CS 577: Introduction to Algorithms Fall 2022

Homework 8

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

This homework covers applications of network flow. **Problem 3 must be submitted for grading by 2:29pm on 11/15.** Please refer to the homework guidelines on Canvas for detailed instructions.

Warm-up problems

1. Your friends are attending a convention and want to ask questions to the panelists. They worked together to create a set S of n different topics they would like to ask about. Each of your m friends has a VIP pass that guarantees them the ability to ask at most three questions each.

   Unfortunately, due to the dense subject matter of the convention, not all of your friends understand every topic well enough to ask about it. For each friend $i = 1, 2, \ldots, m$, they have a set S_i of topics they are capable of asking about. Finally, to make sure each topic is thoroughly addressed, the friends want to ask at least k different questions about each of the n topics. (Note that one person should not ask about the same topic more than once.) They are having trouble figuring out who should ask about which topics.

   (a) Design a polynomial-time algorithm that takes the input to an instance of this problem (the n topics, the sets S_i for each of the m friends, and the parameter k) and decides whether there is a way to ask k questions about each of the n topics, while each friend asks at most three questions each.

   (b) You show your friends a solution computed by your algorithm from (a), but to your surprise, one of them exclaims, "That won't do at all– the first topic is only asked about by Optimists!" You hadn't heard anything about optimists; this is an extra wrinkle they neglected to mention earlier.

   Each of your friends is either an Optimist or a Pessimist. This refers to their general attitude- they each fall into exactly one of these two categories and will behave that way for the entire convention, regardless of the topic they are asking about. To keep the panel's responses as fair as possible, your friends agree that there should be no topic for which all k questions come from people with the same attitude.

   Describe how to modify your algorithm from (a) into a new polynomial-time algorithm that decides whether there exists a solution satisfying all of the conditions from (a), plus the new requirements about optimists and pessimists.

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.

Regular problems

3. [Graded] 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.

4. Some of your friends with jobs out West decide they really need some extra time each day to sit in front of their laptops, and the morning commute from Woodside to Palo Alto seems like the only option. So they decide to carpool to work.

   Unfortunately, they all hate to drive, so they want to make sure that any carpool arrangement they agree upon is fair, and doesn't overload any individual with too much driving. Some sort of simple round-robin scheme is out, because none of them goes to work every day, and so the subset of them in the car varies from day to day.

   Here's one way to define fairness. 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 $p_{i_t} \in S_t$. 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.

5. A *vertex cover* of a graph $G = (V, E)$ is a collection of vertices $C \subseteq V$ such that every edge $e \in E$ has at least one vertex in C.

   Show that for bipartite graphs, the minimum size of a vertex cover equals the maximum size of a matching.

Challenge problem

6. Here is a variant of the game "Six Degrees of Kevin Bacon."

   You start with a set X of n actresses and a set Y of n actors, and two players P_0 and P_1. Player P_0 names an actress $x_1 \in X$, player P_1 names an actor y_1 who has appeared in a movie with x_1, player P_0 names an actress x_2 who has appeared in a movie with y_1, and so on. Thus, P_0 and P_1 collectively generate a sequence $x_1, y_1, x_2, y_2, \ldots$ such that each actor/actress in the sequence has costarred with the actress/actor immediately preceding. A player $P_i (i = 0, 1)$ loses when it is P_i's turn to move, and she cannot name a member of her set who hasn't been named before.

   Suppose you are given a specific pair of such sets X and Y, with complete information on who has appeared in a movie with whom. A *strategy* for P_i in our setting is an algorithm that takes a current sequence $x_1, y_1, x_2, y_2, \ldots$ and generates a legal next move for P_i (assuming it's P_i's turn to move). Design a polynomial-time algorithm that, given some instance of the game, decides at the start of the the game which of the two players can force a win.

Programming problem

7. SPOJ problem Bank robbery (problem code BANKROB).

CS 577: Introduction to Algorithms Fall 2022

Homework 8 Solutions to Warm-up Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

Problem 1

Your friends are attending a convention and want to ask questions to the panelists. They worked together to create a set S of n different topics they would like to ask about. Each of your m friends has a VIP pass that guarantees them the ability to ask at most three questions each.

Unfortunately, due to the dense subject matter of the convention, not all of your friends understand every topic well enough to ask about it. For each friend $i = 1, 2, \ldots, m$, they have a set S_i of topics they are capable of asking about. Finally, to make sure each topic is thoroughly addressed, the friends want to ask at least k different questions about each of the n topics (Note that one person should not ask about the same topic more than once). They are having trouble figuring out who should ask about which topics.

a) Design a polynomial-time algorithm that takes the input to an instance of this problem (the n topics, the sets S_i for each of the m friends, and the parameter k) and decides whether there is a way to ask k questions about each of the n topics, while each friend asks at most three questions each.

b) You show your friends a solution computed by your algorithm from (a), but to your surprise, one of them exclaims, "That won't do at all– the first topic is only asked about by Optimists!" You hadn't heard anything about optimists; this is an extra wrinkle they neglected to mention earlier.

Each of your friends is either an Optimist or a Pessimist. This refers to their general attitude- they each fall into exactly one of these two categories and will behave that way for the entire convention, regardless of the topic they are asking about. To keep the panel's responses as fair as possible, your friends agree that there should be no topic for which all k questions come from people with the same attitude.

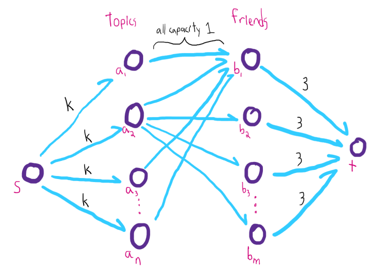
Describe how to modify your algorithm from (a) into a new polynomial-time algorithm that decides whether there exists a solution satisfying all of the conditions from (a), plus the new requirements about optimists and pessimists.

Part (a)

We want to decide whether we can discuss n topics with our m friends (of which friend i can only ask about topics in S_i), asking at least k questions about each topic, and with no person asking more than three questions. We can do this by looking at a maximum flow on a particular network. Note that if there is a solution that asks about each topic at least k times, then there must be a solution that asks about each topic exactly k times; we restrict ourselves both here and in part (b) to solutions where each topic is discussed 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 topics that need to be checked, while on the right, there is a vertex for each friend. We call the left (topic) vertices $a_1 \cdots a_n$, and the right (friend) vertices $b_1 \cdots b_m$. We include an edge of capacity 1 from a vertex a_i on the left to a vertex b_j on the right if friend j is able to ask about topic i (if S_j contains i). 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 topic vertex). The sink t has an edge of capacity 3 from every vertex on the right (i.e., every friend vertex).

As an example, consider the situation where friend 1 can ask about any topic (S_1 contains $1, 2, \ldots, n$) and every other friend only knows about topic 2 (all S_i contain 2). The network would be built as follows:



Correctness. We claim that the value of a maximum flow in G is kn if and only if there is a way to ask k questions about each of the n topics, with each friend asking at most 3 questions (each on a different topic).

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 friends to topics, i.e., an assignment that is both satisfactory, in that it asks enough, and valid, in that no one person asks more than three questions. Then we can construct a flow of value kn in G as follows: Push k units of flow from s to each topic vertex, then one unit from each topic vertex to each of the k friends that are asking about it (there must be exactly k such friends for each topic, since this is a satisfactory assignment); for each friend, we then push a number of units equal to the number of questions that friend is making (which must be 0, 1, 2, or 3 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 topic vertices pushing flow into exactly k of the edges that connect it to the friends. These are the friends we use to ask about that topic. Every topic is asked

about by precisely k different friends, as required. Moreover, since each friend vertex on the right can push at most 3 units of flow to t, no friend asks more than 3 questions. Thus, the assignment works.

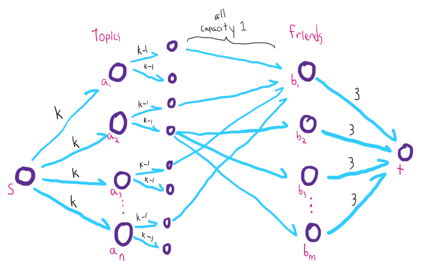
Analysis. There are $1 + m + n + 1$ vertices, and the number of edges is bounded by $O(mn)$ as there are up to mn edges connecting the topic nodes to the friend nodes. Thus, he network can be constructed in time $O(mn)$. We can then apply any of the polynomial-time algorithms we know for computing the value of a maximum flow and check whether it equals kn. If we use the $O(|V| \cdot |E|)$ algorithm to compute a maximum flow, we get a final running time of $O((m + n)mn)$, which is polynomial in m and n.

Part (b)

We can handle the extra wrinkle – that every friend is an Optimist or Pessimist, and no topic can be asked about by only optimists or only pessimists – by augmenting the graph we constructed in part (a). To model this additional constraint, we add two more vertices for each topic (a total of $2n$ additional vertices). These vertices are interposed in a new layer between the topic vertices and the friend vertices (i.e., between the left and right sides of G). Remove the edges from (a) connecting the n topic nodes directly to the m friend nodes.

For each topic vertex $j = 1, 2, \ldots, n$ add an edge of capacity $k - 1$ to vertices $2j - 1$ and $2j$ in the intermediate layer. This will represent the subsets of questions about j asked by Optimists and Pessimists, respectively. From each odd-numbered intermediate node (referenced earlier by $2j - 1$), create an edge of capacity 1 to each of the right (person) nodes that can ask about the topic and are considered optimists. Do the same to connect each even-numbered intermediate node (referenced earlier by $2j$) with an edge of capacity 1 to each of the right (person) nodes that can ask about the topic and are considered pessimists.

Continuing the example from (a), say we are given the additional information that friend 1 is an optimist and all other friends are pessimists. The network would now be built as follows:



Correctness. Thus, a topic vertex cannot send all k units of flow that it receives through only its corresponding optimist intermediate node or only its pessimist intermediate node. This prevents any topic from having all k of its questions asked by people of the same attitude. Once again, there is a way of asking enough questions subject to all of the constraints if and only if a maximum flow in this network has value kn.

Analysis. In this modification, there are $1 + n + 2n + m + 1$ vertices. The number of edges is still bounded by $O(mn)$ from the set of edges between the intermediate and friend nodes. 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. As before, this takes time $O((m + n)mn)$ with the $O(|V| \cdot |E|)$ algorithm for computing a maximum flow.

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 $O(nm)$.

CS 577: Introduction to Algorithms Fall 2022

Homework 8 Solutions to Regular Problems

Instructor: Dieter van Melkebeek TA: Nicollas Mocelin Sdroievski

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.

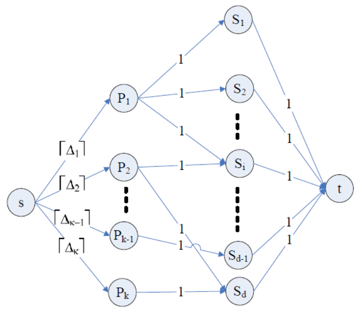
So let's say that a driving schedule is a choice of a driver for each day -— i.e., a sequence $p_{i_1}, p_{i_2}, \ldots, p_{i_d}$ with $p_{i_t} \in S_t$ -— and that a fair driving schedule is one in which each p_j is chosen as the driver on at most $\lceil\Delta_j\rceil$ days.

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 $\lceil\Delta_j\rceil$ times, we add an edge (s, p_j) for each p_j to the graph. This edge has a capacity of $\lceil\Delta_j\rceil$. 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

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

□

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 algorithm to it. The running time of the construction is $O(|V| + |E|) = O(k + d + kd)$. Then, computing a max-flow using the $O(|V| \cdot |E|)$ algorithm from class takes time $O((k+d)kd)$, which is polynomial in k and d.

Problem 5

A *vertex cover* of a graph $G = (V, E)$ is a collection of vertices $C \subseteq V$ such that every edge $e \in E$ has at least one vertex in C.

Show that for bipartite graphs, the minimum size of a vertex cover equals the maximum size of a matching.

Fix a bipartite graph G. Let c denote the minimum number of vertices in a vertex cover for G. Let m denote the size of a maximum matching in G.

1. $c \geq m$.

   Let C be an arbitrary vertex cover, and M an arbitrary matching. Consider the edges in M. We know these are all disjoint, meaning no two edges share any vertex. C must include at least one vertex for each of these disjoint edges, so $|C| \geq |M|$. Since our choices for C and M were arbitrary, this is true for all vertex covers and all matchings; hence, it follows that $c \geq m$.

2. $c \leq m$.

   Suppose the two bipartite components of G are L (left) and R (right). Consider the matching network corresponding to G: Connect the source s to every vertex in L with unit capacity edges, connect all vertices in R to the sink t with unit capacity edges, and direct every edge in G from left to right with infinite capacity. Note that since the incoming capacity for any vertex in L and the outgoing capacity for any vertex in R is exactly 1, no edge in G can ever carry more than 1 unit of flow; thus, the maximum flow in this network corresponds to a maximum matching with size equal to the value of the maximum flow. Applying the max-flow-min-cut theorem, the capacity of a minimum cut in this network equals m. Consider any cut (S, T) of finite capacity. Since all edges from L to R have infinite capacity, there can be no such edge with the left vertex in S and the right vertex in T. Therefore, every edge in G has either its left vertex in T, its right vertex in S, or both. Therefore, $C = (L \cap T) \cup (R \cap S)$ is a vertex cover for G. Moreover, the edges that cross the cut from S to T are precisely those that go from s to a vertex in $L \cap T$, or from a vertex in $R \cap S$ to t. Therefore, $|C| = c(S, T)$. Since this shows that every finite capacity cut has a corresponding vertex cover of equal value, we may conclude that c is no more than the capacity of a minimum cut, i.e., $c \leq m$.

These two inequalities, combined, prove that $c = m$.

COMP SCI 577 Homework 08 Problem 3

Network Flow

Ruixuan Tu

rtu7@wisc.edu

University of Wisconsin-Madison

15 November 2022

Algorithm

For the minimum cut algorithm, we reference that on page 3 of the handout on 10 November 2022.

For Algorithm 1, we first create one vertex labeled i for every $i \in [n]$; then we create the source vertex s and the sink vertex t and connect every (i, j) $(i, j \in [n])$ pair in network with each other, s, and t as in Figure 1. We find the minimum cut on this network. After that, we have a loop to put all components in T to A and all components in S to O. At this point, we have finished.

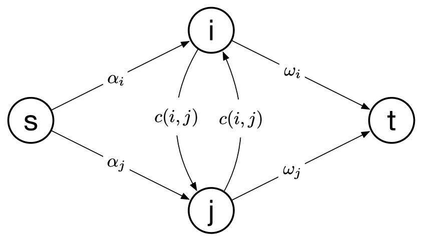


Figure 1: Network N for every pair of components (i, j)

```
Function BuyComponents(n, α, ω, c):
    Input: components $i \in [n]$; $\alpha_i \in \alpha$ Alpha charges for component $i$; $\omega_i \in \omega$ Omega
           charges for component $i$; $c(i,j)$ incompatibility cost for components $i$ and $j$
           from different suppliers
    Output: 2-partition of $[n]$ as component sets $A$ for Alpha and $O$ for Omega
            minimizing $(*) := \sum_{i \in A} \alpha_i + \sum_{j \in O} \omega_j + \sum_{\substack{i \sim j \\ (i,j) \in A \times O}} c(i,j)$
    Denote $c(S,T)$ with $S$ and $T$ being set to be the capacity function of an $st$-cut,
      instead of the cost function $c(i,j)$ with $i, j \in [n]$;
    Construct network $N$ such that $(*) = c(S,T)$ where $S = A \cup \{s\}$ and $T = O \cup \{t\}$ as
      in Figure 1;
    Find the minimum $st$-cut $\min c(S,T)$ and the partition $(S,T)$;  // referenced, by
      max-flow algorithm and searching in residual network
    $A \leftarrow \emptyset$;
    $O \leftarrow \emptyset$;
    foreach $i \in [n]$ do
        if $i \in T$ then
            $A \leftarrow A \cup \{i\}$;
        else if $i \in S$ then // must be satisfied without check
            $O \leftarrow O \cup \{i\}$;
    return $A, O$;
```

Algorithm 1: Main routine

Proof

Correctness

Claim: For network N, $(*) = c(S,T)$.

Proof: As $c(S,T)$ is a partition of the network N, $i \in [n]$ is in either S or T as if we do not cut so, then there is still flow in the residual network on the path (V,E) where $V = \{s, i_n, t\}$ and $E = \{\alpha_i, \omega_i\}$, which is a contradiction to the property of partition. So that we have partitioned V_N into S and T, i.e., $A \cup \{s\}$ and $O \cup \{t\}$, while partitioned $[n]$ into A and O. For the flow already pushed through α_i, we must use another ω_j to redirect it, which passes through $c(i,j)$. We have the capacities of edges on the cut be identical to $(*)$ as we must pay all the costs as capacities along the edges, regardless if it is optimal. ■

Corollary: (min-cut min-cost property) $\min(*) = \min c(S,T)$

Proof: It is obvious from the first claim, as applying a function, i.e., min, on two identical functions does not change the identity of values of both sides. ■

Claim: $i \in A$ iff $i \in [n]$ and $i \in T$.

One side: $i \in A$ if $i \in [n]$ and $i \in T$. **Proof**: Recall that we have proved from the first claim that there must be a cut on either α_i or ω_i. As $i \in [n]$ and $i \in T$, $\alpha_i \in S \times T \cup E$, i.e., the min-cut. So that $i \in A$, as we want to pay for the product from the Alpha company by the min-cut min-cost property. $\square$

Another side: $i \in [n]$ and $i \in T$ if $i \in A$. **Proof**: As $i \in A$, $\alpha_i \in S \times T \cup E$, by the min-cut min-cost property, so i must be in T. As $A \subset [n]$ by the definition that (A, O) is a partition of $[n]$, $i \in A$ implies that $i \in [n]$. $\blacksquare$

Claim: $i \in O$ iff $i \in [n]$ and $i \in S$.

One side: $i \in O$ if $i \in [n]$ and $i \in S$. **Proof**: This is similar to the proof of the second claim. As $i \in [n]$ and $i \in S$, $\omega_i \in S \times T \cup E$. So that $i \in O$, as we want to pay for the product from the Omega company by the min-cut min-cost property. $\square$

Another side: $i \in [n]$ and $i \in S$ if $i \in O$. **Proof**: This is similar to the proof of the second claim. As $i \in O$, $\omega_i \in S \times T \cup E$, by the min-cut min-cost property, so i must be in S. As $O \subset [n]$ by the definition that (A, O) is a partition of $[n]$, $i \in O$ implies that $i \in [n]$. $\blacksquare$

Termination

We are running the minimum cut algorithm on an arbitrary network we have built, and this algorithm we have referenced must end, and there is no more recursive part (only loop) of Algorithm 1. Therefore, Algorithm 1 should always terminate regardless of the input. $\blacksquare$

Complexity

According to our referenced minimum cut algorithm, it costs $O(\underset{\text{\# vertices}}{n} \cdot \underset{\text{\# edges}}{m})$. For this problem, we redefine n to be the number of components. For every vertex i of one component, we have $\underset{s \text{ and } t}{2} + \underset{\text{other components } j}{(n-1)} = O(n)$ edges, and for vertices s, t, they connect to $[n]$ by identical number of edges $n = O(n)$. Then the total number of edges is $\underset{\text{\# components}}{n} \cdot O(n) + 2O(n) = O(n^2)$. The total number of vertices is $\underset{s \text{ and } t}{2} + \underset{\text{\# components}}{n} = O(n)$. So the time complexity for the minimum cut algorithm is $O(n) \cdot \underset{O(m)}{O(n^2)} = O(n^3)$.

For the loop following the minimum cut in `foreach`, there is one implicit loop in checking if $i \in T$ or $i \in S$, but the implicit loop can be optimized out by using `foreach i in T` and checking if $i \in [n]$ as the main loop. So that the remaining $i \in [n]$ are all in S, as from the property of partition. Thus, the main loop has the time complexity $\underset{\text{\# components}}{O(n)}$.

For the whole Algorithm 1, the time complexity is $O(n^3) + O(n) = O(n^3)$.