Jacques Fracchia
862010872
jfrac001@ucr.edu
May 5th, 2020

CS170: Intro to Artificial Intelligence
Dr. Niloofar

Project 1: 8 Puzzle Solver

Introduction:

This eight puzzle solver uses the powerful A* search. The puzzle offers three different implementations of the A* search, a uniform cost search, A* with the misplaced Tile heuristic and A* with the Euclidean Distance heuristic. The user has the option to use a default puzzle hard codded into the game, or they have an option to enter their own puzzle. This 8 puzzle solver will then output the different expanded nodes for each step and show what their g(n) and h(n) values are. Final results will show if the goal state was reached or not, and if so the program outputs how many total expanded nodes and the maximum number of nodes in the queue at any time.

Uniform Cost Search:

Uniform Cost Search is an algorithm where A* search has been degraded by setting h(n) to 0. Disregarding the h(n) component the A* search selects the cheapest node in regards to g(n). The cost is uniform because each expanded node only has a cost of 1.

A* with the Misplaced Tile heuristic:

A* search with misplaced tile heuristic stores a lot more data in memory to allow the algorithm determine the next best move to find a more optimal path to the goal. The misplaced tile heuristic goes through the puzzle and records the number of misplaced tiles in the puzzle, disregarding the blank space. The algorithm sets h(n) to the total number of misplaced tiles in the puzzle and places nodes in a queue. The algorithm then goes through the queue and expands the node that has the smallest sum of g(n) and h(n) values to find a more optimal path than uniform cost search.

A* with the Euclidean Distance heuristic:

A* search with Euclidean Distance heuristic is similar to the misplaced tile heuristic where it calculates the sum of the total distances between each tile and the goal state, however it calculates that distance with the euclidean formula compared to the misplaced tile heuristic more general formula. Using the euclidean formula takes into account the distance between the diagonals as well, and though it can be more costly for time and memory than misplaced tile heuristic the additional information can in theory, lead it to more optimal paths. The algorithm computes the total number of misplaced tiles in the puzzle using euclidean distance formula then sets h(n) to that value. The g(n) and h(n) values are then summed up and put into a queue for each node. A* search then expands the least expensive node.

High Level Design

The 8 puzzle game was written with an object oriented design. Each tile was represented as a node in a struct with each possible move operation as leaves and a heuristic cost and uniform cost associated with each node or tile. The node's heuristic values are calculated differently depending on which heuristic the user chooses to use. The algorithm places each new node into the priority queue and places the node into a vector of nodes to represent nodes we have already seen. Next it pops out the next best possible option from the priority queue and expands out the possible operations that tile can move placing those new nodes into the priority queue. The algorithm continues to do this until the priority queue is empty resulting in each tile being computed and compared to their goal state, leading to the most optimal solution.

Data

Six different puzzles were analyzed when comparing the different optimizations of heuristics.

Trivial:

1	2	3
4	5	6
7	8	0

Very Easy:

1	2	3
4	5	0
7	8	6

Easy:

1	2	3
4	5	6
0	7	8

Hard:

4	1	2
0	5	3
7	8	6

Very Hard:

1	2	3
6	0	4
5	7	8

Impossible:

1	2	3
4	5	6
8	7	0

Maximum Queue Size

Puzzle	Uniform Cost Search	Misplaced Tile Heuristic	Euclidean Distance Heuristic
Trivial	0	0	0
Very Easy	2	1	1
Easy	3	2	2
Hard	35	5	5
Very Hard	5461	302	303
Impossible	-	-	-

Figure 1: Table showing various puzzles and their queue sizes for each heuristic used.

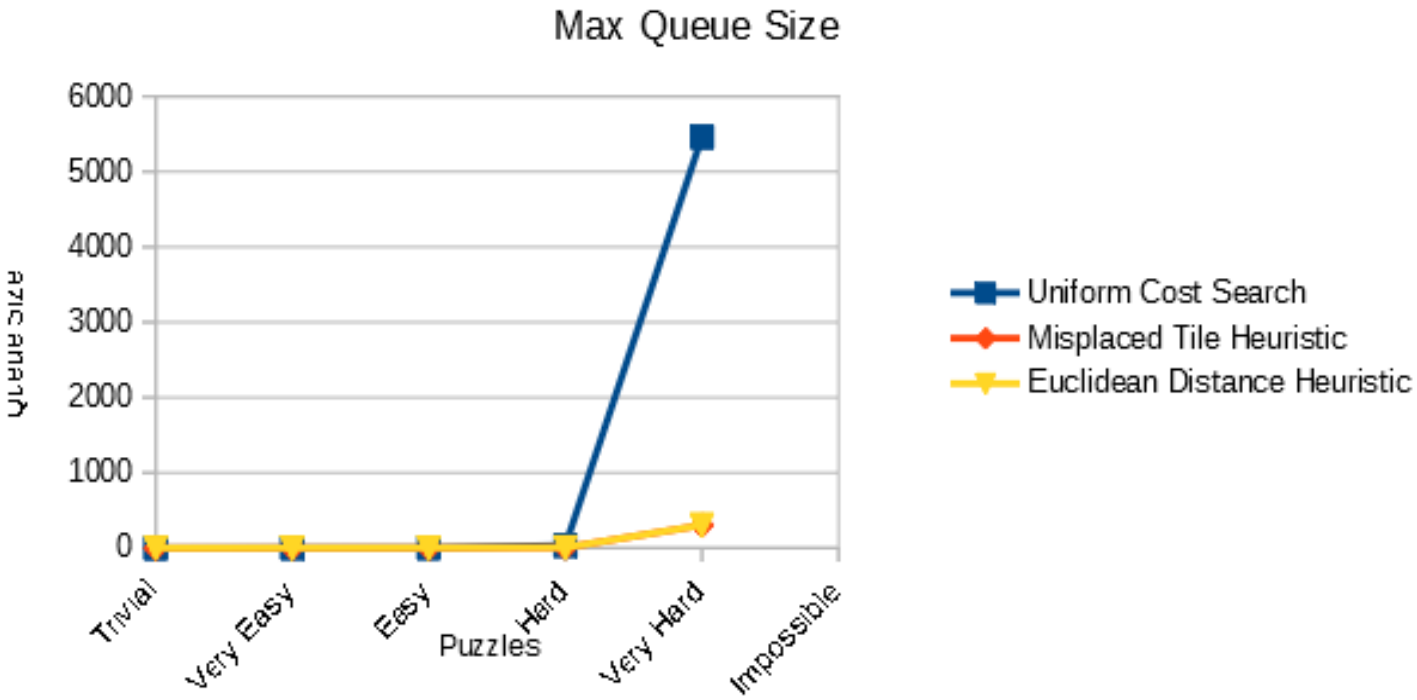

Figure 2: Chart showing the different queue sizes for each heuristic. Notice that uniform cost search has been scaled down for very hard in order to show the closeness between misplaced tile and euclidean distance heuristic.

Number of Nodes Expanded

Puzzle	Uniform Cost Search	Misplaced Tile Heuristic	Euclidean Distance Heuristic
Trivial	0	0	0
Very Easy	3	1	1
Easy	3	2	2
Hard	28	4	4
Very Hard	3500	214	190
Impossible	-	-	-

Figure 3: Table showing various puzzle's number of nodes expanded for each Heuristic.

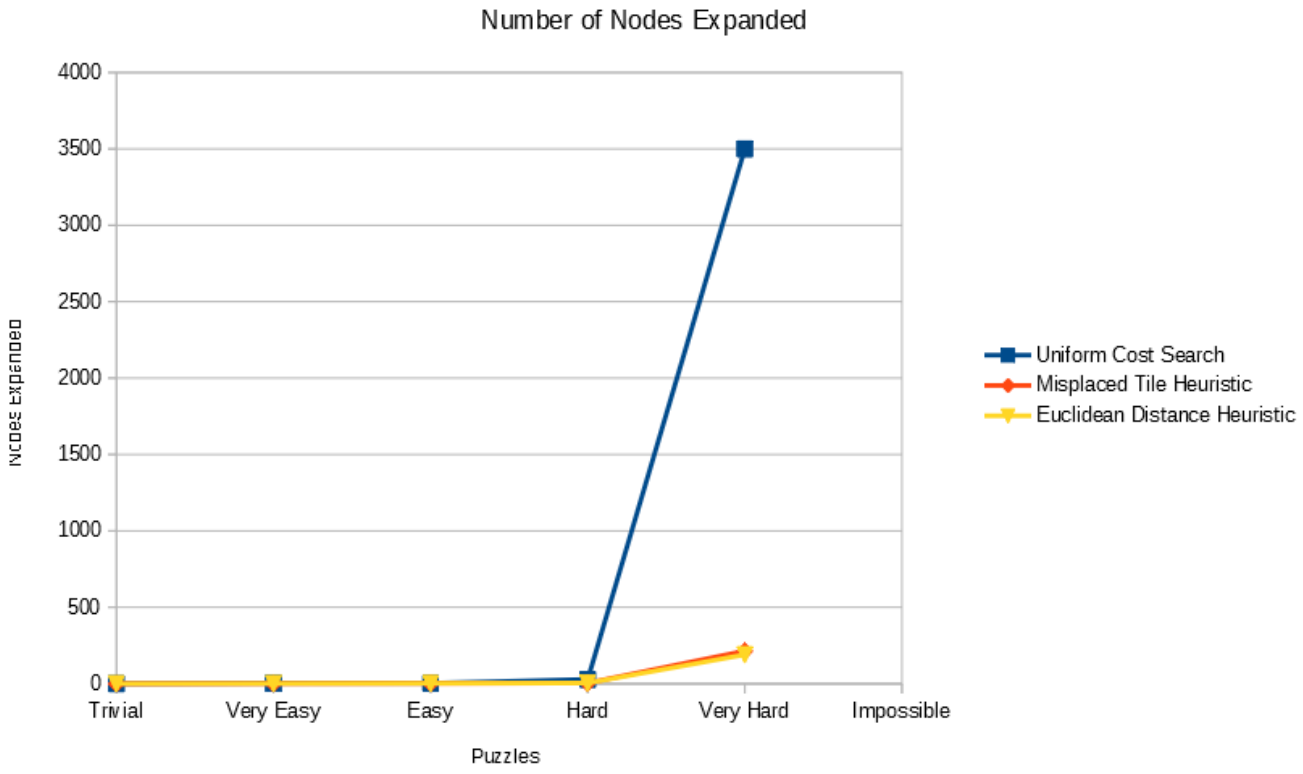

Figure 4: Chart showing the number of nodes expanded for each heuristic over the six different puzzles.

Conclusion

Taking a look at the data in tables from figure 1 and 3 it is clear to see that for the first few easy puzzles there was not a big difference between the different heuristics. We can see that the numbers only changed slightly between each one. This is expected because these puzzles were made to be completed in only a couple of steps.

Analyzing the more difficult puzzles it is clear that the misplaced tile and euclidean distance heuristics outperformed the uniform cost search by magnitudes. Looking at the graphs from figures 2 and 4 we can see that the two misplaced and euclidean heuristics generally performed about the same, with euclidean distance slightly outperforming misplaced tile heuristic.

The two heuristics, Euclidean distance and misplaced tile, outperformed uniform cost search because the heuristics used up more memory for keeping track of more useful data when determining the best next path to take. These heuristics take advantage of both g(n), the cost it took for the path to go from the starting point to its current point, and h(n), the cost of the distance from its current location to its goal destination.

Uniform cost search hard codes this h(n) value to zero so it saves on space but for difficult puzzles it will have much longer to find the optimal path. While the misplaced tile and euclidean distance takes the time and memory to calculate the distance each tile's move would be to get to the goal, then taking that into account when figuring out which node to expand next.

For trivial puzzles it does not matter which heuristic you use, they will all find the path quickly. But for more difficult non trivial puzzles using a good heuristic will save magnitudes of time when finding the most optimal path to the goal.

Traceback

Puzzle:

1	0	3
4	2	6
7	5	8

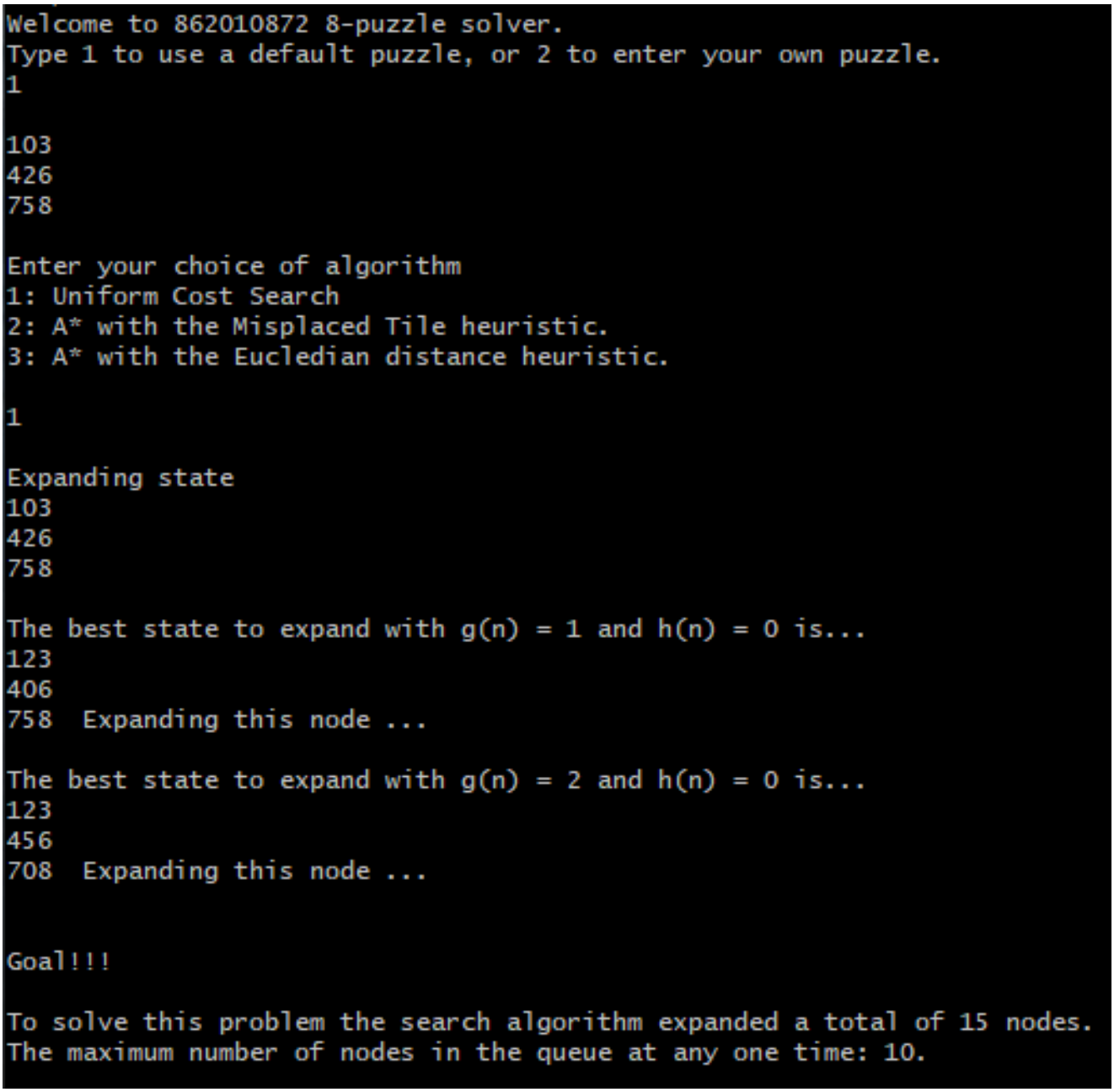

Figure 5: Traceback for requested puzzle running with Uniform Cost Search

```
Welcome to 862010872 8-puzzle solver.
Type 1 to use a default puzzle, or 2 to enter your own puzzle.
1

103
426
758

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

2

Using Misplaced Heuristic

Expanding state
103
426
758

The best state to expand with g(n) = 1 and h(n) = 2 is...
123
406
758  Expanding this node ...

The best state to expand with g(n) = 2 and h(n) = 1 is...
123
456
708  Expanding this node ...


Goal!!!

To solve this problem the search algorithm expanded a total of 3 nodes.
The maximum number of nodes in the queue at any one time: 5.
```

Figure 6: Traceback for requested puzzle running with Misplaced Tile heuristic.

```
Welcome to 862010872 8-puzzle solver.
Type 1 to use a default puzzle, or 2 to enter your own puzzle.
1

103
426
758

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

3

Using Eucledian Heuristic

Expanding state
103
426
758

The best state to expand with g(n) = 1 and h(n) = 2 is...
123
406
758  Expanding this node ...

The best state to expand with g(n) = 2 and h(n) = 1 is...
123
456
708  Expanding this node ...


Goal!!!

To solve this problem the search algorithm expanded a total of 3 nodes.
The maximum number of nodes in the queue at any one time: 5.
```

Figure 7: Traceback for requested puzzle running with Euclidean Distance heuristic

Code:

main.cpp

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include "operators.h"
#include "node.h"
#include "heuristics.h"

using namespace std;

vector <node> hit;
priority_queue <node, vector<node>, GreaterThan> search_queue;

bool is_hit(vec vector);
void print_nodes(node val, int size, int expanded_nodes);
void graph_search(vec puzzle, string heuristics);

int main() {
	vec puzzle(3, vector<int>(3));
	vector <int> input;
	string heuristic;
	int option = 0;
	int index = 0;

	//Default values
	int i = 1, j = 0, k = 3;
	int l = 4, m = 2, n = 6;
	int o = 7, p = 5, q = 8;

	cout << "Welcome to 862010872 8-puzzle solver." << endl;
	cout << "Type 1 to use a default puzzle, or 2 to enter your own puzzle." << endl;
	cin >> option;
	cout << endl;

	if(option == 2){
		cout << "Enter your puzzle, use a zero to represent the blank" << endl;
		cout << "Enter the first row, use space or tabs between numbers" << endl;
		cin >> i >> j >> k;
		cout << "Enter the second row, use space or tabs between numbers" << endl;
		cin >> l >> m >> n;
		cout << "Enter the third row, use space or tabs between numbers" << endl;
		cin >> o >> p >> q;
	}


	input.push_back(i);
	input.push_back(j);
	input.push_back(k);
	input.push_back(l);
	input.push_back(m);
	input.push_back(n);
	input.push_back(o);
	input.push_back(p);
```

```
	input.push_back(q);

	//Make our puzzle from the inputs
	for (int i = 0; i < 3; i++) {
		for (int j = 0; j < 3; j++) {
			puzzle.at(i).at(j) = input.at(index); //assign the vector index to the
input for that index
			cout << puzzle.at(i).at(j); //print out each square
			index++;
		}
		cout << endl;
	}

	cout << endl;
	cout << "Enter your choice of algorithm" << endl;
	cout << "1: Uniform Cost Search" << endl;
	cout << "2: A* with the Misplaced Tile heuristic." << endl;
	cout << "3: A* with the Eucledian distance heuristic." << endl << endl;
	cin >> option;
	cout << endl;


	switch (option) {
		case 1:
			heuristic = "Default";
			break;

		case 2:
			heuristic = "Misplaced";
			break;

		case 3:
			heuristic = "Eucledian";
			break;

		default:
			heuristic = "Default";
			break;
	}

	graph_search(puzzle, heuristic);

	return 0;
}

bool is_hit(vec vector) {
	bool result = false;
	int size = hit.size();

	for (int i = 0; i < size; i++) {
		if (hit.at(i).vector == vector)
			result = true;
	}

	return result;
}

void print_nodes(node val, int size, int expanded_nodes) {
	node temp = val;
	stack<node> stack;
```

```
	stack.push(temp);                                    //add new node to stack

	while (temp.root != NULL) {                 //add nodes to stack, stop at root
		temp = *temp.root;
		stack.push(temp);

	}

	cout << "Expanding state" << endl;
	stack.top().print();
	cout << endl << endl;

	stack.pop();

	while (stack.size() >= 2) { //Expand out every node and explain
		cout << "The best state to expand with g(n) = " << stack.top().uni_price << " and
h(n) = " << stack.top().heu_price << " is..." << endl;
		stack.top().print();
		cout << "Expanding this node ..." << endl << endl;
		stack.pop();
	}

	cout << endl << "Goal!!!" << endl << endl;
	cout << "To solve this problem the search algorithm expanded a total of " <<
expanded_nodes << " nodes." << endl;
	cout << "The maximum number of nodes in the queue at any one time: " << size << "." <<
endl;

}

void graph_search(vec puzzle, string heuristics) {
	node curr(puzzle);
	int size = 0;
	int expanded_nodes = 0;

	if (heuristics == "Misplaced") {
		cout << "Using Misplaced Heuristic" << endl << endl;
	}

	else if (heuristics == "Eucledian") {
		cout << "Using Eucledian Heuristic" << endl << endl;
	}

	hit.push_back(curr);
	search_queue.push(curr);

	while (!search_queue.empty()) {

		node* curr = new node(search_queue.top());

		if (search_queue.top().vector == goal) {
			print_nodes(search_queue.top(), size, expanded_nodes);
			break;

		} else {

			hit.push_back(search_queue.top()); //Add to hit list

			int temp = search_queue.size();
```

```
                        if (temp > size) {                  //get max queueueue size
                                size = temp;
                        }

                        search_queue.pop();
                        expanded_nodes++;

                        //Make our node opeartions
                        node* node_right = new node(move_right(curr->vector), curr->uni_price +
1);
                        node* node_left = new node(move_left(curr->vector), curr->uni_price + 1);
                        node* node_up = new node(move_up(curr->vector), curr->uni_price + 1);
                        node* node_down = new node(move_down(curr->vector), curr->uni_price + 1);

                        //Get our prices for each operation
                        if (heuristics == "Misplaced") {
                                node_right->heu_price = misplaced_heuristics(node_right->vector);
                                node_left->heu_price = misplaced_heuristics(node_left->vector);
                                node_up->heu_price = misplaced_heuristics(node_up->vector);
                                node_down->heu_price = misplaced_heuristics(node_down->vector);
                        }

                        if (heuristics == "Eucledian") {
                                node_right->heu_price = eucledian_heruistics(node_right->vector);
                                node_left->heu_price = eucledian_heruistics(node_left->vector);
                                node_up->heu_price = eucledian_heruistics(node_up->vector);
                                node_down->heu_price = eucledian_heruistics(node_down->vector);
                        }

                        //calculate total sum for each operation

                        //Get right operation
                        node_right->sum = node_right->uni_price + node_right->heu_price;
                        //if we havnt hit right node add it to our search queue
                        if (!is_hit(node_right->vector)) {
                                curr->leaf_3 = node_right;
                                node_right->root = curr;
                                search_queue.push(*node_right);
                        }

                        //Get left operation
                        node_left->sum = node_left->uni_price + node_left->heu_price;
                        //if we havnt hit left node add it to our search queue
                        if (!is_hit(node_left->vector)) {
                                curr->leaf_2 = node_left;
                                node_left->root = curr;
                                search_queue.push(*node_left);
                        }

                        //Get up operation
                        node_up->sum = node_up->uni_price + node_up->heu_price;
                        //if we havnt hit up node add it to our search queue
                        if (!is_hit(node_up->vector)) {
                                curr->leaf_1 = node_up;
                                node_up->root = curr;
                                search_queue.push(*node_up);
                        }

                        //Get down operation
                        node_down->sum = node_down->uni_price + node_down->heu_price;
```

```
                        //if we havnt hit down node add it to our search queue
                        if (!is_hit(node_down->vector)) {
                                curr->leaf_0 = node_down;
                                node_down->root = curr;
                                search_queue.push(*node_down);
                        }
                }
        }

        if (search_queue.empty())
                cout << endl << "No Solution" << endl;

}
```

heuristic.h

```
#ifndef HEURISTICS_H
#define HEURISTICS_H
#include <iostream>
#include <vector>
#include <math.h>
using namespace std;

const vec goal = {{1, 2, 3}, {4, 5, 6}, {7, 8, 0}};

void next_state(int vector, int &x, int &y) { //euclidian helper
        for(int i = 0; i<3; i++) {
                for(int j = 0; j<3; j++) {

                        if(goal.at(i).at(j) == vector) { //find next state to work at
                                x = i;
                                y = j;
                                return;       //return puzzle with updated indicies
                        }

                }
        }
}

int eucledian_heruistics(vec vector) { // result = sqrt(pow(x1-y1, 2) + pow(x2-y2, 2))
        int result = 0;

        for (int i = 0; i<3; i++) {
                for (int j = 0; j<3; j++) {

                        if (vector.at(i).at(j) == 0)
                                continue;                           //blank space


                        if (vector.at(i).at(j) != goal.at(i).at(j)) { //incorrect state
                                int x = 0;
                                int y = 0;

                                next_state(vector.at(i).at(j), x, y);      //get correct index
                                x = pow(x - y, 2); //Euclidean Formula!
                                y = pow(i - j, 2); //Euclidean Formula!
```

```
                        while ( (x != 0) || (y != 0) ) {         //calculate distance and
make sure not negative

                                if (x < 0) {
                                        x = x + 1;
                                        result++;
                                }

                                if (x > 0) {
                                        x = x - 1;
                                        result++;
                                }

                                if (y < 0) {
                                        y = y + 1;
                                        result++;
                                }

                                if (y > 0) {
                                        y = y - 1;
                                        result++;
                                }

                        }

                }//columns
        }//rows

    }

    return result;
}

int misplaced_heuristics(vec vector) {
    int result = 0;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {

            if (vector.at(i).at(j) == 0)
                continue;

            if (goal.at(i).at(j) != vector.at(i).at(j))
                result++;

        }//columns
    }//rows

    return result;
}

#endif
```

node.h

```
#ifndef NODE_H
#define NODE_H
#include <iostream>
#include <vector>
using namespace std;
typedef vector <vector <int>> vec;

struct node {
	vec vector;
	int heu_price = 0;
	int uni_price = 0;
	int sum = 0;

	node *root = NULL;
	node *leaf_0 = NULL;
	node *leaf_1 = NULL;
	node *leaf_2 = NULL;
	node *leaf_3 = NULL;

	node(vec new_vector) {
		vector = new_vector;

		heu_price = 0;
		uni_price = 0;
	};

	node(vec new_vector, int uniform_price) {
		vector = new_vector;

		heu_price = 0;
		uni_price = uniform_price;
	};

	node(const node &new_vector) {
		vector = new_vector.vector;

		heu_price = new_vector.heu_price;
		uni_price = new_vector.uni_price;
		sum = new_vector.sum;

		leaf_0 = new_vector.leaf_0;
		leaf_1 = new_vector.leaf_1;
		leaf_2 = new_vector.leaf_2;
		leaf_3 = new_vector.leaf_3;
		root = new_vector.root;
	}

	void operator=(const node &new_vector) {
		vector = new_vector.vector;

		heu_price = new_vector.heu_price;
		uni_price = new_vector.uni_price;
		sum = new_vector.sum;

		leaf_0 = new_vector.leaf_0;
		leaf_1 = new_vector.leaf_1;
		leaf_2 = new_vector.leaf_2;
		leaf_3 = new_vector.leaf_3;
		root = new_vector.root;
```

```
	}

	void print() {
		for (int i = 0; i < 3; i++) {
			for (int j = 0; j < 3; j++) {
				cout << vector.at(i).at(j);
			}
			if (i < 2) {
				cout << endl; //next row
			}
			else {
				cout << "  ";
			}
		}
	}

};


//Compare nodes values
struct GreaterThan  {
	bool operator() (const node& left, const node& right)  {
		bool result;
		result = left.sum > right.sum;

		return result;
	}
};

#endif
```

operators.h

```
#ifndef OPERATORS_H
#define OPERATORS_H
#include <iostream>
#include <vector>

using namespace std;

typedef vector <vector <int>> vec;

vec swap(vec vector, int i, int j, int k, int l) {
	int temp = vector.at(k).at(l);

	vector.at(i).at(j) = temp;
	vector.at(k).at(l) = 0;

	return vector;
}


//Returns vector that is moved left. If zero is in the right column then return the same vector
vec move_left(vec vector) {
	int x = 0;
	int y = 0;

	//check for illegal condition
	for (int i = 0; i < 3; i++) {
		if (vector.at(i).at(2) == 0)
```

```
                return vector;
        }

        for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 2; j++) {

                        if (vector.at(i).at(j) == 0) {
                                x = i;
                                y = j;
                        }

                }
        }

        return swap(vector, x, y, x, y+1);
}

//Returns vector that is moved right. If zero is in the left column then return the same vector
vec move_right(vec vector) {
        int x = 0;
        int y = 0;

        //check for illegal condition
        for (int i = 0; i < 3; i++) {
                if (vector.at(i).at(0) == 0)
                        return vector;
        }


        for (int i = 0; i < 3; i++) {
                for (int j = 1; j < 3; j++) {

                        if (vector.at(i).at(j) == 0) {
                                x = i;
                                y = j;
                        }

                }
        }

        return swap(vector, x, y, x, y-1);
}

//Returns vector that is moved up. If zero is in the bottom row then return the same vector
vec move_up(vec vector) {
        int x = 0;
        int y = 0;

        //check for illegal condition
        for (int i = 0; i < 3; i++) {
                if (vector.at(2).at(i) == 0)
                        return vector;
        }

        for (int i = 0; i < 2; i++) {
                for (int j = 0; j < 3; j++) {

                        if (vector.at(i).at(j) == 0) {
                                x = i;
                                y = j;
```

```
                        }

                }
        }

        return swap(vector, x, y, x+1, y);
}

//Returns vector that is moved down. If zero is in the top row then return the same vector
vec move_down(vec vector) {
        int x = 0;
        int y = 0;

        //check for illegal condition
        for (int i = 0; i < 3; i++) {
                if (vector.at(0).at(i) == 0)
                        return vector;
        }

        for (int i = 1; i < 3; i++) {
                for (int j = 0; j < 3; j++) {

                        if(vector.at(i).at(j) == 0) {
                                x = i;
                                y = j;
                        }

                }
        }
        return swap(vector, x, y, x-1, y);
}

#endif
```

makefile

```
all:
        g++ -std=c++11 main.cpp -o main.o

clean:
        rm -rf bin
```

Sources

In completing this assignment I consulted:

1. All content related to heuristics in power point slides and reading material offered by the professor.
2. A few Geeks for Geeks websites:
    1. https://www.geeksforgeeks.org/a-search-algorithm/
    2. https://www.geeksforgeeks.org/euclidean-algorithms-basic-and-extended/
    3. https://www.geeksforgeeks.org/8-puzzle-problem-using-branch-and-bound/
    4. https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/
3. A few websites for general ideas for what classes and functions are needed to be implemented
    1. https://faramira.com/solving-the-8-puzzle-problem-using-a-star-algorithm/
    2. https://blog.goodaudience.com/solving-8-puzzle-using-a-algorithm-7b509c331288
4. About a dozen random stack overflow posts for debugging