1. (a) Suppose you are given a dag G with a unique source s and a unique sink t. Describe an algorithm to find the smallest path cover of G in which every path starts at s and ends at t.

Solution (direct via flows): Let $G = (V, E)$ be the input graph. We construct a flow network $G' = (V', E')$ as follows:

- $V' = \{v_i, v_o \mid v \in V\}$
- $E' = \{v_i \to v_o \mid v \in V\} \cup \{u_o \to v_i \mid u \to v \in E\}$
- All edges have infinite capacity.
- Edges have lower bounds $\ell(v_i \to v_o) = 1$ for every $v \in V \setminus \{s, t\}$ and $\ell(u_o \to v_i) = 0$ for all $u \to v \in E$.

Claim 1. *For any integer k, there is a feasible integral (s_i, t_o)-flow in G' with value k if and only if G has a path cover of size k.*

Proof: Let $f : E' \to \mathbb{R}$ be a feasible integral flow in G' such that $|f| = k$. The flow decomposition theorem implies that f is the sum of k paths from s_o to t_i, each of the form $s_o \to u_i \to u_o \to v_i \to \cdots \to z_o \to t_i$. For each such path in G' there is a corresponding path $s \to u \to v \to \cdots \to z \to t$ in G; let C be the set of all k such paths. For every vertex $v \in V$, we have $f(v_i \to v_o) \geq 1$, because f is feasible, so at least one path in C contains v. We conclude that C is a path cover of size k.

Conversely, let C be a path cover of size k. For each path $s \to u \to v \to \cdots \to z \to t \in C$, there is a corresponding path $s_o \to u_i \to u_o \to v_i \to \cdots \to z_o \to t_i$ in G'. Let $f : E' \to \mathbb{R}$ be the flow obtained by summing these k paths. For each vertex v in G, at least one path in C contains v, because C is a path cover, and therefore $f(v_i \to v_o) \geq 1$. We conclude that f is a feasible flow with value k. □

Claim 1 implies that we need to find a feasible (s_o, t_i)-flow in G' with *minimum* value. We compute such a flow in two stages:

- First compute an *arbitrary* feasible (s_o, t_i)-flow f, using the reduction to maximum flows described in class and in the notes.
- Then compute a maximum (t_i, s_o)-flow f' in the residual graph G'_f. Then $f^* = f - f'$ is a minimum-value feasible flow in G.
- Finally, compute and return a path decomposition of f^*.

If we use Orlin's maxflow algorithm to compute f and f', the overall algorithm runs in $\boldsymbol{O(VE)}$ ***time***.

Solution (transitive closure, 7/8): The *transitive closure* G^T of G contains an edge $u \to v$ if and only if there is an edge from u to v in G. Since we can compute the reachable set of any vertex in $O(E)$ time by whatever-first search, we can also compute G^T in $O(VE)$ time. We retain the WFS spanning trees rooted at each node, so that we can map edges in G^T back to corresponding paths in G.

Once we have the transitive closure $G^T = (V, E^T)$, we compute the smallest *disjoint* path cover P of G^T in $O(VE^T) = O(V^3)$ time, using the algorithm described in class. Then we replace each edge of each path in P with the corresponding path in G, to obtain a non-disjoint path cover of G; this replacement requires $O(V)$ time per path, and there are at most V paths in P, so the overall postprocessing time is $O(V^2)$.

The overall algorithm runs in $\boldsymbol{O(V^3)}$ ***time***. ■

Rubric: Max 8 points = 3 for reduction to minimum feasible flow + 3 for solving minimum feasible flow via maxflow + 2 for running time. Max 7 points for the slower transitive-closure algorithm. These are not the only correct solutions. The proof for the first solution is more detailed than necessary for full credit.

(b) Describe an algorithm to find the smallest path cover of an arbitrary dag G, with no additional restrictions on the paths. *[Hint: Use part (a).]*

Solution: Let H be the graph obtained from H by adding new vertices s and t along with edges $s \to v$ and $v \to t$ for every vertex v. Any path in G can be extended to an $s \rightsquigarrow t$ path in H, and any $s \rightsquigarrow t$ path can be transformed to a path in G by deleting s and t. Thus, to compute a minimum path cover in G, we can invoke the algorithm from part (a) on H, and then remove s and t from every path in that cover. The algorithm runs in $O(VE)$ time. ■

Rubric: 2 points

2. Describe an efficient algorithm to compute a *minimum-cost* maximum flow from s to t in an (s,t)-series-parallel graph G in which every edge has capacity 1 and arbitrary cost. *[Hint: First consider the special case where G has only two vertices but lots of edges.]*

Solution (dynamic programming): First we build the series-parallel decomposition tree T of the input graph G and compute its maximum flow value, in $O(V+E)$ time, using the recursive algorithms in Homework 6 problem 3. As in the Homework 6 solutions, I will use the word "node" to refer to nodes in T and the word "vertex" to refer to vertices in G.

For any node α in T, let $G(\alpha)$ denote the corresponding subgraph of G, and let $cap(\alpha)$ denote the maximum amount of flow that can be sent through $G(\alpha)$. The algorithm from Homework 6 actually computes $cap(\alpha)$ for every node α. Because every edge has unit capacity, $cap(\alpha)$ is an integer and $cap(\alpha) \le E$ for every node α. Let $k = cap(T.root)$ be the overall maximum flow value for G.

For each node α and each integer $1 \le i \le k$, let $cost(\alpha, i)$ denote the minimum cost of sending the ith unit of flow through $G(\alpha)$. We need to compute $\sum_{i=1}^{k} cost(T.root, i)$. There are three cases to consider:

- Is α is a leaf, then
$$cost(\alpha, i) = \begin{cases} \$(\alpha.edge) & \text{if } i = 1, \\ \infty & \text{otherwise.} \end{cases}$$
- If α is a serial node, then $cost(\alpha, i) = cost(\alpha.left, i) + cost(\alpha.right, i)$.
- If α is a parallel node, then $cost(\alpha, i)$ is the ith smallest value in the set
$$\left\{ cost(\alpha.left, j),\ cost(\alpha.right, j) \;\middle|\; 1 \le i \le j \right\}.$$

We can memoize this function into an array $\alpha.cost[1..k]$ at every node α; we can fill the memoization structure with a postorder traversal of T in the outer loop, filling each array $\alpha.cost[1..k]$ in the inner loop.

We can fill the cost array at each leaf or serial node in $O(k)$ time by brute force. To compute the cost array at each parallel node α, we can *merge* the cost arrays at the children of α in $O(k)$ time, using the standard algorithm from mergesort, because the cost arrays are always sorted in non-decreasing order. The resulting algorithm runs in $\boldsymbol{O(Ek) = O(E^2)}$ ***time***.

With more care, we can improve the running time of the algorithm to $O(E \log k + Vk) = O(VE)$. Consider any parallel node α whose parent (if any) is not a parallel node. The graph $G(\alpha)$ has only two vertices and $E(\alpha)$ or more parallel edges. We can compute the cost array $\alpha.cost$ directly by sorting the costs of the k cheapest edges in $O(E_\alpha + \min\{E_\alpha, k\} \log k)$ time—instead of going through the recurrence in $O(E_\alpha k)$ time—and for the rest of the algorithm treat each parallel bundle as a single edge. The total time to preprocess all such vertices is $O(E \log k)$. An easy inductive argument implies that a series-parallel graph with no parallel edges has at most $3V$ edges, so the rest of the recursive algorithm requires only $O(Vk)$ time.

This algorithm can be extended to series-parallel graphs with *arbitrary* capacities and arbitrary costs; the running time is $O(E^2)$ time via direct dynamic programming, or $O(VE)$ time is we preprocess parallel edges first. Moreover, with appropriate data structures, which

are well beyond the scope of this class, the running time can be improved to $O(E \log E)$, even for arbitrary capacities and costs.[1] ■

Rubric: 10 points: standard dynamic-programming rubric. +3 extra credit for $O(VE)$ time.

Solution (modified successive shortest paths): We begin by computing the maximum flow value k, in $O(V+E)$ time, using the recursive algorithm in Homework 6 problem 3, and assigning balances $b(s) = -k$ and $b(t) = k$ to the terminals and $b(v) = 0$ to every other vertex in G. Finally, we compute the minimum-cost flow with value k using the successive-shortest path algorithm with one crucial modification: In each iteration, instead of recomputing the residual graph, we *delete all edges that carry non-zero flow*. (Alternatively, instead of computing the maximum flow value in advance, we can repeatedly compute and delete shortest paths from s to t until there are no such paths.)

A straightforward induction argument implies that G is a dag, and therefore any subgraph of G is a dag. Thus, at each iteration, we can compute the next shortest path in $O(E)$ time by dynamic programming. Since we push exactly 1 unit of flow in each iteration, the algorithm ends after exactly k iterations. We conclude that the overall algorithm runs in $\boldsymbol{O(Ek) = O(E^2)}$ ***time***.

But why is the algorithm correct? In fact, our algorithm is emulating the standard successive shortest-paths algorithm, but is taking advantage of a special property of series-parallel graphs to reduce the running time.

Let's define a ***dance*** in a directed graph to be a sequence of vertices, where every adjacent pair is connected by an edge in one direction or the other. Each step of a dance can be either forward along an edge or backward along an edge. A dance that only moves forward is called a ***walk***; a dance is ***simple*** if it never repeats vertices; and a simple walk is called a ***path***.

Claim 2. *Every simple dance from s to t in an (s,t)-series-parallel graph is a path.*[2]

Proof: Let G be an arbitrary (s,t)-series-parallel graph. Let Δ be a dance from s to t in G with at least one backward edge; we need to prove that Δ is not simple.

Let $v \leftarrow w$ be the first backward edge in Δ, and let $u \rightarrow v$ be the previous forward edge. Let α be the lowest node in the decomposition tree of G such that the subgraph $G(\alpha)$ contains both $u \rightarrow v$ and $w \rightarrow v$. If $u = w$, then Δ is not simple, so assume otherwise. There are three cases to consider:

- Suppose α is a leaf. Then we have an immediate contradiction; $G(\alpha)$ consists of the single edge $u \rightarrow v$, which implies $u = w$.
- Suppose α is a serial node. Our choice of α implies that $u \rightarrow v$ and $w \rightarrow v$ lie in different serial components of $G(\alpha)$. It follows that v is the unique vertex shared by those two components. But now we have a contradiction, because all edges into v come from the left component.

[1] Heather Booth and Robert E. Tarjan. Finding the minimum-cost maximum flow in a series-parallel network. *J. Algorithms* 15(3):416–446, 1993.

[2] In fact, this is a precise characterization of series-parallel graphs. A directed graph G is (s,t)-series-parallel if and only if every simple dance from s to t is a path.

- Suppose α is a parallel node. Our choice of α implies that $u \to v$ and $w \to v$ lie in different parallel components of $G(\alpha)$. It follows that v is the unique sink of $G(\alpha)$. Let z be the unique source of $G(\alpha)$. Every walk from s to u visits z (even if $u = z$). Thus, Δ visits z before visiting v and then w. If $w = z$, then Δ is not simple, so assume otherwise. Every *dance* from w to t visits either z or v. We conclude that Δ visits either z or v more than once.

In all cases, either we have a contradiction or we conclude that Δ is not simple. □

Every iteration of the successive shortest-path algorithm pushes flow along a *simple* path from s to t in the residual graph G_f. Every path in G_f is equivalent to a simple dance in the original graph G, and therefore is also a path in G by Claim 1. We conclude that *the successive shortest-path algorithm never uses a backward residual edge*. Equivalently, once an edge is saturated, we can discard it forever.

Notice that the correctness argument never uses the fact that edges have unit capacity! Indeed, this variant of the successive shortest-path algorithm can be used with *arbitrary* capacities (and arbitrary costs). Since each iteration saturates (and deletes) at least one edge, the algorithm ends after at most E iterations and therefore still runs in $O(E^2)$ time.

With more effort, we can speed up the algorithm to $\boldsymbol{O(VE)}$ ***time***. The main idea is to replace each bundle of parallel edges in G with a single parallel edge, attached to a sorted list of costs (and, if necessary, capacities); all of these sorted lists can be calculated in $O(E \log E)$ time. A simple inductive argument implies that any series-parallel graph without parallel edges has at most $3V$ edges, so we can compute the shortest path in each iteration in $O(V)$ time.

With appropriate data structures, which are well beyond the scope of this class,[3] we can further improve the running time to $O(E \log E)$.

Rubric: 10 points = 4 for algorithm + 2 for time bound + 4 for proof of correctness.

[3] Different data structures than the ones Boot and Tarjan used!

3. Describe and analyze an efficient algorithm that either solves Dumbledore's problem, or correctly reports that there is no valid assignment whose total cost is finite.

Solution: We modify the standard reduction from bipartite matching to maximum flows. Define a flow network $G = (V, E)$ as follows:

- V contains a source vertex s, one vertex p_i for each professor, and one vertex c_j for each committee.
- E contains two types of edges:
  - Edges $s \to p_i$ for each professor, each with capacity 3 and cost 0.
  - Edges $p_i \to c_j$ for each professor i and committee j such that professor i is both suitable and willing to serve on committee j. Each such edge has capacity 1 and cost equal to that professor's price for serving on that committee.
- For each committee j, the balance $b(c_j)$ is the number of instructors required for that committee. The source vertex s has balance $-\sum_j b(c_j)$, and all faculty vertices have balance 0.

Let f^* be the minimum-cost feasible flow in G. Because all capacities and balances are integers, we can assume that f^* is integral, so we can decompose f^* into $|f^*|$ paths of the form $s \to p_i \to c_j$. For each such path, assign professor i to committee j.

We compute the minimum-cost flow in G using the FastSuccessiveShortestPaths algorithm described in section G.3 of the lecture notes. The algorithm halts after at most $3n$ iterations, because the edges leaving s have total capacity $3n$, and each iteration runs in $O(E \log V)$ time. Thus, the overall running time is $O(nE \log V) = O(VE \log V) = \boldsymbol{O(N^2 \log N)}$ ***time***, where $N \le 2cn$ is the total size of the input lists. (Orlin's algorithm, cited in the same section of the notes, yields a running time of $O(N^2 \log^2 N)$.)

To prove that the algorithm correct, we follow the usual strategy for matchings.

- Suppose there is a valid committee assignment with cost k. Construct a flow $f : E \to \mathbb{R}$ as follows:

$$f(s \to p_i) = \text{the number of committees assigned to professor } i$$
$$f(p_i \to c_j) = 1 \text{ if professor } i \text{ assigned to committee } j, \text{ 0 otherwise}$$

  Routine definition-chasing implies that f is a feasible flow with cost k.
- On the other hand, given a feasible flow f with cost k, our algorithm constructs a valid committee assignment with cost k.

We conclude that the cost of the cheapest feasible flow in G is equal to the cost of the cheapest valid committee assignment; thus, our algorithm is correct. ■

Rubric: 10 points: standard graph-reduction rubric. No penalty for using a slower polynomial-time min-cost flow algorithm.