Assignment 1 CS170: Introduction to Artificial Intelligence
Luis Sanchez Dr. Eamonn Keogh
SID:862046663
Lsanc044@ucr.edu
2-20-2021

In completing this assignment I consulted:

- The Blind Search, Heuristic Search and Eight-Puzzle briefing lecture slides and notes annotated from lecture.
- Microsoft C++

All important code is original. Only c++ standard library includes were copied

- iostream
- Vector
- String
- Cmath
- Queue
- List

Outline of this report:

CS170: Assignment 1: The eight-puzzle

Luis Sanchez, SID 862046663 2/20/2021

Introduction:

The eight puzzle is a smaller form of the larger classic 15 puzzle. *"The **15 puzzle** (also called **Gem Puzzle**, **Boss Puzzle**, **Game of Fifteen**, **Mystic Square** and many others) is a sliding puzzle that consists of a frame of numbered square tiles in random order with one tile missing."[1]* The goal of the puzzle is to slide around the tiles until you reach a numerically ordered list with the blank in the bottom right corner. In figure 1 you can see on the left is the scrambled puzzle and on the right is the finished completed puzzle.

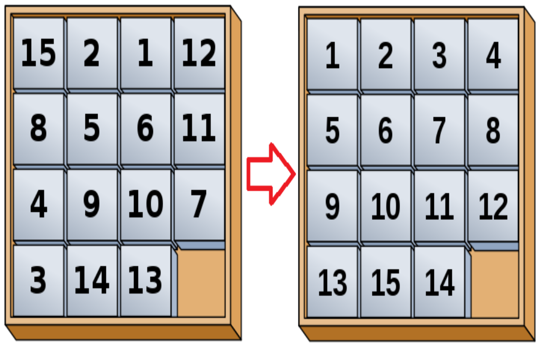


Figure 1: scrambled and completed 15 puzzles.
Souce: 15 puzzle[1]

In my Introduction to Artificial Intelligence course, Dr.Keogh explained the process of blind search and heuristic search. Dr.Keogh tasked me with creating an 8 puzzle solver that uses blind and heuristic searches to solve the puzzle. I focused on a uniform cost search, misplaced tiles search, and manhattan search to complete the project.

Types of Searches Used:

The first search I coded was a uniform cost search. The uniform search uses the cost to reach the next node as the basis to decide on which node to expand. This method as you will find out later in the report is a poor choice for solving 8 puzzles.

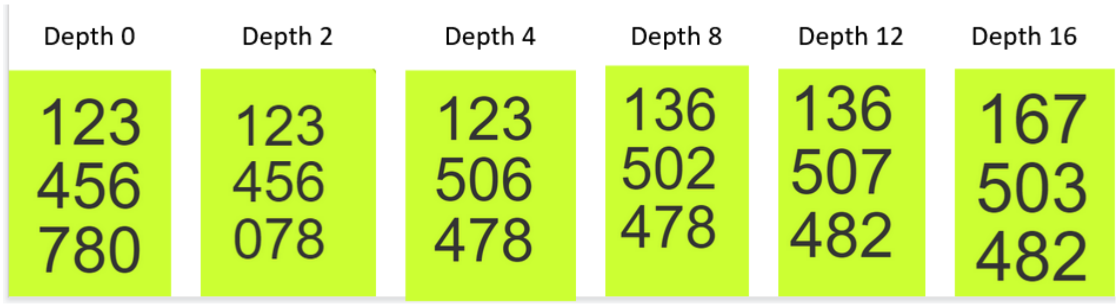
The second search is a A* variant that focuses on the next node cost and the number of misplaced tiles. A misplaced tile is a tile that is not in the goal state location. IE tile 2 being in tile 5's spot. This approach did perform better than uniform but was not the best choice.

The last search that I coded and tested was another A* variant that focused on the cost to reach the next node and the number of moves it takes to slide a tile into the correct spot. This variant was by far the best choice out of the three to solve 8 puzzles.

[1]15 puzzle. Wikipedia. 2/20/2021, from https://en.wikipedia.org/wiki/15_puzzle

Methoalogy and Testing:

To test each search I used a set of 6 different tile scrambles with varying depths. The depths and scrambles can be seen in figure 2.



Depth 0	Depth 2	Depth 4	Depth 8	Depth 12	Depth 16
123	123	123	136	136	167
456	456	506	502	507	503
780	078	478	478	482	482

Figure 2: Test cases for each search. Source: Slides by Dr. Eamonn Keogh [2]

I used a 10-minute cut-off point for each search if the search was unable to find the solution. Only the manhattan search was able to complete all 6 puzzles within 10 minutes. Neither the A* misplaced tile search nor the uniform cost search was able to complete the depth 16 puzzle in 10 minutes. The uniform cost was also unsuccessful in completing the depth 12 puzzle within the 10 minutes.

I focused on comparing the number of nodes expanded and the max size of the queue. I focused on the number of nodes expanded to be able to estimate the runtime complexity of each search. I also focused on the max size of the queue to be able to estimate the space complexity of each algorithm.

Test Results:

As stated earlier in the report the uniform cost search was the worst-performing, misplaced tiles search being in the middle and manhattan being the best search for the puzzle.

As we can see in figure 3, the uniform search expanded the most nodes out of the three. The misplaced tiles and Manthan were close for the first few depths but as the depth increased the difference grew larger and larger.

The depth 16 test for the misplaced tile algorithm is a 10minute snapshot due to the algorithm failing to solve the puzzle in under 10 minutes. The depth 12 test for the uniform cost search is also a 10-minute snapshot.

In figure 4, the max size of the queue vs depth we see a similar trend for each algorithm.

[2] Dr. Eamonn Keogh.Eight-Puzzle_briefing_and_review_of_search. ilearn, uploaded by Eamonn, 11 January. 2021, ilearn.ucr.edu

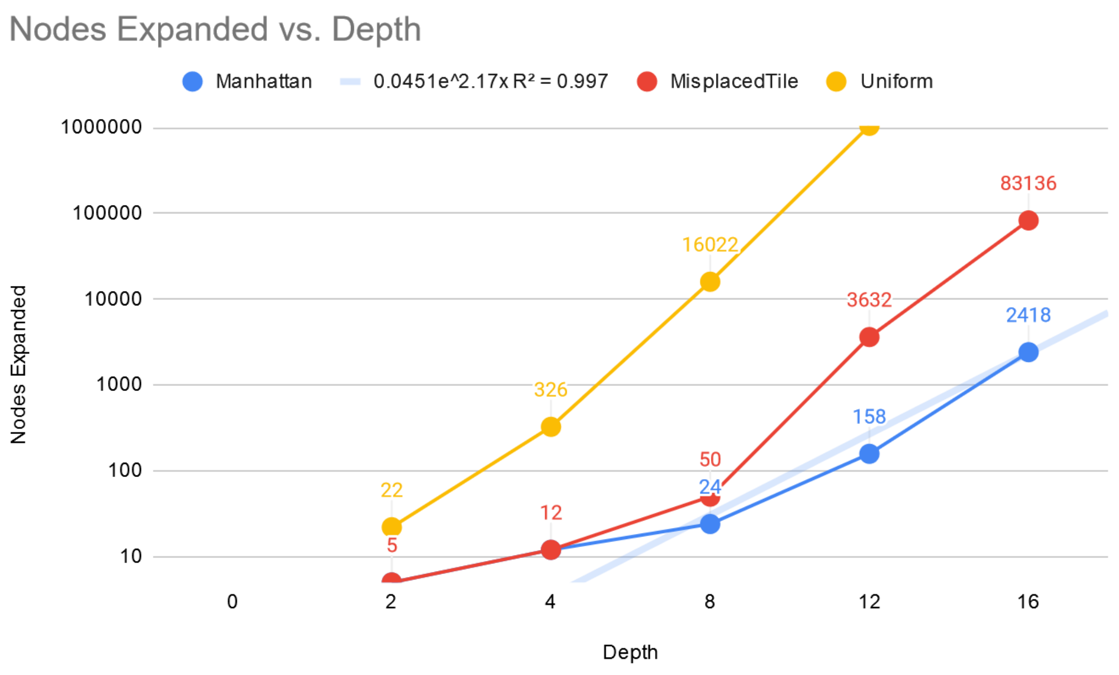


Figure 3: Nodes expanded vs. Depth for each search algorithm used. Data points for uniform depth 12 and misplaced tiles depth 16 are 10 minute snapshots.

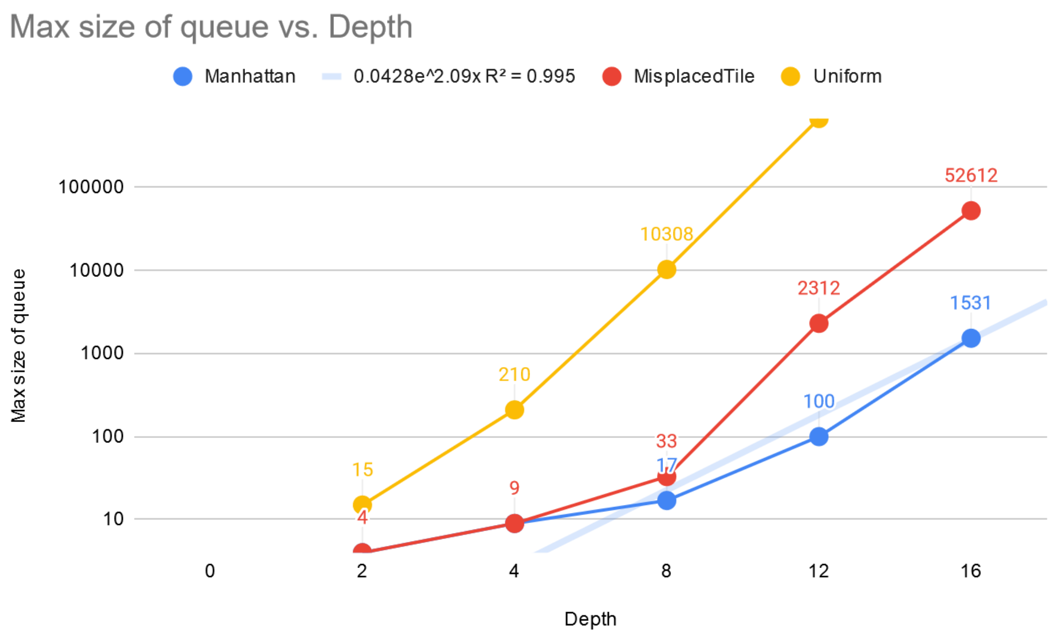


Figure 4: Max size of queue vs. Depth for each search algorithm used. Data points for uniform depth 12 and misplaced tiles depth 16 are 10 minute snapshots.

In figures 5 and 6 I attached an exponential trendline to see what type of growth the data set had. The data had an $R^2$of 1 for both misplaced and uniform. The manhattan had a R^2 of 0.997 for the number of nodes and 0.996 for the max size of the queue. Both these values are within reason to state that the number of nodes and max size of the queue grow exponentially with the depth solution of the puzzle.

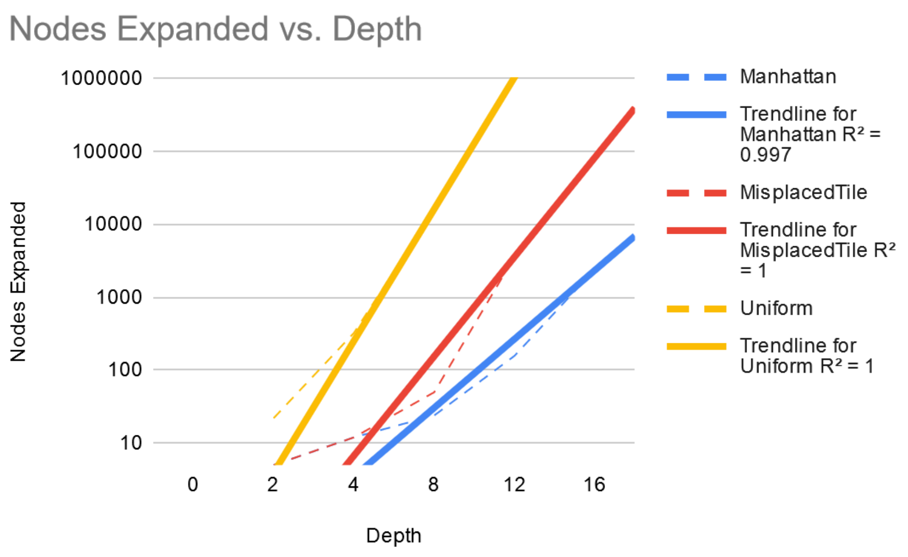


Figure 5: Nodes vs depth with exponential trend line in solid colors and the number of nodes in dashed lines

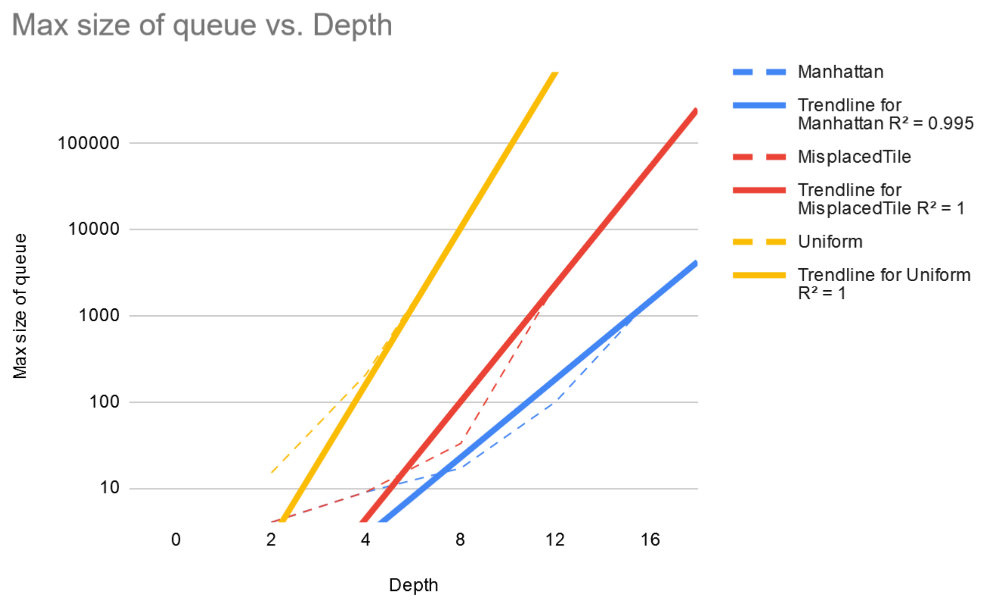


Figure 6: Max size of queue vs depth with exponential trend line in solid colors and the number of max size of the queue in dashed lines

Findings and conclusion:

From my findings, I conclude that the manhattan search algorithm is the best choice for solving 8 puzzles. The manhattan search used the least amount of memory space and evaluated the least number of nodes out of the three.

The misplaced tile approach did follow closely for shallow depths to the manhattan search but still used more memory and expanded more nodes at every level.

The uniform search was the absolute worst search to use for the 8 puzzles. The uniform search used the most space and expanded the most nodes.

From the trend lines from the number of nodes and the max size of the queue I can estimate the space complexity and runtime complexity to be around$O(2^n)$for both space and time.

Traceback of an easy puzzle:

```
Welcome to Luis Sanchez's 8-puzzle solver.
Type 1 to use a default puzzle, or 2 to enter your own puzzle
2
Enter your puzzle, use a 0 to represent the blank
Enter the first row, use space or tabs between numbers 123
Enter the second row, use space or tabs between numbers 406
Enter the third row, use space or tabs between numbers 758
Enter your choice of algorithm
1. Uniform Cost Search
2. A* with the Misplaced Tile heuristic.
3. A* with the Manhattan distance heuristic.
3
Expanding this node:
|1 |2 |3 |
|4 |b |6 |
|7 |5 |8 |
The best state to expand with a g(n) = 1 and h(n) = 1 is...
|1 |2 |3 |
|4 |5 |6 |
|7 |b |8 |
The best state to expand with a g(n) = 2 and h(n) = 0 is...
|1 |2 |3 |
|4 |5 |6 |
|7 |8 |b |


Goal!!

To solve this problem the search algorithm expanded a total of 7 nodes.
The maximum number of nodes in the queue at any one time was 6.
The depth of the goal node was 2
```

Traceback of an hard puzzle:

```
Type 1 to use a default puzzle, or 2 to enter your own puzzle
2
Enter your puzzle, use a 0 to represent the blank
Enter the first row, use space or tabs between numbers 136
Enter the second row, use space or tabs between numbers 507
Enter the third row, use space or tabs between numbers 482
Enter your choice of algorithm
1. Uniform Cost Search
2. A* with the Misplaced Tile heuristic.
3. A* with the Manhattan distance heuristic.
3
Expanding this node:
|1 |3 |6 |
|5 |b |7 |
|4 |8 |2 |
The best state to expand with a g(n) = 1 and h(n) = 9 is...
|1 |3 |6 |
|5 |7 |b |
|4 |8 |2 |
The best state to expand with a g(n) = 2 and h(n) = 8 is...
|1 |3 |6 |
|5 |7 |2 |
|4 |8 |b |
The best state to expand with a g(n) = 2 and h(n) = 8 is...
|1 |3 |b |
|5 |7 |6 |
|4 |8 |2 |
The best state to expand with a g(n) = 3 and h(n) = 7 is...
|1 |b |3 |
|5 |7 |6 |
|4 |8 |2 |
The best state to expand with a g(n) = 1 and h(n) = 9 is...
|1 |3 |6 |
|b |5 |7 |
|4 |8 |2 |
The best state to expand with a g(n) = 2 and h(n) = 8 is...
|1 |3 |6 |
|4 |5 |7 |
|b |8 |2 |
The best state to expand with a g(n) = 3 and h(n) = 9 is...
|1 |3 |6 |
|4 |5 |7 |
|8 |b |2 |
The best state to expand with a g(n) = 4 and h(n) = 8 is...
|1 |3 |6 |
|4 |5 |7 |
|8 |2 |b |
The best state to expand with a g(n) = 5 and h(n) = 7 is...
|1 |3 |6 |
|4 |5 |b |
|8 |2 |7 |
The best state to expand with a g(n) = 6 and h(n) = 6 is...
|1 |3 |b |
|4 |5 |6 |
|8 |2 |7 |
The best state to expand with a g(n) = 7 and h(n) = 5 is...
|1 |b |3 |
|4 |5 |6 |
|8 |2 |7 |
The best state to expand with a g(n) = 4 and h(n) = 8 is...
|1 |3 |6 |
|4 |5 |7 |
|b |8 |2 |
The best state to expand with a g(n) = 3 and h(n) = 9 is...
|1 |3 |6 |
|b |5 |7 |
|4 |8 |2 |
The best state to expand with a g(n) = 4 and h(n) = 8 is...
|1 |3 |6 |
|4 |5 |7 |
|b |8 |2 |
The best state to expand with a g(n) = 2 and h(n) = 10 is...
|b |3 |6 |
|1 |5 |7 |
|4 |8 |2 |
The best state to expand with a g(n) = 3 and h(n) = 9 is...
|1 |3 |6 |
|b |5 |7 |
|4 |8 |2 |

*****Omitted nodes******

The best state to expand with a g(n) = 5 and h(n) = 7 is...
|1 |3 |6 |
|5 |2 |b |
|4 |7 |8 |
The best state to expand with a g(n) = 6 and h(n) = 6 is...
|1 |3 |b |
|5 |2 |6 |
|4 |7 |8 |
The best state to expand with a g(n) = 7 and h(n) = 5 is...
|1 |b |3 |
|5 |2 |6 |
|4 |7 |8 |
The best state to expand with a g(n) = 8 and h(n) = 4 is...
|1 |2 |3 |
|5 |b |6 |
|4 |7 |8 |
The best state to expand with a g(n) = 9 and h(n) = 3 is...
|1 |2 |3 |
|b |5 |6 |
|4 |7 |8 |
The best state to expand with a g(n) = 10 and h(n) = 2 is...
|1 |2 |3 |
|4 |5 |6 |
|b |7 |8 |
The best state to expand with a g(n) = 11 and h(n) = 1 is...
|1 |2 |3 |
|4 |5 |6 |
|7 |b |8 |
The best state to expand with a g(n) = 12 and h(n) = 0 is...
|1 |2 |3 |
|4 |5 |6 |
|7 |8 |b |


Goal!!

To solve this problem the search algorithm expanded a total of 158 nodes.
The maximum number of nodes in the queue at any one time was 100.
The depth of the goal node was 12
```

My C++ code: GitHub: https://github.com/mapleCows/The-eight-puzzle

```
#include <iostream>
#include <vector>
#include <string>
#include <cmath>
#include <queue>
#include <list>
using namespace std;

class Child {
        public:
        vector<char> table;
        int h;
        int g;
        Child();
        Child(vector<char>, int, int);
        int GetGH();
        int GetG();

};

void PrintVector(vector<string>);               //Helper fucntion when testing to print vector
void PrintTable(Child);                         //Prints tile board for game
vector<string> FindValidMoves(Child);                   //Calculates all valid moves for the blank
space
vector<char> CreateTable(string, int);                  //Create the starting parent table
int FindBlank(Child);                           //Finds blank in table
void MoveBlank(Child&, string, int);                    //Moves(swaps) the blank in the table
vector<char> CreateGoal(int);                   //Creates the goal board to be compared to
bool IsGoal(Child);                             //compare goal with child board
vector<Child> CreateChildren(vector<string>, Child);            //Creates children board based of
valid moves
void CalcG(Child&);                             //Calulates the G
void CalcMisplacedTile(Child&);                         //Calc H for misplaced
void CalcManhattanDistance(Child&);                     //Calc H for manhattan
int FindGoalLocation(char);                     //Finds the index of the specific tile in goal board
int FindCurrLocation(Child, char);                      //Find index of specific tile in child board
void PutIntoList(vector<Child>);                //Puts children in correct order in list
void UniformCostSearch(Child);                          //Uniform cost search
void AStarMisplacedSearch(Child);                       //A* Misplace
void AStarManhattanSearch(Child);                       //A* Manhattan
void EndGame(Child);                            //End print screen
void ChangeZeroToB(string&);                    //Changing user input to replace the 0 with b
for the blank
void RemoveSpacesAndTabs(string&);              //Removes spaces and tabs from user input

//GLOBALS
int root;                           //Square root of the board size
Child goal;                         //Goal board
list<Child> L;                      //List of children
int numOfNodes = 0;                         //Number of nodes explored
int maxNumOfQ = 0;                          //Max size of the list



int main()
{
        string input;
        string usergame = "";
        string defaultGame = "1235b6478";
        string searchChoice;

        cout << "Welcome to Luis Sanchez's 8-puzzle solver.\nType 1 to use a default puzzle, or
2 to enter your own puzzle\n";
        cin >> input;
        if (input == "2") {
        cout << "Enter your puzzle, use a 0 to represent the blank\nEnter the first row, use space
or tabs between numbers ";
        cin.ignore();
        getline(cin, input);
        usergame += input;
        cout << "Enter the second row, use space or tabs between numbers ";

        getline(cin, input);
        usergame += input;
        cout << "Enter the third row, use space or tabs between numbers ";

        getline(cin, input);
        usergame += input;
        RemoveSpacesAndTabs(usergame);  //remove spaces and tabs from user input
        ChangeZeroToB(usergame);            //change 0 to b
        }
        else { usergame = defaultGame; }


        //Game set up
        int size = usergame.length();  //size to check if perfect square root
        root = sqrt(size + 1);  //cacl root
        vector<char> pTable = CreateTable(usergame, size); //create Parent table from user
input
        vector<char> gTable = CreateGoal(size);      //create Goal Table based off size
        Child parent = Child(pTable, 0, 0);             //Create Parent object CALC mixmatch on
parents
        goal.table = gTable;                          //Goal object


        cout << "Enter your choice of algorithm\n1. Uniform Cost Search\n2. A* with the
Misplaced Tile heuristic.\n";
        cout << "3. A* with the Manhattan distance heuristic.\n";
        cin >> searchChoice;

        if (searchChoice == "1") { UniformCostSearch(parent); }
        if (searchChoice == "2") { AStarMisplacedSearch(parent); }
        if (searchChoice == "3") { AStarManhattanSearch(parent); }

        return 0;
}


void UniformCostSearch(Child parent)
{
        //General search
        /*
        1. check if goal
        2. find all valid moves
        3. calculate the order to push into Q (Later with A*)
        4. push into Q
        5. repeat
        */

        parent.h = 0;
        L.push_back(parent);
        vector<Child> children;
        cout << "Expanding State:";
        PrintTable(L.front());
        while (!L.empty())
        {

        if (IsGoal(L.front())) { EndGame(L.front()); break; }   //Checking if goal state
        vector<string> validMoves = FindValidMoves(L.front());  //Find all valid moves
        numOfNodes += validMoves.size();                        //Tally Fronteir nodes

        children = CreateChildren(validMoves, L.front());      //Creates children;
        for (int i = 0; i < children.size(); ++i) {            //Calculate the G for each child
        CalcG(children.at(i));
        }

        L.pop_front();                                         //pop parent of list
        PutIntoList(children);                                 //Put new Children into list


        cout << "The best state to expand with a g(n) = " << L.front().g;
        PrintTable(L.front());


        if (L.size() > maxNumOfQ) { maxNumOfQ = L.size(); }

        }
}

void AStarMisplacedSearch(Child parent)
{
        //General search
        /*
        1. check if goal
        2. find all valid moves
        3. calculate the order to push into Q (Later with A*)
        4. push into Q
        5. repeat
        */


        L.push_back(parent);
        vector<Child> children;
        cout << "Expanding this node:";
        PrintTable(L.front());
        while (!L.empty())
        {

        if (IsGoal(L.front())) { EndGame(L.front()); break; }   //Checking if goal state
        vector<string> validMoves = FindValidMoves(L.front());  //Find all valid moves
        numOfNodes += validMoves.size();                        //Tally Fronteir nodes

        children = CreateChildren(validMoves, L.front());      //Creates children;

        for (int i = 0; i < children.size(); ++i) {            //Calculate the G and H for each child
        CalcMisplacedTile(children.at(i));
        CalcG(children.at(i));
        }

        L.pop_front();                                          //pop parent off list
        PutIntoList(children);                                  //Put new Children into list


        cout << "The best state to expand with a g(n) = " << L.front().g << " and h(n) = " <<
L.front().h << " is...";
        PrintTable(L.front());


        if (L.size() > maxNumOfQ) { maxNumOfQ = L.size(); }         //keeping track of max size of
nodes in list

        }
}

void AStarManhattanSearch(Child parent)
{
        //General search
  /*
        1. check if goal
        2. find all valid moves
        3. calculate the order to push into Q (Later with A*)
        4. push into Q
        5. repeat
  */



        L.push_back(parent);
        vector<Child> children;
        CalcManhattanDistance(parent);
        cout << "Expanding this node:";
        PrintTable(L.front());
        while (!L.empty())
        {
```

```
        if (IsGoal(L.front())) { EndGame(L.front()); break; }   //Checking if goal state


        vector<string> validMoves = FindValidMoves(L.front());  //Find all valid moves
        numOfNodes += validMoves.size();                        //Tally Fronteir nodes

        children = CreateChildren(validMoves, L.front());       //Creates children;

        for (int i = 0; i < children.size(); ++i) {             //Calculate the G and H for each child
        CalcManhattanDistance(children.at(i));
        CalcG(children.at(i));
        }

        L.pop_front();                                          //pop parent off list
        PutIntoList(children);                                  //Put new Children into list


        cout << "The best state to expand with a g(n) = " << L.front().g << " and h(n) = " <<
L.front().h << " is...";
        PrintTable(L.front());


        if (L.size() > maxNumOfQ) { maxNumOfQ = L.size(); }     //keeping track of max size of
nodes in list

        }
}

void PrintVector(vector<string> v)
{
        for (unsigned int i = 0; i < v.size(); ++i) {
        cout << v.at(i) <<" ";
        }
}

void PrintTable(Child c)
{
        vector<char> v = c.table;
        cout << "\n|";
        int k = 0;
        for (unsigned int i = 0; i < v.size(); ++i) {
        cout << v.at(i) << " |";
        ++k;
        if (k%3 == 0 && k != v.size()) { cout << "\n|"; }
        }
        cout << endl;
}

vector<string> FindValidMoves(Child c)
{
        vector<string> validMoves;
        int bLocation = FindBlank(c);           //Blank index in child
        int x = bLocation % root;       //Calculating x cord
        int y = bLocation / root;       //y cord

        if (y != 0) { validMoves.push_back("UP"); } //up

        if (y != root - 1) { validMoves.push_back("DOWN"); } //down

        if (x != 0) { validMoves.push_back("LEFT"); } //left

        if (x != root - 1) { validMoves.push_back("RIGHT"); } //right

        return validMoves;
}

vector<char> CreateTable(string s, int size)
{
        vector<char> table;

        for (unsigned int i = 0; i < size; ++i) {
        table.push_back(s.at(i));   //pushing user input into char vector
        }

        return table;
}

int FindBlank(Child c)
{
        vector<char> table = c.table;
        int index = -1;
        for (unsigned int i = 0; i < table.size(); ++i) {
        if (table.at(i) == 'b') { index = i; }
        }
        return index;
}
```

```
void MoveBlank(Child &c, string move, int bLocation)
{
	vector<char> table = c.table;
	char temp;
	//up -root
	if (move == "UP") {
	temp = table.at(bLocation);
	table.at(bLocation) = table.at(bLocation - root);
	table.at(bLocation - root) = temp;
	}
	//down -root
	if (move == "DOWN") {
	temp = table.at(bLocation);
	table.at(bLocation) = table.at(bLocation + root);
	table.at(bLocation + root) = temp;
	}
	//right -root
	if (move == "RIGHT") {
	temp = table.at(bLocation);
	table.at(bLocation) = table.at(bLocation +1);
	table.at(bLocation +1) = temp;
	}
	//left -root
	if (move == "LEFT") {
	temp = table.at(bLocation);
	table.at(bLocation) = table.at(bLocation - 1);
	table.at(bLocation - 1) = temp;
	}
	c.table = table;
}

vector<char> CreateGoal(int size)
{
	vector<char> table;
	for (int i = 1; i < size; ++i) {
	string s = to_string(i);		//appending 12345678 or any size of the board given
	table.push_back(s[0]);
	}
	table.push_back('b');		//Adding a blank at the end
	return table;
}

bool IsGoal(Child c)
{
	if (c.table == goal.table) { return true; }

	return false;
}

vector<Child> CreateChildren(vector<string> validMoves, Child p)
{
	int bLocation = 0;
	Child c;
	vector<Child> children;

	for (int i = 0; i < validMoves.size(); ++i) {
	c = p;					//copying prarent -> child
	bLocation = FindBlank(c);			//finding blank
	MoveBlank(c, validMoves.at(i), bLocation);	//moving tile
	//cout << "\nThisis child:" << validMoves.at(i) << endl;
	//PrintTable(c);
	//CalcG(c);				//Calc G for child
	//CalcH(c);				//Calc H for child
	children.push_back(c);			//putting into q to be added to list
	}

	return children;
}

void CalcG(Child &c)
{
	++c.g;
}

void CalcMisplacedTile(Child& c)
{
	int tally = 0;
	for (int i = 0; i < c.table.size(); ++i) {

	if (c.table.at(i) == 'b') {		//skipping blank
	continue;
	}

	if (c.table.at(i) != goal.table.at(i)) {	//compare if misplaced or not
	++tally;
	}
	}
	c.h = tally;
```

```
}

void CalcManhattanDistance(Child& c)
{
        int hy = 0 , hx = 0, h = 0;
        int currTile = 0;
        int goalTile = 0;
        int cx = 0, cy = 0, gx = 0, gy = 0;
        c.h = 0;                                //Reseting H from parent

        for (int i = 0; i < c.table.size(); ++i) {
        if (c.table.at(i) == 'b') {
        continue;
        }
        currTile = FindCurrLocation(c, c.table.at(i));  //getting index of curr location
        goalTile = FindGoalLocation(c.table.at(i));     //getting index of destination

        cx = currTile % root;                   //calculating cords
        cy = currTile / root;

        gx = goalTile % root;
        gy = goalTile / root;

        c.h += abs(cx - gx) + abs(cy - gy);             //Calculating distance/num of moves
needed

        }

}

int FindGoalLocation(char c) {
        for (int i = 0; i < goal.table.size(); ++i) {
        if (c == goal.table.at(i)) {
        return i;
        }
        }
}

int FindCurrLocation(Child c, char tile) {
        for (int i = 0; i < c.table.size(); ++i) {
        if (tile == c.table.at(i)) {
        return i;
        }
        }
```

```
}

void PutIntoList(vector<Child> children)
{
	Child currC;	//current child
	Child it;	// List Child

	if (L.empty()) {			//in case the list is empty when adding in the new children
	L.push_back(children.at(0));
	children.erase(children.begin());
	}


	for(int k = 0; k < children.size(); ++k) {
	currC = children.at(k);

	for (list<Child>::iterator i = L.begin(); i != L.end(); i++) {
	it = *i;

	if (currC.GetGH() <= it.GetGH()) {	//if curr GH is less than List child insert into list
		L.insert(i, currC);
		break;
	}
	if (currC.GetGH() >= L.back().GetGH()) {	//If curr is greather than the end of the list
		L.insert(L.end(), currC);
		break;
	}
	}
	}

}

void EndGame(Child c)
{
	//PrintTable(q.front());
	cout << "\n\nGoal!!\n\nTo solve this problem the search algorithm expanded a total of
"<<numOfNodes <<" nodes.\n";
	cout << "The maximum number of nodes in the queue at any one time was " <<
maxNumOfQ << ".\n";
	cout << "The depth of the goal node was " << L.front().g << endl;

}

void ChangeZeroToB(string& s )
```

```
{
        for (int i = 0; i < s.size(); ++i) {
        if (s.at(i) == '0') { s.at(i) = 'b'; }
        }
}

void RemoveSpacesAndTabs(string& s)
{

        for (string::iterator i = s.begin(); i != s.end(); ++i) {
        if (*i == ' ' || *i == '\t') {
        s.erase(i);
        --i;
        }
        }
}

Child::Child()
{
}

Child::Child(vector<char> table, int g, int h)
{
        this->table = table;
        this->h = h;
        this->g = g;
}

int Child::GetGH()
{
        return (this->g + this->h);
}

int Child::GetG()
{
        return (this->g);
}
```