CS 577: Introduction to Algorithms Fall 2022

Homework 7

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

This homework covers network flow. **Problem 3 must be submitted for grading by 2:29pm on 11/8.** Please refer to the homework guidelines on Canvas for detailed instructions.

Warm-up problems

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.

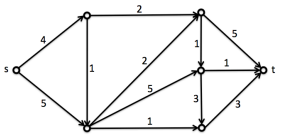
2. Given a flow network $N = (V, E, c)$ with source s and sink t, we say that a node $v \in V$ is *upstream* if, for all minimum s-t cuts (S, T) of G, $v \in S$. In other words, v lies on the s-side of every minimum s-t cut. Analogously, we say that v is *downstream* if $v \in T$ for every minimum s-t 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.

Regular problems

3. [Graded] 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 $O(m(n+m))$, where n denotes the number of vertices and m the number of edges. Can you make the running time linear?

4. A given network N 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 N with the greatest number of edges crossing from S to T.

   Suppose we have access to a black box called IntegralMaxFlow. IntegralMaxFlow takes as input a network N' with integer capacities and outputs an integral flow of maximum value for N'. Design algorithms for each of the following tasks. Each algorithm can most at most one call of to IntegralMaxFlow. Outside of IntegralMaxFlow, the algorithms should run in linear time assuming that standard arithmetic operations can be done in constant time.

   (a) Finding a densest minimum s-t cut in N.

   (b) Determining whether N has a unique densest minimum s-t cut.

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.

Challenge problem

6. Problem J from the 2007 ACM-ICPC World Finals (see next page). Your algorithm should run in time polynomial in $n \doteq R + T$.

Programming problem

7. SPOJ problem Potholers (problem code POTHOLE).

東京 2007 World Finals
acm International Collegiate Programming Contest
IBM event sponsor
HOSTS ACM Japan Chapter IBM Tokyo Research Lab

Problem J

Tunnels

Input File: tunnels.in

Curses! A handsome spy has somehow escaped from your elaborate deathtrap, overpowered your guards, and stolen your secret world domination plans. Now he is running loose in your volcano base, risking your entire evil operation. He must be stopped before he escapes!

Fortunately, you are watching the spy's progress from your secret control room, and you have planned for just such an eventuality. Your base consists of a complicated network of rooms connected by non-intersecting tunnels. Every room has a closed-circuit camera in it (allowing you to track the spy wherever he goes), and every tunnel has a small explosive charge in it, powerful enough to permanently collapse it. The spy is too quick to be caught in a collapse, so you'll have to strategically collapse tunnels to prevent him from traveling from his initial room to the outside of your base.

Damage to your base will be expensive to repair, so you'd like to ruin as few tunnels as possible. Find a strategy that minimizes the number of tunnels you'll need to collapse, no matter how clever the spy is. To be safe, you'll have to assume that the spy knows all about your tunnel system. Your main advantage is the fact that you can collapse tunnels whenever you like, based on your observations as the spy moves through the tunnels.

Input

The input consists of several test cases. Each test case begins with a line containing integers R ($1 \le R \le 50$) and T ($1 \le T \le 1000$), which are the number of rooms and tunnels in your base respectively. Rooms are numbered from 1 to R. T lines follow, each with two integers x, y ($0 \le x, y \le R$), which are the room numbers on either end of a tunnel; a 0 indicates that the tunnel connects to the outside. More than one tunnel may connect a pair of rooms.

The spy always starts out in room 1. Input is terminated by a line containing two zeros.

Output

For each test case, print a line containing the test case number (beginning with 1) followed by the minimum number of tunnels that must be collapsed, in the worst case. Use the sample output format and print a blank line after each test case.

Sample Input	Output for the Sample Input
4 6 1 2 1 3 2 4 3 4 4 0 4 0 4 6 1 2 1 3 1 4 2 0 3 0 4 0 0 0	Case 1: 2 Case 2: 2

CS 577: Introduction to Algorithms Fall 2022

Homework 7 Solutions to Warm-up Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

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).

CS 577: Introduction to Algorithms Fall 2022

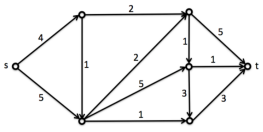
Homework 7 Solutions to Regular Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Problem 3

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.

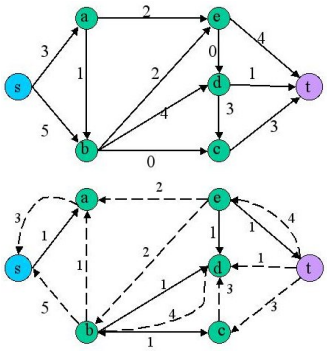


(b) Design an algorithm for finding all the upper-binding edges in a network G when given G and a maximum flow f^* in G. Your algorithm should run in linear time.

(c) Design 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 use 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 different 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 4

A given network N 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 N with the greatest number of edges crossing from S to T.

Suppose we have access to a black box called IntegralMaxFlow. IntegralMaxFlow takes as input a network N' with integer capacities and outputs an integral flow of maximum value for N'. Design algorithms for each of the following tasks. Each algorithm can most at most one call of to IntegralMaxFlow. Outside of IntegralMaxFlow, the algorithms should run in linear time assuming that standard arithmetic operations can be done in constant time.

(a) Finding a densest minimum s-t cut in N.

(b) Determining whether N has a unique densest minimum s-t cut.

Part (a)

If we input G into the IntegralMaxFlow black box and get an integral 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. One such way we could try to do this is by modifying the network in a way that a minimum cut in the new network corresponds to a densest minimum cut in the original network. This can be accomplished by reducing 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. There are, however, two problems with this idea. The first one is that we should not reduce the capacities by so much that non minimum cuts become minimum and the second one is that we need to make sure the capacities are integral.

To solve the first problem, we decrease edge capacities by a really small $\epsilon > 0$. 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 $\epsilon = 1/(m + 1)$, 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. As $\epsilon < 1$, this leads to a network with non integral capacities, but we can solve this by multiplying the new capacities by $(m + 1)$. While this changes the capacity of a minimum cut, it does not change the minimum cuts themselves.

With this discussion in mind, the algorithm creates a new network G' with the same vertices and edges as G, but where each edge capacity c is mapped to $c \cdot (m + 1) - 1 = (c - \epsilon) \cdot (m + 1)$. 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'. It then uses the IntegralMaxFlow black box to find (S^*, T^*), a minimum s-t cut of G' and returns (S^*, T^*).

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 IntegralMaxFlow, our algorithm runs in linear time.

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 $u \in S$ 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. Call IntegralMaxFlow on G' to find f, an integral flow of maximum value in G'.
2. Construct the residual network G'_f.
3. Construct the set S^* of vertices reachable from s in G'_f (using BFS or DFS).
4. Construct the set T_* of vertices that are reachable from t in G'_f with all edges reversed (using BFS or DFS).
5. Output "Yes" iff $S^* \cup T_*$ contains all vertices.

Outside of the black box IntegralMaxFlow, 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 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?

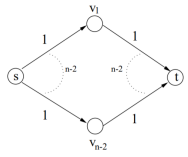
c) Design an algorithm that, given a network, generates a collection of minimum st-cuts $(S_1, T_1), (S_2, T_2), \ldots$ such that every minimum cut of the network can be written as

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

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), $1 \le i \le k$, 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.

COMP SCI 577 Homework 07 Problem 3

Network Flow

Ruixuan Tu

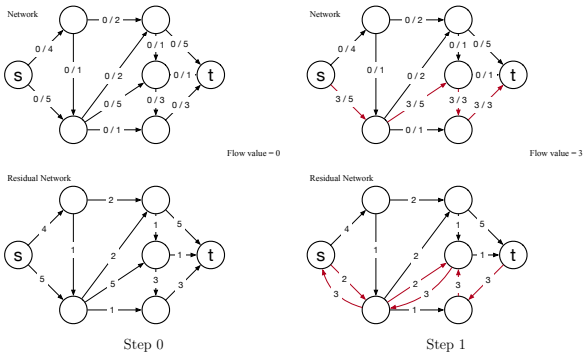
rtu7@wisc.edu

University of Wisconsin-Madison

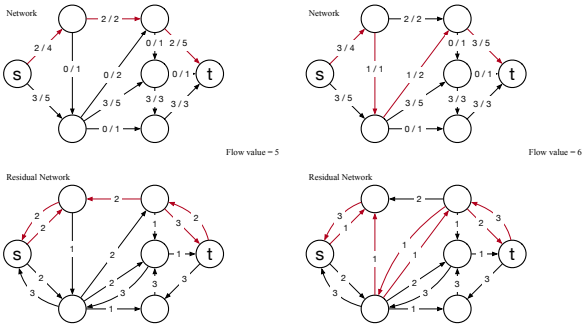
8 November 2022

Question (a)

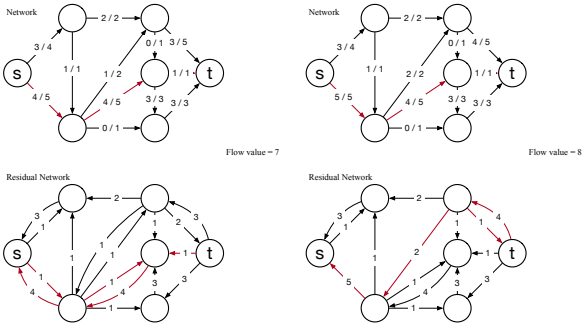
By augmentation along the path of maximum residual capacity,



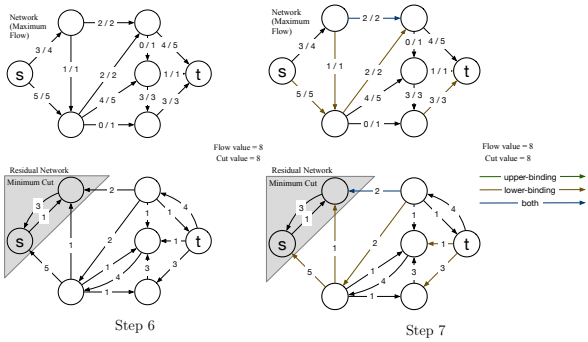
Step 0 Step 1



Step 2 Step 3



Step 4 Step 5



Step 6 Step 7

Question (b)

Algorithm

The subroutine `GetResidualNetwork` is a summary of page 2 and page 5 of the handout on 3 November 2022, so we do not need to explain or prove it.

Function `GetResidualNetwork`(G, f)**:**
  Input: network $G = (V, E, c, s, t)$; flow $f : E \to [0, \infty)$ satisfying capacity constraint and conservation constraint
  Output: residual network of G as $G_f = (V, E_f, c_f, s, t)$
  foreach $e = (u, v) \in E$ **do**
    $c_f(e) \leftarrow c(e) - f(e)$;
    if $f(e) > 0$ **then**
      $e' \in E_f \leftarrow (v, u)$ with $c_f(e') \coloneqq f(e)$;
  $G_f \leftarrow (V, E_f, c_f, s, t)$;
  return G_f;

For the main routine `GetUpperBindingEdges`, we first define S and T to be upstream and downstream vertices defined in Problem 2 of this problem set, which are identical to that S be

all vertices reachable from s, and T be all vertices reachable to t. Then we iterate every edge e from S to T and add the edge into E_u which is the collection of upper-binding edges.

```
Function GetUpperBindingEdges(G, f*):
    Input: network G = (V,E,c,s,t); maximum flow f* ∈ G
    Output: upper-binding edges E_u ∈ G
    S ← {v ∈ V : ∃ path L ⊂ G_f* from s to v} ;        // calculated by DFS or BFS
    T ← {v ∈ V : ∃ path L ⊂ G_f* from v to t} ;        // calculated by DFS or BFS
    foreach e = (u,v) ∈ E ∩ S×T & f*(e) = c(e) do
        E_u ← E_u ∪ {e};
    return E_u;
```

Proof

There is no need to argue termination, as we have referenced searching and loops which should always end.

Complexity

Denote n be the number of vertices, and m be the number of edges.

`GetResidualNetwork` costs $O(n+m)$ time, which is linear, as written on page 12 of the handout on 3 November 2022. For `GetUpperBindingEdges`, the check for all edges costs $O(n+m)$ time with accessing 2 vertices within one loop. Therefore, this paradigm costs $O(n+m)$ time overall, which is linear.

Correctness

Claim: If $e \in E$ is lower-binding or upper-binding, then $f^*(e) = c(e)$.

Proof: Let us take the contrapositive of this argument: if $f^*(e) \neq c(e)$, e is neither lower-binding nor upper-binding. Suppose $f^*(e) < c(e)$, then the flow passes through e does not occupy the whole capacity $c(e)$, so e must not be upper-binding as increasing its value will not consume more flow from s; on the other hand, e must not be lower-binding as decreasing its value might cause the flow to occupy all capacity, which is not the case which causes the flow to decrease. There is no case that $f^*(e) > c(e)$ from the capacity constraint in the definition of the flow, so we are done. ■

Claim: If $e \in E$ is upper-binding, there is no central vertex on e.

Proof: Let us take the contrapositive of this argument: If there is a central vertex on e, e is not upper-binding. As there is a central vertex on e, there are more than one distinct cuts

on e, then increasing the capacity of any edge in one minimum cut could not increase the maximum flow, as we have proved in the strong duality that $\max_f \nu(f) = \min_{\text{st-cut}(S,T)} c(S,T)$ that the two cuts both fulfill this requirement, so a flow attempting to increase at the edge must be blocked by another minimum cut, and there is no upper-binding edge in this case. Thus, if e is upper-binding, there is no central vertex on e. ■

Claim: $\forall e \in E$, e is upper-binding, iff $e \in E \cap S \times T$, i.e., $e \in S \times T$.

One side: If $e \in E$ is upper-binding, $e \in S \times T$. **Proof**: Let us take the contrapositive of this argument: $\forall e \in E$, if $e \notin E \cap S \times T$, then e is not upper-binding. If $\forall e \notin S \times T$, then $e \in S \times S$, $e \in T \times T$, or $e \in S \times C \cap C \times T$ with C be central vertices. For the first two cases, there is no minimum cut available in these subsets, so that even if we increase its capacity by 1, the maximum flow could not increase, as blocked by the minimum cut $(S,\bar{S})$ or $(T,\bar{T})$ with similar reasoning above to prove the previous lemma. For the last case, if we increase the capacity by 1 on some edge, the flow will be blocked by both minimum cuts $(S,\bar{S})$ and $(T,\bar{T})$. □

Another side: If $e \in E \cap S \times T$, e is upper-binding. **Proof**: $E \cap S \times T$ contains and only contains all edges which have no central vertex. So if we increase the capacity of such an e on one minimum cut, there must be no more edges that occupy their whole capacities (in other minimum cut) for the maximum flow to block the value of maximum flow to increase. Therefore, e is upper-binding. ■

Question (c)

Algorithm

For the main routine `GetLowerBindingEdges`, we check the only condition for every edge e: if there is a path in the residual network, then put it into E_l which is the collection of lower-binding edges, which is proven in the correctness part.

```
Function GetLowerBindingEdges(G, f*):
    Input: network G = (V,E,c,s,t); maximum flow f* ∈ G
    Output: lower-binding edges E_l ∈ G
    foreach e = (u,v) ∈ G & f*(e) = c(e) do
        if ∄ path from u to v ⊂ G_f* then          // calculated by DFS or BFS
            E_l ← E_l ∪ {e};
    return E_l;
```


Proof

There is no need to argue termination, as we have referenced searching and loops which should always end.

Complexity

For one iteration of e, we have searching that costs $O(n+m)$ time. There are m iterations for all e's, so the total cost of this paradigm is $O(m \cdot (n+m))$.

Correctness

Claim: $e = (u,v) \in E$ is lower-binding iff there is no path from u to v in the residual network G_{f^*}.

One side: If $e = (u,v) \in E$ is lower-binding, then there is no path from u to v in the residual network G_{f^*}. **Proof**: As $f^*(e) = c(e)$, for the chosen $e = (u,v) \in G$, there must be an inversed edge $e' = (v,u) \in G_{f^*}$ with weight of $f^*(e)$, while there is no $e'' = (u,v) \in G_{f^*}$. As e is lower-binding, we must be able to reduce 1 unit of the capacity of e' and reduces the maximum flow. With the 1 unit decrease in capacity, if we have a route from u to v in the residual network, then the maximum flow does not change, as the flow will continue to pass along the new path instead of the deducted edge, which is a contradiction. So we do not have a route from u to v in the residual network. □

Another side: If there is no path from u to v in the residual network G_{f^*}, then $e = (u,v) \in E$ is lower-binding. **Proof**: Suppose $e = (u,v) \in E$ is not lower-binding. As there is no path from u to v in the residual network, the deducted capacity could not be routed to the original vertex, so there must be some other vertices for the flow to be redirected to, but it must be blocked by the capacity of some other edge, as the residual network is not connected for some other vertices on minimum cut. All lower-binding edges are on some minimum cut, according to the strong duality. Therefore, as we cannot find a route to redirect the flow, the flow should be decreased by 1, which results in a contradiction. ■