22C:44 Homework 8 Solution

1. Order the intervals by the *left endpoints* and process them in this order. Maintain a collection of sets $C_1, C_2, \ldots$ each initialized to the empty set. Insert each interval x into a set C_i with smallest i such that x is compatible with every interval in C_i.

   Note: This is very similar to the incorrect greedy algorithm proposed in Problem 1 of HW7. The only difference is that intervals are processed by left endpoint order rather than right endpoint order.

   Here is the proof of correctness of the algorithm. Suppose that the solution produced by the algorithm is $C_1, C_2, \ldots, C_k$ for some integer $k \geq 1$. Consider an interval $a_{i_k} \in C_k$. The fact that a_{i_k} is in C_k implies that when it was inserted into C_k, each set C_j, $1 \leq j \leq k-1$, had an interval a_{i_j} that a_{i_k} was incompatible with. Furthermore, each interval a_{i_j}, $1 \leq j \leq k$, starts *before* a_{i_k} and therefore each of these intervals passes through the left endpoint, $\ell(i_k)$ of a_{i_k}. This implies that the intervals $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ are all mutually incompatible. *Any* solution to the problem has to assign these k intervals to different sets and therefore any solution to the problem is of size at least k. This implies that any solution of size k is optimal.

2. The greedy algorithm for this problem repeatedly finds the leftmost point x not yet covered and adds the interval $[x, x+1]$ to the solution. The algorithm can be implemented to run in $\Theta(n)$ time, as shown in the following pseudocode. In the pseudocode x keeps track of the left endpoint of the interval most recently added to the solution I.

   Sort the points and assume that $x_1 < x_2 < \cdots < x_n$;
   $I \leftarrow \{[x_1, x_1+1]\}$;
   $x \leftarrow x_1$;
   for $i \leftarrow 2$ **to** n **do**
       if $(x_i > x+1)$ **then** {
           $I \leftarrow I \cup \{[x_i, x_i+1]\}$;
           $x \leftarrow x_i$;
       }

   The proof of correctness of this algorithm is as follows. Let $G = \{[a_1, a_1+1], [a_2, a_2+1], \ldots, [a_p, a_p+1]\}$ be the greedy solution produced by the above algorithm. Let $O = \{[b_1, b_1+1], [b_2, b_2+1], \ldots, [b_q, b_q+1]\}$ be an optimal solution that shares most intervals with G. Without loss of generality, assume that $a_1 < a_2 < \cdots < a_p$ and $b_1 < b_2 < \cdots < b_q$. To obtain a contradiction, suppose that G is not optimal. This implies that $p > q$. Let t be the smallest integer such that $[a_i, a_i+1] = [b_i, b_i+1]$ for all i, $1 \leq i \leq t-1$ and $[a_t, a_t+1] \neq [b_t, b_t+1]$. If $b_t < a_t$ then replacing $[b_t, b_t+1]$ by $[a_t, a_t+1]$ in O yields yet another optimal solution and this has more intervals in common with G - a contradiction. If $b_t > a_t$ then there is an input point at a_t that is not covered by *any* interval in O - again a contradiction.

3. See Problem 1 in practice problem set HW9 from fall 2000.

4. Consider a shortest path tree T. Note that this has at most $|V|-1$ edges. To start, call the vertex s *relaxed*. Pick an edge from a relaxed vertex to its not-yet-relaxed child and relax that

edge. Once this edge is relaxed, call the child relaxed as well. Repeat this until all vertices are relaxed. It is easy to see that when a vertex v is relaxed, $d[v] = distance(s, v)$. Note that any vertex v that is not reachable from s satisfies $d[v] = distance(s, v) = \infty$. Therefore, all vertices have correct d-values as the result of at most $|V| - 1$ relax operations.

5.
```
BELLMAN-FORD(G, s)
    INITIALIZE(G, s);
    for i ← 1 to |V| − 1 do
        for each edge (u, v) ∈ E do
            RELAX(u, v);

    for each edge (u, v) ∈ E do
        if (d[v] > d[u] + w(u, v)) then
            d[u] ← −∞;

    for i ← 1 to |V| − 1 do
        for each edge (u, v) ∈ E do
            RELAX(u, v);
```

The idea behind the above algorithm is based on a claim we showed in class that told us that there is at least one edge in every negative cycle that is not relaxed at the end of the $|V| - 1$ stages. Therefore, the middle block of code assigns to at least one vertex in each cycle the value $-\infty$. The third block of code is responsible for getting these $-\infty$-values to propagate through the graph.