Advanced Algorithms
Test 2: Solutions

Problem 1. (Greedy Algorithms)
Recall that the subgraph G' of G induced by a subset V' of V is the graph $G' = (V', E')$, where E' is the subset of E containing exactly those edges with both endpoints in V'. You are given an undirected graph $G = (V, E)$ (as an array of adjacency lists) and a positive integer K and asked to find a maximum subset of vertices such that the subgraph of G induced by these vertices has degree at least K (i.e., every vertex in the induced subgraph has degree of at least K).

Consider the following greedy algorithm:

> repeatedly remove the remaining vertex of smallest (updated) degree, until all remaining vertices have degree at least K.

We use a priority queue to retrieve the remaining vertex of smallest degree; since removing a vertex decreases the degree of its neighbors, we use a Fibonacci heap to take advantage of its efficient `DecreaseKey` *operation. Thus our algorithm begins by building a Fibonacci heap containing all vertices, using the degree of each vertex as the priority key. Our main loop then simply runs a* `DeleteMin` *operation on the heap. If the heap is empty, the algorithm stops and returns the empty set; otherwise, it tests the key (degree) of the returned vertex v against K. If the degree is at least K, the algorithm stops and returns the collection of vertices still in the heap. Otherwise, the algorithm updates the degrees of the neighbors of v (listed in the adjacency list of v) using* `DecreaseKey` *and begins the next iteration.*

Prove the correctness of this algorithm and that it runs in $O(|E| + |V| \cdot \log(|V|))$ worst-case time.

Solution:
The algorithm clearly returns a subset of vertices that all have degree at least K, since degrees can decrease through the procedure, but never increase. Thus we need only show that the subset returned is a maximum subset among all those where every vertex has degree at least K. Such a proof is needed because the algorithm actually decreases vertex degrees, so that a vertex with initial degree larger than K may get eliminated. We prove it by induction on the number of vertices. Assume that the algorithm returns the correct answer for graphs of up to n vertices and consider a graph with $n+1$ vertices. Pick the vertex of least degree in the graph; if that vertex has degree less than K, then it clearly cannot be part of the solution and so can safely be removed. Removing it will decrease by 1 the degree of each of its neighbors and leave a new graph of n vertices, which obeys the inductive hypothesis, so that the set of vertices returned for it by the algorithm is optimal and thus also optimal for the graph of $n+1$ vertices. If, on the other hand, the vertex of smallest degree has degree K or larger, then every vertex has degree K or larger and so the entire graph is the solution.

As to running time, the algorithm can build the heap in linear time, then carries out both `DeleteMin` and `DecreaseKey` operations. The number of `DeleteMin` operations cannot exceed the number of vertices and thus these operations take $O(|V| \log |V|)$ worst-case time overall. (Fibonacci heaps are amortized data structures, but here we analyze them over the duration of the algorithm, with initial potential of 0 and final potential in $O(n)$, so that the sum of amortized times is in fact a worst-case time for the algorithm.) The number of `DecreaseKey` operations is bounded by the number of edges, since an edge is used at most once; since the amortized time of a `DecreaseKey` is constant and the time to list all k neighbors of a vertex (by traversing the adjacency list) is $\Theta(k)$, the total time over the algorithm is $O(|E|)$ worst-case time. Summing the two contributions gives the desired result.

Problem 2. (Minimum Spanning Trees)
Give a polynomial-time algorithm for each of the following problems—sketch a proof of their correctness and analyze their running time.

You are given an undirected graph $G = (V, E)$ with (not necessarily distinct) positive integral weights for the edges, and an edge $e_0 \in E$.

1. *Decide whether* every *minimum spanning tree of G contains* e_0.
2. *Decide whether* some *minimum spanning tree of G contains* e_0.

Solution:
For the first part, it suffices to compute the cost of the MST of the given graph, call it c_0, then to increase the cost of e_0 (by any positive amount ε, say 1) and recompute the cost of the MST, say c_1. If we have $c_1 = c_0$, then there must exist an MST that does not include e_0, since any MST that does will have an increased cost; if $c_1 = c_0 + \varepsilon$, then the increase for e_0 appears in every MST and thus e_0 appears in every MST, since it is the only edge to have an increased cost. The running time is the time to run two MST computations and so is in $O(|E| + |V| \log |V|)$.

For the second part, we take the reverse approach. We again compute the cost of the MST of the given graph, c_0, then we *decrease* the cost of e_0 by one, and recompute the cost of the MST, say c_2. If at least one of the MSTs of cost c_1 included e_0, that same tree now has cost $c_2 = c_1 - 1$; if none of the trees of cost c_1 included e_0, then we must have $c_2 = c_1$ (although we may have gained some new MSTs of cost c_1). The running time is again that of an MST algorithm.

An alternative solution for the second part, one that is considerably more complicated, but also more efficient, is as follows. Remove from G all edges of cost larger than or equal to the cost of e_0, except for e_0 itself, then use a breadth-first search from the endpoints of e_0 to test whether the resulting graph has a cycle that includes e_0. In such a cycle, e_0 has strictly larger cost than any of the other edges. Note that, in order to detect the presence of a such a cycle, we cannot just examine cycles containing e_0 in the original graph, as the number of such cycles could be exponential, hence the preprocessing step to remove edges of cost at least as large as that of e_0.

We claim that e_0 can be part of some MST if and only if no such cycle exists. Suppose such a cycle c exists; then we can reduce the cost of any spanning tree T that contains e_0 as follows. We add to T every edge of c that is not already in T, creating several cycles, but in particular creating the cycle c itself. We start by breaking cycle c by removing e_0, then, for each remaining cycle, we remove its most expensive edge. In removing the most expensive edge of a cycle we either remove an added edge (no change in cost) or an existing edge (with a possible decrease in cost); in removing e_0 to break c, we decreased the cost, since e_0 has the strictly largest cost in that cycle. Thus our procedure produces a new spanning tree that is of lower cost than T, so that T cannot have been an MST. Hence no MST can contain e_0. Conversely, suppose that e_0 belongs to some MST T^*. Then there cannot be any cycle c in G such that e_0 has the strictly largest cost of any edge in the cycle. Otherwise we could use the same procedure on T^*. This would produce a new spanning tree of lower cost than T^*, but that cannot be, since T^* is optimal. Removing all edges of cost at least as large as that of e_0 takes $O(|E|)$ time; running the breadth-first search for a cycle including e_0 takes $O(|E|)$ time as well, so the entire procedure takes linear time.

Problem 3. (Bipartite Matchings)
Give a polynomial-time algorithm for each of the following problems—sketch a proof of their correctness and analyze their running time.

You are given a bipartite graph $G = (V_1, V_2, E)$, $|V_1| = |V_2|$, *and an edge* $e_0 \in E$.

1. *Decide whether* every *perfect matching of G contains* e_0.
2. *Decide whether* some *perfect matching of G contains* e_0.

Solution:
The first part is very similar to the previous problem, but now we do not have costs—or, rather, the cost is 0/1: either we get a matched edge or we do not. Thus we first verify that G has

a perfect matching (by computing a maximum matching and verifying that every vertex is matched); if it does not, we return true. If it does, we remove e_0 from the graph (just the edge, not its two endpoints) and compute a maximum matching again; if the maximum matching is a perfect matching, then not every perfect matching requires e_0 and so we return false; otherwise, removing e_0 removed every perfect matching, so that e_0 occurs in every perfect matching and we return true.

The second part is just a generalization of the first. We again verify that G has a perfect matching; if not, we stop and return false. If it does, we remove every edge incident to one or the other endpoint of e_0—but we do not remove e_0 itself; this modification ensure that e_0 is part of any maximum matching. Now, we compute a maximum matching on the new graph. If that maximum matching is a perfect matching, then we returns true—since that matching contains e_0. Otherwise, we return false, since not maximum matching that contains e_0 can reach the size of a maximum matching.

In each case, the maximum bipartite matching time, $O(\sqrt{|V|} \cdot |E|)$, dominates.

Problem 4. (Maximum Flow)
*You are given an undirected network $N = (V,E)$ with integral edge capacities, a source, s, and a sink, t. Define N to be k-*stable *if and only if the value of the maximum s-t flow does not increase under any k-edge alteration. A k-edge alteration consists of picking k edges of the network and assigning them arbitrary new (positive integral) capacities.*

1. *Design an algorithm to test whether N is 1-stable; your algorithm should run in $O(|V|^2 \cdot |E|)$ time.*
2. *Design an algorithm to test whether N is 2-stable; your algorithm should also run in $O(|V|^2 \cdot |E|)$ time, although it is likely to involve significantly more work.*

Solution:
Both parts require that we first compute a maxflow and save the residual graph. Since we have a maxflow, there is no path in the residual graph from s to t. However, if we can make t reachable from s (creating an augmenting path) by modifying at most k edges, then the network is not k-stable. Define V_s to be the set of all vertices reachable from s in the residual graph (including s itself), and V_t to be the set of all vertices from which t can be reached in the residual graph (including t itself). By our previous remark, these two sets are disjoint. Let all other vertices form the set V_o (V_o may be empty). Observe that both $V_s, V_t \cup V_o$ and $V_s \cup V_o, V_t$ are mincuts.

Note (added after grading the tests): it is important to realize that V_s, V_t is not a cut when V_o is not empty. A cut is partition of all vertices of the network such that s lies on one side of the partition and t on the other side, and $V_s \cup V_t$ does not contain all vertices of the graph when V_o is not empty.

The solutions we give below are optimized for $k = 1$ and $k = 2$, not designed to illustrate how to develop a general solution for any k. First consider the case $k = 1$. We claim that G is *not* 1-stable if and only if there exists an edge $e = \{u,v\}$ in the network with $u \in V_s$ and $v \in V_t$. If such an edge exists, we can increase its capacity to yield an augmenting s-t path, which means G is not 1-stable. If no such edge exists, then there is no common edge to the mincuts $V_s, V_t \cup V_o$ and $V_s \cup V_o, V_t$, so that a change to a single edge can remove only one of the two mincuts and the maxflow remains unchanged; therefore, G is 1-stable.

Note (added after grading the tests): a very important part of the argument here is the use of mincuts, not specific flows. Using a specific flow would defeat the argument, because we would only be arguing about ways to find a new augmenting path with respect to this specific flow—and such an argument would not automatically carry over to all possible maxflows. Moreover, we cannot use all possible maxflows: first, we do not know how to generate them all, and second, there may be an exponential number of them. Using mincuts, however, we reason

directly about the graph structure and so are completely independent of any flow values; there can of course be an exponential number of different mincuts, but we need not generate them—we are simply reasoning based on their existence and properties. Finally, it is not possible to solve the problem by simply putting every saturated edge back in the residual graph and looking for an augmenting path with at most 1 such edge in it: the number of augmenting paths is potentially exponential and there is nothing to say that one of those we are looking for (if any exists) has to be the shortest.

Suppose that G is 1-stable. Now, in order to make an augmenting *s*-*t* path, we have to connect V_s and V_o by increasing the capacity of one edge in the cut $V_s, V_t \cup V_o$, and connect V_o and V_t by increasing the capacity of one edge in the cut $V_s \cup V_o, V_t$. We do not know how to choose these two edges, so instead we increase the capacity of *every* edge in each of the two cuts and test whether there exists an *s*-*t* path in the resulting residual graph. Such a path will use one edge from each of the two cuts and thus two edges in all, so that it suffices to increase the capacity of these two edges to augment the maxflow, verifying that the network is not 2-stable. We need to run the maxflow algorithm to get the residual graph, taking $O(|V|^2 \cdot |E|)$ (or less with a more efficient algorithm), increase the capacity of the cut edges and update the residual graph to reflect the increase, both taking $O(|E|)$ time, and finally search for an *s*-*t* path in the residual graph, again taking $O(|E|)$ time. The maxflow time dominates the procedure.