Project 1 CS 170 Introduction to Artificial Intelligence

Thomas Shiels Instructor: Dr. Eamonn Keogh
ID 861151979
tshie001@ucr.edu
8-November-2018

In completing this assignment I consulted:

- For help with using a priority queue: http://www.cplusplus.com/reference/queue/priority_queue/
- For help with using a custom comparator with a priority queue: https://stackoverflow.com/questions/16111337/declaring-a-priority-queue-in-c-with-a-custom-comparator
- The Blind Search and Heuristic Search slides from lecture.
- For clarification on the A* algorithm: https://algorithmsinsight.wordpress.com/graph-theory-2/a-star-in-general/
- Further clarification on the A* algorithm: https://www.cs.princeton.edu/courses/archive/fall15/cos226/assignments/8puzzle.html
- For learning how to use the <map> data structure in C++: http://www.cplusplus.com/reference/map/map/
- For learning how to search for elements in the <map> data structure: http://www.cplusplus.com/reference/map/map/find/
- For learning how to insert an element into the <map> data structure: http://www.cplusplus.com/reference/map/map/insert/
- To learn how to utilize the calculation for the Manhattan distance heuristic for 8-puzzle: https://stackoverflow.com/questions/29470768/finding-target-coordinates-for-manhattan-distance-in-8-puzzle-c
- For generating random 8-puzzles: http://mypuzzle.org/sliding

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

- The function "abs" from <cstdlib>
- All subroutines from <vector> standard library
- The "priority_queue" data type from <queue> library
- All subroutines from the <map> standard library

```cpp
#include <iostream>
#include <vector>
#include <map>
#include <queue>
#include <cstdlib>
using namespace std;

int max_q = 0;
int num_expanded = 0;
int final_depth = 0;

struct Node
{
    int g_n;
    int h_n;
    int f_x;
    vector<vector<int> > state = {{0,0,0},{0,0,0},{0,0,0}};
};

struct Compare
{
    bool operator()(const Node& l, const Node& r) const
    {
        return l.f_x > r.f_x;
    }
};

int misplaced_tile(Node test, Node goal)
{
    int count = 0;

    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            if (test.state.at(i).at(j) != goal.state.at(i).at(j))
            {
                if (test.state[i][j] != 0)
                {
                    ++count;
                }
            }
        }
    }
    return count;
}

int manhattan_dist(Node test, Node goal)
{
    int dist = 0;
    int x = 0;
    int y = 0;
    int temp;
    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            if (test.state.at(i).at(j) != 0)
            {
                temp = test.state.at(i).at(j);
                x = abs(i - ((temp - 1) / 3));
                y = abs(j - ((temp - 1) % 3));
```

```
                x = abs(i - ((temp - 1) / 3));
                y = abs(j - ((temp - 1) % 3));
                dist += x + y;
            }
        }
    }
    return dist;
}

void print_board(Node ndptr)
{
    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            cout << ndptr.state.at(i).at(j) << ' ';
        }
        cout << endl;
    }
}

void expand(Node curr, priority_queue<Node, vector<Node>, Compare> & open,
map<vector<vector<int> >, bool> & visited, int heuristic)
{
    Node temp, temp2, temp3, temp4, goal;
    goal.state = {{1,2,3},{4,5,6},{7,8,0}};

    temp.g_n = curr.g_n + 1;
    temp2.g_n = curr.g_n + 1;
    temp3.g_n = curr.g_n + 1;
    temp4.g_n = curr.g_n + 1;

    temp.state = curr.state;
    temp2.state = curr.state;
    temp3.state = curr.state;
    temp4.state = curr.state;

    //find position of 0
    int x = 0;
    int y = 0;
    for (int i = 0; i < 3; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            if (curr.state.at(i).at(j) == 0)
            {
                x = i;
                y = j;
            }
        }
    }
    //check all moves

    //move right
    if (y < 2)
    {
        int temp_int = 0;
        temp_int = temp.state.at(x).at(y);
        temp.state.at(x).at(y) = temp.state.at(x).at(y + 1);
        temp.state.at(x).at(y + 1) = temp_int;

        if (heuristic == 1) //uniform cost search
```

```cpp
bool first_iteration = 1;

Node a_star(Node init, Node goal, int heuristic)
{
    priority_queue<Node, vector<Node>, Compare> open; //create queue
    open.push(init);

    map <vector<vector<int> >, bool> visited; //visited nodes
    visited.insert(pair<vector<vector<int> >,bool>(init.state, 0));

    while (1)
    {
        if (open.empty()) //IF EMPTY then return Failure
        {
            cout << "Could not find a solution." << endl;
            exit(0);
        }

        Node curr = open.top(); //REMOVE_FRONT
        open.pop();

        if (!first_iteration)
        {
            cout << "The best state to expand with a g(n) = "
                << curr.g_n << " and h(n) = " << curr.h_n << " is..." << endl;
            print_board(curr);
            cout << "Expanding this node..." << endl;
        }
        else
        {
            cout << "Expanding state: " << endl;
            print_board(curr);
            cout << endl;
            first_iteration = 0;
        }
        //print_board(curr);
        //cout << endl;

        if (max_q < open.size())
        {
            max_q = open.size();
        }
        //1=UCS, 2=MTH, 3=MDH
        if ((misplaced_tile(curr, goal) == 0) && (heuristic < 3)) //if GOAL state
        {
            final_depth = curr.g_n;
            return curr;
        }
        else if ((manhattan_dist(curr, goal) == 0) && (heuristic == 3))
        {
            final_depth = curr.g_n;
            return curr;
        }
        expand(curr, open, visited, heuristic);
    }
}

int main()
{
    Node start;
    start.g_n = 0;
    start.h_n = 0;
    start.f_x = 0;
```

379:18 C and C++

```
tshie001@cs_private:~/workspace/cs170 $ ./a.out
Welcome to Bertie Woosters 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.
Enter the first row, use space or tabs between numbers: 0 1 2
Enter the second row, use space or tabs between numbers: 4 5 3
Enter the third row, use space or tabs between numbers: 7 8 6
Enter your choice of algorithm.
1. Uniform Cost Search
2. A* with Misplaced Tile heuristic
3. A* with Manhattan Distance heuristic
2
Expanding state:
0 1 2
4 5 3
7 8 6

The best state to expand with a g(n) = 1 and h(n) = 3 is...
1 0 2
4 5 3
7 8 6
Expanding this node...
The best state to expand with a g(n) = 2 and h(n) = 2 is...
1 2 0
4 5 3
7 8 6
Expanding this node...
The best state to expand with a g(n) = 3 and h(n) = 1 is...
1 2 3
4 5 0
7 8 6
Expanding this node...
The best state to expand with a g(n) = 4 and h(n) = 0 is...
1 2 3
4 5 6
7 8 0
Expanding this node...

Goal!!
To solve this problem the search algorithm expanded a total of 7 nodes.
The maximum number of nodes in the queue at any one time was 3
The depth of the goal node was 4
tshie001@cs_private:~/workspace/cs170 $
```

CS170: Project 1 Write Up

Thomas Shiels, SID 861151979

Introduction

This purpose of this first project in Dr. Eamonn Keogh's Introduction to Artificial Intelligence course is to learn about solving the 8-puzzle game using a search tree, along with several different heuristics. This report will detail my findings in solving the game using Uniform Cost Search, Misplaced Tile, and Manhattan Distance heuristics. The programming language I used was C++, as this is the language I am most comfortable with.

Uniform Cost Search

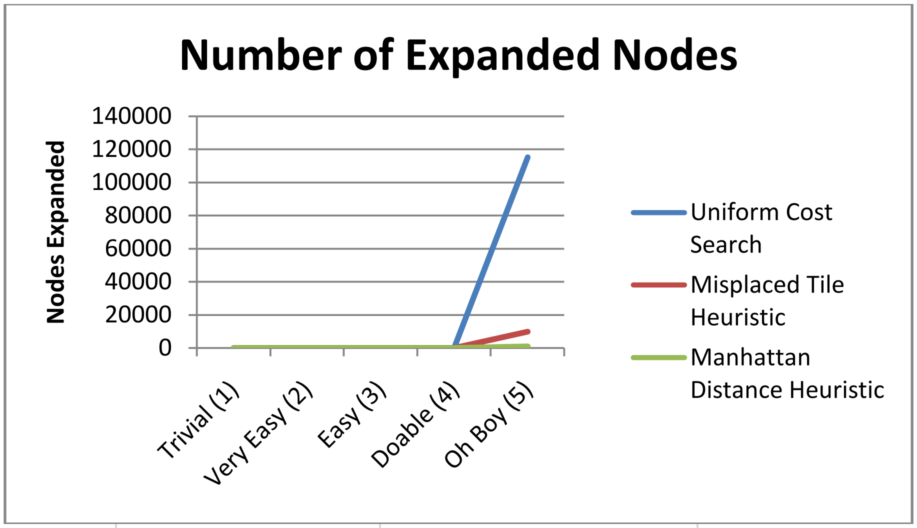
The Uniform Cost Search heuristic is the same as A*, with the only metric being the depth of the node in the search tree. In effect, h(n) becomes hardcoded to zero, and the search degenerates into Breadth First Search. There are no weights to expansions, and each node has a cost of 1.

Misplaced Tile Heuristic

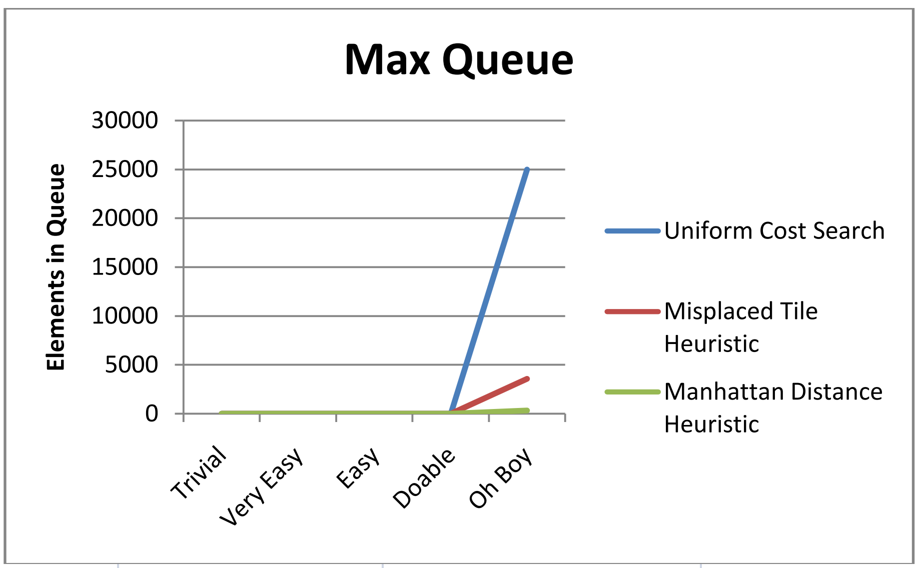
The first true heuristic in this project was the Misplaced Tile heuristic, which counts the number of tiles that are in the wrong position. This gives a vague idea of how close we are to solving the puzzle. The closer to solving the problem we are, the lower the score. As opposed to Uniform Cost Search, this heuristic is added to the cost of the current node, giving each edge of the tree a distinct weight. The nodes at a particular depth are inserted into the queue in ascending order of cost.

Manhattan Distance Heuristic

The Manhattan Distance Heuristic is the most sophisticated of the heuristics used in this assignment. It is similar to the Misplaced Tiles heuristic, but in addition to noting all misplaced tiles, it takes into account the distance from its correct position. The final output of the heuristic is the sum total of the distances of all the misplaced tiles.

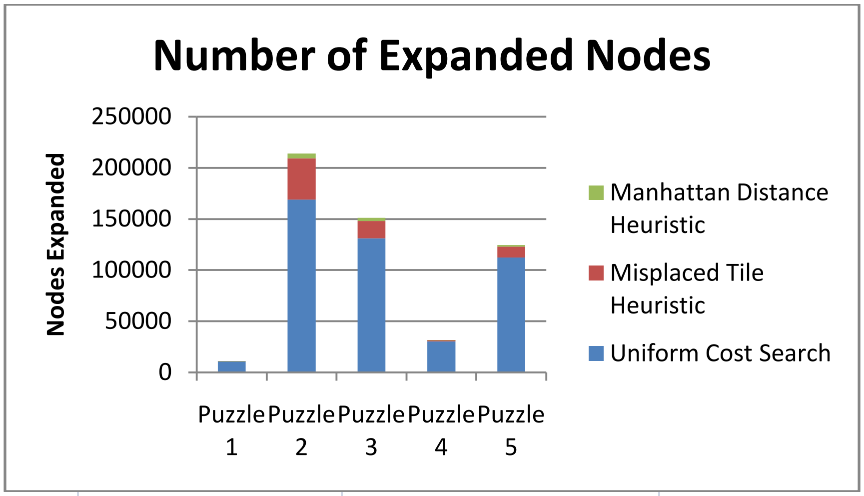


	Uniform Cost Search	Misplaced Tile Heuristic	Manhattan Distance Heuristic
Trivial (1)	0	0	0
Very Easy (2)	3	3	3
Easy (3)	10	4	4
Doable (4)	30	7	7
Oh Boy (5)	115133	9733	833

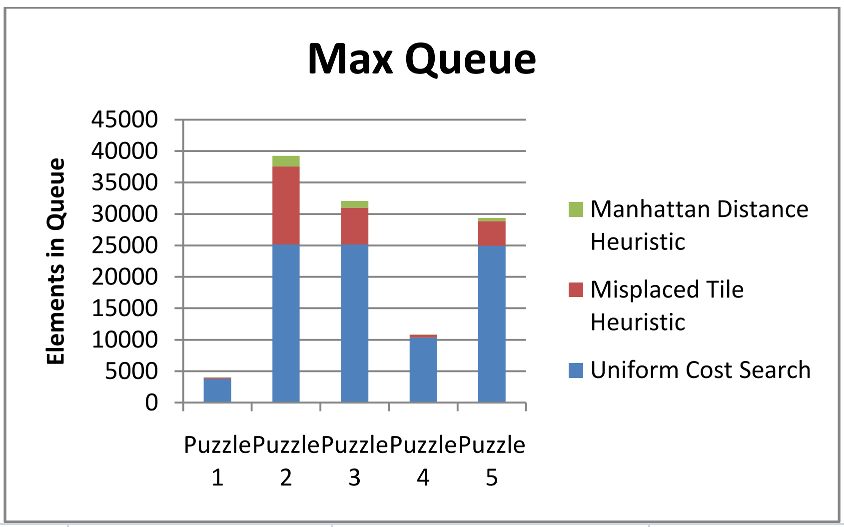


	Uniform Cost Search	Misplaced Tile Heuristic	Manhattan Distance Heuristic
Trivial	1	1	1
Very Easy	2	2	2
Easy	5	2	2
Doable	15	3	3
Oh Boy	24968	3581	312

Random Puzzles



	Uniform Cost Search	Misplaced Tile Heuristic	Manhattan Distance Heuristic
Puzzle 1	10509	484	68
Puzzle 2	168997	40218	4648
Puzzle 3	131235	16915	3013
Puzzle 4	30404	1171	117
Puzzle 5	112257	10759	1403



	Uniform Cost Search	Misplaced Tile Heuristic	Manhattan Distance Heuristic
Puzzle 1	3790	189	30
Puzzle 2	25164	12423	1655
Puzzle 3	25164	5840	1084
Puzzle 4	10351	454	49
Puzzle 5	24968	3884	520

Conclusion

There was a variety of difficulty in the puzzles, ranging from trivial to extremely difficult. The trivial puzzle was an already solved puzzle, where no nodes needed to be expanded, and where there was only ever one element in the queue at any given time. The Manhattan Distance heuristic was by far the most effective of all. In the more difficult puzzles it made most of the difference, solving the puzzle faster by orders of magnitude. The second most effective heuristic was the Misplaced Tiles heuristic. Uniform Cost Search performed the worst by far with having no way to discern a more optimal path. While a heuristic can provide a huge benefit, it is important to make sure your heuristic is a good fit for the problem at hand.