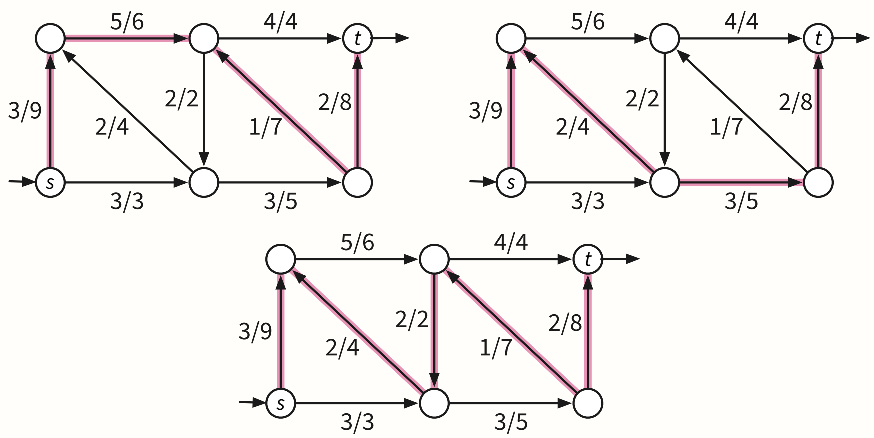
1. The figure below shows a flow network G, along with an (s,t)-flow f that is *not* a maximum flow. ***Clearly*** indicate the following structures in G:

   (a) An augmenting path for f.

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

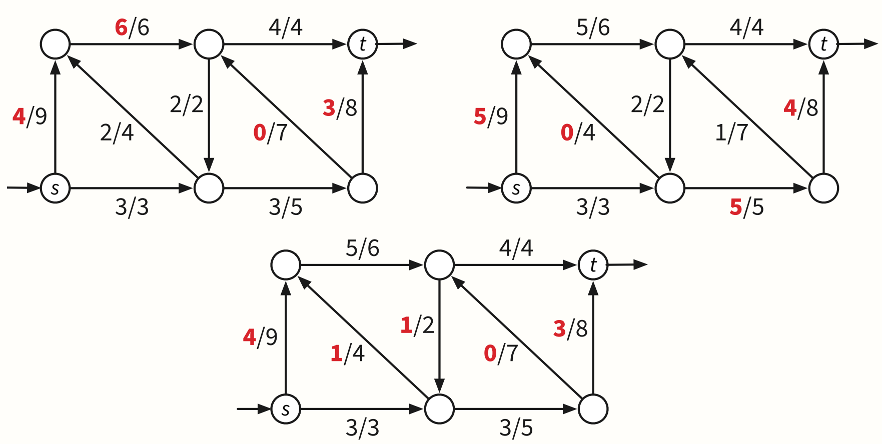


■

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:

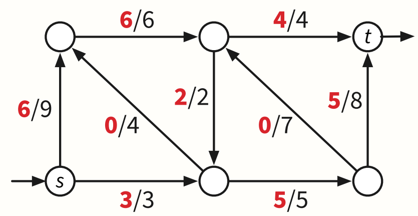


■

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.

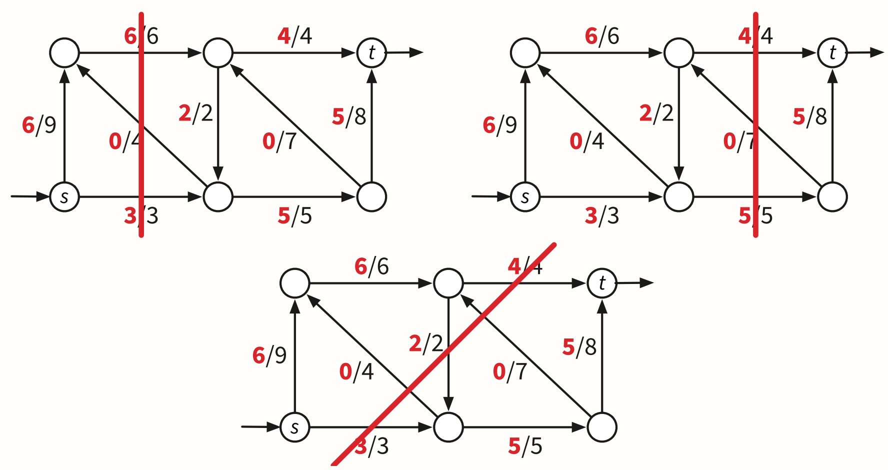


■

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

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

Solution: There are three 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 ***restrained*** if each element after the first two is (loosely) between its two immediate predecessors; that is, for every index $i > 2$, we have $\min\{x_{i-1}, x_{i-2}\} \le x_i \le \max\{x_{i-1}, x_{i-2}\}$. Describe an efficient algorithm to compute the length of the longest restrained subsequence of a given array $A[1..n]$ of numbers.

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

This function satisfies the following recurrence:

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

(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 $Rest[1..n, 1..n]$, 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 $Rest[i, j]$, so the entire algorithm runs in $\boldsymbol{O(n^3)}$ ***time***. ■

Solution (dynamic programming): For any indices $i < j < k$, let $Rest(i, j, k)$ denote the length of the longest restrained subsequence of A whose first element is $A[i]$, whose second element is $A[j]$, and whose remaining elements all come from the suffix $A[k..n]$. We need to compute $\max_{i,j} Rest(i, j, j+1)$.

This function satisfies the following recurrence:

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

We can memoize this function into a three-dimensional array $Rest[-1..n, 0..n, 1..n]$, 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:

- $V = \{(i, j) \mid 1 \le i < j \le n\}$
- $E = \{(i, j) \to (j, k) \mid \min\{A[i], A[j]\} \le A[k] \le \max\{A[i], A[j]\}\}$

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 restrained 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 $O(V + E) = \boldsymbol{O(n^3)}$ ***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 $\boldsymbol{O(n^2)}$ 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 ***bishop*** is a chess piece that attacks every square on the same diagonal or back-diagonal; that is, a bishop on square (i, j) attacks every square of the form $(i+k, j+k)$ or $(i+k, j-k)$. Describe an algorithm to places as many bishops on the board as possible, each on a legal square, so that no two bishops attack each other.

Solution: First let's establish some terminology. The dth *diagonal* consists of all squares (i, j) such that $i + j = d$, and the bth *back-diagonal* consists of all squares (i, j) such that $i - j = b$. Thus, the square in row i and column j lies on diagonal $i + j$ and back-diagonal $i - j$.

Construct a bipartite graph $G = (D \sqcup B, E)$ as follows:

- D contains a vertex for each diagonal;
- B contains a vertex for each back-diagonal;
- E contains an edge between diagonal $i + j$ and back-diagonal $i - j$ if and only if $Legal[i, j] = \text{True}$.

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 graph reduction rubric. This is not the only correct solution.

4. Suppose you buy random Pokémon cards until you own exactly $n/4$ of the n possible card types. We can break your Pokémon-collection process into *phases*; for any index k, the kth phase ends just after you purchase the kth distinct card type.

   (a) ***Prove*** that for all $1 \le k \le n/4$ and for all $m \ge 0$, the probability that you purchase more than m cards in the kth phase is at most 4^{-m}.

   Solution: During the kth phase, we own at most $k-1 < n/4$ types of cards. Thus, a random card during the kth insertion has a type we already own with probability less than $1/4$. Because cards are independent, the probability that the first m purchases all have types we already own is less than $(1/2)^m$. ■

   Rubric: 3 points.

   (b) ***Prove*** that for all $1 \le k \le n/4$, the probability that the kth phase requires more than $2\log_2 n$ purchases is at most $1/n^2$.

   Solution: Let $\#cards(k)$ denote the number of cards we purchase during the Kth phase. If we set $m = 2\log_2 n$, we immediately have

$$\begin{aligned}
\Pr[\#cards(k) > 2\log_2 n] &= \Pr[\#cards(k) > m] \\
&\le 4^{-m} && \textit{by part (a)} \\
&= 4^{-2\log_2 n} = \frac{1}{n^4} \le \frac{1}{n^2}
\end{aligned}$$

   ■

   Rubric: 2 points.

   (c) ***Prove*** that with probability at least $1 - 1/n$, none of the $n/4$ phases requires more than $2\log_2 n$ purchases.

   Solution:

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

   ■

   Rubric: 2 points.

(d) What is the *exact* expected *total* number of purchases to collect $n/4$ different card types? (A tight $O(\cdot)$ bound is worth significant partial credit.)

Solution: During the kth phase, we own exactly $k-1$ of the n card types, so the probability of each purchase having a new type is $(n-k+1)/n$. It follows that the expected number of purchases during the kth phase is exactly $n/(n-k+1)$. Linearity of expectation now implies

$$\begin{aligned}
\mathrm{E}\left[\sum_{k=1}^{n/4} \#cards(k)\right] &= \sum_{k=1}^{n/4} \mathrm{E}[\#cards(k)] \\
&= \sum_{k=1}^{n/4} \frac{n}{n-k+1} \\
&= n \cdot \sum_{k=1}^{n/4} \frac{1}{n-k+1} \\
&= n \cdot \sum_{\ell=3n/4+1}^{n} \frac{1}{\ell} && [\ell = n-k+1] \\
&= \boxed{\boldsymbol{n \cdot (H_n - H_{3n/4})}} \\
&\approx n \cdot (\ln n - \ln(3n/4)) \\
&= \ln(4/3)\, n \;\approx\; 0.28768n \;=\; \Theta(n)
\end{aligned}$$

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

$$\begin{aligned}
\mathrm{E}\left[\sum_{k=1}^{n/4} \#cards(k)\right] &= \sum_{k=1}^{n/4} \frac{n}{n-k+1} \\
&\le \sum_{k=1}^{n/4} \frac{n}{3n/4} \\
&= \sum_{k=1}^{n/4} \frac{4}{3} \;=\; \frac{n}{3} \;=\; O(n)
\end{aligned}$$

■

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 bipartite graph $G = (L \sqcup R, E)$ and a maximum matching M in G. Describe and analyze efficient algorithms for the following operations. Both of your algorithms should be significantly faster than recomputing the maximum matching from scratch. *[Hint: Think about the reduction from maximum matchings to maximum flows.]*

   (a) INSERT(u, v): Insert an edge between $u \in L$ and $v \in R$ and update the maximum matching. (You can assume that uv is not an edge before this function is called.)

Solution: First we construct a flow network H from G by adding a new source vertex s, edges from s to every vertex in L, a new target vertex t, and edges from every vertex in R to t. Finally, we direct every edge in the original graph G from L to R. Every edge in H has capacity 1. The maximum matching M corresponds to a maximum flow f^* in H.

To INSERT edge uv, we add a directed edge $u{\rightarrow}v$ with capacity 1 to the flow network H, and then perform one iteration of the Ford-Fulkerson augmenting path algorithm: Build the residual graph H_{f^*}, look for a path from s to t in H_{f^*}, and if such a path is found, push 1 unit of flow along it.

Finally, we translate the new maximum flow in H back to a matching in G. The entire algorithm runs in ***$O(E)$ time***. ■

We can prove the algorithm correct as follows. Adding one edge increases the number of edges in the maximum matching by at most 1. Every edge in the residual network H_{f^*} has capacity 1, so a single iteration of Ford-Fulkerson either fails to find a residual path (so the matching does not change) or increases the flow value by exactly 1 (so the matching size increases by 1).

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) DELETE(uv): Delete edge uv and update the maximum matching. (You can assume that uv is actually an edge before this function is called.)

Solution: Assume that the deleted edge uv is in the matching M, since otherwise, M is still a maximum matching after uv is deleted. Let $G' = G - uv$ and $M' = M - uv$. The modified matching M' is a matching in G', but it might not be a maximum matching.

To find a maximum matching in G', we convert G' into a flow network and M' into a maximum flow, run one iteration of Ford-Fulkerson, and convert the new maximum floe back into a matching, exactly as in the solution to part (a). The entire algorithm runs in ***$O(E)$ time***. ■

We can prove the algorithm correct as follows. Removing one edge decreases the number of edges in the maximum matching by at most 1. Just as in part (a), a single iteration of Ford-Fulkerson either leaves the current matching unchanged or increases the size of the matching by 1.

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.

6. Let $G = (V, E)$ be an undirected graph. The ***neighborhood*** of a vertex v consists of v and every vertex adjacent to v. A ***double-dominating set*** in G is a set S of vertices such that for each vertex v, the neighborhood of v contains at least two vertices in S.

   Suppose you are given a graph G where every vertex has degree $d-1$ (and thus the neighborhood of every vertex contains exactly d vertices), and each vertex v has a non-negative weight w_v. Your goal is to find a double-dominating set S in G whose total weight $\sum_{v \in S} w_v$ is as small as possible. Solving this problem *exactly* is NP-hard.

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

   Solution: Let $N(v)$ denote the neighborhood of any vertex v. For each vertex v, we have a variable x_v that equals 1 if $v \in S$ and 0 otherwise.

$$\begin{array}{lll} \text{minimize} & \displaystyle\sum_v w_v \cdot x_v & \\ \text{subject to} & \displaystyle\sum_{v \in N(u)} x_u \geq 2 & \text{for each vertex } u \\ & x_v \in \{0, 1\} & \text{for each vertex } v \end{array}$$

   ■

   Rubric: 5 points = 1½ for objective + 2 for double-dominating constraints + 1½ for indicator constraints

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

Solution: This subproblem was broken; the correct approximation ratio from LP rounding is actually $\mathbf{d-1}$, not $d/2$. **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 $0 \le x_v \le 1$. Let x^* denote the optimal fractional solution to this LP, and let $OPT^* = \sum_v w_v x_v^*$. 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 vertex v, let

$$x'_v = \begin{cases} 1 & \text{if } x_j^* \ge \mathbf{1/(d-1)} \\ 0 & \text{otherwise} \end{cases}$$

Correctness: For each vertex u, we have $\sum_{v \in N(u)} x_u^* \ge 2$. So there must be at least two vertices v and v' in the neighborhood $N(u)$ such that

$$x_v^* \ge 1/(d-1) \quad \text{and} \quad x_{v'}^* \ge 1/(d-1).$$

(Otherwise, even if $x_u^* = 1$, we would have $x_v^* < 1/(d-1)$ for each of the other $d-1$ vertices $v \in N(u)$, which implies $\sum_{v \in N(u)} x_v^* < 2$.) Our rounding rule implies $x'_v = x'_{v'} = 1$. It follows that $\sum_{v \in N(u)} x'_v \ge 2$ for each vertex u. In other words, x' is a feasible solution for our ILP.

Approximation ratio: For each vertex v, we have $x'_v \le (d-1) \cdot x_v^*$, so

$$\sum_v w_v x'_v \;\le\; (d-1) \cdot \sum_v w_v x_v^* \;=\; (d-1) \cdot OPT^* \;\le\; (d-1) \cdot OPT.$$

Thus x' is a $(d-1)$-approximation of the optimum double-dominating set. ■

Rubric: 5 points. Everyone gets full credit.