6 Network Flow

1. Suppose you are given a directed graph $G = (V, E)$, with a positive integer capacity c_e on each edge e, a designated source $s \in V$, and a designated sink $t \in V$. You are also given a maximum s-t flow in G, defined by a flow value f_e on each edge e. The flow $\{f_e\}$ is *acyclic*: there is no cycle in G on which all edges carry positive flow.

   Now, suppose we pick a specific edge $e^* \in E$ and reduce its capacity by 1 unit. Show how to find a maximum flow in the resulting capacitated graph in time $O(m)$, where m is the number of edges in G.

2. Consider the following problem. You are given a flow network with unit-capacity edges: it consists of a directed graph $G = (V, E)$, a source $s \in V$, and a sink $t \in V$; and $c_e = 1$ for every $e \in E$. You are also given a parameter k.

   The goal is delete k edges so as to reduce the maxmimum s-t flow in G by as much as possible. In other words, you should find a set of edges $F \subseteq E$ so that $|F| = k$ and the maximum s-t flow in $G' = (V, E - F)$ is as small as possible subject to this.

   Give a polynomial-time algorithm to solve this problem.

   Solution. If the minimum s-t cut has size $\leq k$, then we can reduce the flow to 0. Otherwise, let $f > k$ be the value of the maximum s-t flow. We identify a minimum s-t cut (A, B), and delete k of the edges out of A. The resulting subgraph has a maximum flow value of at most $f - k$.

   But we claim that for any set of edges F of size k, the subgraph $G' = (V, E - F)$ has an s-t flow of value at least $f - k$. Indeed, consider any cut (A, B) of G'. There are at least f edges out of A in G, and at most k have been deleted, so there are at least $f - k$ edges out of A in G'. Thus, the minimum cut in G' has value at least $f - k$, and so there is a flow of at least this value.

3. Suppose you're looking at a flow network G with source s and sink t, and you want to be able to express something like the following intuitive notion: some nodes are clearly on the "source side" of the main bottlenecks; some nodes are clearly on the "sink side" of the main bottlenecks; and some nodes are in the middle. However, G can have many minimum cuts, so we have to be careful in how we try making this idea precise.

   Here's one way to divide the nodes of G into three categories of this sort.

   - We say a node v is *upstream* if for all minimum s-t cuts (A, B), we have $v \in A$ — that is, v lies on the source side of every minimum cut.
   - We say a node v is *downstream* if for all minimum s-t cuts (A, B), we have $v \in B$ — that is, v lies on the sink side of every minimum cut.
   - We say a node v is *central* if it is neither upstream nor downstream; there is at least one minimum s-t cut (A, B) for which $v \in A$, and at least one minimum s-t cut (A', B') for which $v \in B'$.

   Give an algorithm that takes a flow network G, and classifies each of its nodes as being upstream, downstream, or central. The running time of your algorithm should be within in a constant factor of the time required to compute a *single* maximum flow.

   Solution. Consider the cut (A^*, B^*) found by performing breadth-first search on the residual graph G_f at the end of any maximum flow algorithm. We claim that a node v is upstream if and only if $v \in A^*$. Clearly, if v is upstream, then it must belong to A^*; since otherwise, it lies on the sink-side of the minimum cut (A^*, B^*). Conversely, suppose $v \in A^*$ were not upstream. Then there would be a minimum cut (A', B') with $v \in B'$. Now, since $v \in A^*$, there is a path P in G_f from s to v. Since $v \in B'$, this path must have an edge (u, w) with $u \in A'$ and $w' \in B'$. But this is a contradiction, since no edge in the residual graph can go from the source side to the sink side of any minimum cut.

   A completely symmetric argument shows the following. Let B_* denote the nodes that can reach t in G_f, and let $A_* = V - B_*$. Then (A_*, B_*) is a minimum cut, and a node w is downstream if and only if $w \in B_*$.

   Thus, our algorithm is to compute a maximum flow f, build G_f, and use breadth-first search to find the sets A^* and B_*. These are the upstream and downstream nodes respectively; the remaining nodes are central.

4. Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and non-negative edge capacities $\{c_e\}$. Give a polynomial time algorithm to decide whether G has a *unique* minimum s-t cut. (I.e. an s-t of capacity strictly less than that of all other s-t cuts.)

5. In a standard minimum s-t cut problem, we assume that all capacities are non-negative; allowing an arbitrary set of positive and negative capacities results in an NP-complete problem. (You don't have to prove this.) However, as we'll see here, it is possible

to relax the non-negativity requirement a little, and still have a problem that can be solved in polynomial time.

Let $G = (V, E)$ be a directed graph, with source $s \in V$, sink $t \in V$, and edge capacities $\{c_e\}$. Suppose that for every edge e that has neither s nor t as an endpoint, we have $c_e \geq 0$. Thus, c_e can be negative for edges e that have at least one end equal to either s or t. Give a polynomial-time algorithm to find an s-t cut of minimum value in such a graph. (Despite the new non-negativity requirements, we still define the value of an s-t cut (A, B) to be the sum of the capacities of all edges e for which the tail of e is in A and the head of e is in B.)

6. Let M be an $n \times n$ matrix with each entry equal to either 0 or 1. Let m_{ij} denote the entry in row i and column j. A *diagonal entry* is one of the form m_{ii} for some i.

   Swapping rows i and j of the matrix M denotes the following action: we swap the values m_{ik} and m_{jk} for $k = 1, 2, \ldots, n$. Swapping two columns is defined analogously.

   We say that M is *re-arrangeable* if it is possible to swap some of the pairs of rows and some of the pairs of columns (in any sequence) so that after all the swapping, all the diagonal entries of M are equal to 1.

   (a) Give an example of a matrix M which is not re-arrangeable, but for which at least one entry in each row and each column is equal to 1.

   (b) Give a polynomial-time algorithm that determines whether a matrix M with 0-1 entries, is re-arrangeable.

7. You're helping to organize a class on campus that has decided to give all its students wireless laptops for the semester. Thus, there is a collection of n wireless laptops; there is also have a collection of n wireless *access points*, to which a laptop can connect when it is in range.

   The laptops are currently scattered across campus; laptop ℓ is within range of a *set* S_ℓ of access points. We will assume that each laptop is within range of at least one access point (so the sets S_ℓ are non-empty); we will also assume that every access point p has at least one laptop within range of it.

   To make sure that all the wireless connectivity software is working correctly, you need to try having laptops make contact with access points, in such a way that each laptop and each access point is involved in at least one connection. Thus, we will say that a *test set* T is a collection of ordered pairs of the form (ℓ, p), for a laptop ℓ and access point p, with the properties that

   (i) If $(\ell, p) \in T$, then ℓ is within range of p. (I.e. $p \in S_\ell$).

   (ii) Each laptop appears in at least one ordered pair in T.

   (iii) Each access point appears in at least one ordered pair in T.

This way, by trying out all the connections specified by the pairs in T, we can be sure that each laptop and each access point have correctly functioning software.

The problem is: Given the sets S_ℓ for each laptop (i.e. which laptops are within range of which access points), and a number k, decide whether there is a test set of size at most k.

Example: Suppose that $n = 3$; laptop 1 is within range of access points 1 and 2; laptop 2 is within range of access point 2; and laptop 3 is within range of access points 2 and 3. Then the set of pairs

> (laptop 1, access point 1), (laptop 2, access point 2),
> (laptop 3, access point 3)

would form a test set of size three.

(a) Give an example of an instance of this problem for which there is no test set of size n. (Recall that we assume each laptop is within range of at least one access point, and each access point p has at least one laptop within range of it.)

(b) Give a polynomial-time algorithm that takes the input to an instance of this problem (including the parameter k), and decides whether there is a test set of size at most k.

8. Back in the euphoric early days of the Web, people liked to claim that much of the enormous potential in a company like *Yahoo!* was in the "eyeballs" — the simple fact that it gets millions of people looking at its pages every day. And further, by convincing people to register personal data with the site, it can show each user an extremely targeted advertisement whenever he or she visits the site, in a way that TV networks or magazines couldn't hope to match. So if the user has told *Yahoo!* that they're a 20-year old computer science major from Cornell University, the site can throw up a banner ad for apartments in Ithaca, NY; on the other hand, if they're a 50-year-old investment banker from Greenwich, Connecticut, the site can display a banner ad pitching Lincoln Town Cars instead.

   But deciding on which ads to show to which people involves some serious computation behind the scenes. Suppose that the managers of a popular Web site have identified k distinct *demographic groups* $G_1, G_2, \ldots, G_k$. (These groups can overlap; for example G_1 can be equal to all residents of New York State, and G_2 can be equal to all people with a degree in computer science.) The site has contracts with m different *advertisers*, to show a certain number of copies of their ads to users of the site. Here's what the contract with the ith advertiser looks like:

- For a subset $X_i \subseteq \{G_1, \ldots, G_k\}$ of the demographic groups, advertiser i wants its ads shown only to users who belong to at least one of the demographic groups in the set X_i.
- For a number r_i, advertiser i wants its ads shown to at least r_i users each minute.

Now, consider the problem of designing a good *advertising policy* — a way to show a single ad to each user of the site. Suppose at a given minute, there are n users visiting the site. Because we have registration information on each of these users, we know that user j (for $j = 1, 2, \ldots, n$) belongs to a subset $U_j \subseteq \{G_1, \ldots, G_k\}$ of the demographic groups. The problem is: is there a way to show a single ad to each user so that the site's contracts with each of the m advertisers is satisfied for this minute? (That is, for each $i = 1, 2, \ldots, m$, at least r_i of the n users, each belonging to at least one demographic group in X_i, are shown an ad provided by advertiser i.)

Give an efficient algorithm to decide if this is possible, and if so, to actually choose an ad to show each user.

Solution. We define a flow network $G = (V, E)$ as follows.

- There is a source s, vertices $v_1, \ldots, v_n$ for each person vertices $w_1, \ldots, w_m$ for each advertiser, and sink t.
- There is an edge of capacity 1 from v_i to each w_j for which person i belongs to a demographic group that advertiser j wants to target.
- There is an edge with a capacity of 1 from s to each v_i; and for each j, there is an edge with lower bound r_j from w_j to t.
- Finally, the source has a demand of $-\sum_j r_j$, and the sink has a demand of $\sum_j r_j$. All other nodes have demand 0.

Now, if there is a valid circulation in this graph, then there is an integer circulation. In such a circulation, one unit of flow on the edge (v_i, w_j) means that we show an ad from advertiser j to person i. With this meaning, each advertiser shows their required number of ads to the appropriate people.

Conversely, if there is a way to satisfy all the advertising contracts, then we can construct a valid circulation as follows. We place a unit of flow on each edge (v_i, w_j) for which i is shown an ad from advertiser j; we put a flow on the edge (w_j, t) equal to the number of ads shown from advertiser j; and we put a unit of flow on each edge (s, v_i) for which person i sees an ad.

Thus, there is a valid circulation in this graph if and only if there is a way to satisfy all the advertising contracts; and the flow values in an integer-valued circulation can be used, as above, to decide which ads to show to which people.

9. Some of your friends have recently graduated and started a small company called WebExodus, which they are currently running out of their parents' garages in Santa

Clara. They're in the process of porting all their software from an old system to a new, revved-up system; and they're facing the following problem.

They have a collection of n software applications, $\{1, 2, \ldots, n\}$, running on their old system; and they'd like to port some of these to the new system. If they move application i to the new system, they expect a net (monetary) benefit of $b_i \geq 0$. The different software applications interact with one another; if applications i and j have extensive interaction, then the company will incur an expense if they move one of i or j to the new system but not both — let's denote this expense by $x_{ij} \geq 0$.

So if the situation were really this simple, your friends would just port all n applications, achieving a total benefit of $\sum_i b_i$. Unfortunately, there's a problem ...

Due to small but fundamental incompatibilities between the two systems, there's no way to port application 1 to the new system; it will have to remain on the old system. Nevertheless, it might still pay off to port some of the other applications, accruing the associated benefit and incurring the expense of the interaction between applications on different systems.

So this is the question they pose to you: which of the remaining applications, if any, should be moved? Give a polynomial-time algorithm to find a set $S \subseteq \{2, 3, \ldots, n\}$ for which the sum of the benefits minus the expenses of moving the applications in S to the new system is maximized.

Solution. We define a directed graph $G = (V, E)$ with nodes $s, v_1, v_2, \ldots, v_n$, where our sink t will correspond to v_1. Define an edge (s, v_i) of capacity b_i, and edges (v_i, v_j) and (v_j, v_i) of capacity p_{ij} for each $p_{ij} > 0$. Define $B = \sum_i b_i$. Now, let (X, Y) be the minimum s-v_1 cut in G. The capacity of (X, Y) is

$$\begin{aligned} c(X, Y) &= \sum_{i \in Y} b_i + \sum_{i \in X, j \in Y} p_{ij} \\ &= B - \sum_{i \in X} b_i + \sum_{i \in X, j \in Y} p_{ij} \end{aligned}$$

Thus, finding a cut of minimum capacity is the same as finding a set X maximizing $\sum_{i \in X} b_i - \sum_{i \in X, j \notin X} p_{ij}$.

10. Consider a variation on the previous problem. In the new scenario, any application can potentially be moved, but now some of the benefits b_i for moving to the new system are in fact *negative*: if $b_i < 0$, then it is preferable (by an amount quantifed in b_i) to keep i on the old system. Again, give a polynomial-time algorithm to find a set $S \subseteq \{1, 2, \ldots, n\}$ for which the sum of the benefits minus the expenses of moving the applications in S to the new system is maximized.

    Solution. Choose a number r larger than any $|b_i|$, and say that the *scaled benefit* b_{i0} of keeping application i on the old system is $b_{i0} = r$, while the scaled benefit of moving it is $b_{i1} = r + b_i$. So our problem is equivalent to that of finding a partition of the

applications that maximizes the sum of the scaled benefits minus the sum of p_{ij} for all pairs i and j that are split.

Define a directed graph $G = (V, E)$ with nodes $s, t, v_1, v_2, \ldots, v_n$, edges $(s, v_i), (v_i, s)$ of capacity b_{i0}, $(t, v_i), (v_i, t)$ of capacity b_{i1}, and $(v_i, v_j), (v_j, v_i)$ of capacity p_{ij}. Let $B = \sum_i (b_{i0} + b_{i1})$. Now, the capacity of an s-t cut (X, Y) can be written as

$$\begin{aligned} c(X,Y) &= \sum_{i \notin X} b_{i0} + \sum_{i \notin Y} b_{i1} + \sum_{i \in X, j \in Y} p_{ij} \\ &= B - \sum_{i \in X} b_{i0} - \sum_{i \in Y} b_{i1} + \sum_{i \in X, j \in Y} p_{ij} \end{aligned}$$

Thus, finding a cut of minimum capacity is the same as maximizing the objective function in the previous paragraph.

11. Consider the following definition. We are given a set of n countries that are engaged in trade with one another. For each country i, we have the value s_i of its budget surplus; this number may be positive or negative, with a negative number indicating a deficit. For each pair of countries i, j, we have the total value e_{ij} of all exports from i to j; this number is always non-negative. We say that a subset S of the countries is *free-standing* if the sum of the budget surpluses of the countries in S, minus the total value of all exports from countries in S to countries not in S, is non-negative.

    Give a polynomial-time algorithm that takes this data for a set of n countries, and decides whether it contains a non-empty free-standing subset that is not equal to the full set.

    Solution. We define the following flow network. There will be a node v_i for each country i, plus a source s and sink t. For each pair of nodes (v_i, v_j) with $e_{ij} > 0$, we include an edge of capacity e_{ij}. For each node v_i with $s_i > 0$, we include an edge (s, v_i) of capacity s_i and for each node v_i with $s_i < 0$, we include an edge (v_i, t) of capacity $-s_i$. Let

$$N = \sum_{i : s_i > 0} s_i.$$

    Now, consider an s-t cut (A, B), and write $S = A - \{s\}$, $T = B - \{t\}$. This cut's capacity is

$$\begin{aligned} c(A,B) &= \sum_{i \in T, s_i > 0} s_i + \sum_{i \in S, s_i < 0} -s_i + \sum_{i \in S, j \in T} e_{ij} \\ &= N - \sum_{i \in S, s_i > 0} s_i - \sum_{i \in S, s_i < 0} s_i + \sum_{i \in S, j \in T} e_{ij} \\ &= N - \left(\sum_{i \in S} s_i - \sum_{i \in S, j \in T} e_{ij} \right). \end{aligned}$$

    The last expression inside parentheses is precisely what is specified in the definition of *free-standing*.

Thus, there is a non-empty free-standing set if and only if there is a minimum s-t cut (A, B) in which the capacity is at most N and in which A contains at least one node other than s. (This latter part is crucial since otherwise we end up with an empty free-standing set S.) Hence, we need to be able to check whether $(\{s\}, V - \{s\})$ is the unique minimum cut. One can verify that this holds if and only if all nodes other than s have a path to t in the residual graph at the end of the Ford-Fulkerson algorithm.

12. $(*)$ In sociology, one often studies a graph G in which nodes represent people, and edges represent those who are friends with each other. Let's assume for purposes of this question that friendship is symmetric, so we can consider an undirected graph.

    Now, suppose we want to study this graph G, looking for a "close-knit" group of people. One way to formalize this notion would be as follows. For a subset S of nodes let $e(S)$ denote the number of edges in S, i.e., the number of edges that have both ends in S. We define the *cohesiveness* of S as $e(S)/|S|$. A natural thing to search for would be the set S of people achieving the maximum cohesiveness.

    (a.) Give a polynomial time algorithm that takes a rational number α and determines whether there exists a set S with cohesiveness at least α.

    (b.) Give a polynomial time algorithm to find a set S of nodes with maximum cohesiveness

    Solution. First, the maximum edge density of a subset has a numerator bounded by $\binom{n}{2}$ and a denominator bounded by n; thus, it can assume one of $O(n^3)$ possible values. Thus, if we can decide for a given rational Δ whether there is a subset of edge density at least Δ, we can find the maximum density using binary search over these possible values, incurring a multiplicative increase of $O(\log n)$ in the running time.

    Now, for a node i in G, let $d(i)$ denote its degree. Also, for a set $R \subseteq V$, let $e(R)$ denote the number of edges induced among nodes of R, and let $d(R)$ denote the number of edges with exactly one end in R.

    Consider the definition of a free-standing set from a previous question. We define an instance of the free-standing set problem in which $s_i = d(i) - 2\Delta$ and $e_{ij} = e_{ji} = 1$ for $(i, j) \in E$, $e_{ij} = e_{ji} = 0$ otherwise. Write

    $$f(R) = \sum_{i \in R} s_i - \sum_{i \in R, j \notin R} e_{ij}.$$

    We claim that R has edge density at least Δ if and only if it is a free-standing subset with respect to $\{s_i\}, \{e_{ij}\}$; that is, if and only if $f(R) \geq 0$. Indeed, observe that for any $R \subset V$, the quantity $\sum_{i \in R} d(i)$ is equal to $2e(R) + d(R)$. Thus,

    $$f(R) = 2e(R) + d(R) - 2|R|\Delta - d(R) = 2e(R) - 2|R|\Delta,$$

    and hence

    $$\frac{e(R)}{|R|} - \Delta = \frac{f(R)}{2|R|}.$$

13. Suppose we are given a directed network $G = (V, E)$ with a root node r, and a set of *terminals* $T \subseteq V$. We'd like to disconnect many terminals from r, while cutting relatively few edges.

    We make this trade-off precise as follows. For a set of edges $F \subseteq E$, let $q(F)$ denote the number of nodes $v \in T$ such that there is no r-v path in the subgraph $(V, E - F)$. Give a polynomial-time algorithm to find a set F of edges that maximizes the quantity $q(F) - |F|$. (Note that setting F equal to the empty set is an option.)

    Solution. We first observe the following fact about flow networks: there is a minimum s-t cut (A, B) such that there is a path from s to each node in A, passing only through nodes in A. (We'll call such a cut a *compact cut.*) Indeed, consider any s-t cut (A, B), and let A' be the set of nodes $v \in A$ such that there is an s-v path using only nodes in A. We claim that the cut $(A', B \cup (A - A')$ has no greater capacity, since the only edges out of A' go to nodes of B; none go to $(A - A')$.

    Now we consider the problem at hand. We can assume that in G, there is a path from r to each node in V; otherwise, we can initially re-define V to simply be the subset of nodes that are reachable from r. We can also assume $r \notin T$.

    We now construct a flow network G' by adding to G a sink node t, and a node (v, t) for each $v \in T$. All edges are given capacity 1. Let c^* be the value of the minimum cut in G. We know from the argument above that there is in fact a compact cut (A, B) of capacity c^*; let F^* be the set of edges out of A that do not go directly to t. Then every terminal in A can still be reached from r after the deletion of F^*, since (A, B) is a compact cut, so $q(F^*) = |B \cap T|$. The capacity $c(A, B)$ is equal to $c^* = |F^*| + |A \cap T| = |T| - (q(F^*) - |F^*|)$. So, in particular, there is a set of edges F^* for which $q(F^*) - |F^*| = |T| - c^*$.

    Now, consider a set of edges F' that achieves the optimal value of $q(F') - |F'|$. Let A denote the set of nodes that can be reached from r after the deletion of F', and let $B = (V - A) \cup \{t\}$. We know that each edge $e \in F'$ must have its tail in A and its head in $V - A$, for otherwise we could delete e from F' and obtain a better set of edges. Thus we have

    $$c^* \leq c(A, B) = |A \cap T| + |F'| = |T| - (q(F') - |F'|),$$

    and so $q(F') - |F'| \leq |T| - c^*$. It follows that the set of edges F^* defined in the previous paragraph is optimal.

14. 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; unfortunately, 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) Give an algorithm to compute a fair driving schedule with running time polynomial in k and d.

(c)$(*)$ One could expect k to be a much smaller parameter than d (e.g. perhaps $k = 5$ and $d = 365$). So it could be worth reducing the dependence of the running time on d even at the expense of a much worse dependence on k. Give an algorithm to compute a fair driving schedule whose running time has the form $O(f(k) \cdot d)$, where $f(\cdot)$ can be an arbitrary function.

Solution. **(a, b)** We define a graph $G = (V, E)$ with source s, vertices $v_1, \ldots, v_d$ for each day, vertices $w_1, \ldots, w_k$ for each person, and sink t. There is an edge of capacity 1 from s to each v_i, an edge of capacity 1 from v_i to each w_j with $p_j \in S_i$, and an edge of capacity $\lceil \Delta_j \rceil$ from w_j to t. We know there is a feasible fractional flow in this graph of value d, obtained by assigning a flow of value of $\frac{1}{|S_i|}$ to each edge (v_i, w_j), and flow values to all other edges as implied by the conservation condition. Thus there is a feasible integer flow, and the flow values on the edges of the form (v_i, w_j) define a fair driving schedule in the following way: p_j drives on day i if and only if $f(v_i, w_j) = 1$. This also gives a polynomial-time algorithm to compute the schedule.

(c) In place of $\{v_1, \ldots, v_d\}$, we could make a node $u_{S'}$ for every subset $S' \subseteq S$. There is an edge $(s, u_{S'})$ of capacity equal to the number of days $d_{S'}$ that S' was the set of people going to work. There is an edge of capacity $d_{S'}$ from to w_j for each $p_j \in u_{S'}$, and an edge of capacity $\lceil \Delta_j \rceil$ from w_j to t. This graph G has a size that is dependent only on k (the linear dependence on d comes simply from constructing it); thus we can compute a maximum flow in G in time depending only on k. Again, there is a flow of value d in G — assign flow value $\frac{d_{S'}}{|S'|}$ to each edge $(u_{S'}, w_j)$, and other flow values to ensure conservation — so there is an integer flow of value d. We can use the integer flow values on edges of the form $(d_{S'}, w_j)$ to define a fair driving schedule: of the days on which the set S' goes to work, p_j should be the driver on $f(d_{S'}, w_j)$ of them.

15. Suppose you live in a big apartment with a bunch of friends. Over the course of a year, there are a lot of occasions when one of you pays for an expense shared by some subset

of the apartment, with the expectation that everything will get balanced out fairly at the end of the year. For example, one of you may pay the whole phone bill in a given month, another will occasionally make communal grocery runs to the nearby organic food emporium; and a third might sometimes use a credit card to cover the whole bill at the local Italian-Indian restaurant, *Little Idli*.

In any case, it's now the end of the year, and time to settle up. There are n people in the apartment; and for each ordered pair (i, j) there's an amount $a_{ij} \geq 0$ that i owes j, accumulated over the course of the year. We will require that for any two people i and j, at least one of the quantities a_{ij} or a_{ji} is equal to 0. This can be easily made to happen as follows: if it turns out that i owes j a positive amount x, and j owes i a positive amount $y < x$, then we will subtract off y from both sides and declare $a_{ij} = x - y$ while $a_{ji} = 0$. In terms of all these quantities, we now define the *imbalance* of a person i to be the sum of the amounts that i is owed by everyone else, minus the sum of the amounts that i owes everyone else. (Note that an imbalance can be positive, negative, or zero.)

In order to restore all imbalances to 0, so that everyone departs on good terms, certain people will write checks to others; in other words, for certain ordered pairs (i, j), i will write a check to j for an amount $b_{ij} > 0$. We will say that a set of checks constitutes a *reconciliation* if for each person i, the total value of the checks received by i, minus the total value of the checks written by i, is equal to the imbalance of i. Finally, you and your friends feel it is bad form for i to write j a check if i did not actually owe j money, so we say that a reconciliation is *consistent* if, whenever i writes a check to j, it is the case that $a_{ij} > 0$.

Show that for any set of amounts a_{ij} there is always a consistent reconciliation in which at most $n - 1$ checks get written, by giving a polynomial-time algorithm to compute such a reconciliation.

Solution. Construct a graph $G = (V, E)$ where the nodes represent people that are either owed or owe money. Let there be an edge (u, v) if person u owes person v money and let the cost of this edge be the amount of money owed. Note that if (u, v) exists, then (v, u) does not exist since we may just adjust for the difference.

Let $\overline{G}$ be the undirected graph obtained from G by ignoring the directions of the edges. We repeatedly run BFS on $\overline{G}$ to find undirected cycles and eliminate them as specified below. We do this as follows:

```
Search for an undirected CYCLE While CYCLE != NULL (a cycle exists)
Find the edge in CYCLE with minimum cost.  Let this minimum cost
be stored in MINCOST and the correspoding edge in MINEDGE. For each
edge EDGE in CYCLE If EDGE has the same direction as MINEDGE reset
EDGE's cost to its current cost less MINCOST Else reset EDGE's cost
to its current cost plus MINCOST Endif Endfor Remove all edges whose
cost has been reduced to 0 (including MINEDGE) Search for a new
undirected CYCLE Endwhile
```

Note that each time through the while loop, we get rid of a cycle. Since there are m edges, we go through the outer loop at most m times. Also finding a cycle via BFS takes $O(m+n)$ time. Thus the overall running time is $O(m(m+n))$.

Note that in each iteration, we preserve all imbalances; so at the end we will have a reconciliation. Further, we preserve the direction of the edges since we modify according to the direction and only by the minimum cost edge in the cycle; thus, at the end, we will have a consistent reconciliation. Since $\overline{G}$ has no cycles at termination, it must be a tree or a forest, and therefore has at most $n-1$ edges. Thus we have produced consistent reconciliation in which at most $n-1$ checks get written.

Grader's Comments: It was necessary to search for undirected cycles in G, rather than directed cycles, since eliminating only the directed cycles in G will not necessarily reduce the number of edges to $n-1$.

There were three key points in the proof. First, imbalances are preserved as each cycle is processed. Second, directions of edges are never reversed, so the resulting reconciliation is consistent. Third, since all undirected cycles are eliminated, there are at most n-1 edges and at most $n-1$ checks written.

Some solutions simply ran a flow algorithm with a super-source attached to the people with positive imbalances, super-sink attached to the people with negative imbalances, and edges of infinite capacity joining pairs who owed each other initially; after this, they claimed that the resulting flow would be positive on at most $n-1$ edges. It seems hard to find a rule for choosing augmenting paths that will guarantee this, and also hard to prove that this property holds. (Of course, if one explicitly cancels cycles after finding the flow, then this would be correct by analogy with the above solution.)

16. Consider a set of mobile computing clients in a certain town who each need to be connected to one of several possible *base stations*. We'll suppose there are n clients, with the position of each client specified its by (x, y) coordinates in the plane. There are also k base stations; the position of each of these is specified by (x, y) coordinates as well.

    For each client, we wish to connect it to exactly one of the base stations. Our choice of connections is constrained in the following ways. There is a *range parameter* r — a client can only be connected to a base station that is within distance r. There is

also a *load parameter* L — no more than L clients can be connected to any single base station.

Your goal is to design a polynomial-time algorithm for the following problem. Given the positions of a set of clients and a set of base stations, as well as the range and load parameters, decide whether every client can be connected simultaneously to a base station, subject to the range and load conditions in the previous paragraph.

17. You can tell that cellular phones are at work in rural communities, from the giant microwave towers you sometimes see sprouting out of corn fields and cow pastures. Let's consider a very simplified model of a cellular phone network in a sparsely populated area.

    We are given the locations of n *base stations*, specified as points $b_1, \ldots, b_n$ in the plane. We are also given the locations of n cellular phones, specified as points $p_1, \ldots, p_n$ in the plane. Finally, we are given a *range parameter* $\Delta > 0$. We call the set of cell phones *fully connected* if it is possible to assign each phone to a base station in such a way that

    - Each phone is assigned to a different base station, and
    - If a phone at p_i is assigned to a base station at b_j, then the straight-line distance between the points p_i and b_j is at most Δ.

    Suppose that the owner of the cell phone at point p_1 decides to go for a drive, traveling continuously for a total of z units of distance due east. As this cell phone moves, we may have to update the assignment of phones to base stations (possibly several times) in order to keep the set of phones *fully connected*.

    Give a polynomial-time algorithm to decide whether it is possible to keep the set of phones fully connected at all times during the travel of this one cell phone. (You should assume that all other phones remain stationary during this travel.) If it is possible, you should report a sequence of assignments of phones to base stations that will be sufficient in order to maintain full connectivity; if it is not possible, you should report a point on the traveling phone's path at which full connectivity cannot be maintained.

    You should try to make your algorithm run in $O(n^3)$ time if possible.

    Example: Suppose we have phones at $p_1 = (0, 0)$ and $p_2 = (2, 1)$; we have base stations at $b_1 = (1, 1)$ and $b_2 = (3, 1)$; and $\Delta = 2$. Now consider the case in which the phone at p_1 moves due east a distance of 4 units, ending at $(4, 0)$. Then it is possible to keep the phones fully connected during this motion: We begin by assigning p_1 to b_1 and p_2 to b_2, and we re-assign p_1 to b_2 and p_2 to b_1 during the motion. (For example, when p_1 passes the point $(2, 0)$.)

    Solution. We first decide whether the phones can be fully connected in the starting position of p_1. To do this, we construct a bipartite graph $G = (X \cup Y, E)$, where X is the set of phones, Y is the set of base stations, and there is an edge (p_i, b_j) if and only

if the distance from p_i to b_j is at most Δ. By the definition of G, a perfect matching corresponds to a legal assignment of phones to base stations; thus, the phones can be fully connected if and only if G has a perfect matching, and this can be checked in $O(n^3)$ time.

Define an *event* to be a position t on the path of phone p_1 at which it first comes into range of a base station, or first goes out of range of a base station. The path of p_1 is a line, and the set of points in range of a base station b_j is a circle of radius Δ; since the line can intersect the circle at most twice, there is at most one event in which p_1 comes into range of b_j and at most one event in which it goes out of range of b_j. Thus, there are at most $2n$ events total; we can sort them by x-coordinate in the order $t_1, t_2, \ldots, t_k$, with $k \leq 2n$.

In the interval between consecutive events, the bipartite graph G does not change. At an event t_i, the bipartite graph changes because some edge incident to p_1 is either added or deleted. Let $G[t_i]$ denote the bipartite graph just after event t_i. Since these are the only points at which the graph changes, the phones can be fully connected if and only if each of $G, G[t_1], G[t_2], \ldots, G[t_k]$ has a perfect matching.

Thus, an $O(n^4)$ algorithm would be to test each of these graphs for a perfect matching. To bring the running time down to $O(n^3)$, we make the following observation. A perfect matching M in $G[t_i]$ becomes a matching of size either n or $n-1$ in $G[t_{i+1}]$, depending on whether one of the edges in M is the edge that is deleted during the event t_i. In the first case, M is already a perfect matching in $G[t_{i+1}]$, so we don't have to do any work at all for this event. In the second case, we simply to need to search for *a single augmenting path* in order to decide whether the existing matching of size $n-1$ can be increased to a perfect matching. This takes time $O(|E|) = O(n^2)$. Moreover, the sequence of assignments can be recorded by keeping track of the sequence of matching constructed over all the events.

Thus, the total running time of the improved algorithm is $O(n^3)$ (for the initial perfect matching in G) plus $O(kn^2)$ (for the (at most) one augmenting path at each event). Since $k \leq 2n$, the total running time is $O(n^3 + kn^2) = O(n^3)$.

18. Suppose you're managing a collection of processors and must schedule a sequence of jobs over time.

    The jobs have the following characteristics. Each job j has an arrival time a_j when it is first available for processing, a length ℓ_j which indicates how much processing time it needs, and a deadline d_j by which it must be finished. (We'll assume $0 < \ell_j \leq d_j - a_j$.) Each job can be run on any of the processors, but only on one at a time; it can also be pre-empted and resumed from where it left off (possibly after a delay) on another processor.

    Moreover, the collection of processors is not entirely static either: you have an overall pool of k possible processors; but for each processor i, there is an interval of time $[t_i, t_i']$ during which it is available; it is unavailable at all other times.

Given all this data about job requirements and processor availability, you'd like to decide whether the jobs can all be completed or not. Give a polynomial-time algorithm that either produces a schedule completing all jobs by their deeadlines, or reports (correctly) that no such schedule exists. You may assume that all the parameters associated with the problem are integers.

Example. Suppose we have two jobs J_1 and J_2. J_1 arrives at time 0, is due at time 4, and has length 3. J_2 arrives at time 1, is due at time 3, and has length 2. We also have two processors P_1 and P_2. P_1 is available between times 0 and 4; P_2 is available between times 2 and 3. In this case, there is a schedule that gets both jobs done:

- At time 0, we start job J_1 on processor P_1.
- At time 1, we pre-empt J_1 to start J_2 on P_1.
- At time 2, we resume J_1 on P_2. (J_2 continues processing on P_1.)
- At time 3, J_2 completes by its deadline. P_2 ceases to be available, so we move J_1 back to P_1 to finish its remaining one unit of processing there.
- At time 4, J_1 completes its processing on P_1.

Notice that there is no solution that does not involve pre-emption and moving of jobs.

Solution. Let a^* be the earliest arrival time of any job, and d^* the latest deadline of any job. We break up the interval $I = [a^*, d^*]$ at each value of any a_j, d_j, t_i, or t'_i. Let the resulting sub-intervals of I be denoted $I_1, I_2, \ldots, I_r$, with $I_i = [s_i, s'_i]$. Note that $s'_i = s_{i+1}$ in our notation; we let $q_i = s'_i - s_i$ denote the length of interval I_i in time steps. Observe that the set of processors available is constant throughout each interval I_i; let n_i denote the size of this set. Also, the set of jobs that have been released but are not yet due is constant throughout each interval I_i.

We now construct a flow network G that tells us, for each job j and each interval I_i, how much time should be spent processing job j during interval I_i. ¿From this, we can construct the schedule. We define a node u_j for each job j, and a node v_i for each interval I_i. If $I_i \subseteq [a_j, d_j]$, then we add an edge (u_j, v_i) of capacity q_i. We define an edge from the source s to each u_j with capacity ℓ_j, and define an edge from each v_i to the sink t with capacity $n_i q_i$.

Now, suppose there is a schedule that allows each job to complete by its deadline, and suppose it processes job j for z_{ji} units of time in the interval I_i. Then we define a flow with value ℓ_j on the edge (s, u_j), value z_{ji} on the edge (u_j, v_i), and sufficient flow on the edge (v_i, t) to satisfy conservation. Note that the capacity of (v_i, t) cannot be exceeded, since we have a valid schedule.

Conversely, given an integer flow of value $\sum_j \ell_j$ in G, we run job j for $z_{ji} = f(u_j, v_i)$ units of time during interval I_i. Since the flow has value $\sum_j \ell_j$, it clearly saturates each edge (s, u_j), and so u_j will be processed for ℓ_j units of time, as required, if we can guarantee that all jobs j can really be scheduled for z_{ji} units of time during interval I_i.

The issue here is the following: we are told that job j must receive z_{ji} units of processing time during I_i, and it can move from one processor to another during the interval, but we need to avoid having two processors working on the same job at the same point in time. Here is a way to assign jobs to processors that avoids this. Let $P_1, P_2, \ldots, P_{n_i}$ denote the processors, and let $y_j = \sum_{r<j} z_{ri}$. For each $k = y_j + 1, y_j + 2, \ldots, y_{j+1}$, we have processor $\lceil k/q_i \rceil$ spend the $(k - q_i \lfloor k/q_i \rfloor)^{\text{th}}$ step of interval I_i working on job j. Since $\sum_j z_{ji} \leq n_i q_i$, each job gets a sufficient number of steps allocated to it; and since $z_{ji} \leq q_i$ for each j, this allocation scheme does not involve two processors working on the same job at the same point in time.

19. In a lot of numerical computations, we can ask about the "stability" or "robustness" of the answer. This kind of question can be asked for combinatorial problems as well; here's one way of phrasing the question for the minimum spanning tree problem.

    Suppose you are given a graph $G = (V, E)$, with a cost c_e on each edge e. We view the costs as quantities that have been measured experimentally, subject to possible errors in measurement. Thus, the minimum spanning tree one computes for G may not in fact be the "real" minimum spanning tree.

    Given error parameters $\varepsilon > 0$ and $k > 0$, and a specific edge $e' = (u, v)$, you would like to be able to make a claim of the following form:

    > $(*)$ Even if the cost of *each* edge were to be changed by at most ε (either increased or decreased), and the costs of k of the edges *other than* e' were further changed to arbitrarily different values, the edge e' would still not belong to any minimum spanning tree of G.

    Such a property provides a type of guarantee that e' is not likely to be belong to the minimum spanning tree, even assuming significant measurement error.

    Give a polynomial-time algorithm that takes G, e', ε, and k, and decides whether or not property $(*)$ holds for e'.

    Solution. We test whether $e' = (u, v)$ could enter the MST as follows. We first lower the cost of e' by ε and raise the cost of all other edges by ε. We then form a graph G' by deleting all edges whose cost is greater than or equal to that of e'. Finally, we determine whether the minimum u-v cut in G has at most k edges.

    If the answer is "yes," then by raising the cost of these k edges to ∞, we see that e' is one of the cheapest edges crossing a u-v cut, and so it belongs to an MST. Conversely, suppose e' can be made to enter the MST. Then this remains true if we continue to lower the cost of e' by as much as possible, and if we continue to raise the costs of all other edges by as much as possible. Now, at this point, consider the set E' of k edges whose costs we need to alter arbitrarily. If we delete E' from the graph, then there cannot be a u-v path consisting entirely of edges lighter than e'; thus, there is no u-v path in the graph G' defined above, and so E' defines a u-v cut of value at most k.

20. Let $G = (V, E)$ be a directed graph, and suppose that for each node v, the number of edges into v is equal to the number of edges out of v. That is, for all v,

$$|\{(u,v) : (u,v) \in E\}| = |\{(v,w) : (v,w) \in E\}|.$$

Let x, y be two nodes of G, and suppose that there exist k mutually edge-disjoint paths from x to y. Under these conditions, does it follow that there exist k mutually edge-disjoint paths from y to x? Give a proof, or a counter-example with explanation.

Solution. If we put a capacity of 1 on each edge, then by the integrality theorem for maximum flows, there exist k edge-disjoint x-y paths if and only if there exists a flow of value k. By the max-flow min-cut theorem, this latter condition holds if and only if there is no x-y cut (A, B) of capacity less than k.

Now suppose there were a y-x cut (B', A') of capacity strictly less than k, and consider the x-y cut (A', B'). We claim that the capacity of (A', B') is equal to the capacity of (B', A'). For if we let $\delta^-(v)$ and $\delta^+(v)$ denote the number of edges into and out of a node v respectively, then we have

$$\begin{aligned} c(A', B') - c(B', A') &= |\{(u,v) : u \in A', v \in B'\}| - |\{(u,v) : u \in B', v \in A'\}| \\ &= |\{(u,v) : u \in A', v \in B'\}| + |\{(u,v) : u \in A', v \in A'\}| - \\ &\quad\ |\{(u,v) : u \in A', v \in A'\}| - |\{(u,v) : u \in B', v \in A'\}| \\ &= \sum_{v \in A} \delta^+(v) - \sum_{v \in A} \delta^-(v) \\ &= 0. \end{aligned}$$

It follows that $c(A', B') < k$ as well, contradicting our observation in the first paragraph. Thus, every y-x cut has capacity at least k, and so there exist k edge-disjoint y-x paths.

21. Given a graph $G = (V, E)$, and a natural number k, we can define a relation $\xrightarrow{G,k}$ on pairs of vertices of G as follows. If $x, y \in V$, we say that $x \xrightarrow{G,k} y$ if there exist k mutually edge-disjoint paths from x to y in G.

Is it true that for every G and every $k \geq 0$, the relation $\xrightarrow{G,k}$ is transitive? That is, is it always the case that if $x \xrightarrow{G,k} y$ and $y \xrightarrow{G,k} z$, then we have $x \xrightarrow{G,k} z$? Give a proof or a counter-example.

22. Give a polynomial time algorithm for the following minimization analogue of the max-flow problem. You are given a directed graph $G = (V, E)$, with a source $s \in V$ and sink $t \in V$, and numbers (capacities) $\ell(v, w)$ for each edge $(v, w) \in E$. We define a flow f, and the value of a flow, as usual, requiring that all nodes except s and t satisfy flow conservation. However, the given numbers are lower bounds on edge flow, i.e., they require that $f(v, w) \geq \ell(v, w)$ for every edge $(v, w) \in E$, and there is no upper bound on flow values on edges.

(a) Give a polynomial time algorithm that finds a feasible flow of minimum possible value.

(b) Prove an analog of the max-flow min-cut theorem for this problem (i.e., does min-flow = max-cut?)

23. We define the *escape problem* as follows. We are given a directed graph $G = (V, E)$ (picture a network of roads); a certain collection of nodes $X \subset V$ are designated as *populated nodes*, and a certain other collection $S \subset V$ are designated as *safe nodes*. (Assume that X and S are disjoint.) In case of an emergency, we want evacuation routes from the populated nodes to the safe nodes. A set of evacuation routes is defined as a set of paths in G so that (i) each node in X is the tail of one path, (ii) the last node on each path lies in S, and (iii) the paths do not share any edges. Such a set of paths gives a way for the occupants of the populated nodes to "escape" to S, without overly congesting any edge in G.

    (a) Given G, X, and S, show how to decide in polynomial time whether such a set of evacuation routes exists.

    (b) Suppose we have exactly the same problem as in (a), but we want to enforce an even stronger version of the "no congestion" condition (iii). Thus, we change (iii) to say "the paths do not share any *nodes*."

    With this new condition, show how to decide in polynomial time whether such a set of evacuation routes exists.

    Also, provide an example with a given G, X, and S, in which the answer is "yes" to the question in (a) but "no" to the question in (b).

24. You are helping the medical consulting firm *Doctors Without Weekends* set up a system for arranging the work schedules of doctors in a large hospital. For each of the next n days, the hospital has determined the number of doctors they want on hand; thus, on day i, they have a requirement that *exactly* p_i doctors be present.

    There are k doctors, and each is asked to provide a list of days on which he or she is willing to work. Thus, doctor j provides a set L_j of days on which he or she is willing to work.

    The system produced by the consulting firm should take these lists, and try to return to each doctor j a list L'_j with the following properties.

    (A) L'_j is a subset L_j, so that doctor j only works on days he or she finds acceptable.

    (B) If we consider the whole set of lists $L'_1, \ldots, L'_k$, it causes exactly p_i doctors to be present on day i, for $i = 1, 2, \ldots, n$.

    (a) Describe a polynomial-time algorithm that implements this system. Specifically, give a polynomial-time algorithm that takes the numbers $p_1, p_2, \ldots, p_n$, and the lists $L_1, \ldots, L_k$, and does one of the following two things.

- Return lists $L'_1, L'_2, \ldots, L'_k$ satisfying properties (A) and (B); or
- Report (correctly) that there is no set of lists $L'_1, L'_2, \ldots, L'_k$ that satisfies both properties (A) and (B).

You should prove your algorithm is correct, and briefly analyze the running time.

(b) The hospital finds that the doctors tend to submit lists that are much too restrictive, and so it often happens that the system reports (correctly, but unfortunately) that no acceptable set of lists $L'_1, L'_2, \ldots, L'_k$ exists.

Thus, the hospital relaxes the requirements as follows. They add a new parameter $c > 0$, and the system now should try to return to each doctor j a list L'_j with the following properties.

(A*) L'_j contains at most c days that do not appear on the list L_j.

(B) *(Same as before.)* If we consider the whole set of lists $L'_1, \ldots, L'_k$, it causes exactly p_i doctors to be present on day i, for $i = 1, 2, \ldots, n$.

Describe a polynomial-time algorithm that implements this revised system. It should take the numbers $p_1, p_2, \ldots, p_n$, the lists $L_1, \ldots, L_k$, and the parameter $c > 0$, and do one of the following two things.

- Return lists $L'_1, L'_2, \ldots, L'_k$ satisfying properties (A*) and (B); or
- Report (correctly) that there is no set of lists $L'_1, L'_2, \ldots, L'_k$ that satisfies both properties (A*) and (B).

In this question, you need only describe the algorithm; you do not need to explicitly write a proof of correctness or running time analysis. (However, your algorithm must be correct, and run in polynomial time.)

Solution. We construct a flow network as follows. There is a node u_j for each doctor j and a node v_i for each day i. There is an edge (u_j, v_i) of capacity 1 if doctor j can work on day i, and no edge otherwise. There is a source s, and an edge (s, u_j) of capacity $|L_j|$ for each j. There is a sink t, and an edge (v_i, t) of capacity p_i for each i.

Now we ask: is there an s-t flow of value $\sum_{i=1}^{n} p_i$ in this flow network? If there is, then there is an integer-valued flow, and we can produce a set of lists $\{L'_i\}$ from this as follows: assign doctor j to day i if there is a unit of flow on the edge (u_j, v_i). In this way, each doctor only works on days he or she finds acceptable, and each day i has p_i working on it. Conversely, if there is a valid set of lists for the doctors, then there will be a flow of value $\sum_{i=1}^{n} p_i$ in the network: we simply raise one unit of flow on each path s-u_j-v_i-t, where doctor j works on day i.

The total running time for this algorithm is dominated by the time for a flow computation on a graph with $O(kn)$ edges, where the total capacity out of the sink is $O(kn)$; thus, the total running time is $O(k^2n^2)$.

(b) We take the previous flow network and add some nodes to it, modeling the requirements. For each doctor j, we add a "spill-over" node u'_j. There is an edge (u'_j, v_i) of capacity 1 for each day i such that doctor j *doesn't* want to work on i. There is an edge (s, u'_j) of capacity c for each j.

Again we ask: is there an s-t flow of capacity $\sum_{i=1}^{n} p_i$ in this flow network? If there is, then there is an integer-valued flow, and we can produce a set of lists $\{L'_i\}$ from this as follows: assign doctor j to day i if there is a unit of flow on the edge (u_j, v_i) *or* if there is a unit of flow on the edge (u'_j, v_i).

25. You are consulting for an environmental statistics firm. They collect statistics, and publish the collected data in a book. The statistics is about populations of different regions in the world, and is in the millions. Examples of such statistics would look like the top table.

Country	A	B	C	Total
grownup men	11.998	9.083	2.919	24.000
grownup women	12.983	10,872	3.145	27.000
children	1.019	2.045	0.936	4.000
Total	26.000	22.000	7.000	55.000

We will assume here for simplicity that our data is such that all row and column sums are integers. The census rounding problem is to round all data to integers without changing any row or column sum. Each fractional number can be rounded either up or down without. For example a good rounding for the above data would be as follows.

Country	A	B	C	Total
grownup men	11.000	10.000	3.000	24.000
grownup women	13.000	10.000	4.000	27.000
children	2.000	2.000	0.000	4.000
Total	26.000	22.000	7.000	55.000

(a) Consider first the special case when all data is between 0 and 1. So you have a matrix of fractional numbers between 0 and 1, and your problem is to round each fraction that is between 0 and 1 to either 0 or 1 without changing the row or column sums. Use a flow computation to check if the desired rounding is possible.

(b) Consider the census data rounding problem as defined above, where row and column sums are integers, and you want round each fractional number α to either $\lfloor \alpha \rfloor$ or $\lceil \alpha \rceil$. Use a flow computation to check if the desired rounding is possible.

(c) Prove that the rounding we are looking for in (a) and (b) always exists.

26. (∗) Some friends of yours have grown tired of the game "Six degrees of Kevin Bacon" (after all, they ask, isn't it just breadth-first search?) and decide to invent a game with a little more punch, algorithmically speaking. Here's how it works.

    You start with a set X of n actresses and a set Y of n actors, and two players P_0 and P_1. P_0 names an actress $x_1 \in X$, P_1 names an actor y_1 who has appeared in a movie with x_1, 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 co-starred with the actress/actor immediately preceding. A player P_i $(i = 0, 1)$ loses when it is P_i's turn to move, and he/she cannot name a member of his/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). Give a polynomial-time algorithm that decides which of the two players can force a win, in a particular instance of this game.

    Solution. Build a flow network G with vertices s, v_i for each $x_i \in X$, w_j for each $y_j \in Y$, and t. There are edges (s, v_i) for all i, (w_j, t) for all j, and (v_i, w_j) iff x_i and y_j have appeared together in a movie. All edges are given capacity 1. Consider a maximum s-t flow f in G; by the integrality theorem, it consists of a set $\{R_1, \ldots, R_k\}$ of edge-disjoint s-t paths, where k is the value of f. (Note that this is simply the construction we used to reduce bipartite matching to maximum flow.)

    Suppose the value of f is n. Then each time player 1 names an actress x_i, player 2 can name the actor y_j so that (v_i, w_j) appear on a flow path together. In this way, player 1 must eventually run out of actresses and lose.

    On the other hand, suppose the value of f is less than n. Player 1 can thus start with an actress x_i so that v_i lies on no flow path. Now, at every point of the game, we claim the vertex for the actor y_j named by player 2 lies on some flow path R_t. For if not, consider the first time when w_j does not lie on a flow path; if we take the sequence of edges in G traversed by the two players thus far, add s to the beginning and t to the end, we obtain an augmenting s-t path, which contradicts the maximality of f. Hence, each time player 2 names an actor y_j, player 1 can name an actress x_ℓ so that (x_ℓ, y_j) appear on a flow path together. In this way, player 2 must eventually out of actors and lose.

27. Statistically, the arrival of spring typically results in increased accidents, and increased need for emergency medical treatment, which often requires blood transfusions. Consider the problem faced by a hospital that is trying to evaluate whether their blood supply is sufficient.

    The basic rule for blood donation is the following. A person's own blood supply has certain *antigens* present (we can think of antigens as a kind of molecular signature); and a person cannot receive blood with a particular antigen if their own blood does not

have this antigen present. Concretely, this principle underpins the division of blood into four *types*: A, B, AB, and O. Blood of type A has the A antigen, blood of type B has the B antigen, blood of type AB has both, and blood of type O has neither. Thus, patients with type A can receive only blood types A or O in a transfusion, patients with type B can receive only B or O, patients with type O can receive only O, and patients with type AB can receive any of the four types.[1]

(a) Let s_O, s_A, s_B and s_{AB} denote the supply in whole units of the different blood types on hand. Assume that the hospital knows the projected demand for each blood type d_O, d_A, d_B and d_{AB} for the coming week. Give a polynomial time algorithm to evaluate if the blood on hand would suffice for the projected need.

(b) Consider the following example. Over the next week, they expect to need at most 100 units of blood. The typical distribution of blood types in US patients is 45% type O, 42% type A, 10% type B, and 3% type AB. The hospital wants to know if the blood supply they have on hand would be enough if 100 patients arrive with the expected type distribution. There is a total of 105 units of blood on hand. The table below gives these demands, and the supply on hand.

blood type:	O	A	B	AB
supply:	50	36	11	8
demand:	45	42	10	3

Is the 105 units of blood on hand enough to satisfy the 100 units of demand? Find an allocation that satisfies the maximum possible number of patients. Use an argument based on a minimum capacity cut to show why not all patients can receive blood. Also, provide an explanation for this fact that would be understandable to the clinic administrators, who have not taken a course on algorithms. (So, for example, this explanation should not involve the words "flow," "cut," or "graph" in the sense we use them in CS 482.)

Solution. To solve this problem, construct a graph $G = (V, E)$ as follows: Let the set of vertices consist of a super source node, four supply nodes (one for each blood type) adjacent to the source, four demand nodes and a super sink node that is adjacent to the demand nodes. For each supply node u and demand node v, construct an edge (u, v) if type v can receive blood from type u and set the capactity to ∞ or the demand for type v. Construct an edge (s, u) between the source s and each supply node u with the capacity set to the available supply of type u. Similarly, for each each demand node v and the sink t, construct an edge (v, t) with the capacity set to the demand for type v.

Now run the F-F algorithm on this graph to find the max-flow. If the edges from the demand nodes to the sink are all saturated in the resulting max-flow, then there is

[1]The Austrian scientist Karl Landsteiner received the Nobel Prize in 1930 for his discovery of the blood types A, B, O, and AB.

sufficient supply for the projected need. Using the running time in Section 6.5, we get a time bound of $O(\log C)$, where C is the total supply. (Note that the graph itself has constant size.)

To see why this must be the case, consider a max-flow where at least one of the demand to sink edges is not saturated. Denote this edge by (v, t). Let S be the set of of blood types that can donate to a person with blood type v. Note that all edges of the form (s, u) with $u \in S$ must then be saturated. If this were not the case we could push more flow along (u, v) for some $u \in S$. Now also note that since the capacities on the demand-sink edges is the expected demand, the max-flow at best satisfies this demand. If the demand-sink edges are saturated, then clearly the demand can be satisfied.

(b) Consider a cut containing the source, and the supply and demand nodes for B and A. The capacity of this cut is $50+36+10+3 = 99$, and hence all 100 units of demand cannot be satisfied.

An explanation for the clinic administrators: There are 87 people with demand for blood types O and A; these can only be satisfied by donors with blood types O and A; and there are only 86 such donors.

Grader's Comments: Since part (b) asked for an argument based on a minimum cut, it was necessary to actually specify a cut.

Part (a) can also be solved by a greedy algorithm; basically, it works as follows: The O group can only receive blood from O donors; so if the O group is not satisfied, there is no solution. Otherwise, satisfy the A and B groups using the leftovers from the O group; if this is not possible, there is no solution. Finally, satisfy the AB group using any remaining leftovers. A short explanation of correctness (basically following the above reasoning) is necessary for this algorithm, as it was with the flow algorithm.

28. Suppose you and your friend Alanis live, together with $n-2$ other people, at a popular off-campus co-operative apartment, The Upson Collective. Over the next n nights, each of you is supposed to cook dinner for the co-op exactly once, so that someone cooks on each of the nights.

    Of course, everyone has scheduling conflicts with some of the nights (e.g. prelims, concerts, etc.) — so deciding who should cook on which night becomes a tricky task. For concreteness, let's label the people

    $$\{p_1, \ldots, p_n\},$$

    the nights

    $$\{d_1, \ldots, d_n\};$$

    and for person p_i, there's a set of nights $S_i \subset \{d_1, \ldots, d_n\}$ when they are *not* able to cook.

    A *feasible dinner schedule* is an assignment of each person in the co-op to a different night, so that each person cooks on exactly one night, there is someone cooking on each night, and if p_i cooks on night d_j, then $d_j \notin S_i$.

(a) Describe a bipartite graph G so that G has a perfect matching if and only if there is a feasible dinner schedule for the co-op.

(b) Anyway, your friend Alanis takes on the task of trying to construct a feasible dinner schedule. After great effort, she constructs what she claims is a feasible schedule, and heads off to class for the day.

Unfortunately, when you look at the schedule she created, you notice a big problem. $n-2$ of the people at the co-op are assigned to different nights on which they are available: no problem there. But for the other two people — p_i and p_j — and the other two days — d_k and d_ℓ — you discover that she has accidentally assigned both p_i and p_j to cook on night d_k, and assigned no one to cook on night d_ℓ.

You want to fix Alanis's mistake, but without having to re-compute everything from scratch. Show that it's possible, using her "almost correct" schedule, to decide in only $O(n^2)$ time whether there exists a feasible dinner schedule for the co-op. (If one exists, you should also output it.)

Solution. **(a)** Let $G=(V,E)$ be a bipartite graph with a node $p_i \in V$ representing each person and a node $d_j \in V$ representing each night. The edges consist of all pairs (p_i, d_j) for which $d_j \notin S_i$.

Now, a perfect matching in G is a pairing of people and nights so that each person is paired with exactly one night, no two people are paired with the same night, and each person is available on the night they are paired with. Thus, if G has a perfect matching, then it has a feasible dinner schedule. Conversely, if G has a feasible dinner schedule, consisting of a set of pairs $S=\{(p_i,d_j)\}$, then each $(p_i,d_j)\in S$ is an edge of G, no two of these pairs share an endpoint in G, and hence these edges define a perfect matching in G.

(b) An algorithm is as follows. First, construct the bipartite graph G from part (a): it takes $O(1)$ time to decide for each pair (p_i,d_j) whether it should be an edge in G, so the total time to construct G is $O(n^2)$. Now, let Q denote the set of edges constructed by Alanis. Delete the edge (p_j,d_k) from Q, obtaining a set of edges Q'. Q' has size $n-1$, and since no person or night appears more than once in Q', it is a matching.

We now try to find an augmenting path in G with respect to Q', in time $O(|E|)=O(n^2)$. If we find such an augmenting path P, then increasing Q' using P gives us a matching of size n, which is a perfect matching and hence by (a) corresponds to a feasible dinner schedule. If G has no augmenting path with respect to Q', then by a theorem from class, Q' must be a maximum matching in G. In particular, this means that G has no perfect matching, and hence by (a) there is no feasible dinner schedule.

29. We consider the *bipartite matching* problem on a bipartite graph $G=(V,E)$. As usual, we say that V is partitioned into sets X and Y, and each edge has one end in X and the other in Y.

    If M is a matching in G, we say that a node $y \in Y$ is *covered* by M if y is an end of one of the edges in M.

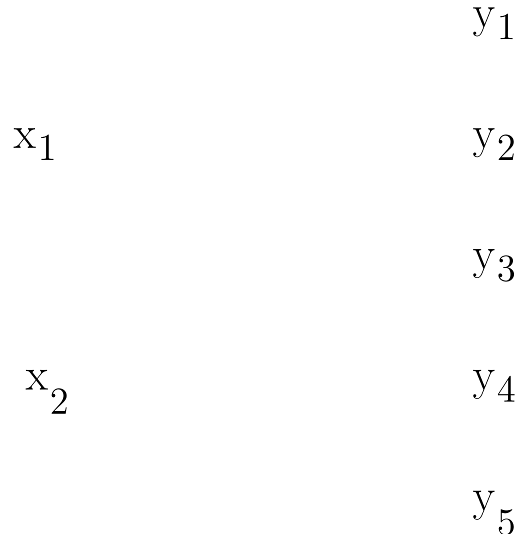


Figure 5: An instance of Coverage Expansion.

(a) Consider the following problem. We are given G and a matching M in G. For a given number k, we want to decide if there is a matching M' in G so that

(i) M' has k more edges than M does, *and*

(ii) every node $y \in Y$ that is covered by M is also covered by M'.

We call this the Coverage Expansion problem, with input G, M, and k. and we will say that M' is a *solution* to the instance.

Give a polynomial-time algorithm that takes an instance of Coverage Expansion and either returns a solution M' or reports (correctly) that there is no solution. (You should include an analysis of the running time, and a brief proof of why it is correct.)

Note: You may wish to also look at part (b) to help in thinking about this.

Example: Consider the accompanying figure, and suppose M is the matching consisting of the one edge (x_1, y_2). Suppose we are asked the above question with $k = 1$.

Then the answer to this instance of Coverage Expansion is "yes." We can let M' be the matching consisting (for example) of the two edges (x_1, y_2) and (x_2, y_4); M' it has 1 more edge than M, and y_2 is still covered by M'.

(b) Give an example of an instance of Coverage Expansion — specified by G, M, and k — so that the following situation happens.

> The instance has a solution; but in any solution M', the edges of M do not form a subset of the edges of M'.

(c) Let G be a bipartite graph, and let M be any matching in G. Consider the following two quantities.

- K_1 is the size of the largest matching M' so that every node y that is covered by M is also covered by M'.
- K_2 is the size of the largest matching M'' in G.

Clearly $K_1 \leq K_2$, since K_2 is obtained by considering *all possible* matchings in G.

Prove that in fact $K_1 = K_2$; that is, we can obtain a maximum matching even if we're constrained to cover all the nodes covered by our initial matching M.

Solution. **(a)** We set up a flow problem. We define a new graph G' for our flow problem by taking G and directing all edges from X to Y, and adding a super source s to the graph, and add edges (s, x) for all nodes $x \in X$ with capacity 1, and have each edge of G have capacity 1 also. We add a super sink t and add edges (y, s) for all nodes y that are **not covered** by the matching M. Now we define a flow problem where node s has a supply of $|M| + k$, node t has a demand of k, all nodes $y \in Y$ that are covered in M have a demand of 1. We claim that a flow satisfying these demands exists if an only if the matching M' exists.

If there is a flow in G' satisfying the demands, then by the integrality theorem there is an integer flow f'. The edges e in G that have $f'(e) = 1$ form the desired matching M'.

If there is a matching M', then we can define a flow f' in G' as follows. For edges $e = (x, y)$ we have $f'(e) = 1$ if e is in M' and 0 otherwise, we set $f'(e) = 1$ for edges $e = (s, x)$ if node x is covered by M', and finally, we set $f'(e) = 1$ for edges $e = (y, t)$ if node y is covered by M', but not covered by M. This satisfies all the demands.

It takes $O(m)$ time to build the flow network, and $O(mC) = O(mn)$ time to determine whether the desired flow exists.

(b) For example, take a 4 node graph with two nodes on each side, and edges $(x_1, y_1), (x_1, y_2)$ and (x_2, y_2). Let M have the single edge (x_1, y_2) and set $k = 1$. Then the solution exists, but only by deleting the edge (x_1, y_2), and reassigning y_2 to x_2.

(c) Consider the graph G' analogous to the one we used in part (a), but connecting **all** nodes $y \in Y$ to the new sink t via an edge of capacity 1. Instead of using demands, we view this as a standard maximum flow problem — this is just the construction from the notes. We will show that there is a maximum matching that covers all nodes in Y that were covered by M. This implies that $K_1 \geq K_2$, and hence they are equal.

The matching M gives rise to a flow of value $|M|$ in G. Let f denote this flow. We use the Ford-Fulkerson algorithm, but instead of starting from the all-zero flow, we

start from the integer flow f. This results in an integer maximum flow f'. The value of f' is K_2. We claim that the corresponding matching M' covers all nodes in Y that are covered by matching M. A matching corresponding to an integer flow f in G' covers exactly those nodes in Y for which $f(e) = 1$ for $s = (y, t)$. The the statement above follows from the observation that for any node $y \in Y$ and edge $e = (y, t)$ if $f(e) = 1$ and we obtain f' by the Ford-Fulkerson algorithm starting from the flow f, then $f'(e) = 1$ also. For a flow augmentation to decrease the value of the flow on an edge e, we would have to use the corresponding backwards edge in the augmenting path, but this backwards leaves t, and hence is not part of any simple s-t paths.

30. Suppose you're a consultant for the Ergonomic Architecture Commission, and they come to you with the following problem.

    They're really concerned about designing houses that are "user-friendly," and they've been having a lot of trouble with the set-up of light fixtures and switches in newly designed houses. Consider, for example, a one-floor house with n light fixtures and n locations for light switches mounted in the wall. You'd like to be able to wire up one switch to control each light fixture, in such a way that a person at the switch can *see* the light fixture being controlled.

    Sometimes this is possible, and sometimes it isn't. Consider the two simple floor plans for houses in the accompanying figure. There are three light fixtures (labeled a, b, c) and three switches (labeled 1, 2, 3). It is possible to wire switches to fixtures in the example on the left so that every switch has line-of-sight to the fixture, but this is not possible in the example on the right.

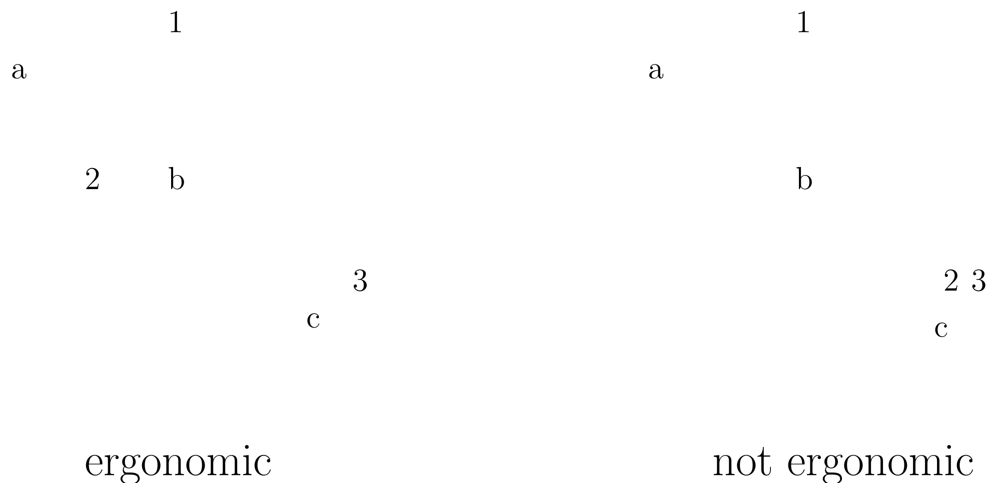


Figure 6: Two floor plans with lights and switches.

Let's call a floor plan — together with n light fixture locations and n switch locations — "ergonomic" if it's possible to wire one switch to each fixture so that every fixture

is visible from the switch that controls it. A floor plan will be represented by a set of m horizontal or vertical line segments in the plane (the walls), where the ith wall has endpoints $(x_i, y_i), (x_i', y_i')$. Each of the n switches and each of the n fixtures is given by its coordinates in the plane. A fixture is *visible* from a switch if the line segment joining them does not cross any of the walls.

Give an algorithm to decide if a given floor plan, in the above representation, is ergonomic. The running time should be polynomial in m and n. You may assume that you have a subroutine with $O(1)$ running time that takes two line segments as input and decides whether or not they cross in the plane.

Solution. We build the following bipartite graph $G = (V, E)$. V is partitioned into sets X and Y, with a node $x_i \in X$ representing switch i, and a node $y_j \in Y$ representing fixture j. (x_i, y_j) is an edge in E if and only if the line segment from x_i to y_j does not intersect any of the m walls in the floor plan. Thus, whether $(x_i, y_j) \in E$ can be determined initially by m segment-intersection tests; so G can be built in time $O(n^2 m)$.

Now, we test in $O(n^3)$ time whether G has a perfect matching, and declare the floor plan to be "ergonomic" if and only if G does have a perfect matching. Our answer is always correct, since a perfect matching in G is a pairing of the n switches and the n fixtures in such a way that each switch can see the fixture it is paired with, by the definition of the edge set E; conversely, such a pairing of switches and fixtures defines a perfect matching in G.

31. Some of your friends are interning at the small high-tech company WebExodus. A running joke among the employees there is that the back room has less space devoted to high-end servers than it does to empty boxes of computer equipment, piled up in case something needs to be shipped back to the supplier for maintainence.

    A few days ago, a large shipment of computer monitors arrived, each in its own large box; and since there are many different kinds of monitors in the shipment, the boxes do not all have the same dimensions. A bunch of people spent some time in the morning trying to figure out how to store all these things, realizing of course that less space would be taken up if some of the boxes could be *nested* inside others.

    Suppose each box i is a rectangular parallelepiped with side lengths equal to (i_1, i_2, i_3); and suppose each side length is strictly between half a meter and one meter. Geometrically, you know what it means for one box to nest inside another — it's possible if you can rotate the smaller so that it fits inside the larger in each dimension. Formally, we can say that box i with dimensions (i_1, i_2, i_3) *nests* inside box j with dimensions (j_1, j_2, j_3) if there is a permutation a, b, c of the dimensions $\{1, 2, 3\}$ so that $i_a < j_1$, and $i_b < j_2$, and $i_c < j_3$. Of course, nesting is recursive — if i nests in j, and j nests in k, then by putting i inside j inside k, only box k is visible. We say that a *nesting arrangement* for a set of n boxes is a sequence of operations in which a box i is put inside another box j in which it nests; and if there were already boxes nested inside i,

then these end up inside j as well. (Also notice the following: since the side lengths of i are more than half a meter each, and since the side lengths of j are less than a meter each, box i will take up more than half of each dimension of j, and so after i is put inside j, nothing else can be put inside j.) We say that a box k is *visible* in a nesting arrangement if the sequence of operations does not result in its ever being put inside another box.

So this is the problem faced by the people at WebExodus: Since only the visible boxes are taking up any space, how should a nesting arrangement be chosen so as to minimize the *number* of visible boxes?

Give a polynomial-time algorithm to solve this problem.

Example. Suppose there are three boxes with dimensions $(.6, .6, .6)$, $(.75, .75, .75)$, and $(.9, .7, .7)$. Then the first box can be put into either of the second or third boxes; but in any nesting arrangement, both the second and third boxes will be visible. So the minimum possible number of visible boxes is two, and one solution that achieves this is to nest the first box inside the second.

Solution. This is a complete parallel to the airline crew scheduling problem in the book. The boxes are like the flights, and where we previously encoded the idea that one crew can perform flight i followed by flight j, we analogously encode the idea that box i can be nested inside box j.

More precisely we reduce the given problem to a max flow problem where units of flow correspond to sets of boxes nested inside one visible box. We construct the following graph G:

- For each box i, G has two nodes u_i and v_i and an edge between them that corresponds to this box. This edge (u_i, v_i) has a lower bound of 1 and a capacity of 1. *(Each box is exactly in one set of boxes nested one in another.)*
- For each i and j so that box j nests inside box i, there is an edge (v_i, u_j). *(One can store box j inside i.)*
- G also has a source node s (corresponding to the back room where boxes are stored) and a sink node t (corresponding to nothing inside empty boxes).
- For each i, G has an edge (s, u_i) with a lower bound of 0 and a capacity of 1. *(Any box can be visible.)*
- For each j, G has an edge (v_j, t) with a lower bound of 0 and a capacity of 1. *(Any box can be empty.)*

We claim the following:

Fact 1 *There is a nesting arrangement with k visible boxes if and only of there is a feasible circulation in G with demand $-k$ in the source node s and demand k in the sink t.*

Proof. First, suppose there is a nesting arrangement with k visible boxes. Each sequence of nested boxes inside one visible box i_1, i_2, ..., i_n defines a path from s to t:

$$(s, u_{i_1}, v_{i_1}, u_{i_2}, v_{i_2}, \dots, u_{i_n}, v_{i_n}, t)$$

Therefore we have k paths from s to t. The circulation corresponding to all these paths satisfy all demands, capacity and lower bound.

Conversely, consider a feasible circulation in our network. Without lost of generality, assume that this circulation has integer flow values.

There are exactly k edges going to t that carries one unit of flow. Consider one of such edges (v_i, t). We know that (u_i, v_i) has one unit of flow. Therefore, there is a unique edge into u_i that carries one unit of flow. If this edge is of the kind (v_j, u_i) then put box i inside j and continue with box j. If this edge of the kind (s, u_i), then put the box i in the back room. This box became visible. Continuing in this way we pack all boxes into k visible ones.

So we can answer the question whether there is a nesting arrangement with exactly k visible boxes. Now to find the minimum possible number of visible boxes we answer this question for $k = 1, 2, 3$, and so on, until we find a positive answer. The maximum number of this iteration is n, therefore the algorithm is polynomial since we can find a feasible circulation in polynomial time.

32. **(a)** Suppose you are given a flow network with integer capacities, together with a maximum flow f that has been computed in the network. Now, the capacity of one of the edges e out of the source is raised by one unit. Show how to compute a maximum flow in the resulting network in time $O(m + n)$, where m is the number of edges and n is the number of nodes.

    (b) You're designing an interactive image segmentation tool that works as follows. You start with the image segmentation set-up described in Section 6.8, with n pixels, a set of neighboring pairs, and parameters $\{a_i\}$, $\{b_i\}$, and $\{p_{ij}\}$. We will make two assumptions about this instance. First, we will suppose that each of the parameters $\{a_i\}$, $\{b_i\}$, and $\{p_{ij}\}$ is a non-negative integer between 0 and d, for some number d. Second, we will suppose that the neighbor relation among the pixels has the property that each pixel is a neighbor of at most four other pixels (so in the resulting graph, there are at most four edges out of each node).

    You first perform an *initial segmentation* (A_0, B_0) so as to maximize the quantity $q(A_0, B_0)$. Now, this might result in certain pixels being assigned to the background, when the user knows that they ought to be in the foreground. So when presented with the segmentation, the user has the option of mouse-clicking on a particular pixel v_1, thereby bringing it to the foreground. But the tool should not simply bring this pixel into the foreground; rather, it should compute a segmentation (A_1, B_1) that maximizes the quantity $q(A_1, B_1)$ *subject to the condition that* v_1 *is in the foreground.* (In practice,

this is useful for the following kind of operation: in segmenting a photo of a group of people, perhaps someone is holding a bag that has been accidentally labeled as part of the background. By clicking on a single pixel belonging to the bag, and re-computing an optimal segmentation subject to the new condition, the whole bag will often become part of the foreground.)

In fact, the system should allow the user to perform a sequence of such mouse-clicks $v_1, v_2, \ldots, v_t$; and after mouse-click v_i, the system should produce a segmentation (A_i, B_i) that maximizes the quantity $q(A_i, B_i)$ subject to the condition that all of $v_1, v_2, \ldots, v_i$ are in the foreground.

Give an algorithm that performs these operations so that the initial segmentation is computed within a constant factor of the time for a single maximum flow, and then the interaction with the user is handled in $O(dn)$ time per mouse-click.

(Note: Part (a) is a useful primitive for doing this. Also, the symmetric operation of forcing a pixel to belong to the background can be handled by analogous means, but you do not have to work this out here.)

Solution. This part is easy. We just check whether it's possible to augment once in the residual graph.

Let G be an original network and G' be a network where the capacity of edge e is raised by 1. Let f be a maximum flow of G.

Fact 2a If there is a augment path in the residual graph G'_f then the maximum flow of G' is f augmented by this path. Otherwise, the maximum flow of G' still f.

Proof. It is clear that in a new network the value of any cut may be increased at most by 1, because we increase the capacity of only one edge by 1. Therefore any integer flow in G' with the value grater than the value of f is a maximum flow.

If there is a path in the residual graph G'_f, then there is an augment of f. The resulting flow has a greater value than f has, therefore it is a maximum flow.

On the other hand, if there is no path in the residual graph G'_f, then we know that f is maximum flow.

Using BFS or DFS we can check whether there is a $s-t$ path in G'_f in time $O(m+n)$.

(b) To make sure that a selected pixel v will be in the foreground we consider the same network as in textbook, but increase the capacity on the edge (s, v) to $5d+1$. Note that this is greater than the total capacity of all edges outgoing from v (at most four edges to neighbor pixels, plus the edge to t). So the minimum cut will now definitely has v on the source side (otherwise we could move v to source side and decrease the capacity of the cut).

Therefore the following algorithm works. Compute a maximum flow f and find a minimum cut, as in the book. Then, for each pixel v selected by the user, we want to raise the capacity on the edge (s, v) to $5d+1$. To do so we increase the capacity of this edge $5d + 1 - a$ times by 1 each time (where a is the old capacity). Each time we will use algorithm of part (a) to compute a new maximum flow in $O(m + n)$ time. Then we compute a minimum cut in $O(m)$ time.

The overall time per mouse-click is $(5d + 1)O(m + n) + O(m) = O(d(m + n))$. Since each non-source node in our graph has at most 5 outgoing edges and s has n outgoing edges, the total number of edges in the graph is $m \leq 5n + n = O(n)$. Therefore $O(d(m + m)) = O(dn)$.

33. We now consider a different variation on the image segmentation problem from Section 6.8. We will develop a solution to an *image labeling* problem, where the goal is to label each pixel with a rough estimate of its distance from the camera (rather than the simple *foreground/background* labeling used in the text). The possible labels for each pixel will be $0, 1, 2, \ldots, M$ for some integer M.

    Let $G = (V, E)$ denote the graph whose nodes are pixels, and edges indicate neighboring pairs of pixels. A *labeling* of the pixels is a partition of V into sets $A_0, A_1, \ldots, A_M$, where A_k is the set of pixels that is labeled with distance k for $k = 0, \ldots, M$. We will seek a labeling of minimum *cost*; the cost will come from two types of terms. By analogy with the foreground/background segmentation problem, we will have an *assignment cost*: for each pixel i and label k, the cost $a_{i,k}$ is the cost of assigning label k to pixel i. Next, if two neighboring pixels $(i, j) \in E$ are assigned different labels, there will be a *separation* cost. In the book, we use a separation penalty p_{ij}. In our current problem, the separation cost will also depend on how far the two pixels are separated; specifically, it will be proportional to the difference in value between their two labels.

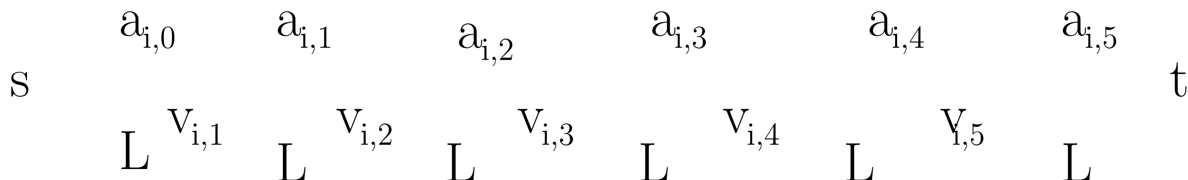
    Thus, the overall cost q' of a labeling is defined as follows:

    $$q'(A_0, \ldots, A_M) = \sum_{k=0}^{M} \sum_{i \in A_i} a_{i,k} + \sum_{k<\ell} \sum_{\substack{(i,j) \in E \\ i \in A_k, j \in A_\ell}} (\ell - k) p_{ij}.$$

    The goal of this problem is to develop a polynomial-time algorithm that finds the optimal labeling given the graph G and the penalty parameters $a_{i,k}$ and p_{ij}. The algorithm will be based on constructing a flow network, and we will start you off on designing the algorithm by providing a portion of the construction.

    The flow network will have a source s and a sink t. In addition, for each pixel $i \in V$ we will have nodes $v_{i,k}$ in the flow network for $k = 1, \ldots, M$ as shown in the accompanying figure. ($M = 5$ in the example in the figure.)

    For notational convenience, the nodes $v_{i,0}$ and $v_{i,M+1}$ will refer to s and t respectively, for any choice of $i \in V$.



We now add edges $(v_{i,k}, v_{i,k+1})$ with capacity $a_{i,k}$ for $k = 0, \ldots, M$; and edges $(v_{i,k+1}, v_{i,k})$ in the opposite direction with very large capacity L. We will refer to this collection of nodes and edges as the *chain* associated with pixel i.

Notice that if we make this very large capacity L large enough, then there will be no minimum cut (A, B) so that an edge of capacity L leaves the set A. (How large do we have to make it for this to happen?). Hence, for any minimum cut (A, B), and each pixel i, there will be exactly one low-capacity edge in the chain associated with i that leaves the set A. (You should check that if there were two such edges, then a large-capacity edge would also have to leave the set A.)

Finally, here's the question: Use the nodes and edges defined so far to complete the construction of a flow network with the property that a minimum-cost labeling can be efficiently computed from a minimum (s, t)-cut. You should prove that your construction has the desired property, and show to recover the minimum-cost labeling from the cut.

Solution. First, consider a cut (A_0, B_0), where A_0 is just $\{s\}$. Note, that there no edges with capacity L going out s, therefore the capacity of the cut (A_0, B_0) does not depend on L. Let L be greater than this capacity. Then any cut cutting an edge of capacity L has capacity at least L. So such cut can not be a minimum cut, because it has larger capacity than (A_0, B_0) cut.

Now consider a minimum cut (A, B). Let i be an arbitrary pixel. If $v_{i,k} \in A$ for some k then $v_{i,k-1}$ is also in A, since there is an edge $(v_{i,k}, v_{i,k-1})$ with capacity L and no edge with capacity L is out of A. Let $f(i)$ is the maximum number such that $v_{i,f(i)} \in A$ (such number exists because $v_{i,0} = s \in A$). Then A contains all nodes $v_{i,k}$ where $k \leq f(i)$, and A contains all nodes $v_{i,k}$ where $k > f(i)$.

This number $f(i)$ will represent the label of the pixel i (i.e. we put i in A_k when $f(i) = k$). So any labeling of the pixels corresponds to a cut (A, B) where $A = \{(v_{i,k} \mid k \leq \text{label of } i\}$. We have just proved that any *minimum* cut corresponds to some labeling. Up to now the capacity of such cut is

$$\sum_{k=0}^{M} a_{i,f(i)} = \sum_{k=0}^{M} \sum_{i:f(i)=k} a_{i,k}$$

which is equal to the first term of our cost function.

So the main issue remaining is to encode the separation costs. If pixels i and j are neighbors, we join $v_{i,k}$ and $v_{j,k}$, for each k, by two edges in both directions of capacity

$p_{i,j}$. Consequently, (A, B) cuts the edges of the kind $(v_{i,k}, v_{j,k})$ when $v_{i,k} \in A$ and $v_{j,k} \in B$, i.e., when $k \leq f(i)$ and $k > f(j)$. For any pair of neighbors i and j if $f(j) \geq f(i)$ then there are exactly $f(j) - f(i)$ such k that $f(i) \leq k < f(j)$. Therefore there are $f(j) - f(i)$ edges out of A. The overall capacity of these edges is $(f(j) - f(i))p_{i,j}$ which is exactly the desired separation cost.

So we have proved that the cost of any labeling is equal to capacity of the corresponding cut. Hence, the minimum cost labeling corresponds to the minimum cut.

34. The goal of this problem is to suggest variants of the preflow-push algorithm that speed up the practical running time without ruining its worst case complexity. Recall that the algorithm maintains the invariant that $h(v) \leq h(w) + 1$ for all edges (v, w) in the residual graph of the current preflow. We proved that if f is a flow (not just a preflow) with this invariant, then it is a maximum flow. Heights were monotone increasing and the whole running time analysis depended on bounding the number of times nodes can increase their heights. Practical experience shows that the algorithm is almost always much faster than suggested by the worst case, and that the practical bottleneck of the algorithm is relabeling nodes (and not the unsaturating pushes that lead to the worst case in the theoretical analysis). The goal of the problems below is the decrease the number of relabelings by increasing heights faster than one-by-one. Assume you have a graph G with n nodes, m edges, capacities c, source s and sink t.

    (a) The preflow-push algorithm, as described in the text, starts by setting the flow equal to the capacity c_e on all edges e leaving the source, setting the flow to 0 on all other edges, setting $h(s) = n$, and setting $h(v) = 0$ for all other nodes $v \in V$. Give an $O(m)$ procedure that for initializing node heights that is better than what we had in class. Your method should set the height of each node v be as high as possible given the initial flow.

    (b) In this part we will add a new step, called *gap relabeling* to preflow-push, that will increase the labels of lots of nodes by more than one at a time. Consider a preflow f and heights h satisfying the invariant. A *gap* in the heights is an integer $0 < h < n$ so that no node has height exactly h. Assume h is a gap value, and let A be the set of nodes v with heights $n > h(v) > h$. *Gap relabeling* is to change the height of all nodes in A to n. Prove that the preflow/push algorithm with Gap relabeling is a valid max-flow algorithm. Note that the only new thing that you need to prove is that gap relabeling preserves the invariant above.

    (c) In Section **??** we proved that $h(v) \leq 2n - 1$ throughout the algorithm. Here we will have a variant that has $h(v) \leq n$ throughout. The idea is that we "freeze" all nodes when they get to height n, i.e., nodes at hight n are no longer considered active, and hence are not used for push and relabel. This way at the end of the algorithm we have a preflow and height function that satisfies the invariant above, and so that all excess is at height n. Let B be the set of nodes v so that there is a path from v to t

in the residual graph of the current preflow. Let $A = V - B$. Prove that at the end of the algorithm (A, B) is a minimum capacity $s - t$ cut.

(d) The algorithm in part (c) computes a minimum $s - t$ cut, but fails to find a maximum flow (as it ends with a preflow that has excesses). Give an algorithm that takes the preflow f at the end of the algorithm of part (c) and converts it into a max flow in at most $O(mn)$ time. Hint: consider nodes with excess and try to send the excess back to s using only edges that the flow came on.