Homework 5 - Solutions

Deadline: 1:30pm, May 6, 2013
Available points: 100. Perfect score: 100.

Problem 1 (30 points):

a) Provide an efficient algorithm that returns a classification of your customer base into d communities that achieves the minimum ics value over all possible classifications into d communities.

We can answer this problem using a modified version of Kruskal's Algorithm to perform clustering of the customers. There are two key differences, the first is that we prematurely stop the execution in order to return disconnected clusters rather than a single MST. To return d clusters, we simply omit the *last* $d-1$ edges which would be added by Kruskal's algorithm. The second difference is instead of sorting edges by increasing weight, we will sort by decreasing weight, where the weights are the similarity measure.

b) What data structures do you assume for the implementation of your algorithm? What is the running time of your approach given the data structures you employ?

It is assumed that all pairs of customers have a similarity score computed for them, so this data would likely be kept in an adjacency matrix. We need to sort these values and take the top $|V|-d+1$ from this, so we would employ a priority queue. Furthermore, we use a disjoint sets data structure to keep track of which components each of the vertices is in. Thus, the total running time is dominated by the sorting, yielding a running time of $O(E \log E)$ in the general case.

c) Prove the correctness of your algorithm.

We will prove this by showing that the maximum similarity between clusters is provided by this algorithm as opposed to any other reduced spanning tree. The $(d-1)^{th}$ edge in the optimal 'MST' represents the maximum similarity between any two clusters produced, and call this similarity s^*. We must show that the $(d-1)^{th}$ edge in any other spanning tree will have larger similarity.

Consider any other spanning tree, call this T'. It must be that there exist two points that are in the same cluster in the original tree T but are in different clusters in T', call them i and j, and the similarity between i and j is s_{ij}. It follows then that along a path from i to j in T', we must jump from one cluster to another along some edge, and call this edge's similarity value s^p. We know also that all edges in the path containing this edge must have higher similarity than s^*. It follows then that $s^p \geq s^*$, completing the proof.

Problem 2 (20 points):

In order to show that this problem is NP-Complete, we must show both that an answer to the problem is verifiable in polynomial time, and that given an efficient solution to this problem, we can solve some other NP-complete problem efficiently. In order to verify such a solution, we simply need to perform a search over a given answer. We perform the search, and make sure to check for the given criteria (i.e., there are no cycles and no vertex has degree greater than d.).

It turns out that we can efficiently solve the Rudrata (Hamiltonian) Path problem efficiently, given our efficient solution to this problem. Let's call the algorithmic solution to the given problem as A. Given the graph G we will run A with input parameter $d=2$ and with the weights on the graph edges increased by 1. When running A with $d=2$, the spanning tree we would return is actually a

path, and it turns out that this path should tour all of the vertices in G. Also, should A return that no tree exists, then we can use this information to report that no Rudrata Path exists. Therefore, by showing that this algorithm lets us solve a known NP-Complete problem efficiently and that it is verifiable in polynomial time, it shows that this problem is NP-Complete.

Some intuition on why simply running a MST algorithm is insufficient to solve this problem. The complication comes from the first constraint in that simply running a spanning tree algorithm over the graph G will possibly return a tree with some node having degree larger than d. In fact, in order for the algorithm to report that no such tree G' exists, it must be that the algorithm has somehow reasoned over all possible spanning trees over G, in order to know that no such tree exists.

Problem 3 (25 points):

(a) Provide a polynomial time algorithm that solves Julie's problem exactly when Jack has only two armies.

When Jack has only two armies, we can treat this problem as finding a minimum $s-t$ cut, where we separate the two cities containing Jack's armies into separate connected components. Solving this problem can be done in $O(VE^2)$ time.

(b) Provide a solution that achieves an approximation ratio of at most 2 *when Jack has three armies.*

We will simply do the intuitive approach, where we use the minimal cut algorithm twice in succession. After computing the first minimal cut, we know that one of the two components created will have two of the cities with army in it. We then run the minimal cut algorithm over that connected component.

We argue that this method will achieve an approximation ratio of 2. We know there exists an optimal cut, call it E^*, which separates the graph into three connected components. E^* can be defined as the union of three edge sets, E_i, which collectively remove each army a_i. We know that we have created minimal cuts in both steps. Next, define a function, δ, such that $\delta(V_i)$ is the set of all edges exiting or leaving the component which contains army a_i. Then, we claim that $w(E_i) \leq w(\delta(V_i)),\ \forall\ 1 \leq i \leq k$, which follows intuitively since the optimal cut edges E_i collectively partition the armies a_i where the minimal cuts $\delta(V_i)$ remove these armies individually. We then argue that $2w(E^*) = w(\delta(V_1)) + w(\delta(V_2)) + w(\delta(V_3))$ because each edge in E^* is incident to two of these three groups V_i and exists in both of the cuts $\delta(V_i)$. Therefore, the sum of the weights of these cuts is twice the optimal cut. Or alternatively:

$$\sum_{i=1}^{k} w(E_i) \leq \sum_{i=1}^{k} w(\delta(V_i)) = 2w(E^*)$$

(c) Propose a local search approach for the general case where Jack has m armies.

There are many ways we could perform local search for this problem. The local step can be a removal of an edge or a set of edges from the graph G followed by a test to see if the cities with armies have all become disconnected. There are a couple of different strategies on how to pick edges:

Randomly: The simplest, probably most naive approach. Randomly remove edges and then test for connectivity.
Greedily: Remove the lowest weight edges first.

Then, we must select from which edges we consider removal. There are a couple of considerations we have here as well:
All: Consider all edges for removal, using either the greedy or random approach.
Approximation: First run the 2-approximation proposed in part (b). Then, from that set of edges, remove them either randomly or greedily.

One strategy to keep in mind, especially when considering all edges for removal, is to test if a single army has become isolated. If such an army has become isolated, do not consider any edges within its component for further removal.

Problem 4 (25 points):

In order to construct this approximation, we must leverage the fact that the roads satisfy a metric function, and in particular, that they satisfy the triangular inequality, while the graph is a complete one. This is known as the Minimum Steiner Tree problem.

Let the set of cities, V be partitioned into two groups, R, the cities with armies, and S, the other cities in the network. A Minimum Steiner Tree, $T = (V, E)$ connects all the cities in R using edges which connect nodes in both R and S with minimum cost. We argue we can construct a minimum spanning tree, $T' = (R, E')$ over only the required cities, where $cost(T') \leq cost(T)$.

Consider a DFS traversal of T, that is, a sequence:

$$x_0, x_1, \ldots, x_m = x_0$$

listing the vertices of T in the order in which they are considered during a DFS, including each time we return to a vertex at the end of each recursive call. That is, a node is added to this sequence whenever it is first visited on the search down, and when the backtracking of DFS re-visits it. The sequence describes a cycle over the elements of V whose total length $\sum_{i=0}^{m} d(x_i, x_{i+1})$ is precisely $2cost(T)$, because the cycle uses each edge of the tree twice.

Let now $y_0, y_1, \ldots y_k$ be the sequence obtained from $x_0, x_1, \ldots x_m$ by removing the vertices in S and keeping only the first occurrence of each vertex in R.

Then $y_0, y_1, \ldots y_k$ is a path that includes all the vertices of R, and no other vertex, and its cost $\sum_{i=0}^{m} d(x_i, x_{i+1})$ is at most the cost of the cycle $x_0, x_1, \ldots x_m$ (here we are using the triangle inequality), and so it is at most $2cost(T)$.

But now note that $y_0, y_1, \ldots y_k$, being a path, is also a tree, and so we can take T' to be tree (R, E') where E' is the edge set $\{(y_i, y_{i+1})\}_{i=0\ldots k}$.