CS 577: Introduction to Algorithms Fall 2019

Homework 8 Solutions

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Problem 1

There are n different atmospheric conditions that need to be measured and m different balloons that could measure those. Each balloon can make at most two measurements, and for each balloon $i = 1, 2, \ldots, m$ there is a set S_i of conditions that balloon i can measure. Each condition has to measured by at least k different balloons, but a single balloon is not allowed to measure the same condition twice.

a) Design a polynomial-time algorithm that takes the input to an instance of this problem (the n conditions, the sets S_i for each of the m balloons, and the parameter k), and decides whether there is a way to measure each condition by k different balloons, while each balloon measures at most two conditions.

b) A new requirement is that there can be no condition for which all k measurements come from balloons produced by a single subcontractor, given that each balloon is produced by one of three different subcontractors. Explain how to modify your polynomial-time algorithm from part (a) into a new algorithm that decides whether there exists a solution satisfying all the conditions from (a), plus the new requirements about subcontractors.

Part (a)

We want to decide whether we can measure our n conditions with our m balloons (of which balloon i can measure only the set of conditions S_i), measuring each condition at least k times, and with no balloon measuring more than two conditions. We can do this by looking at a maximum flow on a particular network. Note that if there is a solution that measures each condition at least k times, then there must be a solution that measures each condition exactly k times; we restrict ourselves both here and in part (b) to solutions where each condition is measured exactly k times, since it facilitates the construction of the flow network we use to solve these problems.

We construct our network G as a bipartite graph. On the left side of the graph, there is a vertex for each of the n conditions that need to be checked, while on the right, there is a vertex for each balloon. We call the left (condition) vertices $c_1 \cdots c_n$, and the right (balloon) vertices $b_1 \cdots b_m$. We include an edge of capacity 1 from a vertex c_i on the left to a vertex b_j on the right if condition i can be checked by balloon j. Then we add a source and a sink to G; the source s has an edge of capacity k to every vertex on the left (i.e., every condition vertex). The sink t has an edge of capacity 2 from every vertex on the right (i.e., every balloon vertex).

We claim that the value of a maximum flow in G is kn if and only if there is a way to measure each of the n conditions k times, with each balloon measuring at most 2 conditions.

We can see that no flow with value more than kn can exist, since s has n edges leaving it, and each has capacity k. Suppose there exists a working assignment of balloons to conditions, i.e., an assignment that is both satisfactory, in that it measures enough, and valid, in that no balloon measures more than twice. Then we can construct a flow of value kn in G as follows: Push k

units of flow from s to each condition vertex, then one unit from each condition vertex to each of the k balloons that are measuring that condition (there must be exactly k such balloons for each condition, since this is a satisfactory assignment); for each balloon, we then push a number of units equal to the number of measurements that balloon is making (which must be 0, 1, or 2, since this is a valid assignment) to the sink.

To see the other direction of the claim, suppose there is a flow in G with value kn. Then there is an integer-valued flow with value kn because all capacities are integer-valued. In this integer-valued flow, we must have each of the n condition vertices pushing flow into exactly k of the edges that connect it to the balloons. These are the balloons we use to check that condition. Every condition is checked by precisely k different balloons, as required. Moreover, since each balloon vertex on the right can push at most 2 units of flow to t, no balloon handles more than 2 conditions. Thus, the assignment works.

Part (b)

We can handle the extra wrinkle – that each balloon is made by one of three subcontractors, and no condition can be measured only by balloons from a single subcontractor – by augmenting the graph we constructed in part (a) as in Figure 1.

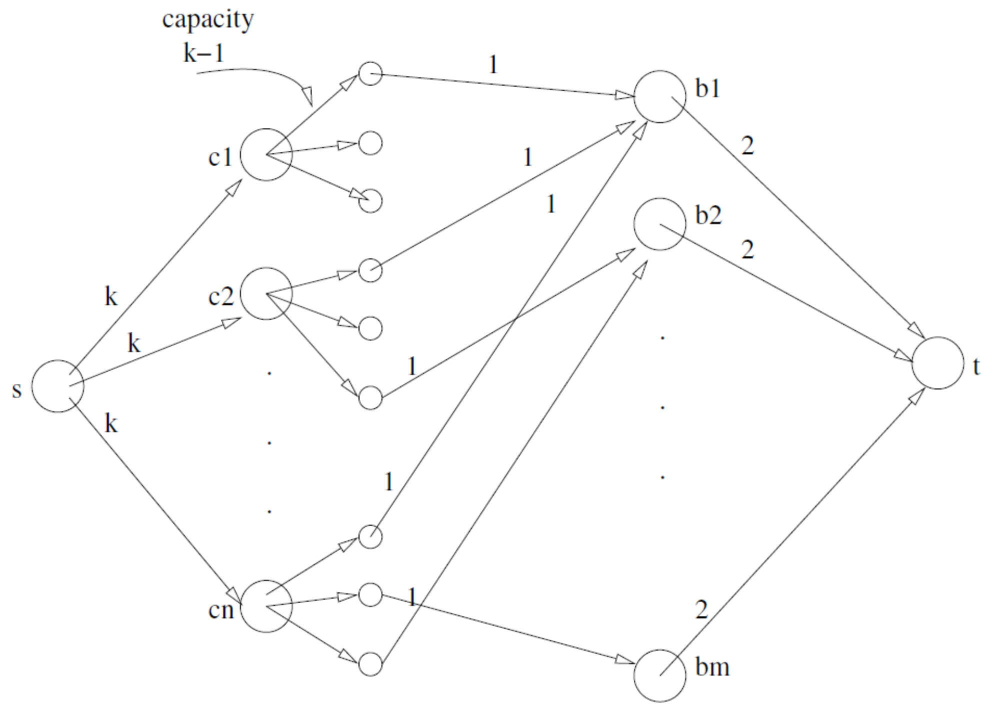


Figure 1: In this illustration, the first subcontractor manufactures balloon b_1, which can check c_1 and c_2. The second subcontractor manufactures balloon b_m, and the third subcontractor manufactures b_2

To model this additional constraint, we add three more vertices for each condition (a total of $3n$ additional vertices). These vertices are interposed between the condition vertices and the balloon vertices (i.e., between the left and right sides of G). Each condition vertex has an edge of capacity $k-1$ to each of these three subcontractor vertices, and each subcontractor vertex has an edge of capacity 1 to every balloon it makes.

Thus, a condition vertex cannot send all k units of flow that it receives through the same subcontractor vertex. This prevents any condition from having all k of its measurements done by balloons from the same subcontractor. Once again, there is a way of measuring the conditions subject to all of the constraints if and only if a maximum flow in this network has value kn.

Analysis. In both parts (a) and (b), the networks involved can be constructed in linear time, when we're given the information about which conditions can be measured by which balloons (and, in (b), which balloons are made by which subcontractors). We can then apply any of the polynomial-time algorithms we know for computing the value of a maximum flow and check whether it equals kn.

Problem 2

The Packers are playing the Bears tonight, and you'd like to invite some of your friends to watch the game at your place. All of your friends love football, and are either Packers or Bears fans, but some of them are known not to get along very well. In order to avoid possible trouble, you do not want to invite two people who are on bad terms with each other *and* root for a different team. (Having people who are on bad terms but root for the same team is OK, as is any of the other two combinations.) Also, although you like all of your friends, you like some better than others, and you have assigned a positive value to each of your friends.

Design a polynomial-time algorithm to figure out which friends to invite so as to maximize their total value under the above constraints.

Intuition This problem can be solved by reduction to minimum cut. A natural approach starts by creating a vertex for each friend, as well as separate vertices s and t. An st-cut (S, T) corresponds to a partition of the friends into two sets. Initially, we might try regarding such a partition as separating the friends into those that get invited and those that do not. This turns out not to work, but let us proceed with this idea for now, as to motivate the idea for fixing it. Let's say we try to make S to be the friends that get invited, and T to be those that are not.

What remains is to encode the friend values and incompatibility constraints as edges in the network. While our original problem was a maximization problem, it is equivalent to minimizing the total value of all friends who were *not* invited. So to encode friend values, the idea is to add edges so that they are cut precisely when a friend is not invited. Based on our rule of inviting friends iff their vertex is in S, it works to use an edge from s to that friend's vertex. This edge has capacity equal to the value of that friend.

Given that, we just need to handle the incompatibility constraints. We could try to do this using infinite capacity edges. However, there is no good place for these edges! Placing them between the incompatible friends is no good, because we only pay the infinite cost when one friend is invited while the other is not. Nothing else seems to work either.

How to fix this? The idea of using an infinite-capacity edge to encode the incompatibility constraint seems good, so let's start there and try to derive the rest. Let p be a Packers fan and b be a Bears fan, and assume they are incompatible. We want to encode the incompatibility with an infinite capacity edge from p to b. Encoding the incompatibility means that cutting this edge should correspond to inviting both p and b. That is, for a cut S, T, when $p \in S$ and $b \in T$, this should correspond to inviting both p and b. This suggests we try a new rule for deciding who to invite. Instead of inviting precisely the friends in S, we invite the Packers fans in S and the Bears fans in T. Does this work?

We can implement friend values similarly as before. For a Packers fan vertex v, we make an edge from s to v whose value is that friend's value. For a Bears fan vertex v, we make an edge from v to t whose value is that friend's value. These edges encode the costs of not inviting these friends to the party. An infinite capacity edge between every pair of incompatible Packers and Bears fans (always oriented from Packers fan to Bears fan) encodes the incompatibility relation. Thus we have captured all the aspects of the problem, so we just need to write this solution up.

We mention, however, that this network is an instance of the kind of network that appears when reducing project selection to minimum cut. This suggests we can reduce our problem to project selection, and save some effort in the write-up. The following paragraphs present this idea.

Solution This problem can be cast as an instance of project selection. There is a project for every friend P_i that is a Packer fan, namely "inviting P_i," with a benefit equal to the value of P_i. There is tool for every friend B_j that is a Bears fan, namely "not inviting B_j," with a cost equal to the value of B_j. The tools needed for P_i are those B_j with which P_i is on bad terms. In that setup, the project selection requirement of buying all tools needed for a selected project exactly corresponds to the given requirement of allowing all possible combinations but the ones where we invite a P_i and a B_j that are on bad terms. Also, the net gain of the project selection equals the total value of the invited friends minus the sum of the values of all friends that are Bear fans. As the latter is a constant, an optimal project selection exactly corresponds to an optimal invitation plan, i.e., we invite P_i if the corresponding project is selected, and we invite B_j if the corresponding tool is not selected.

Up to constant factors, the running time is the one for project selection, which is the one for maximum flow on a graph with n vertices and m edges, where n denotes the total number of friends, and m the number of pairs (B_i, P_j) that are on bad terms. Using the strongly polynomial-time network flow algorithm mentioned in class, the resulting running time is $O(nm)$.

Problem 3

You are building a system consisting of n components. There are two possible suppliers for each component: Alpha and Omega. Alpha charges α_i for component i, and Omega charges ω_i. You'd like to spend as little money as possible, but also want to take the costs due to incompatibilities between components of different suppliers into account. In particular, if you buy components i and j from different suppliers, there is an incompatibility cost of $c(i, j)$.

Design an efficient algorithm to determine from which supplier you should buy the components so as to minimize the sum of the purchase costs and the incompatibility costs.

Stated another way, our goal in this problem is to find a partition $A \sqcup \Omega$ of the products $[n]$ (where A represents the set of products purchased from vendor Alpha and Ω those purchased from vendor Omega) so that the total cost of the partition, given by the following expression, is minimized:

$$\sum_{i \in A} \alpha_i + \sum_{j \in \Omega} \omega_j + \sum_{(i,j) \in A \times \Omega} c(i, j).$$

Note that this objective is strongly reminiscent of the objective in image segmentation, and in fact almost reduces to image segmentation directly. In image segmentation, the objective we would like to minimize is

$$\sum_{i \in F} b_i + \sum_{j \in B} f_j + \sum_{(i,j) \in F \times B : i \sim j} c,$$

where

- F and B are the sets of pixel indices we take to be foreground and background pixels, respectively;
- f_i and b_i are the likelihoods of pixel i being in the foreground or background, respectively;
- $i \sim j$ is a relation that indicates whether i and j correspond to adjacent pixels; and
- c is the cost of putting adjacent pixels in different regions.

In particular, if we identify A and Ω with F and B, and α_i and ω_i with b_i and f_i, respectively, the only two differences in the objective lie in the last summand. First, in the product problem we would like to include a penalty for any pair of products that we buy from different vendors, and not just "adjacent" products. Second, we must allow this penalty to vary for each pair of products instead of just including a flat penalty c that is inflicted for every pair.

The first difference could be taken care of by defining $\sim$ so that every pair of products is "adjacent", but the second prevents us from reducing directly to image segmentation. However, a small tweak to the reduction from image segmentation to minimum cut transforms it into a reduction from the product problem to minimum cut instead.

Define a flow network $G = (V, E)$, where the vertices are $V = \{s, t\} \sqcup [n]$. Exactly as in image segmentation, we will add edges to this graph so that any s-t cut (S, T) induces a purchasing strategy given by $A = S \backslash \{s\}$ and $\Omega = T \backslash \{t\}$. Also exactly as in image segmentation, we can then add edges from s to each $i \in [n]$ of capacity ω_i, and edges from each $i \in [n]$ to t of capacity α_i to capture the $\sum_{i \in A} \alpha_i$ and $\sum_{j \in \Omega} \omega_j$ terms in the objective (since if $i \in A$, the edge of weight α_i crosses the cut and if $i \in \Omega$, the edge of weight ω_i crosses the cut). To take care of the final

$\sum_{(i,j)\in A\times\Omega} c(i,j)$ term, for every $i < j \in [n]$ we add a bidirectional edge of weight $c(i,j)$ between i and j, so that if they are on opposite sides of the cut (i.e., we purchase them from different vendors), we add $c(i,j)$ to the capacity of the cut, or 0 otherwise.

Hence, we can obtain a minimum-cost purchasing strategy for the original problem by constructing G, running a maximum flow algorithm on G to obtain a minimum cut (S,T), and outputting $S\backslash\{s\}$ and $T\backslash\{t\}$ as the sets of products we buy from vendors Alpha and Omega, respectively. Since G has $O(n)$ nodes and $O(n^2)$ edges, we can construct G in time $O(n^2)$, compute a maximum flow on G in $O(n^3)$ time using the $O(|V|\cdot|E|)$-time algorithm mentioned in class, obtain a minimum cut from the maximum flow in $O(|E|+|V|)=O(n^2)$ time, and compute the output sets in $O(n)$ time for a total complexity of $O(n^3)$.

Problem 4

You are organizing the carpool to work for you and your friends. Let the people be labeled $S = \{p_1, \ldots, p_k\}$. We say that the *total driving obligation* of p_j over a set of days is the expected number of times that p_j would have driven, had a driver been chosen uniformly at random from among the people going to work each day. More concretely, suppose the carpool plan lasts for d days, and on the ith day a subset $S_i \subseteq S$ of the people go to work. Then the above definition of the total driving obligation Δ_j for p_j can be written as $\Delta_j = \sum_{i:p_j \in S_i} \frac{1}{|S_i|}$. Ideally, we'd like to require that p_j drives at most Δ_j times; however, Δ_j may not be an integer.

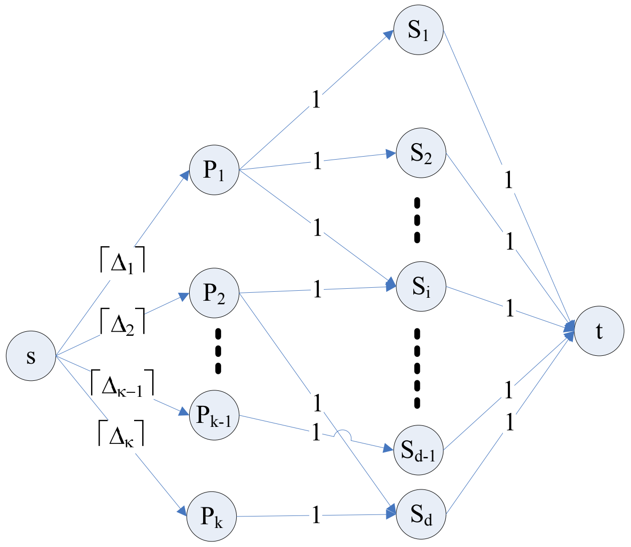
So let's say that a driving schedule is a choice of a driver for each day -— i.e., a sequence $p_{i_1}, p_{i_2}, \ldots, p_{i_d}$ with $p_{i_t} \in S_t$ -— and that a fair driving schedule is one in which each p_j is chosen as the driver on at most $\lceil \Delta_j \rceil$ days.

a) Prove that for any sequence of sets $S_1, \ldots, S_d$, there exists a fair driving schedule.

b) Design an algorithm to compute a fair driving schedule in time polynomial in k and d.

Part (a)

We can reduce the construction of a fair schedule to the following network flow problem. There is a source node s and a sink node t in the graph. Let us consider each person p_j as a node connected to s. Because p_j can drive at most $\lceil \triangle_j \rceil$ times, we add an edge (s, p_j) for each p_j to the graph. This edge has a capacity of $\lceil \triangle_j \rceil$. Similarly, let us consider each day S_i as a node connected to t and add an edge (S_i, t) for each S_i to the graph. This edge has a capacity of 1 because at most one person can drive at that day. Moreover, for each pair (p_j, S_i) where $p_j \in S_i$, we add an edge (p_j, S_i) whose capacity is 1 to indicate that p_j can drive once on the i^{th} day. The final flow network is as follows:



We divide the proof of the existence of a fair schedule into two parts. First, we prove that a fair schedule exists if and only if there exists a flow of value d in the network. Second, we proceed to show that such a flow exists.

Claim 1. *A fair schedule exists if and only if there exists a flow of value d in above network.*

Proof. First, if there is a fair schedule, we can construct the following flow. If in the *driving schedule*, a person p_j is chosen as the driver on the i^{th} day, then we send one unit of flow along the path s, p_j, S_i, t; we do this for all d days. Since the *driving schedule* satisfies all the constraints, the flow satisfies the capacity requirements and it sends d units of flow out of s and into t.

Conversely, if there is a flow of value d, then we can construct a *driving schedule*. As all capacities are integral, there is a feasible flow of value d in which all flow values are integers. Therefore, if the edge (p_j, S_i) carries a unit of flow, we have person p_j drive on day i. Because of the capacities, each person p_j will drive at most $\lceil \triangle_j \rceil$ times and for each day, there will be a person who drives on that day. So the driving schedule is fair. □

Claim 2. *There exists a flow of value d in above network.*

Proof. We exhibit a flow of value d. We send $\triangle_i$ units of flow from s to every p_i, $1/|S_j|$ units of flow from every p_i to S_j if there is an edge between them, and 1 unit of flow from every S_j to t. By construction, capacity constraints are satisfied and the size of the flow is d. Flow conservation is satisfied at p_i because of the definition of $\triangle_i$, and is satisfied at S_j because

$$\sum_{i \text{ connected to } S_j} 1/|S_j| = 1.$$

□

Part (b)

Above we proved that the problem can be reduced to a network flow problem. The algorithm is immediate: we construct the flow network and apply an efficient network flow algorithms to it. The running time of the construction is $O(|V| + |E|) = O(k + d + kd)$. Since the value of the max-flow is at most d and all capacities are integral, the complexity of applying the Ford-Fulkerson scheme on the graph is $O(|E|d) = O(kd^2)$. In conclusion, the total complexity is $O(kd^2)$.