Justin Luo
jluo036@ucr.edu
SID: 862013749
Due: 2/20/21

CS170: Introduction to Artificial Intelligence
Dr. Eamonn Keogh

In completing this assignment I consulted:
- https://www.geeksforgeeks.org/stl-priority-queue-for-structure-or-class/
- https://www.geeksforgeeks.org/clearing-the-input-buffer-in-cc/
- https://www.geeksforgeeks.org/sum-manhattan-distances-pairs-points/
- Dr. Eamonn Keogh's lecture videos
- All puzzle layouts used for testing provided by Dr. Eamonn Keogh

Table of contents:

CS170 Project 1: 8-puzzle solver report

Intro:
The purpose of this project is to solve the 8-puzzle using a general search algorithm with three different heuristics. By doing so, we can see the impact a good/bad heuristic (or no heuristic) has on the time and space complexity of a problem. 3 algorithms (uniform cost search, A* with misplaced tile heuristic, and A* with manhattan distance heuristic) were tested on 8 puzzles of increasing difficulty. The number of nodes expanded and max queue size for each puzzle is documented below. My language of choice was C++ and the full code is included at the end.

8-puzzle overview:

- Objective: slide tiles numbered 1-8 in a 3x3 grid so that the numbers are in order
- A move consists of sliding a tile up, down, left, or right into the blank space.
- At any given state of the puzzle, 2-4 different moves (move tile up, down, left, or right) can be made to transition to a new state
- If the blank tile is in a corner, only 2 possible moves can be made.
- If the blank tile is in the center, all 4 moves can be made.
- If the blank tile is not in the center nor a corner then 3 possible moves can be made
- Thus the branching factor (average number of possible moves that can be made at any given state) of the 8-puzzle is (4*2+1*4+4*3)/9 ≈ 2.67.

Algorithm comparison:

- General A* search:
    - Expand the state with the smallest f(n) value, where f(n) = g(n) + h(n) and g(n) is the depth of the state and h(n) is the value of the heuristic of that state.
    - Recursively expand states based on the smallest f(n) until the goal state is found

- Uniform cost search:
    - No heuristic function, h(n) is hardcoded to 0, so f(n) = g(n)
    - Expands all the states at depth n before expanding any states at depth n+1.
    - Time complexity to find a state at depth *d* is O(b^d) where *b* is the branching factor; as described earlier, the branching factor of the 8-puzzle is about 2.67, thus the runtime is O(2.67^d).

- A* with misplaced tile heuristic:
    - Adds a heuristic to the general A* algorithm where h(n) equals the number of misplaced tiles (not including the blank tile) .

- A* with manhattan distance heuristic:
    - Adds a heuristic to the general A* algorithm where h(n) equals the sum of the minimal number of moves it would take to get every misplaced tile (not including the blank tile) in the correct position, assuming no other tiles are in the way.

Comparison of algorithms on nodes expanded vs depth:
Figure 1 below shows the number of nodes expanded for 8 difficulties of the 8-puzzle. For readability, a scaled chart is shown (figure 2).

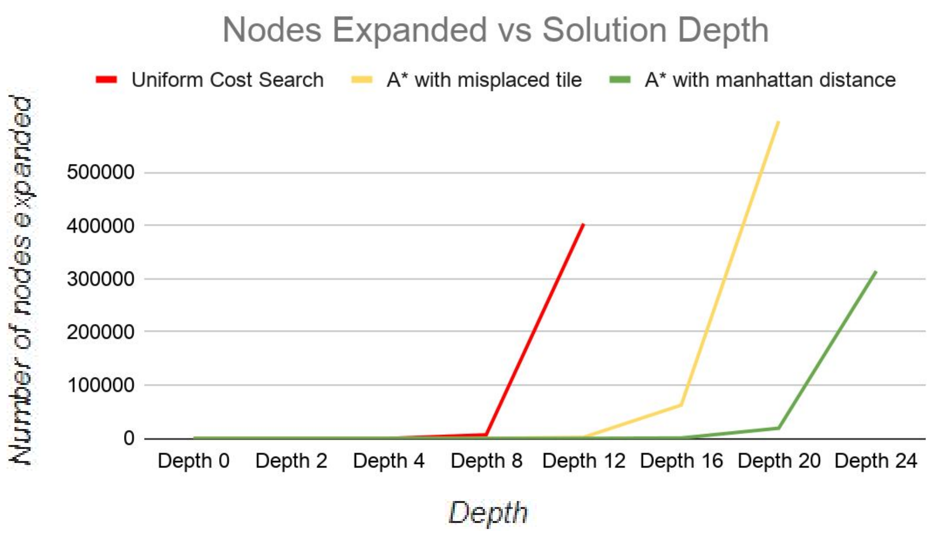


Figure 1: Y-axis represents the number of nodes expanded, x-axis represents the depth. Note that the data for Uniform cost search could not be recorded for depth 16-24. A* with misplaced tile could not be recorded for depth 24.

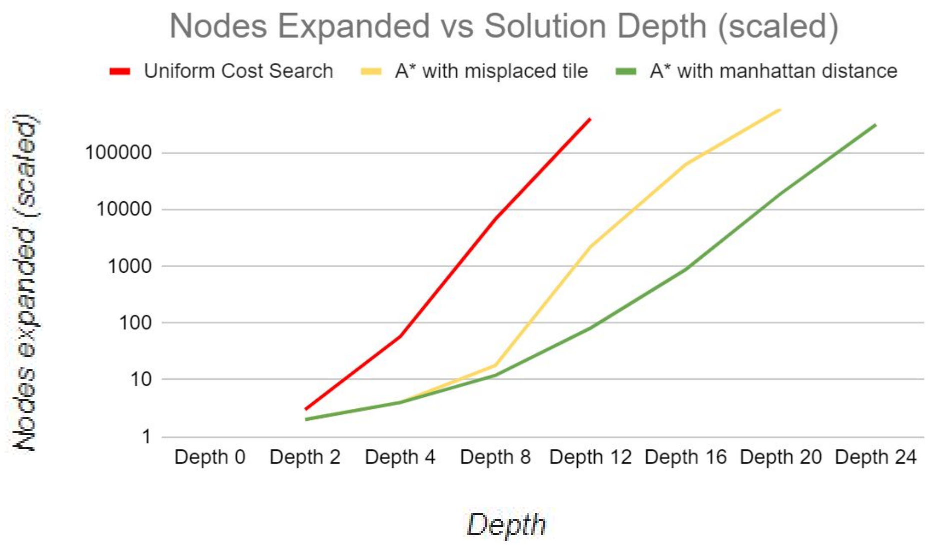


Figure 2: Scaled version of the chart in figure 1. Note that nodes expanded at depth 0 was 0 for all three algorithms.

The number of nodes expanded positively correlates with the runtime of the program, so the less nodes expanded means the faster the program runs. In figure 1 we see that all three lines grow exponentially with increasing depth. In figure 2, the lines look more linear but that is only because the y axis is increasing exponentially. Looking at figure 2, we see that uniform cost search requires the most node expansions of the three algorithms, thus it is by far the slowest. We see that uniform cost search required more expansions at depth 12 than A* with manhattan distance needed at depth 24! Comparing the misplaced tile vs the manhattan distance heuristic, we see that they performed similarly up until depth 8. Then from depth 12 - 20, manhattan distance performed noticeably better.

Comparison of algorithms on max queue size vs depth:
Figure 3 shows the maximum number of nodes enqueued at a given time for the 8 difficulties. For readability, a scaled chart is shown (figure 4).

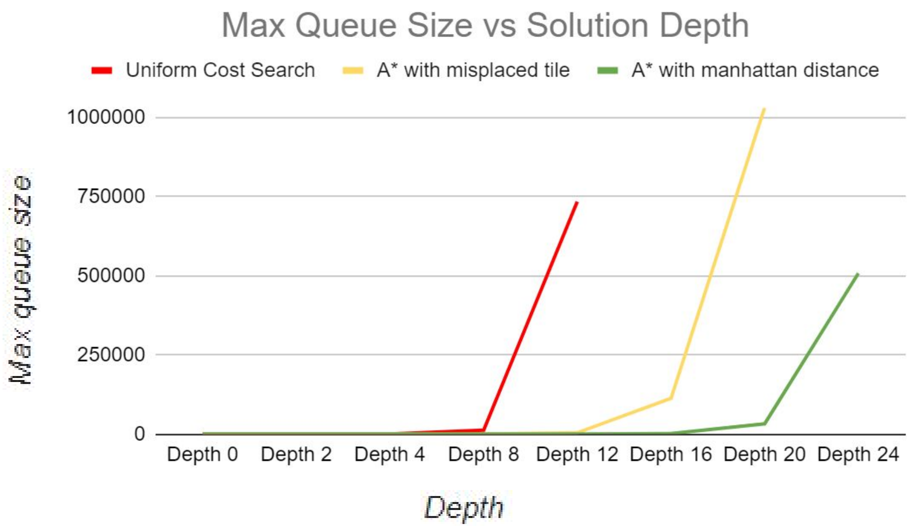


Figure 3: Y-axis represents the max queue size, x-axis represents the depth. Note that the data for Uniform cost search could not be recorded for depth 16-24. A* with misplaced tile could not be recorded for depth 24.

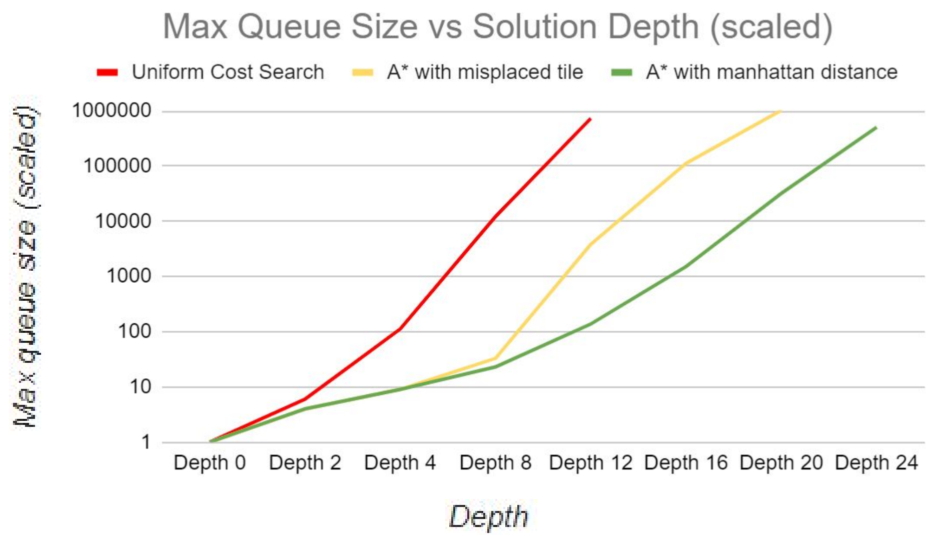


Figure 4: Scaled version of the chart in figure 3

Max queue size positively correlates with the amount of memory the program needs, so the smaller the max queue size the better. Max queue size also positively correlates with the number of nodes expanded, since all children of expanded nodes are enqueued. The max queue size also grows exponentially with increasing depth. Looking at figure 2, the max queue size for uniform cost search is noticeably higher than the other two at all depths besides 0. Comparing the misplaced tile vs the manhattan distance heuristic, we see that they had similar max queue sizes up until depth 8. Then from depth 12 - 20, manhattan distance had a noticeably less max queue size.

In conclusion:

- **Time and space complexity grows exponentially as depth increases for all three algorithms.**
- **There is little difference between the performance of the algorithms at lower depths, but this gap widens quickly as depth increases.**
- **Uniform cost search is the slowest and takes up the most space, since there is no heuristic (h(n) is always 0), so the algorithm checks and expands ALL nodes at each depth, essentially performing a breadth first search.**
- **Adding a heuristic to a search algorithm can dramatically improve time and space complexity since not all nodes need to be expanded to solve the problem; only the "promising" nodes will be expanded.**
- **The quality of the heuristic is important too. In this case, the Manhattan distance heuristic outperforms the misplaced tile heuristic. In other words, using the manhattan distance heuristic provides a more accurate indicator of how close a particular puzzle state is to being solved compared to using the misplaced tile heuristic.**

Depth 0
1,2,3,4,5,6,7,8,0 - (passing in a puzzle that already solved):
- Uniform cost:
    - nodes expanded: 0
    - max nodes enqueued: 1
- A* with misplaced tile heuristic:
    - nodes expanded: 0
    - max nodes enqueued: 1
- A* with manhattan distance heuristic:
    - nodes expanded: 0
    - max nodes enqueued: 1

Depth 2
1,2,3,4,5,6,0,7,8:
- Uniform cost:
    - nodes expanded: 3
    - max nodes enqueued: 6
- A* with misplaced tile heuristic:
    - nodes expanded: 2
    - max nodes enqueued: 4
- A* with manhattan distance heuristic:
    - nodes expanded: 2
    - max nodes enqueued: 4

Depth 4
1,2,3,5,0,6,4,7,8:
- Uniform cost:
    - nodes expanded: 58
    - max nodes enqueued: 113
- A* with misplaced tile heuristic:
    - nodes expanded: 4
    - max nodes enqueued: 9
- A* with manhattan distance heuristic:
    - nodes expanded: 4
    - max nodes enqueued: 9

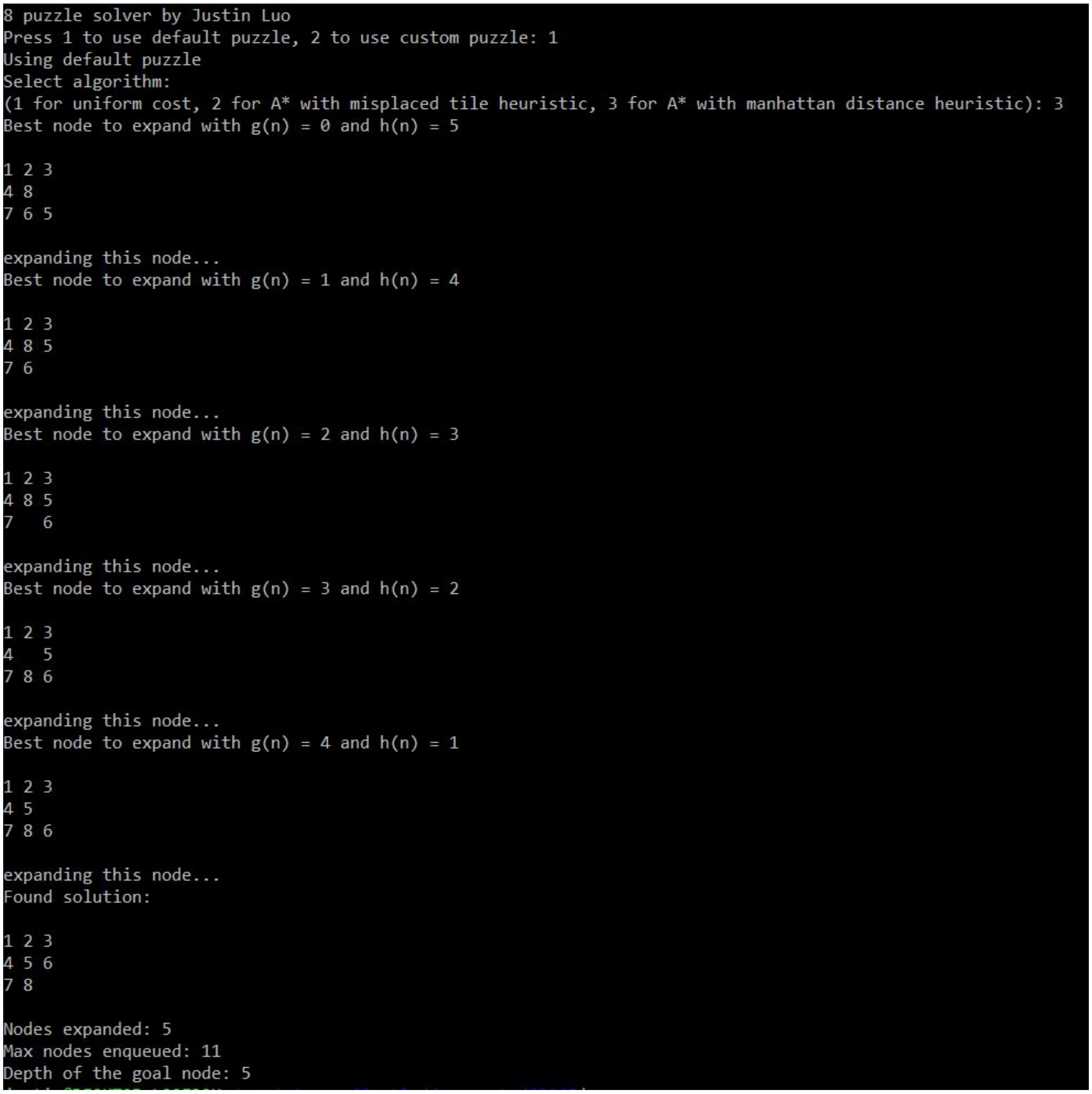
Depth 8
1,3,6,5,0,2,4,7,8:
- Uniform cost:
    - nodes expanded: 6814
    - max nodes enqueued: 12287
- A* with misplaced tile heuristic:
    - nodes expanded: 18
    - max nodes enqueued: 33
- A* with manhattan distance heuristic:
    - nodes expanded: 12
    - max nodes enqueued: 23

Depth 12
1,3,6,5,0,7,4,8,2:
- Uniform cost:
    - nodes expanded: 403586
    - max nodes enqueued: 734557
- A* with misplaced tile heuristic:
    - nodes expanded: 2223
    - max nodes enqueued: 3859
- A* with manhattan distance heuristic:
    - nodes expanded: 82
    - max nodes enqueued: 138

Depth 16
1,6,7,5,0,3,4,8,2:
- Uniform cost:
    - Timed out(30 min cutoff)
- A* with misplaced tile heuristic:
    - nodes expanded: 62623
    - max nodes enqueued: 112836
- A* with manhattan distance heuristic:
    - nodes expanded: 881
    - max nodes enqueued: 1521

Depth 20
7,1,2,4,8,5,6,3,0:
- Uniform cost:
    - Timed out(30 min cutoff)
- A* with misplaced tile heuristic:
    - nodes expanded: 596447
    - max nodes enqueued: 1031191
- A* with manhattan distance heuristic:
    - nodes expanded: 19225
    - max nodes enqueued: 32310

Depth 24
0,7,2,4,6,1,3,5,8:
- Uniform cost:
    - Timed out(30 min cutoff)
- A* with misplaced tile heuristic:
    - Timed out(30 min cutoff)
- A* with manhattan distance heuristic:
    - nodes expanded: 314421
    - max nodes enqueued: 508096



```
8 puzzle solver by Justin Luo
Press 1 to use default puzzle, 2 to use custom puzzle: 1
Using default puzzle
Select algorithm:
(1 for uniform cost, 2 for A* with misplaced tile heuristic, 3 for A* with manhattan distance heuristic): 3
Best node to expand with g(n) = 0 and h(n) = 5

1 2 3
4 8
7 6 5

expanding this node...
Best node to expand with g(n) = 1 and h(n) = 4

1 2 3
4 8 5
7 6

expanding this node...
Best node to expand with g(n) = 2 and h(n) = 3

1 2 3
4 8 5
7   6

expanding this node...
Best node to expand with g(n) = 3 and h(n) = 2

1 2 3
4   5
7 8 6

expanding this node...
Best node to expand with g(n) = 4 and h(n) = 1

1 2 3
4 5
7 8 6

expanding this node...
Found solution:

1 2 3
4 5 6
7 8

Nodes expanded: 5
Max nodes enqueued: 11
Depth of the goal node: 5
```

Trace of Manhattan Distance A* on the puzzle 1,2,3,4,8,0,7,6,5

8puzzle.cpp
URL: https://github.com/justinhluo/CS170

```
//Justin Luo - 862013749
#include <iostream>
#include <vector>
#include <queue>
#include <bits/stdc++.h>
using namespace std;
static const vector <int> goalmatrix = {1,2,3,4,5,6,7,8,0};

struct node {
        vector <int> state;
        int g_n;
        int h_n;
        int f_n;
        node(vector <int> v, int g_n, int h_n){ //constructor
                this->state = v;
                this->g_n = g_n;
                this->h_n = h_n;
                f_n = g_n + h_n;
        }
        void print() { //print the vector as a 3x3 matrix, omitting the 0
                cout << endl;
                for(int i = 0; i < state.size(); i++){
                        if(state.at(i) == 0) {
                                cout << "  ";
                        }else{
                                cout << state.at(i) << " ";
                        }
                        if ((i+1) % 3 == 0){
                                cout << endl;
                        }
                }
                cout << endl;
                return;
        }
        int getg_n() {
                return g_n;
        }
        int geth_n() {
                return h_n;
        }
        int getf_n() {
                return f_n;
        }
        vector<int> getstate() {
                return state;
        }
};

int misplaced (vector <int> v) { //return h(n) using misplaced tile
        int heuristic = 0;
        for(int i = 0; i < v.size(); ++i) {
                if (v.at(i) != goalmatrix.at(i) && v.at(i) != 0){ //if tile is misplaced, excluding the blank
                        heuristic++;
                }
        }
        return heuristic;
}
```

```
int mhat_row (int n) { //helper function, returns correct row for a given number
        if(n <= 3) {
                return 1;
        }else if (n >= 4 && n <= 6) {
                return 2;
        }else{
                return 3;
        }
}

int mhat_col (int n) { //helper function, returns correct column for a given number
        if((n + 2) % 3 == 0){
                return 1;
        }else if((n + 1) % 3 == 0){
                return 2;
        }else{
                return 3;
        }
}

int manhattan (vector <int> v) { //return h(n) using manhattan distance
        int heuristic = 0;
        int curr_row = 1;
        int curr_col = 1;
        for (int i = 0; i < v.size(); ++i){
                if(i != 0 && i % 3 == 0) { //set every thrid iteration as new row and reset column, but don't do this for first iteration
                        curr_row ++;
                        curr_col = 1;
                }
                if(v.at(i) != goalmatrix.at(i) && v.at(i) != 0) { //don't count blank as misplaced
                        heuristic += (abs(mhat_row(v.at(i)) - curr_row) + abs(mhat_col(v.at(i)) - curr_col)); //formula for
manhattan distance, consulted https://www.geeksforgeeks.org/sum-manhattan-distances-pairs-points/
                }
                curr_col++;
        }
        return heuristic;
}

bool operator<(const node& n1, const node& n2){ //overload to get the min(f_n) on top of priority queue, consulted
https://www.geeksforgeeks.org/stl-priority-queue-for-structure-or-class/
                return n1.f_n > n2.f_n;
}

vector<node> expandNode(node n, int qfunc){//returns a vector containing all children of a node. get h(n) from helper functions.
increment g(n).
        vector<node> children;
        int temp;
        vector<int> temp_state;
        int newg_n = n.getg_n() + 1;
        int newh_n;
        vector<int> curr_state = n.getstate();
        int index = -1;
        for(int i = 0; i < curr_state.size(); ++i) { //find index of 0
                if(curr_state.at(i) == 0) {
                        index = i;
                        break;
                }
        }
        //each node has a max of 4 children(if the 0 is in the middle) and a min of 2 children(if the 0 is in a corner)
        if((index % 3) != 0){//if 0 is not in the first column (0,3,6) then we can move the 0 left
```

```
                temp_state = curr_state;
                temp = curr_state.at(index-1);
                temp_state.at(index-1) = 0;
                temp_state.at(index) = temp;
                if(qfunc == 1) {
                        newh_n = 0; //hardcode h(n) to 0 for uniform cost search
                }else if(qfunc == 2) {
                        newh_n = misplaced(temp_state);
                }else if(qfunc == 3) {
                        newh_n = manhattan(temp_state);
                }
                node n(temp_state, newg_n, newh_n);
                children.push_back(n);
        }
        if(((index + 1) % 3) != 0){//if 0 is not in the last column (2,5,8) then we can move the 0 right
                temp_state = curr_state;
                temp = curr_state.at(index+1);
                temp_state.at(index+1) = 0;
                temp_state.at(index) = temp;
                if(qfunc == 1) {
                        newh_n = 0;
                }else if(qfunc == 2) {
                        newh_n = misplaced(temp_state);
                }else if(qfunc == 3) {
                        newh_n = manhattan(temp_state);
                }
                node n(temp_state, newg_n, newh_n);
                children.push_back(n);
        }
        if(index > 2){//if 0 is not in the first row (0,1,2) then we can move the 0 up
                temp_state = curr_state;
                temp = curr_state.at(index-3);
                temp_state.at(index-3) = 0;
                temp_state.at(index) = temp;
                if(qfunc == 1) {
                        newh_n = 0;
                }else if(qfunc == 2) {
                        newh_n = misplaced(temp_state);
                }else if(qfunc == 3) {
                        newh_n = manhattan(temp_state);
                }
                node n(temp_state, newg_n, newh_n);
                children.push_back(n);
        }
        if(index < 6){//if 0 is not in the last row (6,7,8) then we can move the 0 down
                temp_state = curr_state;
                temp = curr_state.at(index+3);
                temp_state.at(index+3) = 0;
                temp_state.at(index) = temp;
                if(qfunc == 1) {
                        newh_n = 0;
                }else if(qfunc == 2) {
                        newh_n = misplaced(temp_state);
                }else if(qfunc == 3) {
                        newh_n = manhattan(temp_state);
                }
                node n(temp_state, newg_n, newh_n);
                children.push_back(n);
        }
        return children;
}
```

```
void generalsearch(vector<int> v, int qfunc){
	int max_queued = 1; //initial state always get enqueued
	int num_expanded = 0;
	int initial_heuristic;
	node temp_node(v, 0, 0);
	vector <node> temp_vec;
	if(qfunc == 1) {
		initial_heuristic = 0;
	}else if(qfunc == 2) {
		initial_heuristic = misplaced(v);
	}else if(qfunc == 3) {
		initial_heuristic = manhattan(v);
	}
	node initial_node(v,0,initial_heuristic); //create root node
	priority_queue<node> q;
	q.push(initial_node);
	while(1) {
		if(q.empty()){
			cout << "No solution" << endl;
			return;
		}
		temp_node = q.top();
		q.pop();
		if(temp_node.getstate() == goalmatrix) { //if popped node has goal state, then return, else expand it
			cout << "Found solution:" << endl;
			temp_node.print();
			cout << "Nodes expanded: " << num_expanded << endl;
			cout << "Max nodes enqueued: " << max_queued << endl;
			cout << "Depth of the goal node: " << temp_node.getg_n() << endl;
			return;
		}
		cout << "Best node to expand with g(n) = " << temp_node.getg_n() << " and h(n) = " << temp_node.geth_n() <<
endl;
		temp_node.print();
		cout << "expanding this node..." << endl;
		num_expanded++;
		temp_vec = expandNode(temp_node, qfunc);
		for(int i = 0; i < temp_vec.size(); ++i) { //push children of expanded node into queue
			q.push(temp_vec.at(i));
		}
		if(q.size() > max_queued) { //update max_queue size
			max_queued = q.size();
		}
	}
}

int main() {
	int input;
	int input2;
	vector <int> matrix = {1,2,3,4,8,0,7,6,5};  //default puzzle, gets overwritten if user chooses to use custom puzzle
	vector <int> test = {0,1,2,3,4,5,6,7,8};
	vector <int> usermatrix;
	cout << "8 puzzle solver by Justin Luo" << endl;
	cout << "Press 1 to use default puzzle, 2 to use custom puzzle: ";
	cin >> input;x
	if (input == 1) {
		cout << "Using default puzzle" << endl;
	} else if (input == 2) {
		int num;
		cout << "Separate each number with space or enter. Use only 1-8 one time each. Use 0 to represnt the blank 
space" << endl;
		for (int i = 0; i < goalmatrix.size(); ++i){
			cin >> num;
			usermatrix.push_back(num);
		}
		vector <int> temp = usermatrix;
		sort(temp.begin(), temp.end());
		if(temp != test) { // sort the vecotr user inputted and compare with test vector
			cout << "invalid puzzle! try again" << endl;
			exit(0);
		}
		matrix = usermatrix;
	}else {
		cout << "Invalid input! try again" << endl;
		exit(0);
	}
	cin.ignore(numeric_limits<streamsize>::max(),'\n'); //clear input buffer in case user entered more numbers, consulted 
https://www.geeksforgeeks.org/clearing-the-input-buffer-in-cc/
	cout << "Select algorithm: " << endl << "(1 for uniform cost, 2 for A* with misplaced tile heuristic, 3 for A* with manhattan 
distance heuristic): ";
	cin >> input2;
	if(input2 == 1) {
		generalsearch(matrix, 1);
	}else if(input2 == 2){
		generalsearch(matrix, 2);
	}else if(input2 == 3){
		generalsearch(matrix, 3);
	}else {
		cout << "Invalid input! try again" << endl;
		exit(0);
	}
	return 0;
}
```