 123
 456
 078
 g(n) = 2
 h(n) = 3
 Expanding this Node

 123
 456
 708
 g(n) = 3
 h(n) = 1
 Expanding this Node

 ----------
 Success!

 123
 456
 780
 g(n) = 4
 h(n) = 0
 max queue size: 9
 num expanded: 5
 ----------
 (base) andretran@Andres-MacBook-Pro Project %
```

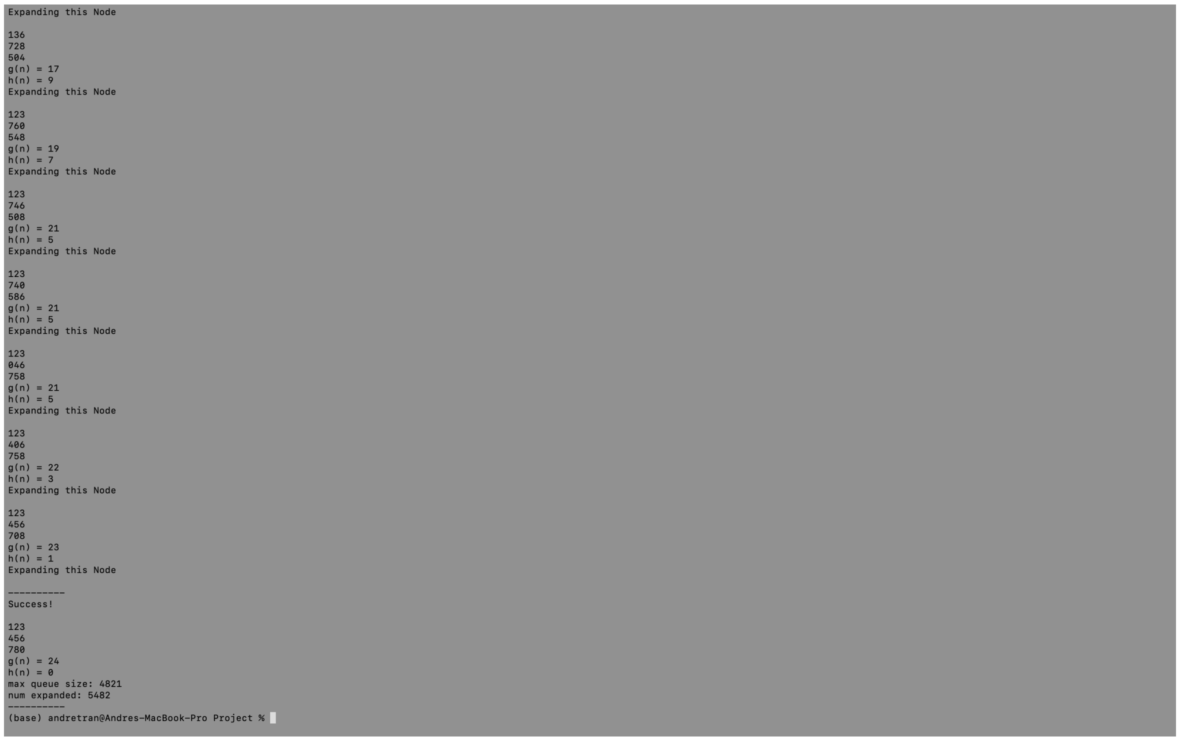
Hard Puzzle

Below is the beginning of the hard trace of depth 24.



```
[(base) andretran@Andres-MacBook-Pro Project % g++ -std=c++11 -o test source.cpp Puzzle.cpp
[(base) andretran@Andres-MacBook-Pro Project % ./test
 Type “1” to use a default puzzle, or “2” to enter your own puzzle.
 1

 Using default puzzle
 Puzzle set to depth 24

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

 072
 461
 358
 g(n) = 0
 h(n) = 17
 Expanding this Node

 702
 461
 358
 g(n) = 1
 h(n) = 15
 Expanding this Node

 720
 461
 358
 g(n) = 2
 h(n) = 13
 Expanding this Node

 721
 460
 358
 g(n) = 3
 h(n) = 11
 Expanding this Node

 721
 406
 358
 g(n) = 4
 h(n) = 11
 Expanding this Node

 721
 456
 308
 g(n) = 5
 h(n) = 9
 Expanding this Node

 721
 456
 380
 g(n) = 6
 h(n) = 8
 Expanding this Node
```

Below is the end of the trace.



```
 Expanding this Node

 136
 728
 504
 g(n) = 17
 h(n) = 9
 Expanding this Node

 123
 760
 548
 g(n) = 19
 h(n) = 7
 Expanding this Node

 123
 746
 508
 g(n) = 21
 h(n) = 5
 Expanding this Node

 123
 740
 586
 g(n) = 21
 h(n) = 5
 Expanding this Node

 123
 046
 758
 g(n) = 21
 h(n) = 5
 Expanding this Node

 123
 406
 758
 g(n) = 22
 h(n) = 3
 Expanding this Node

 123
 456
 708
 g(n) = 23
 h(n) = 1
 Expanding this Node

 ----------
 Success!

 123
 456
 780
 g(n) = 24
 h(n) = 0
 max queue size: 4821
 num expanded: 5482
 ----------
 (base) andretran@Andres-MacBook-Pro Project %
```

Code

Link to Github Repository: https://github.com/atran112/8Puzzle

Compiled the code with: g++ -std=c++11 -o test source.cpp Puzzle.cpp Functions.cpp

Ran the program with: ./test

Here is a code snippet of my trace algorithm called findSolution:

```
bool findSolution(const Puzzle& currPuzzle, const string& heuristicType) {

    time_t start = time(NULL);
    int seconds = 60 * 15; // end loop after this time has elapsed
    time_t end = start + seconds;

    int maxQSize = 0;
    int numExpanded = 0;

    vector<int> gridSolution {1, 2, 3, 4, 5, 6, 7, 8, 0};

    priority_queue<Puzzle> puzzleQ;
    set<vector<int>> puzzleSet;
    vector<int> temp = currPuzzle.grid;
    temp.push_back(currPuzzle.cost);
    temp.push_back(currPuzzle.heuristic);
    puzzleSet.insert(temp);

    puzzleQ.push(currPuzzle);

    while (!puzzleQ.empty()) {
        Puzzle currPuzzle = puzzleQ.top();
        puzzleQ.pop();

        ++numExpanded;

        if (currPuzzle.grid == gridSolution) {

            cout << endl << "----------" << endl << "Success!" << endl;
            currPuzzle.display(); //
            cout << "max queue size: " << maxQSize << endl;
            cout << "num expanded: " << numExpanded << endl;
            cout << "----------" << endl;
            return true;
        }

        currPuzzle.display(); //DO NOT DELETE

        cout << "Expanding this Node" << endl; //DO NOT DELETE

        vector<Puzzle> nextPuzzles = currPuzzle.expandPuzzle(heuristicType);

        for (unsigned i = 0; i < nextPuzzles.size(); ++i) {

            Puzzle nextPuzzle = nextPuzzles.at(i);

            if (nextPuzzle.cost <= 31) {

                vector<int> temp = nextPuzzle.grid;
                temp.push_back(nextPuzzle.cost);
                temp.push_back(nextPuzzle.heuristic);

                if (puzzleSet.count(temp) == 0) {
                    puzzleSet.insert(temp);
                    puzzleQ.push(nextPuzzles.at(i));
                }
            }

        }

        if (maxQSize < puzzleQ.size()) {
            maxQSize = puzzleQ.size();
        }

        if (time(NULL) >= end) {
            cout << "Timing Out. The function suprassed " << seconds/60 << " minutes." << endl;
            break;
        }

    }

    cout << "failure" << endl;
    return false;

}
```