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

Sidharth Ramkumar
SID 862129657
Email sramk002@ucr.edu
Date May 11, 2023

In completing this assignment I consulted:

- The Blind Search and Heuristic Slides from CS205 Lecture
- Copying 2D array to another 2D array: https://cplusplus.com/forum/general/263317/
- How to use priority queue STL for objects: https://stackoverflow.com/questions/19535644/how-to-use-the-priority-queue-stl-for-objects
- Convert ArrayList<String> to String[] array [duplicate]: https://stackoverflow.com/questions/5374311/convert-arrayliststring-to-string-array

All important code is original. The only modified subroutine is:

- Overloaded comparison (**struct comp**) operator used to sort priority queue data structure.

Outline of this report:

CS205: Assignment 1 - The Eight Puzzle

Sidharth Ramkumar, SID 862129657 May 11, 2023

Introduction

This project is meant to emulate the standard 8-puzzle problem. This implementation of the puzzle utilizes a 3 by 3 grid where each grid holds a number tile. The way to solve this puzzle is to use a set combination of tiles on the grid and a blank that allows for the sliding of these tiles to reach a goal state. The tiles can either be moved up, down, left, or right depending on how the tiles are positioned relative to the space on the grid.

This project was developed for Dr. Eamonn Keogh's CS205 AI course taught at the University of California, Riverside in Spring 2023 [1]. I used C++ to implement the 8-puzzle. My implementation of the project is meant to take the 8-puzzle problem space and transform it in a way applicable to specific search algorithms. I had designed this project to fit the graph-search function and perform uniform-cost search and A* search with very minimal changes. I had used graph search to reduce the amount of time it takes to solve larger puzzles while keeping track of the relevant information for each step.

UCS and Heuristics

For this project, I implemented Uniform Cost Search, A star search using the Misplaced Tile heuristic, and A star search using the Manhattan Distance heuristic.

Uniform Cost Search

This algorithm assumes that total cost = g(n) + h(n), where g(n) is a cost of 1 and the heuristic h(n) was set to a default value of 0. The algorithm operates under the assumption that each move has a cost of 1 and tries to find the most optimal moves by simply going through the entire search tree.

Misplaced Tile Heuristic

This algorithm now assigns a value to h(n) using the same formula: total cost = g(n) + h(n). The heuristic identifies the number of tiles that don't match the positions of the goal state (ignoring the blank tile) and sets h(n) to that value. As a result this eliminates the need to expand every node in the search tree.

[1] https://www.dropbox.com/s/c757j41ciksav1a/Project_1_The_Eight_Puzzle_CS_205.pdf?dl=0

Manhattan Distance Heuristic

For this algorithm I modified the h(n) value or the heuristic function to now use the manhattan distance algorithm. Manhattan distance is a metric in which the distance between two points is the sum of the absolute differences of their Cartesian coordinates [2]. Therefore the distance from a tile is calculated through the goal index values minus the current index values to provide a cost for h(n).

Test Cases for Algorithms

To efficiently test my implementation and observe the costs for each of the algorithms, I have made my own test cases, which can be seen below. Each test case and its complexity is presented by the number of possible moves to solve that given puzzle. Since the maximum possible moves for any 8-puzzle is 31, the most difficult puzzle has a solution depth of exactly 31 moves. The impossible puzzle cannot be solved by any of the algorithms and returns NULL.

```
//0 moves [1]
const int no_moves[N][N] = {{1, 2, 3},
                            {4, 5, 6},
                            {7, 8, 0,}};

//1 move [2]
const int one_move[N][N] = {{1, 2, 3},
                            {4, 5, 6},
                            {7, 0, 8,}};

//2 moves [3]
const int two_moves[N][N] = {{1, 2, 0},
                             {4, 5, 3},
                             {7, 8, 6,}};

//3 moves [4]
const int three_moves[N][N] = {{1, 2, 3},
                               {0, 5, 6},
                               {4, 7, 8,}};

//16 moves [5]
const int sixteen_moves[N][N] = {{1, 5, 3},
                                 {2, 4, 6},
                                 {7, 8, 0,}};
//22 moves [6]
const int twenty_two_moves[N][N] = {{8, 7, 1},
                                    {6, 0, 2},
                                    {5, 4, 3,}};
//23 moves [7]
const int twenty_three_moves[N][N] = {{7, 2, 5},
                                      {3, 1, 0},
                                      {6, 4, 8,}};
```

```
//31 moves [8]
const int max_moves[N][N] = {{8, 6, 7},
                             {2, 5, 4},
                             {3, 0, 1,}};

//impossible to solve [9]
const int impossible[N][N] = {{1, 2, 3},
                              {4, 5, 6},
                              {8, 7, 0,}};
```

These are the “default” puzzles available on the program. They can be selected through the menu options.

[2] https://medium.com/analytics-vidhya/euclidean-and-manhattan-distance-metrics-in-machine-learning-a5942a8c9f2f#:~:text=Manhattan%20distance%20is%20a%20metric,-coordinates%20and%20y-coordinates.

Comparison of Algorithms

I mainly used the two metrics to analyze the time and space cost of each algorithm. These metrics are the number of nodes expanded and the max nodes in the priority queue. The bar charts below show these metrics for each puzzle.

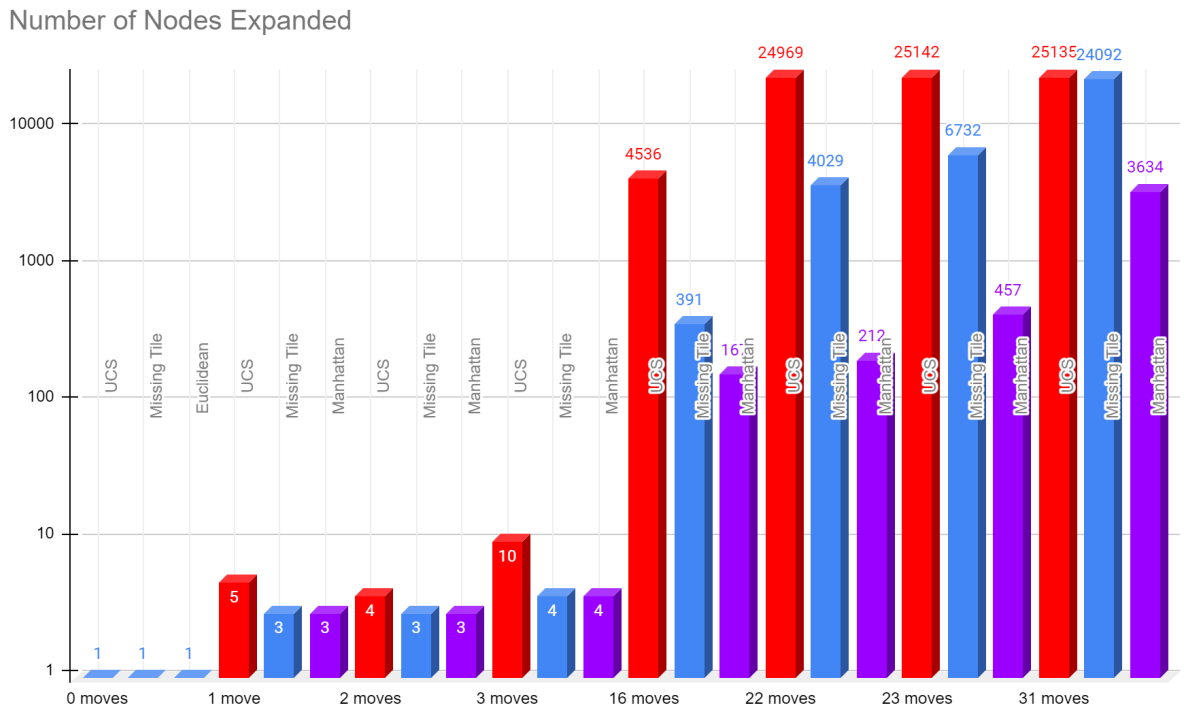

Figure 1: A comparison of nodes expanded between all algorithms ranging in puzzle difficulty using log-scaled bar chart

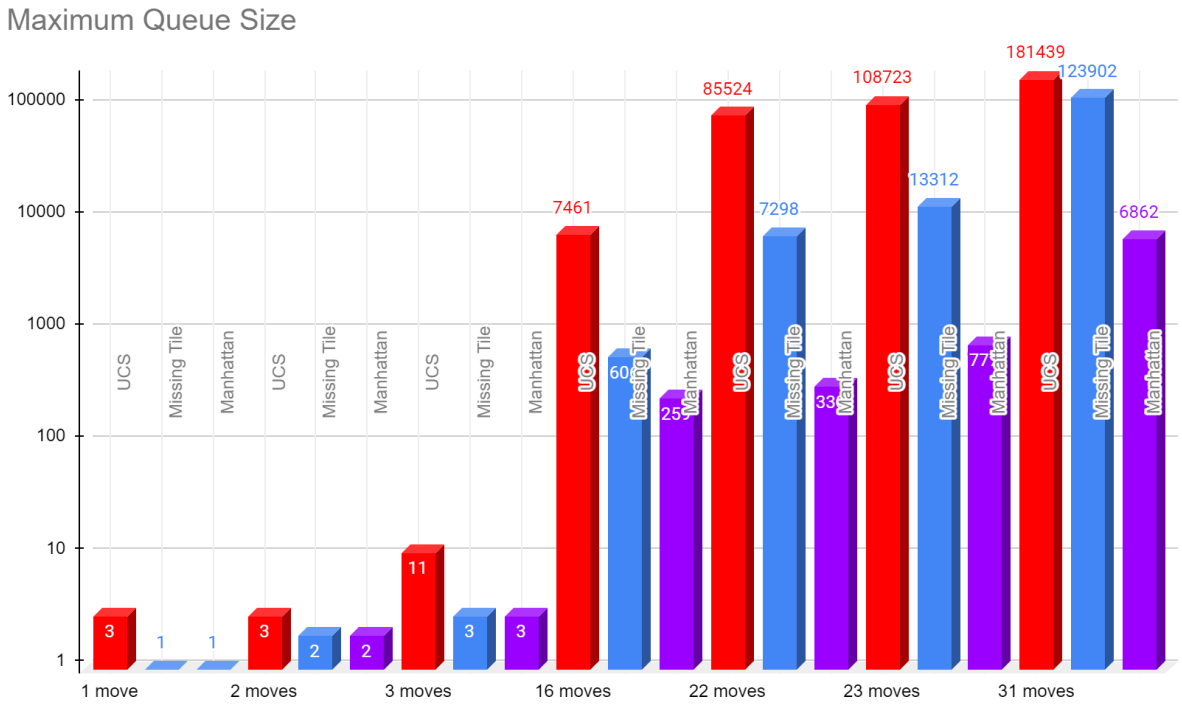

Figure 2: A comparison of maximum queue size between all algorithms ranging in difficulty using a log-scaled bar chart

By analyzing **figure** 1, we can observe the time complexity through the number of nodes expanded. There is an apparent spike in the number of nodes expanded when it comes to any 8-puzzle that takes 10 or more moves to complete. This time cost is especially apparent with the weakest algorithm Uniform Cost Search, where a large exponential growth is observed through the bar chart visual. The A* search with misplaced tile seems to have less of a time complexity until it reaches the most difficult puzzle. The Manhattan distance heuristic has a much lower time complexity in comparison, only expanding a fraction of the nodes that UCS or missing tile heuristic expands.

The metric for maximum queue size demonstrates the spatial complexity of each search algorithm. The patterns shown in **Figure 2** are nearly identical with Manhattan showing a significantly lower complexity for the higher values than missing tile or UCS. One interesting observation is how these algorithms don't differ too much when it comes to the smaller values, but the peak differences can be observed in the mid-range difficulty 16-23 moves.

Conclusion

By observing the performance of each algorithm, it is very clear the A star search with the Manhattan Distance heuristic is the best performing algorithm out of the three. We can also see that as the difficulty of the puzzle increases, there is a large degree of separation between the time and space complexity between each algorithm. Finally, this project as a whole shows that a better heuristic can lead to an exponentially better approach to identifying a goal state in a search space.

Traceback of an easy puzzle

Welcome to my CS205 8-puzzle solver. Type '1' to use a default puzzle. or '2' to enter your own puzzle.
2
==========
Enter your puzzle, use a zero to represent the blank. Please only use valid 8-puzzles.
Enter the puzzle with spaces in between the tiles. Hit 'enter' only when finished.

Enter the first row, use space between numbers:1 2 3
Enter the second row, use space between numbers:4 0 6
Enter the third row, use
 space between numbers:7 5 8
1 2 3
4 * 6
7 5 8
==========
Enter your choice of algorithm
[1] Uniform Cost Search.
[2] A* with Misplaced Tile heuristic.
[3] A* with Manhattan Distance Heuristic.
3
1 2 3
4 * 6
7 5 8
expanding this node...

The best state to expand with g(n) = 1 and h(n) = 1 is...
1 2 3
4 5 6
7 * 8
expanding this node...

1 2 3
4 5 6
7 8 *
Goal state!

The depth of the solution: 2
Number of nodes expanded: 2
Maximum queue size: 5
==========

Traceback of a challenging puzzle

Welcome to my CS205 8-puzzle solver. Type '1' to use a default puzzle. or '2' to enter your own puzzle.
1
==========
Select the desired level of difficulty for your puzzle
[1] no_moves
[2] one_move
[3] two_moves
[4] three_moves
[5] sixteen_moves
[6] twenty_two_moves
[7] twenty_three_moves
[8] max_moves
5
You have chosen: sixteen moves.
1 5 3
2 4 6
7 8 *
==========
Enter your choice of algorithm
[1] Uniform Cost Search.
[2] A* with Misplaced Tile heuristic.
[3] A* with Manhattan Distance Heuristic.
3
1 5 3
2 4 6
7 8 *
expanding this node...

The best state to expand with g(n) = 1 and h(n) = 5 is...
1 5 3
2 4 *
7 8 6
expanding this node...

The best state to expand with g(n) = 1 and h(n) = 5 is...
1 5 3
2 4 6
7 * 8
expanding this node...

The best state to expand with g(n) = 2 and h(n) = 6 is...
1 5 *
2 4 3
7 8 6
expanding this node...

The best state to expand with g(n) = 2 and h(n) = 6 is...
1 5 3
2 * 6
7 4 8
expanding this node...

—----------
Skipping to bottom of the trace
—----------

The best state to expand with g(n) = 10 and h(n) = 6 is...
4 2 *
1 3 5
7 8 6
expanding this node...

1 2 3
4 5 6
7 8 *
Goal state!

The depth of the solution: 16
Number of nodes expanded: 259
Maximum queue size: 167
==========
Would you like to try another puzzle?
Hit 'y' to continue or 'n' to quit.

Search Algorithm Code

The entire code can be found at: https://github.com/sidrk01/eightpuzzle.git

```
#include "../header/searchalgos.h"

SearchAlgos::SearchAlgos(Problem problem, int num) {
   p = &problem;
   algo_choice = num;
}

void SearchAlgos::graph_search() {
   Node head(*p);
   vector<Node> list; //list of expanded Nodes
   bool first = true;
   unsigned int frontier_size = frontier.size();
   unsigned int size_nodes = 0;

   frontier.push(head); //initialize the frontier using initial state of
problem

   while(!frontier.empty()){ //loop do
       if(frontier.empty()){
           cout << "Unable to perform search." << endl;
           return;
       }

       if (frontier_size < frontier.size())
           frontier_size = frontier.size();

       Node temp = frontier.top();
       frontier.pop(); //choose a leaf node and remove it from the frontier


       if(temp.goal_test()){ //if a node contains a goal state then return the
corresponding solution
           temp.print_result(); //prints output of search
           cout << "Goal state!" << endl << endl;
           cout << node_goal_depth << temp.depth << endl;
           cout << nodes_expand << size_nodes << endl;
           cout << node_queue_max << frontier_size << endl;
           return;
       }

       //prints "trace" of expanded nodes (comment out for faster execution
time)
       if (frontier_size > 1) {
           cout << "The best state to expand with g(n) = " << temp.g_cost  << "
and h(n) = " << temp.h_cost << " is..." << endl;
       }
```

```
        temp.print_result();
        cout <<"expanding this node..." << endl << endl;

        list = expand(temp); //expand the chosen node
        explored_set.insert(head.state); //add to first node to explored set

        for (auto & i : list){ //adding resulting nodes to the frontier
            frontier.push(i);
        }
        size_nodes += 1;
    }
}

//helper function
vector<Node> SearchAlgos::expand(Node& curr) {
    int row = 0;
    int col = 0;
    curr.detect_space(row, col);
    vector<Node> list;

    for (unsigned i = 0; i < 4; i++){
        Node temp(curr);

        switch(i) {
            case 0: //slide up
                if (row > 0) {
                    temp.slide_up();
                    break;
                }
                else {
                    continue;
                }
            case 1: //slide down
                if (row < 2) {
                    temp.slide_down();
                    break;
                }
                else {
                    continue;
                }
            case 2: //slide left
                if (col > 0) {
                    temp.slide_left();
                    break;
                }
                else {
                    continue;
                }
            case 3: //slide right
```

```
                if (col < 2) {
                    temp.slide_right();
                    break;
                }
                else {
                    continue;
                }
            default:
                break;
        }
        //increments cost to match algorithm and sets other values
        if (explored_set.count(temp.state) == 0) {
            ++temp.g_cost;
            ++temp.depth;
            temp.set_heuristic(algo_choice);
            list.push_back(temp);
            explored_set.insert(temp.state); //adds node to the explored set
        }
    }

    return list;
}
```