CS205: Artificial Intelligence, Dr. Eamonn Keogh

Project 1

Huy Dinh Tran
SID: 862325308
Email: htran197@ucr.edu
Date: 05/13/2022

GitHub Repo: https://github.com/huydinhtran/8-Puzzle-Solver

In completing this assignment I consulted:

- All of the lecture slides and undergraduate video lectures from Dr. Eamonn Keogh
- C++ documentation from https://www.cplusplus.com/, programming Q&A forum https://stackoverflow.com/, and as well as programming tutorial websites such as https://www.geeksforgeeks.org/ and https://www.w3schools.com/

All important core code is written by myself originally. Here is a list of mandatory and additional libraries which have functions that were used to assist me implementing the algorithm.

- #include <stdio.h>
- #include <iostream>
- #include <algorithm>
- #include <queue>
- #include <vector>
- #include <list>
- #include <string>
- #include <cstdlib>
- #include <chrono>

Outline of the report:

CS205 Project 1
The 8-Puzzle Solver

Huy Dinh Tran, SID: 862325308

Introduction

The 8-puzzle is a 3 by 3 sliding tile puzzle that consists of 8 square tiles that are numbered from 1 to 8 and an empty tile. The tiles will be randomly shuffled by moving each square tile into the empty tile. The goal of this puzzle is to revert the board back to its original state which counts from 1 to 8 starting from row 1, column 1 down to the empty tile in row 3, column 3. Figure 1 shows an example of the initial state and the goal state.

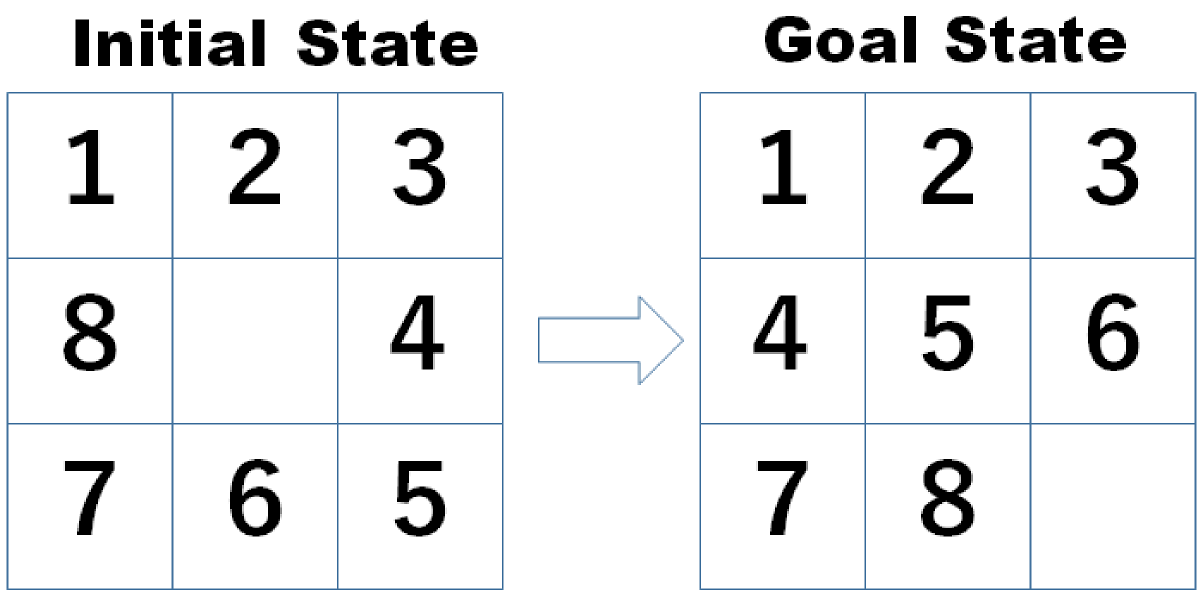

Figure 1. Initial state vs goal state of the 8-puzzle

In this project that was assigned by Dr. Eamonn Keogh, we are assigned to implement a program to solve the 8-puzzle. There are 3 main searching algorithms to implement which are Uniform Cost Search, A-star with Misplaced Tile Heuristic, and A-star with Manhattan Distance Heuristic. I implemented the algorithm using C++ and screenshots of my source code are included in the last section of this report.

Comparison of Algorithms

Uniform Cost Search

As mentioned in lectures and the project handout, Uniform Cost Search is similar to an A-star search but with the heuristic set as 0. We only care about the movement cost when determining which node to expand which is always 1 for each depth. With this in mind, since each movement has the same cost, we can view this search as a Breath-First Search for this particular puzzle.

A-Star with Misplaced Tile Heuristic (Hamming)

A-Star is a very optimal and complete searching algorithm that can search straight to the goal node by using a calculated heuristic. For this version of A-star, we are using the Misplaced Tile Heuristic which compares each tile of the current board to the goal board. Each misplaced tile will increment the heuristic number, as well as the movement cost, which is also added to the equation. When selecting a node to expand, we will compare all of the heuristics of all the nodes within the queue, and the one with the smallest heuristic number will be chosen to expand.

A-Star with Manhattan Distance Heuristic

For the Manhattan Distance Heuristic, it is quite similar to the Misplaced Tile Heuristic but it gives us a little more detail on the closeness of the state. For each tile, it calculates the minimum number of steps to reach the goal tile. For example, tile 1 should be at row 0, column 0 but the current board has tile 1 at row 2, column 2. The distance to reach the goal position is 4 which is going up 2 steps and left 2 steps. We would calculate for all of the tiles and output the heuristic number. Similar to the Misplaced Tile Heuristic, the node with the smallest heuristic will be chosen to expand. Figure 2 gives examples of the calculation of both heuristics.

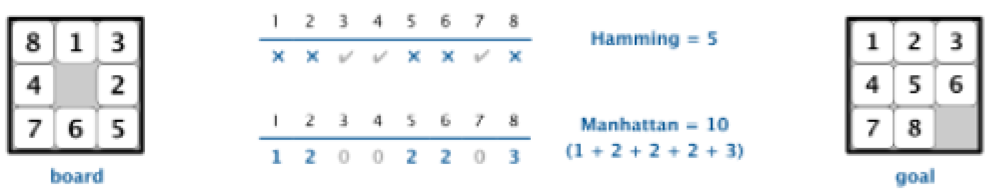

Figure 2. Calculations for Misplaced Tile and Manhattan Distance Heuristic

Comparison of Algorithms on Sample Puzzle

I would be using the test cases provided by Dr. Eamon Keogh shown in Figure 3 to run and graph the results for comparison. For all the runs, I will eliminate printing the puzzle each iteration to give us precise execution time measurements. I included the graph and the result table below.

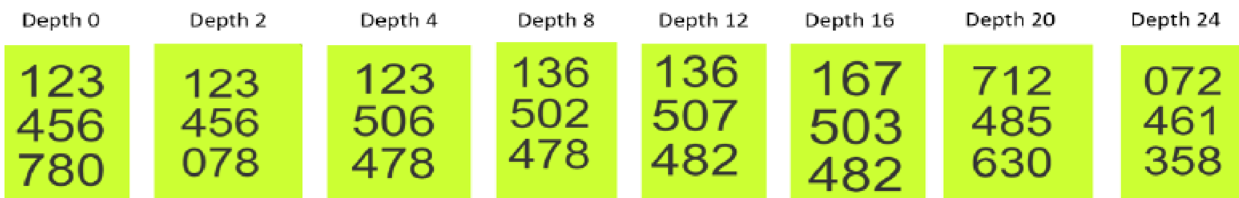

Figure 3. Provided test cases with depths

Total Node Expanded								
Depth	**0**	**2**	**4**	**8**	**12**	**16**	**20**	**24**
Uni Cost Search	0	10	997	542901	23285	542901		
A* Misplace Tile	0	5	34	295	159	908	32795	30488
A* Manhattan Distance	0	5	25	104	53	84	1763	144

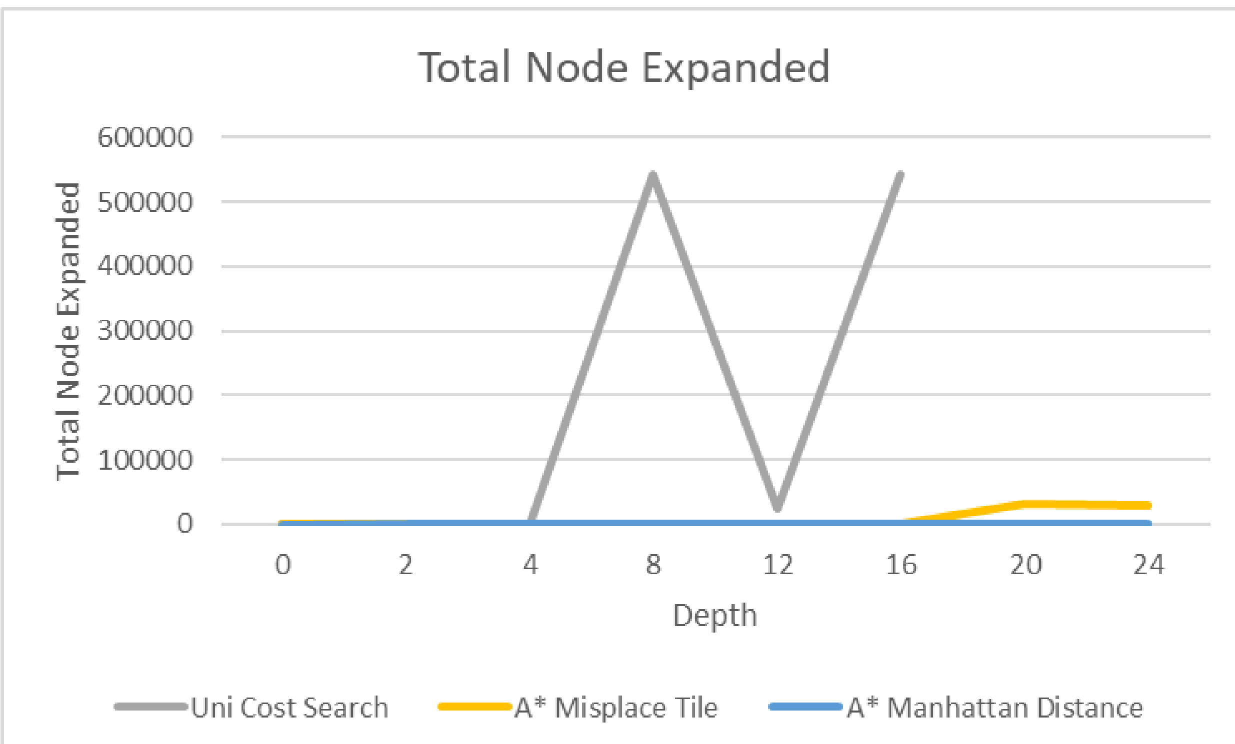

Figure 4. Total node expanded comparison between 3 searching algorithms

Total Queue Size								
Depth	**0**	**2**	**4**	**8**	**12**	**16**	**20**	**24**
Uni Cost Search	1	8	792	430464	18464	430464		
A* Misplace Tile	1	4	25	229	132	721	26391	23412
A* Manhattan Distance	1	4	20	82	45	71	1385	119

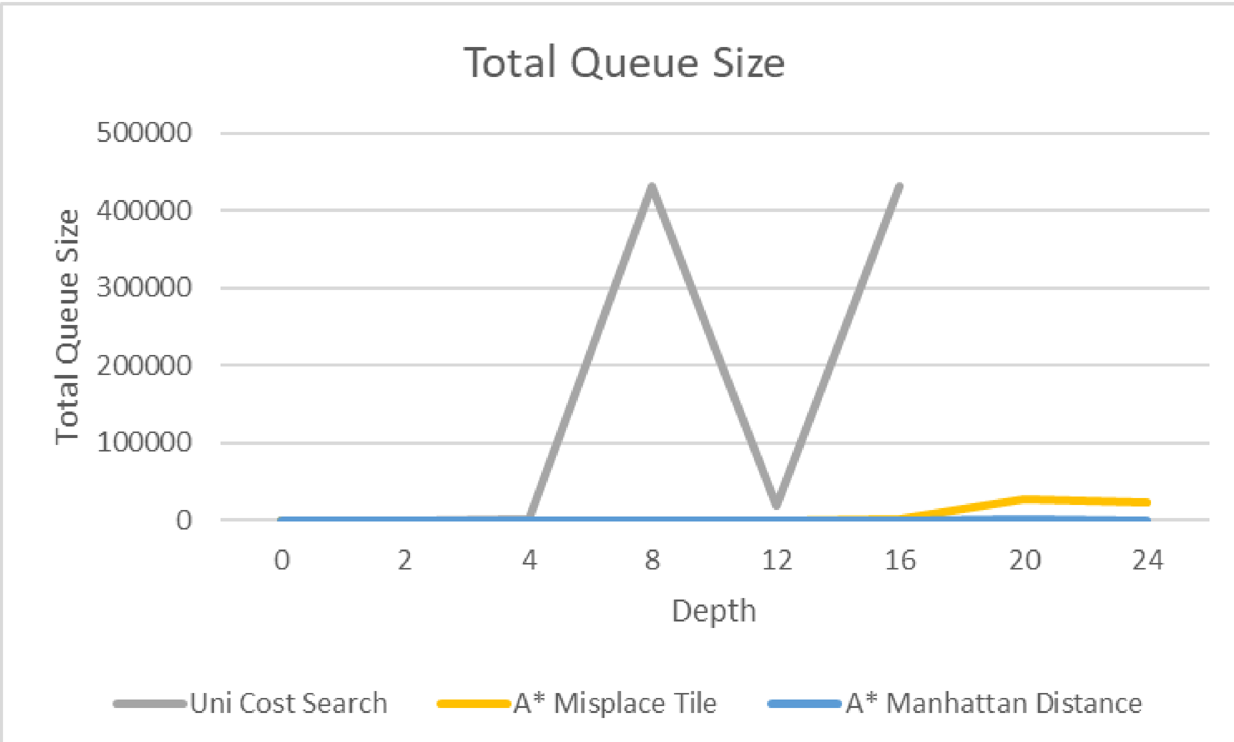

Figure 5. Total queue size comparison between 3 searching algorithms

Execution Time (in ms)								
Depth	**0**	**2**	**4**	**8**	**12**	**16**	**20**	**24**
Uni Cost Search	0.3	0.3	1.7	45243.4	108.932	45280.3		
A* Misplace Tile	0.3	0.3	0.4	0.6	0.5	1.3	40.9	40.7
A* Manhattan Distance	0.4	0.3	0.4	0.4	0.4	0.4	2.1	0.6

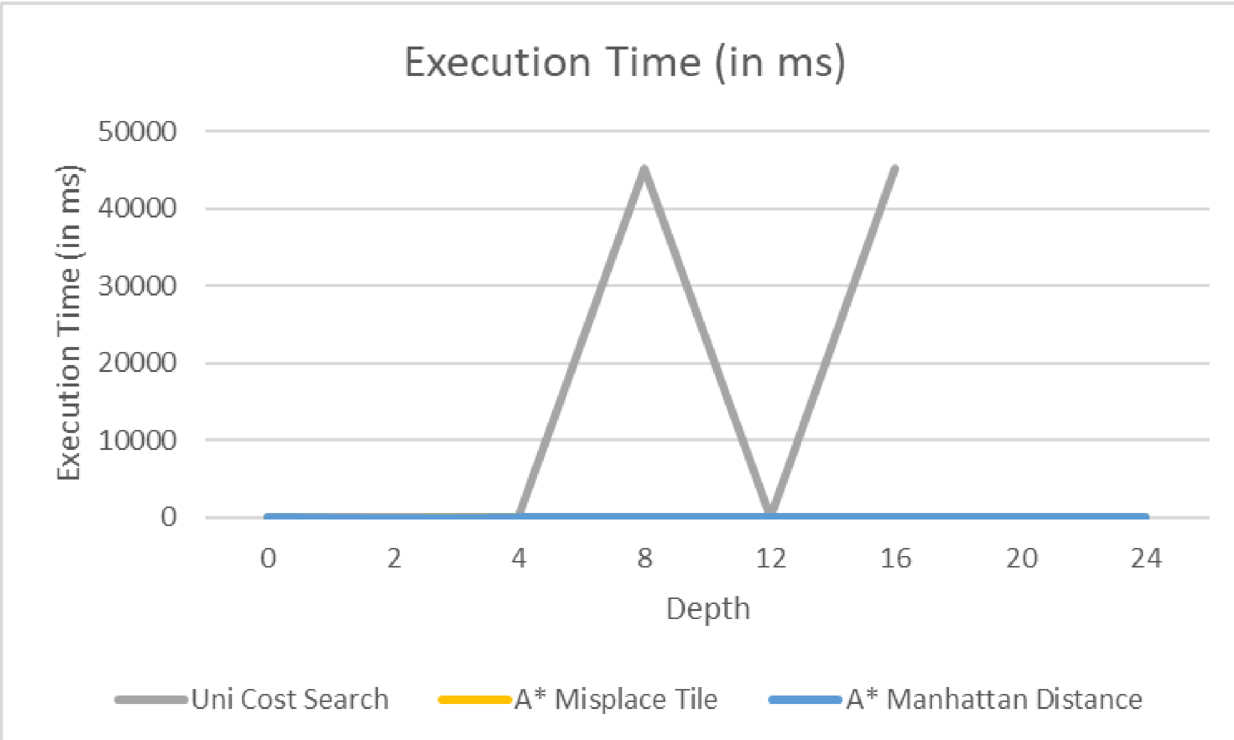

Figure 6. Execution time (in ms) comparison between 3 searching algorithms

Looking at Figures 4, 5, and 6, we can see the big difference in terms of the performance of Uniform Cost Search and A-star Search algorithms. This is due to A-star searches having heuristics that help the algorithm to know how close the state is to the goal node. This gives many hints to search all the way down the tree without caring about other nodes that have higher heuristics. Out of the 2 heuristic calculations, the Manhattan Distance Heuristic is a better implementation since it gives a better and more precise measurement of the distance to the goal state.

However, I have to admit that my implementation of the algorithm is not the best so it gives us results that are not in the range of the theoretical expectation. I could have got the logic wrong or my calculation is off. But, my implementation can still search for the goal state and display each algorithm's strength or weakness.

Conclusion

The good sides of the 3 searching algorithms are they are both complete and optimal by a degree. However, after taking measurements and data of the test cases for the 3 algorithms, we can see clearly that both A-star searching algorithms run much more optimal compared to the Uniform Cost Search algorithm. One of the main reasons is the A-star algorithms have an idea of the distance or closeness to the goal state by calculating the heuristic numbers while Uniform Cost Search is just expanding every child node. We can also conclude that A-star with Manhattan Distance Heuristic performs quite better compared to the Misplaced Tile Heuristic. This is because the calculation of the Manhattan Distance Heuristic gives the algorithm a better and much more precise sense of whether the current state is close to the goal state or not. After working on this project, we can conclude that the A-star search is probably the best, most optimal searching algorithm out of all the algorithms we learn in class.

Sample trace on an easy problem (depth 4) using A-star Manhattan Distance Heuristic

```
Welcome to Huy Dinh Tran (862325308, htran197) 8-Puzzle Solver.
Type '1' to to use a default puzzle, or '2' to create your own.
Default puzzle: [ 1 2 3 ]
                [ 5 0 6 ]
                [ 4 7 8 ]
1
Select algorithm:
(1) for Uniform Cost Search
(2) for the Misplaced Tile Heuristic
(3) the Manhattan Distance Heuristic
3
g(n) = 0 and h(n) = 0
[ 1 2 3 ]
[ 5 0 6 ]
[ 4 7 8 ]
g(n) = 1 and h(n) = 3
[ 1 2 3 ]
[ 0 5 6 ]
[ 4 7 8 ]
g(n) = 2 and h(n) = 2
[ 1 2 3 ]
[ 4 5 6 ]
[ 0 7 8 ]
g(n) = 2 and h(n) = 2
[ 1 2 3 ]
[ 4 5 6 ]
[ 0 7 8 ]
g(n) = 3 and h(n) = 1
[ 1 2 3 ]
[ 4 5 6 ]
[ 7 0 8 ]
g(n) = 3 and h(n) = 1
[ 1 2 3 ]
[ 4 5 6 ]
[ 7 0 8 ]
g(n) = 4 and h(n) = 0
[ 1 2 3 ]
[ 4 5 6 ]
[ 7 8 0 ]
Solution found!
Solution depth: 4
Number of nodes expanded: 25
Max queue size: 20
Execution time (seconds): 9.7355 ms
```

Sample trace on a hard problem (depth 16) using A-star Manhattan Distance Heuristic

```
Welcome to Huy Dinh Tran (862325308, htran197) 8-Puzzle Solver.
Type '1' to to use a default puzzle, or '2' to create your own.
Default puzzle: [ 1 2 3 ]
                [ 5 0 6 ]
                [ 4 7 8 ]
2
Enter your puzzle, using a zero to represent the blank. Please only enter valid 8-puzzles.
Enter the puzzle demilimiting the numbers with a space. Press ENTER only when finished.
Enter the first row: 1 6 7
Enter the second row: 5 0 3
Enter the third row: 4 8 2
Select algorithm:
(1) for Uniform Cost Search
(2) for the Misplaced Tile Heuristic
(3) the Manhattan Distance Heuristic
3
g(n) = 0 and h(n) = 0
[ 1 6 7 ]
[ 5 0 3 ]
[ 4 8 2 ]
g(n) = 1 and h(n) = 11
[ 1 6 7 ]
[ 0 5 3 ]
[ 4 8 2 ]
g(n) = 1 and h(n) = 11
[ 1 0 7 ]
[ 5 6 3 ]
[ 4 8 2 ]
g(n) = 2 and h(n) = 10
[ 1 6 7 ]
[ 4 5 3 ]
[ 0 8 2 ]
g(n) = 2 and h(n) = 10
[ 1 7 0 ]
[ 5 6 3 ]
[ 4 8 2 ]
```

………. Deleted few pages to save space

```
g(n) = 6 and h(n) = 4
[ 1 2 3 ]
[ 5 7 6 ]
[ 4 8 0 ]
g(n) = 7 and h(n) = 2
[ 1 2 3 ]
[ 7 5 6 ]
[ 4 8 0 ]
g(n) = 8 and h(n) = 0
[ 1 2 3 ]
[ 4 5 6 ]
[ 7 8 0 ]
Solution found!
Solution depth: 8
Number of nodes expanded: 84
Max queue size: 71
Execution time (seconds): 19.6336 ms
```

Source code

```cpp
/*
Huy Dinh Tran
ID: 862325308
Email: htran197@ucr.edu

CS205 AI Spring 2022
Project 1
Dr. Eamonn Keogh
*/
#include <stdio.h>
#include <iostream>
#include <algorithm>
#include <queue>
#include <vector>
#include <list>
#include <string>
#include <cstdlib>
#include <chrono>

using namespace std;

struct Node { //struct of a Node which has pointers to childs and parent with attribute of heuristic, movement cost and location of the blank within matrix.
    int matrix[3][3];
    Node* child1;
    Node* child2;
    Node* child3;
    Node* child4;
    Node* parent;
    int blankX, blankY;
    int hMan;
    int hTile;
    int moveCost;
};

struct compareHTile { //used for priority_queue, place lowest MisTile heuristic at the top
    bool operator()(Node* &a,  Node* &b)
    {
        return a->hTile > b->hTile;
    }
};

struct compareUni { //used for priority_queue, place lowest movement cost at the top
    bool operator()(Node*& a, Node*& b)
    {
        return a->moveCost > b->moveCost;
    }
};
```

```cpp
struct compareHMan { //used for priority_queue, place lowest Manhattan heuristic at the top
    bool operator()(const Node* a, const Node* b) const
    {
        return a->hMan > b->hMan;
    }
};

bool checkGoal(int input[3][3], int goal[3][3]) { //return true if input matrix the same as goal matrix, else, return false
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (input[i][j] != goal[i][j]) {
                return false;
            }
        }
    }
    return true;
}

int hMisTile(int input[3][3], int goal[3][3]) { //return the heuristic value for misplace tiles
    int h=0;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (input[i][j] != goal[i][j] && goal[i][j] != 0)
                h++;
        }
    }
    return h;
}
```

```cpp
int hManhattan(int input[3][3], int goal[3][3]) { //return the heuristic value for Manhattan
    int h = 0;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (input[i][j] != goal[i][j]) {
                switch (input[i][j]) {
                    case 1:
                        h += abs(i - 0) + abs(j - 0);
                        break;
                    case 2:
                        h += abs(i - 0) + abs(j - 1);
                        break;
                    case 3:
                        h += abs(i - 0) + abs(j - 2);
                        break;
                    case 4:
                        h += abs(i - 1) + abs(j - 0);
                        break;
                    case 5:
                        h += abs(i - 1) + abs(j - 1);
                        break;
                    case 6:
                        h += abs(i - 1) + abs(j - 2);
                        break;
                    case 7:
                        h += abs(i - 2) + abs(j - 0);
                        break;
                    case 8:
                        h += abs(i - 2) + abs(j - 1);
                        break;
                }
            }
        }
    }
    return h;
}

void printBoard(int input[3][3]) { //print the current board
    for (int i = 0; i < 3; i++) {
        cout << "[ ";
        for (int j = 0; j < 3; j++) {
            cout << input[i][j] << " ";
        }
        cout << "]" << endl;
    }
}
```

```cpp
void copyBoard(int (&target)[3][3], int (&matrix)[3][3]) { //copying matrix from input to target
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            target[i][j] = matrix[i][j];
        }
    }
}

bool isDup(vector<Node*>vec, Node* input) { //searching through vector to see if there is any duplication
    for (int i = 0; i < vec.size(); i++) {
        if (vec[i]->matrix == input->matrix) {
            return true;
        }
    }
    return false;
}

void assignChild1(Node* &curr, int (&childMatrix)[3][3], int blankX, int blankY, int &nodeExpand) { //making child1
    Node* temp = new Node;
    int goal[3][3] = { {1,2,3},{4,5,6},{7,8,0} };
    temp->moveCost = curr->moveCost + 1;
    copyBoard(temp->matrix, childMatrix);
    temp->parent = curr;
    temp->blankX = blankX;
    temp->blankY = blankY;
    temp->hMan = hManhattan(childMatrix,goal) + temp->moveCost;
    temp->hTile = hMisTile(childMatrix, goal) + temp->moveCost;
    nodeExpand++;
    curr->child1 = temp;
}

void assignChild2(Node* &curr, int (&childMatrix)[3][3], int blankX, int blankY, int& nodeExpand) { //making child2
    Node* temp = new Node;
    int goal[3][3] = { {1,2,3},{4,5,6},{7,8,0} };
    temp->moveCost = curr->moveCost + 1;
    copyBoard(temp->matrix, childMatrix);
    temp->parent = curr;
    temp->blankX = blankX;
    temp->blankY = blankY;
    temp->hMan = hManhattan(childMatrix, goal) + temp->moveCost;
    temp->hTile = hMisTile(childMatrix, goal) + temp->moveCost;
    nodeExpand++;
    curr->child2 = temp;
}
```

```cpp
void assignChild3(Node* &curr, int (&childMatrix)[3][3], int blankX, int blankY, int& nodeExpand) { //making child3
    Node* temp = new Node;
    int goal[3][3] = { {1,2,3},{4,5,6},{7,8,0} };
    temp->moveCost = curr->moveCost + 1;
    copyBoard(temp->matrix, childMatrix);
    temp->parent = curr;
    temp->blankX = blankX;
    temp->blankY = blankY;
    temp->hMan = hManhattan(childMatrix, goal) + temp->moveCost;
    temp->hTile = hMisTile(childMatrix, goal) + temp->moveCost;
    nodeExpand++;
    curr->child3 = temp;
}

void assignChild4(Node* &curr, int (&childMatrix)[3][3], int blankX, int blankY, int& nodeExpand) { //making child4
    Node* temp = new Node;
    int goal[3][3] = { {1,2,3},{4,5,6},{7,8,0} };
    temp->moveCost = curr->moveCost + 1;
    copyBoard(temp->matrix, childMatrix);
    temp->parent = curr;
    temp->blankX = blankX;
    temp->blankY = blankY;
    temp->hMan = hManhattan(childMatrix, goal) + temp->moveCost;
    temp->hTile = hMisTile(childMatrix, goal) + temp->moveCost;
    nodeExpand++;
    curr->child4 = temp;
}
```

```cpp
void moveBlankHTile(Node* &node, priority_queue <Node*, vector<Node*>, compareHTile> &q, int &nodeExpand, int &queueSize)  {//moving empty tile, creating childs and also pushing into queue for A*MisTile
    int temp[3][3];
    switch (node->blankX) {
    case 0:
        switch (node->blankY) {
        case 0:
            copyBoard(temp, node->matrix);
            swap(temp[0][0], temp[0][1]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 0, 1, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[0][0], temp[1][0]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 1, 0, nodeExpand);
                q.push(node->child2);
            }
            break;
        case 1:
            copyBoard(temp, node->matrix);
            swap(temp[0][1], temp[0][0]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 0, 0, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[0][1], temp[0][2]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 0, 2, nodeExpand);
                q.push(node->child2);
            }
            copyBoard(temp, node->matrix);
            swap(temp[0][1], temp[1][1]);
            if (node->parent->matrix != temp) {
                assignChild3(node, temp, 1, 1, nodeExpand);
                q.push(node->child3);
            }
            break;
```

```cpp
        case 2:
            copyBoard(temp, node->matrix);
            swap(temp[0][2], temp[0][1]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 0, 1, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[0][2], temp[1][2]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 1, 2, nodeExpand);
                q.push(node->child2);
            }
            break;
        }
    case 1:
        switch (node->blankY) {
        case 0:
            copyBoard(temp, node->matrix);
            swap(temp[1][0], temp[0][0]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 0, 0, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[1][0], temp[1][1]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 1, 1, nodeExpand);
                q.push(node->child2);
            }
            copyBoard(temp, node->matrix);
            swap(temp[1][0], temp[2][0]);
            if (node->parent->matrix != temp) {
                assignChild3(node, temp, 2, 0, nodeExpand);
                q.push(node->child3);
            }
            break;
```

```cpp
        case 1:
            copyBoard(temp, node->matrix);
            swap(temp[1][1], temp[1][0]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 1, 0, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[1][1], temp[0][1]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 0, 1, nodeExpand);
                q.push(node->child2);
            }
            copyBoard(temp, node->matrix);
            swap(temp[1][1], temp[1][2]);
            if (node->parent->matrix != temp) {
                assignChild3(node, temp, 1, 2, nodeExpand);
                q.push(node->child3);
            }
            copyBoard(temp, node->matrix);
            swap(temp[1][1], temp[2][1]);
            if (node->parent->matrix != temp) {
                assignChild4(node, temp, 2, 1, nodeExpand);
                q.push(node->child4);
            }
            break;
        case 2:
            copyBoard(temp, node->matrix);
            swap(temp[1][2], temp[1][1]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 1, 1, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[1][2], temp[0][2]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 0, 2, nodeExpand);
                q.push(node->child2);
            }
            copyBoard(temp, node->matrix);
            swap(temp[1][2], temp[2][2]);
            if (node->parent->matrix != temp) {
                assignChild3(node, temp, 2, 2, nodeExpand);
                q.push(node->child3);
            }
            break;
        }
```

```cpp
    case 2:
        switch (node->blankY) {
        case 0:
            copyBoard(temp, node->matrix);
            swap(temp[2][0], temp[1][0]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 1, 0, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[2][0], temp[2][1]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 2, 1, nodeExpand);
                q.push(node->child2);
            }
            break;
        case 1:
            copyBoard(temp, node->matrix);
            swap(temp[2][1], temp[2][0]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 2, 0, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[2][1], temp[1][1]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 1, 1, nodeExpand);
                q.push(node->child2);
            }
            copyBoard(temp, node->matrix);
            swap(temp[2][1], temp[2][2]);
            if (node->parent->matrix != temp) {
                assignChild3(node, temp, 2, 2, nodeExpand);
                q.push(node->child3);
            }
            break;
```

```cpp
        case 2:
            copyBoard(temp, node->matrix);
            swap(temp[2][2], temp[2][1]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 2, 1, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[2][2], temp[1][2]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 1, 2, nodeExpand);
                q.push(node->child2);
            }
            break;
        }
    }
}
```

```cpp
void moveBlankHMan(Node*& node, priority_queue <Node*, vector<Node*>, compareHMan> &q, int& nodeExpand, int& queueSize) { //moving empty tile, creating childs and also pushing into queue for A*Manhattan
    int temp[3][3];
    switch (node->blankX) {
    case 0:
        switch (node->blankY) {
        case 0:
            copyBoard(temp, node->matrix);
            swap(temp[0][0], temp[0][1]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 0, 1, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[0][0], temp[1][0]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 1, 0, nodeExpand);
                q.push(node->child2);
            }
            break;
        case 1:
            copyBoard(temp, node->matrix);
            swap(temp[0][1], temp[0][0]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 0, 0, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[0][1], temp[0][2]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 0, 2, nodeExpand);
                q.push(node->child2);
            }
            copyBoard(temp, node->matrix);
            swap(temp[0][1], temp[1][1]);
            if (node->parent->matrix != temp) {
                assignChild3(node, temp, 1, 1, nodeExpand);
                q.push(node->child3);
            }
            break;
```

```cpp
            case 2:
                copyBoard(temp, node->matrix);
                swap(temp[0][2], temp[0][1]);
                if (node->parent->matrix != temp) {
                    assignChild1(node, temp, 0, 1, nodeExpand);
                    q.push(node->child1);
                }
                copyBoard(temp, node->matrix);
                swap(temp[0][2], temp[1][2]);
                if (node->parent->matrix != temp) {
                    assignChild2(node, temp, 1, 2, nodeExpand);
                    q.push(node->child2);
                }
                break;
            }
        case 1:
            switch (node->blankY) {
            case 0:
                copyBoard(temp, node->matrix);
                swap(temp[1][0], temp[0][0]);
                if (node->parent->matrix != temp) {
                    assignChild1(node, temp, 0, 0, nodeExpand);
                    q.push(node->child1);
                }
                copyBoard(temp, node->matrix);
                swap(temp[1][0], temp[1][1]);
                if (node->parent->matrix != temp) {
                    assignChild2(node, temp, 1, 1, nodeExpand);
                    q.push(node->child2);
                }
                copyBoard(temp, node->matrix);
                swap(temp[1][0], temp[2][0]);
                if (node->parent->matrix != temp) {
                    assignChild3(node, temp, 2, 0, nodeExpand);
                    q.push(node->child3);
                }
                break;
```

```cpp
            case 1:
                copyBoard(temp, node->matrix);
                swap(temp[1][1], temp[1][0]);
                if (node->parent->matrix != temp) {
                    assignChild1(node, temp, 1, 0, nodeExpand);
                    q.push(node->child1);
                }
                copyBoard(temp, node->matrix);
                swap(temp[1][1], temp[0][1]);
                if (node->parent->matrix != temp) {
                    assignChild2(node, temp, 0, 1, nodeExpand);
                    q.push(node->child2);
                }
                copyBoard(temp, node->matrix);
                swap(temp[1][1], temp[1][2]);
                if (node->parent->matrix != temp) {
                    assignChild3(node, temp, 1, 2, nodeExpand);
                    q.push(node->child3);
                }
                copyBoard(temp, node->matrix);
                swap(temp[1][1], temp[2][1]);
                if (node->parent->matrix != temp) {
                    assignChild4(node, temp, 2, 1, nodeExpand);
                    q.push(node->child4);
                }
                break;
            case 2:
                copyBoard(temp, node->matrix);
                swap(temp[1][2], temp[1][1]);
                if (node->parent->matrix != temp) {
                    assignChild1(node, temp, 1, 1, nodeExpand);
                    q.push(node->child1);
                }
                copyBoard(temp, node->matrix);
                swap(temp[1][2], temp[0][2]);
                if (node->parent->matrix != temp) {
                    assignChild2(node, temp, 0, 2, nodeExpand);
                    q.push(node->child2);
                }
                copyBoard(temp, node->matrix);
                swap(temp[1][2], temp[2][2]);
                if (node->parent->matrix != temp) {
                    assignChild3(node, temp, 2, 2, nodeExpand);
                    q.push(node->child3);
                }
                break;
            }
        case 2:
```

```cpp
        case 2:
            switch (node->blankY) {
            case 0:
                copyBoard(temp, node->matrix);
                swap(temp[2][0], temp[1][0]);
                if (node->parent->matrix != temp) {
                    assignChild1(node, temp, 1, 0, nodeExpand);
                    q.push(node->child1);
                }
                copyBoard(temp, node->matrix);
                swap(temp[2][0], temp[2][1]);
                if (node->parent->matrix != temp) {
                    assignChild2(node, temp, 2, 1, nodeExpand);
                    q.push(node->child2);
                }
                break;
            case 1:
                copyBoard(temp, node->matrix);
                swap(temp[2][1], temp[2][0]);
                if (node->parent->matrix != temp) {
                    assignChild1(node, temp, 2, 0, nodeExpand);
                    q.push(node->child1);
                }
                copyBoard(temp, node->matrix);
                swap(temp[2][1], temp[1][1]);
                if (node->parent->matrix != temp) {
                    assignChild2(node, temp, 1, 1, nodeExpand);
                    q.push(node->child2);
                }
                copyBoard(temp, node->matrix);
                swap(temp[2][1], temp[2][2]);
                if (node->parent->matrix != temp) {
                    assignChild3(node, temp, 2, 2, nodeExpand);
                    q.push(node->child3);
                }
                break;
```

```cpp
            case 2:
                copyBoard(temp, node->matrix);
                swap(temp[2][2], temp[2][1]);
                if (node->parent->matrix != temp) {
                    assignChild1(node, temp, 2, 1, nodeExpand);
                    q.push(node->child1);
                }
                copyBoard(temp, node->matrix);
                swap(temp[2][2], temp[1][2]);
                if (node->parent->matrix != temp) {
                    assignChild2(node, temp, 1, 2, nodeExpand);
                    q.push(node->child2);
                }
                break;
            }
        }
}
```

```cpp
void moveBlankUni(Node*& node, priority_queue <Node*, vector<Node*>, compareUni> &q, int& nodeExpand, int& queueSize) { //moving empty tile, creating childs and also pushing into queue for UniCostSearch
    int temp[3][3];
    switch (node->blankX) {
    case 0:
        switch (node->blankY) {
        case 0:
            copyBoard(temp, node->matrix);
            swap(temp[0][0], temp[0][1]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 0, 1, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[0][0], temp[1][0]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 1, 0, nodeExpand);
                q.push(node->child2);
            }
            break;
        case 1:
            copyBoard(temp, node->matrix);
            swap(temp[0][1], temp[0][0]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 0, 0, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[0][1], temp[0][2]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 0, 2, nodeExpand);
                q.push(node->child2);
            }
            copyBoard(temp, node->matrix);
            swap(temp[0][1], temp[1][1]);
            if (node->parent->matrix != temp) {
                assignChild3(node, temp, 1, 1, nodeExpand);
                q.push(node->child3);
            }
            break;
        case 2:
            copyBoard(temp, node->matrix);
            swap(temp[0][2], temp[0][1]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 0, 1, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[0][2], temp[1][2]);
```

```cpp
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 1, 2, nodeExpand);
                q.push(node->child2);
            }
            break;
        }
    case 1:
        switch (node->blankY) {
        case 0:
            copyBoard(temp, node->matrix);
            swap(temp[1][0], temp[0][0]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 0, 0, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[1][0], temp[1][1]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 1, 1, nodeExpand);
                q.push(node->child2);
            }
            copyBoard(temp, node->matrix);
            swap(temp[1][0], temp[2][0]);
            if (node->parent->matrix != temp) {
                assignChild3(node, temp, 2, 0, nodeExpand);
                q.push(node->child3);
            }
            break;
        case 1:
            copyBoard(temp, node->matrix);
            swap(temp[1][1], temp[1][0]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 1, 0, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[1][1], temp[0][1]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 0, 1, nodeExpand);
                q.push(node->child2);
            }
            copyBoard(temp, node->matrix);
            swap(temp[1][1], temp[1][2]);
            if (node->parent->matrix != temp) {
                assignChild3(node, temp, 1, 2, nodeExpand);
                q.push(node->child3);
            }
            copyBoard(temp, node->matrix);
            swap(temp[1][1], temp[2][1]);
```

```
            if (node->parent->matrix != temp) {
                assignChild4(node, temp, 2, 1, nodeExpand);
                q.push(node->child4);
            }
            break;
        case 2:
            copyBoard(temp, node->matrix);
            swap(temp[1][2], temp[1][1]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 1, 1, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[1][2], temp[0][2]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 0, 2, nodeExpand);
                q.push(node->child2);
            }
            copyBoard(temp, node->matrix);
            swap(temp[1][2], temp[2][2]);
            if (node->parent->matrix != temp) {
                assignChild3(node, temp, 2, 2, nodeExpand);
                q.push(node->child3);
            }
            break;
        }
    case 2:
        switch (node->blankY) {
        case 0:
            copyBoard(temp, node->matrix);
            swap(temp[2][0], temp[1][0]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 1, 0, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[2][0], temp[2][1]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 2, 1, nodeExpand);
                q.push(node->child2);
            }
            break;
        case 1:
            copyBoard(temp, node->matrix);
            swap(temp[2][1], temp[2][0]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 2, 0, nodeExpand);
                q.push(node->child1);
            }
```

```
            }
            copyBoard(temp, node->matrix);
            swap(temp[2][1], temp[1][1]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 1, 1, nodeExpand);
                q.push(node->child2);
            }
            copyBoard(temp, node->matrix);
            swap(temp[2][1], temp[2][2]);
            if (node->parent->matrix != temp) {
                assignChild3(node, temp, 2, 2, nodeExpand);
                q.push(node->child3);
            }
            break;
        case 2:
            copyBoard(temp, node->matrix);
            swap(temp[2][2], temp[2][1]);
            if (node->parent->matrix != temp) {
                assignChild1(node, temp, 2, 1, nodeExpand);
                q.push(node->child1);
            }
            copyBoard(temp, node->matrix);
            swap(temp[2][2], temp[1][2]);
            if (node->parent->matrix != temp) {
                assignChild2(node, temp, 1, 2, nodeExpand);
                q.push(node->child2);
            }
            break;
        }
    }
}

void UniCostSearch(int input[3][3], int goal[3][3], Node* root, int &depth, int &nodeExpand, int &queueSize) { //search algorithm for Uniform Cost Search
    priority_queue <Node*, vector<Node*>, compareUni> q;
    priority_queue <Node*, vector<Node*>, compareUni> q2;
    Node* temp = new Node;
    vector<Node*> duplicate;
    q.push(root);
    while (q.size() > 0) {
        if (q.empty() == true) {
            cout << "Failure";
            return;
        }
        q2 = q;
        while (q.size() > 0) {
            if (queueSize < q.size()) {
                queueSize = q.size();
            }
            temp = q.top();
```

```cpp
            q.pop();

            cout << "g(n) = " << temp->moveCost << " and h(n) = 0" << endl;
            printBoard(temp->matrix);

            if (checkGoal(temp->matrix, goal) == true) {
                cout << "Solution found!" << endl;
                cout << "Solution depth: " << temp->moveCost << endl;
                cout << "Number of nodes expanded: " << nodeExpand << endl;
                cout << "Max queue size: " << queueSize << endl;
                return;
            }
        }
        while (q2.size() > 0) {
            temp = q2.top();
            q2.pop();
            if (isDup(duplicate, temp) == false) {
                moveBlankUni(temp, q, nodeExpand, queueSize);
                duplicate.insert(duplicate.begin(), temp);
            }
        }
    }
}

void AMisTile(int input[3][3], int goal[3][3], Node* root, int &depth, int &nodeExpand, int &queueSize) { //search algorithm for A star Misplaced Tile Heuristic
    priority_queue <Node*, vector<Node*>, compareHTile> q;
    Node* temp = new Node;
    vector<Node*> duplicate;
    q.push(root);
    while (q.size() > 0) {
        if (q.empty() == true) {
            cout << "Failure";
            return;
        }
        if (queueSize < q.size()) {
            queueSize = q.size();
        }

        temp = q.top();
        q.pop();

        cout << "g(n) = " << temp->moveCost << " and h(n) = " << temp->hTile - temp->moveCost << endl;
        printBoard(temp->matrix);

        if (checkGoal(temp->matrix, goal) == true) {
            cout << "Solution found!" << endl;
            cout << "Solution depth: " << temp->moveCost << endl;
            cout << "Number of nodes expanded: " << nodeExpand << endl;
            cout << "Max queue size: " << queueSize << endl;
```

```cpp
            return;
        }
        else {
            moveBlankHTile(temp, q, nodeExpand, queueSize);
            duplicate.insert(duplicate.begin(), temp);
        }
    }
}

void AManhattan(int input[3][3], int goal[3][3], Node* root, int &depth, int &nodeExpand, int &queueSize) { //search algorithm for A star Manhattan Distance Heuristic
    priority_queue <Node*, vector<Node*>, compareHMan> q;
    Node* temp = new Node;
    vector<Node*> duplicate;
    q.push(root);
    while (q.size() > 0) {
        if (q.empty() == true) {
            cout << "Failure";
            return;
        }
        if (queueSize < q.size()) {
            queueSize = q.size();
        }
        temp = q.top();
        q.pop();

        cout << "g(n) = " << temp->moveCost << " and h(n) = " << temp->hMan - temp->moveCost << endl;
        printBoard(temp->matrix);

        if (checkGoal(temp->matrix, goal) == true) {
            cout << "Solution found!" << endl;
            cout << "Solution depth: " << temp->moveCost << endl;
            cout << "Number of nodes expanded: " << nodeExpand << endl;
            cout << "Max queue size: " << queueSize << endl;
            return;
        }
        else {
            moveBlankHMan(temp, q, nodeExpand, queueSize);
            duplicate.insert(duplicate.begin(), temp);
        }
    }
}

int main() //Choosing and user inputing current state into searching algorithm
{
    int initChoose;
    int blankX=1, blankY=1;
    int depth, nodeExpand, queueSize;
    int matrix[3][3] = { {1,2,3},{5,0,6},{4,7,8} };
    int goal[3][3] = { {1,2,3},{4,5,6},{7,8,0} };
```

```cpp
    cout << "Welcome to Huy Dinh Tran (862325308, htran197) 8-Puzzle Solver.\nType '1' to to use a default puzzle, or '2' to create your own.\n";
    cout << "Default puzzle: [ 1 2 3 ]" << endl;
    cout << "                [ 5 0 6 ]\n";
    cout << "                [ 4 7 8 ]\n";
    cin >> initChoose;

    if (initChoose == 2) {
        cout << "Enter your puzzle, using a zero to represent the blank. Please only enter valid 8-puzzles. \nEnter the puzzle demilimiting the numbers with a space. Press ENTER only when finished.\n";
        cout << "Enter the first row: ";
        for (int i = 0; i < 3; i++) {
            cin >> matrix[0][i];
            if (matrix[0][i] == 0) {
                blankX = 0;
                blankY = i;
            }
        }
        cout << "Enter the second row: ";
        for (int i = 0; i < 3; i++) {
            cin >> matrix[1][i];
            if (matrix[1][i] == 0) {
                blankX = 1;
                blankY = i;
            }
        }
        cout << "Enter the third row: ";
        for (int i = 0; i < 3; i++) {
            cin >> matrix[2][i];
            if (matrix[2][i] == 0) {
                blankX = 2;
                blankY = i;
            }
        }
    }

    Node* node = new Node;
    for (int i = 0; i < 3; i++){
        for (int j = 0; j < 3; j++) {
            node->matrix[i][j]=matrix[i][j];
        }
    }

    //initializing root node
    node->parent = NULL;
    node->hMan = 0;
    node->hTile = 0;
    node->moveCost = 0;
    node->blankX = blankX;
    node->blankY = blankY;
    depth = 0;
```

```cpp
    nodeExpand = 0;
    queueSize = 0;

    int algoChoose;
    cout << "Select algorithm: \n(1) for Uniform Cost Search \n(2) for the Misplaced Tile Heuristic \n(3) the Manhattan Distance Heuristic" << endl;
    cin >> algoChoose;

    auto start = chrono::steady_clock::now();

    switch (algoChoose) {
    case 1:
        UniCostSearch(matrix, goal, node, depth, nodeExpand, queueSize);
        break;
    case 2:
        AMisTile(matrix, goal, node, depth, nodeExpand, queueSize);
        break;
    case 3:
        AManhattan(matrix, goal, node, depth, nodeExpand, queueSize);
        break;
    }

    auto end = chrono::steady_clock::now();
    auto diff = end - start;

    cout << "Execution time (seconds): " << chrono::duration <double, milli>(diff).count() << " ms" << endl;

    return 0;
}
```