Nelson Tran
SID: 862332246
Email: ntran189@ucr.edu
Date: February 8, 2025

While working this project, I consulted:

- The “Project 1 The Eight Puzzle” and “Eight-Puzzle Briefing and Review of Search” prompt slides, and the "Blind Search" and "Heuristic Search" lecture slides by Dr. Eamonn Keogh, in addition to notes annotated from his lectures.
- C++ 11, 14, 17, 20, 23, and 26 Documentation. This is the URL to the Table of Contents of the aforementioned C++ versions: https://en.cppreference.com/w/.
- https://deniz.co/8-puzzle-solver/ for the random premade initial puzzles.

All crucial code is original. Noncrucial subroutines that are not fully original are:

- All subroutines used from **chrono**, to track the time of the search function.
- All subroutines used from **thread**, to keep parts of the code running in parallel.
- All subroutines used from **iostream**, to input and output.
- All subroutines used from **vector**, to represent the 2D puzzles and puzzle nodes.
- All subroutines used from **string**, to represent the different heuristic types.
- All subroutines used from **cmath**, to keep the Manhattan Distance positive.
- All subroutines used from **algorithm**, to reverse the trace path of the solution.
- All subroutines used from **queue**, to store the nodes in order of smallest costs.
- All subroutines used from **map**, to track nodes that have already been visited.

Report Outline

CS170 - Project 1: The Eight Puzzle

Nelson Tran | SID 862332246 | February 8, 2025

Introduction

The Eight Puzzle is a puzzle game that is played on a three-by-three grid with eight numbered tiles. The game will typically start out where the tiles are shuffled randomly, as shown in Figure 1, and the player must move the tiles around until the game is solved. The tiles can only be moved left, right, up, or down (no diagonal moves). The game is considered to be solved when the tiles are in numerical order from one to eight with a blank slot at the bottom right, also shown in Figure 1. There exists different variations of this puzzle, such as the Fifteen Puzzle that is played on a four-by-four grid with fourteen tiles instead. These variations are consistent where the bottom right slot must not have a tile on it and all the tiles must be in numerical order from left to right, top to bottom.

Shuffled	Solved
[1, 2, 0]	[1, 2, 3]
[5, 7, 3]	[4, 5, 6]
[6, 4, 8]	[7, 8, 0]

Figure 1: Shuffled and Solved Examples of the 8-puzzle with 0 representing the blank space.

This report will go over the first assigned project in Dr. Eamonn Keogh's Introduction to Artificial Intelligence (CS170) course that I took at the University of California, Riverside during the academic quarter of Winter 2025. It summarizes my discoveries in detail through the whole process of completing this project. Furthermore, the report will also examine the algorithms of Uniform Cost Search, A* using the Misplaced Tile Heuristic, and A* using the Manhattan Distance Heuristic. My objective is to evaluate the effectiveness of these algorithms in solving the Eight Puzzle (and all variations of this puzzle). The coding language that I chose to complete this assignment was C++ (version 17). The full code for the project is included at the bottom of this report.

Algorithms Used in the Project

As mentioned prior, the algorithms that were applied to solve the puzzle are the Uniform Cost Search, A* using the Misplaced Tile Heuristic, and A* using the Manhattan Distance Heuristic.

Uniform Cost Search

The **Uniform Cost Search** algorithm functions by expanding all possible nodes. Every single step is considered to have a cost of one, meaning that the cost is simply the amount of moves made at a given step of the game. It behaves like Breadth-First Search due to the fact that all steps have the same cost. Compared to the other algorithms, Uniform Cost Search is similar where it uses A* to search, but unlike the others, it does not use a heuristic. Consider Figure 2:

	Current State	
	[1, 2, 3]	
	[4, 5, 0]	
	[7, 8, 6]	
Move Up	Move Left	Move Down
[1, 2, 0]	[1, 2, 3]	[1, 2, 3]
[4, 5, 3]	[4, 0, 5]	[4, 5, 6]
[7, 8, 6]	[7, 8, 6]	[7, 8, 0]

Figure 2: Shows a state of the game where three moves are possible. Moving down will win the game.

In Figure 2, we see that in the current state of the game, the blank tile can be moved in three different ways: up, left, or down. Now to us, it would seem that all we would have to do is to swap the blank tile and the sixth tile in order to win the game. However, since all of these moves have the same cost, Uniform Cost Search will evaluate all possible moves until it finds the winning move. Depending on how the algorithm was set up in terms of which direction should be checked first, it could choose the winning move early or late. Using this on game states further away from the solved state, Uniform Cost Search depends on the luck of its choice to determine if we can solve the game fast or slow.

A* Using the Misplaced Tile Heuristic

The **A*** algorithm with the **Misplaced Tile Heuristic** functions by expanding the node(s) with the cheapest cost. The cost is determined from both the number of moves made and the heuristic. The heuristic counts the number of tiles that are in the incorrect position based on the solved state. Consider Figure 3:

Current State	Solved State
[1, 0, 3]	[1, 2, 3]
[4, 2, 5]	[4, 5, 6]
[7, 8, 6]	[7, 8, 0]

Figure 3: An example of how the Misplaced Tile Heuristic works.

In Figure 3, we see that the algorithm has counted three tiles to be misplaced in the current state. Note how it does not count the blank tile to be misplaced despite the fact that it is not located in the bottom right corner. If it did, the heuristic would no longer be valid. This is how the A* algorithm with the Misplaced Tile Heuristic operates every time it checks to determine what the heuristic value should be. It prioritizes the moves that lead into game states with the least amount of misplaced tiles first.

A* Using the Manhattan Distance Heuristic

The **A*** algorithm with the **Manhattan Distance Heuristic** functions by expanding the node(s) with the cheapest cost. The cost is also determined from both the number of moves made and the heuristic. However this time, the heuristic instead considers the distance each tile needs to move in order to reach its correct position based on the solved state. Consider Figure 4:

Current State	Solved State
[1, 2, 3]	[1, 2, 3]
[4, 8, 5]	[4, 5, 6]
[7, 6, 0]	[7, 8, 0]

Figure 4: An example of how the Misplaced Tile Heuristic and the Manhattan Distance Heuristic differ. The yellow tiles each have a distance of one, and the red tile has a distance of two.

Here in Figure 4, we see that three of the tiles are misplaced, so the Misplaced Tile Heuristic would return a heuristic value of three. On the other hand, the Manhattan Distance Heuristic would return a heuristic value of four instead. This is because while the eighth and the fifth tiles are one tile away from where they should be, the sixth tile is two tiles away. The sum of all of the distances is four, which is what this algorithm uses to determine its heuristic for the cost of this current state.

Algorithm Comparisons on Sample Puzzles

To help compare the three algorithms and how they handle different puzzle depths, Dr. Keogh provided eight test cases, which can be seen in Figure 5.

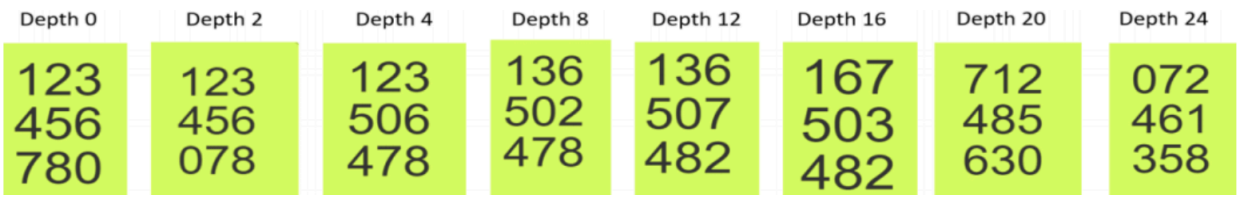

Figure 5: A screenshot of the test cases. These test cases are provided by Dr. Eamonn Keogh and are not originally mine. They originate from his “Project 1 The Eight Puzzle” prompt slides.

First, we will compare the relationship between the solution depth and the number of nodes expanded for each of the three search algorithms, as shown by Figure 6.

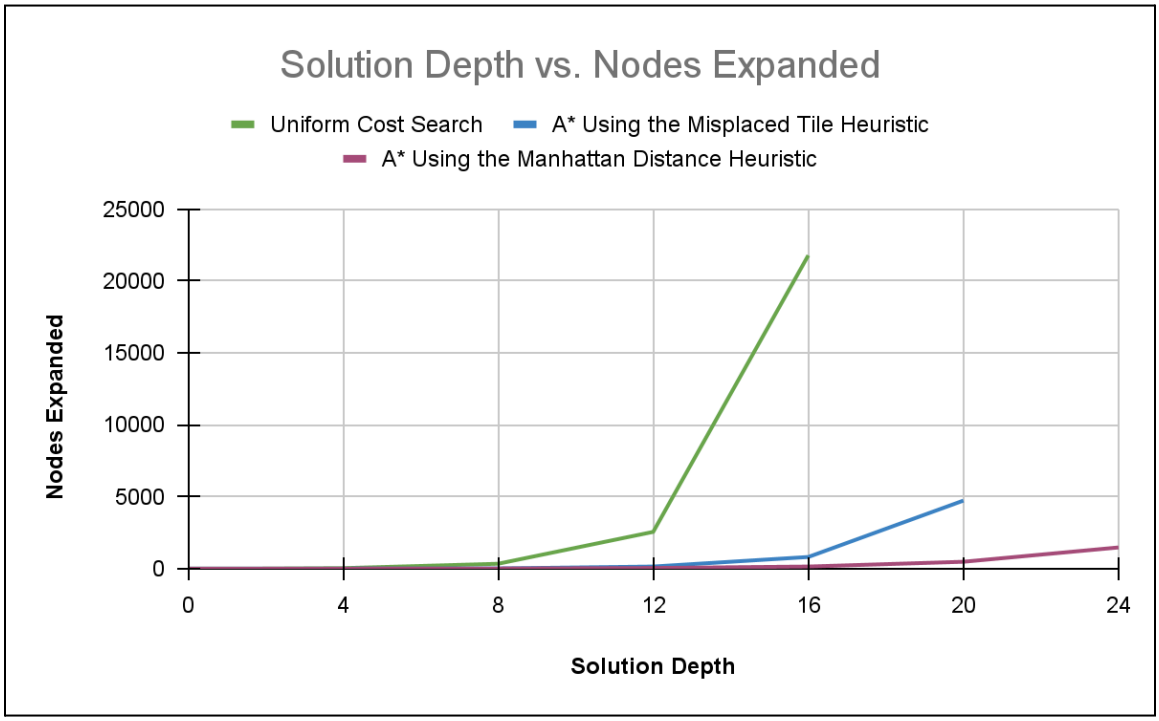

Figure 6: A graph of the solution depth vs. the nodes expanded by the three search algorithms. Note how Uniform Cost Search timed out at depth 16 and deeper, and A* with Misplaced Tile Heuristic timed out at depth 20 and deeper.

I used the test cases mentioned in Figure 5 for the graph in Figure 6. As we can see here, the common trend is that all of the algorithms expand roughly the same amount of nodes with solution depths of lower values. The Uniform Cost Search algorithm starts to expand much more nodes than the others past a depth of eight, whereas the A* with Misplaced Tile Heuristic algorithm expands a noticeable amount of nodes than the A* with Manhattan Distance algorithm past a depth of twelve. We can determine from the graph that Uniform Cost Search is the least efficient and A* using the Manhattan Distance Heuristic is the most efficient.

Now instead of comparing the relationship between solution depth and the number of nodes expanded, let's take a look into the relationship between the solution depth and the maximum size of the queue for the algorithms, which is displayed by Figure 7. I also used the same test cases again.

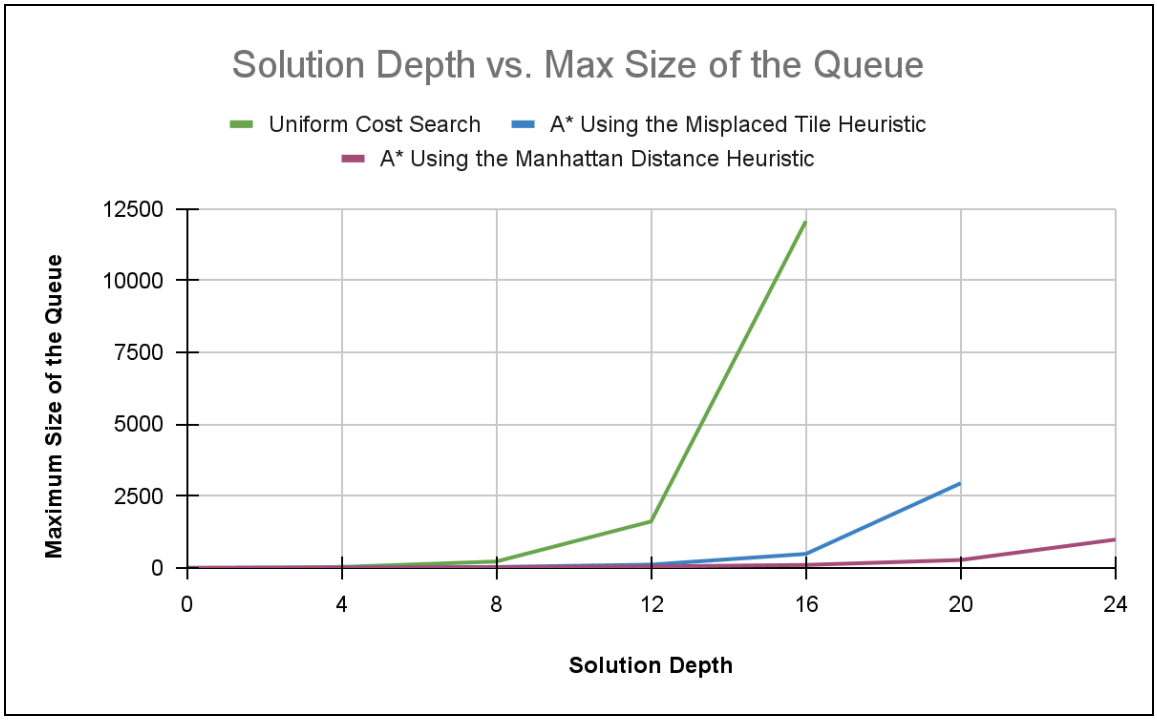

Figure 7: A graph of the solution depth vs. the maximum size of the queue by the three search algorithms. Note how Uniform Cost Search timed out at depth 16 and deeper, and A* with Misplaced Tile Heuristic timed out at depth 20 and deeper.

The trend here in Figure 7 is practically identical to the trend in Figure 8. In fact, if we were to remove the numbers and words of the graph, and just take a look at the three colored lines, we would not be able to tell the graphs apart. Similar to the previous trend, this one displays how the three algorithms share almost identical maximum sizes of their queues with solution depths of lower values. Uniform Cost Search is shown to still be the least efficient due to the fact that its maximum sizes ramp up much faster than the other algorithms do as we raise the depth's value. On the other hand, A* using the Manhattan Distance Heuristic is still the most efficient algorithm since it maintains relatively smaller maximum queue sizes when compared to the other algorithms.

Additional Comparison

I also compared the algorithms through the relationship between the solution depth and time measured in milliseconds. Consider Figure 8:

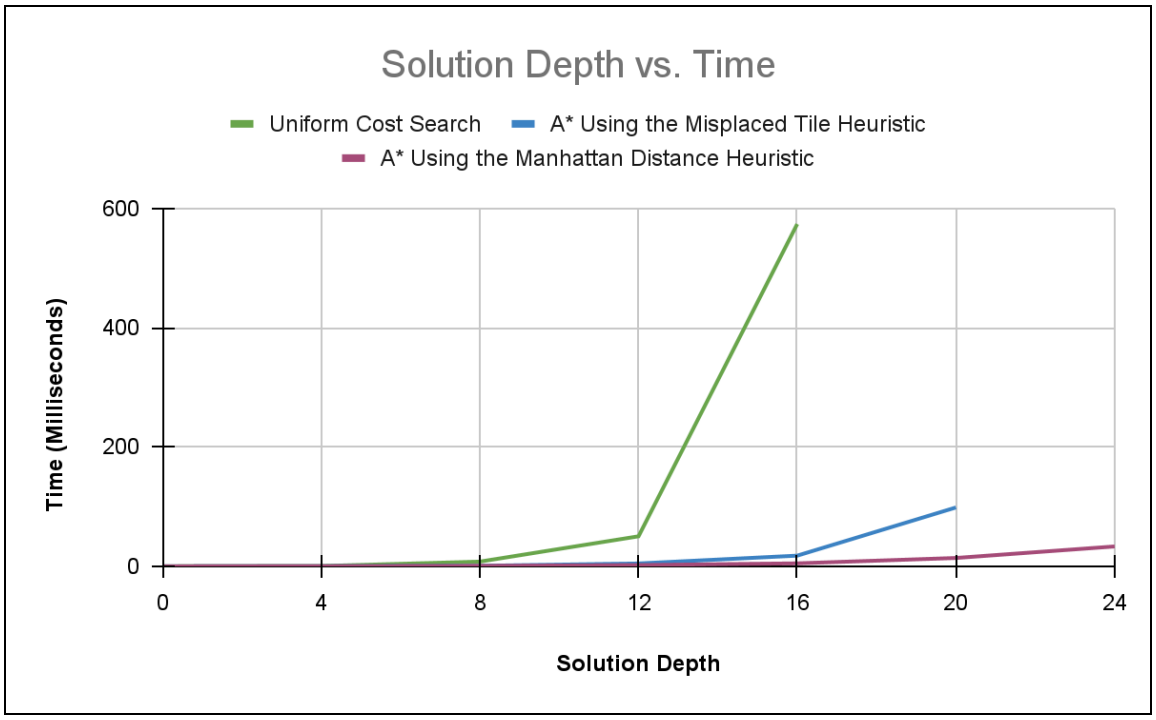

Figure 8: A graph of the solution depth vs. time (milliseconds). Note how Uniform Cost Search timed out at depth 16 and deeper, and A* with Misplaced Tile Heuristic timed out at depth 20 and deeper.

Figure 8 highlights the same trend as Figures 6 and 7. Whether it is the relationship between depth and nodes expanded, depth and max queue size, or depth and time, all three relationships display the same pattern. If my graphs were not labeled, I genuinely could not tell my graphs apart; the colored lines are so similar, borderline indistinguishable.

Conclusion

When looking at the Uniform Cost Search, the A* using the Misplaced Tile Heuristic, and the A* using the Manhattan Distance Heuristic algorithms used in this project and how they compare against one another, this project demonstrated that:

- ★ Amongst the algorithms, A* using the Manhattan Distance Heuristic is the most efficient, followed by the A* using the Misplaced Tile Heuristic, with Uniform Cost Search being the least efficient.
- ★ Heuristic-based approaches significantly improve the efficiency of searching algorithms, solving a game of the Eight Puzzle much faster. This trend can be seen from Figures 6, 7, and 8 with how the colored lines of the two heuristic algorithms are much closer to each other than they are to Uniform Cost Search.
- ★ The choice of a heuristic can drastically change the speed of an algorithm, as shown with the A* algorithm when it uses the Misplaced Tile Heuristic versus the Manhattan Distance Heuristic.

The following is a sample solution traceback of an **easy** (depth 3) puzzle:

```
Welcome! This is a program customized to solve an Eight Puzzle.
To get started, please do either one of the following:
> Type "1" to choose a premade initial puzzle.
> Type "2" to create your own initial puzzle.
Press ENTER once you have typed your choice.
2

You have chosen to create your own initial puzzle.
Enter your puzzle, separating each number with a space in between.
The blank tile should be represented with "0".
Please ensure that your puzzle is a legit 8-puzzle.
Press ENTER when you finish.

Enter the numbers for row 1: 1 0 3
Enter the numbers for row 2: 4 2 6
Enter the numbers for row 3: 7 5 8

Please select an ALGORITHM by typing the number corresponding to the following:
"1" for Uniform Cost Search
"2" for Misplaced Tile Heuristic
"3" for Manhattan Distance Heuristic
3

The best node to expand with a g(n) = 0 and h(n) = 3 is:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]
The best node to expand with a g(n) = 1 and h(n) = 2 is:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
The best node to expand with a g(n) = 2 and h(n) = 1 is:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
The best node to expand with a g(n) = 3 and h(n) = 0 is:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Goal state!

Solution depth was 3
Number of nodes expanded: 4
Max queue size: 8

Time: 0.268 milliseconds
```

The following is a sample solution traceback of a **hard** (depth 21) puzzle:

```
Welcome! This is a program customized to solve an Eight Puzzle.
To get started, please do either one of the following:
> Type "1" to choose a premade initial puzzle.
> Type "2" to create your own initial puzzle.
Press ENTER once you have typed your choice.
2

You have chosen to create your own initial puzzle.
Enter your puzzle, separating each number with a space in between.
The blank tile should be represented with "0".
Please ensure that your puzzle is a legit 8-puzzle.
Press ENTER when you finish.

Enter the numbers for row 1: 8 2 6
Enter the numbers for row 2: 1 7 0
Enter the numbers for row 3: 5 4 3

Please select an ALGORITHM by typing the number corresponding to the following:
"1" for Uniform Cost Search
"2" for Misplaced Tile Heuristic
"3" for Manhattan Distance Heuristic
3

The best node to expand with a g(n) = 0 and h(n) = 13 is:
[8, 2, 6]
[1, 7, 0]
[5, 4, 3]
The best node to expand with a g(n) = 1 and h(n) = 12 is:
[8, 2, 0]
[1, 7, 6]
[5, 4, 3]
The best node to expand with a g(n) = 2 and h(n) = 13 is:
[8, 0, 2]
[1, 7, 6]
[5, 4, 3]
The best node to expand with a g(n) = 3 and h(n) = 12 is:
[0, 8, 2]
[1, 7, 6]
[5, 4, 3]
The best node to expand with a g(n) = 4 and h(n) = 11 is:
[1, 8, 2]
[0, 7, 6]
[5, 4, 3]
The best node to expand with a g(n) = 5 and h(n) = 10 is:
[1, 8, 2]
[7, 0, 6]
[5, 4, 3]
The best node to expand with a g(n) = 6 and h(n) = 9 is:
[1, 8, 2]
[7, 4, 6]
[5, 0, 3]
```

```
The best node to expand with a g(n) = 7 and h(n) = 8 is:
[1, 8, 2]
[7, 4, 6]
[0, 5, 3]
The best node to expand with a g(n) = 8 and h(n) = 7 is:
[1, 8, 2]
[0, 4, 6]
[7, 5, 3]
The best node to expand with a g(n) = 9 and h(n) = 6 is:
[1, 8, 2]
[4, 0, 6]
[7, 5, 3]
The best node to expand with a g(n) = 10 and h(n) = 7 is:
[1, 8, 2]
[4, 6, 0]
[7, 5, 3]
The best node to expand with a g(n) = 11 and h(n) = 6 is:
[1, 8, 2]
[4, 6, 3]
[7, 5, 0]
The best node to expand with a g(n) = 12 and h(n) = 7 is:
[1, 8, 2]
[4, 6, 3]
[7, 0, 5]
The best node to expand with a g(n) = 13 and h(n) = 8 is:
[1, 8, 2]
[4, 0, 3]
[7, 6, 5]
The best node to expand with a g(n) = 14 and h(n) = 7 is:
[1, 0, 2]
[4, 8, 3]
[7, 6, 5]
The best node to expand with a g(n) = 15 and h(n) = 6 is:
[1, 2, 0]
[4, 8, 3]
[7, 6, 5]
The best node to expand with a g(n) = 16 and h(n) = 5 is:
[1, 2, 3]
[4, 8, 0]
[7, 6, 5]
The best node to expand with a g(n) = 17 and h(n) = 4 is:
[1, 2, 3]
[4, 8, 5]
[7, 6, 0]
The best node to expand with a g(n) = 18 and h(n) = 3 is:
[1, 2, 3]
[4, 8, 5]
[7, 0, 6]
The best node to expand with a g(n) = 19 and h(n) = 2 is:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]
The best node to expand with a g(n) = 20 and h(n) = 1 is:
```

```
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]
The best node to expand with a g(n) = 21 and h(n) = 0 is:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Goal state!

Solution depth was 21
Number of nodes expanded: 1166
Max queue size: 690

Time: 14.403 milliseconds
```

URL to my Code:

https://github.com/Grymrose/the_eight_puzzle/blob/main/Search_N-Puzzle.cpp/

Search_N-Puzzle.cpp

```
#include <chrono>
#include <thread>
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <algorithm>
#include <queue>
#include <map>
using namespace std;

const int PUZZLE_SIDE_LENGTH = 3;   // For a 3x3 puzzle. Can be changed for different puzzle sizes.

// Goal state; 0 is the blank tile. Can be changed for different puzzle sizes.
const vector<vector<int>> PUZZLE_GOAL = {{1, 2, 3},
                                         {4, 5, 6},
                                         {7, 8, 0}};

struct PuzzleNode {
    vector<vector<int>> puzzle;
    int x_blank, y_blank;   // Coordinates of the blank tile.
    int g = 0, h = 0;   // g = cost expended; h = heuristic value (distance to goal);
    PuzzleNode* parent; // Will be used to find the root node for solution tracing.

    PuzzleNode(vector<vector<int>> _p, int _x, int _y, int _g, int _h, PuzzleNode* _parent = nullptr)
        : puzzle(_p), x_blank(_x), y_blank(_y), g(_g), h(_h), parent(_parent) {}

    int f = g + h;  // Estimated cost of cheapest solution.

    // Overloads the > operator based on the f variable for the functionality of the queue.
    bool operator>(const PuzzleNode &_p_) const {
        return f > _p_.f;
    }
};

// Functions declarations.
void print_puzzle_prompt(vector<vector<int>>&, string&);
vector<vector<int>> init_premade_initial_puzzle(int);
int misplaced_tile(vector<vector<int>>);
int manhattan_distance(vector<vector<int>>);
void print_puzzle(vector<vector<int>>, int, int);
void print_puzzle_path(PuzzleNode*);
void print_summary(int, int, int);
void print_failure();
void general_search(vector<vector<int>>, string);

int main() {
    ios::sync_with_stdio(0);    // Fast input and output.

    // These two variables will get their respective values from print_puzzle_prompt().
    vector<vector<int>> initial_puzzle(PUZZLE_SIDE_LENGTH, vector<int>(PUZZLE_SIDE_LENGTH));
    string puzzle_heuristic_type;

    print_puzzle_prompt(initial_puzzle, puzzle_heuristic_type);

    // Keeps track of how long it takes to solve the puzzle.
    auto start = chrono::high_resolution_clock::now();
```

```
    general_search(initial_puzzle, puzzle_heuristic_type);
    auto end = chrono::high_resolution_clock::now();

    // Converts the time from microseconds to milliseconds.
    auto micro_duration = chrono::duration_cast<chrono::microseconds>(end - start);
    chrono::duration<double, milli> milli_duration = micro_duration;
    cout << "Time: " << milli_duration.count() << " milliseconds" << '\n' << '\n';

    return 0;
}

void print_puzzle_prompt(vector<vector<int>> &_puzzle, string &_heuristic_type) {
    cout << "Welcome! This is a program customized to solve an Eight Puzzle." << '\n'
         << "To get started, please do either one of the following:" << '\n'
         << "> Type \"1\" to choose a premade initial puzzle." << '\n'
         << "> Type \"2\" to create your own initial puzzle." << '\n'
         << "Press ENTER once you have typed your choice." << '\n';
    int puzzle_mode;
    cin >> puzzle_mode;
    cout << '\n';

    // Reasks the user if input was invalid.
    while (!(puzzle_mode == 1 || puzzle_mode == 2)) {
        cout << "Invalid input. Please try again." << '\n'
             << "Welcome! This is a program customized to solve an Eight Puzzle." << '\n'
             << "To get started, please do either one of the following:" << '\n'
             << "> Type \"1\" to choose a premade initial puzzle." << '\n'
             << "> Type \"2\" to create your own initial puzzle." << '\n'
             << "Press ENTER once you have typed your choice." << '\n';
        cin >> puzzle_mode;
        cout << '\n';
    }

    switch(puzzle_mode) {
        case 1:
            cout << "You have chosen to use a premade initial puzzle." << '\n'
                 << "Please type in an integer from 0 to 9." << '\n'
                 << "The number inputted will determine the DIFFICULTY to solve the puzzle." << '\n'
                 << "Press ENTER once you have typed your choice." << '\n';
            int puzzle_difficulty;
            cin >> puzzle_difficulty;
            cout << '\n';

            // Reasks the user if input was invalid.
            while (!(puzzle_difficulty >= 0 && puzzle_difficulty <= 9)) {
                cout << "Invalid input. Please try again." << '\n'
                     << "You have chosen to use a premade initial puzzle." << '\n'
                     << "Please type in an integer from 0 to 9." << '\n'
                     << "The number inputted will determine the DIFFICULTY to solve the puzzle." << '\n'
                     << "Press ENTER once you have typed your choice." << '\n';
                cin >> puzzle_difficulty;
                cout << '\n';
            }

            _puzzle = init_premade_initial_puzzle(puzzle_difficulty);
            break;
        case 2:
            cout << "You have chosen to create your own initial puzzle." << '\n'
                 << "Enter your puzzle, separating each number with a space in between." << '\n'
                 << "The blank tile should be represented with \"0\"." << '\n'
                 << "Please ensure that your puzzle is a legit 8-puzzle." << '\n'
                 << "Press ENTER when you finish. " << '\n' << '\n';

            for (int i = 0; i < PUZZLE_SIDE_LENGTH; i++) {  // Row and column size depends on PUZZLE_SIDE_LENGTH.
                cout << "Enter the numbers for row " << i+1 << ": ";
```

```
                for (int j = 0; j < PUZZLE_SIDE_LENGTH; j++) {
                    cin >> _puzzle[i][j];
                }
            }
            cout << '\n';
            break;
        default:
            break;  // Invalid input.
    }

    cout << "Please select an ALGORITHM by typing the number corresponding to the following: " << '\n'
         << "\"1\" for Uniform Cost Search" << '\n'
         << "\"2\" for Misplaced Tile Heuristic" << '\n'
         << "\"3\" for Manhattan Distance Heuristic" << '\n';
    int algorithm_type;
    cin >> algorithm_type;
    cout << '\n';

    // Reasks the user if input was invalid.
    while (!(algorithm_type == 1 || algorithm_type == 2 || algorithm_type == 3)) {
        cout << "Invalid input. Please try again." << '\n'
             << "Please select an ALGORITHM by typing the number corresponding to the following: " << '\n'
             << "\"1\" for Uniform Cost Search" << '\n'
             << "\"2\" for A* Misplaced Tile Heuristic" << '\n'
             << "\"3\" for A* Manhattan Distance Heuristic" << '\n';
        cin >> algorithm_type;
        cout << '\n';
    }

    switch (algorithm_type) {
        case 1:
            _heuristic_type = "Uniform Cost Search";
            break;
        case 2:
            _heuristic_type = "A* Misplaced";
            break;
        case 3:
            _heuristic_type = "A* Manhattan";
            break;
        default:
            break;  // Invalid input.
    }
    return;
}

vector<vector<int>> init_premade_initial_puzzle(int _difficulty) {  // Premade puzzles for testing algorithm.
    switch(_difficulty) {
        case 0:
            return {{1, 2, 3},  // Depth 0
                    {4, 5, 6},
                    {7, 8, 0}};
            break;
        case 1:
            return {{1, 2, 3},  // Depth 1
                    {4, 5, 0},
                    {7, 8, 6}};
            ;
        case 2:
            return {{1, 0, 3},  // Depth 3
                    {4, 2, 6},
                    {7, 5, 8}};
            break;
        case 3:
            return {{2, 3, 0},  // Depth 6
                    {1, 4, 5},
```

```
                {7, 8, 6}};
            break;
        case 4:
            return {{1, 2, 3},  // Depth 9
                    {0, 5, 7},
                    {4, 8, 6}};
            break;
        case 5:
            return {{1, 2, 4},  // Depth 12
                    {7, 5, 3},
                    {0, 8, 6}};
            break;
        case 6:
            return {{7, 4, 3},  // Depth 15
                    {0, 2, 1},
                    {8, 5, 6}};
            break;
        case 7:
            return {{0, 6, 3},  // Depth 18
                    {1, 2, 7},
                    {5, 4, 8}};
            break;
        case 8:
            return {{8, 2, 6},  // Depth 21
                    {1, 7, 0},
                    {5, 4, 3}};
            break;
        case 9:
            return {{4, 6, 0},  // Depth 24
                    {1, 5, 8},
                    {7, 2, 3}};
            break;
        default:
            return {{1, 2, 3},  // Invalid input.
                    {4, 5, 6},
                    {7, 8, 0}};
            break;
    }
}

int misplaced_tile(vector<vector<int>> _p) {
    int dist = 0;
    for (int i = 0; i < PUZZLE_SIDE_LENGTH; i++) {
        for (int j = 0; j < PUZZLE_SIDE_LENGTH; j++) {
            if (_p[i][j] == 0) {
                continue; // Skips the blank tile.
            }
            if (_p[i][j] != PUZZLE_GOAL[i][j]) {
                dist++;
            }
        }
    }
    return dist;
}

int manhattan_distance(vector<vector<int>> _p) {
    int dist = 0;
    for (int i = 0; i < PUZZLE_SIDE_LENGTH; i++) {
        for (int j = 0; j < PUZZLE_SIDE_LENGTH; j++) {
            if (_p[i][j] == 0) {
                continue; // Skips the blank tile.
            }
            for (int x = 0; x < PUZZLE_SIDE_LENGTH; x++) {
                for (int y = 0; y < PUZZLE_SIDE_LENGTH; y++) {
                    if (_p[i][j] == PUZZLE_GOAL[x][y]) {
```

```
                    dist += abs(i-x) + abs(j-y);    // Distance between matching tiles (can't be negative).
                }
            }
        }
    }
    return dist;
}

void print_puzzle(vector<vector<int>> _p, int _g, int _h) {
    cout << "The best node to expand with a g(n) = " << _g << " and h(n) = " << _h << " is:" << '\n';
    for (int i = 0; i < PUZZLE_SIDE_LENGTH; i++) {
        cout << "[";
        for (int j = 0; j < PUZZLE_SIDE_LENGTH; j++) {
            cout << _p[i][j];
            if (j < PUZZLE_SIDE_LENGTH-1) {
                cout << ", ";
            }
        }
        cout << "]" << '\n';
    }
}

void print_puzzle_path(PuzzleNode* _goal) {
    vector<PuzzleNode*> puzzle_path;
    PuzzleNode* curr = _goal;

    while (curr != nullptr) {   // Traces back to root node.
        puzzle_path.push_back(curr);
        curr = curr->parent;
    }

    // Makes the path start with the root and end with the goal.
    reverse(puzzle_path.begin(), puzzle_path.end());

    for (PuzzleNode* node : puzzle_path) {
        print_puzzle(node->puzzle, node->g, node->h);
    }
}

void print_summary(int _g, int _n, int _s) {
    cout << "Goal state!" << '\n' << '\n'
         << "Solution depth was " << _g << '\n'
         << "Number of nodes expanded: " << _n << '\n'
         << "Max queue size: " << _s << '\n' << '\n';
}

void print_failure() {
    cout << "Unfortunately, your puzzle is impossible to solve." << '\n' << '\n';
}

void general_search(vector<vector<int>> problem, string heuristic_type) {
    int x_blank, y_blank;   // Coordinates for the blank tile.
    for (int i = 0; i < PUZZLE_SIDE_LENGTH; i++) {  // Locate the blank tile.
        for (int j = 0; j < PUZZLE_SIDE_LENGTH; j++) {
            if (problem[i][j] == 0) {
                x_blank = i;
                y_blank = j;
                break;
            }
        }
    }

    // Set heuristic type for root_node (original problem/puzzle).
    int heuristic = (heuristic_type == "A* Manhattan") ? manhattan_distance(problem)
```

```
                : (heuristic_type == "A* Misplaced") ? misplaced_tile(problem)
                : 0;  // Uniform Cost Search has a heuristic value of 0.

    // Creates a queue that prioritizes SMALLER costs of cheapest solutions first for the puzzle's nodes.
    priority_queue<PuzzleNode, vector<PuzzleNode>, greater<PuzzleNode>> q;
    PuzzleNode root_node = PuzzleNode(problem, x_blank, y_blank, 0, heuristic, nullptr);
    q.push(root_node);

    // Remembers visited puzzle nodes as keys and booleans as values.
    map<vector<vector<int>>, bool> puzzle_visits;

    const vector<int> move_x = {-1, 1, 0, 0};  // "Up" and "Down" moves.
    const vector<int> move_y = {0, 0, -1, 1};  // "Left" and "Right" moves.

    int nodes_expanded = 0, q_max_size = 1;
    while (!q.empty()) {
        q_max_size = max(q_max_size, (int)q.size());    // Track largest queue size.

        PuzzleNode curr = q.top();
        q.pop();
        nodes_expanded++;

        if (puzzle_visits[curr.puzzle]) {   // Skip already visited nodes.
            continue;
        }
        puzzle_visits[curr.puzzle] = true;  // Remembers this node.

        if (curr.puzzle == PUZZLE_GOAL) {   // Successful search.
            print_puzzle_path(&curr);
            print_summary(curr.g, nodes_expanded, q_max_size);
            return;
        }

        for (int m = 0; m < 4; m++) {   // Considers all possible moves for the blank tile.
            int new_x_blank = curr.x_blank + move_x[m];
            int new_y_blank = curr.y_blank + move_y[m];

            if (new_x_blank < 0 || new_x_blank >= PUZZLE_SIDE_LENGTH || new_y_blank < 0 || new_y_blank >= PUZZLE_SIDE_LENGTH) {
                continue;   // Ignore tiles that are out-of-bounds.
            }

            vector<vector<int>> new_puzzle = curr.puzzle;
            swap(new_puzzle[curr.x_blank][curr.y_blank], new_puzzle[new_x_blank][new_y_blank]);

            // Makes sure new_puzzle has matching heuristic type.
            heuristic = (heuristic_type == "A* Manhattan") ? manhattan_distance(new_puzzle)
                      : (heuristic_type == "A* Misplaced") ? misplaced_tile(new_puzzle)
                      : 0;  // Uniform Cost Search has a heuristic value of 0.

            q.push(PuzzleNode(new_puzzle, new_x_blank, new_y_blank, curr.g+1, heuristic, new PuzzleNode(curr)));
        }
    }
    print_failure();    // Unsuccessful search.
    return;
}
```