Assignment 1

CS170: Introduction to AI, Dr. Eamonn Keogh

Francisco Bustamante
SID 862131115
Email: fbust002@ucr.edu
Date: November 1, 2022

In Completing this assignment, I consulted:

- This is for help initializing a matrix to hold the puzzle information: **https://www.techiedelight.com/initialize-matrix-cpp/**
- I consulted this page to help construct a Binary Search tree: **https://www.geeksforgeeks.org/construct-bst-from-given-preorder-traversa/?ref=lbp**
- This page is where I visited for the Queue header file. **https://iq.opengenus.org/different-ways-to-initialize-queue-cpp/**
- I used this page to see if it was possible to initialize a queue with another queue: **https://stackoverflow.com/questions/11044304/can-i-push-an-array-of-int-to-a-c-vector**

All code is completely original. No part of the code is unoriginal.

Outline of Report:

CS170: Assignment 1: The Eight-Puzzle

Francisco Bustamante, 862131115, Nov 1,2022

Introduction

The eight-puzzle is like the traditional sliding tile puzzle game with the difference being that it only has 8 sliding tiles. The puzzle contains 8 tiles that can be moved in any direction if there is open spot available for the tile to move into. Figure 1 displays the eight-puzzle in a scrambled state. Here only tiles 7, 3, 1, and 2 can move into the empty tile.

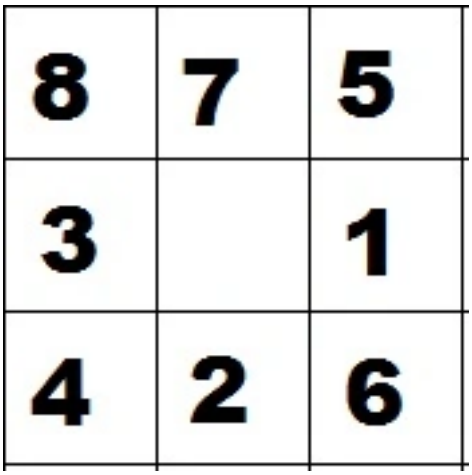

Figure 1: A picture of the eight-puzzle scrambled

The following report details my finding while constructing a program that can solve the eight-puzzle using Uniform search, A* search, and A* search with the Manhattan heuristic. This program is part of the first project assigned by Dr. Eamonn Keogh in his Introduction to AI course at UCR. My language of choice for the project is C++.

For this report, h(n) is the value of the heuristic, g(n) is the depth of a node, and f(n) is the sum of h(n) and g(n).

Algorithms Used

The three algorithms used are:

- Uniform Cost Search using the misplaced tile heuristic.
- A* search using the misplaced tile heuristic.
- A* search using the Manhattan Distance heuristic.

Uniform Cost Search

Uniform Cost search is essentially A* search but with h(n) being 0 at any level of the search tree. This search will only expand the cheapest node. For this project there is no weight added to any expansion. Every expansion is equal to one as to replicate the ease of moving a tile one space. So, this means the search essentially becomes breadth first search, expanding every child.

Missing Tile Heuristic A* Search

This search algorithm has a queue, and it pops the queue based off the node with the smallest f(n) value. F(n) is equal to g(n) + h(n). g(n) is the depth of the node and for this algorithm, h(n) is the number of tiles that are not in the correct position. For example, in figure 1, the h(n) value is 9 as all tiles are out of place. This search will expand the node with the cheapest cost instead of every child such as uniform search.

Manhattan Distance Heuristic A* Search

This Search algorithm uses A* as seen previously but changes the heuristic to the **Manhattan District heuristic**. This heuristic uses the distance of a tile to its original location as the h(n), instead of how many misplaces tiles there are. F(n) for this search algorithm is the sum of g(n) (depth of tree) and h(n) (Manhattan Distance).

Figure 2 This is an 8-puzzle with a Manhattan Distance of 4

Using figure 2, the Manhattan distance is 4 because tile 8 is away from its solution, tile 7 is also one space away, and tile 0 is two spaces away. If all these values are summed up, we get a value of h(n) = 4.

Comparing All Three Algorithms

Figure 3 displays eight different puzzles with different levels of depth for their solution. All these puzzles are used in figure 4 to create a graph that represents how efficient each graph was based on how many nodes were expanded versus the depth of the solution.

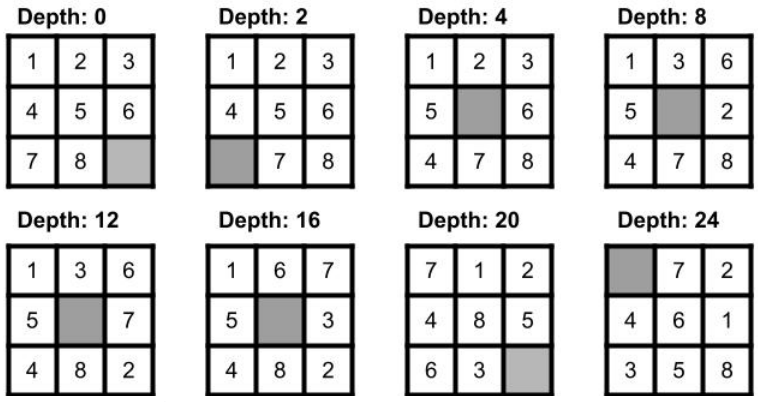

Figure 3 A set of 8 puzzles all with varying solution depths.

From figure 4, we can see that all three algorithms preform relatively the same for the first three puzzles, however, quickly after that the Uniform Search algorithm rapidly escalates the number of nodes it expands followed by A* search. Figure 4 also shows how well A* search does compared to the Uniform Search. This is due to the heuristic. A* search with the misplaced tile heuristic has the advantage of being able to see ahead and know which state could potentially be the best. A* search with the Manhattan Distance is a better heuristic as it expands less nodes and is much faster than A* with the Misplaced tile heuristic.

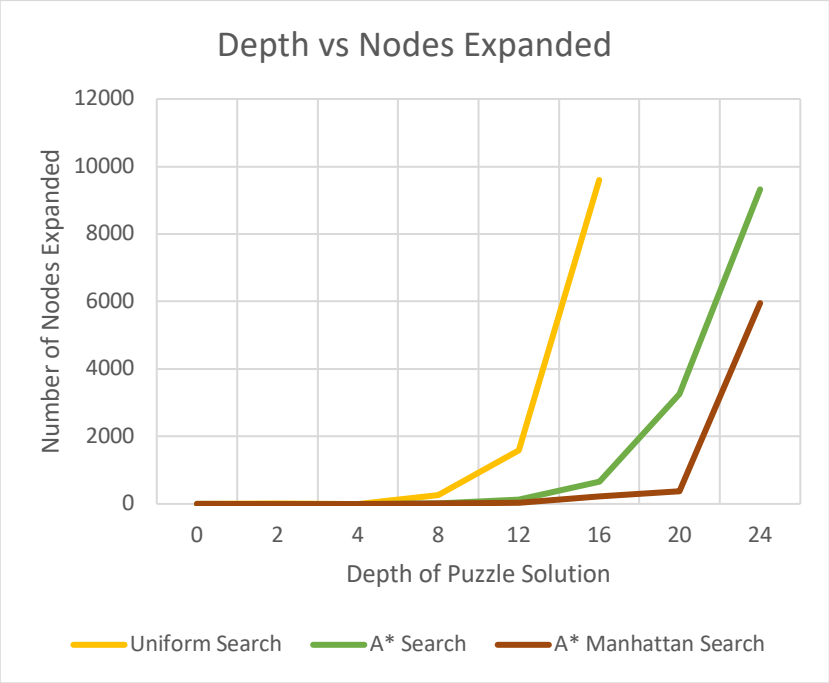

Figure 4 This graph represents the number of nodes expanded by all three algorithms versus the depth of a puzzle solution given in figure 3

In figure 4 we can also see that the Uniform Search's data ends abruptly. This is because after depth 16 Uniform Search takes am excessively long time to look for an answer. This is due to Uniform Search expanding every single child node. This grows exponentially and every concurrent depth level adds a very large number of nodes to search through.

Conclusion

Of all three search algorithms observed, A* with the Manhattan Distance Heuristic by far the best one in terms of time and memory. All three algorithms are optimal, and give the optimal answer for any puzzle, however, they differ in how long they take to retrieve an answer. Uniform Search is impractical in any real-life situation as it takes a large amount of time to retrieve an answer. A* search with a good heuristic is useful in almost any application.

Side Note

When I first started this project, I tried A* search with the missing tile heuristic and g(n) hardcoded to 0 and I got blazing fast solution to even the hardest puzzle, but it was incredibly suboptimal with solutions in depth 200's. I just thought this was interesting.

Assignment 1

The Following is a traceback of an **easy** puzzle:

```
Welcome to 8-Puzzle Solver
Type 1 to use defualt puzzle or 2 to input your own
2
Enter your Puzzle
Input 3 numbers for row 1 with a space in between then hit enter
1 2 3
Input 3 numbers for row 2 with a space in between then hit enter
0 5 6
Input 3 numbers for row 3 with a space in between then hit enter
4 7 8

{ 1 2 3 }
{ 0 5 6 }
{ 4 7 8 }

Chose an Algorithm. (1) for Unifrom Search, (2) for A* misplaced tile Search, (3) for A* Manhattan Search
3
The best state to expand with g(n) = 1 and h(n) = 3 is...

{ 1 2 3 }
{ 4 5 6 }
{ 0 7 8 }

The best state to expand with g(n) = 2 and h(n) = 2 is...

{ 1 2 3 }
{ 4 5 6 }
{ 7 0 8 }

The best state to expand with 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: 3
Max Queue size: 4
Program ended with exit code: 0
```

Assignment 1

The Following is a traceback of **difficult (depth 20)** puzzle:

```
Welcome to 8-Puzzle Solver
Type 1 to use defualt puzzle or 2 to input your own
2
Enter your Puzzle
Input 3 numbers for row 1 with a space in between then hit enter
7 1 2
Input 3 numbers for row 2 with a space in between then hit enter
4 8 5
Input 3 numbers for row 3 with a space in between then hit enter
6 3 0

{ 7 1 2 }
{ 4 8 5 }
{ 6 3 0 }

Chose an Algorithm. (1) for Unifrom Search, (2) for A* misplaced tile Search, (3) for A* Manhattan Search
3
The best state to expand with g(n) = 1 and h(n) = 8 is...

{ 7 1 2 }
{ 4 8 5 }
{ 6 0 3 }

The best state to expand with g(n) = 2 and h(n) = 12 is...

{ 7 1 2 }
{ 4 0 5 }
{ 6 8 3 }
```

...

```
The best state to expand with g(n) = 17 and h(n) = 6 is...

{ 1 2 3 }
{ 0 4 5 }
{ 7 8 6 }

The best state to expand with g(n) = 18 and h(n) = 4 is...

{ 1 2 3 }
{ 4 0 5 }
{ 7 8 6 }

The best state to expand with g(n) = 19 and h(n) = 2 is...

{ 1 2 3 }
{ 4 5 0 }
{ 7 8 6 }

The best state to expand with g(n) = 20 and h(n) = 0 is...

{ 1 2 3 }
{ 4 5 6 }
{ 7 8 0 }

Goal State!

Solution depth was: 20
Number of Nodes Expanded: 376
Max Queue size: 241
Program ended with exit code: 0
```

Assignment 1

##URL to GitHub to run Code:
https://github.com/frab6562/CS170-Eight-Puzzle-Project.git

<<main.cpp>>

```
#include <iostream>
#include <vector>
#include "puzzle.hpp"
#include "algorithm.hpp"

using namespace std;

int main() {
    bool defaul = false;
    bool custom  = false;
    int choice;
    //n and m are used to keep track of where the '0' is in the puzzle. Either n(row) 0-2 or m(Column) 0-2.
    int n = 1;
    int m = 1;
    //This stores the value of the Default puzzle
    int Default_Puzzle[9] = {1,6,7,5,0,3,4,8,2};
    int puzzle[3][3];
    node* root;

    //Intro asking if default or cusstom puzzle.
    cout << "Welcome to 8-Puzzle Solver\nType 1 to use defualt puzzle or 2 to input your own" << endl;
    cin >> choice;
    if(choice == 1)
        defaul = true;
    if(choice == 2)
        custom = true;


    //Deciding what to do if either 1 or 2 is chosen.
    //Default Puzzle
    if(defaul){
        int x = 0;
        cout << "Default Puzzle" << endl;
        for (int i = 0; i < 3; ++i){
            for (int j = 0; j < 3; ++j){
                puzzle[i][j] = Default_Puzzle[x];
                ++x;
```

```
            }
        }
        root = newNode(puzzle,n,m);
        Display_Puzzle(root->GameState);
    }
    //Costum Puzzle
    else{
        bool hasZero = false;
        int x;
        cout << "Enter your Puzzle" << endl;
        for(int i = 0; i < 3; ++i){
            cout << "Input 3 numbers for row " << i+1 << " with a space in between then
hit enter" << endl;
            for(int j = 0; j < 3; ++j){
                cin >> x;
                puzzle[i][j] = x;
                if(x == 0){
                    hasZero = true;
                    n = i;
                    m = j;
                }
            }
        }
        root = newNode(puzzle,n,m);
        Display_Puzzle(root->GameState);
        //Checks if puzzle has a 0, if not it terminates program.
        if(!hasZero){
            cout << "!!ERROR: Puzzle does not contain a 0!!" << endl;
            return 0;
        }
    }

    //Ask User which algoithm they want to run...
    int decision = 0;
    cout << "Chose an Algorithm. (1) for Unifrom Search, (2) for A* misplaced tile
Search, (3) for A* Manhattan Search" << endl;
    cin >> decision;
    if(decision == 1){
        Uniform_Search(root);
    }
    else if(decision == 2){
        A_Star_Search(root);
    }
    else{
        A_Star_Search_Manhattan(root);
    }
```

```
    return 0;
}
```

<<Puzzle.hpp>>

```
#ifndef puzzle_h
#define puzzle_h
#include <iostream>

using namespace std;

int Solved_Puzzle[3][3] = {1,2,3,4,5,6,7,8,0};

template <int rows, int cols>
//This function is used to diplay the puzzle as a 3x3 matrix.
void Display_Puzzle(int (&puzzle)[rows][cols]){
    cout << endl;
    for(int i = 0; i < 3; ++i){
        cout << "{ ";
        for(int j = 0; j < 3; ++j){
            cout << puzzle[i][j];
            cout << " ";
        }
        cout << "}";
        cout << endl;
    }
    cout << endl;
}

//This function is used to make moves within the puzzle using WASD.
template <int rows, int cols>
void Move_Puzzle(int (&puzzle)[rows][cols], char move, int &n, int &m){
    if(move == 'w'){ //up
        if(n != 0){
            puzzle[n][m] = puzzle[n-1][m];
            puzzle[n-1][m] = 0;
            n = n-1;
        }
    }
    else if(move == 'a'){ //left
        if(m != 0){
            puzzle[n][m] = puzzle[n][m-1];
            puzzle[n][m-1] = 0;
            m = m-1;
        }
```

```
    }
    else if(move == 's'){ //down
        if(n != 2){
            puzzle[n][m] = puzzle[n+1][m];
            puzzle[n+1][m] = 0;
            n = n+1;
        }
    }
    else if(move == 'd'){ //right
        if(m != 2){
            puzzle[n][m] = puzzle[n][m+1];
            puzzle[n][m+1] = 0;
            m = m+1;
        }
    }
}

//Used to assign puzzles to each other
template <int rows, int cols>
void AssignPuzzle(int (&puzzle)[rows][cols], int (&puzzle2)[rows][cols]){
    for (int i = 0; i < 3; ++i){
        for (int j = 0; j < 3; ++j){
            puzzle[i][j] = puzzle2[i][j];
        }
    }
}

//Checks to see if the puzzle is solved
template <int rows, int cols>
bool SolvedState(int (&puzzle)[rows][cols]){
    bool solved = false;
    for (int i = 0; i < 3; ++i){
        for (int j = 0; j < 3; ++j){
            if(puzzle[i][j] != Solved_Puzzle[i][j])
                return solved = false;
        }
    }
    return solved = true;
}

//Retruns H(n) value with the Missing tile heuristic
template <int rows, int cols>
int GetHn(int (&puzzle)[rows][cols]){
    int hn = 0;
    for (int i = 0; i < 3; ++i){
        for (int j = 0; j < 3; ++j){
```

```
            if(puzzle[i][j] != Solved_Puzzle[i][j])
                ++hn;
        }
    }
    return hn;
}

//Retruns H(n) value with the Manhattan District heuristic
template <int rows, int cols>
int Get_M_Distance(int (&puzzle)[rows][cols]){
    int M_distance = 0;
    int temp = 0;

    for (int i = 0; i < 3; ++i){
        for (int j = 0; j < 3; ++j){
            if(puzzle[i][j] != Solved_Puzzle[i][j]){
                if(i == 0 && j == 0){ //Checking tile 1
                    for (int x = 0; x < 3; ++x){
                        for (int y = 0; y < 3; ++y){
                            if(puzzle[x][y] == 1){
                                temp = abs(x-i) + abs(y-j);
                                M_distance = M_distance + temp;
                            }
                        }
                    }
                }
                else if(i == 0 && j == 1){ //Checking tile 2
                    for (int x = 0; x < 3; ++x){
                        for (int y = 0; y < 3; ++y){
                            if(puzzle[x][y] == 2){
                                temp = abs(x-i) + abs(y-j);
                                M_distance = M_distance + temp;
                            }
                        }
                    }
                }
                else if(i == 0 && j == 2){ //Checking tile 3
                    for (int x = 0; x < 3; ++x){
                        for (int y = 0; y < 3; ++y){
                            if(puzzle[x][y] == 3){
                                temp = abs(x-i) + abs(y-j);
                                M_distance = M_distance + temp;
                            }
                        }
                    }
                }
```

```
else if(i == 1 && j == 0){ //Checking tile 4
    for (int x = 0; x < 3; ++x){
        for (int y = 0; y < 3; ++y){
            if(puzzle[x][y] == 4){
                temp = abs(x-i) + abs(y-j);
                M_distance = M_distance + temp;
            }
        }
    }
}
else if(i == 1 && j == 1){ //Checking tile 5
    for (int x = 0; x < 3; ++x){
        for (int y = 0; y < 3; ++y){
            if(puzzle[x][y] == 5){
                temp = abs(x-i) + abs(y-j);
                M_distance = M_distance + temp;
            }
        }
    }
}
else if(i == 1 && j == 2){ //Checking tile 6
    for (int x = 0; x < 3; ++x){
        for (int y = 0; y < 3; ++y){
            if(puzzle[x][y] == 6){
                temp = abs(x-i) + abs(y-j);
                M_distance = M_distance + temp;
            }
        }
    }
}
else if(i == 2 && j == 0){ //Checking tile 7
    for (int x = 0; x < 3; ++x){
        for (int y = 0; y < 3; ++y){
            if(puzzle[x][y] == 7){
                temp = abs(x-i) + abs(y-j);
                M_distance = M_distance + temp;
            }
        }
    }
}
else if(i == 2 && j == 1){ //Checking tile 8
    for (int x = 0; x < 3; ++x){
        for (int y = 0; y < 3; ++y){
            if(puzzle[x][y] == 8){
                temp = abs(x-i) + abs(y-j);
                M_distance = M_distance + temp;
```

```
                    }
                }
            }
        }
        else if(i == 2 && j == 2){ //Checking tile 9
            for (int x = 0; x < 3; ++x){
                for (int y = 0; y < 3; ++y){
                    if(puzzle[x][y] == 0){
                        temp = abs(x-i) + abs(y-j);
                        M_distance = M_distance + temp;
                    }
                }
            }
        }
    }
}
return M_distance;
}


#endif /* Puzzle_h */
```

<<algorithm.hpp>>

```
#ifndef algorithm_h
#define algorithm_h
#include <iostream>
#include <vector>
#include "puzzle.hpp"

using namespace std;

class node{
public:
    int GameState[3][3];
    int n; //Row of 0
    int m; //Column of 0
    int hn = 10; //hn is h(n)
    char prev = 'e'; //Previous move of puzzle to prevent loops.
    int depth = 0; //Depth of node, is also g(n)
    int fn = 10;   //g(n) + h(n)
    int fn_MD = 10; //Manhattan District f(n)
    int hn_MD = 10; //Manhattan District h(n)
```

```
    node* left; //node of parent that is left move.
    node* up;  //node of parent that is up move.
    node* down; //node of parent that is down move.
    node* right; //node of parent that is right move.
};
vector<node*> Q;
vector<vector<int>> Puzzles;

//Creates a new node that is null pointers.
node* newNode(int data[3][3], int n, int m){
    node* root = new node();
    for (int i = 0; i < 3; ++i){
        for (int j = 0; j < 3; ++j){
            root->GameState[i][j] = data[i][j];
        }
    }
    root->left = nullptr;
    root->right = nullptr;
    root->up = nullptr;
    root->down = nullptr;
    root->n = n;
    root-> m = m;
    return root;
}

//Checks a node to see if it has a duplicate puzzle.
bool CheckDuplicate(node* parent){
    if(Puzzles.size() == 0)
        return false;

    vector<int> temp(9);
    int y = 0;

    for(int j = 0; j < 3; ++j){
        for(int k = 0; k < 3; ++k){
            temp.at(y) = parent->GameState[j][k];
            ++y;
        }
    }

    for(int x = 0; x < Puzzles.size(); ++x){
        int counter = 0;
        for (int i = 0; i < 9; ++i){
            if(Puzzles.at(x).at(i) == temp.at(i))
                ++counter;
```

```
            if(counter == 9)
                return true;
        }
    }
    return false;
}

//Adds a node's Puzzle to a list of puzzle to check them later.
void AddPuzzle(node* parent){
    vector<int> temp(9);
    int i = 0;
        for(int j = 0; j < 3; ++j){
            for(int k = 0; k < 3; ++k){
                temp.at(i) = parent->GameState[j][k];
                ++i;
            }
        }
    Puzzles.push_back(temp);
}

//Stores max size of Queue
int Q_Msize = 0;
void getQ_Max_Size(){
    double Q_Size = Q.size();
    if(Q_Size > Q_Msize)
        Q_Msize = Q_Size;
}

//This Function expands a node to all possible moves a puzzle can go to. It also stores
all the relevavnt information of a node and also pushes a node to the back of the
queue.
int nodes_Expanded = 0;
void Expand_Node(node* &parent){
    node* temp = newNode(parent->GameState, parent->n, parent->m);
    ++nodes_Expanded;

    if(temp->m != 0 && parent->prev != 'R'){
        AssignPuzzle(temp->GameState, parent->GameState); //temp = puzzle
        Move_Puzzle(temp->GameState, 'a',temp->n,temp->m);
        if(!CheckDuplicate(temp)){
            node* left = newNode(temp->GameState,temp->n,temp->m);
            parent->left = left;
            temp->n = parent->n;
            temp->m = parent->m;
            left->hn = GetHn(left->GameState);
            left->hn_MD = Get_M_Distance(left->GameState);
```

```
            left->prev = 'L';
            left->depth = parent->depth + 1;
            Q.push_back(parent->left);
            left->fn = left->hn + left->depth;
            left->fn_MD = left->hn_MD + left->depth;
            AddPuzzle(left);
        }
        else{
            temp->n = parent->n;
            temp->m = parent->m;
        }
    }


    if(temp->n != 0 && parent->prev != 'D'){
        AssignPuzzle(temp->GameState, parent->GameState); //temp = puzzle
        Move_Puzzle(temp->GameState, 'w', temp->n,temp->m);
        if(!CheckDuplicate(temp)){
            node* up = newNode(temp->GameState,temp->n,temp->m);
            parent->up = up;
            temp->n = parent->n;
            temp->m = parent->m;
            up->hn = GetHn(up->GameState);
            up->hn_MD = Get_M_Distance(up->GameState);
            up->prev = 'U';
            Q.push_back(parent->up);
            up->depth = parent->depth + 1;
            up->fn = up->hn + up->depth;
            up->fn_MD = up->hn_MD + up->depth;
            AddPuzzle(up);
        }
        else{
            temp->n = parent->n;
            temp->m = parent->m;
        }

    }

    if(temp->n != 2 && parent->prev != 'U'){
        AssignPuzzle(temp->GameState, parent->GameState); //temp = puzzle
        Move_Puzzle(temp->GameState, 's', temp->n,temp->m);
        if(!CheckDuplicate(temp)){
            node* down = newNode(temp->GameState,temp->n,temp->m);
            parent->down = down;
            temp->n = parent->n;
            temp->m = parent->m;
```

```
            down->hn = GetHn(down->GameState);
            down->hn_MD = Get_M_Distance(down->GameState);
            down->prev = 'D';
            down->depth = parent->depth + 1;
            Q.push_back(parent->down);
            down->fn = down->hn + down->depth;
            down->fn_MD = down->hn_MD + down->depth;
            AddPuzzle(down);
        }
        else{
            temp->n = parent->n;
            temp->m = parent->m;
        }
    }

    if(temp->m != 2 && parent->prev != 'L'){
        AssignPuzzle(temp->GameState, parent->GameState); //temp = puzzle
        Move_Puzzle(temp->GameState, 'd',temp->n,temp->m);
        node* right = newNode(temp->GameState,temp->n,temp->m);
        if(!CheckDuplicate(temp)){
            parent->right = right;
            right->hn = GetHn(right->GameState);
            right->hn_MD = Get_M_Distance(right->GameState);
            right->prev = 'R';
            right->depth = parent->depth + 1;
            Q.push_back(parent->right);
            right->fn = right->hn + right->depth;
            right->fn_MD = right->hn_MD + right->depth;
            AddPuzzle(right);
        }
        else{
            temp->n = parent->n;
            temp->m = parent->m;
        }
    }
    getQ_Max_Size();
}


//Sorts the Queue in order from greatest to least depth value.
void sortQueue_gn(){
    node* tempi;
    node* tempj;
    for(int i = 0; i < Q.size(); ++i){
        for(int j = i; j < Q.size(); ++j){
            if(Q.at(i)->depth < Q.at(j)->depth){
```

```
                tempi = Q.at(i);
                tempj = Q.at(j);
                Q.at(i) = tempj;
                Q.at(j) = tempi;
            }
        }
    }
}

//Sorts the Queue in order from greatest to least hn value.
void sortQueue_Manhattan(){
    node* tempi;
    node* tempj;
    for(int i = 0; i < Q.size(); ++i){
        for(int j = i; j < Q.size(); ++j){
            if(Q.at(i)->fn_MD < Q.at(j)->fn_MD){
                tempi = Q.at(i);
                tempj = Q.at(j);
                Q.at(i) = tempj;
                Q.at(j) = tempi;
            }
        }
    }
}

//Sort the queue in order from greatest to smalled f(n) values of each node.
void sortQueue_fn(){
    node* tempi;
    node* tempj;
    for(int i = 0; i < Q.size(); ++i){
        for(int j = i; j < Q.size(); ++j){
            if(Q.at(i)->fn < Q.at(j)->fn){
                tempi = Q.at(i);
                tempj = Q.at(j);
                Q.at(i) = tempj;
                Q.at(j) = tempi;
            }
        }
    }

}

//This is the unifrom search algoithm used to solve a puzzle. This search does not take
into account g(n). I didn't explicilty write it but this function does not even look at g(n)
because it is hardcoded to 0.
void Uniform_Search(node* parent){
```

```
    bool solved = false;
    Expand_Node(parent);
    node* Front = Q.at(Q.size()-1);
    cout << "The best state to expand with h(n) = 1 is..." << endl;
    Display_Puzzle(Front->GameState);
    solved = SolvedState(Front-> GameState);
    while(!solved){
        if(Q.empty()){
            cout << "Failure" << endl;
            return;
        }
        Front = Q.at(Q.size()-1);
        Q.pop_back();
        Expand_Node(Front);
        sortQueue_gn();
        Front = Q.at(Q.size()-1);
        cout << "The best state to expand with h(n) = 1 is..." << endl;
        Display_Puzzle(Front->GameState);
        solved = SolvedState(Front-> GameState);
    }
    cout << "Goal State!" << endl;
    cout << "\nSolution depth was: " << Front->depth << endl;
    cout << "Number of Nodes Expanded: " << nodes_Expanded << endl;
    cout << "Max Queue size: " << Q_Msize << endl;
}

//This search does look at h(n) and g(n) together
void A_Star_Search(node* parent){
    bool solved = false;
    Expand_Node(parent);
    sortQueue_fn();
    node* Front = Q.at(Q.size()-1);
    cout << "The best state to expand with g(n) = " << Front->depth << " and h(n) = "
<< Front->hn << " is..." << endl;
    Display_Puzzle(Front->GameState);
    solved = SolvedState(Front-> GameState);
    while(!solved){
        if(Q.empty()){
            cout << "Failure" << endl;
            return;
        }
        Front = Q.at(Q.size()-1);
        Q.pop_back();
        Expand_Node(Front);
        sortQueue_fn();
        Front = Q.at(Q.size()-1);
```

```
        cout << "The best state to expand with g(n) = " << Front->depth << " and h(n) =
" << Front->hn << " is..." << endl;
        Display_Puzzle(Front->GameState);
        solved = SolvedState(Front-> GameState);
    }
    cout << "Goal State!" << endl;
    cout << "\nSolution depth was: " << Front->depth << endl;
    cout << "Number of Nodes Expanded: " << nodes_Expanded << endl;
    cout << "Max Queue size: " << Q_Msize << endl;
}


//This search uses the Manhattan District heuristic as h(n) and the depth of the node
as g(n).
void A_Star_Search_Manhattan(node* parent){
    bool solved = false;
    Expand_Node(parent);
    sortQueue_Manhattan();
    node* Front = Q.at(Q.size()-1);
    cout << "The best state to expand with g(n) = " << Front->depth << " and h(n) = "
<< Front->hn << " is..." << endl;
    Display_Puzzle(Front->GameState);
    solved = SolvedState(Front-> GameState);
    while(!solved){
        if(Q.empty()){
            cout << "Failure" << endl;
            return;
        }
        Front = Q.at(Q.size()-1);
        Q.pop_back();
        Expand_Node(Front);
        sortQueue_Manhattan();
        Front = Q.at(Q.size()-1);
        cout << "The best state to expand with g(n) = " << Front->depth << " and h(n) =
" << Front->hn_MD << " is..." << endl;
        Display_Puzzle(Front->GameState);
        solved = SolvedState(Front-> GameState);
    }
    cout << "Goal State!" << endl;
    cout << "\nSolution depth was: " << Front->depth << endl;
    cout << "Number of Nodes Expanded: " << nodes_Expanded << endl;
    cout << "Max Queue size: " << Q_Msize << endl;

}

#endif /* algorithm_h */
```