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 J. 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, "Five Hundred Miles (I'm Gonna Be)", *Sunshine on Leith* (2001)

Almost there. . . Almost there. . .

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

27 All-Pairs Shortest Paths

In the previous lecture, we saw algorithms to find the shortest path from a source vertex s to a target vertex t in a directed graph. As it turns out, the best algorithms for this problem actually find the shortest path from s to every possible target (or from every possible source to t) by constructing a shortest path tree. 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 (if any) from s to v;
- $pred(v)$ is the second-to-last vertex (if any) the shortest path (if any) from s to v.

In this lecture, we want to generalize the shortest path problem even further. In the *all pairs shortest path* problem, we want to find the shortest path from *every* possible source to *every* possible destination. Specifically, for every pair of vertices u and v, we need to compute the following information:

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

For example, for any vertex v, we have $dist(v, v) = 0$ and $pred(v, v) = \text{NULL}$. If the shortest path from u to v is only one edge long, then $dist(u, v) = w(u \to v)$ and $pred(u, v) = u$. If there is *no* shortest path from u to v—either because there's no path at all, or because there's a negative cycle—then $dist(u, v) = \infty$ and $pred(v, v) = \text{NULL}$.

The output of our shortest path algorithms will be a pair of $V \times V$ arrays encoding all V^2 distances and predecessors. Many maps include a distance matrix—to find the distance from (say) Champaign to (say) Columbus, you would look in the row labeled 'Champaign' and the column labeled 'Columbus'. In these notes, I'll focus almost exclusively on computing the distance array. The predecessor array, from which you would compute the actual shortest paths, can be computed with only minor additions to the algorithms I'll describe (hint, hint).

27.1 Lots of Single Sources

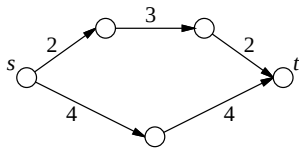
The obvious solution to the all-pairs shortest path problem is just to run a single-source shortest path algorithm V times, once for every possible source vertex! Specifically, to fill in the one-dimensional subarray $dist[s, \cdot]$, we invoke either Dijkstra's or Shimbel's algorithm starting at the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

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

The running time of this algorithm depends on which single-source shortest path algorithm we use. If we use Shimbel's algorithm, the overall running time is $\Theta(V^2E) = O(V^4)$. If all the edge weights are non-negative, we can use Dijkstra's algorithm instead, which decreases the running time to $\Theta(VE + V^2 \log V) = O(V^3)$. For graphs with negative edge weights, Dijkstra's algorithm can take exponential time, so we can't get this improvement directly.

27.2 Reweighting

One idea that occurs to most people is increasing the weights of all the edges by the same amount so that all the weights become positive, and then applying Dijkstra's algorithm. Unfortunately, this simple idea doesn't work. Different paths change by different amounts, which means the shortest paths in the reweighted graph may not be the same as in the original graph.



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

However, there is a more complicated method for reweighting the edges in a graph. Suppose each vertex v has some associated *cost* $c(v)$, which might be positive, negative, or zero. We can define a new weight function w' as follows:

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

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

Now it's not too hard to show that the shortest paths with the new weight function w' are exactly the same as the shortest paths with the original weight function w. In fact, for *any* path $u \rightsquigarrow v$ from one vertex u to another vertex v, we have

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

We pay $c(u)$ in exit fees, plus the original weight of of the path, minus the $c(v)$ entrance gift. At every intermediate vertex x on the path, we get $c(x)$ as an entrance gift, but then immediately pay it back as an exit tax!

27.3 Johnson's Algorithm

Johnson's all-pairs shortest path algorithm finds a cost $c(v)$ for each vertex, so that when the graph is reweighted, every edge has non-negative weight.

Suppose the graph has a vertex s that has a path to every other vertex. Johnson's algorithm computes the shortest paths from s to every other vertex, using Shimbel's algorithm (which doesn't care if the edge weights are negative), and then sets $c(v) \leftarrow dist(s, v)$, so the new weight of every edge is

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

Why are all these new weights non-negative? Because otherwise, Shimbel's algorithm wouldn't be finished! 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. The only exception is if the graph has a negative cycle, but then shortest paths aren't defined, and Johnson's algorithm simply aborts.

But what if the graph *doesn't* have a vertex s that can reach everything? No matter where we start Shimbel's algorithm, some of those vertex costs will be infinite. Johnson's algorithm avoids this problem by adding a new vertex s to the graph, with zero-weight edges going from s to every other vertex, but *no* edges going back into s. This addition doesn't change the shortest paths between any other pair of vertices, because there are no paths into s.

So here's Johnson's algorithm in all its glory.

```
JOHNSONAPSP(V, E, w) :
  create a new vertex s
  for every vertex v
      w(s→v) ← 0
      w(v→s) ← ∞
  dist[s, ·] ← SHIMBEL(V, E, w, s)
  if SHIMBEL found a negative cycle
      fail gracefully
  for every edge (u, v) ∈ E
      w'(u→v) ← dist[s, u] + w(u→v) − dist[s, v]
  for every vertex u
      dist[u, ·] ← DIJKSTRA(V, E, w', u)
      for every vertex v
          dist[u, v] ← dist[u, v] − dist[s, u] + dist[s, v]
```

The algorithm spends $\Theta(V)$ time adding the artificial start vertex s, $\Theta(VE)$ time running SHIMBEL, $O(E)$ time reweighting the graph, and then $\Theta(VE + V^2 \log V)$ running V passes of Dijkstra's algorithm. Thus, the overall running time is $\boldsymbol{\Theta(VE + V^2 \log V)}$.

27.4 Dynamic Programming

There's a completely different solution to the all-pairs shortest path problem that uses dynamic programming instead of a single-source algorithm. For *dense* graphs where $E = \Omega(V^2)$, the dynamic programming approach eventually leads to the same $O(V^3)$ running time as Johnson's algorithm, but with a much simpler algorithm. In particular, the new algorithm avoids Dijkstra's algorithm, which gets its efficiency from Fibonacci heaps, which are rather easy to screw up in the implementation. **In the rest of this lecture, I will assume that the input graph contains no negative cycles.**

As usual for dynamic programming algorithms, we first need to come up with a recursive formulation of the problem. Here is an "obvious" recursive definition for $dist(u, v)$:

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

In other words, to find the shortest path from u to v, we consider all possible last edges $x \to v$ and recursively compute the shortest path from u to x. **Unfortunately, this recurrence doesn't work!** To compute $dist(u, v)$, we may need to compute $dist(u, x)$ for every other vertex x. But to compute $dist(u, x)$, we may need to compute $dist(u, v)$. We're stuck in an infinite loop!

To avoid this circular dependency, we need an additional parameter that decreases at each recursion, eventually reaching zero at the base case. One possibility is to include the number of edges in the shortest path as this third magic parameter, just as we did in the dynamic programming formulation of Shimbel's

algorithm. Let $dist(u, v, k)$ denote the length of the shortest path from u to v that uses *at most k edges*. Since we know that the shortest path between any two vertices has at most $V-1$ vertices, $dist(u, v, V-1)$ is the actual shortest-path distance. As in the single-source setting, we have the following recurrence:

$$dist(u, v, k) = \begin{cases} 0 & \text{if } u = v \\ \infty & \text{if } k = 0 \text{ and } u \neq v \\ \min_{x \to v} \big(dist(u, x, k-1) + w(x \to v) \big) & \text{otherwise} \end{cases}$$

Turning this recurrence into a dynamic programming algorithm is straightforward. To make the algorithm a little shorter, let's assume that $w(v \to v) = 0$ for every vertex v. Assuming the graph is stored in an adjacency list, the resulting algorithm runs in $\boldsymbol{\Theta(V^2E)}$ ***time***.

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

This algorithm was first sketched by Shimbel in 1955; in fact, this algorithm is just running V different instances of Shimbel's single-source algorithm, one for each possible source vertex. Just as in the dynamic programming development of Shimbel's single-source algorithm, we don't actually need the inner loop over vertices v, and we only need a two-dimensional table. After the kth iteration of the main loop in the following algorithm, $dist[u, v]$ lies between the true shortest path distance from u to v and the value $dist[u, v, k]$ computed in the previous algorithm.

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

  for k ← 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)
```

27.5 Divide and Conquer

But we can make a more significant improvement. The recurrence we just used broke the shortest path into a slightly shorter path and a single edge, by considering all predecessors. Instead, let's break it

into two shorter paths at the *middle* vertex of the path. This idea gives us a different recurrence for $dist(u, v, k)$. Once again, to simplify things, let's assume $w(v \to v) = 0$.

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

This recurrence only works when k is a power of two, since otherwise we might try to find the shortest path with a fractional number of edges! But that's not really a problem, since $dist(u, v, 2^{\lceil \lg V \rceil})$ gives us the overall shortest distance from u to v. Notice that we use the base case $k = 1$ instead of $k = 0$, since we can't use half an edge.

Once again, a dynamic programming solution is straightforward. Even before we write down the algorithm, we can tell the running time is $\mathbf{\Theta(V^3 \log V)}$—we consider V possible values of u, v, and x, but only $\lceil \lg V \rceil$ possible values of k.

```
FastDynamicProgrammingAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)
  for i ← 1 to ⌈lg V⌉                ⟨⟨k = 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]
```

This algorithm is ***not*** the same as V invocations of any single-source algorithm; in particular, the innermost loop does not simply relax tense edges. However, we can remove the last dimension of the table, using $dist[u, v]$ everywhere in place of $dist[u, v, i]$, just as in Shimbel's single-source algorithm, thereby reducing the space from $O(V^3)$ to $O(V^2)$.

```
FastShimbelAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)
  for i ← 1 to ⌈lg V⌉
      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]
```

This faster algorithm was discovered by Leyzorek *et al.* in 1957, in the same paper where they describe Dijkstra's algorithm.

27.6 Aside: '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 multiplying two $n \times n$ matrices A and B with the inner loop of our first dynamic programming algorithm. (I've changed the variable names in the second algorithm slightly to make the similarity clearer.)

```
MATRIXMULTIPLY(A, B):
  for i ← 1 to n
      for j ← 1 to n
          C[i, j] ← 0
          for k ← 1 to n
              C[i, j] ← C[i, j] + A[i, k] · B[k, j]
```

```
APSPINNERLOOP:
  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] + w[x, v]}
```

The *only* difference between these two algorithms is that we use addition instead of multiplication and minimization instead of addition. For this reason, the shortest path inner loop is often referred to as 'funny' matrix multiplication.

DYNAMICPROGRAMMINGAPSP is the standard iterative algorithm for computing the $(V-1)$th 'funny power' of the weight matrix w. The first set of for loops sets up the 'funny identity matrix', with zeros on the main diagonal and infinity everywhere else. Then each iteration of the second main for loop computes the next 'funny power'. FASTDYNAMICPROGRAMMINGAPSP replaces this iterative method for computing powers with repeated squaring, exactly like we saw at the beginning of the semester. The fast algorithm is simplified slightly by the fact that unless there are negative cycles, every 'funny power' after the Vth is the same.

There are faster methods for multiplying matrices, similar to Karatsuba's divide-and-conquer algorithm for multiplying integers. (Google for 'Strassen's algorithm'.) Unfortunately, these algorithms us subtraction, and there's no 'funny' equivalent of subtraction. (What's the inverse operation for min?) So at least for general graphs, there seems to be no way to speed up the inner loop of our dynamic programming algorithms.

Fortunately, this isn't true. There a beautiful randomized algorithm, discovered by Alon, Galil, Margalit, and Noar[1], that computes all-pairs shortest paths in undirected graphs in $O(M(V)\log^2 V)$ expected time, where $M(V)$ is the time to multiply two $V \times V$ integer matrices. A simplified version of this algorithm for *unweighted* graphs was discovered by Seidel.[2]

27.7 Floyd-(Roy-Kleene-)Warshall

Our fast dynamic programming algorithm is still a factor of $O(\log V)$ slower than Johnson's algorithm. A different formulation that removes this logarithmic factor was proposed in 1962 by Robert Floyd, slightly generalizing an algorithm of Stephen Warshall published earlier in the same year. (In fact, Warshall's algorithm was independently discovered by Bernard Roy in 1959, but the underlying technique was used even earlier by Stephen Kleene[3] in 1951.) Warshall's (and Roy's and Kleene's) insight was to use a different third parameter in the dynamic programming recurrence.

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:

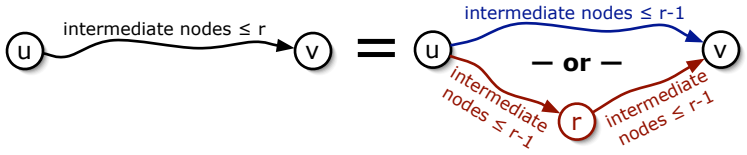
[1]Noga Alon, Zvi Galil, Oded Margalit*, and Moni Naor. Witnesses for Boolean matrix multiplication and for shortest paths. *Proc. 33rd FOCS* 417-426, 1992. See also Noga Alon, Zvi Galil, Oded Margalit*. On the exponent of the all pairs shortest path problem. *Journal of Computer and System Sciences* 54(2):255–262, 1997.

[2]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 conference version at least) the algorithm is completely described and analyzed *in the abstract* of the paper.

[3]Pronounced "clay knee", not "clean" or "clean-ee" or "clay-nuh" or "dimaggio".

$\pi(u, v, r)$:= the shortest path from u to v where every intermediate vertex (that is, every vertex except u and v) is numbered at most r.

If $r = 0$, we aren't allowed to use any intermediate vertices, so $\pi(u, v, 0)$ is just the edge (if any) from u to v. If $r > 0$, then either $\pi(u, v, r)$ goes through the vertex numbered r, or it doesn't. If $\pi(u, v, r)$ does contain vertex r, it splits into a subpath from u to r and a subpath from r to v, where every intermediate vertex in these two subpaths is numbered at most $r - 1$. Moreover, the subpaths are as short as possible with this restriction, so they must be $\pi(u, r, r-1)$ and $\pi(r, v, r-1)$. On the other hand, if $\pi(u, v, r)$ does not go through vertex r, then every intermediate vertex in $\pi(u, v, r)$ is numbered at most $r - 1$; since $\pi(u, v, r)$ must be the *shortest* such path, we have $\pi(u, v, r) = \pi(u, v, r-1)$.



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

This recursive structure implies the following recurrence for the length of $\pi(u, v, r)$, which we will denote by $dist(u, v, r)$:

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

We need to compute the shortest path distance from u to v with no restrictions, which is just $dist(u, v, V)$. Once again, we should immediately see that a dynamic programming algorithm will implement this recurrence in $\boldsymbol{\Theta(V^3)}$ time.

```
FLOYDWARSHALL(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]
```

Just like our earlier algorithms, we can simplify the algorithm by removing the third dimension of the memoization table. Also, because the vertex numbering was chosen arbitrarily, there's no reason to refer to it explicitly in the pseudocode.

```
FLOYDWARSHALL2(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]
```

Now compare this algorithm with FASTSHIMBELAPSP. Instead of $O(\log V)$ passes through all triples of vertices, FLOYDWARSHALL2 only requires a single pass, but only because it uses a different nesting order for the three for-loops!

27.8 Converting DFAs to regular expressions

Floyd's algorithm is a special case of a more general method for solving problems involving paths between vertices in graphs. The earliest example (that I know of) of this technique is an 1951 algorithm of Stephen Kleene to convert a deterministic finite automaton into an equivalent regular expression.

Recall that a deterministic finite automaton (DFA) formally consists of the following components:

- A finite set Σ, called the ***alphabet***, and whose elements we call ***symbols***.
- A finite set Q, whose elements are called ***states***.
- An ***initial state*** $s \in Q$.
- A subset $A \subseteq Q$ of ***accepting states***.
- A ***transition function*** $\delta : Q \times \Sigma \to Q$.

The ***extended transition function*** $\delta^* : Q \times \Sigma^* \to Q$ is recursively defined as follows:

$$\delta^*(q, w) := \begin{cases} q & \text{if } w = \varepsilon, \\ \delta^*(\delta(q, a), x) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^*. \end{cases}$$

Finally, a DFA ***accepts*** a string $w \in \Sigma^*$ if and only if $\delta^*(s, w) \in A$.

Equivalently, a DFA is a directed (multi-)graph with labeled edges whose vertices are the states, such that each vertex (state) has exactly one outgoing edge (transition) labeled with each symbol in Σ. There is a special "start" vertex s, and a subset A of the vertices are marked as "accepting". For any string $w \in \Sigma^*$, there is a unique walk starting at s whose sequence of edge labels is w. The DFA accepts w if and only if this walk ends at a state in A.

Kleene described the following algorithm to convert DFAs into equivalent regular expressions. Suppose we are given a DFA M with n states, where (without loss of generality) each state is identified by an integer between 1 and n. Let $\boldsymbol{L(i, j, r)}$ denote the set of all strings that describe walks in M that start at state i and end at state j, such that every intermediate state has index at most r. Thus, the language accepted by M is precisely

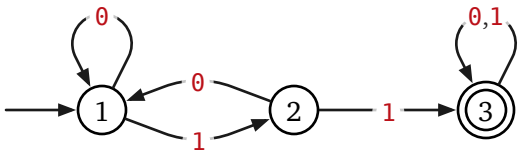
$$L(M) = \bigcup_{q \in A} L(s, q, n).$$

We prove inductively that every language $L(i, j, r)$ is regular, by recursively constructing a regular expression $R(i, j, r)$ that represents $L(i, j, r)$. There are two cases to consider.

- First, suppose $r = 0$. The language $L(i, j, 0)$ contains the labels walks from state i to state j that do not pass through *any* intermediate states. Thus, every string in $L(i, j, 0)$ has length at most 1. Specifically, for any symbol $a \in \Sigma$, we have $a \in L(i, j, 0)$ if and only if $\delta(i, a) = j$, and we have $\varepsilon \in L(i, j, 0)$ if and only if $i = j$. Thus, $L(i, j, 0)$ is always finite, and therefore regular.

  For example, the DFA sown on the next page defines the following regular languages $L(i, j, 0)$.

$$\begin{array}{lll} R[1,1,0] = \varepsilon + \texttt{0} & R[2,1,0] = \texttt{0} & R[3,1,0] = \varnothing \\ R[1,2,0] = \texttt{1} & R[2,2,0] = \varepsilon & R[3,2,0] = \varnothing \\ R[1,3,0] = \varnothing & R[2,3,0] = \texttt{1} & R[3,3,0] = \varepsilon + \texttt{0} + \texttt{1} \end{array}$$

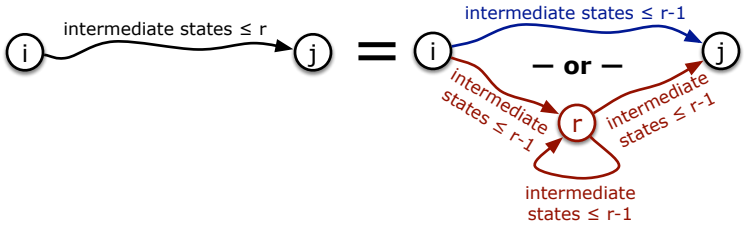


An example DFA

- Now suppose $r > 0$. Each string $w \in L(i,j,r)$ describes a walk from state i to state j where every intermediate state has index at most r. If this walk does not pass through state r, then $w \in L(i,j,r-1)$ by definition. Otherwise, we can split w into a sequence of substrings $w = w_1 \cdot w_2 \cdots w_\ell$ at the points where the walk visits state r. These substrings have the following properties:
  - The prefix w_1 describes a walk from state i to state r and thus belongs to $L(i,r,r-1)$.
  - The suffix w_ℓ describes a walk from state r to state j and thus belongs to $L(r,j,r-1)$.
  - For every other index k, the substring w_k describes a walk from state r to state r and thus belongs to $L(r,r,r-1)$.

  We conclude that

$$L(i,j,r) \;=\; L(i,j,r-1) \;\cup\; L(i,r,r-1) \bullet L(r,r,r-1)^* \bullet L(r,j,r-1).$$



Recursive structure of the regular language $L(i,j,r)$.

Putting these pieces together, we can recursively define a regular ***expression*** $R(i,j,r)$ that describes the language $L(i,j,r)$, as follows:

$$R(i,j,r) := \begin{cases} \varepsilon + \sum_{\delta(i,a)=j} a & \text{if } r = 0 \text{ and } i = j \\ \sum_{\delta(i,a)=j} a & \text{if } r = 0 \text{ and } i \neq j \\ R(i,j,r-1) + R(i,r,r-1) \bullet R(r,r,r-1)^* \bullet R(r,j,r-1) & \text{otherwise} \end{cases}$$

Kleene's algorithm evaluates this recurrence bottom-up using the natural dynamic programming algorithm. We memoize the previous recurrence into a three-dimensional array $R[1..n, 1..n, 0..n]$, which we traverse by increasing r in the outer loop, and in arbitrary order in the inner two loops.

```
KLEENE(Σ, n, δ, F):
  ⟨⟨Base cases⟩⟩
  for i ← 1 to n
      for j ← 1 to n
          if i = j then R[i, j, 0] ← ε else R[i, j, 0] ← ∅
          for all symbols a ∈ Σ
              if δ[i, a] = j
                  R[i, j, 0] ← R[i, j, 0] + a
  ⟨⟨Recursive cases⟩⟩
  for r ← 1 to n
      for i ← 1 to n
          for j ← 1 to n
              R[i, j, r] ← R[i, j, r − 1] + R[i, r, r − 1] • R[r, r, r − 1]* • R[r, j, r − 1]
  ⟨⟨Assemble the final result⟩⟩
  R ← ∅
  for q ← 0 to n − 1
      if q ∈ F
          R ← R + R[1, q, n − 1]
      return R
```

For purposes of analysis, let's assume the alphabet Σ has constant size. Assuming each alternation ($+$), concatenation ($\bullet$), and Kleene closure (*) operation requires constant time, the entire algorithm runs in $\boldsymbol{O(n^3)}$ ***time***.

However, regular expressions over an alphabet Σ are normally represented either as standard strings (arrays) over the larger alphabet $\Sigma \cup \{+, \bullet, *, (,), \varepsilon\}$, or as regular expression *trees*, whose internal nodes are $+$, $\bullet$, and $*$ operators and whose leaves are symbols and εs. In either representation, the regular expressions in Kleene's algorithm grow in size by roughly a factor of 4 in each iteration of the outer loop, at least in the worst case. Thus, in the worst case, each regular expression $R[i, j, r]$ has size $O(4^r)$, the size of the final output expression is $O(4^n n)$, and entire algorithm runs in $\boldsymbol{O(4^n n^2)}$ ***time***.

So we shouldn't do this. After all, the running time is exponential, and exponential time is bad. Right? Moreover, this exponential dependence is unavoidable; Hermann Gruber and Markus Holzer proved in 2008[4] that there are n-state DFAs over the binary alphabet $\{0, 1\}$ such that any equivalent regular expression has length $2^{\Omega(n)}$.

Well, maybe it's not so bad. The output regular expression has exponential size *because it contains multiple copies of the same subexpressions*; similarly, the regular expression tree has exponential size *because it contains multiples copies of several subtrees*. But it's precisely this exponential behavior that we use dynamic programming to avoid! In fact, it's not hard to modify Kleene's algorithm to compute a ***regular expression dag*** of size $O(n^3)$, ***in*** $\boldsymbol{O(n^3)}$ ***time***, that (intuitively) contains each subexpression $R[i, j, r]$ only once. This regular expression dag has exactly the same relationship to the regular expression *tree* as the dependency graph of Kleene's algorithm has to the recursion tree of its underlying recurrence.

Exercises

1. All of the algorithms discussed in this lecture fail if the graph contains a negative cycle. Johnson's algorithm detects the negative cycle in the initialization phase (via Shimbel's algorithm) 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.

[4] Hermann Gruber and Markus Holzer. Finite automata, digraph connectivity, and regular expression size. *Proc. 35th ICALP*, 39–50, 2008.

Specifically, if there is a path from vertex u to a negative cycle and a path from that negative cycle to vertex v, the algorithm should report that $dist[u, v] = -\infty$. If there is no directed path from u to v, the algorithm should return $dist[u, v] = \infty$. Otherwise, $dist[u, v]$ should equal the length of the shortest directed path from u to v.

(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 the Floyd-Warshall algorithm (FLOYDWARSHALL2) to return the correct shortest-path distances, even if the graph has negative cycles.

2. All of the shortest-path algorithms described in this note can also be modified to return an explicit description of some negative cycle, instead of simply reporting that a negative cycle exists.

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

   (b) Describe how to modify the Floyd-Warshall algorithm (FLOYDWARSHALL2) to return either the matrix of shortest-path distances or a negative cycle.

   If the graph contains more than one negative cycle, your algorithms may choose one arbitrarily.

3. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could 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:

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

   Describe an algorithm to compute $\delta(i, j)$ for all i and j in time $O(V^2 + kE \log E)$.

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

   (a) How could 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 $G' = (V', E')$ with weighted edges, where $V' = V \setminus \{v\}$, and the shortest-path distance between any two nodes in H is equal to the shortest-path distance between the same two nodes 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 from v to every other vertex, and from every other vertex to v, in the original graph G, 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 *not* the same as Floyd-Warshall!)

5. In this problem we will discover how you, too, 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 that 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 his 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 $\$ \to ¥ \to € \to \$$ 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.

*6. 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 using fast matrix multiplication. Assume that we have a subroutine MATRIXMULTIPLY that multiplies two $n \times n$ matrices in $\Theta(n^\omega)$ time, for some unknown constant $\omega \geq 2$.[5]

(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 $O(n^2)$ additional time.

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

(d) 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 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 of shortest-path distances in G in $O(n^\omega \log n)$ time.

[5] The matrix multiplication algorithm you already know runs in $\Theta(n^3)$ time, but this is not the fastest algorithm known. The current record is $\omega \approx 2.3727$, due to Virginia Vassilevska Williams. Determining the smallest possible value of ω is a long-standing open problem; many people believe there is an undiscovered $O(n^2)$-time algorithm for matrix multiplication.