Shortest Paths in a Graph (cont’d)

Fundamental Algorithms
All-Pairs Shortest Paths

We now want to compute a table giving the length of the shortest path between any two vertices. (We also would like to get the shortest paths themselves.)

We could just call Dijkstra or Bellman-Ford $|V|$ times, passing a different source vertex each time.

It can be done in $\Theta(V^3)$, which seems to be as good as you can do on dense graphs.
Doing APSP with SSSP

Dijkstra would take time

$$\Theta(V \times V^2) = \Theta(V^3)$$ (standard version)

$$\Theta(V \times (V\lg V + E)) = \Theta(V^2\lg V + VE))$$
(modified, Fibonacci heaps),

but doesn’t work with negative-weight edges.

Bellman-Ford would take $$\Theta(V \times VE) = \Theta(V^2E)$$.
The Floyd-Warshall Algorithm

Represent the directed, edge-weighted graph in adjacency-matrix form.

$W =$ matrix of weights =

$$
\begin{bmatrix}
 w_{11} & w_{12} & w_{13} \\
 w_{21} & w_{22} & w_{23} \\
 w_{31} & w_{32} & w_{33}
\end{bmatrix}
$$

- $w_{ij}$ is the weight of edge $(i, j)$, or $\infty$ if there is no such edge.
- Return a matrix $D$, where each entry $d_{ij}$ is $\delta(i,j)$.
- Could also return a predecessor matrix, $P$, where each entry $\pi_{ij}$ is the predecessor of $j$ on the shortest path from $i$. 
Floyd-Warshall: Idea

Consider *intermediate vertices* of a path:

\[ i \rightarrow \ldots \rightarrow \ldots \rightarrow j \]

Say we know the length of the shortest path from \( i \) to \( j \) whose intermediate vertices are only those with numbers 1, 2, ..., \( k-1 \). Call this length \( d_{ij}^{(k-1)} \).

Now how can we extend this from \( k-1 \) to \( k \)? In other words, we want to compute \( d_{ij}^{(k)} \). Can we use \( d_{ij}^{(k-1)} \), and if so how?
Two possibilities:

1. Going through the vertex $k$ doesn’t help— the path through vertices 1...$k-1$ is still the shortest.

2. There is a shorter path consisting of two subpaths, one from $i$ to $k$ and one from $k$ to $j$. Each subpath passes only through vertices numbered 1 to $k-1$. 
Floyd-Warshall Idea,

Thus,

\[ d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \]

Also, \( d_{ij}^{(0)} = w_{ij} \)

(since there are no intermediate vertices.)

When \( k = |V| \), we’re done.
Dynamic Programming

Floyd-Warshall is a *dynamic programming* algorithm:

Compute and store solutions to sub-problems. Combine those solutions to solve larger sub-problems.

Here, the sub-problems involve finding the shortest paths through a subset of the vertices.
Code for Floyd-Warshall

Floyd-Warshall(W)

1. \( n \leftarrow \text{rows}[W] \) // number of vertices
2. \( D^{(0)} \leftarrow W \)
3. for \( k \leftarrow 1 \) to \( n \)
4. do for \( i \leftarrow 1 \) to \( n \)
5. do for \( j \leftarrow 1 \) to \( n \)
6. \( d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \)
7. return \( D^{(n)} \)

Running time: a zippy \( \theta(V^3) \). (Small constant of proportionality, because operations are simple.)
Example of Floyd-Warshall

\[ D^{(0)} = \begin{bmatrix} \infty & 4 & 7 \\ 1 & \infty & 2 \\ 6 & \infty & \infty \end{bmatrix} \]

\[ D^{(1)} = \begin{bmatrix} \infty & 4 & 7 \\ 1 & \infty & 2 \\ 6 & 10 & \infty \end{bmatrix} \]

\[ D^{(2)} = \begin{bmatrix} \infty & 4 & 6 \\ 1 & \infty & 2 \\ 6 & 10 & \infty \end{bmatrix} \]

\[ D^{(3)} = \begin{bmatrix} \infty & 4 & 6 \\ 1 & \infty & 2 \\ 6 & 10 & \infty \end{bmatrix} \]
Johnson’s Algorithm

Makes clever use of Bellman-Ford and Dijkstra to do All-Pairs-Shortest-Paths efficiently on sparse graphs.

Motivation: By running Dijkstra $|V|$ times, we could do APSP in time $\Theta(V^2 \lg V + V E \lg V)$ (Modified Dijkstra), or $\Theta(V^2 \lg V + V E)$ (Fibonacci Dijkstra). This beats $\Theta(V^3)$ (Floyd-Warshall) when the graph is sparse.

Problem: negative edge weights.
The Basic Idea

Reweight the edges so that:

1. No edge weight is negative.
2. Shortest paths are preserved. (A shortest path in the original graph is still one in the new, reweighted graph.)

An obvious attempt: subtract the minimum weight from all the edge weights. E.g. if the minimum weight is -2:

\[-2 - (-2) = 0\]
\[3 - (-2) = 5\]

etc.
Counterexample

Subtracting the minimum weight from every weight doesn’t work.

Consider:

Paths with more edges are unfairly penalized.
Johnson’s Insight

Add a vertex \( s \) to the original graph \( G \), with edges of weight 0 to each vertex in \( G \):

\[
\begin{array}{c}
S \\
0 \\
0 \\
0
\end{array}
\]

Assign new weights \( \hat{w} \) to each edge as follows:

\[
\hat{w}(u, v) = w(u, v) + \delta(s, u) - \delta(s, v)
\]
Question 1

Are all the $\hat{w}$’s non-negative? Yes:

$\delta(s, u) + w(u, v) \text{ must be } \geq \delta(s, v)$

Otherwise, $s \Rightarrow u \rightarrow v$ would be shorter than the shortest path from $s$ to $v$.

$\delta(s, u) + w(u, v) \geq \delta(s, v)$

Rewriting:

$w(u, v) + \delta(s, u) - \delta(s, v) \geq 0$

$\hat{w}(u, v)$
Question 2

Does the reweighting preserve shortest paths? Yes: Consider any path \( p = v_1, v_2, \ldots, v_k \)

\[
\hat{w}(p) = \sum_{i=1}^{k-1} \hat{w}(v_i, v_{i+1}) \\
= w(v_1, v_2) + \delta(s, v_1) - \delta(s, v_2) \\
+ w(v_2, v_3) + \delta(s, v_2) - \delta(s, v_3) \\
\vdots \\
+ w(v_{k-1}, v_k) + \delta(s, v_{k-1}) - \delta(s, v_k) \\
=w(p) + \underbrace{\delta(s, v_1) - \delta(s, v_k)}_{\text{A value that depends only on the endpoints, not on the path.}}
\]

In other words, we have adjusted the lengths of all paths by the same amount. So this will not affect the relative ordering of the paths—shortest paths will be preserved.
Question 3

How do we compute the $\delta(s, v)$’s?

Use Bellman-Ford.

This also tells us if we have a negative-weight cycle.
Johnson’s: Algorithm

1. Compute G’, which consists of G augmented with s and a zero-weight edge from s to every vertex in G.

2. Run Bellman-Ford(G’, w, s) to obtain the $\delta(s, v)$’s

3. Reweight by computing $\hat{w}$ for each edge

4. Run Dijkstra on each vertex to compute $\hat{\delta}$

5. Undo reweighting factors to compute $\delta$
Johnson's: CLRS

**JOHNSON(G)**

1. compute $G'$, where $V[G'] = V[G] \cup \{s\}$,
   
   $E[G'] = E[G] \cup \{(s, v) : v \in V[G]\}$, and
   
   $w(s, v) = 0$ for all $v \in V[G]$

2. if **BELLMAN-FORD**(G', w, s) = FALSE

3. then print "the input graph contains a negative-weight cycle"

4. else for each vertex $v \in V[G']$

5. do set $h(v)$ to the value of $\delta(s, v)$
   
   computed by the Bellman-Ford algorithm

6. for each edge $(u, v) \in E[G']$

7. do $\hat{w}(u, v) \leftarrow w(u, v) + h(u) - h(v)$

8. for each vertex $u \in V[G]$

9. do run **DIJKSTRA**(G, $\hat{w}$, u) to compute $\hat{\delta}(u, v)$ for all $v \in V[G]$

10. for each vertex $v \in V[G]$

11. do $d_{uv} \leftarrow \hat{\delta}(u, v) + h(v) - h(u)$

12. return $D$

8a-ShortestPathsMore
Johnson: reweighting

\[ \hat{w}(u, v) = w(u, v) + d(s, u) - d(s, v) \]
Johnson using Dijkstra
Johnson’s: Running Time

1. Computing $G'$: $\Theta(V)$
2. Bellman-Ford: $\Theta(VE)$
3. Reweighting: $\Theta(E)$
4. Running (Modified) Dijkstra: $\Theta(V^2\lg V + VE\lg V)$
5. Adjusting distances: $\Theta(V^2)$

Total is dominated by Dijkstra: $\Theta(V^2\lg V + VE\lg V)$