The tree which fills the arms grew from the tiniest sprout;
the tower of nine storeys rose from a (small) heap of earth;
the journey of a thousand li commenced with a single step.

— Lao-Tzu, *Tao Te Ching*, chapter 64 (6th century BC),
translated by James Legge (1891)

And I would walk five hundred miles,
And I would walk five hundred more,
Just to be the man who walks a thousand miles
To fall down at your door.

— The Proclaimers, "I'm Gonna Be (500 Miles)",
Sunshine on Leith (2001)

Almost there… Almost there…

— Red Leader [Drewe Henley], *Star Wars* (1977)

9

All-Pairs Shortest Paths

9.1 Introduction

In the previous chapter, we discussed several algorithms to find the shortest paths from a single source vertex s to every other vertex of the graph, by constructing a shortest path tree rooted at s. The shortest path tree specifies two pieces of information for each node v in the graph:

- $dist(v)$ is the length of the shortest path from s to v;
- $pred(v)$ is the second-to-last vertex in the shortest path from s to v.

In this chapter, we consider the more general *all pairs shortest path* problem, which asks for the shortest path from *every* possible source to every possible destination. For every pair of vertices u and v, we want to compute the following information:

- $dist(u, v)$ is the length of the shortest path from u to v;
- $pred(u, v)$ is the second-to-last vertex on the shortest path from u to v.

These intuitive definitions exclude a few boundary cases, all of which we already saw in the previous chapter.

- If there is no path from u to v, then there is no *shortest* path from u to v; in this case, we define $dist(u, v) = \infty$ and $pred(u, v) = \text{NULL}$.
- If there is a negative cycle between u and v, then there are paths[1] from u to v with arbitrarily negative length; in this case, we define $dist(u, v) = -\infty$ and $pred(u, v) = \text{NULL}$.
- Finally, if u does not lie on a negative cycle, then the shortest path from u to itself has no edges, and therefore doesn't have a last edge; in this case, we define $dist(u, u) = 0$ and $pred(u, u) = \text{NULL}$

The desired output of the all-pairs shortest path problem is a pair of $V \times V$ arrays, one storing all V^2 shortest-path distances,[2] the other storing all V^2 predecessors. In this chapter, I'll focus almost exclusively on computing the distance array. The predecessor array, from which we can compute the actual shortest paths, can be computed with only minor modifications (hint, hint).

9.2 Lots of Single Sources

The obvious solution to the all-pairs shortest path problem is to run a single-source shortest path algorithm V times, once for each possible source vertex. Specifically, to fill in the one-dimensional subarray $dist[s, \cdot]$, we invoke a single-source algorithm starting at the source vertex s.

```
OBVIOUSAPSP(V, E, w):
  for every vertex s
    dist[s, ·] ← SSSP(V, E, w, s)
```

The running time of this algorithm obviously depends on which single-source shortest path algorithm we use. Just as in the single-source setting, there are four natural options, depending on the structure of the graph and its edge weights:

- If the edges of the graph are unweighted, breadth-first search gives us an overall running time of $O(VE)$.
- If the graph is acyclic, scanning the vertices in topological order gives us an overall running time of $O(VE)$.

[1] formally, walks

[2] Back when road maps used to be printed on paper and had to be searched manually, it was fairly common for them to include a triangular "distance table". To find the distance from Champaign to Columbus, for example, you would look in the row labeled "Champaign" and the column labeled "Columbus".

- If all edge weights are non-negative, Dijkstra's algorithm gives us a running time to $\boldsymbol{O(VE \log V)} = O(V^3 \log V)$.[3]
- Finally, in the most general setting, the Bellman-Ford algorithm gives us an overall running time of $\boldsymbol{O(V^2E)} = O(V^4)$.

9.3 Reweighting

Negative edges slow us down; can we get rid of them? One simple idea that occurs to many people is increasing the weights of all the edges by the same amount so that all the weights become positive, so that we can use Dijkstra's algorithm instead of Bellman-Ford. Unfortunately, this simple idea doesn't work, intuitively because our two natural notions of "length" are incompatible—paths with more edges can have smaller total weight than paths with fewer edges. If we increase all edge weights at the same rate, paths with more edges get longer faster than paths with fewer edges; as a result, the shortest path between two vertices might change.

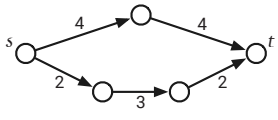

Figure 9.1. Increasing all the edge weights by 2 changes the shortest path from s to t.

However, there is a more subtle method for reweighting edges that does preserve shortest paths. This reweighting method is often attributed to Donald Johnson, who described its application to shortest path algorithms in 1973. In fact, Johnson attributed the method to a 1972 paper of Jack Edmonds and Richard Karp; the same method was also described in a slightly different form by Delbert Fulkerson in 1961.

Suppose each vertex v has some associated *price* $\pi(v)$, which might be positive, negative, or zero. We can define a new weight function w' as follows:

$$w'(u \to v) = \pi(u) + w(u \to v) - \pi(v)$$

To give some intuition, imagine that when we leave vertex u, we have to pay an exit tax of $\pi(u)$, and when we enter v, we get $\pi(v)$ as an entrance gift.

It's not hard to show that shortest paths with the new weight function w' are exactly the same as shortest paths with the original weight function w. In

[3]Again, if we replace the binary heap in our implementation of Dijkstra's algorithm with an unsorted array, the overall running time becomes $O(V^3)$ (no matter how many edges the graph has), and if we replace the binary heap with a Fibonacci heap, the running time drops to $O(V(E + V \log V)) = O(VE + V^2 \log V) = O(V^3)$.

fact, for *any* path $u \rightsquigarrow v$ from one vertex u to another vertex v, we have

$$w'(u \rightsquigarrow v) = \pi(u) + w(u \rightsquigarrow v) - \pi(v).$$

We pay $\pi(u)$ in exit fees, plus the original weight of of the path, minus the $\pi(v)$ entrance gift. At every intermediate vertex x on the path, we get $\pi(x)$ as an entrance gift, but then immediately pay it back as an exit tax! Since all paths from u to v change length by exactly the same amount, the shortest path from u to v does not change. (Paths between different pairs of vertices could change lengths by different amounts, so their order could change.)

9.4 Johnson's Algorithm

Johnson's all-pairs shortest path algorithm computes a cost $\pi(v)$ for each vertex, so that the new weight of every edge is non-negative, and then computes shortest paths with respect to the new weights using Dijkstra's algorithm.

First, suppose the input graph has a vertex s that can reach *all* the other vertices. Johnson's algorithm computes the shortest paths from s to the other vertices, using Bellman-Ford (which doesn't care if the edge weights are negative), and then reweights the graph using the price function $\pi(v) = dist(s, v)$. The new weight of every edge is

$$w'(u \to v) = dist(s, u) + w(u \to v) - dist(s, v).$$

These new weights non-negative *because* Bellman-Ford halted! Recall that an edge $u \to v$ is *tense* if $dist(s, u) + w(u \to v) < dist(s, v)$, and that single-source shortest path algorithms eliminate all tense edges. (If Bellman-Ford detects a negative cycle, Johnson's algorithm aborts, because shortest paths are not well-defined.)

If there is no suitable vertex s that can reach everything, then no matter where we start Bellman-Ford, some of the resulting vertex prices will be infinite. To avoid this issue, we *always* add a new vertex s to the graph, with zero-weight edges from s to the other vertices, but *no* edges going back into s. This addition doesn't change the shortest paths between any pair of original vertices, because there are no paths into s.

Complete pseudocode for Johnson's algorithm is shown in Figure 9.2. The running time of this algorithm is dominated by the calls to Dijkstra's algorithm. Specifically, we spend $O(VE)$ time running BellmanFord once, $O(VE \log V)$ time running Dijkstra V times, and $O(V + E)$ time doing other bookkeeping. Thus, the overall running time is $\boldsymbol{O(VE \log V)} = O(V^3 \log V)$.[4] Negative edges don't slow us down after all!

[4] . . . assuming the default binary-heap implementation; see the previous footnote.

```
JohnsonAPSP(V, E, w) :
  ⟨⟨Add an artificial source⟩⟩
  add a new vertex s
  for every vertex v
      add a new edge s→v
      w(s→v) ← 0
  ⟨⟨Compute vertices prices⟩⟩
  dist[s, ·] ← BellmanFord(V, E, w, s)
  if BellmanFord found a negative cycle
      fail gracefully
  ⟨⟨Reweight edges⟩⟩
  for every edge (u, v) ∈ E
      w'(u→v) ← dist[s, u] + w(u→v) − dist[s, v]
  ⟨⟨Compute reweighted shortest paths⟩⟩
  for every vertex u
      dist'[u, ·] ← Dijkstra(V, E, w', u)
  ⟨⟨Compute original shortest-path distances⟩⟩
  for every vertex u
      for every vertex v
          dist[u, v] ← dist'[u, v] − dist[s, u] + dist[s, v]
```

Figure 9.2. Johnson's all-pairs shortest paths algorithm

9.5 Dynamic Programming

We can also solve the all-pairs shortest path problem using dynamic programming, instead of invoking a single-source algorithm. For *dense* graphs, where $E = \Omega(V^2)$, the dynamic programming approach eventually yields an algorithm that is both simpler and (slightly) faster than Johnson's algorithm. **For the rest of this chapter, I will assume that the input graph contains no negative cycles.**

As usual for dynamic programming algorithms, we first need a recurrence. Just as in the single-source setting, the "obvious" recursive definition

$$dist(u, v) = \begin{cases} 0 & \text{if } u = v \\ \min\limits_{x \to v} \big(dist(u, x) + w(x \to v) \big) & \text{otherwise} \end{cases}$$

only works when the input graph is a dag; any directed cycles drive the recurrence into an infinite loop.

We can break this infinite loop by introducing as an additional parameter, exactly as we did for Bellman-Ford; let $dist(u, v, \ell)$ denote the length of the shortest path from u to v that uses *at most* ℓ *edges*. The shortest path between any two vertices traverses at most $V - 1$ edges, so the true shortest-path distance is $dist(u, v, V - 1)$. Bellman's single-source recurrence adapts to this setting

immediately:

$$dist(u, v, \ell) = \begin{cases} 0 & \text{if } \ell = 0 \text{ and } u = v \\ \infty & \text{if } \ell = 0 \text{ and } u \neq v \\ \min \left\{ \begin{array}{c} dist(u, v, \ell - 1) \\ \min\limits_{x \to v} \left(dist(u, x, \ell - 1) + w(x \to v) \right) \end{array} \right\} & \text{otherwise} \end{cases}$$

Turning this recurrence into a dynamic programming algorithm is straightforward; the resulting algorithm runs in $\boldsymbol{O(V^2E)} = O(V^4)$ time.

```
SHIMBELAPSP(V, E, w):
  for all vertices u
      for all vertices v
          if u = v
              dist[u, v, 0] ← 0
          else
              dist[u, v, 0] ← ∞

  for ℓ ← 1 to V − 1
      for all vertices u
          for all vertices v ≠ u
              dist[u, v, ℓ] ← dist[u, v, ℓ − 1]
              for all edges x→v
                  if dist[u, v, ℓ] > dist[u, x, ℓ − 1] + w(x→v)
                      dist[u, v, ℓ] ← dist[u, x, ℓ − 1] + w(x→v)
```

This algorithm was first sketched by Alfonso Shimbel in 1954.[5] Just like Bellman's formulation of Bellman-Ford, we don't need the inner loop over vertices v or the iteration index ℓ. The modified algorithm is shown below.

```
ALLPAIRSBELLMANFORD(V, E, w):
  for all vertices u
      for all vertices v
          if u = v
              dist[u, v] ← 0
          else
              dist[u, v] ← ∞

  for ℓ ← 1 to V − 1
      for all vertices u
          for all edges x→v
              if dist[u, v] > dist[u, x] + w(x→v)
                  dist[u, v] ← dist[u, x] + w(x→v)
```

[5]Shimbel assumed the input was a complete $V \times V$ matrix of distances, so his original algorithm actually runs in $O(V^4)$ time no matter how many edges the graph has.

Given how we derived it, it should come as no surprise that the resulting algorithm is exactly the same as interleaving V different executions of Bellman-Ford, one running at each vertex. In particular, for all vertices u and v, after the ℓth iteration of the main for-loop, $dist[u, v]$ is *at most* the length of the shortest path from u to v containing at most ℓ edges.

9.6 Divide and Conquer

But we can make a more significant improvement, suggested by Michael Fischer and Albert Meyer in 1971. Bellman's recurrence breaks the shortest path into a slightly shorter path and a single edge, by considering all possible predecessors of the target vertex. Instead, let's break the shortest paths into two shorter shortest paths at the *middle* vertex. This idea gives us a different recurrence for same the function $dist(u, v, \ell)$. Here we need to stop at the base case $\ell = 1$ instead of $\ell = 0$, because a path with at most one edge has no "middle" vertex. To simplify the recurrence slightly, let's define $w(v \to v) = 0$ for every vertex v.

$$dist(u, v, \ell) = \begin{cases} w(u \to v) & \text{if } i = 1 \\ \min_x \big(dist(u, x, \ell/2) + dist(x, v, \ell/2) \big) & \text{otherwise} \end{cases}$$

As stated, this recurrence only works when ℓ is a power of 2, since otherwise we might try to find the shortest path with (at most) a fractional number of edges! But that's not really a problem; $dist(u, v, \ell)$ is the true shortest-path distance from u to v *for all* $\ell \geq V - 1$; in particular, we can use $\ell = 2^{\lceil \lg V \rceil} < 2V$.

Once again, a dynamic programming solution is straightforward. Even before we write down the algorithm, we can tell the running time is $\boldsymbol{O(V^3 \log V)}$—we need to consider V possible values of u, v, and x, but only $\lceil \lg V \rceil$ possible values of ℓ. In the following pseudocode for Fischer and Meyer's algorithm, the array entry $dist[u, v, i]$ stores the value of $dist(u, v, 2^i)$.

```
FischerMeyerAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)

  for i ← 1 to ⌈lg V⌉              ⟨⟨ℓ = 2^i⟩⟩
      for all vertices u
          for all vertices v
              dist[u, v, i] ← ∞
              for all vertices x
                  if dist[u, v, i] > dist[u, x, i−1] + dist[x, v, i−1]
                      dist[u, v, i] ← dist[u, x, i−1] + dist[x, v, i−1]
```

Unlike our earlier algorithms, FischerMeyerAPSP is *not* the same as V invocations of any single-source shortest-path algorithm; in particular, the

innermost loop does *not* simply relax tense edges. Nevertheless, we can still remove the last dimension of the table, using $dist[u, v]$ everywhere in place of $dist[u, v, i]$, just as we did in Bellman-Ford and our earlier dynamic programming algorithm; this reduces the space from $O(V^3)$ to $O(V^2)$. This more polished algorithm was described by Leyzorek *et al.* in 1957, in the same paper where they describe Dijkstra's algorithm.

```
LeyzorekAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)

  for i ← 1 to ⌈lg V⌉                ⟨⟨ℓ = 2^i⟩⟩
      for all vertices u
          for all vertices v
              for all vertices x
                  if dist[u, v] > dist[u, x] + dist[x, v]
                      dist[u, v] ← dist[u, x] + dist[x, v]
```

9.7 Funny Matrix Multiplication

There is a very close connection (first observed by Shimbel, and later independently by Bellman) between computing shortest paths in a directed graph and computing powers of a square matrix. Compare the following algorithm for squaring an $n \times n$ matrix A with the inner loop of FischerMeyerAPSP. (I've slightly modified the notation in the second algorithm to make the similarity clearer.)

```
MatrixSquare(A):
  for i ← 1 to n
      for j ← 1 to n
          A'[i, j] ← 0
          for k ← 1 to n
              A'[i, j] ← A'[i, j] + A[i, k] · A[k, j]
```

```
FischerMeyerInnerLoop(D):
  for all vertices u
      for all vertices v
          D'[u, v] ← ∞
          for all vertices x
              D'[u, v] ← min{D'[u, v], D[u, x] + D[x, v]}
```

The *only* difference between these two algorithms is that the second algorithm uses addition instead of multiplication, and minimization instead of addition. For this reason, the shortest path inner loop is sometimes referred to as "min-plus" or "distance" or "funny" matrix multiplication.

Our slower algorithm ShimbelAPSP is the standard iterative algorithm for computing the $(V-1)$th "min-plus power" of the weight matrix w. The first set of loops sets up the min-plus identity matrix, with 0s on the main diagonal and ∞ everywhere else, and each iteration of the second main loop computes the next "min-plus power". FischerMeyerAPSP replaces this iterative method for computing powers with repeated squaring, exactly as we saw at the end of Chapter 1. Once again, we see the influence of ancient Egyptian *ἀρπεδονάπται*!

There are faster divide-and-conquer algorithms for (standard) matrix multiplication, similar to Karatsuba's divide-and-conquer algorithm for multiplying integers. The first such algorithm, described by Volker Strassen in 1969, reduces the problem of multiplying two $n \times n$ matrices to *seven* instances of multiplying two $n/2 \times n/2$ matrices; Strassen's algorithm runs in $O(n^{\lg 7}) = O(n^{2.807355})$. Strassen's algorithm has been improved many times over the last fifty years; as of 2018, the fastest matrix-multiplication algorithm known runs in $O(n^{2.372864})$ time.[6] Unfortunately, *all* of these faster algorithms use subtraction, and there's no "funny" equivalent of subtraction. (What's the inverse operation for min?) So at least for general graphs, there's no obvious way to speed up the inner loop of our dynamic programming algorithms.

But "not obvious" does not mean "impossible"! In fact, there are several significantly faster algorithms for *special cases* of the all-pairs shortest paths problem. One of the nicest is a simple randomized algorithm discovered in 1991 by Zvi Galil and Oded Margalit, and further simplified in 1992 by Raimund Seidel, that computes all-pairs shortest path *distances* in *unweighted, undirected* graphs in $O(M(V) \log V)$ *expected* time, where $M(n) = O(n^{2.372864})$ is the time required to (seriously) multiply two $n \times n$ integer matrices.[7] Galil, Margalit, and Seidel's approach has since been extended to compute actual shortest paths, deterministically, in directed graphs, with small integer edge weights, in strongly subcubic time.

On the other hand, despite considerable progress in the small-integer-weight setting, nobody knows how to compute all-pairs shortest paths for more general edge weights in $O(V^{2.999999})$ time, for any number of 9s. Moreover, there is some evidence that such an algorithm is impossible! So maybe "not obvious" does mean "impossible" after all.

[6] Determining the minimum time required to multiply two arbitrary $n \times n$ matrices is a long-standing open problem; many people believe there is an undiscovered algorithm that runs in $O(n^{2+\varepsilon})$ time for any $\varepsilon > 0$, or possibly even in $O(n^2)$ time.

[7] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400-403, 1995. This is one of the few algorithms papers where (in the 1992 conference version at least) the algorithm is completely described and analyzed *in the abstract* of the paper. See also: Noga Alon, Zvi Galil, Oded Margalit*. On the exponent of the all pairs shortest path problem. *Journal of Computer and System Sciences* 54(2):255–262, 1997.

9.8 (Kleene-Roy-)Floyd-Warshall(-Ingerman)

Our fast dynamic programming algorithm is still a factor of $O(\log V)$ slower in the worst case than the standard implementation of Johnson's algorithm. A different formulation of shortest paths that removes this logarithmic factor was proposed twice in 1962, first by Robert Floyd and later independently by Peter Ingerman, both slightly generalizing an algorithm of Stephen Warshall published earlier in the same year. In fact, Warshall's algorithm was previously discovered by Bernard Roy in 1959, and the underlying recursion pattern was used by Stephen Kleene[8] in 1951.

Warshall's (and Roy's and Kleene's) insight was to use a different third parameter in the dynamic programming recurrence. Instead of considering paths with a limited number of edges, they considered paths that can pass through only certain vertices. Here, "pass through" means "both enter and leave"; for example, the path $w \to x \to y \to z$ starts at w, *passes through* x and y, and ends at z.

Number the vertices arbitrarily from 1 to V. For every pair of vertices u and v and every integer r, we define a path $\pi(u, v, r)$ as follows:

> $\pi(u, v, r)$ is the shortest path (if any) from u to v that passes through only vertices numbered at most r.

In particular, $\pi(u, v, V)$ is the true shortest path from u to v. Kleene and Roy and Warshall all observed that these paths have a simple recursive structure.

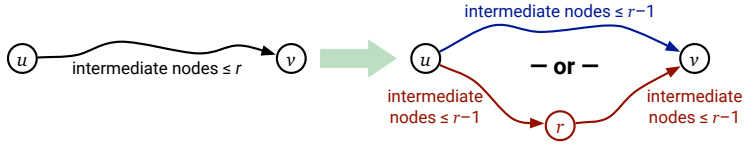

Figure 9.3. Recursive structure of the restricted shortest path $\pi(u, v, r)$.

- The path $\pi(u, v, 0)$ can't pass through any intermediate vertices, so it must be the edge (if any) from u to v.
- For any integer $r > 0$, either $\pi(u, v, r)$ passes through vertex r or it doesn't.
  - If $\pi(u, v, r)$ passes through vertex r, it consists of a subpath from u to r, followed by a subpath from r to v. Both of those subpaths pass through only vertices numbered at most $r - 1$; moreover, those subpaths are as short as possible with this restriction. So the two subpaths must be $\pi(u, r, r - 1)$ and $\pi(r, v, r - 1)$.

[8] Pronounced "clay knee", not "clean" or "clean-ee" or "clay-nuh" or "dimaggio". Specifically, Kleene described an inductive proof that every finite automata has an equivalent regular expression; Kleene's induction pattern is essentially identical to the Floyd-Warshall recurrence.

– On the other hand, if $\pi(u, v, r)$ does not pass through vertex r, then it passes through only vertices numbered at most $r-1$, and it must be the *shortest* path with this restriction. So in this case, we must have $\pi(u, v, r) = \pi(u, v, r-1)$.

Now let $dist(u, v, r)$ denote the *length* of the path $\pi(u, v, r)$. The recursive structure of $\pi(u, v, r)$ immediately implies the following recurrence:

$$dist(u, v, r) = \begin{cases} w(u \to v) & \text{if } r = 0 \\ \min \left\{ \begin{array}{c} dist(u, v, r-1) \\ dist(u, r, r-1) + dist(r, v, r-1) \end{array} \right\} & \text{otherwise} \end{cases}$$

Our goal is to compute $dist(u, v, V)$ for all vertices u and v. Once again, this recurrence can be evaluated by a straightforward dynamic programming algorithm in $\boldsymbol{O(V^3)}$ ***time***.

```
KleeneAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)

  for r ← 1 to V
      for all vertices u
          for all vertices v
              if dist[u, v, r-1] < dist[u, r, r-1] + dist[r, v, r-1]
                  dist[u, v, r] ← dist[u, v, r-1]
              else
                  dist[u, v, r] ← dist[u, r, r-1] + dist[r, v, r-1]
```

Like all our previous dynamic programming algorithms for shortest paths, we can simplify KleeneAPSP by removing the third dimension of the memoization table. Also, because we chose the vertex numbering arbitrarily, there's no reason to refer to it explicitly in the pseudocode. We finally arrive at Floyd's improvement of Warshall's algorithm:

```
FloydWarshall(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)

  for all vertices r
      for all vertices u
          for all vertices v
              if dist[u, v] > dist[u, r] + dist[r, v]
                  dist[u, v] ← dist[u, r] + dist[r, v]
```

It's interesting to compare FloydWarshall with our earlier, slightly slower dynamic programming algorithm LeyzorekAPSP. Instead of $O(\log V)$ passes

through all triples of vertices, FloydWarshall requires only a single pass, but only because it uses a different nesting order for the three loops!

Exercises

1. (a) Describe a modification of LeyzorekAPSP that returns an array of predecessor pointers, in addition to the array of shortest path distances, still in $O(V^3 \log V)$ time.

   (b) Describe a modification of FloydWarshall that returns an array of predecessor pointers, in addition to the array of shortest path distances, still in $O(V^3)$ time.

2. All of the algorithms discussed in this chapter fail if the graph contains a negative cycle. Johnson's algorithm detects the negative cycle in the initialization phase (via Bellman-Ford) and aborts; the dynamic programming algorithms just return incorrect results. However, *all* of these algorithms can be modified to return correct shortest-path distances, even in the presence of negative cycles. Specifically, for all vertices u and v:

   - If u cannot reach v, the algorithm should return $dist[u, v] = \infty$.
   - If u can reach a negative cycle that can reach v, the algorithm should return $dist[u, v] = -\infty$.
   - Otherwise, there is a shortest path from u to v, so the algorithm should return its length.

   (a) Describe how to modify Johnson's algorithm to return the correct shortest-path distances, even if the graph has negative cycles.

   (b) Describe how to modify LeyzorekAPSP to return the correct shortest-path distances, even if the graph has negative cycles.

   (c) Describe how to modify Floyd-Warshall to return the correct shortest-path distances, even if the graph has negative cycles.

3. The algorithms described in this chapter can also be modified to return an explicit description of some negative cycle in the input graph G, if one exists, instead of only reporting whether or not G contains a negative cycle.

   (a) Describe how to modify Johnson's algorithm to return either the array of all shortest-path distances or a negative cycle.

   (b) Describe how to modify LeyzorekAPSP to return either the array of all shortest-path distances or a negative cycle.

   (c) Describe how to modify Floyd-Warshall to return either the array of all shortest-path distances or a negative cycle.

In all cases, if the input graph contains more than one negative cycle, your algorithms may choose one arbitrarily.

4. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights can be positive, negative, or zero, but there are no negative cycles.
   (a) Describe an efficient algorithm that either finds a cycle of length zero in G, or correctly reports that no such cycle exists.
   (b) Describe an efficient algorithm that constructs a subgraph H of G with the following properties:
   - Every vertex of G is a vertex of H.
   - Every directed cycle in H has length 0.
   - Every directed cycle of length 0 in G is also a cycle in H.

   In particular, if there are no zero-cycles in G, then H has no edges.

5. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights can be positive, negative, or zero. Suppose the vertices of G are partitioned into k disjoint subsets $V_1, V_2, \ldots, V_k$; that is, every vertex of G belongs to exactly one subset V_i. For each i and j, let $\delta(i, j)$ denote the minimum shortest-path distance between vertices in V_i and vertices in V_j:

$$\delta(i, j) = \min \left\{ dist(v_i, v_j) \;\middle|\; v_i \in V_i \text{ and } v_j \in V_j \right\}.$$

   Describe an algorithm to compute $\delta(i, j)$ for all i and j. For full credit, your algorithm should run in $O(VE + kV \log V)$ time.

6. In this problem we will discover how you, yes ***you***, can be employed by Wall Street and cause a major economic collapse! The *arbitrage* business is a money-making scheme that takes advantage of differences in currency exchange. In particular, suppose 1 US dollar buys 120 Japanese yen; 1 yen buys 0.01 euros; and 1 euro buys 1.2 US dollars. Then, a trader starting with \$1 can convert their money from dollars to yen, then from yen to euros, and finally from euros back to dollars, ending with \$1.44! The cycle of currencies \$ → ¥ → € → \$ is called an ***arbitrage cycle***. Of course, finding and exploiting arbitrage cycles before the prices are corrected requires extremely fast algorithms.

   Suppose n different currencies are traded in your currency market. You are given the matrix $R[1..n, 1..n]$ of exchange rates between every pair of currencies; for each i and j, one unit of currency i can be traded for $R[i, j]$ units of currency j. (Do *not* assume that $R[i, j] \cdot R[j, i] = 1$.)
   (a) Describe an algorithm that returns an array $V[1..n]$, where $V[i]$ is the maximum amount of currency i that you can obtain by trading, starting with one unit of currency 1, assuming there are no arbitrage cycles.

(b) Describe an algorithm to determine whether the given matrix of currency exchange rates creates an arbitrage cycle.

(c) Modify your algorithm from part (b) to actually return an arbitrage cycle, if it exists.

7. Morty needs to retrieve a stabilized plumbus from the Clackspire Labyrinth. He must enter the labyrinth using Rick's interdimensional portal gun, traverse the Labyrinth to a plumbus, then take that plumbus through the Labyrinth to a fleeb to be stabilized, and finally take the stabilized plumbus back to the original portal to return home. Plumbuses are stabilized by fleeb juice, which any fleeb will release immediately after being removed from its fleebhole. An unstabilized plumbus will explode if it is carried more than 137 flinks from its original storage unit. The Clackspire Labyrinth smells like farts, so Morty wants to spend as little time there as possible.

   Rick has given Morty a detailed map of the Clackspire Labyrinth, which consist of a directed graph $G = (V, E)$ with non-negative edge weights (indicating distance in flinks), along with two disjoint subsets $P \subset V$ and $F \subset V$, indicating the plumbus storage units and fleebholes, respectively. Morty needs to identify a start vertex s, a plumbus storage unit $p \in P$, and a fleebhole $f \in F$, such that the shortest-path distance from p to f is at most 137 flinks long, and the length of the shortest walk $s \rightsquigarrow p \rightsquigarrow f \rightsquigarrow s$ is as short as possible.

   Describe and analyze an algo(burp)rithm to so(burp)olve Morty's problem. You can assume that it is in fact possible for Morty to succeed.

8. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero.

   (a) How would we delete an arbitrary vertex v from this graph, without changing the shortest-path distance between any other pair of vertices? Describe an algorithm that constructs a directed graph $G' = (V \setminus \{v\}, E')$ with weighted edges, such that the shortest-path distance between any two vertices in G' is equal to the shortest-path distance between the same two vertices in G, in $O(V^2)$ time.

   (b) Now suppose we have already computed all shortest-path distances in G'. Describe an algorithm to compute the shortest-path distances in the original graph G from v to every other vertex, and from every other vertex to v, all in $O(V^2)$ time.

   (c) Combine parts (a) and (b) into another all-pairs shortest path algorithm that runs in $O(V^3)$ time. (The resulting algorithm is *almost* the same as Floyd-Warshall!)

9. A third type of matrix multiplication is also relevant for shortest-path algorithms. Suppose A and B are *boolean* $n \times n$ matrices. The *boolean* or *and-or* product of A and B is the $n \times n$ matrix C where $C[i,j] = \bigvee_k (A[i,k] \wedge B[k,j])$.

   (a) Reduce boolean matrix multiplication to min-plus matrix multiplication. That is, given a subroutine MinPlusMultiply that computes the min-plus product of two $n \times n$ matrices, describe and analyze an algorithm BooleanMatrixMultiply that multiplies two boolean matrices.

   (b) Reduce boolean matrix multiplication to standard matrix multiplication. That is, given a subroutine MatrixMultiply that computes the standard product of two $n \times n$ matrices, describe and analyze an algorithm BooleanMatrixMultiply that multiplies two boolean matrices.

10. The *transitive closure* of a directed graph G contains an edge $u \to v$ if and only if there is a directed path from u to v in G.

   (a) Suppose we can multiply two $n \times n$ boolean matrices in $O(n^\omega)$ time, for some constant $2 \le \omega < 3$. (Problem 9(b) implies $\omega \le 2.372864$.) Describe an algorithm to compute the transitive closure of an n-vertex directed graph in $O(n^\omega \log n)$ time.

   (b) Now suppose G is a directed *acyclic* graph. Describe an algorithm to compute the transitive closure of G in $O(n^\omega)$ time. *[Hint: Do what you always do with dags, and then divide and conquer. Use the fact that $\omega \ge 2$.]*

   (c) Finally, describe an algorithm to compute the transitive closure of an *arbitrary* directed graph in $O(n^\omega)$ time. *[Hint: Do what you always do to turn an arbitrary directed graph into a dag.]*

   (d) Now let's reverse the previous reduction. Given a subroutine TransitiveClosure that computes the transitive closure of an n-vertex directed graph in $O(n^\alpha)$ time, for some constant $2 \le \alpha < 3$, describe and analyze an algorithm for boolean matrix multiplication that runs in $O(n^\alpha)$ time.

11. Prove that the following recursive algorithm correctly computes all-pairs shortest-path distances in $O(n^3)$ time. For simplicity, you may assume n is a power of 2. As usual, the array D is passed *by reference* to the helper function RecAPSP. *[Hint: This is a jumbled version of Floyd-Warshall, with significantly better cache behavior.[9]]*

[9] Joon-Sang Park, Michael Penner, and Viktor K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Trans. Parallel and Distributed Systems* 15(9):769–782, 2004. For a significant generalization to a wider class of dynamic programming problems, see: Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-oblivious dynamic programming. *Proc. 17th SODA* 591–600, 2006.

```
RECURSIVEAPSP(V, E, w):
  n ← |V|
  for i ← 1 to n
      for j ← 1 to n
          if i = j
              D[i, j] ← 0
          if i→j ∈ E
              D[i, j] ← w(i→j)
          else
              D[i, j] ← ∞
  RECAPSP(D, n, 1, 1, 1)
  return D[1..n, 1..n]
```

```
RECAPSP(D, n, i, j, k):
  if n = 1
      D[i, j] ← min{D[i, j], D[i, k] + D[j, k]}
  else
      m ← n/2
      RECAPSP(D, n/2, i,     j,     k     )
      RECAPSP(D, n/2, i,     j,     k + m)
      RECAPSP(D, n/2, i,     j + m, k     )
      RECAPSP(D, n/2, i,     j + m, k + m)
      RECAPSP(D, n/2, i + m, j,     k     )
      RECAPSP(D, n/2, i + m, j,     k + m)
      RECAPSP(D, n/2, i + m, j + m, k     )
      RECAPSP(D, n/2, i + m, j + m, k + m)
```

♥12. Let $G = (V, E)$ be an undirected, unweighted, connected, n-vertex graph, represented by the adjacency matrix $A[1..n, 1..n]$. In this problem, we will derive Seidel's sub-cubic algorithm to compute the $n \times n$ matrix $D[1..n, 1..n]$ of shortest-path distances in G using fast matrix multiplication. Assume that we have a subroutine MATRIXMULTIPLY that computes the standard product of two $n \times n$ matrices in $O(n^\omega)$ time, for some unknown constant $\omega \geq 2$.

(a) Let G^2 denote the graph with the same vertices as G, where two vertices are connected by a edge if and only if they are connected by a path of length at most 2 in G. Describe an algorithm to compute the adjacency matrix of G^2 using a single call to MATRIXMULTIPLY and $O(n^2)$ additional time.

(b) Suppose we discover that G^2 is a complete graph. Describe an algorithm to compute the matrix D of shortest path distances in G in $O(n^2)$ additional time.

(c) Suppose we recursively compute the matrix D^2 of shortest-path distances in G^2. Prove that the shortest-path distance in G from node i to node j is either $2 \cdot D^2[i, j]$ or $2 \cdot D^2[i, j] - 1$.

(d) Now suppose G^2 is *not* a complete graph. Let $X = D^2 \cdot A$, and let $\deg(i)$ denote the degree of vertex i in the original graph G. Prove that the shortest-path distance from node i to node j in G is $2 \cdot D^2[i, j]$ if and only if $X[i, j] \geq D^2[i, j] \cdot \deg(i)$.

(e) Describe an algorithm to compute the matrix D of shortest-path distances in G in $O(n^\omega \log n)$ time.

13. Gideon Yuval proposed the following reduction from min-plus matrix multiplication to standard matrix multiplication in 1976. Suppose we are given two $n \times n$ matrices A and B of integers, all between 0 and M, and we want to compute their min-sum product matrix C, defined by setting

$C[i,k] = \min_j(A[i,j] + B[j,k])$. Define two new $n \times n$ matrices A' and B', where

$$A'[i,j] = n^{M-A[i,j]} \qquad \text{and} \qquad B'[i,j] = n^{M-B[i,j]}.$$

Finally, let C' be the (standard) product of A' and B', defined by setting $C'[i,k] = \sum_j A'[i,j] \cdot B'[j,k]$.

(a) Describe an algorithm to construct A' from A using only standard arithmetic operations ($+$, $-$, $\times$).

(b) Describe an algorithm to extract the min-sum product C from C', using only standard arithmetic operations ($+$, $-$, $\times$).[10]

(c) Suppose we can (seriously) multiply two $n \times n$ integer matrices using $O(n^\omega)$ arithmetic operations, for some constant $2 \le \omega < 3$. How many arithmetic operations does Yuval's algorithm need to compute the min-sum product C?

(d) Given a single $n \times n$ integer matrix A, how many arithmetic operations are required to compute the nth "funny" power of A using Yuval's algorithm? (Recall that if A is the weighted adjacency matrix of a graph, then the nth "funny" power of A is the matrix of shortest-path distances.)

(e) Why doesn't Yuval's algorithm imply an all-pairs shortest path algorithm that is faster than Floyd-Warshall for *arbitrary* edge weights? How are we cheating?

[10] In particular, do *not* use division or the floor function $\lfloor x \rfloor$. Trust me—this is a can of worms you do *not* want to open.

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 first 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

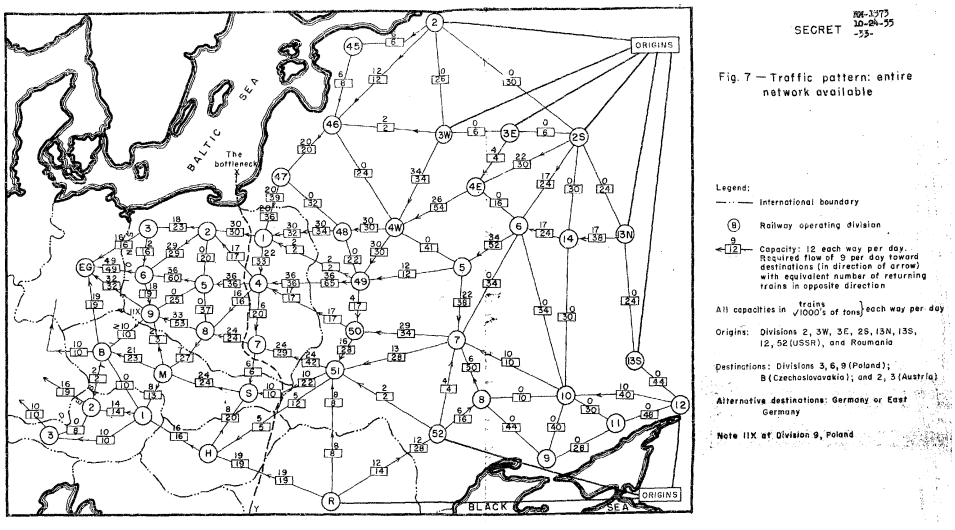

Figure 10.1. Harris and Ross's map of the Warsaw Pact rail network. (See Image Credits at the end of the book.)

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.

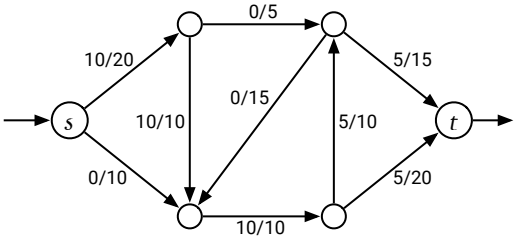

Figure 10.2. A feasible (s, t)-flow with value 10. Each edge is labeled with its flow/capacity.

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 large as possible.

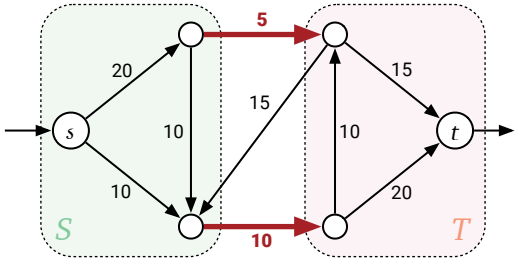

Figure 10.3. An (s,t)-cut with capacity 15. Each edge is labeled with its capacity.

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.

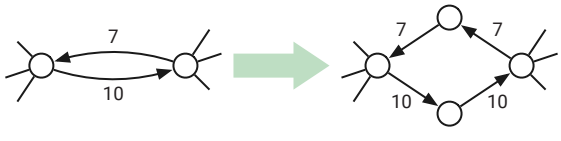

Figure 10.4. Enforcing the one-direction assumption.

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 $v{\to}u$.

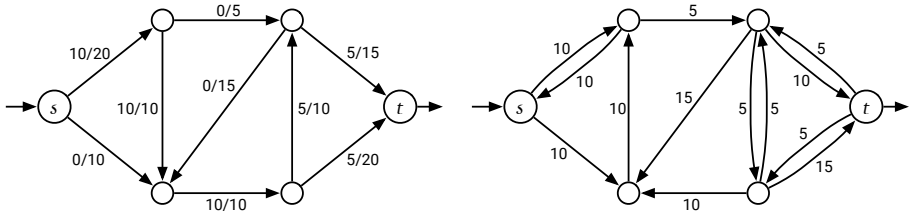

Figure 10.5. A flow f in a weighted graph G and the corresponding residual graph G_f.

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:

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

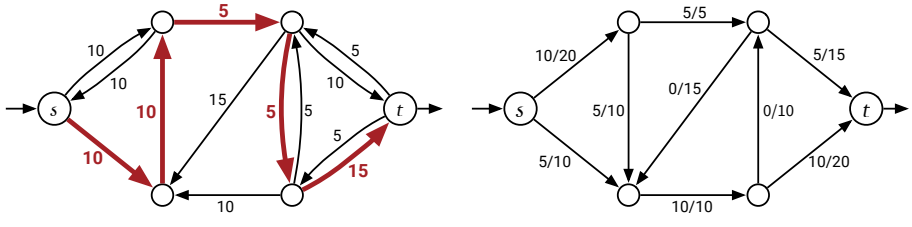

Figure 10.6. An augmenting path with value $F = 5$ and the resulting augmented flow f'.

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.

The following example shows that this running time analysis is essentially tight. Consider the 4-node network illustrated below, 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 $\Theta(X) = \Omega(|f^*|)$.

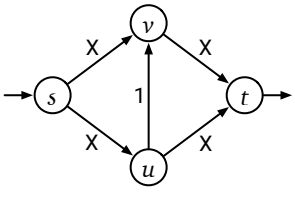

Figure 10.7. A bad example for the Ford-Fulkerson algorithm.

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 $X-3$.)

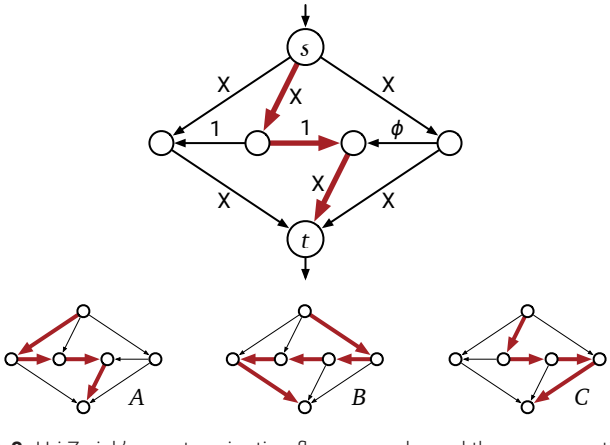

Figure 10.8. Uri Zwick's non-terminating flow example, and three augmenting paths.

[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 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 $\phi^{2n-2}, 0, \phi^{2n-1}$. 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 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 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

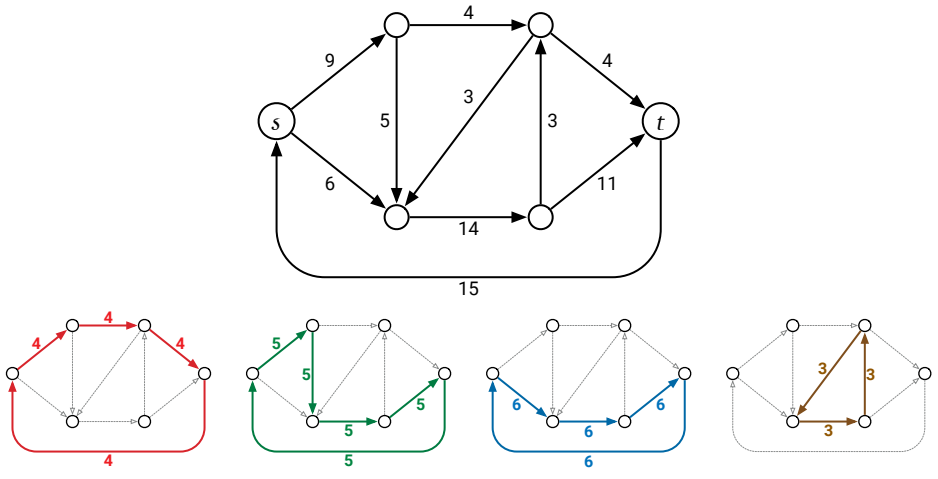

Figure 10.9. Decomposing a circulation into weighted directed cycles.

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 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

[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 $v \to u$.

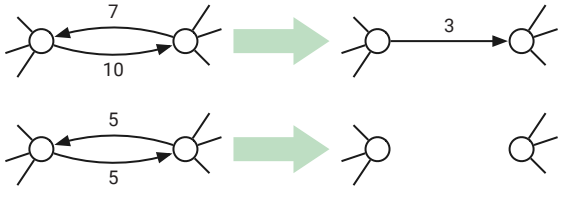

Figure 10.11. Enforcing the one-direction assumption.

   (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 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 $O(E)$ 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*.

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.

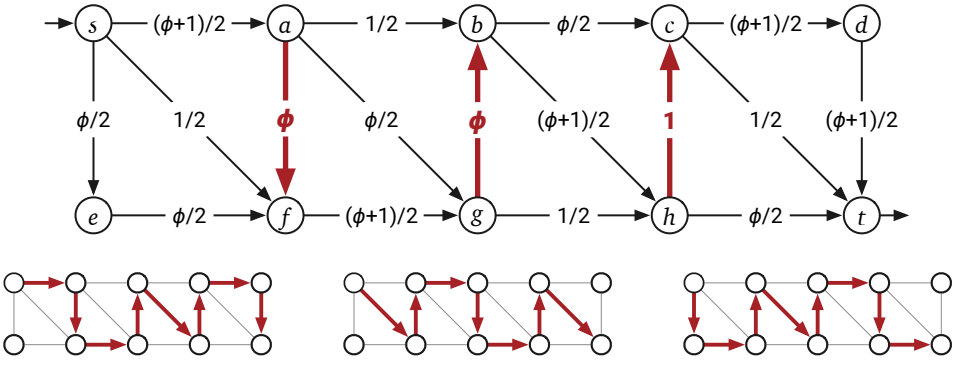

Figure 10.12. Queyranne's network, and a sequence of "fattest path" augmentations.

(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 10.12.)

```
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.

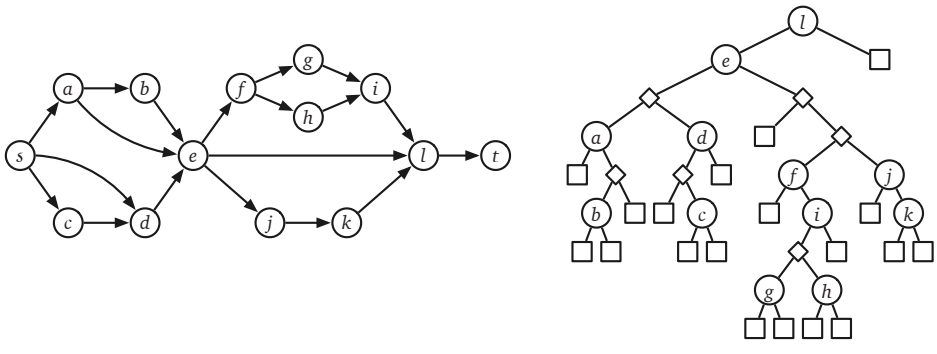

Figure 10.13. A series-parallel graph and a corresponding decomposition tree. Squares in the decomposition tree are leaves; diamonds are parallel nodes.

(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" heuristic, 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.

The algorithm maintains a bottleneck threshold Δ; initially, Δ is the maximum capacity among all edges in the graph. 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 the algorithm execute in the worst case, if the edge capacities are integers?

(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 this capacity scaling algorithm, assuming all the edge capacities are integers less than U?