Assignment 1 CS170: Introduction to Artificial Intelligence, Dr. Eamonn Keogh

Kevin Carabajal

SID 862090928

Email: kcara003@ucr.edu

Date: Nov-3-2021

I consulted the following sources:

Google (Mostly syntax errors. Did not use anyone else’s code.)

C++ standard Library

Blind Search and Heuristic Slides

All Code is original, with the exception of C++ libraries to make certain implementations earlier. Like a priority queue.

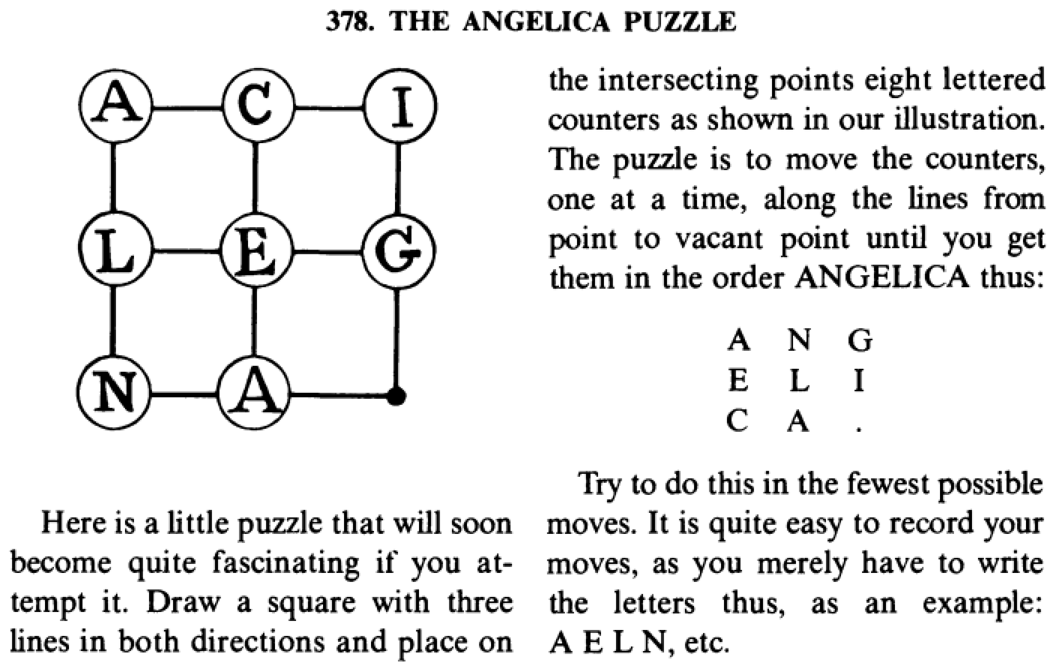
Outline of Report:

- Cover Page
- My report
- Sample Trace on easy problem
- Sample Trace on hard problem
- My code. Here is a link to my online repo of the code.
  https://github.com/Ghost-Kozak/CS170

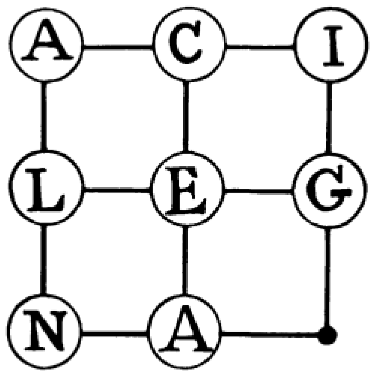
CS170: Assignment 1: The Angelica Puzzle

Kevin Carabajal, SID 862090928, Nov-3-2021

Introduction



378. THE ANGELICA PUZZLE



Here is a little puzzle that will soon become quite fascinating if you attempt it. Draw a square with three lines in both directions and place on the intersecting points eight lettered counters as shown in our illustration. The puzzle is to move the counters, one at a time, along the lines from point to vacant point until you get them in the order ANGELICA thus:

A N G
E L I
C A .

Try to do this in the fewest possible moves. It is quite easy to record your moves, as you merely have to write the letters thus, as an example: A E L N, etc.

This puzzle is very similar to the traditional sliding 8 puzzle. However, there is a key difference. Instead of being ordered read left to right 1 through 8. It is read in the same manner but must end up spelling the name Angelica. At first glance it seems that there is now difference logically as each letter can pair with a number. But this logic is exactly how to see what makes it different.It is because the name Angelica has two of the letter A. So it is similar to the sliding 8 puzzle, with the caveat that 1 and 8 are interchangeable in their placement.

This idea can be further extended with other words. But for these purposes, we will stick with Angelica. (Although my code will work for any 8 letter word with a minor modification to the goal state.) This project can perform Uniform Cost Search, Misplaced Tile Heuristic, and Manhattan Distance Heuristic. Which one is used is selected by the user at runtime. With the end result being a list of the tables, in the order of the moves you must take to get the solution read from top to bottom.

Comparison of Algorithms

Uniform Cost Search

Simplist to implement, but also the slowest of the three. Better known as Breadth First Search (BFS). No dynamic weights so all Nodes are equal in their cost.

The Misplaced Tile Heuristic

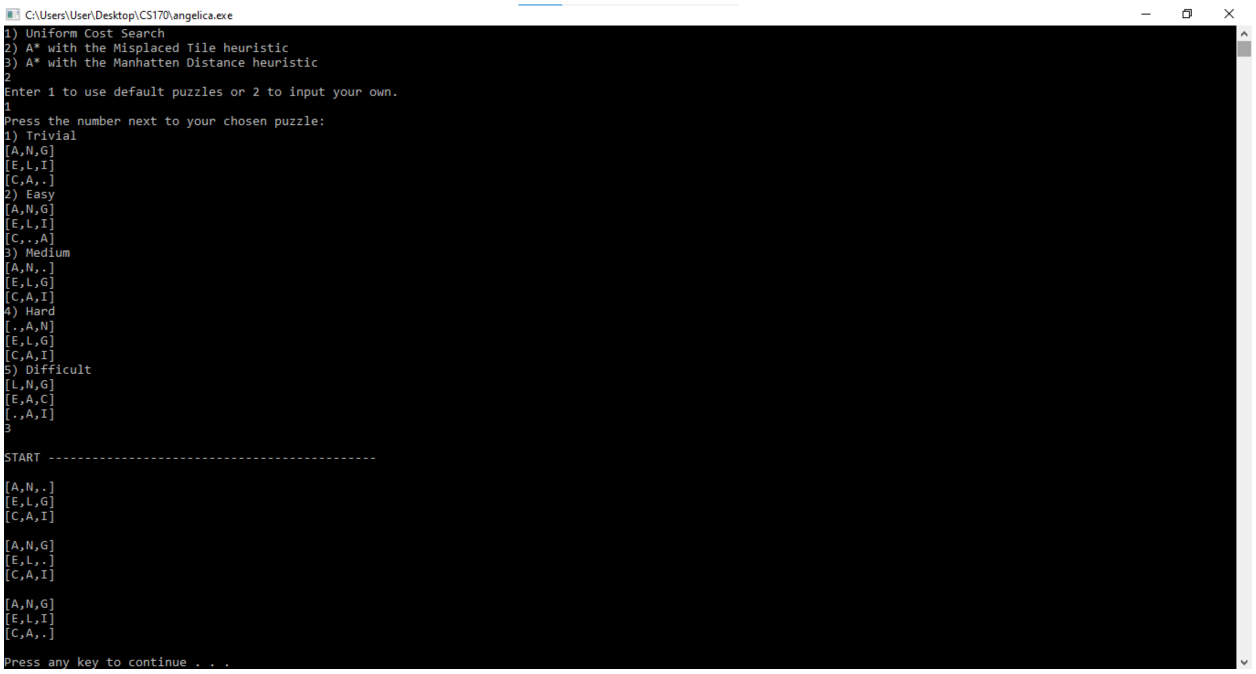
Still simple to implement, but increases performance significantly. Puts a lower cost on boards that are already closer to the solution. Rather vague in what is “close” but for its ease of implementation, still fine to use. Solely counts the number of tiles that are not in the correct position. Regardless of how far they are from their ideal position.

The Manhattan Distance Heuristic

Very similar to the Misplaced Tile heuristic, however this does try to add cost depending on how far the tile is from their desired position. This distance complicates the calculation quite a bit. But it drastically reduces the number of nodes traversed before finding a solution. So of the three algorithms mentioned, this performs the best.

Comparisons and Conclusions

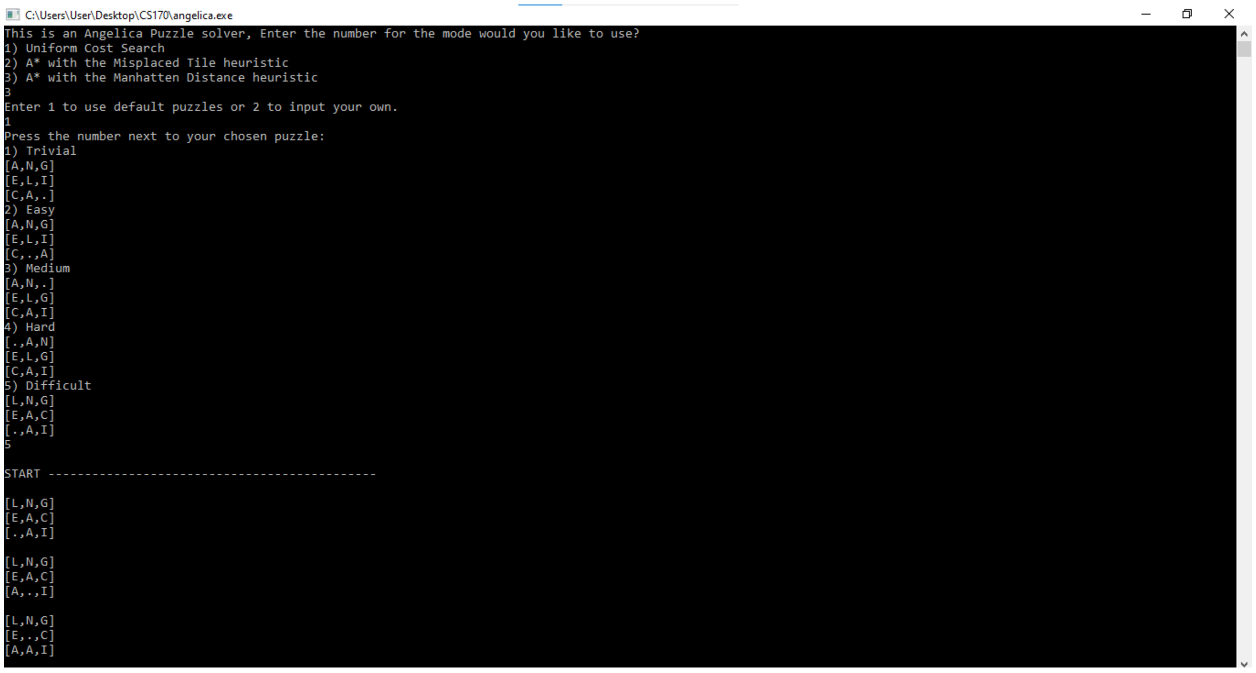
When the puzzle is simple or easy, the results between the different algorithms are miniscule. It is on the harder and longer puzzles that the algorithms start to show the difference in their performance. Of the three they performed in order of best to worst (comparatively) is Manhattan Distance Heuristic, Misplaced Tiles Heuristic, and Uniform Cost (BFS).
BFS being the longest in time and space complexity. While the other two are still within the same level of time and space complexity, they regularly perform better to lower their average significantly in comparison.



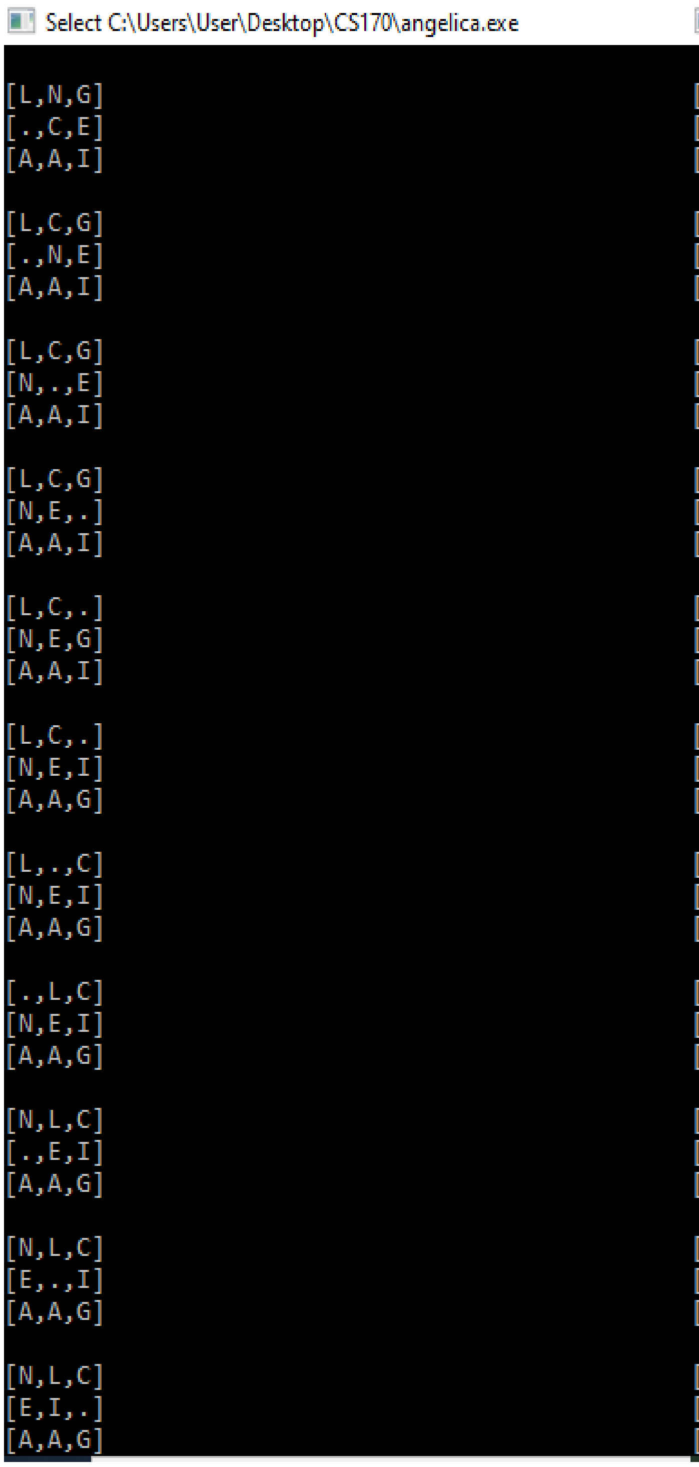
```
C:\Users\User\Desktop\CS170\angelica.exe
1) Uniform Cost Search
2) A* with the Misplaced Tile heuristic
3) A* with the Manhatten Distance heuristic
2
Enter 1 to use default puzzles or 2 to input your own.
1
Press the number next to your chosen puzzle:
1) Trivial
[A,N,G]
[E,L,I]
[C,A,.]
2) Easy
[A,N,G]
[E,L,I]
[C,.,A]
3) Medium
[A,N,.]
[E,L,G]
[C,A,I]
4) Hard
[.,A,N]
[E,L,G]
[C,A,I]
5) Difficult
[L,N,G]
[E,A,C]
[.,A,I]
3

START --------------------------------------------

[A,N,.]
[E,L,G]
[C,A,I]

[A,N,G]
[E,L,.]
[C,A,I]

[A,N,G]
[E,L,I]
[C,A,.]

Press any key to continue . . .
```

Easy Demo. Few steps

Hard Demo

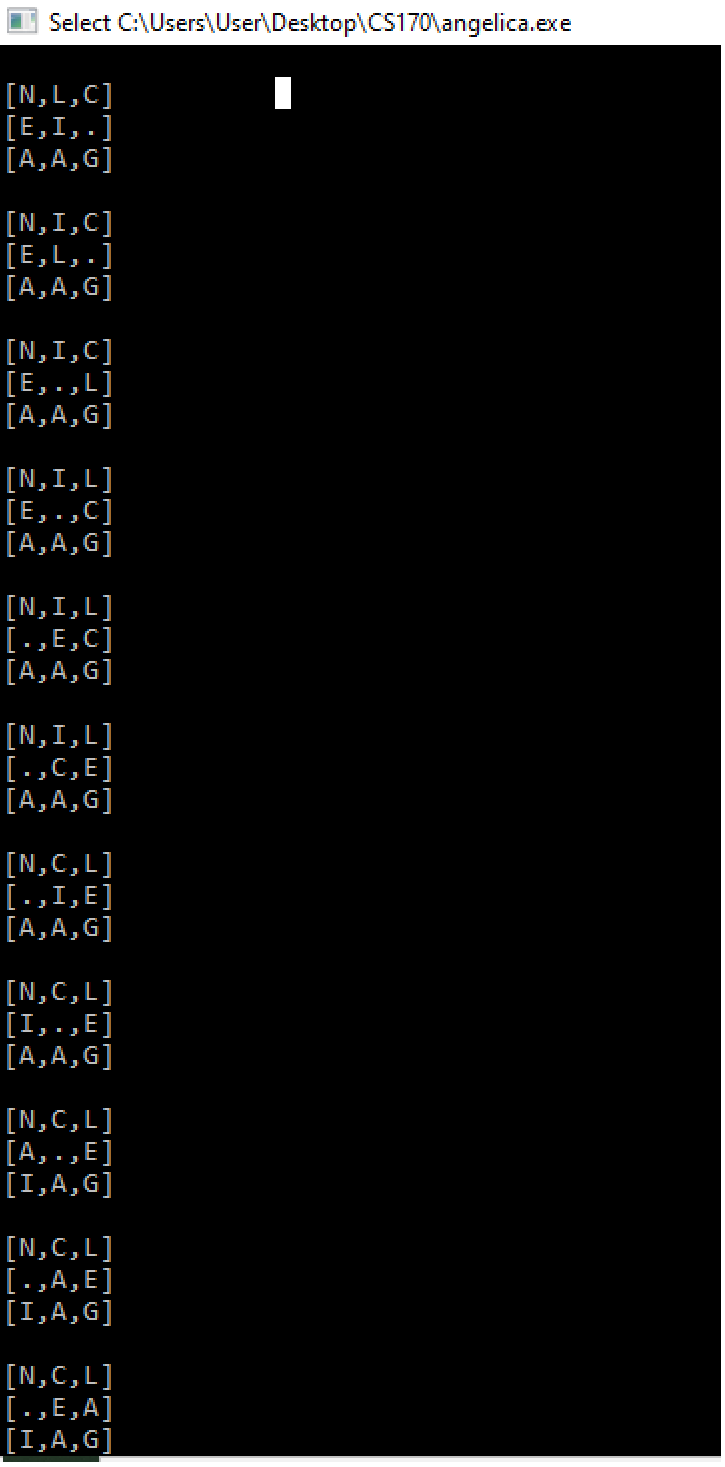


```
C:\Users\User\Desktop\CS170\angelica.exe
This is an Angelica Puzzle solver, Enter the number for the mode would you like to use?
1) Uniform Cost Search
2) A* with the Misplaced Tile heuristic
3) A* with the Manhatten Distance heuristic
3
Enter 1 to use default puzzles or 2 to input your own.
1
Press the number next to your chosen puzzle:
1) Trivial
[A,N,G]
[E,L,I]
[C,A,.]
2) Easy
[A,N,G]
[E,L,I]
[C,.,A]
3) Medium
[A,N,.]
[E,L,G]
[C,A,I]
4) Hard
[.,A,N]
[E,L,G]
[C,A,I]
5) Difficult
[L,N,G]
[E,A,C]
[.,A,I]
5

START --------------------------------------------

[L,N,G]
[E,A,C]
[.,A,I]

[L,N,G]
[E,A,C]
[A,.,I]

[L,N,G]
[E,.,C]
[A,A,I]
```



Select C:\Users\User\Desktop\CS170\angelica.exe

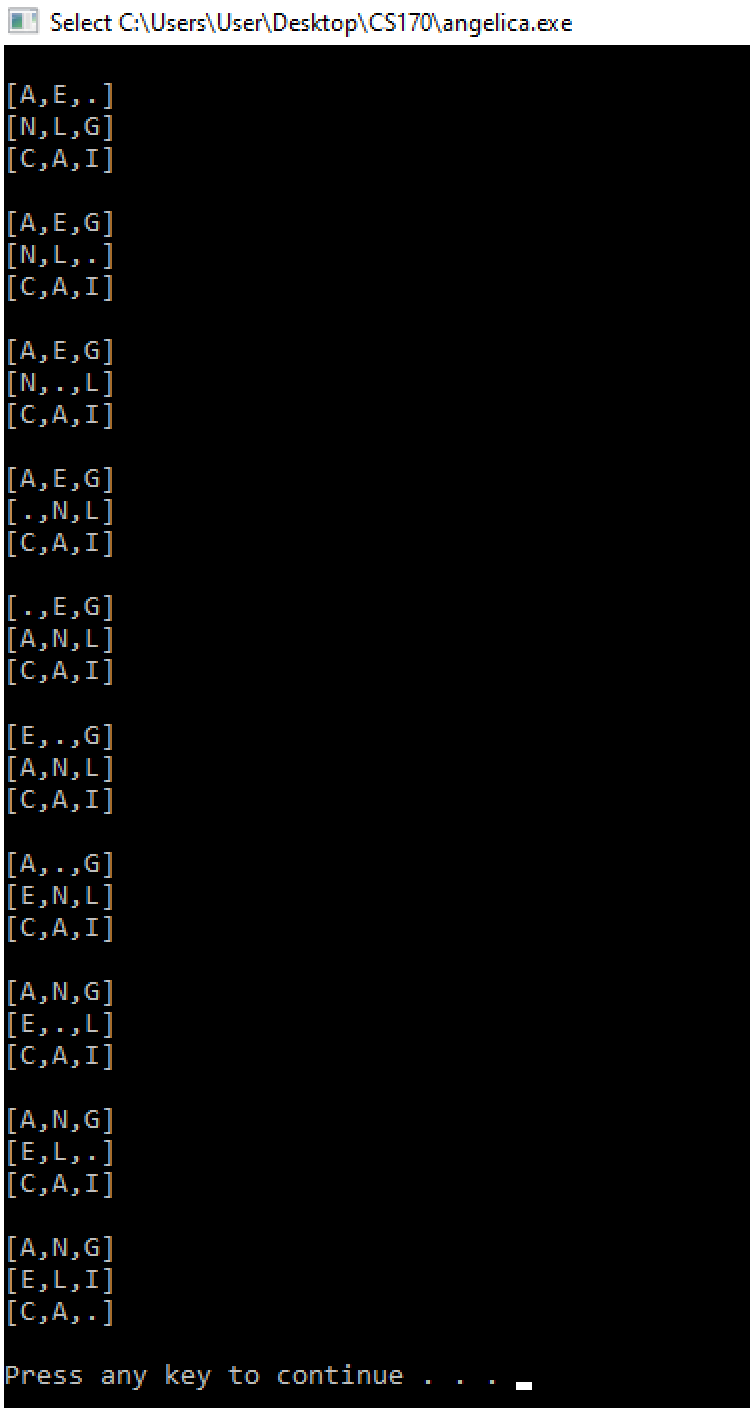
```
[L,N,G]
[.,C,E]
[A,A,I]

[L,C,G]
[.,N,E]
[A,A,I]

[L,C,G]
[N,.,E]
[A,A,I]

[L,C,G]
[N,E,.]
[A,A,I]

[L,C,.]
[N,E,G]
[A,A,I]

[L,C,.]
[N,E,I]
[A,A,G]

[L,.,C]
[N,E,I]
[A,A,G]

[.,L,C]
[N,E,I]
[A,A,G]

[N,L,C]
[.,E,I]
[A,A,G]

[N,L,C]
[E,.,I]
[A,A,G]

[N,L,C]
[E,I,.]
[A,A,G]
```



Select C:\Users\User\Desktop\CS170\angelica.exe

```
[N,L,C]
[E,I,.]
[A,A,G]

[N,I,C]
[E,L,.]
[A,A,G]

[N,I,C]
[E,.,L]
[A,A,G]

[N,I,L]
[E,.,C]
[A,A,G]

[N,I,L]
[.,E,C]
[A,A,G]

[N,I,L]
[.,C,E]
[A,A,G]

[N,C,L]
[.,I,E]
[A,A,G]

[N,C,L]
[I,.,E]
[A,A,G]

[N,C,L]
[A,.,E]
[I,A,G]

[N,C,L]
[.,A,E]
[I,A,G]

[N,C,L]
[.,E,A]
[I,A,G]
```

Skipping what would be another 7 pictures into the final one …………………………



Select C:\Users\User\Desktop\CS170\angelica.exe

```
[A,E,.]
[N,L,G]
[C,A,I]

[A,E,G]
[N,L,.]
[C,A,I]

[A,E,G]
[N,.,L]
[C,A,I]

[A,E,G]
[.,N,L]
[C,A,I]

[.,E,G]
[A,N,L]
[C,A,I]

[E,.,G]
[A,N,L]
[C,A,I]

[A,.,G]
[E,N,L]
[C,A,I]

[A,N,G]
[E,.,L]
[C,A,I]

[A,N,G]
[E,L,.]
[C,A,I]

[A,N,G]
[E,L,I]
[C,A,.]

Press any key to continue . . .
```

URL to my code : https://github.com/Ghost-Kozak/CS170

```
#include <iostream>

#include <list>

#include <string>

#include <vector>

#include <utility>

#include <algorithm>

#include <queue>

using namespace std;


// Declaring Node class

struct Node

{

    string key;

    vector<Node *>child;

    Node * parent;

    int shifter;

    int cost;


    bool operator==(const Node& p) const

    {

        return key == p.key;

    }

};


// Node constructor

Node *newNode(string key, Node * parent)
```

```
{
    Node *temp = new Node;
    temp->key = key;
    temp->parent = parent;
    return temp;
};

// Comparator for priority queue of Nodes
struct LessThanByCost
{
    bool operator()(const Node* lhs, const Node* rhs) const
    {
        return lhs->cost <= rhs->cost;
    }
};

// Global defaults
string goalState = "ANGELICA.";
string trivialState  = "ANGELICA.";
string easyState = "ANGELIC.A";
string mediumState = "AN.ELGCAI";
string hardState = ".ANELGCAI";
string difficultState = "LNGEAC.AI";


// Takes a string and prints in the shape of actual game in terminal
void printPuzzle(string puzzle) {
    for (int i = 0; i < puzzle.size(); ++i) {
        if (i % 3 == 0) {
            cout << "[" << puzzle[i] << ",";
        }
        else if (i % 3 == 1) {
            cout << puzzle[i] << ",";
        }
        else {
            cout << puzzle[i] << "]\n";
        }
    }
}

// Read choices from user, and prepare the code for it.
vector<int> printMenu(string &custom) {
    vector<int> choices;
    int input = 0;
    // Intro and mode selection
    cout << "This is an Angelica Puzzle solver, Enter the number for the
mode would you like to use? \n";
    cout << "1) Uniform Cost Search \n";
    cout << "2) A* with the Misplaced Tile heuristic \n";
    cout << "3) A* with the Manhatten Distance heuristic \n";
    cin >> input;
    choices.push_back(input);

    // Default vs custom puzzle
    cout << "Enter 1 to use default puzzles or 2 to input your own.\n";
    cin >> input;
    choices.push_back(input);
    if (input == 2) {
        cout << "Please enter a string of 9 characters, with a valid
angelica solution please.\n";
        cin >> input;
        custom = input;
    }
    else {
        cout << "Press the number next to your chosen puzzle:" << "\n";
        cout << "1) Trivial" << "\n";
        printPuzzle(trivialState);
        cout << "2) Easy" << "\n";
        printPuzzle(easyState);
        cout << "3) Medium" << "\n";
        printPuzzle(mediumState);
        cout << "4) Hard" << "\n";
        printPuzzle(hardState);
        cout << "5) Difficult" << "\n";
        printPuzzle(difficultState);
        cin >> input;
        choices.push_back(input);
    }
    return choices;
}
```

```
void printVector(vector<string> output) {

    cout << "[";

    for (int i = 0; i < output.size() - 1; ++i) {

        cout << output[i] << ",";

    }

    cout << output[output.size() - 1] << "]" << "\n";

}


// Position of "." in string

int findShifter(string puzzle) {

    for (int i = 0; i < puzzle.size(); ++i) {

        if (puzzle[i] == '.') {

            return i;

        }

    }

    return -1;

}


// Distance Formula

int distance(int index, int goal) {

    int num = 0;


    if ((index / 3) != (goal / 3)) {

        // If true, they are on the different y axis.

        if (goal >= 5) {

            while (index < 5) {

                index += 3;
                num++;
            }
        }
        else if(goal <= 2) {
            while (index > 2) {
                index -= 3;
                num++;
            }
        }

    }
    while (index < goal) {
        index++;
        num++;
    }
    while (index > goal) {
        index--;
        num++;
    }
    return num;
}

// Mode is what cost function, calclate cost and rank nodes accordingly
int getCost(string puzzle, int mode) {
    switch(mode) {
        case 1: {
            // Uniform Search Cost (BFS)
            // expand if possible, left up down right
            return 1;
            break;
        }
        case 2: {
            // A* with the Misplaced Tile heuristic
            // Calculate lowest cost, then expand in that order.
            int count = 0;
            for (int i = 0; i < goalState.size(); ++i) {
                if ( (puzzle[i] != goalState[i]) && (puzzle[i] != '.') ) {
                    count++;
                }
            }
            return count;
            break;
        }
        case 3: {
            // A* with the Manhatten Distance heuristic
            int total = 0;
            for (int i = 0; i < goalState.size(); ++i) {
                if ( (puzzle[i] != goalState[i]) && (puzzle[i] != '.') ) {
                    switch (puzzle[i]) {
                        case 'A': {
                            int dist1 = distance(i, 0);
                            int dist2 = distance(i, 7);
                            int min = dist1;
                            if (dist2 < dist1) {
```

```
        min = dist2;
    }
    total += min;
    break;
}
case 'N': {
    total += distance(i, 1);
    break;
}
case 'G': {
    total += distance(i, 2);
    break;
}
case 'E': {
    total += distance(i, 3);
    break;
}
case 'L': {
    total += distance(i, 4);
    break;
}
case 'I': {
    total += distance(i, 5);
    break;
}
case 'C': {
    total += distance(i, 6);
```

```
                            break;
                        }
                    }
                }
            }
            return total;
            break;

        }

    }
    return -1;
}

Node* generalSearch(string problem, int mode) {
    // Logging the nodes I have found to prevent searching a puzzle
already queued
    vector<string> catalog;

    // Initialize root node
    Node *root = newNode(problem, NULL);
    root->shifter = findShifter(problem);
    root->cost = getCost(root->key, mode);

    // Prepare queue
    priority_queue<Node* , vector<Node*>, LessThanByCost> checklist;
    checklist.push(root);
```

```
catalog.push_back(root->key);
while(!checklist.empty()) {
    if (checklist.top()->key == goalState) {
        return checklist.top();
    }
    else {
        int index = checklist.top()->shifter;

        // left, swap one left (if index % 3 = 0 then can't do it)
        if (index % 3 != 0) {
            string movement = checklist.top()->key;
            char temp = movement[index];
            movement[index] = movement[index - 1];
            movement[index - 1] = temp;
            bool found = false;
            for (int i = 0; i < catalog.size(); ++i) {
                if (catalog[i] == movement) {
                    found = true;
                }
            }
            if (!found) {
                Node *left = newNode(movement, checklist.top());
                checklist.top()->child.push_back(left);
                left->cost = getCost(left->key, mode);
                left->shifter = findShifter(left->key);
                catalog.push_back(movement);
                checklist.push(left);
            }
        }
        // right, swap one right (if index % 3 = 2 then can't do it)
        if (index % 3 != 2) {
            string movement = checklist.top()->key;
            char temp = movement[index];
            movement[index] = movement[index + 1];
            movement[index + 1] = temp;
            bool found = false;
            for (int i = 0; i < catalog.size(); ++i) {
                if (catalog[i] == movement) {
                    found = true;
                }
            }
            if (!found) {
                Node *right = newNode(movement, checklist.top());
                checklist.top()->child.push_back(right);
                right->cost = getCost(right->key, mode);
                right->shifter = findShifter(right->key);
                catalog.push_back(movement);
                checklist.push(right);
            }
        }
        // up, swap 3 over left (if index < 3 then can't)
        if (index >= 3) {
            string movement = checklist.top()->key;
            char temp = movement[index];
```

```
        movement[index] = movement[index - 3];
        movement[index - 3] = temp;
        bool found = false;
        for (int i = 0; i < catalog.size(); ++i) {
            if (catalog[i] == movement) {
                found = true;
            }
        }
        if (!found) {
            Node *up = newNode(movement, checklist.top());
            checklist.top()->child.push_back(up);
            up->cost = getCost(up->key, mode);
            up->shifter = findShifter(up->key);
            catalog.push_back(movement);
            checklist.push(up);
        }
    }
    // down, swap 3 over right (if index > 5 then can't)
    if (index < 6) {
        string movement = checklist.top()->key;
        char temp = movement[index];
        movement[index] = movement[index + 3];
        movement[index + 3] = temp;
        bool found = false;
        for (int i = 0; i < catalog.size(); ++i) {
            if (catalog[i] == movement) {
                found = true;
```

```
                    }
                }
                if (!found) {
                    Node *down = newNode(movement, checklist.top());
                    checklist.top()->child.push_back(down);
                    down->cost = getCost(down->key, mode);
                    down->shifter = findShifter(down->key);
                    catalog.push_back(movement);
                    checklist.push(down);
                }
            }

            // Queue has been extended
            checklist.pop();
        }
    }
    return NULL;
}

int main() {

    string userInput = "ANGELICA.";

    vector<int> choices;
    // Input menu
    choices = printMenu(userInput);
```

```
if (choices.size() == 3) {
    switch(choices[2]) {
    case 1:
        userInput = trivialState;
        break;
    case 2:
        userInput = easyState;
        break;
    case 3:
        userInput = mediumState;
        break;
    case 4:
        userInput = hardState;
        break;
    case 5:
        userInput = difficultState;
        break;
    default:
        userInput = trivialState;
    }
}

Node * temp = generalSearch(userInput, choices[0]);


cout << "\nSTART ---------------------------------------------\n\n";
```

```
    // prints the solution and progrssion of the winning solution from top
to bottom in that order.
    // States what steps to take for the winning game


    vector<string> solution;
    while (temp != NULL) {
        solution.push_back(temp->key);
        temp = temp->parent;
    }
    for (int i = solution.size() - 1; i >= 0; --i) {
        printPuzzle(solution[i]);
        cout << "\n";
    }



    system("pause");


    return 0;
}
```