CS-213 Lab assignment 6

Due date: 15th April, 2016

In this assignment, you will program one of the classic AI algorithms. This algorithm is called the A* algorithm. The algorithm is a search algorithm. A search algorithm finds a path to the goal, given a starting point and what the goal is.

Consider a graph, with one node marked as source and one node marked as goal. The graph is a weighted graph, and we want to find the shortest path to the goal. The simplest strategy is to implement Dijkstra's algorithm for shortest paths. One disadvantage of that is it will explore a lot of paths to get the shortest path to the goal. A* algorithm prevents exploration of the entire graph.

As you know, nothing in the world comes for free. A* needs something more to achieve this reduction in the number of edges/nodes explored. The magic ingredient is the following: For each node, A* requires an estimated distance to the goal. It uses this estimated distance to prune out certain nodes that need not be explored. This estimate is called the heuristic value.

A* works the following way:

1. Two sets called open set and closed set. Open set has the nodes that need to be explored, closed set has the nodes that have already been explored.
2. Start with the starting node in the open set.
3. Pick the node i from the open set with the smallest value of f(i) + h(i), where f(i) is the path length from the starting node to node i.
4. Put this node in the closed set, put all its neighbors which are not in the closed set in the open set. If a neighbor was already present in the open set, just update its f value to the smaller one.
5. Repeat from step 3 until the goal node is picked up from the open set.
6. The f value of the goal node is the length of the shortest path.

For A* to work on a graph, for each node i, we need the value of h(i), and h(i) must have the following properties:

1. h(i) <= G(i), where G(i) is the actual shortest path distance to the goal from node i. This means that the heuristic must never over-estimate the distance to the goal
2. For two adjacent nodes i and j, h(i) <= d(i,j) + h(j).
   This is called the monotone restriction. What this does is it guarantees that if a node is found, we have found the shortest path to it and it need not be explored again.

If the heuristic is 0, the algorithm is nothing but Dijkstra's shortest path algorithm.

Note that the algorithm needs the following information:

1. The heuristic value for each node
2. Keeping track of f values
3. The children and edge weights for a node
4. Comparison of nodes for equality

This allows the algorithm to be generic to solve a large class of problems with different implementations of the node.
Write a templated A* algorithm or use object oriented principles (inheritence, subclass, superclass) to write a generic A* that can solve different kinds of problems based on the kind of nodes that are passed to it. We will be solving two different problems using A*. Divide the code in the following manner:

1. A_star.cpp and A_star.hpp - These will be the generic A* implementation
2. prob1.cpp - All the structures and code needed to solve problem 1. It has to use A_star.hpp
3. prob2.cpp - All the structures and code needed to solve problem 2. It has to use A_star.hpp

Problem 1

You are given two strings, A and B. Starting from A, what is the shortest number of operations to transform it to B?
The operation allowed is swapping of two characters

Eg.
A = APPLE
B = LEPPA

One possible way is:
APPLE -> LPPAE -> LEPAP -> LEPPA

And it turns out that this is the shortest way to do it (3 turns)

Input:
A and B on two separate lines
Eg.

APPLE
LEPPA
Output:
The minimum number of moves
Eg.
3

Problem 2

The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. The goal is to rearrange the blocks so that they are in order. It is permitted to slide blocks horizontally or vertically into the blank square. The following shows a sequence of legal moves from an initial board position (left) to the goal position (right).

```
    1  3        1     3        1  2  3        1  2  3        1  2  3
 4  2  5   =>   4  2  5   =>   4     5   =>   4  5     =>   4  5  6
 7  8  6        7  8  6        7  8  6        7  8  6        7  8

 initial                                                    goal
```

Given an initial arrangement of the tiles, you have to find the shortest number of moves to get to the goal.

Input: Label the positions from 1 to 9 as follows

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

The input will be 9 characters, separated by whitespace, with the characters 1 to 8 denoting the tiles and 0 denoting the empty tile.

Eg.
0 1 3
4 2 5
7 8 6

Output
The minimum number of moves needed to get to the goal state. If not possible, output -1.