CS 161
Spring 2017

Problem Set 7 with Solutions
Due: June 2, 2017, 3pm

Please answer each of the following problems. Refer to the course webpage for the **collaboration policy,** as well as for **helpful advice** for how to write up your solutions.

Note: For all problems, if you include pseudocode in your solution, please also include a brief English description of what the pseudocode does.

1. **True or False? (3 points)** Decide whether the following statement is true or false. If it is true, give a short justification. If it is false, give a counterexample.

   Let $G = (V, E)$ be an arbitrary connected weighted undirected graph with no self-loops (that is, no edges from a vertex to itself), so that the weight of an edge $e \in E$ is $w(e)$, and all the weights are distinct. Let $e^* \in E$ be the cheapest edge: that is, the edge so that $w(e^*) = \min_{e \in E} w(e)$. Then there is a minimum spanning tree T of G that contains e^*.

   [We are expecting: your true/false answer, and either a justification (a few sentences) or a counterexample (a graph).]

   SOLUTION: The statement is true.

   Solution 1. Suppose that T is a MST that does not contain e^*. Then adding e^* to T will create a cycle. Remove any other edge from this cycle (after adding e^*). Just as we saw in class, this will result in a new spanning tree with cost that's at most the cost of T. Since T was an MST, this new tree is also an MST that contains e^*.

   Solution 2. We can appeal to the proof that Kruskal's algorithm works: Kruskal's algorithm will always return a tree that includes the cheapest edge, and we proved in class that it always returns a minimum spanning tree.

2. **Greedy algorithms don't always work. (3 points)** Let $G = (V, E)$ be an undirected unweighted graph. Say that a vertex v can "see" an edge if v is an endpoint of that edge. Say that a set $S \subseteq V$ is "all-seeing" if every edge $e \in E$ is seen by at least one vertex in S. In this problem, we will try to greedily construct the smallest all-seeing set possible.

   Consider the following greedy algorithm to find an all-seeing subset:

```
findAllSeeingSubset(G):
  S = {}
  while G contains edges:
    choose an edge e = (u,v) in G
    S.add(u)
    S.add(v)
    remove u and all of its adjacent edges from G
    remove v and all of its adjacent edges from G
  return S
```

   (a) (1 pt) Prove that `findAllSeeingSubset(G)` always returns an all-seeing subset.
   [We are expecting: A short but rigorous proof (a few sentences).]

   (b) (2 pts) Give an example of a graph on which `findAllSeeingSubset(G)` does not return a smallest all-seeing subset, for at least one way of choosing edges.
   [We are expecting: your example, and a brief justification that it is an example of this.]

(c) (0 pts) **[BONUS: this question is NOT REQUIRED but it might be fun to think about. It will not affect your grade.]**

   i. Can you find a greedy algorithm that does find a smallest all-seeing subset?

   ii. `findAllSeeingSubset` has a very nice property. Let $OPT(G)$ denote the size of the smallest all-seeing subset in G. Prove that for all graphs G,

$$\frac{|\texttt{findAllSeeingSubset}(G)|}{OPT(G)} \leq 2.$$

[We are expecting: Nothing, this problem is optional.]

SOLUTION:

(a) An edge is removed only if one or both of its endpoints are placed in S, which means that it is seen. Since we have no edges left at the termination of `findAllSeeingSubset`, all edges must have been seen, and so S is all-seeing.

(b) Consider the graph that is a square on the vertices A, B, C, D. The smallest all-seeing set has size 2, but `findAllSeeingSet` will return a set of size 4.

3. **Transportation Networks. (5 points)** Given a set of n cities, we would like to build a transportation system such that there is some path from any city to any other city. There are two ways to travel: by driving or by flying. Initially all of the cities are disconnected. It costs $c_{i,j}$ to build a road between city i and city j. It costs a_i to build an airport in city i. For any two cities i and j, we can fly directly from i to j if there is an airport in both cities. Give an efficient algorithm for determining which roads and airports to build to minimize the cost of connecting the cities. Here, "connecting the cities" means that there should be some way to get from any city to any other.

You algorithm should take as input the costs $c_{i,j}$ and a_i, and return a list of roads and airports to build. It should run in time $O(m \log(n))$.

[**We are expecting: pseudocode and a short informal explanation of why it is correct. You may (and, hint, you may wish to) call any algorithm we have seen in class.**]

SOLUTION:

Consider a graph G constructed as follows. We have a vertex $v_i \in V$ for $i = 1, \ldots, n$, one for each of the n cities, and there is an edge between v_i and v_j with cost $c_{i,j}$. Consider also the graph G_{sky}, which is the same as G and where we additionally add a vertex `sky` $\in V$ representing the sky. We add an edge between v_i and `sky` with cost a_i. Now our algorithm is:

```
planTransit(costs c_{i,j}, a_i):
     generate the graph G as above
     T = KRUSKAL(G) // or any MST algorithm
     T_sky = KRUSKAL( G_sky )
     if cost(T) < cost(T_sky):
          T_actual = T
     else:
          T_actual = T_sky
     airports = []
     roads = []
     for i = 1,...n:
          if {v_i, air} is an edge in T_actual:
               airports.add(i)
          for j =1,...,n:
              if {v_i, v_j} is an edge in T_actual:
                roads.add( {i,j} )
    return airports, roads
```

Next, we argue that this is correct. There are two cases: either the optimal solution uses an airport, or it does not. If the optimal solution does not include an airport, then the optimal solution is given by an MST on G; that is, the minimum cost solution using only roads. In contrast, if the optimal solution does use an airport, then the cost is given by an MST on G_{air}. The pseudocode above returns the better of the two.

4. **Placing receivers. (8 points)** Suppose there are n transmitters fixed in place along a linear track. The i'th transmitter has communication range $[a_i, b_i]$, for $a_i \leq b_i$. That is, any receiver placed within the range $[a_i, b_i]$ can receive signals from the i'th transmitter. Assume that the transmitters are sorted by the right endpoint of their communication range: that is, if $i < j$, then $b_i \leq b_j$.

   We want to pick a set of points on the track to place receivers such that we can receive signals from every transmitter while minimizing the number of receivers necessary. That is, we want to find a minimum set of points S on the line such that for every i, there is some $s \in S$ such that $a_i \leq s \leq b_i$.

   In this problem, we will design an algorithm that finds the minimum set S in expected time $O(n)$, and prove that it is correct.

   Consider the greedy algorithm which works as follows: we place receivers one at a time. At each step, suppose that i^* is the smallest i so that transmitter i cannot be heard by any receiver placed so far, and place a receiver at b_{i^*}. Continue placing receivers in this way until all the transmitters can be heard.

   (a) (2 points) Based on this English description, write pseudocode to implement the algorithm in time $O(n)$.

   [We are expecting: detailed pseudocode, and an informal justification of the running time.]

   (b) (6 points) Prove that this algorithm is correct, following the outline below. We will use induction on t, where the inductive hypothesis should be something like the following: after we have processed t transmitters and have a set S of receivers, there is an optimal set S^* of receiver locations so that $S \subseteq S^*$.

      i. (1 points) Formalize the inductive hypothesis that is described above. (The details of how you state it may depend on how you've written your pseudocode in part (a); if the statement above makes sense with the pseudocode you have written and the rest of your argument, you may just repeat it).

      ii. (1 point) Prove the base case.

      iii. (3 points) Prove the inductive step.

      iv. (1 point) Finish the argument. That is, once the induction argument is complete, show that this implies that the algorithm is correct.

   [We are expecting: for i, a statement of an inductive hypothesis. For ii,iii,iv, we are expecting a formal proof, including a statement of what it is you are proving.]

Solution:

(a) The pseudocode is:

```
placeReceivers(a,b): \\ a and b are arrays of a_i and b_i
     biggestReceiver = -Infinity
     receivers = []
     for i = 1,...,n:
          if a[i] <= biggestReceiver and biggestReceiver <= b[i]:
               continue
          else:
                receivers.append(b[i])
                biggestReceiver = b[i]
     return receivers
```

This runs in time $O(n)$ because the outer loop is over n things, and inside this loop we do $O(1)$ work to update the list `receivers` and the variable `biggestReceiver`.

i. The inductive hypothesis is: after the i'the iteration of the for loop, the set `receivers` is contained in an optimal set S^*.

ii. The base case is when $i = 0$. In this case, the inductive hypothesis says that after 0 iterations of the for loop, the set `receivers` is contained in an optimal set S^*. Since `receivers` is empty at this point, the base case holds trivially.

iii. To prove the inductive step, we must show that if the base case holds for $0 \leq i \leq n-1$, then it holds for $i+1$. Suppose that after the i'th iteration of the for loop, the set `receivers` is contained in an optimal set S^*.
Now in the $i+1$'st iteration, two things could happen. The first is that transmitter $i+1$ is already covered, in which case `receivers` does not change, and so the inductive hypothesis holds at step $i+1$.
On the other hand, suppose that transmitter $i+1$ is not covered already. Since `receivers` does not cover transmitter $i+1$, there must be some $p \in S^* \setminus$ `receivers` that covers transmitter $i+1$. Let p be the smallest such element. Notice that this implies that $p \leq b_{i+1}$, since $p \in [a_{i+1}, b_{i+1}]$ covers b_{i+1}.
Consider the set $S' = S^* \cup \{b_{i+1}\} \setminus \{p\}$. Notice that the size of S' is the same as the size of S^*.
Claim: S' covers all of the transmitters. The only intervals we need to worry about are those that are covered by p and by no other point in S^*. Suppose that $[a_k, b_k]$ is covered by p and by no other point in S^*. We need to show that $a_k \leq b_{i+1} \leq b_k$, because then this interval will be covered by the new transmitter at b_{i+1}.
   - First, since $p \leq b_{i+1}$ and p covers $[a_k, b_k]$, we have $a_k \leq p \leq b_{i+1}$.
   - Second, for all $j < i+1$, $[a_j, b_j]$ is covered by `receivers` by construction. In particular, $[a_j, b_j]$ is covered by something other than p, so we must have $k \geq i+1$. But then we have $b_k \geq b_{i+1}$, since they are sorted.

   Together, these two points establish that $a_k \leq b_{i+1} \leq b_k$, which is what we needed to show. This completes the proof of the **Claim**.
Thus, S' is also an optimal set of covering transmitter locations, and $b_{i+1} \in S'$. This establishes the inductive hypothesis for $i+1$.

iv. Finally, at for $i = n$, the inductive hypothesis reads: after the n'th iteration of the for loop, the set `receivers` is contained in an optimal set S^*. Since by construction `receivers` covers all of the transmitters, and it contained in S^*, we have $|$`receivers`$| \leq |S^*|$, and hence `receivers` also has optimal size.

5. **Well-connected subgraphs. (10 points)** Given an undirected, unweighted graph $G = (V, E)$ and a set of vertices $S \subseteq V$, we say that the **subgraph of G induced by S** is the graph $G' = (S, E')$, where the vertex set is S, and the edge set is

$$E' = \{(u, v) \in E \, : \, u \in S, v \in S\}.$$

That is, the induced subgraph is formed by keeping all the vertices in S, and keeping all the edges with both endpoints in S.

We seek an algorithm to solve the following problem:

- **Input:** A graph $G = (V, E)$ and an integer k
- **Output:** A set $S \subseteq V$ so that every vertex in the subgraph $G' = (S, E')$ induced by S has degree at least k in G', so that S is as large as possible given that it has this property, if such a set exists. If no such set exists, return "`Does not exist.`"

(a) (4 pts) Give pseudocode for an efficient greedy algorithm that solves the problem above. Your algorithm should run in expected time $O((m+n)^2)$ if $m = |E|$ and $n = |V|$. (Note: It is possible to come up with an algorithm that runs faster than this, but anything no worse than this running time is fine for full credit). You may assume that G is stored using the adjacency-list representation.

[**We are expecting: pseudocode, and a short informal analysis of the runtime.**]

(b) (6 pts) Prove formally that your algorithm is correct.

[**We are expecting: a formal proof. If you use induction, make sure you explicitly state what your inductive hypothesis is, and make sure that you explicitly conclude that your algorithm is correct.**]

SOLUTION:

(a) We will use the following greedy algorithm:

```
# This code assumes the vertices are represented by ids ranging from 0 to n - 1.
# If we cannot assume this, we may maintain two dictionaries: one that maps
# vertices to ids, and the other that maps ids to vertices, and use these
# to convert between vertices and ids as needed in O(1) expected time.

# This code also assumes that the adjacency list is given as a list of lists,
# e.g. adj = [[1, 2, 3], [1, 2], [0], []] would represent the graph
# 0 -> 1, 2, 3
# 1 -> 1, 2
# 2 -> 0
# 3 -> no outgoing neighbors

import sys

def kcore(k, n, adj):
    # Populate the array of vertex degrees
    D = [0] * n    # O(n)
    # If len is an O(1) operation, this costs O(n) time;
    # otherwise, it costs O(m + n).
    for src in xrange(n):
        D[src] = len(adj[src])

    # Note that sets are implemented with hashing, so set operations
    # take expected O(1) time.
    S = set(range(n))

    # Runs at most O(n) times.  If len(S) is not an O(1) operation,
    # we may maintain an auxiliary variable S_size that gets
    # decremented each time an element is removed from S, and use
    # that in the while loop instead.
    while len(S) > 0:
        # Find vertex with smallest degree that is still in S (O(n)).
        minDegree = sys.maxint
        v = None
        for i in S:
            if D[i] < minDegree:
                minDegree = D[i]
                v = i

        if D[v] >= k:
            # We are done
            return S
        else:
            # Remove the vertex of smallest degree
            S.remove(v)
            for u in adj[v]:   # O(m) across all iterations of for loop
                D[u] -= 1
    return S


# Consider the graph
# 0 -- 1
# 0 -- 2
# 1 -- 2
# 2 -- 3
# The k-core should be {0, 1, 2}.

print "Expected result: {0, 1, 2}"
print kcore(k = 2, n = 4, adj = [[1, 2], [0, 2], [0, 1, 3], [2]])

# Consider the graph at
# http://www.geeksforgeeks.org/find-k-cores-graph/
print "Expected result: {2, 3, 4, 6, 7}"
print kcore(k = 3, n = 9, adj = [[1, 2], [0, 2, 5], [0, 1, 3, 4, 5, 6], [2, 4,
6, 7],
[2, 3, 6, 7], [1, 2, 6, 8], [2, 3, 4, 5, 7, 8], [3, 4, 6], [5, 6]])

print "Expected result: {0, 1}"
print kcore(k = 2, n = 2, adj = [[0, 1], [0, 1]])
```

Output:

```
Jessicas-MBP:cs161-spring-2017 jessica$ python q5.py
Expected result: {0, 1, 2}
set([0, 1, 2])
Expected result: {2, 3, 4, 6, 7}
set([2, 3, 4, 6, 7])
Expected result: {0, 1}
set([0, 1])
```

This algorithm takes time $O(m+n^2)$: the while loop runs for at most n iterations. In each iteration, we take time $O(n)$ to find the vertex with the lowest degree. Then we take $O(\deg(v)) + O(1)$ time to remove all of the edges connected to v. This adds up to

$$\sum_{v \in V} O(n) + O(\deg(v)) + O(1) = O(n^2 + m) = O(n^2).$$

Note that if you put the degrees in an ordinary heap, rather than an array, it costs $O(n \log n)$ time to extract n minimal elements, and across all iterations of the while loop, it costs $O(m \log n)$ time to decrease the degrees while maintaining the heap invariants. This would give you a runtime of $O((n+m)\log n)$. For dense graphs, this is worse than our $O(n^2)$ bound, but many graphs in the real world are sparse, and for such graphs, $O((n+m)\log n)$ would be a substantially better runtime. One application of k-cores is detecting communities in social networks, which tend to be sparse graphs, since most people are not friends with most other people.

(b) Now we prove formally that this works. Say that a set S is k-good if the induced graph has degree at least k. Say that at interation t of the while loop, the vertex set is V_t and the edge set is E_t.

We argue by induction, with the following inductive hypothesis:

Inductive Hypothesis: After t iterations of the algorithm, there is a maximal k-good set S contained in the vertex set V_t.

For the **base case**, we plug in $t = 0$ to the inductive hypothesis, and it reads "after 0 iterations, there is a maximal k-good set S contained in the vertex set." This is true, since after 0 iterations, the vertex set is just V, the original vertex set.

For the **inductive step,** we have to show that if the inductive hypothesis holds for t, then it holds for $t+1$. Suppose that after t iterations, there is a maximal k-good set S contained in V_t. Let v be the vertex we remove. It has degree less than k, so it can't be in S. Thus $S \subseteq V_t \setminus \{v\} = V_{t+1}$. Thus, after $t+1$ iterations, there is a maximal k-good set S contained in V_{t+1}.

Finally, for the **conclusion,** suppose that the while loop stops after t^* iterations. Then the inductive hypothesis implies that there is a maximal k-good set S contained in V_{t^*}. But V_{t^*} is itself k-good (because we could not remove any more vertices). So V_{t^*} must be a maximal k-good set.