Project 1 CS205: Introduction to Artificial Intelligence, Dr. Eamonn Keogh

Jakin Chan
SID 862181945
Email jchan419@ucr.edu
Date May-15-2023

In completing this assignment, I consulted:

- The Project instructions and outline
    - https://www.dropbox.com/s/c757j41ciksav1a/Project_1_The_Eight_Puzzle_CS_205.pdf?dl=0
- Problem Solution
    - https://sites.google.com/site/geduldspiele/SolutionNineMenInATrench
- Student Shruti Jawale
    - Student Shruti Jawale reportedly emailed Dr. Eamonn Keogh on the specifics of the requirements regarding the alternate project.

All important code is original. Non original libraries/structures and functions are outlined below:

- Library chrono
    - For time tracking purposes
- Other
    - Sets
        - For tracking duplicate states
    - Priority queue
        - For deciding what state to look at next

Outline of this report

CS205: Project 1: Nine Men in a Trench

Jakin Chan, SID 862181945 May-15-2023

Introduction

The Nine Men in a Trench problem consists of 13 cells and 9 persons, labeled 1-9. 10 of the cells are in a row, with the first cell empty, the next 8 cells holding persons 2-9, and the last cell holding person 1. The three extra cells are empty and adjacent to the cells of persons 4, 6, and 8. The goal of the puzzle is to end with a goal state where the first 9 cells are holding persons 1-9 in that order.

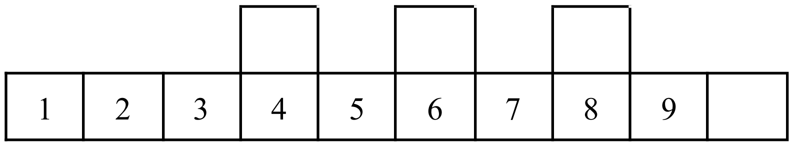

Figure 1: Goal State for the Nine Men in a Trench Problem

The rules are such that moves are limited to movement from a person in one cell to an adjacent empty cell. This means that each cell can only hold a maximum of one person, and that each person can only move to an adjacent cell not including diagonals.

This project is for Dr. Eamonn Keogh's Introduction to Artificial Intelligence course, at the University of California, Riverside, during the Spring Quarter of 2023.

Comparison of Algorithms

In this section, the algorithms implemented include Uniform Cost Search, A* with the Displaced Person Heuristic, and A* with Manhattan Distance Heuristic.

Uniform Cost Search

As noted in the assignment breakdown document provided, Uniform Cost Search is just A*, but with h(n) hardcoded to 0. This means that only g(n) is taken into account. With every move having the same worth, g(n) is essentially the number of moves from the start of the puzzle.

Displaced Person Heuristic

A* with the Displaced Person Heuristic is the same as Uniform Cost Search, but in this case, we take advantage of h(n). In A* with the Displaced Person Heuristic, h(n) is calculated by adding up the number of persons that are not in their correct final cell. In Figure 2 below, the highlighted cells include a person that is not in their expected final cell. There are two such cases, so h(n) is 2.

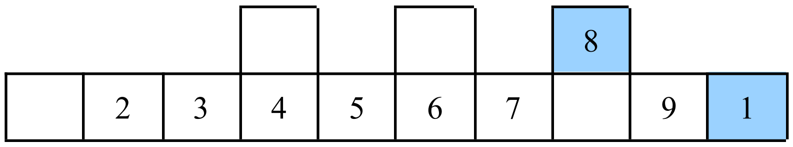

Figure 2: Example State with displaced persons highlighted

Manhattan Distance Heuristic

The Manhattan Distance Heuristic uses h(n) on top of g(n), similar to the Displaced Person Heuristic. However, in calculating h(n), instead of counting the number of displaced persons, it finds the Manhattan distance between each person and their target cell. Looking at Figure 2 above, h(n) would be 10, with 8 and 1 being 1 and 9 cells from their target cell respectively.

Comparison of Algorithms

When running the algorithms, it was found that Uniform Cost Search and A* with Displaced Person Heuristic would result in out-of-memory errors. This persisted even after throwing away duplicate states.

As A* with the Manhattan Distance Heuristic was the only algorithm that I was able to run to completion, there is only data on this algorithm. In Figure 3 below, the algorithm was run 5 times and the data is shown along with the average. The number of nodes expanded and the max queue size stayed the same across all the times that the algorithm was run. This makes sense as it is solving the problem the same way each time. As for the run-time, it averages out at 279 seconds and 110 seconds if the -O4 flag was used.

Run	Nodes Expanded	Max Queue Size	Runtime (seconds)	Runtime (seconds) (with -O4)
1	3146739	490687	287	110
2	3146739	490687	277	111
3	3146739	490687	278	112
4	3146739	490687	279	108
5	3146739	490687	274	109
Average	3146739	490687	279	110

Figure 3: Data for running A* with Manhattan Distance

In Figure 4 below, there is a graph involving g(n) and the time it takes in seconds to see the first instance of a state with that g(n) from the queue. The overall shape is exponential, which is expected as the problem expands as a tree, so each total number of moves adds exponentially more total states. However, what is interesting is the inclusion of steps in the graph. From g(n) =

46-51, 58-64, and 69-75, the difference in the amount of time to reach the next g(n) threshold is minimal. During these times, the most promising state to expand is derived from a state just expanded or close to it..

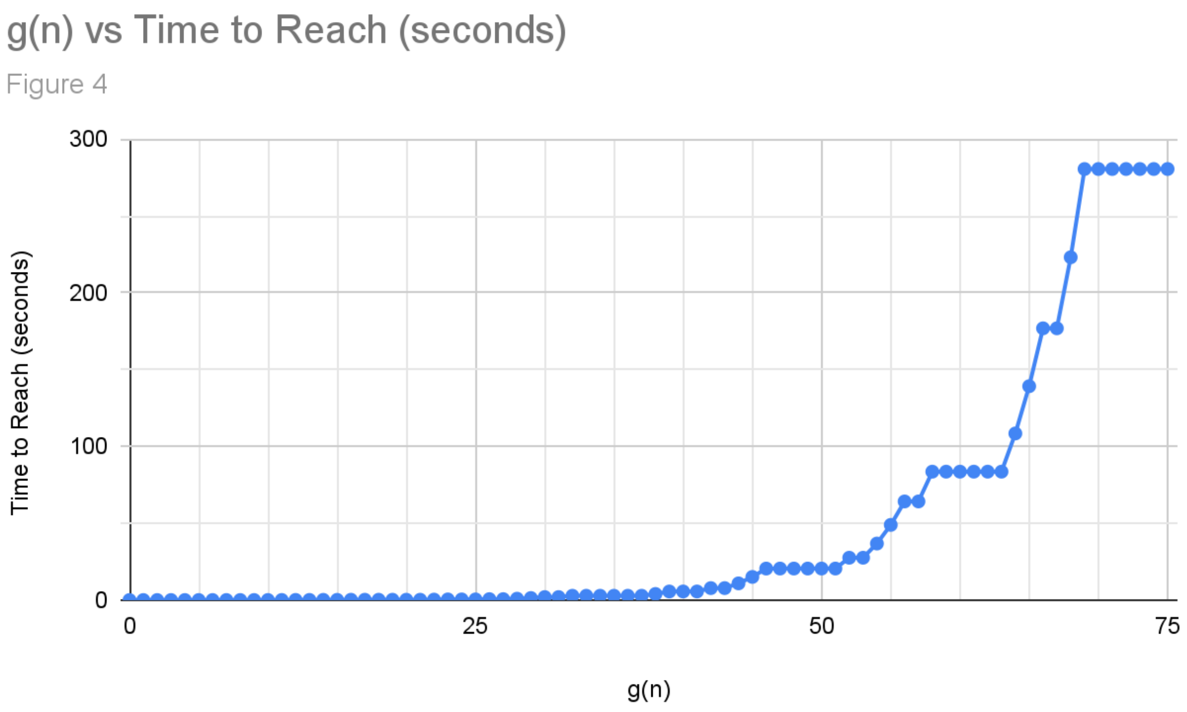

Figure 4: g(n) vs Time to Reach (seconds)

Conclusion

In conclusion, from running the A* with Manhattan Distance Heuristic on the Nine Men in a Trench problem, it can be concluded that:

- Every run of the same algorithm results in the same amount of nodes expanded and nodes in the queue.
- The time it takes to reach each g(n) threshold follows an exponential trend.
- As h(n) is hardcoded to 0 for Uniform Cost Search, it is essentially breadth first search as it checks each state in increasing g(n) and the cost is 1 for every move.

The following is a traceback of using Manhattan Distance for the heuristic

```
1: for Uniform Cost Search
2: for A* Misplaced Tile Heuristic
3: for A* Manhattan Distance
4: for A* Ignoring Soldier Placements
3
The best state to expand with a g(n) = 0 and h(n) = 9 is...
         0    0    0
 0  2  3  4  5  6  7  8  9  1
The best state to expand with a g(n) = 1 and h(n) = 10 is...
         4    0    0
 0  2  3  0  5  6  7  8  9  1
The best state to expand with a g(n) = 1 and h(n) = 10 is...
         0    0    8
 0  2  3  4  5  6  7  0  9  1
The best state to expand with a g(n) = 1 and h(n) = 10 is...
         0    0    0
 2  0  3  4  5  6  7  8  9  1

                    //Deleted a lot of lines for space

The best state to expand with a g(n) = 72 and h(n) = 3 is...
         0    0    7
 1  2  3  4  5  6  0  8  0  9
The best state to expand with a g(n) = 73 and h(n) = 2 is...
         0    0    7
 1  2  3  4  5  6  0  8  9  0
The best state to expand with a g(n) = 73 and h(n) = 2 is...
         0    5    0
 1  2  3  4  0  6  7  8  9  0
The best state to expand with a g(n) = 73 and h(n) = 2 is...
         0    0    0
 1  2  3  4  5  6  7  0  8  9
The best state to expand with a g(n) = 74 and h(n) = 1 is...
         0    0    0
 1  2  3  4  5  6  7  8  0  9
Solution Found!
g: 75 h: 0
4 Up
5 Left
```

6 Left
7 Left
8 Left
9 Left
1 Left
2 Left
3 Left
5 Left
6 Left
7 Left
8 Left
9 Left
1 Left
1 Up
9 Right
9 Right
8 Right
8 Right
9 Right
8 Right
1 Down
1 Left
1 Left
1 Up
7 Right
7 Right
6 Right
6 Right
3 Right
2 Right
5 Right
5 Right
3 Right
3 Right
5 Up
3 Right
2 Right
2 Right
2 Right
1 Down

```
1 Left
2 Left
3 Left
1 Left
2 Left
3 Left
1 Left
4 Left
4 Left
2 Left
3 Left
4 Left
6 Left
5 Down
5 Left
6 Left
7 Down
7 Left
8 Left
9 Left

Number of Nodes Expanded: 3146739
Max Queue Size: 490687
Time to run: 5703354 milliseconds
```

//The difference in runtime between what is shown in Figure 3 and here is due to traceback printing

URL to my code is https://github.com/JakinChan200/CS205
NineMenTrench.cpp

```
#include <bits/stdc++.h>

using namespace std;


int numNodesExpanded = 0;
int maxQueueSize = 0;
int checkpoint;

vector<vector<int>> start = {{-1, -1, -1, 0, -1, 0, -1, 0, -1, -1},
                            {0, 2, 3, 4, 5, 6, 7, 8, 9, 1}};

// vector<vector<int>> start = {{-1, -1, -1, 0, -1, 8, -1, 0, -1, -1},
//                              {1, 2, 3, 4, 5, 6, 7, 0, 9, 0}};

// vector<vector<int>> start = {{-1, -1, -1, 1, -1, 5, -1, 8, -1, -1},
//                              {2, 0, 0, 0, 0, 3, 4, 6, 7, 9}};

struct node{
    vector<vector<int>> state;
    int h = 0;
    bool isSolution = false;
    vector<string> path;
};

struct Compare{
    bool operator()(node &a, node &b){
        //return a.path.size() + a.h > b.path.size() + b.h;
        return a.h + a.path.size() != b.h + b.path.size() ? a.h + a.path.size() > b.h + b.path.size() :
a.path.size() > b.path.size();
        //return a.h != b.h ? a.h > b.h : a.path.size() > b.path.size();
    }
};

void printState(vector<vector<int>> &state){
    for(int i = 0; i < state.size(); i++){
        for(int k = 0; k < state[i].size(); k++){
            if(state[i][k] < 0){
                cout << "    ";
            }else{
                cout << "   " << state[i][k];
            }
        }
        cout << endl;
    }
    return;
}

bool isDupe(node &trial, set<vector<vector<int>>> &dupeStash){
    int dupeStashSize = dupeStash.size();
    dupeStash.insert(trial.state);
    return dupeStashSize == dupeStash.size();
}
```

```
bool checkCorrectness(node &trial, int algo){
    trial.h = 0;
    for(int i = 0; i < trial.state.size(); i++){
        for(int j = 0; j < trial.state[i].size(); j++){
            if(trial.state[i][j] != -1 && trial.state[i][j] != 0){

                if(i == 1 && j == trial.state[i][j]-1){
                    continue;
                }

                switch(algo){
                    case 4:
                        if(trial.state[i][j] == 1){
                            trial.h += (1-i) + abs(j-(trial.state[i][j]-1));
                            trial.isSolution = trial.h == 0;
                            return trial.isSolution;
                        }
                        break;
                    case 3:
                        trial.h += (1-i) + abs(j-(trial.state[i][j]-1));
                        break;
                    case 2:
                        trial.h++;
                        break;
                    default:
                        return false;
                        break;
                }
            }
        }
    }
    if(trial.h == 0){
        trial.isSolution = true;
    }
    return trial.h == 0;
}

void movePerson(node &trial, int startPosX, int startPosY, int endPosX, int endPosY, string step){
    trial.state[endPosX][endPosY] = trial.state[startPosX][startPosY];
    trial.state[startPosX][startPosY] = 0;
    trial.path.push_back(step);
}

node algorithmn(node &startingState, int algo){

    // auto startingTime = std::chrono::high_resolution_clock::now();
    // auto checkpoint1 = std::chrono::high_resolution_clock::now();
    // int gCounter = 0;

    numNodesExpanded = 0;
    set<vector<vector<int>>> duplicates;
    node currentNode;
    node temp;

    priority_queue<node, vector<node>, Compare> tree;
    tree.push(startingState);
    if(checkCorrectness(startingState, algo)){return startingState;}
    isDupe(startingState, duplicates);
    while(!tree.empty()){
        maxQueueSize = max((int)tree.size(), maxQueueSize);
        currentNode = tree.top();
        tree.pop();
        numNodesExpanded++;


        // if(currentNode.path.size() > gCounter){
        //     auto checkpoint2 = std::chrono::high_resolution_clock::now();
        //     cout << "Time elapsed to finish g(n): " << gCounter << " is " <<
std::chrono::duration_cast<std::chrono::milliseconds>(checkpoint2-checkpoint1).count() << "
milliseconds." << endl;
        //     cout << "Time elapsed to reach g(n): " << currentNode.path.size() << " is " <<
std::chrono::duration_cast<std::chrono::milliseconds>(checkpoint2-startingTime).count() << "
milliseconds." << endl;
        //     checkpoint1 = checkpoint2;
        //     gCounter = currentNode.path.size();
        // }

        // cout << "The best state to expand with a g(n) = " << currentNode.path.size() << " and h(n) =
" << currentNode.h << " is... " << endl;
        // printState(currentNode.state);
        string step = "";

        //Checking the upper level
        for(int i = 0; i < currentNode.state[0].size(); i++){
            if(currentNode.state[0][i] == 0){
                if(currentNode.state[1][i] == 0){ continue;}
                temp = currentNode;

                step = to_string(currentNode.state[1][i]) + " Up\n";
                movePerson(temp, 1, i, 0, i, step);
                if(!isDupe(temp, duplicates)){
                    if(checkCorrectness(temp, algo)){
                        return temp;
                    }else{
                        tree.push(temp);
                    }
                }
            }
        }

        //Checking the Lower Level
        for(int i = 0; i < currentNode.state[1].size(); i++){
            if(currentNode.state[1][i] == 0){
                if(i != 0){
                    if(currentNode.state[1][i-1] != 0){
                        temp = currentNode;

                        step = to_string(currentNode.state[1][i-1]) + " Right\n";
                        movePerson(temp, 1, i-1, 1, i, step);
                        if(!isDupe(temp, duplicates)){
                            if(checkCorrectness(temp, algo)){
                                return temp;
                            }else{
                                tree.push(temp);
                            }
                        }
                    }
                }

                if(currentNode.state[0][i] > 0){
                    temp = currentNode;

                    step = to_string(currentNode.state[0][i]) + " Down\n";
                    movePerson(temp, 0, i, 1, i, step);
                    if(!isDupe(temp, duplicates)){
                        if(checkCorrectness(temp, algo)){
                            return temp;
                        }else{
                            tree.push(temp);
                        }
                    }
                }

                if(i != currentNode.state[1].size()-1){
                    if(currentNode.state[1][i+1] != 0){
                        temp = currentNode;

                        step = to_string(currentNode.state[1][i+1]) + " Left\n";
                        movePerson(temp, 1, i+1, 1, i, step);
                        if(!isDupe(temp, duplicates)){
                            if(checkCorrectness(temp, algo)){
                                return temp;
                            }else{
                                tree.push(temp);
                            }
                        }
                    }
                }
            }
        }

    }

    return currentNode;
}

int main(int argc, char** argv){
    int algo = 0;

    while(algo < 1 || algo > 4){
        cout << "1: for Uniform Cost Search\n2: for A* Misplaced Tile Heuristic \n3: for A* Manhattan
Distance \n4: for A* Ignoring Soldier Placements" << endl;
        cin >> algo;
    }

    node startingState;
    startingState.state = start;
    bool correct = checkCorrectness(startingState, algo);

    auto start = std::chrono::high_resolution_clock::now();
    node answer = algorithmn(startingState, algo);
    auto stop = std::chrono::high_resolution_clock::now();

    if(answer.isSolution){
        cout << "Solution Found!" << endl;
        cout << "g: " << answer.path.size() << " h: " << answer.h << endl;
        for(int i = 0; i < answer.path.size(); i++){
            cout << answer.path[i];
        }
    }else{
        cout << "Solution Not Found!" << endl;
    }

    cout << endl;
    cout << "Number of Nodes Expanded: " << numNodesExpanded << endl;
    cout << "Max Queue Size: " << maxQueueSize << endl;
    cout << "Time to run: " << std::chrono::duration_cast<std::chrono::seconds>(stop - start).count() <<
" seconds" << endl;

    return 0;
}
```