CMSC 451: Lecture 11
All-Pairs Shortest Paths and the Floyd-Warshall Algorithm

All-Pairs Shortest Paths: Earlier, we saw that Dijkstra's algorithm and the Bellman-Ford algorithm both solved the problem of computing shortest paths in graphs from a single source vertex. Suppose that we want instead to compute shortest paths between *all pairs* of vertices. We could do this applying either Dijkstra and/or Bellman-Ford using every vertex as a source (and this might be the fastest, especially if the graph is sparse). Today, we will consider an alternative approach, which is based on dynamic programming.

Let $G = (V, E)$ be a directed graph with edge weights. For each edge (u, v) E, let $w(u, v)$ denote its weight. As before, the *cost* of a path is the sum of its edge weights, and the *distance* between two vertices is the minimum cost of any path between u and v. We allow negative weight edges, but no negative-cost cycles. (Recall that a negative-cost cycle implies that the shortest path may not be defined, since you make the cost arbitrarily small by repeating the cycle.)

The algorithm that we will present is called the *Floyd-Warshall algorithm*, which was discovered independently at about the same time by Robert Floyd and Stephen Warshall. It runs in $O(n^3)$ time, where $n = |V|$. It dates back to the early 1960's. The idea is quite natural. It is closely related to Kleene's algorithm (which converts a finite-state automaton to a regular expression). The algorithm was actually discovered even earlier by a French computer scientiest Bernard Roy.

The algorithm can be adapted for use in a number of related applications as well.

Transitive Closure: (This was the application Warshall was interested in). You are given a *binary relation* R on a set X, by which we mean that R is a subset of ordered pairs $(x, y) \subseteq X \times X$. A relation is said to be *transitive* if for any $x, y, z \in X$, if $(x, y) \in R$ and $(y, z) \in R$ then $(x, z) \in R$. The *transitive closure* of R, denoted R^* is the smallest extension of R that is transitive.

We can think of (X, R) as a directed graph, where X are the vertices and the pairs of R are edges. The transitive closure of R is effectively the same as the *reachability* relation in this graph, that is, $(x, y) \in R^*$ if and only if there exists a path from x to y in R. The Floyd-Warshall algorithm can be modified to compute the transitive closure in time $O(n^3)$, where $n = |X|$.

All-Pairs Max-Capacity Paths: Let $G = (V, E)$ be a directed graph with positive edge capacities $c(u, v)$. Think of each edge as a pipe, and the capacity $c(u, v)$ as the amount of flow that can be pushed through this pipe per unit interval. The *capacity* of a path is the minimum capacity of any edge along the path. (Intuitively, the minimum capacity edge forms a bottleneck, which limits the total amount of flow along the path. By the way, the capacity of the trivial path from u to u is $+\infty$.)

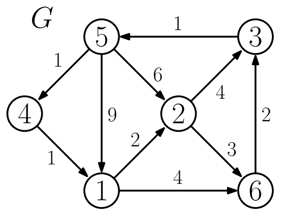
Given any two nodes u and v, we would like to know the maximum capacity path between them. It is easy to modify the Floyd-Warshall algorithm to compute the maximum capacity path between every pair of vertices in $O(n^3)$ time, where $n = |V|$.

Input/Output Representation: We assume that the digraph is represented as an adjacency matrix, rather than the more common adjacency list. (Adjacency lists are generally more efficient for sparse graphs, but storing all the inter-vertex distances will require $\Omega(n^2)$ storage anyway.) Because the algorithm is matrix-based, we will employ common matrix notation, using i, j and k to denote vertices rather than u, v, and w as we usually do.

The input is an $n \times n$ matrix w of edge weights, which are based on the edge weights in the digraph. We let w_{ij} denote the entry in row i and column j of w.

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ +\infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

(See Fig. 1(b).) Setting $w_{ij} = \infty$ if there is no edge, intuitively means that there is no direct link between these two nodes, and hence the direct cost is infinite. The reason for setting $w_{ii} = 0$ is that there is always a trivial path of length 0 (using no edges) from any vertex to itself.



(a)

$$\begin{array}{c|cccccc} w & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 2 & \infty & \infty & \infty & 4 \\ 2 & \infty & 0 & 4 & \infty & \infty & 3 \\ 3 & \infty & \infty & 0 & \infty & 1 & \infty \\ 4 & 1 & \infty & \infty & 0 & \infty & \infty \\ 5 & 9 & 6 & \infty & 1 & 0 & \infty \\ 6 & \infty & \infty & 2 & \infty & \infty & 0 \end{array}$$

(b)

$$\begin{array}{c|cccccc} d & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 2 & 6 & 8 & 7 & 4 \\ 2 & 7 & 0 & 4 & 6 & 5 & 3 \\ 3 & 3 & 5 & 0 & 4 & 1 & 7 \\ 4 & 1 & 3 & 7 & 0 & 8 & 5 \\ 5 & 1 & 4 & 8 & 1 & 0 & 6 \\ 6 & 5 & 7 & 2 & 4 & 3 & 0 \end{array}$$

(c)

Fig. 1: (a) A weighted digraph G, (b) the weight matrix w, and (c) the distance matrix d.

The output will be an $n \times n$ distance matrix $D = d_{ij}$ where $d_{ij} = \delta(i,j)$, the shortest path cost from vertex i to j (see Fig. 1(c)). This provides the distance, but how to get the shortest paths? You might think that that this will require $O(n^3)$ storage, since there are $\binom{n}{2} = O(n^2)$ pairs of vertices, and each shortest path might need up to $n-1$ edges. However, there is a cute trick for reducing the storage to $O(n^2)$. We will create an additional matrix $H = h_{ij}$ of "hooks," which is defined as follows.

$$h_{ij} \;=\; \begin{cases} \emptyset & \text{if } i = j \text{ or shortest path is single edge } (i,j) \\ k & \text{where } k \text{ is any vertex along shortest path.} \end{cases}$$

As we shall see below, this allows us to reconstruct a shortest path of any length, by repeatedly inserting a vertex into the path. (This exploits the fact that every subpath of a shortest path is also a shortest path.)

Floyd-Warshall Algorithm: As with any DP algorithm, the key is reducing a large problem to smaller problems. What will these subproblems be? A natural way of doing this is to follow an approach similar to that of Bellman-Ford, constructing shortest paths based on the *number of edges* in the shortest path. For example, we might define $d_{ij}^{(\ell)}$ to be the shortest path from i to j that uses at most ℓ edges. For the basis case, $d_{ij}^{(1)}$ would just be w_{ij}. Then, we could build up larger paths through repeated doubling. The shortest path using at most 2ℓ edges is built by concatenating two paths of length at most ℓ through all possible intermediate vertices k. For example, we might try the following:

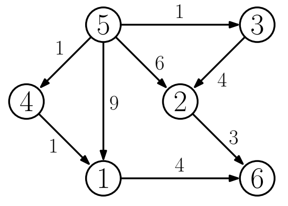
$$d_{ij}^{(2\ell)} = \min_{1 \le k \le n} \left(d_{ik}^{(\ell)} + d_{kj}^{(\ell)} \right). \qquad \text{(Possible DP alternative to Floyd-Warshall)}$$

(What would the running time of this approach be if implemented? I'll leave it as an exercise, but it will be *slower* than the Floyd-Warshall algorithm.)

Rather than restricting the *number* of edges on the path, the trick is to restrict the *set of intermediate vertices* that the path is allowed to use. Given a path $p = \langle v_1, v_2, \ldots, v_\ell \rangle$, we refer to the vertices $v_2, \ldots, v_{\ell-1}$ as the *intermediate vertices* of this path. Note that a path consisting of a single edge has no intermediate vertices.

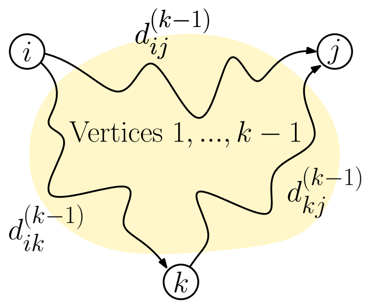
- Given $1, \le i, j \le n$, and $0 \le k \le n$, define $d_{ij}^{(k)}$ to be the shortest path from i to j such that any intermediate vertices on the path are chosen from the set $\{1, \ldots, k\}$.

In other words, we consider a path from i to j which either consists of the single edge (i, j), or it visits some intermediate vertices along the way, but these intermediate can only be chosen from among $\{1, \ldots, k\}$. (It does not need to visit all of these vertices, and it may visit none of them.) The path is free to visit any subset of these vertices, and to do so in any order. For example, in the digraph shown in the Fig. 2(a), notice how the value of $d_{5,6}^{(k)}$ changes as k varies.



$$d_{5,6}^{(0)} = \infty \text{ (no path)}$$
$$d_{5,6}^{(1)} = 13 \ \langle 5,1,6 \rangle$$
$$d_{5,6}^{(2)} = 9 \ \ \langle 5,2,6 \rangle$$
$$d_{5,6}^{(3)} = 8 \ \ \langle 5,3,2,6 \rangle$$
$$d_{5,6}^{(4)} = 6 \ \ \langle 5,4,1,6 \rangle$$
$$d_{5,6}^{(5)} = d_{5,6}^{(6)} = 6 \ \text{ (no change)}$$

(a)



(b)

Fig. 2: Limiting intermediate vertices. For example $d_{5,6}^{(3)}$ can go through any combination of the intermediate vertices $\{1, 2, 3\}$, of which $\langle 5, 3, 2, 6 \rangle$ has the lowest cost of 8.

How do we compute $d_{ij}^{(k)}$ assuming that we have already computed the previous matrix $d^{(k-1)}$? For the basis case ($k = 0$) the path cannot go through any intermediate vertices, and so $d_{ij}^{(0)} = w(i, j)$ for all i, j. For the induction step ($k \ge 1$), there are two cases, depending on the ways that we might get from vertex i to vertex j, assuming that the intermediate vertices are chosen from $\{1, 2, \ldots, k\}$:

Don't go through k at all: The shortest path from node i to node j uses intermediate vertices $\{1, \ldots, k-1\}$, and hence the length of the shortest path is $d_{ij}^{(k-1)}$.

Go through k: First observe that a shortest path does not pass through the same vertex twice, so we can assume that we pass through k exactly once. (The assumption that there are no negative cost cycles is being used here.) That is, we go from i to k, and then from k to j. In order for the overall path to be as short as possible we should take the shortest path from i to k, and the shortest path from k to j. Since both of these paths use intermediate vertices only in $\{1, \ldots, k-1\}$, the length of the path is $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.[1]

From the above discussion, we have the following recursive rule (the DP formulation) for computing $d^{(k)}$, which is illustrated in Fig. 2(b).

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1. \end{cases}$$

The final answer is $d_{ij}^{(n)}$, as this allows all possible vertices as intermediate vertices.

Bottom-up Implementation: As with other DP problems, a recursive implementation of this rule would be prohibitively slow because the same values may be reevaluated many times. While we could use memoization, we will present a bottom-up implementation.

We will maintain a matrix $d[1..n, 1..n]$. Because the function $d_{ij}^{(k)}$ involves three parameters, a faithful implementation of the above rule would involve a 3-dimensional array, $d[i, j, k]$. We shall see, however, that the algorithm will compute the same final result even if we "cheat" by ignoring the third (k) component.

As usual, we will store a set of "hooks" to save the decisions made. This takes the form of a parallel matrix, $H[1..n, 1..n]$. If we decide that it is best to go through vertex k, we will set $h[i, j] \leftarrow k$ to record this decision. Recall that it stores any vertex along the shortest path from i to j. If the path goes through k, then we can store k as the hook. The complete algorithm is presented in the code fragment below. An example of the algorithm's execution is shown in Fig. 3.

Clearly, the algorithm's running time is $O(n^3)$. The space used by the algorithm is $O(n^2)$.

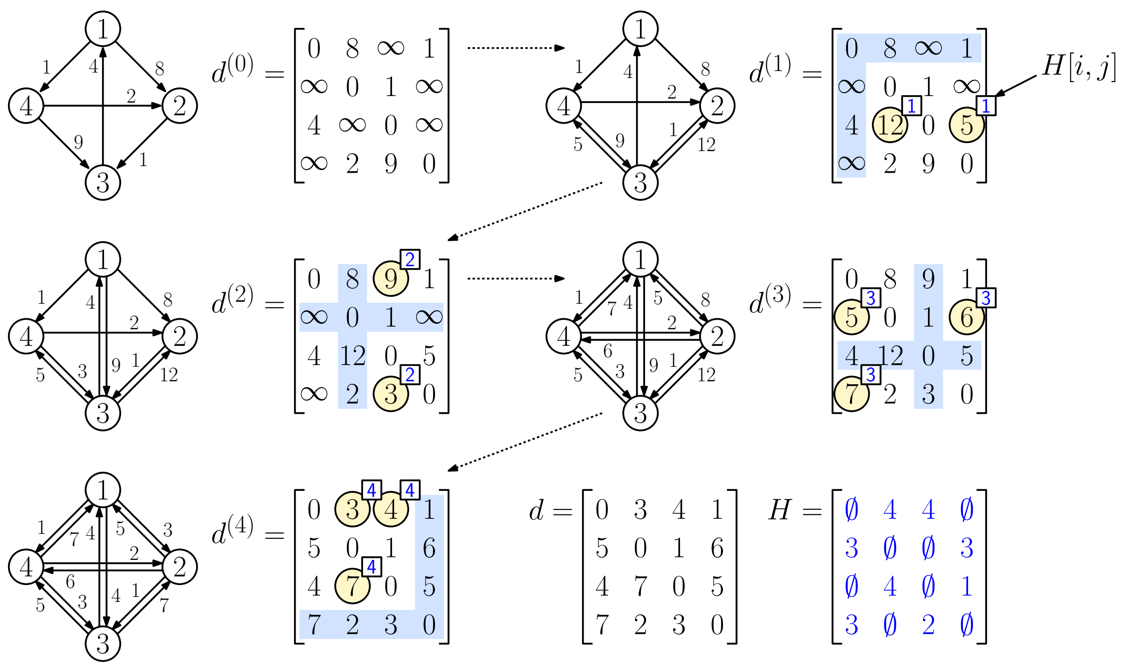
Overwriting Entries: Recall that a faithful would involve a 3-dimensional array $d[i, j, k]$, and we cheated by omitting the kth component. (Actually, we don't really need to store a separate matrix for every value of k, since at any time we are only using two matrices, one for the current value, k, and one for the previous value, $k-1$.) By omitting this third parameter, we might overwrite some entry $d[i, j]$ and later attempt to access its value. We will show that our laziness does not affect the algorithm's correctness.

Let us consider some iteration k. Overwriting is an issue whenever we alter the value of $d[i, j]$ by setting it to `newCost`, and later we attempt to read this same entry to compute the value of some other entry $d[i', j']$. Which entries are used to compute `newCost`? During iteration k, these are entries of the form $d[i, k]$ or $d[k, j]$. Thus, if entry $d[i, j]$ was overwritten and then

[1]Although the figure suggests that $i \neq j \neq k$, you should convince yourself that this holds even if some combination of i, j, and k are equal to each other. For example, if $j = k$, then $d_{kj}^{(k-1)} = d_{jj}^{(k-1)} = w(j, j) = 0$, and so the formula degenerates to $d_{ik}^{(k-1)} + d_{kj}^{(k-1)} = d_{ik}^{(k-1)} = d_{ij}^{(k-1)}$.

Floyd-Warshall Shortest-Path Algorithm

```
floyd-warshall(w[1..n, 1..n]) {                  // all-pairs shortest distances
    for (i = 1 to n) {                           // initialization (k = 0)
        for (j = 1 to n) {
            d[i, j] = w[i, j]                    // initial distance is edge weight
            h[i, j] = null                       // no intermediate vertices
        }
    }
    for (k = 1 to n) {                           // add vertex k as a possible intermediate
        for (i = 1 to n) {                       // ...from i
            for (j = 1 to n) {                   // ...to j
                newCost = d[i, k] + d[k, j]      // cost if we go through k
                if (newCost < d[i, j]) {         // is it better?
                    d[i, j] = newCost            // update distance
                    h[i, j] = k                  // save k as new intermediate
                }
            }
        }
    }
    return d                                     // d[i,j] holds the distance from i to j
}
```



$d^{(0)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix}$
$d^{(1)} = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix}$
$H[i, j]$
$d^{(2)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix}$
$d^{(3)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$
$d^{(4)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$
$d = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$
$H = \begin{bmatrix} \emptyset & 4 & 4 & \emptyset \\ 3 & \emptyset & \emptyset & 3 \\ \emptyset & 4 & \emptyset & 1 \\ 3 & \emptyset & 2 & \emptyset \end{bmatrix}$

Fig. 3: Floyd-Warshall example. Newly updates entries are circled, and H matrix entries are shown in blue.

reused, it must either be that $d[i,j]$ and $d[i,k]$ are the same entry (implying that $j = k$) or $d[i,j]$ and $d[k,j]$ are the same entry (implying that $i = k$).

In summary, the only instances where overwriting $d[i,j]$ can cause an issue occurs either when $i = k$ or $j = k$. Let's see what happens in case when $i = k$.

$$\texttt{newCost} \;=\; d[i,k] + d[k,j] \;=\; d[i,i] + d[i,j] \;=\; d[i,j] \qquad (\text{since } d[i,i] = 0).$$

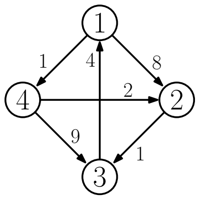
By inspecting the pseudocode, we see that the overwriting takes place only if newCost $< d[i,j]$. But since we have shown that newCost $= d[i,j]$, this clearly cannot happen. Thus, $d[i,j]$ is *not overwritten*, and there is no problem. (A similar argument applies for the $j = k$ case.) Thus, we pay no price for our laziness. (If only this were true for life in general!)

Extracting the Shortest Path: Let's next see how to use the hook values $h[i,j]$ to extract the shortest path. Recall that whenever we discover that the shortest path from i to j passes through an intermediate vertex k, we set $h[i,j] = k$. If the shortest path does not pass through any intermediate vertex, then $h[i,j] =$ null. To find the shortest path from i to j, we consult $h[i,j]$. If it is null, then the shortest path is just the direct path along edge (i,j). Otherwise, we recursively compute the shortest path from i to $h[i,j]$ and concatenate this with the shortest path from $h[i,j]$ to j.

If $d[i,j] = \infty$, there is no path (j is not reachable from i). Otherwise, the function get-path(i,j) shown in the following code block returns the sequence of edges along the shortest path. An example is presented in Fig. 4. Note that the running time is proportional to the number of edges on the path.

Printing the Shortest Path

```
get-path(i, j) {                        // return edges on the shortest path
    if (h[i, j] == null)                // no intermediate?
        return (i, j)                   // shortest path is a single edge
    else {                              // path goes through h[i,j]
        mid = h[i, j]                   // concatenate paths (i --> mid) + (mid --> j)
        return get-path(i, mid) + get-path(mid, j)
    }
}
```



$$H = \begin{bmatrix} \emptyset & 4 & 4 & \emptyset \\ 3 & \emptyset & \emptyset & 3 \\ \emptyset & 4 & \emptyset & 1 \\ 3 & \emptyset & 2 & \emptyset \end{bmatrix}$$

$$\begin{aligned} \texttt{get-path}(2,4): \;& \texttt{get-path}(2,3) \oplus \texttt{get-path}(3,4) \\ & (2,3) \oplus (\texttt{get-path}(3,1) \oplus \texttt{get-path}(1,4)) \\ & (2,3) \oplus (3,1) \oplus (1,4) \end{aligned}$$

Fig. 4: Extracting the shortest path from 2 to 4.

Summary: We have presented the Floyd-Warshall algorithm, an $O(n^3)$, DP-based algorithm for computing all-pairs shortest paths in a directed graph. The algorithm works even if G has negative cost edges, as long as there are no negative-cost cycles. The general algorithm structure can be used for a number of other applications, such as computing the transitive closure of a binary relation (which is equivalent to computing the reachability matrix in a digraph). It can be easily modified for computing a number of other all-pairs path-related problems as well.