## All-Pairs Shortest Paths

Input: Directed graph G = (V,E) with integer edge weights w(e), but no negative weight cycles.

Output: For each u,v (- V, length of a "shortest" (minimum weight) path from u to v.

- Two natural ways to characterize subproblems: restrict number of edges in path or restrict possible vertices on path. We study the latter (known as Floyd-Warshall's algorithm).

- Step 0: Recursive structure.
  Number vertices {1,...,n}. Consider min weight u-v path P. G contains no negative weight cycle => P is simple (no cycle). Let k = largest intermediate vertex on u-v path (intermediate = other than endpoints; k = 0 if P = (u,v)).

  Claim: P = shortest u-k path with intermediates in {1,...,k-1} (P_uk) followed by shortest k-v path with intermediates in {1,...,k-1} (P_kv).

  Proof: For a contradiction, suppose P_1 is u-k path with intermediates in {1,...,k-1} and weight(P_1) < weight(P_uk) -- argument similar for other half of path.
  Either P_1, P_kv have no common intermediates, or they do.
  Case 1: If P_1 has no common intermediates with P_kv, then P_1 + P_kv is simple u-v path with smaller weight than P.
  Case 2: If P_1 shares an intermediate with P_kv, say j in {1,...,k-1}, then P_1 + P_kv is path with smaller weight than P but cycle: u -> j -> k -> j -> v. Because G contains no negative-weight cycle, j -> k -> j has non-negative weight so path u -> j -> v has smaller weight than P_1 + P_kv < weight(P).
  Contradiction either way because P is min weight u-v path.

- Step 1: Array definition.
  A[k,u,v] = min weight of u-v paths with intermediates in {1,...,k}, for u,v in {1,...,n} and k in {0,1,...,n} (to restrict to paths with no intermediate nodes when k = 0).

- Step 2: Recurrence.

```
A[0,u,u] = 0 for all u in {1,...,n}
A[0,u,v] = w(u,v) for all (u,v) in E; oo for all (u,v) not in E
    (Paths with no intermediates = single edges, when they exist;
    degenerate case when u = v.)

A[k,u,v] = min( A[k-1,u,v], A[k-1,u,k] + A[k-1,k,v] )
    for all k,u,v in {1,...,n}
    (Min-weight path either does not use intermediate k, or it
    does.)
```

- Step 3: Bottom-up algorithm.

```
# Base cases.
for u <- 1,...,n:
    for v <- 1,...,n:
        if u = v:             A[0,u,v] <- 0
        else if (u,v) in E:   A[0,u,v] <- w(u,v)
        else:                 A[0,u,v] <- oo
```

```
    # General case.
    for k <- 1,...,n:
        for u <- 1,...,n:
            for v <- 1,...,n:
                if A[k-1,u,k] + A[k-1,k,v] < A[k-1,u,v]:
                    A[k,u,v] <- A[k-1,u,k] + A[k-1,k,v]
                else:
                    A[k,u,v] <- A[k-1,u,v]
```

Runtime? Theta(n^3)
Space? Theta(n^3)

Observation: Space can be reduced.
For all t in {1,...,n}, A[k,t,k] = A[k-1,t,k] and A[k,k,t] = A[k-1,k,t] (because A[k,k,k] = 0). So dimension k can be omitted from array definition: simply overwrite previous values for each k. Careful: loop over k remains -- value of k needed to update array values.

Simplified algorithm:

```
    # Base cases.
    for u <- 1,...,n:
        for v <- 1,...,n:
            if u = v:                A[u,v] <- 0
            else if (u,v) in E:      A[u,v] <- w(u,v)
            else:                    A[u,v] <- oo
    # General case.
    for k <- 1,...,n:
        for u <- 1,...,n:
            for v <- 1,...,n:
                if A[u,k] + A[k,v] < A[u,v]:
                    A[u,v] <- A[u,k] + A[k,v]
```

- Step 4: Optimal answer.

  To reconstruct min-weight u-v path, must know largest intermediate vertex used to achieve weight A[u,v]. Use second array to remember (could also store a pair of values in each entry of A, and additional indexing to extract relevant members).

  Revised algorithm (from previous step):

```
    # Base cases.
    for u <- 1,...,n:
        for v <- 1,...,n:
            if u = v:                A[u,v] <- 0;       B[u,v] <- -1
            else if (u,v) in E:      A[u,v] <- w(u,v);  B[u,v] <- 0
            else:                    A[u,v] <- oo;      B[u,v] <- -1
    # General case.
    for k <- 1,...,n:
        for u <- 1,...,n:
            for v <- 1,...,n:
                if A[u,k] + A[k,v] < A[u,v]:
                    A[u,v] <- A[u,k] + A[k,v]
                    B[u,v] <- k
```

  Recursive algorithm to reconstruct path:

```
    Path(u,v):
        # Assumption: A, B arrays already computed and global.
```

```
        if B[u,v] > 0:
            return Path(u,B[u,v]) + Path(B[u,v],v)
        else if B[u,v] = 0:
            return [(u,v)]  # no intermediate = single edge
        else:
            return []  # no path
```

Runtime? Theta(n) -- every path contains <= n nodes, each recursive call adds one more intermediate node, max n recursive calls executed. All this in addition to Theta(n^3) for computing A, B, of course!

- Trace on input with V = {1,2,3,4},
  E: w(1,4) = 10; w(2,1) = 1; w(3,1) = 10; w(3,2) = 1; w(4,3) = -3.
  For clarity, oo in A and -1 in B left blank.
  Notice: for each k, row k and column k unchanged from k-1.

k = 0:

| A | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | | | 10 |
| 2 | 1 | 0 | | |
| 3 | 10 | 1 | 0 | |
| 4 | | | -3 | 0 |

| B | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | | | 0 |
| 2 | 0 | | | |
| 3 | 0 | 0 | | |
| 4 | | | 0 | |

k = 1:

| A | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | | | 10 |
| 2 | 1 | 0 | | 11 |
| 3 | 10 | 1 | 0 | 20 |
| 4 | | | -3 | 0 |

| B | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | | | 0 |
| 2 | 0 | | | 1 |
| 3 | 0 | 0 | | 1 |
| 4 | | | 0 | |

k = 2:

| A | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | | | 10 |
| 2 | 1 | 0 | | 11 |
| 3 | 2 | 1 | 0 | 12 |
| 4 | | | -3 | 0 |

| B | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | | | 0 |
| 2 | 0 | | | 1 |
| 3 | 2 | 0 | | 2 |
| 4 | | | 0 | |

k = 3:

| A | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | | | 10 |
| 2 | 1 | 0 | | 11 |
| 3 | 2 | 1 | 0 | 12 |
| 4 | -1 | -2 | -3 | 0 |

| B | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | | | 0 |
| 2 | 0 | | | 1 |
| 3 | 2 | 0 | | 2 |
| 4 | 3 | 3 | 0 | |

k = 4:

| A | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 8 | 7 | 10 |
| 2 | 1 | 0 | 8 | 11 |
| 3 | 2 | 1 | 0 | 12 |
| 4 | -1 | -2 | -3 | 0 |

| B | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | 4 | 4 | 0 |
| 2 | 0 | | 4 | 1 |
| 3 | 2 | 0 | | 2 |
| 4 | 3 | 3 | 0 | |

```
Path(2,3): B[2,3] = 4 > 0 so
  . Paths(2,4): B[2,4] = 1 > 0 so
      . Paths(2,1): B[2,1] = 0
        return [(2,1)]
      . Paths(1,4): B[1,4] = 0
        return [(1,4)]
    return [(2,1),(1,4)]
  . Paths(4,3): B[4,3] = 0
    return [(2,1),(1,4),(4,3)]
```

## Transitive Closure

Problem: For directed graph $G = (V,E)$ and every pair of vertices $u,v$ in $V$, determine whether or not $G$ contains a path from $u$ to $v$.

We already know many good algorithms for this: run BFS $n$ times; run DFS $n$ times; use shortest-path algorithm with all weights = 1. We explore a different approach that involves a few interesting techniques.

Start with adjacency matrix, with additional 1's along the diagonal (always possible to reach $u$ from $u$, for every $u$ in $V$); e.g.:

$$A = \begin{matrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \qquad \text{Now consider matrix } A^2 = \begin{matrix} 1 & 0 & 0 & 2 \\ 2 & 1 & 0 & 1 \\ 3 & 2 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Let's examine some entries.

- $A^2[2,4] = A[2,1]*A[1,4] + A[2,2]*A[2,4] + A[2,3]*A[3,4] + A[2,4]*A[4,4]$
  Q: What does this represent?
  A: $A[2,i]*A[i,4] = 1$ if edges $(2,i)$ and $(i,4)$ exist; 0 otherwise
  So $A^2[2,4] > 0$ iff there exists a path of length $\le 2$ from 2 to 4

Trick 1: simplify computation: replace multiplication by conjunction (logical $\wedge$) and addition by disjunction (logical $\vee$).

$$\text{For example: } A^2[2,4] = \begin{matrix} (A[2,1] \wedge A[1,4]) \vee \\ (A[2,2] \wedge A[2,4]) \vee \\ (A[2,3] \wedge A[3,4]) \vee \\ (A[2,4] \wedge A[4,4]) \end{matrix} \qquad A^2 = \begin{matrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Then $A^2[i,j] = 1$ iff there is some path of length $\le 2$ from $i$ to $j$.
So $A^n[i,j] = 1$ iff there is some path in $G$ from $i$ to $j$.

Advantages? Storing boolean matrix more space efficient than storing number matrix; boolean operations faster than arithmetic operations.

Trick 2: Instead of computing $A^2$, $A^3$, ..., $A^n$ one by one, use repeated squaring: compute $A^2$, $(A^2)^2 = A^4$, $(A^4)^2 = A^8$, ... Reach $A^m$ for $m \ge n$ in only $\log n$ steps. More precisely, this can be done recursively.

```
Power(A,n):
    if n = 1:
        return A
    else:
        B = Power(A,floor(n/2))
        if n odd:
            return B x B x A  # using boolean "multiplication" above
        else:
            return B x B  # using boolean "multiplication" above
```

Runtime? At most $\log n$ levels, at most 2 matrix "products" on each level: $\Theta(n^3 \log n)$ -- each boolean matrix product takes time $\Theta(n^3)$.

Trick 3: Divide-and-conquer can be used to speed up matrix multiplication to $O(n^{\log_2 7}) = O(n^{2.8...})$. This makes algorithm above take time $O(n^{2.81} \log n)$ -- strictly better than $O(n^3)$.

---

For Next Week

* Readings: Section 7.2
* Self-Test: Exercise 7.10