Homework 4 - Lyell Read

Question 1:

The Greedy algorithm to solve this question will simply examine all the n hotels. For each day that we are not at the end, choose the furthest away hotel, or the end if it is within reach (of driving time during the day) and go to that location. In that way, the algorithm takes Theta(n) time, as all it has to do is examine the hotels available once.

Question 2: 16.1-2:

[Background]: Give a dynamic-programming algorithm for the activity-selection problem, based on recurrence (16.2). Have your algorithm compute the sizes c[i, j] as defined above and also produce the maximum-size subset of mutually compatible activities. Assume that the inputs have been sorted as in equation (16.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal Solution.

This algorithm is analogous to GREEDY-ACTIVITY-SELECTOR, as when time runs backward, this modified algorithm runs the same as GREEDY-ACTIVITY-SELECTOR when time is moving forwards. When running in reverse, the last to start is the first to finish in forward speed. Therefore, this algorithm can be proven to yield an optimal result because the forward version yielded this optimal result.

This algorithm is greedy as it makes the choice that looks best at each step, whether it is going forwards or backwards.

Question 3:

Consider an undirected graph G=(V,E) with nonnegative edge weights w(u,v) >= 0. Suppose that you have computed a minimum spanning tree T, and that you have also computed shortest paths to all vertices from vertex s∈V. Suppose each edge weight is increased by 1: the new weights w'(u,v) = w(u,v) + 1.

(a) Does the minimum spanning tree change? Give an example if it changes or prove it cannot change.

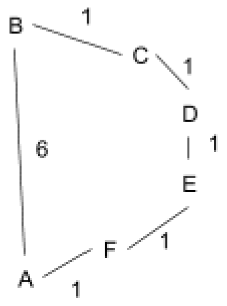
It will not change. If we consider Kruskal's algorithm in light of the changes detailed above.

1. Sort all edges by increasing weight. Adding one to each weight will never change this step, as they will be sorted in the same order.
2. Pick the shortest edge, and add it to the list of taken edges iff it does not form a cycle. The process of determining if something makes a cycle does not consider weight at all, therefore increases or decreases of weight do not factor in here, thus nothing is changed.
3. Repeat 2 while number of edges taken < number of vertices - 1.

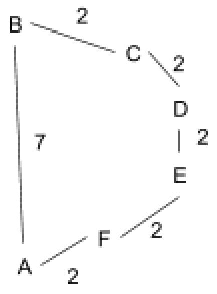
Given that none of these steps are altered at all by increasing all weights by 1, the resulting set of edges is not altered either. The only thing that will change is that the total weight of chosen edges, which will increase by 1*(V-1) where V is the number of vertices.

(b) Do the shortest paths change? Give an example of a graph where the paths change or prove that the paths cannot change.

This does indeed change. We will consider just the change in path from start vertex (s above, B here) to A. Other vertex optimal paths could change too. Consider the following graph:



The optimal path from B to A is BC, CD, DE, EF, FA with weight 5. When the lengths are increased by 1, the most optimal path from B to A changes:



Now, after the change, the best path is AB, with weight 7, as opposed to the old best path BC, CD, DE, EF, FA with weight 10.

Question 4:

A verbal description of your algorithm and data structures

My algorithm works with two primitive data structures - the integer and the list (for the most part, printing and such relies on other structures). It operates by first reading the file, then applying Kruskal Greedy Algorithm to the data it is provided. This algorithm consists of two parts, the first is a driver, that ensures that edges are continually chosen from the top of the (sorted) list of edges until #edges = #vertices-1. The other part consists of a checking algorithm. This algorithm checks each candidate edge, and recursively traces it through the graph to ensure there is no way that, for segment AB, you can get from A to B in the currently selected edges (as this would create a cycle).

The pseudo-code for your algorithm.

```
check_points (a, b, taken_edges):
    ret = 0
    path_exists = 0
    if [a, b, 1] in taken_edges or
        [b, a, 1] in taken_edges or
        [a, b, 0] in taken_edges or
        [b, a, 0] in taken_edges:

        return 1
    for x in taken_edges:
        if x[0] == a and x[2] == 0:
            bak_x = x
            for y in taken_edges:
                if y[0] == x[0] and y[1] == x[1]:
                    y[2] = 1
                if y[1] == x[0] and y[0] == x[1]:
                    y[2] = 1

            ret += check_points(bak_x[1], b, taken_edges)
            path_exists += 1

    if path_exists == 0:
        return 0
    else:
        return ret

check_edge(current_edge, taken_edges):
    taken_edges_bidirectional = []

    for x in taken_edges:
        taken_edges_bidirectional.append([x[0], x[1], 0])
```

```
            taken_edges_bidirectional.append([x[1], x[0], 0])
        return check_points(current_edge[0], current_edge[1],
    taken_edges_bidirectional)


    mst(c, e):

        taken_edges = []
        edge_index = 0
        while len(taken_edges) < len(c)-1:

            current_edge = e[edge_index]
            edge_index += 1

            if check_edge(current_edge, taken_edges) != 1:
                taken_edges.append(current_edge)

        return taken_edges
```

Analysis of the theoretical running time of your algorithm.

Theoretical running time of each function above:

- check_points: At worst n^2, at best n^0 == 1, so on average, about n
- check_edge: n
- mst: n

mst() calls check_edge() once per iteration, so check_edge() runs n times. check_points() is called by check_edge(), so check_points gets called n^2 times. Each time it is called, it runs in about n time. Therefore the theoretical run time of this algorithm is Theta(n^3). Not great, but that's because I chose to write my algorithm using a homemade cycle check, as opposed to DFT or similar search (not compatible with my data structures used.