CS 577: Introduction to Algorithms Fall 2020

Homework 7 Solutions

Instructor: Dieter van Melkebeek TA: Bryan Jin

Problem 1

Suppose we are given an integer k, together with a flow network $N = (V, E, c)$ in which every edge has capacity 1. Design an algorithm to identify k edges in N such that after deleting those k edges, the maximum value of a flow in the remaining network is as small as possible. Your algorithm should run in time polynomial in n and m.

We recall strong duality: the maximum value of a flow in any network is equal to the capacity of a minimum s-t cut in the same network. Suppose we can identify k edges in N such that after deleting these k edges, the capacity of a minimum s-t cut in the remaining network is as small as possible. Strong duality tells us that these k edges would also be a satisfactory solution to the given problem. We now want an algorithm that finds k edges in N such that after removing these k edges, the capacity of a minimum s-t cut in the remaining network is as small as possible.

To develop such an algorithm, we must understand how removing edges from a network changes the capacity of an s-t cut.

Claim 1. *Consider the network N' obtained by removing from the network $N = (V, E, c)$ a subset $F \subseteq E$ of edges. Let (S, T) be any s-t-cut of N. Note that (S, T) is also an s-t-cut of N'. Let $c(S, T)$ be the capacity of (S, T) in N and let $c'(S, T)$ be the capacity of (S, T) in N'. Then we have*

$$c'(S, T) = c(S, T) - |F \cap S \times T|.$$

Less formally, suppose we remove some edges from N. The capacity of (S, T) will decrease by some nonnegative amount. This nonnegative amount is equal to the number of removed edges that crossed from S to T.

The claim follows from the definition of capacity of an s-t cut. The capacity of an s-t cut (S, T) in a network $N = (V, E, c)$ is the sum of the capacities of all edges in N that cross from S to T:

$$c(S, T) = \sum_{e \in E \cap S \times T} c(e).$$

For this problem, we know every edge has capacity 1. Therefore, the above expression simplifies to

$$c(S, T) = \sum_{e \in E \cap S \times T} 1 = |E \cap S \times T|.$$

In words, the capacity of (S, T) in N is the number of edges in N that cross from S to T. Note that since N' is also a network where each edge has capacity 1, we know the capacity of (S, T) in N' is the number of edges in N' that cross from S to T.

Therefore, the difference between $c(S, T)$ and $c'(S, T)$ is the number of removed edges that crossed from S to T.

As a result of this claim, we understand that if we remove k edges from N, the capacity of any s-t cut in N will be reduced by at most k. Specifically, the capacity of a minimum s-t cut can be reduced by at most k. We can achieve this minimum by removing k edges that each go from the source side to the sink side of a fixed minimum s-t cut.

Our algorithm is to find a minimum s-t cut (S^*, T^*) of N. We then output k edges that cross from S^* to T^*. If less than k edges cross from S^* to T^*, we output all the edges that cross from S^* to T^*, guranteeing the capacity of (S^*, T^*) becomes 0.

Our algorithm computes a minimum s-t cut in N and finds up to k edges that cross from the source side to the sink side. The runtime of our algorithm is polynomial in n and m, assuming we find the minimum s-t cut in N using an efficient algorithm, such as the $O(nm)$ network flow algorithm. Finding the k edges can be done by iterating over all the edges, which can be done in time polynomial in n and m.

Problem 2

Given a flow network $N = (V, E, c)$ with source s and sink t, we say that a node $v \in V$ is *upstream* if, for all minimum s-t cuts (S, T) of G, $v \in S$. In other words, v lies on the s-side of every minimum s-t cut. Analogously, we say that v is *downstream* if $v \in T$ for every minimum s-t cut (S, T) of G. We call v *central* if it is neither upstream nor downstream.

Design an algorithm that takes N and a flow f of maximum value in N, and classifies each of the nodes of N as being upstream, downstream, or central. Your algorithm should run in linear time.

Consider the min-cut (S^*, T^*) where S^* consists of all the vertices that are reachable from the source s in the residual network N_f where f is the given maximum flow. We claim that a node v is upstream if and only if $v \in S^*$. Clearly, if v is upstream, then it must belong to S^*; otherwise, it lies on the sink-side of the minimum cut (S^*, T^*). Conversely, suppose that $v \in S^*$ were not upstream. Then there would be a minimum cut (S, T) with $v \in T$. Now, since $v \in S^*$, there is a path in N_f from s to v. Since $v \in T$, this path must have an edge (u, w) with $u \in S$ and $w \in T$. But this is a contradiction since no edge in the residual network N_f corresponding to a max flow f can go from the source side to the sink side of any minimum cut. (For any max flow f and any min cut (S, T), f must saturate every edge from S to T while every edge from T to S must have 0 flow. This is true regardless of whether S is the set of vertices reachable from the source in N_f.)

A symmetric argument shows the following. Let (S_*, T_*) denote the cut where T_* consists of all vertices from which the sink t can be reached in N_f. Then (S_*, T_*) is a minimum cut, and a vertex w is downstream if and only if $w \in T_*$. (Formally, this statement can be obtained from the upstream one by reverting all edges and flows in N_f.)

Thus, our algorithm is to build N_f, and run a graph traversal to find the sets S^* and T_*. These are the upstream and downstream vertices, respectively; the remaining vertices are central.

The running time of our algorithm is linear as we can construct N_f out of f in linear time, and graph traversal can be done in linear time (using BFS of DFS).

Problem 3

A given network G with integer capacities may have more than one minimum s-t cut. Define the *densest* minimum s-t cut to be any minimum s-t cut (S, T) of G with the greatest number of edges crossing from S to T.

a) Suppose we have access to a black box called MaxFlow. MaxFlow takes as input a network N and outputs a flow of maximum value for N. Design an algorithm to find a densest minimum s-t cut of G using at most one call to MaxFlow. Outside of MaxFlow, your algorithm should run in linear time. You may assume that standard arithmetic operations can be done in constant time.

b) Design an algorithm to determine whether G contains a unique densest minimum s-t cut. Once again, you can make at most one call to MaxFlow. Outside of MaxFlow, your algorithm should run in linear time. You may assume that standard arithmetic operations can be done in constant time.

Part (a)

If we input G into the MaxFlow black box and get a flow of maximum value for G, we can construct the residual network corresponding to the flow and find a minimum s-t cut of G using the residual network. Unfortunately, we have no guarantee that the minimum s-t cut of G produced in this way will be a densest minimum s-t cut.

Intuitively, we have to somehow distinguish the densest minimum s-t cut from all the other minimum s-t cuts. We want to reward minimum s-t cuts that have more edges crossing from the source side to the sink side.

One way to do this is to reduce the capacity of each edge by some fixed amount. Then the densest minimum s-t cut, which has the most edges crossing from source side to sink side, will have a smaller capacity in the new network than minimum s-t cuts with fewer edges crossing from source side to sink side.

However, we cannot reduce the capacity of each edge by too much. Otherwise, a minimum s-t cut in the new network would simply be an s-t cut with the most number of edges crossing from source side to sink side.

The good news is that we can distinguish a densest minimum s-t cut from other minimum s-t cuts by reducing all the edge capacities by any fixed positive quantity ϵ, no matter how small.

How small does ϵ need to be? Here we take advantage of the fact that G has integral capacities. Non-minimum s-t cuts have capacity at least 1 greater than the capacity of a minimum s-t cut. Therefore, if $\epsilon \leq 1/(m+1)$, where m is the number of edges in G, we know that an non-minimum s-t cut will still have a greater capacity than a minimum s-t cut after each edge capacity decreases by ϵ.

Our algorithm will create a new network G' with the same vertices and edges as G, but where each edge capacity is reduced by $1/(m+1)$. By our discussion above, we know that (S, T) is a densest minimum s-t cut of G if and only if (S, T) is a minimum s-t cut of G'. We use the MaxFlow black box to find (S^*, T^*), a minimum s-t cut of G'. Then (S^*, T^*) will be a densest minimum s-t cut of G.

Outside of the black box call, our algorithm creates G', creates a residual network given a flow, and finds all the vertices reachable from the source in the residual network. Assuming standard arithmetic operations can be done in constant time, each of the above steps takes linear time. Therefore, outside of the black box MaxFlow, our algorithm runs in linear time.

Sidenote: instead of mapping an edge capacity c in G to $c-1/(m+1)$ in G', we can equivalently map c to $c \cdot (m+1) - 1$. This ensures G' has integral capacities.

Part (b)

Using the correspondence between densest minimum s-t cuts of G and minimum s-t cuts of G', it suffices to determine whether G' has a unique minimum s-t cut.

Recall the construction of a min cut (S^*, T^*) out of a max flow from class: If f is a max flow in G', then we set S^* to be all vertices that are reachable from s in the residual network G'_f. In fact, this set S^* is a subset of the source side S of *every* min cut (S,T) of G'. This is because if $uinS$ and $e \doteq (u,v)$ is an edge in G'_f, then $v \in S$:

- If e is an edge in G'_f because e is an edge in G' and $f(e) < c'(e)$, then e cannot go from S to T as all edges of G' that cross a min cut from the source side S to the sink side T need to be used at full capacity.

- If e is an edge in G'_f because $e' \doteq (v,u)$ is an edge in G' and $f(e') > 0$, then e' cannot go from T to S as all edges of G' that cross a min cut from the sink side T to the source side S cannot be used at all.

Similarly, if we let T_* denote all the vertices from which t can be reached in G'_f for a max flow f, then (S_*, T_*) is a min cut in G' and T_* is a subset of the sink side T of *every* min cut (S,T) of G'. This can be argued in the same way by considering the network G' with all edges reversed.

If follows that G' has a unique min cut iff $S^* = S_*$, or equivalently, if $S^* \cup T_*$ contains all the vertices of G'. This leads to the following algorithm:

1. We call MaxFlow on G' to find f, a flow of maximum value in G'.

2. We construct the residual network G'_f.

3. We construct the set S^* of vertices reachable from s in G'_f (using BFS or DFS).

4. We construct the set T_* of vertices that are reachable from t in G'_f with all edges reversed (using BFS or DFS).

5. We output "Yes" iff $S^* \cup T_*$ contains all vertices.

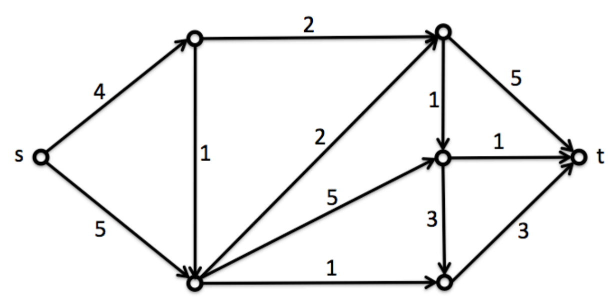
Outside of the black box MaxFlow, every step runs in linear time.

Alternate view The same solution can be obtained using the terminology of problem 2 - we output "yes" iff every vertex is either upstream (in S^*) or downstream (in T_*), or equivalently, there are no central vertices.

Problem 4

Consider a network with integer capacities. An edge is called *upper-binding* if increasing its capacity by one unit increases the maximum flow value in the network. An edge is called *lower-binding* if reducing its capacity by one unit decreases the maximum flow value in the network.

a) For the network G below determine a maximum flow f^*, the residual network G_{f^*}, and a minimum cut. Also identify all of the upper-binding edges and all of the lower-binding edges.

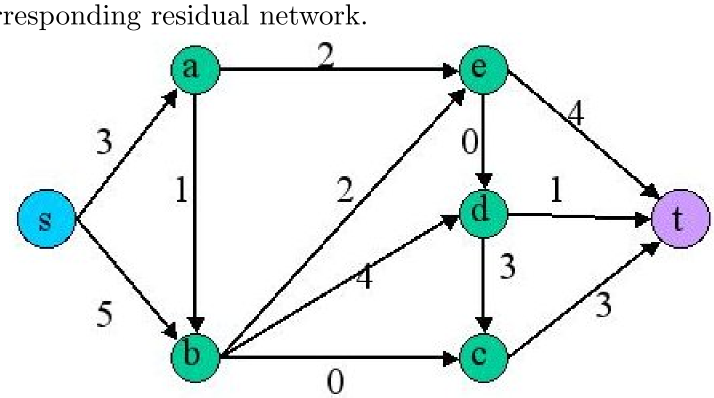


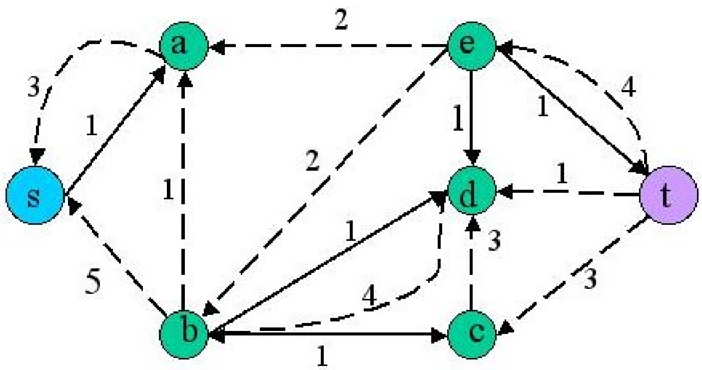
b) Develop an algorithm for finding all the upper-binding edges in a network G when given G and a maximum flow f^* in G. Your algorithm should run in linear time.

c) Develop an algorithm for finding all the lower-binding edges in a network G when given G and an integer maximum flow f^* in G. Your algorithm should run in time polynomial in n and m. Can you make it run in linear time?

Part (a)

The maximum value of a flow in the network is 8 units. The next figure shows such flow and the corresponding residual network.





Note that backward edges in the residual graph have been shown as broken. One s-t min-cut is $S = \{s, a\}$ and $T = \{b, c, d, e, t\}$. Another is $S = \{s, a, b, c, d\}$ and $T = \{e, t\}$. Only the edge (a, e) is upper-binding. The lower-binding edges are $(s, b), (a, b), (a, e), (b, e), (c, t)$, and (d, t).

Part (b)

We can test whether a given edge $e = (u, v)$ in G is upper-binding as follows. Let $G^{(e)}$ denote the same network as G but with the capacity of edge e increased by one unit. Note that f^* is a valid flow in $G^{(e)}$. The edge e is upper-binding iff the flow f^* in $G^{(e)}$ can be improved, which is the case iff there is an s-t path in the residual network $G^{(e)}_{f^*}$. Since we can construct the residual network from f^* in linear time, this gives us a linear-time procedure to check whether a given edge e is upper-binding. Doing this for all edges e yields a quadratic algorithm.

We can do better by exploiting the fact that $G^{(e)}_{f^*}$ and G_{f^*} only differ in the edge e (which is always present in $G^{(e)}_{f^*}$ but not necessarily in G_{f^*}) and that there is no s-t path in G_{f^*} (since the flow f^* has maximum value in G). Thus, there exists an s-t path in $G^{(e)}_{f^*}$ iff there exists an s-u path in G_{f^*} and a v-t path in G_{f^*}.

This observation leads to the following linear-time algorithm to determine all upper-binding edges in G. First compute the residual network G_{f^*} from the given max flow f^*. Then run DFS or BFS from s in G_{f^*} to determine the set U of all vertices that are reachable from s. Next run DFS or BFS from t on G_{f^*} with all edges reversed to determine the set V of all vertices from which t is reachable in G_{f^*}. Finally, cycle over all edges $e = (u, v)$ in G and output e iff $u \in U$ and $v \in V$.

This algorithm spends linear time in constructing the residual network, linear time in running DFS or BFS twice, and then linear time in iterating over all of the edges in G. Therefore its total running time is also linear.

Part (c)

We can test whether a given edge $e = (u, v)$ in G is lower-binding as follows. First, if e has residual capacity in G_{f^*} then e is not lower-binding. This is because f^* remains a valid flow after we reduce the capacity of e by one unit. If e has no residual but there is a u-v path in G_{f^*} then we can reduce the flow through e by one unit by rerouting that unit along a u-v path in G_{f^*}. The modified flow has the same value and remains valid after reducing the capacity of e by one unit.

Conversely, suppose that there is no u-v path in G_{f^*}. We claim that e then belongs to a minimum cut in G, which implies that reducing the capacity of e reduces the minimum cut value and thus the maximum flow value, so e is lower-binding. To argue the claim, note that the hypothesis implies that the edge e does not appear in G_{f^*} and that there is a path in G_{f^*} from t over (v, u) to s. The latter follows because there is a positive amount of flow going through e, which implies that the flow f^* contains a positive amount of flow along a path from s over e to t, and thus G_{f^*} contains the reverse of that path. Let S denote the set of vertices reachable from u in G_{f^*}, and let T denote its complement. Then $s \in S$ (because of the u-s path guaranteed above), $v \in T$ (by our assumption that there is no u-v path), and $t \in T$ (otherwise, the concatenation of the u-t path with the t-v path guaranteed above yields a u-v path). Thus, (S, T) is an s-t cut in G and e belongs to the cut. Moreover, by the proof of the max-flow min-cut theorem from class, the capacity of (S, T) equals the value of the flow f^*, and therefore is a minimum cut.

The above test can be summarized as follows: An edge $e = (u, v)$ is lower-binding iff there is no u-v path in G_{f^*}. Our algorithm to compute all lower-binding edges works as follows. It first constructs G_{f^*} from f^*. It then determines for every vertex u which vertices v are reachable from u in G_{f^*} by running DFS or BFS from u, and stores these results in a table. Finally, it cycles over all edges $e = (u, v)$ in G and outputs e iff the table indicates that v is not reachable from u in G_{f^*}.

The n runs of DFS or BFS take $O(n(m+n))$ time. Moreover, in time $O(n+m)$ we can eliminate all the vertices that are not involved in any edge. After that operation, the number of vertices is at most $2m$. Thus, the overall running time is $O(n + m + nm) = O(nm)$.

In fact, it is possible to solve this problem in linear time by making using of the fact that the strongly connected components of a digraph can be found in linear time. Note that if an edge $e = (u, v)$ is used at full capacity under f^* (a necessary condition for e being lower-binding), G_{f^*} contains the reverse edge (v, u), and therefore there exists a path from u to v in G_{f^*} iff u and v belong to the same strongly connected component of G_{f^*}. Based on that, we can find all lower-binding edges by cycling over all edges $e \in E$, and outputting e iff $f^*(e) = c(e)$ and the end points of e belong to the same strongly connected component of G_{f^*}. This procedure can be implemented to run in time $O(n+m)$ by first constructing G_{f^*} out of f^* and determining the strongly connected components of G_{f^*} in linear time.

Side note: Lower-binding edges are exactly the edges that belong to some minimum $s - t$ cut, and upper-binding edges are exactly the edges that belong to *all* minimum $s - t$ cuts. Think about why that is the case.

Problem 5

A given network can have many minimum st-cuts.

a) Determine precisely how large the number of minimum st-cuts in a graph can be as a function of n.

b) Show that if (S_1, T_1) and (S_2, T_2) are both minimum st-cuts in a given network, then so is $(S_1 \cup S_2, T_1 \cap T_2)$. How does this generalize to more than 2 st-cuts?

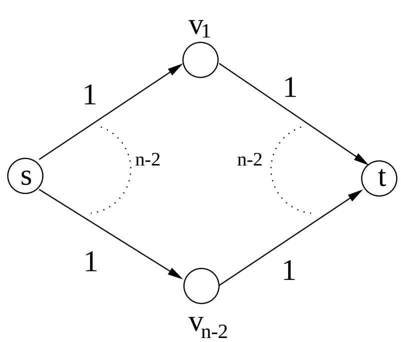
c) Design an algorithm that, given a network, generates a collection of minimum st-cuts $(S_1, T_1), (S_2, T_2), \ldots$ such that every minimum cut of the network can be written as

$$(\cup_{i \in I} S_i, \cap_{i \in I} T_i)$$

for some subset I of indices. Your algorithm should run in time polynomial in n and m.

Part (a)

First consider how many potential st-cuts there are, total. Every vertex, excepting s and t, can be in either of 2 sets: S or T. So, we can view a cut as a binary decision made on each of $n-2$ elements. The total number of st-cuts possible, then, is 2^{n-2}. Is there a scenario where all of these are *minimum* st-cuts? Consider the case in the next figure.



Whether we put some vertex v_i into S or T amounts to either placing our cut through (v_i, t) or (s, v_i). In either case, the edge we cut contributes exactly 1 to the cost of the total cut. So, all 2^{n-2} st-cuts have the minimum weight of $n-2$. Therefore, a graph can have as many as 2^{n-2} minimum weight st-cuts.

Part (b)

By the max-flow-min-cut theorem, given any maximum flow f, an st- cut (S, T) in the network G is minimum iff every edge from S to T is used at full capacity, and no edge from T to S is used at all. Equivalently, in terms of the residual network G_f, the st-cut (S, T) is minimum iff there is no edge in G_f that goes from S to T.

Let (S_1, T_1) and (S_2, T_2) be two minimum st-cuts. We need to argue that $(S_1 \cup S_2, T_1 \cap T_2)$ is a minimum st-cut. First, note that $(S_1 \cup S_2, T_1 \cap T_2)$ is a valid st-cut:

- $S_1 \cup S_2$ contains the source s,
- $T_1 \cap T_2$ contains the sink t,
- $S_1 \cup S_2$ and $T_1 \cap T_2$ do not intersect (otherwise at least one of S_1 and T_1 or S_2 and T_2 would intersect), and
- $S_1 \cup S_2$ and $T_1 \cap T_2$ together contain all vertices of G (otherwise at least one of S_1 and T_1 or S_2 and T_2 would not cover all vertices).

We next argue that the capacity of $(S_1 \cup S_2, T_1 \cap T_2)$ is minimum. Fix a maximum flow f in G. Suppose G_f would contain an edge that goes from $S_1 \cup S_2$ to $T_1 \cap T_2$. Then that same edge would go from S_1 to T_1 or from S_2 to T_2. This contradicts the minimality of (S_1, T_1) or (S_2, T_2), respectively.

We can use induction to generalize the result to more than 2 st-cuts as follows: Let (S_i, T_i), $1 \leq i \leq k$, be minimum st-cuts, then

$$(\cup_{i=1}^{k} S_i, \cap_{i=1}^{k} T_i)$$

is also a minimum st-cut. We've proven the base case above ($k = 2$). Next, we assume it holds for k cuts and show it must hold for $k+1$ cuts. We can choose any two st-cuts, coalesce them into one minimum cut by unioning their S-vertices and intersecting their T-vertices. Then, we can apply our inductive hypothesis to conclude the general case.

Part (c)

We first construct a maximum flow f in the network G. Next, we examine the residual network G_f. As we argued under (b), an st-cut (S, T) is minimum iff there is no edge in G_f that goes from S to T. Now, consider an arbitrary vertex u. The minimality criterion implies that any minimum cut (S, T) such that $u \in S$ has to contain all vertices S_u that are reachable from s or u in G_f. Let $T_u = V \setminus S_u$. By the above, we know that $S = \cup_{u \in S} S_u$. Consequently, $T = V \setminus S = \cap_{u \in S} T_u$. That is, we can write an arbitrary minimum st-cut (S, T) as

$$(S, T) = (\cup_{u \in S} S_u, \cap_{u \in S} T_u).$$

Each of the (S_u, T_u) defines a minimum st-cut unless $t \in S_u$. Since we can construct each of the sets S_u by running DFS on G_f from s and u, test whether $t \in S_u$, and construct T_u as $V \setminus S_u$ in polynomial time, we are done.