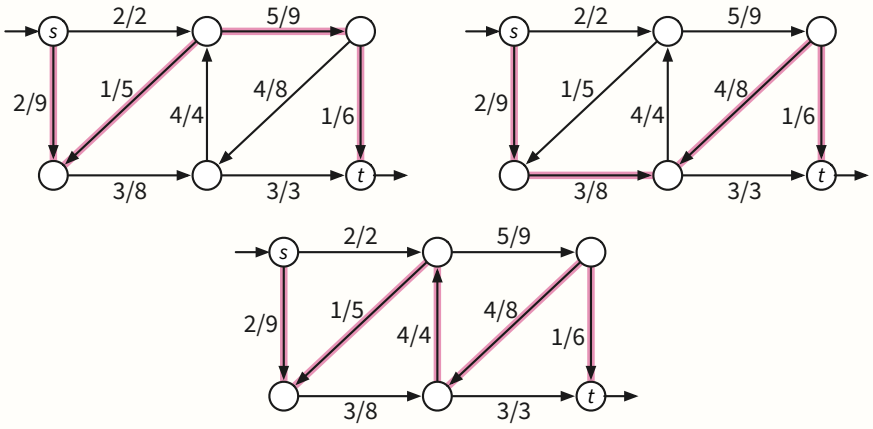
1. The figure below shows a flow network G, along with an (s, t)-flow f that is *not* a maximum flow. ***Clearly*** indicate the following structures in G

   (a) An augmenting path for f.

Solution: There are three augmenting paths, each using at least one backward residual edge:

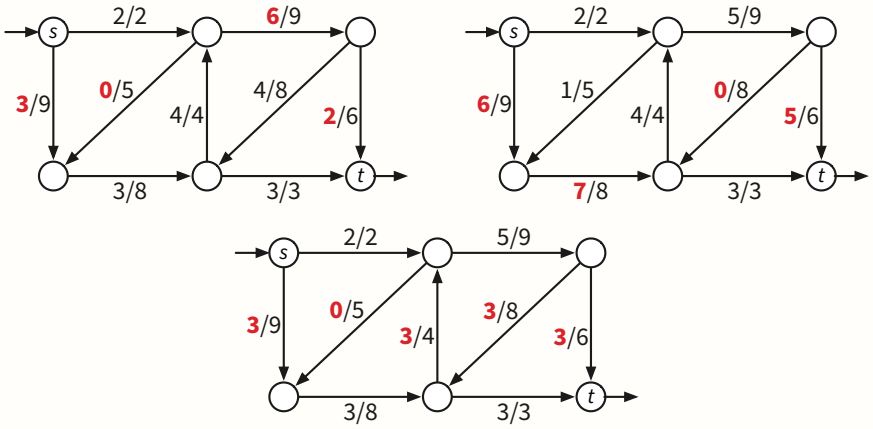


■

Rubric: 2½ points. Any one of these paths is worth full credit.

   (b) The result of augmenting f along that path.

Solution: Each augmenting path yields a different augmented flow:

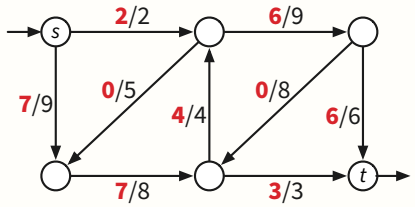


■

Rubric: 2½ points. The flow must be obtained by pushing as much flow as possible *along the path indicated in part (a)*. No credit here if part (a) is incorrect.

(c) A maximum (s, t)-flow in G.

Solution: The maximum flow is unique; the value of the maximum flow is 9.

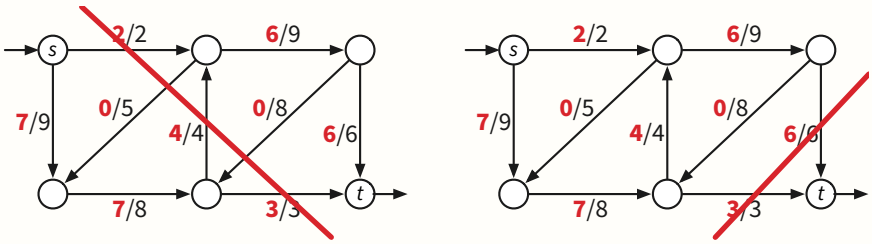


■

Rubric: 2½ points. 1 point for a feasible flow that is not a maximum flow.

(d) A minimum (s, t)-cut in G.

Solution: There are two different minimum cuts, each with capacity 9.



■

Rubric: 2½ points. Any one of these cuts is worth full credit. −1 for only marking forward-crossing edges if there are also backward-crossing edges. 1 point for an (s, t)-cut (that is, a partition of the vertices with s and t in different parts) that is not a minimum cut.

2. A sequence of numbers $x_1, x_2, \ldots, x_\ell$ is ***sort-of-increasing*** if each element (except the first two) is larger than the *average* of the two previous elements; that is, for every index $i > 2$, we have $2x_i > x_{i-1} + x_{i-2}$. Describe an efficient algorithm to compute the length of the longest sort-of-increasing subsequence of a given array $A[1..n]$ of numbers.

Solution (dynamic programming): For any indices $1 \le i < j \le n$, let $SortOf(i, j)$ denote the length of the longest sort-of-increasing subsequence of $A[i..n]$ whose first two elements are $A[i]$ and $A[j]$. We need to compute $\max_{i,j} SortOf(i, j)$.

This function satisfies the following recurrence:

$$SortOf(i,j) = \max\left\{2,\ 1 + \max\left\{ SortOf(j,k) \;\middle|\; \begin{array}{c} j < k \le n \text{ and} \\ 2 \cdot A[k] > A[i] + A[j] \end{array} \right\}\right\}$$

(Here we can define either $\max\varnothing = 0$ or $\max\varnothing = -\infty$; the recurrence is correct either way.)

We can memoize this function into a two-dimensional array $SortOf[1..n, 1..n]$, which we can fill with two nested loops, decreasing i in one and decreasing j in the other. (The nesting order of the loops doesn't matter.) For each i and j, we need $O(n)$ time to compute $SortOf[i, j]$, so the entire algorithm runs in $\boldsymbol{O(n^3)}$ ***time***. ■

Solution (dynamic programming): For any triple of indices $1 \le i < j < k \le n$, let $SortOf(i, j, k)$ denote the length of the longest sort-of-increasing subsequence of A whose first element is $A[i]$, whose second element is $A[j]$, and whose remaining elements all come from the suffix $A[k..n]$. If we add two sentinel elements $A[-1] = +\infty$ and $A[0] = -\infty$, then we need to compute $SortOf(-1, 0, 1) - 2$.

This function satisfies the following recurrence:

$$SortOf(i,j,k) = \begin{cases} 2 & \text{if } k > n \\ SortOf(i,j,k+1) & \text{if } 2 \cdot A[k] \le A[i] + A[j] \\ \max\left\{ \begin{array}{c} 1 + SortOf(j,k,k+1) \\ SortOf(i,j,k+1) \end{array} \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a three-dimensional array $SortOf[-1..n, 0..n, 1..n]$, indexed by i, j, and k. We can fill with three nested loops, decreasing k in the outermost loop and decreasing i and j in the other two. (The nesting order of the inner loops doesn't matter.) The entire algorithm runs in $\boldsymbol{O(n^3)}$ ***time***. ■

Solution (reduction to longest path in a dag): Define a directed acyclic graph $G = (V, E)$ as follows:

- $V = \{(i, j) \mid 1 \le i < j \le n\}$
- $E = \{(i, j) \to (j, k) \mid 2 \cdot A[k] > A[i] + A[j]\}$

Altogether G has $O(n^2)$ vertices and $O(n^3)$ edges. This graph is acyclic, because for every edge $(i, j) \to (j, k)$, we have $i < j$ and $j < k$.

Every directed path of length ℓ in G corresponds to a sort-of-increasing subsequence of A with length $\ell + 2$. Thus, we need to compute the longest path in G (with no fixed start or end vertex). We can compute this path in $O(V + E) = \mathbf{O(n^3)}$ ***time*** using the dag-longest-path algorithm in the textbook. Finally we return the number of edges in the longest path plus 2. ■

Rubric: 10 points, standard dynamic programming or graph reduction rubric, as appropriate. These are not the only solutions. This problem can actually be solved in $O(n^2 \log n)$ time.

3. Suppose you are given a chessboard with certain squares removed, represented as a two-dimensional boolean array $Legal[1..n, 1..n]$. A ***rook*** is a chess piece that attacks every square in the same row or column; that is, a rook on square (i, j) attacks every square of the form (i, k) or (k, j). Describe an algorithm to places as many rooks on the board as possible, each on a legal square, so that no two rooks attack each other.

Solution: Construct a bipartite graph $G = (R \sqcup C, E)$ as follows:

- R contains one vertex r_i for each row i,
- C contains one vertex c_j for each column j,
- E contains the edge $r_i c_j$ for each legal square (i, j).

Compute a maximum matching M in G in $O(VE) = \boldsymbol{O(n^3)}$ ***time***, using the algorithm described in class. Finally, return the number of edges in M. ■

Rubric: 10 points, standard reduction rubric. This is not the only solution.

4. Suppose we insert a sequence of n items into an initially empty hash table of size $2n$, using an *ideal random* open-address hash function.

   (a) ***Prove*** that for all $1 \le k \le n$ and for all $m \ge 0$, the kth insertion requires more than m probes with probability at most 2^{-m}.

   Solution: During the kth insertion, the table contains $k-1 < n$ items; in other words, more than half of the table entries are empty. Thus, a random probe during the kth insertion finds an occupied cell with probability less than $1/2$. Because the probe addresses are independent, the probability that the first m probes finds only occupied cells is less than $(1/2)^m$. ■

   Rubric: 3 points

   (b) ***Prove*** that for all $1 \le k \le n$, the kth insertion requires more than $2\log_2 n$ probes with probability at most $1/n^2$. *[Hint: Use part (a).]*

   Solution: Let $\#probes(k)$ denote the number of probes required by the kth insertion. If we set $m = 2\log_2 n$, we immediately have

$$\begin{aligned}\Pr[\#probes(k) > 2\log_2 n] &= \Pr[\#probes(k) > m] \\ &\le 2^{-m} && \text{by part (a)} \\ &= 2^{-2\log_2 n} = \frac{1}{n^2}\end{aligned}$$

   ■

   Rubric: 2 points

   (c) ***Prove*** that the maximum number of probes over all n insertions is more than $2\log_2 n$ with probability at most $1/n$.

   Solution:

$$\begin{aligned}&\Pr[\max_k \#probes(k) > 2\log_2 n] \\ &\quad= \Pr\left[\bigwedge_{k=1}^{n} \#probes(k) > 2\log_2 n\right] \\ &\quad\le \sum_{k=1}^{n} \Pr[\#probes(k) > 2\log_2 n] && \text{by the union bound} \\ &\quad\le \sum_{k=1}^{n} \frac{1}{n^2} = \frac{1}{n} && \text{by part (b)}\end{aligned}$$

   ■

   Rubric: 2 points

(d) What is the *exact* expected *total* number of probes for all n insertions?

Solution: During the kth phase, exactly $k-1$ of the $2n$ table cells are occupied, so the probability of each probe hitting an empty cell is $(2n-k+1)/2n$. It follows that the expected number of probes for the kth insertion is exactly $2n/(2n-k+1)$. Linearity of expectation now implies

$$\begin{aligned}
\mathrm{E}\left[\sum_{k=1}^{n} \#probes(k)\right] &= \sum_{k=1}^{n} \mathrm{E}[\#probes(k)] \\
&= \sum_{k=1}^{n} \frac{2n}{2n-k+1} \\
&= 2n \cdot \sum_{k=1}^{n} \frac{1}{2n-k+1} \\
&= 2n \cdot \sum_{\ell=n+1}^{2n} \frac{1}{\ell} && [\ell = 2n-k+1] \\
&= \boxed{\boldsymbol{2n \cdot (H_{2n} - H_n)}} \\
&\approx 2n \cdot (\ln(2n) - \ln(n)) \\
&= (2\ln 2)n \;\approx\; 1.38629n \;=\; \Theta(n)
\end{aligned}$$

If we only need a tight $O()$ bound, we can approximate as follows:

$$\begin{aligned}
\mathrm{E}\left[\sum_{k=1}^{n} \#probes(k)\right] &= \sum_{k=1}^{n} \frac{2n}{2n-k+1} \\
&\le \sum_{k=1}^{n} 2 \;=\; 2n \;=\; O(n)
\end{aligned}$$

■

Rubric: 3 points. A correct summation is worth 2 points. "$O(n)$" is worth 2½ points. No proof is required for full credit.

5. Suppose you are given a directed graph $G = (V, E)$ with positive integer edge capacities $c : E \to \mathbb{Z}^+$ and an integer maximum flow $f^* : E \to \mathbb{Z}$ from some vertex s to some other vertex t in G. Describe and analyze efficient algorithms for the following operations:

   (a) INCREMENT(e): Increase $c(e)$ by 1 and update the maximum flow f^*.

Solution: To INCREMENT the edge e, we first increase the capacity $c(e)$ by 1, and then perform one iteration of the Ford-Fulkerson augmenting path algorithm: Build the residual graph G_{f^*}, look for a path from s to t in G_{f^*}, and if such a path is found, push 1 unit of flow along it. The algorithm runs in ***O(E) time***. ■

We can prove the algorithm correct as follows. Let (S, T) be any minimum (s, t)-cut in G, with respect to the original capacities. Increasing $c(e)$ by 1 also increases the capacity of (S, T) by at most 1. The easy half of the maxflow-mincut theorem implies that new maximum flow value is *at most* the new capacity of (S, T). Thus, incrementing $c(e)$ increases the value of the maximum flow in G by at most 1. Because all residual capacities are integers, it follows that a single iteration of Ford-Fulkerson in the residual graph finds the new maximum flow.

Rubric: 5 points = 4 for algorithm + 1 for time analysis. Proof of correctness is not required. This is neither the only correct algorithm nor the only proof of correctness for this algorithm.

   (b) DECREMENT(e): Decrease $c(e)$ by 1 and update the maximum flow f^*.

Solution: Suppose we are asked to DECREMENT the edge $u \to v$. We must have $c(u \to v) > 0$, since otherwise, decreasing $c(u \to v)$ is impossible. Similarly, we can assume $f^*(u \to v) = c(u \to v)$, since otherwise, we can decrease $c(u \to v)$ by 1 and return immediately.

First, we send one unit of flow backward through $u \to v$, either along a path from t to s or along a cycle in in the residual graph G_{f^*}. Temporarily add an edge $s \to t$ to the residual graph G_{f^*}, find a path from u to v in this new larger graph, push one unit of flow along the corresponding edges of G, and finally decrease $f(u \to v)$ by 1.

Now we decrease $c(u \to v)$ by 1. The current flow f is still *feasible*, but it may not be a *maximum* flow. To restore a maximum flow, we run one iteration of Ford-Fulkerson, to push at most one more unit of flow from s to t.

The algorithm runs in ***O(E) time***. ■

To show that this algorithm is correct, we need two prove two claims:

- ***There is a path from u to v in the residual graph plus $s \to t$.*** Temporarily add an edge $t \to s$ to G with $f^*(t \to s) = |f^*|$. Now follow one unit of flow out of v through an arbitrary sequence of directed edges in $G + t \to s$, traversing each edge e at most $f^*(e)$ times. Because flow is conserved at every vertex (including s and t), this walk must eventually reach u. Reversing this walk gives us a walk in the residual graph from u to v. Any walk from u to v contains a path from u to v. (In fact, we can just use the walk directly.)

- ***Decrementing $c(u \to v)$ decreases the maximum flow value by at most 1.*** We can follow the proof from part (a). If $u \to v$ crosses some minimum cut, then decrementing $c(u \to v)$ decreases the capacity of the minimum cut by 1, and therefore decreases the value of the maximum flow by 1. Otherwise, the minimum cut capacity does not change, so the maximum flow value also does not change.

Solution: Suppose we are asked to DECREMENT the edge $u \to v$. We must have $c(u \to v) > 0$, since otherwise, decreasing $c(u \to v)$ is impossible. Similarly, we can assume $f^*(u \to v) = c(u \to v)$, since otherwise, we can decrease $c(u \to v)$ by 1 and return immediately.

First we attempt to reroute one unit of flow through $u \to v$ along another path from u to v. We search for a path from u to v in the residual graph G_{f^*} using whatever-first search in $O(E)$ time. If there is such a path P, we decrease $f(u \to v)$ by 1, push one unit of flow along P, and return the resulting flow. Because the new flow has the same value as f^*, it is also a maximum flow.

If there is no path from u to v in G_{f^*}, decreasing $c(u \to v)$ must reduce the value of the maximum flow. We temporarily add an edge $t \to s$, find a path P from v to u in the augmented residual graph $G_{f^*} + t \to s$ using whatever-first search, decrease $f(u \to v)$ by 1, push one unit of flow along P, remove the edge $t \to s$, and return the resulting flow.

The entire algorithm runs in $\boldsymbol{O(E)}$ ***time***. ■

We can prove that there is a path from u to v in the augmented residual graph as follows. Because f^* saturates $u \to v$, any flow decomposition of f^* contains a path Q from s to t through $u \to v$; the reversal of Q must be a directed path in the residual graph. So $rev(Q) + t \to s$ must be a cycle in the augmented residual graph $G_{f^*} + t \to s$. So $rev(Q) + t \to s - v \to u$ is a path from u to v in the augmented residual graph.

Solution: Suppose we are asked to DECREMENT the edge $x \to y$. We must have $c(x \to y) > 0$, since otherwise, decreasing $c(x \to y)$ is impossible. Similarly, we can assume $f^*(x \to y) = c(x \to y)$, since otherwise, we can decrease $c(x \to y)$ by 1 and return immediately.

First we attempt to reroute one unit of flow through $x \to y$ through another path from x to y. We search for a path from x to y in the residual graph G_{f^*} using whatever-first search in $O(E)$ time. If there is such a path P, we decrease $f(x \to y)$ by 1, push one unit of flow along P, and return the resulting flow. Because the new flow has the same value as f^*, it is also a maximum flow.

If there is no path from x to y in G_{f^*}, decreasing $c(x \to y)$ must reduce the value of the maximum flow. We find a path P from t to y and a path Q from x to s in G_{f^*}, using whatever first search. Finally, we then push one unit of flow along the residual path $P \cdot (y \to x) \cdot Q$.

The entire algorithm runs in $O(E)$ *time*. ■

To prove that this algorithm is correct, we need to show that P and Q are edge-disjoint. (Otherwise, we would push two units of flow through a common edge, but the capacity of that edge might be 1.) Let X be the subset of vertices that x can reach in G_{f^*}, and let Y be the subset of vertices that can reach y in the residual graph G_{f^*}. The subsets X and Y must be disjoint. Every path from t to y visits vertices only in Y, and every path from x to s visits vertices only in X. We conclude that paths P and Q have no vertices in common, and therefore no edges in common.

Rubric: 5 points = 4 for algorithm + 1 for time analysis. Proof of correctness is not required. These are neither the only correct algorithms nor the only proofs of correctness for these algorithms.

6. Suppose you are given an $n \times n$ array of 0s and 1s. Each row of the array is colored either red or blue, there are exactly k 1s in each row, and each column j of the matrix has a non-negative weight w_j. Your goal is to choose a subset of the columns that satisfy the following conditions:

   - In each red row, there are *at least one* 1s in the chosen columns.
   - In each blue row, there is *at least two* 1s in the chosen columns.
   - The sum of the weights of the chosen columns is as small as possible.

   (a) Write an integer linear program that ***exactly*** captures this problem. In particular, each solution of the integer linear program must describe a set of columns, and each set of columns must correspond to a solution of your integer linear program.

Solution: Let $A[1..n, 1..n]$ be the given array of 0s and 1s. For each column j, we have a variable x_i that equals 1 if column j is selected and 0 otherwise.

$$\begin{aligned}
\text{minimize} \quad & \sum_j w_j \cdot x_j \\
\text{subject to} \quad & \sum_j A[i,j] \cdot x_j \geq 1 && \text{for each red row } i \\
& \sum_j A[i,j] \cdot x_j \geq 2 && \text{for each blue row } i \\
& x_j \in \{0,1\} && \text{for each column } j
\end{aligned}$$

■

Rubric: 5 points = 1½ for objective + 2 for red/blue constraints + 1½ for indicator constraints

(b) Describe and analyze an efficient $(k/2)$-approximation algorithm for this problem. Remember to ***prove*** that your algorithm returns a valid solution, and ***prove*** that it achieves an approximation ratio of $k/2$.

Solution: This subproblem was broken; the correct approximation ratio is actually $\boldsymbol{k}$, not $k/2$. (Even for the special case where every row is blue, the correct approximation ratio is actually $k-1$.) **Everyone gets full credit for this subproblem.** Here is the solution for the correct approximation ratio:

Relax the ILP from part (a) to a linear program by replacing the last constraints with $0 \le x_j \le 1$. Let x^* denote the optimal fractional solution to this LP, and let $OPT^* = \sum_j w_j x_j^*$. We immediately have $OPT^* \le OPT$, where OPT is the value of the optimal *integer* solution.

We define a new integer vector x' as follows: For each index j, let

$$x'_j = \begin{cases} 1 & \text{if } x_j^* \ge \mathbf{1/k} \\ 0 & \text{otherwise} \end{cases}$$

Correctness: I claim that x' is a feasible solution for our integer linear program.

- For each red row i, we have $\sum_j A[i,j] \cdot x_j^* \ge 1$, there must be at least one column j such that $A[i,j] = 1$ and $x_j^* \ge 1/k$, and therefore $x'_j \ge 1$. It follows that $\sum_j A[i,j] \cdot x'_j \ge 1$ for each red row i.
- For each blue row i, we have $\sum_j A[i,j] \cdot x_j^* \ge 2$. So there must be at least two columns j and j' such that

  $$A[i,j] = 1 \quad \text{and} \quad x_j^* \ge 1/k \quad \text{and} \quad A[i,j'] = 1 \quad \text{and} \quad x_{j'}^* \ge 1/k.$$

  (Otherwise, we would have $\sum_j A[i,j] \cdot x_j^* < 1 + (k-1)/k < 2$.) Thus, $x'_j = x'_{j'} = 1$. It follows that $\sum_j A[i,j] \cdot x'_j \ge 2$ for each blue row i.

Approximation ratio: For each index j, we have $x'_j \le k \cdot x_j^*$. It follows that

$$\sum_j w_j x'_j \;\le\; k \cdot \sum_j w_j x_j^* \;=\; k \cdot OPT^* \;\le\; k \cdot OPT.$$

So x' is a k-approximation of the true optimum solution. ■

Rubric: 5 points. Everyone gets full credit.