Ford and Fulkerson's algorithm works quite well in many practical situations, or in settings where the maximum flow value $|f^*|$ is small, but without further constraints on the augmenting paths, this is *not* an efficient algorithm in worst case. The example network above 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 $X-3$.)

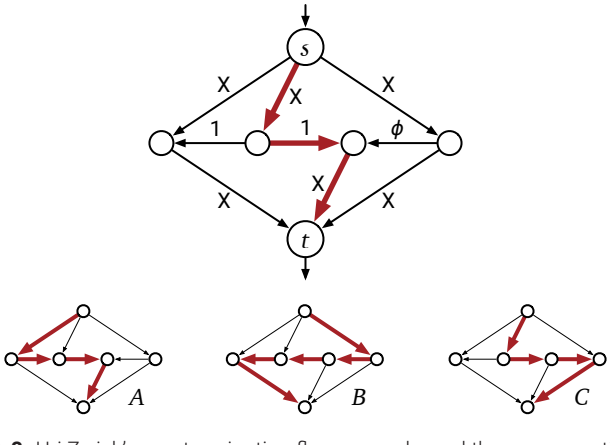


**Figure 10.8.** Uri Zwick's non-terminating flow example, and three augmenting paths.

[2]Ford and Fulkerson described a network with the same bad behavior in 1962, which had 10 vertices and 48 edges.

Suppose the Ford-Fulkerson algorithm starts by choosing the central augmenting path, shown at the top of Figure 10.8. The three horizontal edges, in order from left to right, now have residual capacities 1, 0, and $\phi$. Suppose inductively that the horizontal residual capacities are $\phi^{k-1}$, 0, and $\phi^k$ for some non-negative integer $k$.

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

It follows by induction that after $4n+1$ augmentation steps, the horizontal edges have residual capacities $\phi^{2n-2}, 0, \phi^{2n-1}$. 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 $\alpha$ and $\beta$, 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 very 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

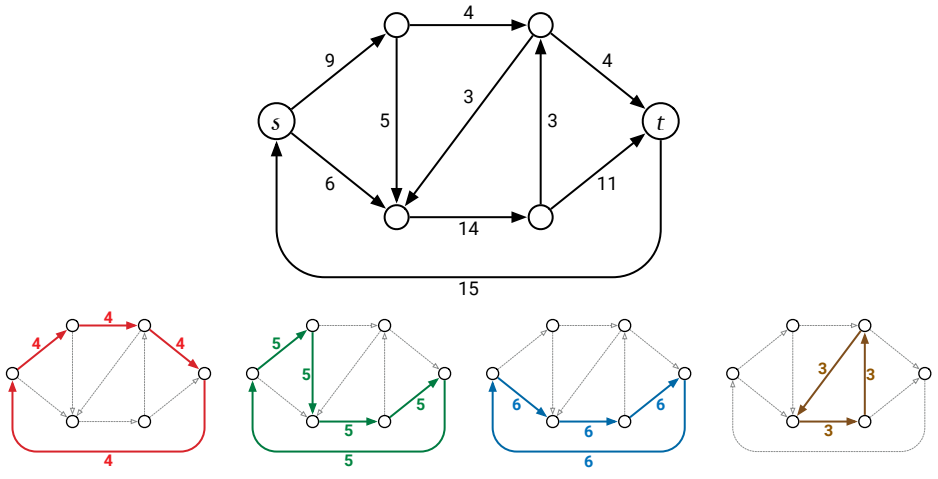


**Figure 10.9.** Decomposing a circulation into weighted directed cycles.

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 an arbitrary circulation $f$ in an arbitrary flow network, and let $\#f$ denote the number of edges $u \to v$ such that $f(u \to v) > 0$. I claim that $f$ can be decomposed into a positive linear combination of at most $\max\{0, \#f - 1\}$ cycles. 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 $j < k$ be the smallest indices such that $v_j = v_k$. 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 a feasible circulation in $G$ with value 0. 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 induction hypothesis implies that $f'$ has a valid decomposition into $\#f' - 1 \leq \#f - 2$ 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. 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 $u \to v$. 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 $\#f' - 1 \leq \#f$ 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. Because each iteration removes at least one edge from the flow graph, 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 analyzed two natural rules for choosing augmenting paths, both of which led to more efficient algorithms.

### Fattest Augmenting Paths

Edmonds and Karp's first rule is essentially a 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 by Ford and Fulkerson in their original maximum-flow paper; a variant of this rule was independently considered by the Russian mathematician Yefim Dinitz[3] around the same time as Edmonds and Karp.

> 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 $i > 0$.*

[3]Dinitz actually discovered an even faster maximum-flow algorithm that runs in $O(V^2E)$ time, while a student in an algorithms class taught by Georgy Adelson-Velsky (the "AV" in AVL trees), in response to an in-class exercise.

**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.

- 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 $i$th 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 $i$th augmenting path, so $level_i(v) = level_i(u) + 1$, and $v \to u$ must be on the $j$th 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

[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!

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, 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 more than 40 years. Karzanov's record was finally broken in 2014, 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. Let $G$ be a flow network that contains an opposing pair of edges $u \to v$ and $v \to u$, both with positive capacity. Let $G'$ be the flow network obtained from $G$ by decreasing the capacities of both of these edges by $\min\{c(u \to v), c(v \to u)\}$. In other words:
   - If $c(u \to v) > c(v \to u)$, change the capacity of $u \to v$ to $c(u \to v) - c(v \to u)$ and delete $v \to u$.
   - If $c(u \to v) < c(v \to u)$, change the capacity of $v \to u$ to $c(v \to u) - c(u \to v)$ and delete $u \to v$.
   - Finally, if $c(u \to v) = c(v \to u)$, delete both $u \to v$ and $v \to u$.

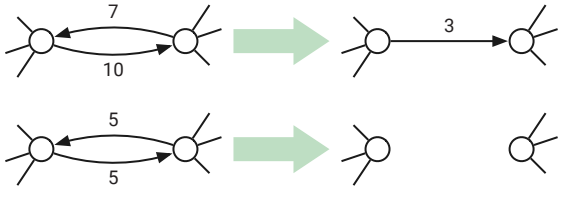


**Figure 10.11.** Enforcing the one-direction assumption.

   (a) Prove that every maximum $(s, t)$-flow in $G'$ is also a maximum flow in $G$. (Thus, by reducing *every* opposing pair of edges in $G$, we obtain a new flow network without opposing edges, but with the same maximum flow value as $G$.)

   (b) Prove that every minimum $(s, t)$-cut in $G$ is also a minimum $(s, t)$-cut in $G'$ *and vice versa*.

   (c) Prove that there is at least one maximum $(s, t)$-flow in $G$ that is ***not*** a maximum $(s, t)$-flow in $G'$.

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—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 $\delta S$ 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 $\pi$ in $G$ from $u$ to $v$, of the minimum-capacity edge along $\pi$.

(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 $O(E)$ 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] More often than not, the adverb *just* is subconscious shorthand for "I'm too lazy to figure out the details, so this is almost certainly wrong, but you should believe me anyway." See also *merely*, *simply*, *clearly*, and *obviously*.