Problem Set 4 Solutions

This problem set is due **at 11:59pm** on **Wednesday, March 8, 2017.**

EXERCISES (NOT TO BE TURNED IN)

Minimum Spanning Tree

- Do Exercise 23.1-5 in CLRS (pg. 629)
- Do Exercise 23.2-8 in CLRS (pg. 637)

Max Flow

- Do Exercises 26.1-6 in CLRS (pg. 714)
- Do Exercises 26.2-10 in CLRS (pg. 731)

Problem 3-1. Save the Bridges! [50 points] Ben Bitdiddle is the mayor of GatesTown, a city that has been at war for a long time with DreyfoosCity. The good people of GatesTown believe that roads are immoral and only use bridges or tunnels instead. Thus, GatesTown has n houses, and no roads. Instead of roads, there are m bridges going from house to house. Bridges are made of stones and the bridge going from house u to house v has $s_{u,v}$ stones. GatesTown is a connected city: from any house, it is possible to reach any other house by following some sequence of bridges.

Ben is worried that Alyssa P. Hacker from DreyfoosCity will try to blow up the bridges in GatesTown, so he asks his engineers to make a plan to protect the bridges. The engineers come back to Ben with a set containing k different plans. Each bridge protection plan i has a "power" p_i and a cost c_i. If the i^{th} plan is enacted, all the bridges with $s_{u,v} \leq p_i$ will be protected, while the others will be destroyed in case of attack. Notice that the number of bridges that are saved does not impact the cost of the plan. If a plan with sufficient power is enacted, there is no limit to how many bridges can be saved.

(a) [20 points] Ben would like to choose a plan that keeps the entire city connected, while spending as little as possible. Develop an efficient algorithm to determine the best plan for this purpose, or to establish that no plan will keep the entire city connected.

Solution:

Pre-Solution Steps The fact that we want to keep the city connected should ring a bell - if we have a connected graph we can certainly reduce it to a tree by removing some of the edges, and the largest $s_{u,v}$ in the resulting tree will be at most as large as the largest $s_{u,v}$ in the full graph.

Intuitively, it seems like finding the MST of the city graph C would point us in the right direction. Notice, however, that the two concepts are a little different, at least on the surface: the MST is the tree with the smallest total weight, and here we are interested in finding a tree that has the smallest possible maximum weight. Are the two the same? Clearly, we can come up with a tree that has the smallest possible maximum weight but is not a MST. However, if we can prove that if we pick a MST of C then its heaviest edge is the smallest possible maximum weight in a spanning tree of C, then we are set.

This intuition turns out to be correct, as we illustrate in the solution below.

Solution The efficient algorithm works as follows: first, apply Prim's algorithm to find a MST for C. Then, iterate through the edges in the MST to find the heaviest one. Finally, iterate through the plans finding the cheapest one that has $p_i >= s_{u,v}$ for the aforementioned heaviest edge (u, v). That is the correct plan to pick. If no plans have sufficiently high power, the algorithm will establish that the city cannot be kept connected.

The proof of correctness of this algorithm depends on the result mentioned in the solution steps, which we express as a lemma.

Lemma 1: Every spanning tree of C has an edge that is at least as heavy as the heaviest edge in any MST of C.

Proof of lemma 1: Consider any MST T of C, let (u, v) be the heaviest edge in that MST, so that u and v are its adjacent vertices. Consider now any other spanning tree S of C. If (u, v) is part of S, then S has an edge that is at least as heavy as (u, v), namely (u, v) itself. If (u, v) is not in S, then consider the cut in C that is crossed by (u, v). By the cut property, (u, v) must be the smallest weight in the cut, and some edge crossing the cut must be present in S since S is a spanning tree. That edge will have weight at least as large as the weight of (u, v), thus proving our lemma.

Now that we have Lemma 1, proving our algorithm's correctness is easy. The heaviest edge in the MST of C gives us a lower bound on the required power, and also an upper bound since we know that the MST itself is good enough to keep the city connected. Out of all plans with the required power, we would like to output the cheapest one, as we in fact do in our iteration.

The running time of this algorithm is $O(m+n \log n+k)$. This comes from the running time of Prim's algorithm and from iterating through the list of plans.

(b) [30 points] It turns out that the city council finds this plan too expensive. Ben decides that the best way to convince the inhabitants is to concretely tell them what their money can achieve. He asks each household u, to tell him what maximum cost c_u^{max} they deem appropriate for the city to spend on bridge protections. He then replies back to each household with the maximum number of houses they would be connected with, if the city was constrained to their suggested budget (each household only gets a reply about their own suggested budget). Design an algorithm to efficiently compute all these numbers. By efficiently, we mean an algorithm that runs in $O(m \log m+k \log k)$ time. By using part (a) and results we have mentioned in class, it is possible to achieve an even better running time.

Solution:

Pre-Solution Steps It seems like in this part of the problem we are interested in finding something that is pretty close to our notion of connected components. Interestingly, we are interested in connected components that evolve over time. Do we know of any way to keep track of how connected components evolve over time? It seems like the Union-Find data structure is what we might need.

The way that Union-Find helps us keep track of connected components is the following: whenever an edge is seen, it makes us join two connected components. In this case, we want to see how connected components change over time as we see edges that increase the required power level for the plan.

Let us develop a naive solution sketch first, as it might be intuitive to do the following: set up a Union-Find data structure, iterate through the edges in increasing order of weight and use them to connect components in the Union-Find data structure. We augment the data structure so that each component stores the number of houses that it contains. As we do so, whenever we cross the power of the best cost in a household's

budget, we mark that household with the number of other houses in the the house's connected component.

This is correct, but it requires a couple of remarks:

- First, we need to set up a fast-access structure that maps each power level with the households to mark.

- Second, we would like our solution to not need to sort the entire set of edges. We can think about part (a) here: what if we only use edges in a MST of C? Lemma 1 from the previous part actually takes care of this detail, as the connected components that we are dealing with are themselves subgraphs of C.

Solution Our algorithm works as follows:

1. First, find a MST of C by using Prim's algorithm.
2. Secondly, sort the edges in the MST in increasing order.
3. Sort the plans in increasing power. Eliminate any plan from the list that has a cost greater or equal than another plan with greater or equal power.
4. Sort the households in order of increasing c_u^{max}.
5. Iterate through the sorted households and the plans at the same time, keeping pointers on both, and adding a pointer from each plan to any households for which that is the most powerful plan cheaper than the household's c_u^{max}.
6. Initialize a Union-Find data structure, that contains all households as single points. Augment the data structure so that it keeps track of the number of houses in each connected component.
7. Iterate through the sorted edges, and use them to connect components in the Union-Find data structure. As we connect components, we update the numbers keeping count of the number of houses in each connected component. Also, as we do so, we keep a pointer that moves on the sorted list of plans; whenever we see all the edges allowed by any plan in the list, we mark all the households that are pointed to by the plan.

For the proof of correctness, we refer to the naive solution in the solution steps first.

Union-Find correctly keeps track of the evolution of connected components over time (because of how Union-Find is defined), so it must follow that at any point during the algorithm Union-Find is tracking connected components. Because we are iterating through edges in increasing order, the connected components track over time what can be saved by better and better plans. Our pointer system allows to mark the households correctly as we cross the relevant plans in our iterations. This proves the correctness of the naive solution.

Going from the naive solution to the actual solution only requires Lemma 1, which was already proven in the previous part of the problem.

Finally, let us consider the running time of this algorithm. Since the MST has $n - 1$ edges, all the sorting operations (and Prim's algorithm with Fibonacci heaps) take

$O(m + n \log n)$ time. Then, sorting the plans takes $O(k \log k)$ time. So, the total running time of the algorithm is $O(m\alpha(m) + n \log n + k \log k)$.

Problem 3-2. Posts and Tunnels [50 points] Ben Bitdiddle's war with DreyfoosCity is lasting much longer than he had originally expected. As the stone bridges around GatesTown are getting destroyed, families are getting separated as people who leave town for a day often have no way of getting back.

As Ben is about to surrender, he discovers a map created by Queen Hopper detailing a network of ancient catacombs underneath the city, dating back to the times of the Cobol Empire. Armed with this knowledge, he cleans up the tunnels and sets up sentry posts at each intersection to help people travel from a meeting point outside of the city to the city center.

More formally, Ben sets up a system of tunnels and posts (junctions where the tunnels intersect) to help people travel into the city. The tunnels and posts can be modeled as the edges and vertices (respectively) of a graph. The goal of this system is to transport as many people as possible safely through the tunnels from the meeting point outside GatesTown to the city center.

However, each tunnel $(i, j) \in E$ which connects post i to post j (note that all tunnels are directed) has a maximum capacity - each tunnel can only hold up to $c_{i,j}$ people per hour. Moreover, each post $i \in V$, except the city center, can only handle m_i incoming people per hour. The city center, which can also be modeled as a post, can handle any rate of incoming people per hour. The meeting point outside of the city can be modeled as the source and we can assume that there is an infinite supply of people who want to go to DreyfoosCity.

(a) [15 points] Construct a flow network where calculating the maximum flow gives the maximum number of people that can get to the city center every hour.

Solution:

Pre-solution steps: We already have a maximum flow problem ready, but we have an additional constraint on it. Specifically, posts (vertices) can only have a fixed amount of incoming flow. Ideally, our answer will be a maximum flow graph like the original one (with no constraints on post capacities), on top of which we will somehow enforce post capacities. The only capacities in max flow problems are edge capacities, so we must somehow use edge capacities to enforce vertex capacities. As we outline in the solution below, we want to add an edge for each vertex, and that edge's capacity will be our constraint.

Solution: The m_i constraint is essentially introducing vertex capacities into the question. We can construct a flow network with V and E, where the outside of the city is s and the city center is t. The capacity of each edge $(i, j) \in E$ is $c_{i,j}$. To deal with the vertex capacities, we can split up each vertex v into two vertices v_1 and v_2 - intuitively, this represents entering and exiting post v - all the incoming edges into v will be connected to v_1. All the outgoing edges from v will be connected out from v_2. The capacity of the edge (v_1, v_2) will be m_v people per hour.

(b) [10 points] Ben Bitdiddle uses a maximum flow algorithm on the network constructed in part (a) and calculates the maximum number of people he can get to the city center per hour. However, some of the workers renovate one of the tunnels and increase its capacity - that is, there exists a tunnel $(i, j) \in E$ whose capacity has increased from $c_{i,j}$ to $c_{i,j} + 1$. Using the flow graph from part (a), design an efficient algorithm to update the value of the max flow correctly. Remember that the capacities represent numbers of people, so they must all be integers.

Solution:

Pre-solution steps: In order to solve this, we need to think about some of the concepts seen in class about flow graphs. Only if the increased edge is on the min-cut that corresponds to the found max-flow, then we might be able to increase the flow in the graph. In this case, adding capacity connects the source and the sink in the residual graph, meaning that there must be augmenting paths.

Solution: Increasing one edge's capacity by one can increase the value of the flow by at most 1. Thus, we can run a single round of BFS (or DFS) on the residual network to search for augmenting paths. If there is an augmenting path, we return the original flow value, incremented by 1. If there is none, then the edge whose capacity was increased was not a bottleneck edge and the value of the max flow is unaffected.

Ben soon realizes that the bottleneck in getting people into town isn't the capacity of the tunnels, but the processing required at each post. Assume that every time a traveler reaches a post, his or her ID gets a new stamp. Instead of trying to maximize the number of people per hour, Ben's new goal is to maximize the number of people who can reach the city center with at most K stamps on their ID. Note that this makes the initial $c_{i,j}$ and m_i constraints irrelevant. Unfortunately, Ben's bureaucratic team complicates matters by imposing the additional constraint that each post i can only send one person with h stamps to post j.

(c) [25 points] Construct a flow network where the max flow equals the maximum number of people who can reach the city center in at most K stamps, while satisfying the above constraints.

Hint: Consider making multiple copies of the graph G

Solution:

Pre-solution steps: We want to follow the hint, and create multiple copies of the graph. Notice that people have different possible paths depending on their stamp. So, the most meaningful way to do that will be to have one copy of the graph per number of stamps. So, we have vertices that represent posts with people going through them after a certain number of hops. This will inform the way that we want to connect vertices between the K copies of the graph.

Solution: Make K+1 copies of our original graph from part (a), one per hop count (from 0 to K). Define v^j as copy j of vertex v - the copy that corresponds to a stamp

of j. Create edges between u^j and v^{j+1} if there exists an edge (u, v) in the original graph (if there's a tunnel between posts u and v). There are now $K + 1$ copies of the source vertex - to deal with this, we create a new source with hop count 0, which is connected with infinite capacity to only the $K + 1$ copies of the original source vertices. Similarly, we create a new sink vertex that has incoming edges from all $K + 1$ copies of the original sink vertices with infinite capacity.