Trung Lam
SID: 861270734
Date: 13 May, 2022

CS205: Artificial Intelligence - Project 1: Adjusting the Counters

In completing the project, the following references were consulted:

- Lecture slides relating to the general search algorithm
- C++ Documentation: https://www.cplusplus.com/reference/
- Puzzle information: http://jnsilva.ludicum.org/HMR13_14/536.pdf

All important code is original. Unimportant subroutines that are not original are:

- chrono library: assist as a seed for randomization of an initial state
- vector library: assist as the main data structure used in the project
- cmath library: assist with the printing of the states

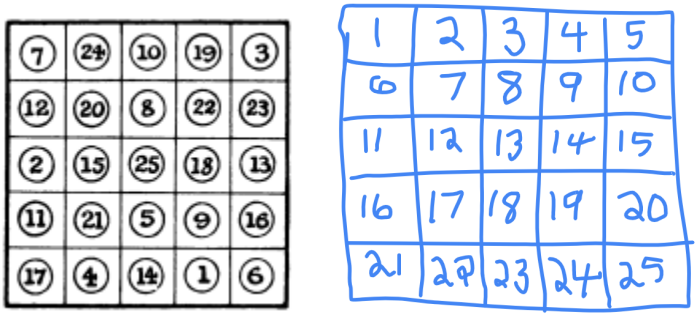
Outline:

For access to the code itself: https://github.com/Krazriel/AdjustTheCounters

Project 1 - Adjusting the Counters

Introduction:

Adjusting the counters is a simple puzzle where the objective is given a random state of a N x N board, continuously swapping counters in a few moves possible until the board is in order. For example, the left image is an initial state, and the right is the goal state:



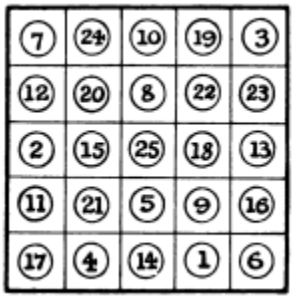
The purpose of this project is to implement a search algorithm using three algorithms, uniform cost search and A* algorithm with two different heuristics: Misplaced Tile Heuristics and Manhattan Distance Heuristic, and to analyze the performance of the search using each of these algorithms.

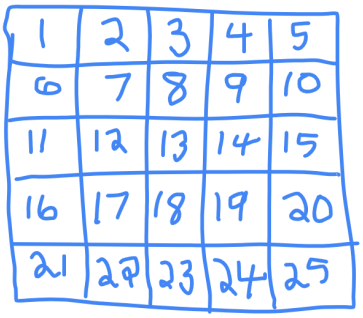
Uniform Cost Search:

Uniform cost search is a search algorithm that essentially expands a node that has the cheapest cost. For this project, uniform cost search is essentially just an algorithm that expands a node, and checks every child of that node, and repeats the process for every child since every node has the same cost. This search algorithm is equivalent to A* with the heuristic being hardcoded to 0.

Misplaced Tile Heuristics:

A* with a Misplaced Tile Heuristics is an algorithm that uses a Misplaced Tile Heuristics that relies on the positioning of the counters in the puzzle. For example in the following images:

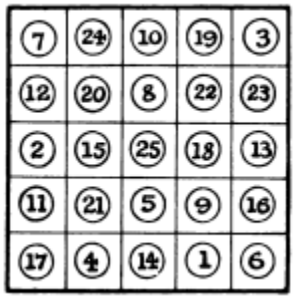




The heuristic will start out with a score of 0. Counter 7 is misplaced since it is supposed to be 1, thus the score will increment by 1. Counter 24 is misplaced since it is supposed to be 2, the score will increment by 1 and so on. For this initial state, the heuristic will output a score of 24. With A*, the score of the heuristic is added to whatever depth the state of which is currently being analyzed and instead of expanding every single child of a node, A* now expands based on the cheapest cost of the score of heuristic + the depth of a node.

Manhattan Distance Heuristic:

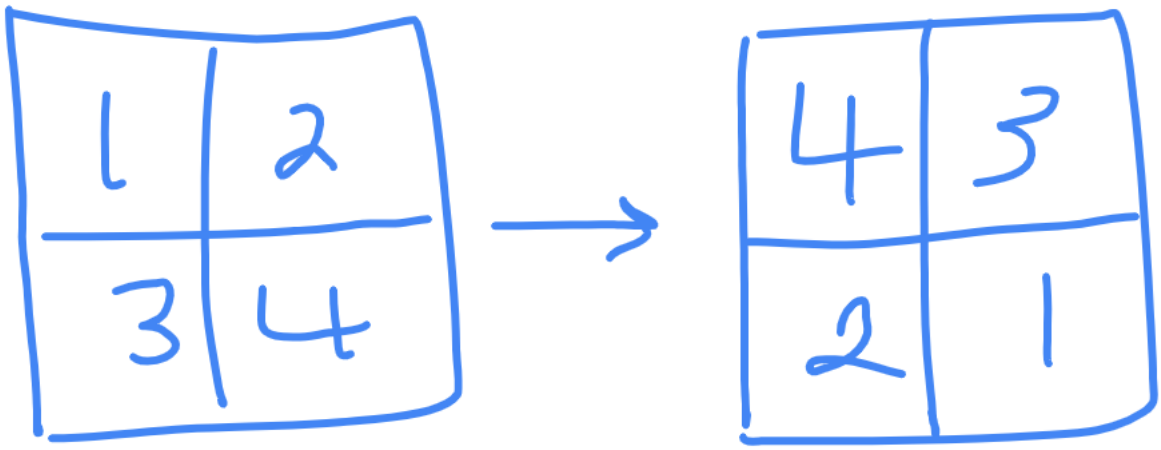
A* with a Manhattan Distance Heuristics is an algorithm that relies on the positioning and the distance of the counters in the puzzle. For example, using the below initial state:



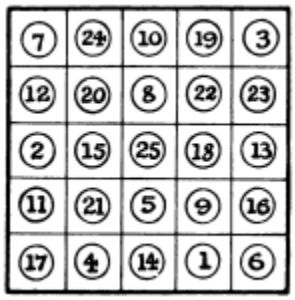
The Manhattan Distance Heuristic score starts out as 0. The counter 7 is 3 units away from its goal position, so the Manhattan Distance Heuristic will increment the score by 3. The counter 24 is 6 units away from its goal position, so the score is incremented by 6, resulting in 9, and so on. Similar to A* with Misplaced Tile Heuristic, the node expanded will depend on the score of the heuristic + the depth of the node.

Comparison of Algorithms:

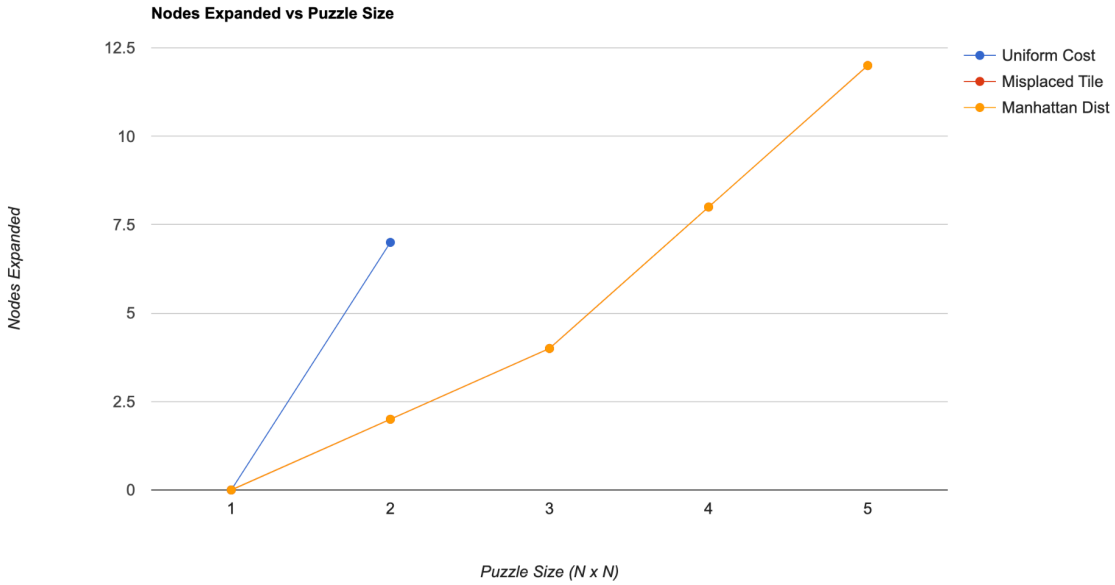
In the process of comparing the performance of the search algorithms, the initial state that will be used to compare the algorithms as the control will be the goal state but in reverse. For example:



Also, the default puzzle will be used to compare the algorithms:

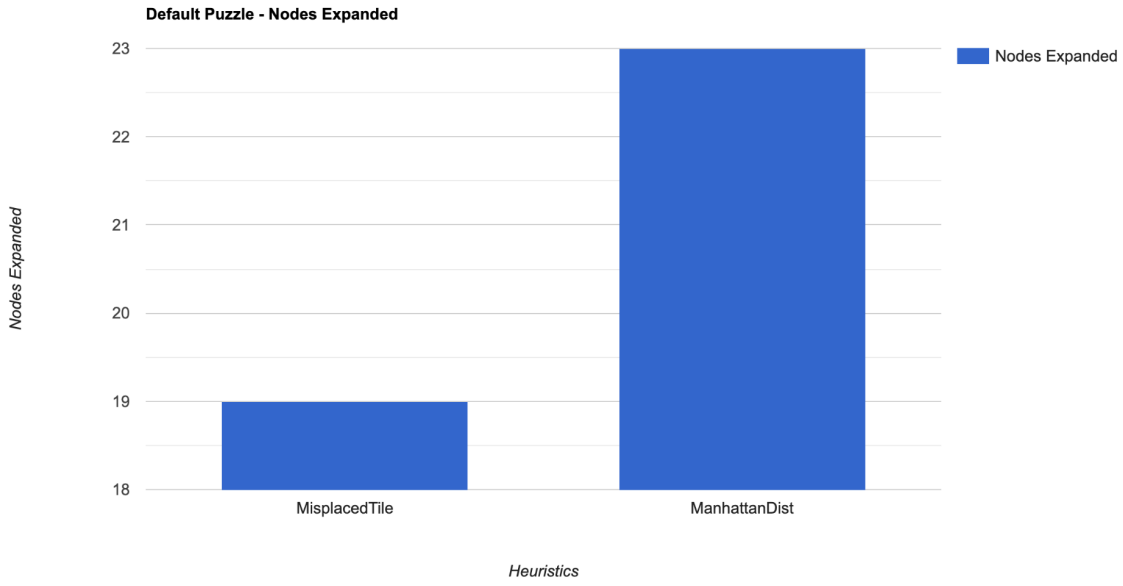


Nodes Expanded vs Size of Puzzle



As expected, uniform cost search does perform the worst compared to the heuristics. Due to time constraints, it is safe to say since the number of nodes exploded after a puzzle of N = 1, that the number of nodes expanded for uniform cost search is expected to continuously explode as the puzzle size increases. As such, moving forward in future analysis of areas such as queue size, it is safe to assume that the queue is extremely large for uniform cost search and thus will not be needed to analyze. On the other hand, misplaced tile and manhattan distance performs exactly the same.

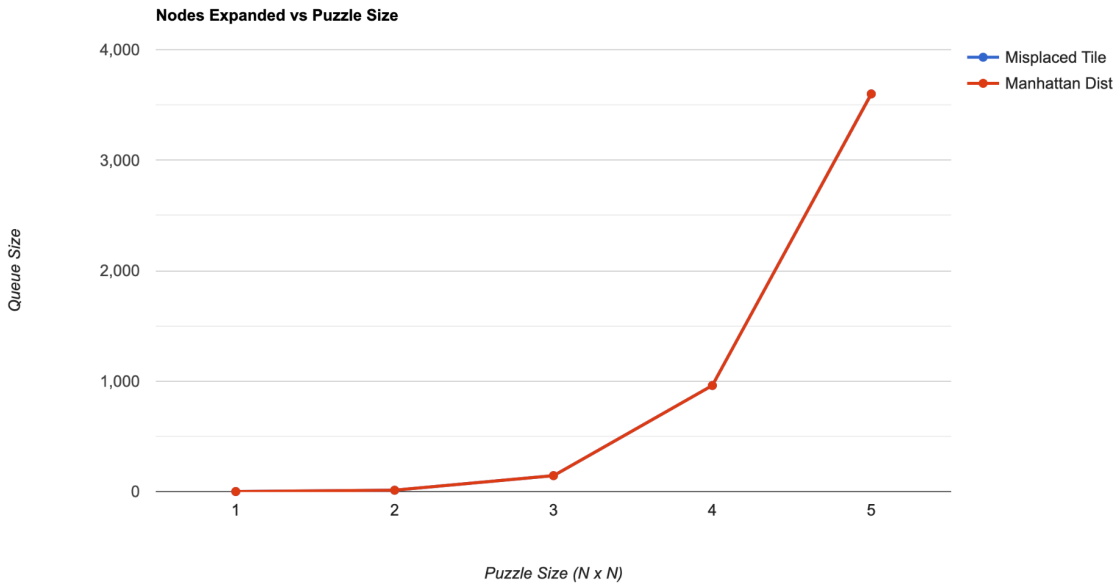
However, if given a more complicated puzzle besides the reverse of the goal state, like the default puzzle itself:



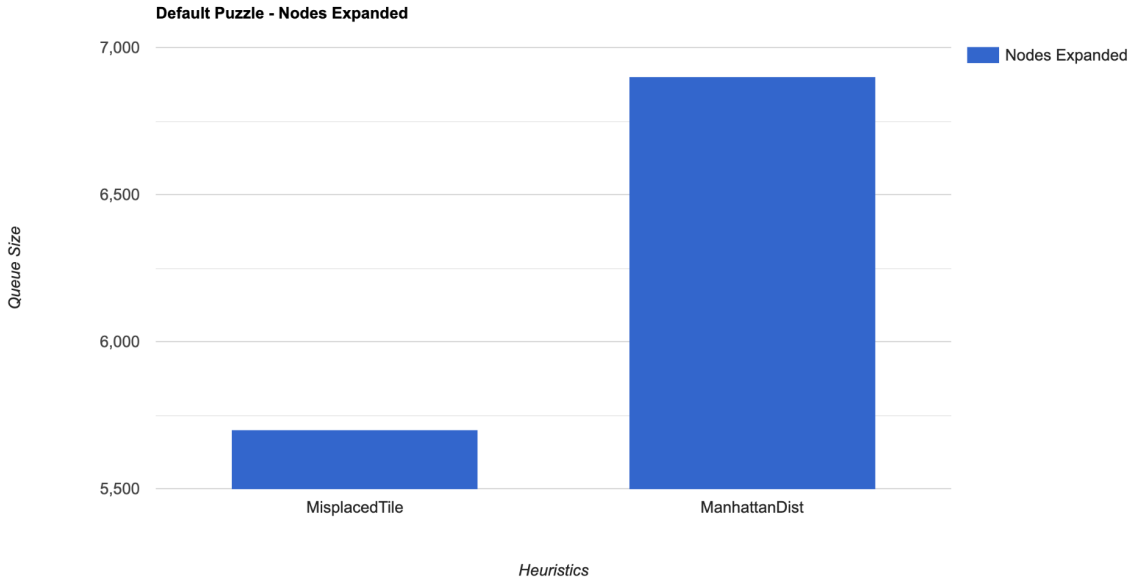
Misplaced tile performs well compared to Manhattan distance where it expanded 19 nodes vs 23 nodes.

Default Puzzle Queue Size: Misplaced Tile VS Manhattan Distance:

Similarly to the previous test of the number of expanded nodes vs the size of the puzzle, misplaced tile and manhattan distance performs exactly the same:



However, just like the previous test:



Misplaced Tile performs a lot better compared to Manhattan distance with a queue size of 5700 vs the queue size of 6900 given the default puzzle.

Conclusion:

Overall, it is expected that uniform cost search to be the worst performing algorithm and A* with Misplaced Tile Heuristic to be the best considering the overall problem of the puzzle which is essentially misplaced counters and not movement based, which explains why Manhattan distance performs worse than misplaced tiles.

Program Trace:

Here is an example of how the program works. Given an initial state, can be the default, custom, or a randomly generated state, the program will output whatever state it chooses to expand. After which, it will print the statistics and also provide the steps in how to solve the puzzle.

```
trunglam@Trungs-MacBook-Pro AdjustTheCounters % ./adjust_the_counters
Enter Algorithm: 0 - Uniform Cost Search, 1 - A* w/ Misplaced Tile Heuristic, 2 - A* w/ Manhattan
Distance Heuristic: 1
Enter 1: Default, 2: Custom, 3: Random: 3
Enter number of counters that are perfect squares: 9

Solution Trace:
-------------------------
Initial State:
3 7 9
6 2 5
4 1 8

g(n) : 1 , h(n) : 8
1 7 9
6 2 5
4 3 8

g(n) : 2 , h(n) : 7
1 2 9
6 7 5
4 3 8

g(n) : 3 , h(n) : 6
1 2 3
6 7 5
4 9 8

g(n) : 4 , h(n) : 4
1 2 3
6 7 5
4 8 9

g(n) : 5 , h(n) : 3
1 2 3
4 7 5
6 8 9

g(n) : 6 , h(n) : 2
1 2 3
4 5 7
6 8 9

g(n) : 7 , h(n) : 0
1 2 3
4 5 6
```

```
7 8 9

-------------------------
Success! Found Solution:
Nodes Expanded: 7
Max Queue Size: 252
1 2 3
4 5 6
7 8 9

-------------------------
Solution Step:
Swap 1, 3
Swap 2, 7
Swap 3, 9
Swap 8, 9
Swap 4, 6
Swap 5, 7
Swap 6, 7
```

Source Code:

```
#include <iostream>
#include <sstream>
#include <vector>
#include <algorithm>
#include <random>
#include <chrono>
#include <cmath>

using namespace std;

//Depth Table (Depth value, state)
vector< pair<int, vector<int> > > depthTable;

//Parent Table (Node Vector, Parent Vector)
vector< pair< vector<int>, vector<int> > > parentTable;

//Action Table (Node Vector, action)
vector< pair< vector<int>, string> > actionTable;

//Global Variables
vector< vector<int> > repeatedNodes;
int nodesExpanded = 0;
int maxQueueSize = 0;
int nodeID = 0;
int parentID = 1;

//print solution steps
void printSolution(vector<int> node, vector<int> initState){
    vector<int> currParent;
    currParent.push_back(-1);


    if(node == initState){
        return;
    }

    //retrieve parent
    for(int i = 0; i < parentTable.size(); i++){
        if(parentTable[i].first == node){
            currParent = parentTable[i].second;
            break;
        }
    }

    //retrieve action
    for(int i = 0; i < actionTable.size(); i++){
        if(actionTable[i].first == node){
            printSolution(currParent, initState);
            cout << actionTable[i].second << endl;
            break;
        }
    }
}

//Set state from user input
vector<int> setState(string userInput){
```

```
    vector<int> temp;

    string counters;
    stringstream s(userInput);
    while(s >> counters){
        temp.push_back(stoi(counters));
    }
    return temp;
}

//set a random state based on user input of number of counters
vector<int> setRandomState(int counters){
    vector<int> temp;
    for(int i = 1; i <= counters; i++){
        temp.push_back(i);
    }
    unsigned seed = chrono::system_clock::now().time_since_epoch().count();
    shuffle(temp.begin(), temp.end(), default_random_engine(seed));

    return temp;
}

//expand nodes
vector< vector<int> > expandNode(vector<int> state){
    nodesExpanded++;
    vector< vector<int> > result;
    int depth = 0;

    for(int i = 0; i < state.size(); i++){
        for(int j = 0; j < state.size(); j++){
            vector<int> temp = state;
            if(i == j){
                continue;
            }
            swap(temp[i], temp[j]);
            result.push_back(temp);

            //add to action and parent tables
            string c1, c2, action;
            c1 = to_string(temp[i]);
            c2 = to_string(temp[j]);
            action = "Swap " + c1 + ", " + c2;
            actionTable.push_back(make_pair(temp, action));
            parentTable.push_back(make_pair(temp, state));
        }

    }
    //remove duplicates
    sort(result.begin(), result.end());
    result.erase(std::unique(result.begin(), result.end()), result.end());


    //retrieve depth of parent
    for(int i = 0; i < depthTable.size(); i++){
        if(depthTable[i].second == state){
            depth = depthTable[i].first;
            break;
        }
    }
```

```
    //update depth table
    depth += 1;
    for(int i = 0; i < result.size(); i++){
        depthTable.push_back(make_pair(depth, result[i]));
    }



    return result;
}

//get the depth of the state
int getDepth(vector<int> node){
    int depth = 0;
    for(int i = 0; i < depthTable.size(); i++){
        if(depthTable[i].second == node){
            depth = depthTable[i].first;
            return depth;
        }
    }
    return -1;
}

//print matrix of the state
void printMatrix(vector<int> node, int matrixSize){

    int index = 0;
    for(int i = 0; i < matrixSize; i++){
        for(int j = 0; j < matrixSize; j++){
            cout << node[index] << " ";
            index++;
        }
        cout << endl;
    }
    cout << endl;
}

//calculate Manhattan distance for a node
int manhattanHeuristic(vector<int> node){
    vector<int> goalState;
    int score = 0;
    for(int i = 1; i <= node.size(); i++){
        goalState.push_back(i);
    }

    for(int i = 0; i < node.size(); i++){
        for(int j = 0; j < goalState.size(); j++){
            if(node[i] == goalState[j]){
                score += abs(i - j);
            }
        }
    }

    return score;

}

//Calculate Misplaced Tile Heuristic for a node
```

```
int misplacedHeuristic(vector<int> node){
    vector<int> goalState;
    int score = 0;
    for(int i = 1; i <= node.size(); i++){
        goalState.push_back(i);
    }

    for(int i = 0; i < node.size(); i++){
        if(node[i] != goalState[i]){
            score++;
        }
    }

    return score;
}

//Calculate A Star for a node based on a heuristic inputted by user
int aStar(vector<int> node, int heuristic){
    if(heuristic == 0){
        return getDepth(node);
    }
    if(heuristic == 1){
        return misplacedHeuristic(node) + getDepth(node);
    }
    else{
        return manhattanHeuristic(node) + getDepth(node);
    }
}

//Check if node is goal
bool goalCheck(vector<int> node){
    vector<int> goalState;
    int score = 0;
    for(int i = 1; i <= node.size(); i++){
        goalState.push_back(i);
    }

    for(int i = 0; i < node.size(); i++){
        if(node[i] != goalState[i]){
            return false;
        }
    }

    return true;
}

//Check if node is a repeat state
bool repeatCheck(vector<int> node){
    for(int i = 0; i < repeatedNodes.size(); i++){
        if(repeatedNodes[i] == node){
            return true;
        }
    }
    return false;
}

//General algorithm inspired from slide
vector<int> generalSearch(vector<int> initState, int heuristic){
```

```
    //check if initial state is goal state
    if(goalCheck(initState)){
        cout << "Success! Found Solution:" << endl;
        cout << "Nodes Expanded: " << nodesExpanded << endl;
        cout << "Max Queue Size: " << maxQueueSize << endl;
        printMatrix(initState, sqrt(initState.size()));
        return initState;
    }

    //Expand Node and insert to queue
    vector< vector<int> > nodes = expandNode(initState);
    repeatedNodes.push_back(initState);

    //Start Loop
    while(1){

        //Sort queue by A_Star w/ Uniform Cost Search (0), Misplaced Heuristic (1), Manhattan Heuristic
(2)
        vector< pair<int, vector<int> > > nodesTable;
        for(int i = 0; i < nodes.size(); i++){
            nodesTable.push_back(make_pair(aStar(nodes[i], heuristic), nodes[i]));
        }
        sort(nodesTable.begin(), nodesTable.end());

        vector< vector<int> > sortedNodes;

        //grab sorted vectors from table into a queue vector
        for(int i = 0; i < nodesTable.size(); i++){
            sortedNodes.push_back(nodesTable[i].second);
        }

        // if queue is empty
        if(sortedNodes.size() == 0){
            cout << "Failure: Unable to Find Solution!" << endl;
            return initState;
        }

        //update max queue size
        if(sortedNodes.size() >= maxQueueSize){
            maxQueueSize = sortedNodes.size();
        }

        //Pop front of element
        vector<int> node = sortedNodes.front();

        //check if its a repeating node
        while(repeatCheck(node)){
            sortedNodes.erase(sortedNodes.begin());
            node = sortedNodes.front();
        }
        sortedNodes.erase(sortedNodes.begin());

        //add node to repeating nodes
        repeatedNodes.push_back(node);

        //if heuristic is uniform cost search
        if(heuristic == 0){
            cout << "g(n) : " << getDepth(node) << endl;
        }
```

```
        //if heuristic is misplaced tile
        if(heuristic == 1){
            cout << "g(n) : " << getDepth(node) << " , h(n) : " << misplacedHeuristic(node) << endl;
        }

        //if heuristic is manhattan distance
        if(heuristic == 2){
            cout << "g(n) : " << getDepth(node) << " , h(n) : " << manhattanHeuristic(node) << endl;
        }

        printMatrix(node, sqrt(node.size()));
        //if goal is found
        if(goalCheck(node)){
            cout << "-------------------------" << endl;
            cout << "Success! Found Solution:" << endl;
            cout << "Nodes Expanded: " << nodesExpanded << endl;
            cout << "Max Queue Size: " << maxQueueSize << endl;
            printMatrix(node, sqrt(node.size()));
            cout << "-------------------------" << endl;
            cout << "Solution Step:" << endl;
            return node;
        }

        //Expand and add to queue
        vector< vector<int> > temp = expandNode(node);
        for(int i = 0; i < temp.size(); i++){
            nodes.push_back(temp[i]);
        }
    }
    return initState;

}

int main(){
    string userInput;
    int userChoice;
    int nSize;
    int nCounters;
    int heuristicChoice;

    cout << "Enter Algorithm: 0 - Uniform Cost Search, 1 - A* w/ Misplaced Tile Heuristic, 2 - A* w/
Manhattan Distance Heuristic: ";
    cin >> heuristicChoice;
    cout << "Enter 1: Default, 2: Custom, 3: Random: ";
    cin >> userChoice;
    cin.ignore();

    string defaultState = "7 24 10 19 3 12 20 8 22 23 2 15 25 18 13 11 21 5 9 16 17 4 14 1 6";
    vector<int> initState = setState(defaultState);
    nSize = sqrt(initState.size());

    if(userChoice == 2){
        cout << "Enter counters from left to right, top to bottom seperated by space: ";
        getline(cin, userInput);
        initState = setState(userInput);
        nSize = sqrt(initState.size());
    }
```

```
    if(userChoice == 3){
        cout << "Enter number of counters that are perfect squares: ";
        cin >> nCounters;
        initState = setRandomState(nCounters);
        nSize = sqrt(initState.size());
    }

    //initialize depth table
    depthTable.push_back(make_pair(0, initState));

    //initialize parent table and action table
    parentTable.push_back(make_pair(initState, initState));

    //Start search
    cout << endl << "Solution Trace:" << endl;
    cout << "------------------------" << endl;
    cout << "Initial State: " << endl;
    printMatrix(initState, nSize);
    printSolution(generalSearch(initState, heuristicChoice), initState);
    cout << endl;


    return 0;
}
```