A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it.

— The First Law of Mentat, in Frank Herbert's *Dune* (1965)

Contrary to expectation, flow usually happens not during relaxing moments of leisure and entertainment, but rather when we are actively involved in a difficult enterprise, in a task that stretches our mental and physical abilities.... Flow is hard to achieve without effort. Flow is not "wasting time."

— Mihaly Csíkszentmihályi, *Flow: The Psychology of Optimal Experience* (1990)

There's a difference between knowing the path and walking the path.

— Morpheus [Laurence Fishburne], *The Matrix* (1999)

10

Maximum Flows & Minimum Cuts

In the mid-1950s, U. S. Air Force researcher Theodore E. Harris and retired U. S. Army general Frank S. Ross wrote a classified report studying the rail network that linked the Soviet Union to its satellite countries in Eastern Europe. The network was modeled as a graph with 44 vertices, representing geographic regions, and 105 edges, representing links between those regions in the rail network. Each edge was given a weight, representing the rate at which material could be shipped from one region to the next. Essentially by trial and error, they determined both the maximum amount of stuff that could be moved from Russia into Europe, as well as the cheapest way to disrupt the network by removing links (or in less abstract terms, blowing up train tracks), which they called "the bottleneck". Their report, which included the drawing of the network in Figure 10.1, was only declassified in 1999.[1]

[1]I learned this story from Alexander Schrijver's fascinating survey "On the history of combinatorial optimization (till 1960)"; the Harris-Ross report was declassified at Schrijver's request. Ford and Fulkerson (who we will meet shortly) credit Harris for formulating the

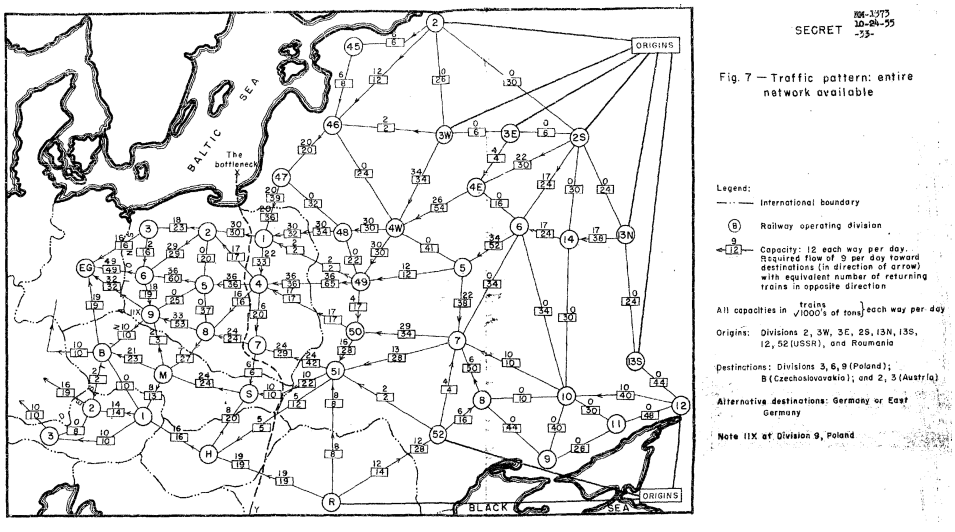

Figure 10.1. Harris and Ross's map of the Warsaw Pact rail network. (See Image Credits at the end of the book.)

This one of the first recorded applications of the ***maximum flow*** and ***minimum cut*** problems. For both problems, the input is a directed graph $G = (V, E)$ with two special vertices s and t, called the *source* and *target*. As in previous chapters, I will write $u \to v$ to denote the directed edge from vertex u to vertex v. Intuitively, the maximum flow problem asks for the maximum rate at which some resource can be moved from s to t; the minimum cut problem asks for the minimum damage needed to separate s from t.

10.1 Flows

An ***(s, t)-flow*** (or just a ***flow*** if the source and target vertices are clear from context) is a function $f : E \to \mathbb{R}$ that satisfies the following ***conservation constraint*** at every vertex v except possibly s and t:

$$\sum_u f(u \to v) = \sum_w f(v \to w).$$

In English, the total flow into v is equal to the total flow out of v. To keep the notation simple, we define $f(u \to v) = 0$ if there is no edge $u \to v$ in the graph. The ***value*** of the flow f, denoted $|f|$, is the total net flow out of the source vertex s:

$$|f| := \sum_w f(s \to w) - \sum_u f(u \to s).$$

maximum-flow problem, although the precise chronology is somewhat muddled; Harris and Ross thank George Dantzig "for assistance in formulating the problem".

It's not hard to prove that $|f|$ is also equal to the total net flow *into* the target vertex t, as follows. To simplify notation, let $\partial f(v)$ denote the total net flow out of any vertex v:

$$\partial f(v) := \sum_u f(u \to v) - \sum_w f(v \to w).$$

The conservation constraint implies that $\partial f(v) = 0$ or every vertex v except s and t, so

$$\sum_v \partial f(v) = \partial f(s) + \partial f(t).$$

On the other hand, any flow that leaves one vertex must enter another vertex, so we must have $\sum_v \partial f(v) = 0$. It follows immediately that $|f| = \partial f(s) = -\partial f(t)$.

Now suppose we have another function $c : E \to \mathbb{R}_{\geq 0}$ that assigns a non-negative ***capacity*** $c(e)$ to each edge e. We say that a flow f is ***feasible*** (with respect to c) if $0 \leq f(e) \leq c(e)$ for every edge e. Most of the time we consider only flows that are feasible with respect to some fixed capacity function c. We say that a flow f ***saturates*** edge e if $f(e) = c(e)$, and ***avoids*** edge e if $f(e) = 0$. The ***maximum flow problem*** is to compute a feasible (s, t)-flow in a given directed graph, with a given capacity function, whose value is as large as possible.

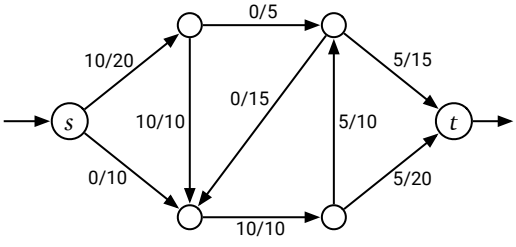

Figure 10.2. A feasible (s, t)-flow with value 10. Each edge is labeled with its flow/capacity.

10.2 Cuts

An ***(s, t)-cut*** (or just ***cut*** if the source and target vertices are clear from context) is a partition of the vertices into disjoint subsets S and T—meaning $S \cup T = V$ and $S \cap T = \emptyset$—where $s \in S$ and $t \in T$.

If we have a capacity function $c : E \to \mathbb{R}_{\geq 0}$, the ***capacity*** of a cut is the sum of the capacities of the edges that start in S and end in T:

$$\|S, T\| := \sum_{v \in S} \sum_{w \in T} c(v \to w).$$

(Again, if $v \to w$ is not an edge in the graph, we assume $c(v \to w) = 0$.) Notice that the definition is asymmetric; edges that start in T and end in S are unimportant.

The ***minimum cut problem*** is to compute an (s,t)-cut whose capacity is as small as possible.

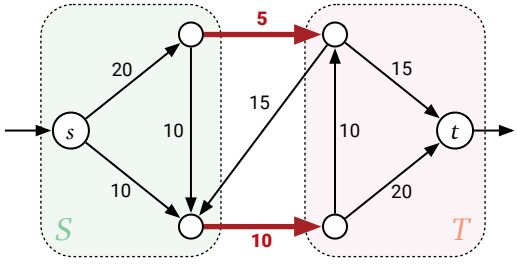

Figure 10.3. An (s,t)-cut with capacity 15. Each edge is labeled with its capacity.

Intuitively, the minimum cut is the cheapest way to disrupt all flow from s to t. Indeed, it is not hard to show the following relationship between flows and cuts:

Lemma 10.1. *Let f be **any** feasible (s,t)-flow, and let (S,T) be **any** (s,t)-cut. The value of f is at most the capacity of (S,T). Moreover, $|f| = \|S,T\|$ if and only if f saturates every edge from S to T and avoids every edge from T to S.*

Proof: Choose your favorite flow f and your favorite cut (S,T), and then follow the bouncing inequalities:

$$
\begin{aligned}
|f| &= \partial f(s) && \text{[by definition]} \\
&= \sum_{v \in S} \partial f(v) && \text{[conservation constraint]} \\
&= \sum_{v \in S} \sum_{w} f(v \to w) - \sum_{v \in S} \sum_{u} f(u \to v) && \text{[math, definition of } \partial\text{]} \\
&= \sum_{v \in S} \sum_{w \notin S} f(v \to w) - \sum_{v \in S} \sum_{u \notin S} f(u \to v) && \text{[removing edges from } S \text{ to } S\text{]} \\
&= \sum_{v \in S} \sum_{w \in T} f(v \to w) - \sum_{v \in S} \sum_{u \in T} f(u \to v) && \text{[definition of cut]} \\
&\leq \sum_{v \in S} \sum_{w \in T} f(v \to w) && \text{[because } f(u \to v) \geq 0\text{]} \\
&\leq \sum_{v \in S} \sum_{w \in T} c(v \to w) && \text{[because } f(v \to w) \leq c(v \to w)\text{]} \\
&= \|S,T\| && \text{[by definition]}
\end{aligned}
$$

In the second step, we are just adding zeros, because $\partial f(v) = 0$ for every vertex $v \in S \setminus \{s\}$. In the fourth step, we are removing flow values $f(x \to y)$ where

both x and y are in S, because they appear in both sums: positively when $v = x$ and $w = y$, and negatively when $v = y$ and $u = x$.

The first inequalities in this derivation is actually an equality if and only if f avoids every edge from T to S. Similarly, the second inequality is actually an equality if and only if f saturates every edge from S to T. □

This lemma immediately implies that if $|f| = \|S, T\|$, then f must be a maximum flow, and (S, T) must be a minimum cut.

10.3 The Maxflow-Mincut Theorem

Surprisingly, in every flow network, there is a feasible (s, t)-flow f and an (s, t)-cut (S, T) such that $|f| = \|S, T\|$. This is the famous *Maxflow-Mincut Theorem*, first proved by Lester Ford (of shortest-path fame) and Delbert Fulkerson in 1954 and independently by Peter Elias, Amiel Feinstein, and Claude Shannon (of information-theory and maze-solving-robot fame) in 1956.

The Maxflow-Mincut Theorem. *In every flow network with source s and target t, the value of the maximum (s, t)-flow is* equal *to the capacity of the minimum (s, t)-cut.*

Ford and Fulkerson proved this theorem as follows. Fix a graph G, vertices s and t, and a capacity function $c : E \to \mathbb{R}_{\geq 0}$. The proof will be easier if we assume that G is ***reduced***, meaning there is at most one edge between any two vertices u and v. In particular, either $c(u \to v) = 0$ or $c(v \to u) = 0$. This assumption is easy to enforce: Subdivide each edge $u \to v$ in G with a new vertex x, replacing $u \to v$ with a path $u \to x \to v$, and define $c(u \to x) = c(x \to v) = c(u \to v)$. The modified graph has the same maximum flow value and minimum cut capacity as the original graph.

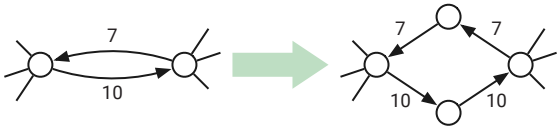

Figure 10.4. Enforcing the one-direction assumption.

Let f be an arbitrary feasible (s, t)-flow in G. We define a new capacity function $c_f : V \times V \to \mathbb{R}$, called the ***residual capacity***, as follows:

$$c_f(u \to v) = \begin{cases} c(u \to v) - f(u \to v) & \text{if } u \to v \in E \\ f(v \to u) & \text{if } v \to u \in E \\ 0 & \text{otherwise} \end{cases}$$

Intuitively, the residual capacity of an edge indicates how much *more* flow can be pushed through that edge. Because $f \geq 0$ and $f \leq c$, these residual capacities are always non-negative. It is possible to have $c_f(u{\to}v) > 0$ even if $u{\to}v$ is not an edge in the original graph G. Thus, we define the ***residual graph*** $G_f = (V, E_f)$, where E_f is the set of edges whose residual capacity is positive. Most residual graphs are *not* reduced; in particular, if $0 < f(u{\to}v) < c(u{\to}v)$, then the residual graph G_f contains both $u{\to}v$ and its reversal $v{\to}u$.

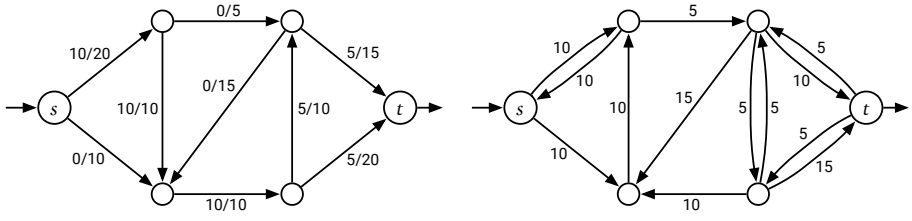

Figure 10.5. A flow f in a weighted graph G and the corresponding residual graph G_f.

Now we have two cases to consider: Either there is a directed path from the source vertex s to the target vertex t in the residual graph G^f, or there isn't.

First suppose the residual graph G_f contains a directed path P from s to t; we call P an ***augmenting path***. Let $F = \min_{u\to v\in P} c_f(u{\to}v)$ denote the maximum amount of flow that we can push through P. We define a new flow $f' \colon E \to \mathbb{R}$ (in the original graph) as follows:

$$f'(u{\to}v) = \begin{cases} f(u{\to}v) + F & \text{if } u{\to}v \in P \\ f(u{\to}v) - F & \text{if } v{\to}u \in P \\ f(u{\to}v) & \text{otherwise} \end{cases}$$

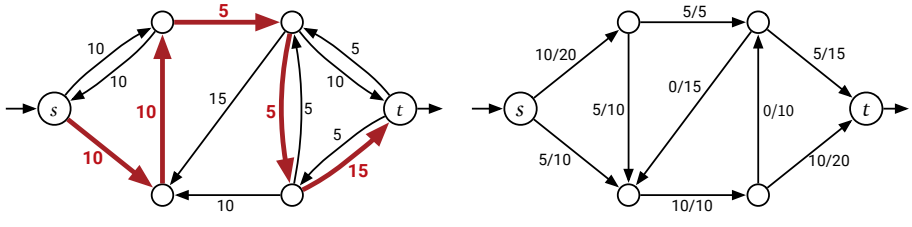

Figure 10.6. An augmenting path with value $F = 5$ and the resulting augmented flow f'.

I claim that this new flow f' is feasible with respect to the original capacities c, meaning $f' \geq 0$ and $f' \leq c$ everywhere. Consider an edge $u{\to}v$ in the original graph G. There are three cases to consider.

- If the augmenting path P contains $u{\to}v$, then

$$f'(u{\to}v) = f(u{\to}v) + F > f(u{\to}v) \geq 0$$

because f is feasible, and

$$\begin{aligned} f'(u{\to}v) &= f(u{\to}v) + F && \text{by definition of } f' \\ &\le f(u{\to}v) + c_f(u{\to}v) && \text{by definition of } F \\ &= f(u{\to}v) + c(u{\to}v) - f(u{\to}v) && \text{by definition of } c_f \\ &= c(u{\to}v) && \text{Duh.} \end{aligned}$$

- If the augmenting path P contains the reversed edge $v{\to}u$, then

$$f'(u{\to}v) \;=\; f(u{\to}v) - F \;<\; f(u{\to}v) \;\le\; c(u{\to}v),$$

again because f is feasible, and

$$\begin{aligned} f'(u{\to}v) &= f(u{\to}v) - F && \text{by definition of } f' \\ &\ge f(u{\to}v) - c_f(v{\to}u) && \text{by definition of } F \\ &= f(u{\to}v) - f(u{\to}v) && \text{by definition of } c_f \\ &= 0 && \text{Duh.} \end{aligned}$$

- Finally, if neither $u{\to}v$ nor $v{\to}u$ is in the augmenting path, then $f'(u{\to}v) = f(u{\to}v)$, and therefore $0 \le f'(u{\to}v) \le c(u{\to}v)$, because f is feasible.

So f is indeed feasible.

Finally, only the first edge in the augmenting path leaves s, which implies $|f'| = |f| + F > |f|$. Thus, f' is a feasible flow with larger value than f. We conclude that if there is a path from s to t in the residual graph G_f, then f is *not* a maximum flow.

On the other hand, suppose the residual graph G_f does *not* contain a directed path from s to t. Let S be the set of vertices that are reachable from s in G_f, and let $T = V \setminus S$. The partition (S, T) is clearly an (s,t)-cut. For every vertex $u \in S$ and $v \in T$, we have

$$c_f(u{\to}v) \;=\; (c(u{\to}v) - f(u{\to}v)) + f(v{\to}u) \;=\; 0.$$

The feasibility of f implies $c(u{\to}v) - f(u{\to}v) \ge 0$ and $f(v{\to}u) \ge 0$, so in fact we must have $c(u{\to}v) - f(u{\to}v) = 0$ and $f(v{\to}u) = 0$. In other words, our flow f saturates every edge from S to T and avoids every edge from T to S. Lemma 10.1 now implies that $|f| = \|S, T\|$, which means f is a maximum flow and (S, T) is a minimum cut.

This completes the proof! □

10.4 Ford and Fulkerson's augmenting-path algorithm

Ford and Fulkerson's proof of the Maxflow-Mincut Theorem immediately suggests an algorithm to compute maximum flows: Starting with the zero flow, repeatedly augment the flow along **any** path from s to t in the residual graph, until there is no such path.

This algorithm has an important but straightforward corollary:

Integrality Theorem. *If all capacities in a flow network are integers, then there is a maximum flow such that the flow through every edge is an integer.*

Proof: We argue by induction that after each iteration of the augmenting path algorithm, all flow values and all residual capacities are integers.

- Before the first iteration, all flow values are 0 (which is an integer), and all residual capacities are the original capacities, which are integers by definition.
- In each later iteration, the induction hypothesis implies that the capacity F of the augmenting path is an integer, so augmenting changes the flow on each edge, and therefore the residual capacity of each edge, by an integer.

In particular, each iteration of the augmenting path algorithm increases the value of the flow by a positive integer. It follows that the algorithm eventually halts and returns a maximum flow. □

If every edge capacity is an integer, then conservatively, the Ford-Fulkerson algorithm halts after at most $|f^*|$ iterations, where f^* is the actual maximum flow. In each iteration, we can build the residual graph G_f and perform a whatever-first-search to find an augmenting path in $O(E)$ time. Thus, in this setting, the algorithm runs in $\boldsymbol{O(E|f^*|)}$ time in the worst case.

Jack Edmonds and Richard Karp observed that this running time analysis is essentially tight. Consider the 4-node network in Figure 10.7, where X is some large integer. The maximum flow in this network is clearly $2X$. However, Ford-Fulkerson might alternate between pushing one unit of flow along the augmenting path $s \to u \to v \to t$ and then pushing one unit of flow along the augmenting path $s \to v \to u \to t$, leading to a running time of $\Theta(X) = \Omega(|f^*|)$.

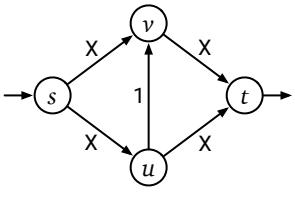

Figure 10.7. Edmonds and Karp's bad example for the Ford-Fulkerson algorithm.

Ford and Fulkerson's algorithm is usually fast in practice, and it is always fast when the maximum flow value $|f^*|$ is small, but without further constraints on the augmenting paths, this is *not* an efficient algorithm in worst case. Edmonds and Karp's bad example network can be described using only $O(\log X)$ bits; thus, the running time of Ford-Fulkerson is actually *exponential* in the input size.

♥Irrational Capacities

But what if the capacities are *not* integers? If we multiply all the capacities by the same (positive) constant, the maximum flow increases everywhere by the same constant factor. It follows that if all the edge capacities are *rational*, then the Ford-Fulkerson algorithm eventually halts, although still in exponential time (in the number of bits used to described the input).

However, if we allow *irrational* capacities, the algorithm can actually loop forever, always finding smaller and smaller augmenting paths. Worse yet, this infinite sequence of augmentations may not even converge to the maximum flow, or even to a significant fraction of the maximum flow! The smallest network that exhibits this bad behavior was discovered by Uri Zwick in 1993.[2]

Consider the six-node network shown in Figure 10.8. Six of the nine edges have some large integer capacity X, two have capacity 1, and one has capacity $\phi = (\sqrt{5}-1)/2 \approx 0.618034$, chosen so that $1-\phi = \phi^2$. To prove that the Ford-Fulkerson algorithm can get stuck, we can watch the residual capacities of the three horizontal edges as the algorithm progresses. (The residual capacities of the other six edges will always be at least $X-3$.)

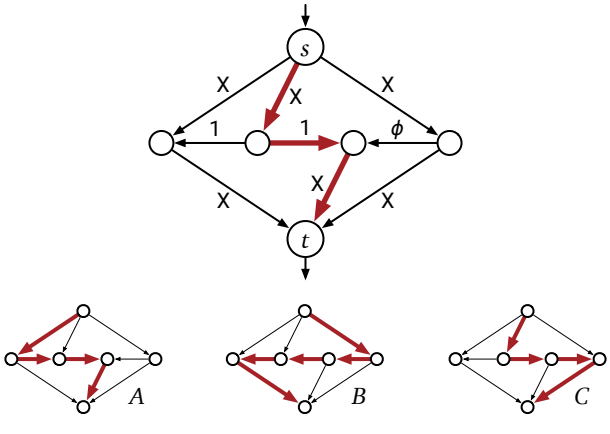

Figure 10.8. Uri Zwick's non-terminating flow example, and three augmenting paths.

Suppose the Ford-Fulkerson algorithm starts by choosing the central augmenting path, shown at the top of Figure 10.8. The three horizontal edges,

[2] In 1962, Ford and Fulkerson described a more complex network, with 10 vertices and 48 edges, with the same bad behavior.

in order from left to right, now have residual capacities 1, 0, and ϕ. Suppose inductively that the horizontal residual capacities are ϕ^{k-1}, 0, and ϕ^k for some non-negative integer k.

1. Augment along path B, adding ϕ^k to the flow; the residual capacities are now ϕ^{k+1}, ϕ^k, and 0.
2. Augment along path C, adding ϕ^k to the flow; the residual capacities are now ϕ^{k+1}, 0, and ϕ^k.
3. Augment along path B, adding ϕ^{k+1} to the flow; the residual capacities are now 0, ϕ^{k+1}, and ϕ^{k+2}.
4. Augment along path A, adding ϕ^{k+1} to the flow; the residual capacities are now ϕ^{k+1}, 0, and ϕ^{k+2}.

It follows by induction that after $4n+1$ augmentation steps, the horizontal edges have residual capacities ϕ^{2n-2}, 0, and ϕ^{2n-1}. As the number of augmentations grows to infinity, the value of the flow converges to

$$1 + 2\sum_{i=1}^{\infty} \phi^i = 1 + \frac{2}{1-\phi} = 4 + \sqrt{5} < 7,$$

even though the maximum flow value is clearly $2X + 1 \gg 7$.

Practically-minded readers might wonder why anyone should care about irrational capacities; after all, computers can't represent anything but (small) integers or (small dyadic) rationals exactly. Good question! The mathematician's answer is that the restriction to integer capacities is literally *artificial*; it's an *artifact* of digital computational hardware (or perhaps the otherwise irrelevant laws of physics), not an inherent feature of the abstract computational problem. But a more practical reason is that the behavior of the algorithm with irrational inputs tells us something about its worst-case behavior *in practice* with floating-point capacities—terrible! Even with very reasonable capacities, a careless implementation of Ford-Fulkerson could enter an infinite loop, simply because of round-off error, without ever coming close to the correct answer.

10.5 Combining and Decomposing Flows

Flows are normally defined as functions on the edges of a graph satisfying certain constraints at the vertices. However, flows have a second characterization that is more natural and useful in certain contexts.

Consider an arbitrary graph G with source vertex s and target vertex t. Fix any two (s,t)-flows f and g and any two real numbers α and β, and consider the function $h : E \to \mathbb{R}$ defined by setting

$$h(u \to v) := \alpha \cdot f(u \to v) + \beta \cdot g(u \to v)$$

for every edge $u \to v$; we can write this definition more simply as $h = \alpha f + \beta g$. Straightforward definition-chasing implies that h is also an (s,t)-flow with value $|h| = \alpha|f| + \beta|g|$. More generally, any linear combination of (s,t)-flows is also an (s,t)-flow.

It turns out that any (s,t)-flow can be written as a weighted sum of flows with a special structure. For any directed path P from s to t, we define a corresponding ***path flow*** as follows:

$$P(u \to v) = \begin{cases} 1 & \text{if } u \to v \in P, \\ -1 & \text{if } v \to u \in P, \\ 0 & \text{otherwise.} \end{cases}$$

Straightforward definition-chasing implies that the function $P : E \to \mathbb{R}$ is indeed an (s,t)-flow with value 1. I am deliberately overloading the variable P to mean both the path (a sequence of vertices and directed edges) and the unit flow along that path.

Similarly, for any directed cycle C, we define a corresponding ***cycle flow*** by setting

$$C(u \to v) = \begin{cases} 1 & \text{if } u \to v \in C, \\ -1 & \text{if } v \to u \in C, \\ 0 & \text{otherwise.} \end{cases}$$

Again, it is easy to verify that $C : E \to \mathbb{R}$ is an (s,t)-flow with value zero.

Our earlier argument implies that any linear combination of path flows and cycle flows is another flow; this weighted sum is called a ***flow decomposition***. Moreover, *every* non-negative flow has a flow decomposition with the following special structure.

Flow Decomposition Theorem. *Every non-negative (s,t)-flow f can be written as a positive linear combination of directed (s,t)-paths and directed cycles. Moreover, a directed edge $u \to v$ appears in at least one of these paths or cycles if and only if $f(u \to v) > 0$, and the total number of paths and cycles is at most the number of edges in the network.*

Proof: We prove the theorem by induction on the number of edges carrying non-zero flow, intuitively by running the Ford-Fulkerson algorithm backward. As long as at least one edge in the graph carries positive flow, we can find either an (s,t)-path or a directed cycle that carries flow. Subtracting as much flow as possible from that path or cycle empties at least one edge, so the Recursion Fairy can give us the rest of the decomposition.

To formalize this argument, we first consider the special case of ***circulations***; these are flows with value 0, where flow is conserved at *every* vertex. Fix

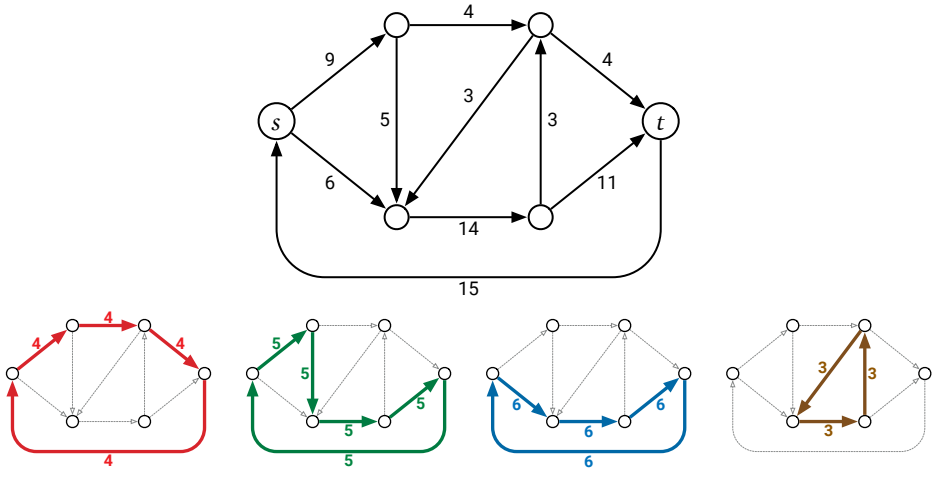

Figure 10.9. Decomposing a circulation into weighted directed cycles.

an arbitrary circulation f in an arbitrary flow network, and let $\#f$ denote the number of edges $u \to v$ such that $f(u \to v) > 0$. We prove that f can be decomposed into a positive linear combination of at most $\max\{0, \#f - 1\}$ cycles, by induction on $\#f$. There are three cases to consider:

- If $\#f = 0$, then f is vacuously a linear combination of zero cycles.
- Suppose $f(u \to v) > 0$ for a single directed cycle of edges $u \to v$. Then $\#f \geq 2$, and f is trivially a linear combination of one cycle.
- Otherwise, pick an arbitrary edge $u \to v$ with $f(u \to v) > 0$. Consider an arbitrary walk $v_0 \to v_1 \to v_2 \to \cdots$ with $v_0 = u$ and $v_1 = v$, such that $f(v_{i-1} \to v_i) > 0$ for every index i. The conservation constraint implies that every vertex with incoming flow also has outgoing flow, so we can make this walk arbitrarily long; in particular, the walk must eventually visit some vertex more than once. Let k be the smallest index such that $v_j = v_k$ for some index $j < k$. The subwalk $v_j \to v_{j-1} \to \cdots \to v_k$ is a simple directed cycle C.

  Define $F := \min_{e \in C} f(e)$, and consider the function $f' := f - F \cdot C$, or more verbosely,

$$f'(u \to v) := \begin{cases} f(u \to v) - F & \text{if } u \to v \in C, \\ f(u \to v) & \text{otherwise.} \end{cases}$$

  Straightforward definition-chasing shows that f' is another feasible circulation in G. There is at least one edge $e \in C$ such that $f(e) = F$, and therefore $f'(e) = 0$, which implies $\#f' \leq \#f - 1$. Since fewer edges carry flow in f' than in f, the Recursion Fairy can decompose f' into at most $\#f' - 1 \leq \#f - 2$ cycles. Adding F units of flow around cycle C gives us a flow decomposition for f; more succinctly: $f = f' + F \cdot C$.

Now let f be an arbitrary (s,t)-flow in an arbitrary flow network, such that $|f| > 0$. Add an edge $t \to s$ to the network, and define a circulation f' by setting $f'(t \to s) = |f|$ and $f'(u \to v) = f(u \to v)$ for every original edge $u \to v$; observe that $\#f' = \#f + 1 \geq 2$. The previous argument implies that the circulation f' is a positive linear combination of at most $\#f' - 1$ directed cycles. Deleting the edge $t \to s$ gives us a decomposition of the original flow f into at most $\#f' - 1 = \#f$ paths and cycles. Specifically, cycles in f' that include $t \to s$ become (s,t)-paths in f, and cycles in f' that do not include $t \to s$ remain cycles in f. □

The proof of the Flow Decomposition Theorem implies stronger results in two interesting special cases.

- Any circulation can be decomposed into a weighted sum of cycles; no paths are necessary.
- Any ***acyclic*** (s,t)-flow can be decomposed into a weighted sum of (s,t)-paths; no cycles are necessary.

Moreover, by canceling flow cycles until no more remain, we can transform any flow into an acyclic flow with the same value. In particular, every flow network supports a maximum (s,t)-flow that is acyclic.

The proof also immediately translates directly into an algorithm, similar to Ford-Fulkerson, to decompose any (s,t)-flow into paths and cycles. The algorithm repeatedly seeks either a directed (s,t)-path or a directed cycle in the remaining flow, and then subtracts as much flow as possible along that path or cycle, until the flow is empty. We can find a flow path or cycle in $O(V)$ time as follows:

- If any edge leaving s has positive flow, follow an arbitrary walk from s in the flow graph until it either reaches t (giving us a flow path) or reaches some vertex for the second time (giving us a flow cycle).
- If no edge leaving s has positive flow, find any other vertex v with positive outflow, and follow an arbitrary walk from v in the flow graph until it reaches some vertex for the second time (giving us a flow cycle).

In both cases, the conservation constraint implies that this algorithm will never get stuck. Each iteration takes $O(V)$ time and removes at least one edge from the flow graph; thus, the entire decomposition algorithm runs in $\boldsymbol{O(VE)}$ ***time***.

Flow decompositions provide a natural lower bound on the running time of any maximum-flow algorithm that builds the flow one path or cycle at a time. Every flow can be decomposed into at most E paths and cycles, each of which uses at most V edges, so the overall complexity of the flow decomposition is $O(VE)$. Moreover, it is easy to construct flows for which *every* flow decomposition has complexity $\Omega(VE)$. Thus, any maximum-flow algorithm that explicitly constructs a flow one path or cycle at a time—in particular, any implementation

of Ford and Fulkerson's augmenting path algorithm—must take $\Omega(VE)$ time in the worst case.

10.6 Edmonds and Karp's Algorithms

Ford and Fulkerson's algorithm does not specify which path in the residual graph to augment; the poor worst-case behavior of the algorithm can be blamed on poor choices for the augmenting path. In the early 1970s, Jack Edmonds and Richard Karp published two natural rules for greedily choosing augmenting paths, both of which led to more efficient algorithms.

Fattest Augmenting Paths

Edmonds and Karp's first rule is perhaps the most natural greedy algorithm:

> Choose the augmenting path with largest bottleneck value.

It's not hard to show that the maximum-bottleneck (s, t)-path in a directed graph can be computed in $O(E \log V)$ time using a "best-first" traversal, similar to Jarník's minimum-spanning-tree algorithm or Dijkstra's shortest-path algorithm. The algorithm grows a directed tree T, rooted at s, one vertex at a time, by repeatedly adding the highest-capacity edge leaving T to T, until T contains a path from s to t. Alternately, one could emulate Kruskal's algorithm—insert edges one at a time in decreasing capacity order until there is a path from s to t—although this approach is less efficient, at least when the graph is directed.

To complete the running-time analysis of the flow algorithm, we need an upper bound on the number of iterations before the algorithm halts. In fact, for arbitrary real capacities, the algorithm may *never* halt; see Exercise 18. For integer capacities, however, we can bound the number of iterations as a function of the maximum flow value $|f^*|$, as follows.

Let f be any flow in G, and let f' be the maximum flow *in the current residual graph* G_f. (At the beginning of the algorithm, $G_f = G$ and $f' = f^*$.) We have already proved that f' can be decomposed into at most E paths and cycles. A simple averaging argument implies that at least one of the paths in this decomposition must carry at least $|f'|/E$ units of flow. It follows immediately that the *fattest* (s, t)-path in G_f carries at least $|f'|/E$ units of flow.

Thus, augmenting f along the maximum-bottleneck path in G_f multiplies the value of the remaining maximum flow in G_f by a factor of at most $1 - 1/E$. In other words, the residual maximum flow value *decays exponentially* with the number of iterations. After $E \cdot \ln|f^*|$ iterations, the maximum flow value in G_f is at most

$$|f^*| \cdot (1 - 1/E)^{E \cdot \ln|f^*|} < |f^*|\, e^{-\ln|f^*|} = 1.$$

(That's Euler's constant e, not the edge e. Sorry.) In particular, after $E \cdot \ln|f^*|$ iterations, the residual maximum flow value is less than 1. *If all capacities are integers*, the residual maximum flow value is also an integer, so it must be 0; in other words, f is a maximum flow!

We conclude that for graphs with integer capacities, the Edmonds-Karp "fattest path" algorithm runs in $\boldsymbol{O(E^2 \log E \log|f^*|)}$ time. Unlike the worst-case running time of raw Ford-Fulkerson, this time bound is actually a polynomial function of the input size.

Just like the original Ford-Fulkerson algorithm, the "fattest path" algorithm can get stuck in an infinite loop in networks with arbitrary real capacities. However, our analysis implies that even if the algorithm never halts, it maintains a flow f that approaches a maximum flow in the limit.

Shortest Augmenting Paths

The second Edmonds-Karp rule was actually proposed as a practical heuristic by Ford and Fulkerson in their original maximum-flow paper; a variant of this rule was independently proposed in 1970 by the Russian mathematician Yefim Dinitz.[3]

> Choose the augmenting path with the smallest number of edges.

The shortest augmenting path can be found in $O(E)$ time by running breadth-first search in the residual graph. Surprisingly, the resulting algorithm halts after a polynomial number of iterations, independent of the actual edge capacities!

The proof of this polynomial upper bound relies on two observations about the evolution of the residual graph. Let f_i be the current flow after i augmentation steps, let G_i be the corresponding residual graph. In particular, f_0 is zero everywhere and $G_0 = G$. For each vertex v, let $\boldsymbol{level_i(v)}$ denote the unweighted shortest-path distance from s to v in G_i, or equivalently, the *level* of v in a breadth-first search tree of G_i rooted at s. In particular, if there is no path from s to v in G_i, then $level_i(v) = \infty$ (because $\min \varnothing = \infty$).

Our first observation is that the level of a vertex can only increase over time.

Lemma 10.2. *$level_i(v) \geq level_{i-1}(v)$ for all vertices v and all integers $i > 0$.*

Proof: Fix an arbitrary positive integer $i > 0$ and an arbitrary vertex v. We prove the claim by induction on $level_i(v)$ (and *not* on the integer i). As an inductive hypothesis, assume for every vertex u such that $level_i(u) < level_i(v)$, that $level_i(u) \geq level_{i-1}(u)$. There are three cases to consider.

[3] Specifically, Dinitz discovered a more complex maximum-flow algorithm, while he was a student in an algorithms class taught by Georgy Adelson-Velsky (the "AV" in AVL trees), in response to an in-class exercise. Dinitz's algorithm also pushes flows along shortest paths, but with additional bookkeeping to reduce the running time from $O(VE^2)$ to $O(V^2E)$.

- If $v = s$, we immediately have $level_i(s) = level_{i-1}(s) = 0$.
- If there is no path from s to v in G_i, then $level_i(v) = \infty \geq level_{i-1}(v)$.
- Otherwise, let $s \to \cdots \to u \to v$ be any unweighted shortest path from s to v in the graph G_i. Because this is a shortest path, we have $level_i(v) = level_i(u) + 1$, so the inductive hypothesis implies $level_i(u) \geq level_{i-1}(u)$. To complete the proof, we need to show that $level_{i-1}(u) \geq level_{i-1}(v) - 1$. We have two subcases to consider.
  - If $u \to v$ is an edge in G_{i-1}, then $level_{i-1}(v) \leq level_{i-1}(u) + 1$, because the levels are defined by breadth-first traversal.
  - On the other hand, if $u \to v$ is not an edge in G_{i-1}, then its reversal $v \to u$ must be an edge in the ith augmenting path, which by definition is the shortest path from s to t in G_{i-1}. It follows that $level_{i-1}(v) = level_{i-1}(u) - 1 \leq level_{i-1}(u) + 1$.

  In both subcases, we conclude that $level_i(v) = level_i(u) + 1 \geq level_{i-1}(u) + 1 \geq level_{i-1}(v)$. □

Whenever we augment the flow, the bottleneck edge in the augmenting path disappears from the residual graph, and some edges in the *reversal* of the augmenting path may (re-)appear. Our second observation is that an edge cannot appear or disappear too many times.

Lemma 10.3. *During the execution of the Edmonds-Karp shortest-augmenting-path algorithm, each edge $u \to v$ disappears from the residual graph G_f at most $V/2$ times.*

Proof: Suppose $u \to v$ is in two residual graphs G_i and G_{j+1}, but not in any of the intermediate residual graphs $G_{i+1}, \ldots, G_j$, for some $i < j$. Then $u \to v$ must be in the ith augmenting path, so $level_i(v) = level_i(u) + 1$, and $v \to u$ must be on the jth augmenting path, so $level_j(v) = level_j(u) - 1$. The previous lemma implies that

$$level_j(u) \;=\; level_j(v) + 1 \;\geq\; level_i(v) + 1 \;=\; level_i(u) + 2.$$

In other words, between the disappearance and reappearance of $u \to v$, the distance from s to u increased by at least 2. Because every level is either less than V or infinite, the number of disappearances is at most $V/2$. □

Now we can derive an upper bound on the number of iterations. Because each edge disappears at most $V/2$ times, there are at most $EV/2$ edge disappearances overall. But at least one edge disappears on each iteration, so the algorithm must halt after at most $EV/2$ iterations. Finally, each iteration requires $O(E)$ time, so the overall algorithm runs in $\boldsymbol{O(VE^2)}$ ***time***.

10.7 Further Progress

This is nowhere near the end of the story for maximum-flow algorithms. Decades of further research have led to several faster algorithms, some of which are summarized in Figure 10.10.[4] All the listed algorithms listed compute a maximum flow in several iterations. Most of these algorithms have two variants: a simpler version that performs each iteration by brute force, and a faster variant that uses sophisticated data structures to maintain a spanning tree of the flow network, so that each iteration can be performed (and the spanning tree updated) in logarithmic time. There is no reason to believe that the best algorithms known so far are optimal; indeed, maximum flows are still a very active area of research.

Technique	Direct	With dynamic trees	Source(s)
Blocking flow	$O(V^2E)$	$O(VE \log V)$	[Dinitz; Karzanov; Even and Itai; Sleator and Tarjan]
Network simplex	$O(V^2E)$	$O(VE \log V)$	[Dantzig; Goldfarb and Hao; Goldberg, Grigoriadis, and Tarjan]
Push-relabel (generic)	$O(V^2E)$	—	[Goldberg and Tarjan]
Push-relabel (FIFO)	$O(V^3)$	$O(VE \log(V^2/E))$	[Goldberg and Tarjan]
Push-relabel (highest label)	$O(V^2\sqrt{E})$	—	[Cheriyan and Maheshwari; Tunçel]
Push-relabel-add games	—	$O(VE \log_{E/(V \log V)} V)$	[Cheriyan and Hagerup; King, Rao, and Tarjan]
Pseudoflow	$O(V^2E)$	$O(VE \log V)$	[Hochbaum]
Pseudoflow (highest label)	$O(V^3)$	$O(VE \log(V^2/E))$	[Hochbaum and Orlin]
Incremental BFS	$O(V^2E)$	$O(VE \log(V^2/E))$	[Goldberg, Held, Kaplan, Tarjan, and Werneck]
Compact networks	—	$O(VE)$	[Orlin]

Figure 10.10. Several purely combinatorial maximum-flow algorithms and their running times.

The fastest known (purely combinatorial) maximum-flow algorithm, announced by James Orlin in 2012, runs in $\boldsymbol{O(VE)}$ ***time***, exactly matching the worst-case complexity of a flow decomposition. The details of Orlin's algorithm are far beyond the scope of this book; in addition to his own new techniques, Orlin uses several older algorithms and data structures as black boxes, most of which are themselves quite complicated. In particular, Orlin's algorithm does *not* construct an explicit flow decomposition; in fact, for graphs with only $O(V)$ edges, an extension of his algorithm actually runs in only $O(V^2/\log V)$ time! Nevertheless, for purposes of analyzing algorithms that *use* maximum flows,

[4]To keep this table short, I have deliberately omitted algorithms whose running time depends on edge capacities or the maximum flow value. Even with this restriction, the list is embarrassingly incomplete!

this is the time bound you should cite. So write the following sentence on your cheat sheets and cite it in your homeworks:

> ***Maximum flows can be computed in $O(VE)$ time.***

Finally, faster maximum-flow algorithms are known for *unit-capacity* networks, where every edge has capacity 1. In 1973, Alexander Karzanov proved that Dinitz's blocking-flow algorithm—the first algorithm listed in the table above—runs in $O(\min\{V^{2/3}, E^{1/2}\}E)$ time in this setting. (This time bound appears to break the $\Omega(VE)$ flow decomposition barrier, but in fact Karzanov's analysis implies that any flow in a unit-capacity network can be decomposed into paths with total complexity $O(\min\{V^{2/3}, E^{1/2}\}E)$.) This was the fastest algorithm known in this setting for four decades. Karzanov's record was finally broken in 2013, when Aleksander Mądry announced a truly remarkable algorithm that computes maximum flows in unit-capacity networks in $O(E^{10/7} \operatorname{polylog} E)$ time. Again, the details of Mądry's algorithm are far beyond the scope of this book, or indeed the expertise of its author.

Exercises

0. Suppose you are given a directed graph $G = (V, E)$, two vertices s and t, a capacity function $c : E \to \mathbb{R}^+$, and a second function $f : E \to \mathbb{R}$. Describe an algorithm to determine whether f is a maximum (s, t)-flow in G.

1. Let f and f' be two feasible (s, t)-flows in a flow network G, such that $|f'| > |f|$. Prove that there is a feasible (s, t)-flow with value $|f'| - |f|$ in the residual network G_f.

2. Let $u \to v$ be an arbitrary edge in an arbitrary flow network G. Prove that if there is a minimum (s, t)-cut (S, T) such that $u \in S$ and $v \in T$, then there is *no* minimum cut (S', T') such that $u \in T'$ and $v \in S'$.

3. Let (S, T) and (S', T') be minimum (s, t)-cuts in some flow network G. Prove that $(S \cap S', T \cup T')$ and $(S \cup S', T \cap T')$ are also minimum (s, t)-cuts in G.

♠4. At the beginning of the chapter, we assumed that flow networks never contain both an edge $u \to v$ and its reversal $v \to u$, and we enforced this assumption when necessary by subdividing edges. This subdivision simplifies the presentation of the algorithms, but increases both the space usage and the running time of our algorithms by a constant factor. There are a few natural approaches that avoid this constant-factor blowup.

- **Just do it:** If the input graph contains both an edge $u \to v$ and its reversal $v \to u$, treat them as independent edges, with independent flow values.
- **Unidirectional flow:** If the input graph contains both an edge $u \to v$ and its reversal $v \to u$, then for every flow f, require either $f(u \to v) = 0$ or $f(v \to u) = 0$.
- **Antisymmetric flow:** If the input graph is *missing* the reversal $v \to u$ of any edge $u \to v$, add the reversed edge $v \to u$ and assign it capacity 0. Then *require* every flow to satisfy the constraint $f(u \to v) = -f(v \to u)$ for every edge $u \to v$, and define a flow to be *feasible* if $f(u \to v) \le c(u \to v)$ for every edge $u \to v$. In particular, flow values may (and sometimes must) be negative.
- **Flow intervals:** Assign each edge $u \to v$ in the input graph a *lower bound* as follows:

$$\ell(u \to v) := \begin{cases} -c(v \to u) & \text{if } v \to u \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$$

  Then if the input graph contains both both an edge $u \to v$ and its reversal $v \to u$, delete one of those two edges. (It doesn't matter which one.) Define a flow to be *feasible* if $\ell(u \to v) \le f(u \to v) \le c(u \to v)$ for every edge $u \to v$. In particular, flow values may (and sometimes must) be negative.

Work out the remaining implementation details for each of these formulations. What is the correct formulation of the conservation constraint? How *exactly* should we define the value of a flow, or the capacity of a cut? How *exactly* should we define the residual graph of a flow? How *exactly* do we find augmenting paths?

5. (a) Describe an efficient algorithm to determine whether a given flow network contains a *unique* maximum (s, t)-flow.

   (b) Describe an efficient algorithm to determine whether a given flow network contains a *unique* minimum (s, t)-cut.

   (c) Describe a flow network that contains a unique maximum (s, t)-flow but does not contain a unique minimum (s, t)-cut.

   (d) Describe a flow network that contains a unique minimum (s, t)-cut but does not contain a unique maximum (s, t)-flow.

6. An (s, t)-flow in a network G is *acyclic* if there are no directed cycles where every edge has a positive flow value; that is, the subgraph of edges with positive flow value is a dag.

(a) Describe and analyze an algorithm to compute an *acyclic* maximum (s, t)-flow in a given flow network. Your algorithm should have the same asymptotic running time as Ford-Fulkerson.

(b) Describe and analyze an algorithm to determine whether *every* maximum (s, t)-flow in a given flow network is acyclic.

7. Let $G = (V, E)$ be a flow network in which every edge has capacity 1 and the shortest-path distance from s to t is at least d.

(a) Prove that the value of the maximum (s, t)-flow is at most E/d.

(b) Now suppose that G is *simple*, meaning that for all vertices u and v, there is at most one edge from u to v. (Flow networks can have parallel edges.) Prove that the value of the maximum (s, t)-flow is at most $O(V^2/d^2)$. *[Hint: How many nodes are in the average level of a BFS tree rooted at s?]*

8. Suppose we are given a flow network $G = (V, E)$ in which every edge has capacity 1, together with an integer k. Describe and analyze an algorithm to identify k edges in G such that after deleting those k edges, the value of the maximum (s, t)-flow in the remaining graph is as small as possible.

9. The analysis in our proof of the Flow Decomposition Theorem can be tightened. Let $G = (V, E)$ be an arbitrary flow network, and let f be an arbitrary (s, t)-flow in G.

(a) Prove that if $|f| = 0$, then f is the weighted sum of at most $E - V + 1$ directed cycles, where $f(e) > 0$ for every edge e in each of these cycles.

(b) Prove that if $|f| > 0$, then f is the weighted sum of at most $E - V + 2$ directed paths and directed cycles, where $f(e) > 0$ for every edge e in each of these paths and cycles.

(c) Prove that both of the previous upper bounds are tight: There are graphs in which some circulations cannot be decomposed into less than $E - V + 1$ cycles, and some flows cannot be decomposed into less than $E - V + 2$ paths and cycles. *[Hint: This is easy.]*

♣10. Our observation that any linear combination of (s, t)-flows is itself an (s, t)-flow implies that the set of all (not necessarily feasible) (s, t)-flows in any graph actually define a real *vector space*, which we can call the ***flow space*** of the graph.

(a) Prove that the flow space of any connected graph $G = (V, E)$ has dimension $E - V + 2$.

(b) Let T be any spanning tree of G. Prove that the following collection of paths and cycles define a basis for the flow space:
- The unique path in T from s to t;
- The unique cycle in $T \cup \{e\}$, for every edge $e \notin T$.

(c) Let T be any spanning tree of G, and let F be the forest obtained by deleting any single edge in T. Prove that the following collection of paths and cycles define a basis for the flow space:
- The unique path in $F \cup \{e\}$ from s to t, for every edge $e \notin F$ that has one endpoint in each component of F;
- The unique cycle in $F \cup \{e\}$, for every edge $e \notin F$ with both endpoints in the same component of F.

(d) Prove or disprove the following claim: Every connected flow network has a flow basis that consists entirely of simple paths from s to t.

11. Cuts are sometimes defined as subsets of the edges of the graph, instead of as partitions of its vertices. In this problem, you will prove that these two definitions are *almost* equivalent.

We say that a subset X of (directed) edges *separates s and t* if every directed path from s to t contains at least one (directed) edge in X. For any subset S of *vertices*, let $\boldsymbol{\delta S}$ denote the set of directed edges leaving S; that is, $\delta S := \{u \to v \mid u \in S, v \notin S\}$.

(a) Prove that if (S, T) is an (s, t)-cut, then δS separates s and t.

(b) Let X be an arbitrary subset of edges that separates s and t. Prove that there is an (s, t)-cut (S, T) such that $\delta S \subseteq X$.

(c) Let X be a *minimal* subset of edges that separates s and t. (Such a set of edges is sometimes called a ***bond***.) Prove that there is an (s, t)-cut (S, T) such that $\delta S = X$.

12. Suppose instead of capacities, we consider networks where each edge $u \to v$ has a non-negative ***demand*** $d(u \to v)$. Now an (s, t)-flow f is *feasible* if and only if $f(u \to v) \geq d(u \to v)$ for every edge $u \to v$. (Feasible flow values can now be arbitrarily large.) A natural problem in this setting is to find a feasible (s, t)-flow of *minimum* value.

(a) Describe an efficient algorithm to compute a feasible (s, t)-flow, given the graph, the demand function, and the vertices s and t as input. *[Hint: Find a flow that is non-zero everywhere, and then scale it up to make it feasible.]*

(b) Suppose you have access to a subroutine MaxFlow that computes *maximum* flows in networks with edge capacities. Describe an efficient

algorithm to compute a *minimum* flow in a given network with edge demands; your algorithm should call MAXFLOW exactly once.

(c) State and prove an analogue of the max-flow min-cut theorem for this setting. (Do minimum flows correspond to maximum cuts?)

13. For any flow network G and any vertices u and v, let $bottleneck_G(u, v)$ denote the maximum, over all paths π in G from u to v, of the minimum-capacity edge along π.

(a) Describe and analyze an algorithm to compute $bottleneck_G(s, t)$ in $O(E \log V)$ time. This is the amount of flow that the Edmonds-Karp fattest-augmenting-paths algorithm pushes in the first iteration.

(b) Now suppose the flow network G is undirected; equivalently, suppose $c(u{\to}v) = c(v{\to}u)$ for every pair of vertices u and v. Describe and analyze an algorithm to compute $bottleneck_G(s, t)$ in $O(V + E)$ time. *[Hint: Find the median edge capacity.]* Why doesn't this speedup work for directed graphs?

♥(c) Again, suppose the flow network G is undirected. Describe and analyze an algorithm to construct a spanning tree T of G such that $bottleneck_T(u, v) = bottleneck_G(u, v)$ for *all* vertices u and v. (Edges in T inherit their capacities from G.) For full credit, your algorithm should run in $O(E)$ time.

14. Suppose you are given a flow network G with ***integer*** edge capacities and an ***integer*** maximum flow f^* in G. Describe algorithms for the following operations:

(a) INCREMENT(e): Increase the capacity of edge e by 1 and update the maximum flow.

(b) DECREMENT(e): Decrease the capacity of edge e by 1 and update the maximum flow.

Both algorithms should modify f^* so that it is still a maximum flow, more quickly than recomputing a maximum flow from scratch.

15. Let G be a network with integer edge capacities. An edge in G is *upper-binding* if increasing its capacity by 1 also increases the value of the maximum flow in G. Similarly, an edge is *lower-binding* if decreasing its capacity by 1 also decreases the value of the maximum flow in G.

(a) Does every network G have at least one upper-binding edge? Prove your answer is correct.

(b) Does every network G have at least one lower-binding edge? Prove your answer is correct.

(c) Describe an algorithm to find all upper-binding edges in G, given both G and a maximum flow in G as input, in $O(E)$ time.

(d) Describe an algorithm to find all lower-binding edges in G, given both G and a maximum flow in G as input, in $O(EV)$ time.

16. A given flow network G may have more than one minimum (s, t)-cut. Let's define the ***best*** minimum (s, t)-cut to be any minimum cut (S, T) with the smallest number of edges crossing from S to T.

(a) Describe an efficient algorithm to find the best minimum (s, t)-cut when the capacities are integers.

(b) Describe an efficient algorithm to find the best minimum (s, t)-cut for *arbitrary* edge capacities.

(c) Describe an efficient algorithm to determine whether a given flow network contains a unique *best* minimum (s, t)-cut.

17. A new assistant professor, teaching maximum flows for the first time, suggests the following greedy modification to the generic Ford-Fulkerson augmenting path algorithm. Instead of maintaining a residual graph, just[5] reduce the capacity of edges along the augmenting path! In particular, whenever we saturate an edge, just remove it from the graph. Who needs all that residual graph nonsense?

```
GreedyFlow(G, c, s, t):
  for every edge e in G
      f(e) ← 0

  while there is a path from s to t
      π ← an arbitrary path from s to t
      F ← minimum capacity of any edge in π
      for every edge e in π
          f(e) ← f(e) + F
          if c(e) = F
              remove e from G
          else
              c(e) ← c(e) − F

  return f
```

(a) Show that GreedyFlow does not always compute a maximum flow.

(b) Show that GreedyFlow is not even guaranteed to compute a good approximation to the maximum flow. That is, for any constant $\alpha > 1$, there is a flow network G such that the value of the maximum flow is

[5]The adverb *just* is almost always subconscious shorthand for "I'm too lazy to figure out the details, but you should believe me anyway", or more succinctly, "This is probably wrong." See also *merely*, *simply*, *clearly*, and *obviously*.

more than α times the value of the flow computed by GREEDYFLOW. *[Hint: Assume that GREEDYFLOW chooses the worst possible path π at each iteration.]*

18. In 1980 Maurice Queyranne published an example of a flow network, shown below, where Edmonds and Karp's "fattest path" heuristic does not halt. As in Zwick's bad example for the original Ford-Fulkerson algorithm, ϕ denotes the inverse golden ratio $(\sqrt{5}-1)/2$. The three vertical edges play essentially the same role as the horizontal edges in Zwick's example.

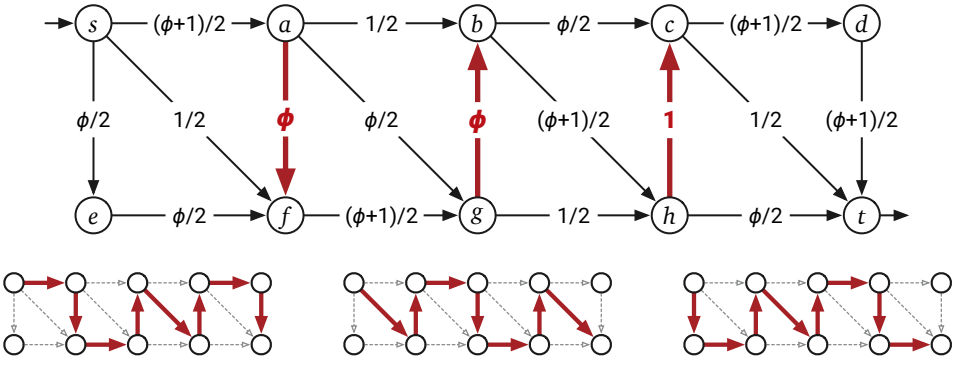

Figure 10.11. Queyranne's network, and a sequence of "fattest path" augmentations.

(a) Show that the following infinite sequence of path augmentations is a valid execution of the Edmonds-Karp "fattest path" algorithm. (See the bottom of Figure 10.11.)

```
QUEYRANNEFATPATHS:
  for i ← 1 to ∞
      push ϕ^(3i−2) units of flow along  s→a→f→g→b→h→c→d→t
      push ϕ^(3i−1) units of flow along  s→f→a→b→g→h→c→t
      push ϕ^(3i)   units of flow along  s→e→f→a→g→b→c→h→t
```

(b) Describe a sequence of $O(1)$ path augmentations that yields a maximum flow in Queyranne's network.

♥19. An ***(s, t)-series-parallel*** graph is a directed acyclic graph with two distinguished vertices s and t and with one of the following structures:

- **Base case:** A single directed edge from s to t.
- **Series:** The union of an (s,u)-series-parallel graph and a (u,t)-series-parallel graph that share a common vertex u but no other vertices or edges.
- **Parallel:** The union of two smaller (s,t)-series-parallel graphs with the same source s and target t, but with no other vertices or edges in common.

Every (s, t)-series-parallel graph G can be represented by a ***decomposition tree***, which is a binary tree with three types of nodes: leaves (which corresponding to edges in G), series nodes (which correspond to vertices other than s and t), and parallel nodes. The same series-parallel graph could be represented by many different decomposition trees.

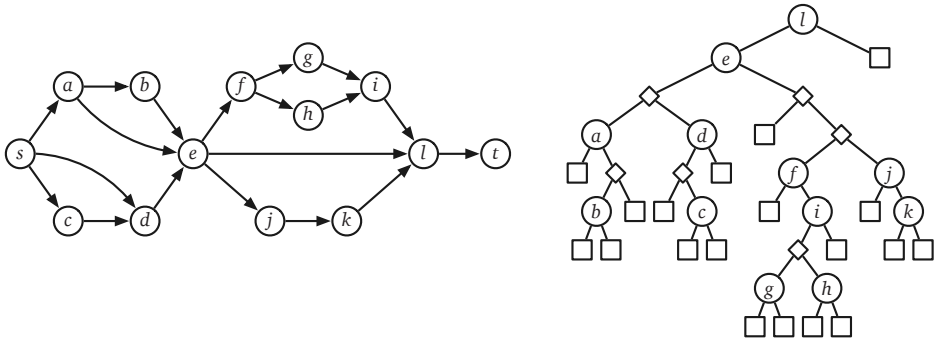

Figure 10.12. A series-parallel graph and a corresponding decomposition tree. Squares in the decomposition tree are leaves; diamonds are parallel nodes.

(a) Suppose you are given a directed graph G with two special vertices s and t. Describe and analyze an algorithm that either builds a decomposition tree for G or correctly reports that G is not (s, t)-series-parallel. *[Hint: Build the tree from the bottom up.]*

(b) Describe and analyze an algorithm to compute a maximum (s, t)-flow in a given (s, t)-series-parallel flow network with arbitrary edge capacities. *[Hint: In light of part (a), you can assume that you are actually given the decomposition tree. First compute the maximum-flow value, then compute an actual maximum flow.]*

20. We can speed up the Edmonds-Karp "fattest path" algorithm, at least for networks with small integer capacities, by relaxing our requirements for the next augmenting path. Instead of finding the augmenting path with maximum bottleneck capacity, we find a path whose bottleneck capacity is at least half of maximum, using the following ***capacity scaling*** algorithm. (This algorithm was actually proposed by Edmonds and Karp.)

Assume all the edge capacities are positive integers less than $U = 2^k$ for some integer k. The scaling algorithm maintains a bottleneck threshold Δ; initially, we set $\Delta \leftarrow U$. In each *phase*, the algorithm augments along paths from s to t in which every edge has residual capacity at least Δ. When there is no such path, the phase ends, we set $\Delta \leftarrow \lfloor \Delta/2 \rfloor$, and the next phase begins. The algorithm ends when $\Delta = 0$.

(a) How many phases will this algorithm execute in the worst case?

(b) Let f be the flow at the end of a phase for a particular value of Δ. Prove that the capacity of a minimum cut in the residual graph G_f is at most $E \cdot \Delta$.

(c) Prove that in each phase of the scaling algorithm, there are at most $2E$ augmentations.

(d) What is the overall running time of the capacity scaling algorithm?