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 $s$
        $dist[s, \cdot] \leftarrow$ 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.

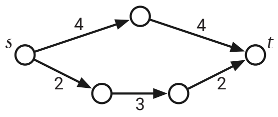

Figure 9.1. Increasing all the edge weights by 2 changes the shortest path from s to t.

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 $\pi(v) = \mathit{dist}(s, v)$. The new weight of every edge is

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

These new weights non-negative *because* Bellman-Ford halted! Recall that an edge $u{\to}v$ is *tense* if $\mathit{dist}(s,u) + w(u{\to}v) < \mathit{dist}(s, v)$, 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 $s \to v$
      $w(s \to v) \leftarrow 0$
  ⟨⟨Compute vertices prices⟩⟩
  $dist[s,\cdot] \leftarrow$ BellmanFord$(V, E, w, s)$
  if BellmanFord found a negative cycle
      fail gracefully
  ⟨⟨Reweight edges⟩⟩
  for every edge $(u,v) \in E$
      $w'(u \to v) \leftarrow dist[s,u] + w(u \to v) - dist[s,v]$
  ⟨⟨Compute reweighted shortest paths⟩⟩
  for every vertex $u$
      $dist'[u,\cdot] \leftarrow$ Dijkstra$(V, E, w', u)$
  ⟨⟨Compute original shortest-path distances⟩⟩
  for every vertex $u$
      for every vertex $v$
          $dist[u,v] \leftarrow dist'[u,v] - dist[s,u] + dist[s,v]$
```

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:

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

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 $u = v$
              $dist[u,v,0] \gets 0$
          else
              $dist[u,v,0] \gets \infty$
  for $\ell \gets 1$ to $V-1$
      for all vertices $u$
          for all vertices $v \neq u$
              $dist[u,v,\ell] \gets dist[u,v,\ell-1]$
              for all edges $x \to v$
                  if $dist[u,v,\ell] > dist[u,x,\ell-1] + w(x \to v)$
                      $dist[u,v,\ell] \gets dist[u,x,\ell-1] + w(x \to v)$
```

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 $u = v$
              $dist[u,v] \gets 0$
          else
              $dist[u,v] \gets \infty$
  for $\ell \gets 1$ to $V-1$
      for all vertices $u$
          for all edges $x \to v$
              if $dist[u,v] > dist[u,x] + w(x \to v)$
                  $dist[u,v] \gets dist[u,x] + w(x \to v)$
```

[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.

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

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* $\ell \ge V - 1$; 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 $v$
            $dist[u, v, 0] \leftarrow w(u\to v)$

    for $i \leftarrow 1$ to $\lceil \lg V \rceil$        ⟨⟨$\ell = 2^i$⟩⟩
        for all vertices $u$
            for all vertices $v$
                $dist[u, v, i] \leftarrow \infty$
                for all vertices $x$
                    if $dist[u, v, i] > dist[u, x, i-1] + dist[x, v, i-1]$
                        $dist[u, v, i] \leftarrow dist[u, x, i-1] + dist[x, v, i-1]$
```

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 $v$
            $dist[u,v] \leftarrow w(u\to v)$
    for $i \leftarrow 1$ to $\lceil \lg V \rceil$        $\langle\langle \ell = 2^i \rangle\rangle$
        for all vertices $u$
            for all vertices $v$
                for all vertices $x$
                    if $dist[u,v] > dist[u,x] + dist[x,v]$
                        $dist[u,v] \leftarrow dist[u,x] + dist[x,v]$
```

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 $i \leftarrow 1$ to $n$
        for $j \leftarrow 1$ to $n$
            $A'[i,j] \leftarrow 0$
            for $k \leftarrow 1$ to $n$
                $A'[i,j] \leftarrow A'[i,j] + A[i,k] \cdot A[k,j]$
```

```
FischerMeyerInnerLoop(D):
    for all vertices $u$
        for all vertices $v$
            $D'[u,v] \leftarrow \infty$
            for all vertices $x$
                $D'[u,v] \leftarrow \min\{D'[u,v],\ D[u,x] + D[x,v]\}$
```

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 **??**. 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.

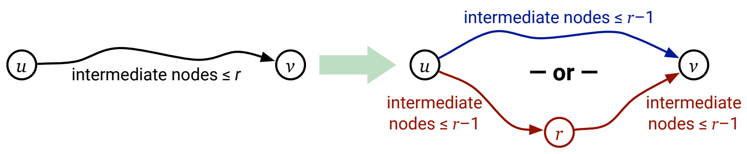

Figure 9.3. Recursive structure of the restricted shortest path $\pi(u, v, r)$.

- 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 ***$O(V^3)$ time***.

```
KleeneAPSP($V, E, w$):
  for all vertices $u$
      for all vertices $v$
          $dist[u,v,0] \gets w(u\to v)$

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

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 $v$
          $dist[u,v] \gets w(u\to v)$

  for all vertices $r$
      for all vertices $u$
          for all vertices $v$
              if $dist[u,v] > dist[u,r] + dist[r,v]$
                  $dist[u,v] \gets dist[u,r] + dist[r,v]$
```

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($V, E, w$):
    $n \leftarrow |V|$
    for $i \leftarrow 1$ to $n$
        for $j \leftarrow 1$ to $n$
            if $i = j$
                $D[i, j] \leftarrow 0$
            if $i \rightarrow j \in E$
                $D[i, j] \leftarrow w(i \rightarrow j)$
            else
                $D[i, j] \leftarrow \infty$
    RecAPSP($D, n, 1, 1, 1$)
    return $D[1..n, 1..n]$
```

```
RecAPSP($D, n, i, j, k$):
    if $n = 1$
        $D[i, j] \leftarrow \min\{D[i, j],\ D[i, k] + D[j, k]\}$
    else
        $m \leftarrow n/2$
        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.