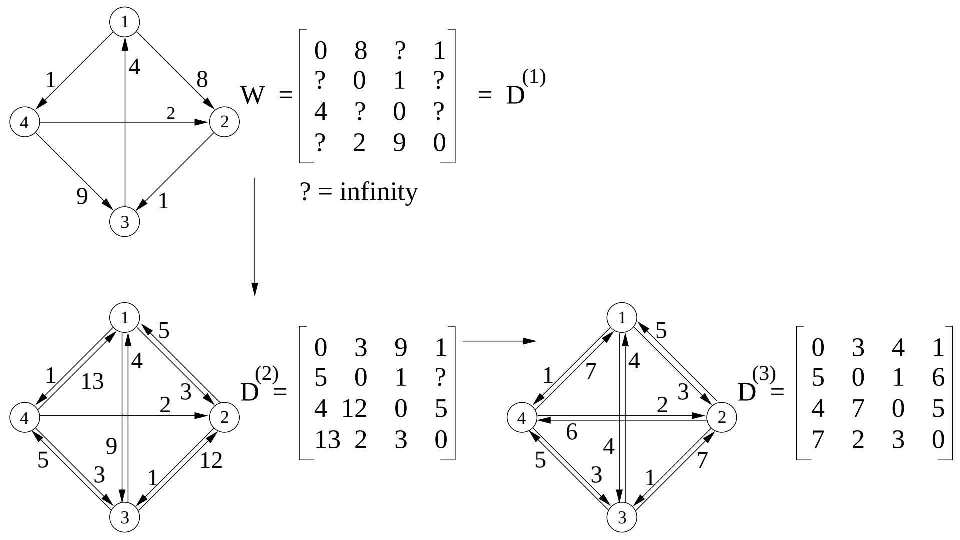


$$W = \begin{bmatrix} 0 & 8 & ? & 1 \\ ? & 0 & 1 & ? \\ 4 & ? & 0 & ? \\ ? & 2 & 9 & 0 \end{bmatrix} = D^{(1)}$$

? = infinity

$$D^{(2)} = \begin{bmatrix} 0 & 3 & 9 & 1 \\ 5 & 0 & 1 & ? \\ 4 & 12 & 0 & 5 \\ 13 & 2 & 3 & 0 \end{bmatrix} \qquad D^{(3)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

Figure 29: Shortest Path Example.

Lecture 24: Floyd-Warshall Algorithm

(Thursday, April 23, 1998)

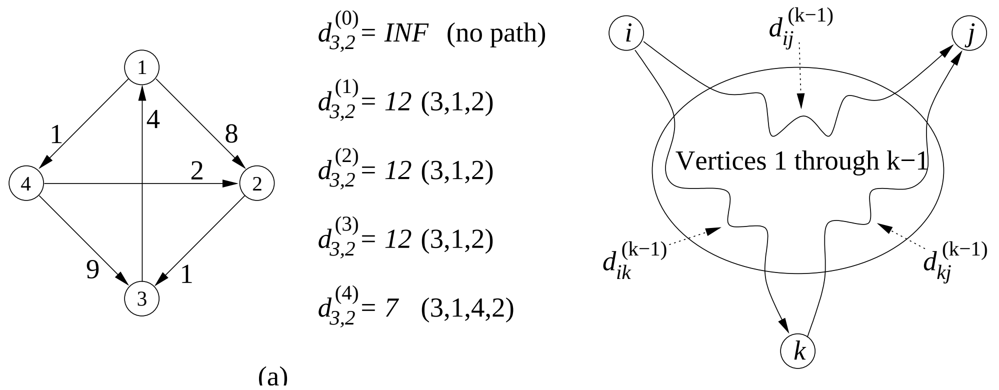
Read: Chapt 26 (up to Section 26.2) in CLR.

Floyd-Warshall Algorithm: We continue discussion of computing shortest paths between all pairs of vertices in a directed graph. The Floyd-Warshall algorithm dates back to the early 60's. Warshall was interested in the weaker question of reachability: determine for each pair of vertices u and v, whether u can reach v. Floyd realized that the same technique could be used to compute shortest paths with only minor variations.

The Floyd-Warshall algorithm improves upon this algorithm, running in $\Theta(n^3)$ time. The genius of the Floyd-Warshall algorithm is in finding a different formulation for the shortest path subproblem than the path length formulation introduced earlier. At first the formulation may seem most unnatural, but it leads to a faster algorithm. As before, we will compute a set of matrices whose entries are $d_{ij}^{(k)}$. We will change the *meaning* of each of these entries.

For a path $p = \langle v_1, v_2, \ldots, v_\ell \rangle$ we say that the vertices $v_2, v_3, \ldots, v_{\ell-1}$ are the *intermediate vertices* of this path. Note that a path consisting of a single edge has no intermediate vertices. We 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 $\{1, 2, \ldots, k\}$. 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 $\{1, 2, \ldots, k\}$. The path is free to visit any subset of these vertices, and to do so in any order. Thus, the difference between Floyd's formulation and the previous formulation is that here the superscript (k) restricts the set of vertices that the path is allowed to pass through, and there the superscript (m) restricts the number of edges the path is allowed to use. For example, in the digraph shown in the following figure, notice how the value of $d_{32}^{(k)}$ changes as k varies.

Floyd-Warshall Update Rule: How do we compute $d_{ij}^{(k)}$ assuming that we have already computed the previous matrix $d^{(k-1)}$? As before, there are two basic cases, depending on the ways that we might get from vertex i to vertex j, assuming that the intermediate vertices are chosen from $\{1, 2, \ldots, k\}$:



1
2
3
4
1
4
8
2
9
1
$d_{3,2}^{(0)}$ = INF (no path)
$d_{3,2}^{(1)}$ = 12 (3,1,2)
$d_{3,2}^{(2)}$ = 12 (3,1,2)
$d_{3,2}^{(3)}$ = 12 (3,1,2)
$d_{3,2}^{(4)}$ = 7 (3,1,4,2)
(a)
i
j
k
$d_{ij}^{(k-1)}$
Vertices 1 through k−1
$d_{ik}^{(k-1)}$
$d_{kj}^{(k-1)}$

Figure 30: Floyd-Warshall Formulation.

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

We do 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. (This is the principle of optimality.) Each of these paths uses intermediate vertices only in $\{1, 2, \ldots, k-1\}$. The length of the path is $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

This suggests the following recursive rule for computing $d^{(k)}$:

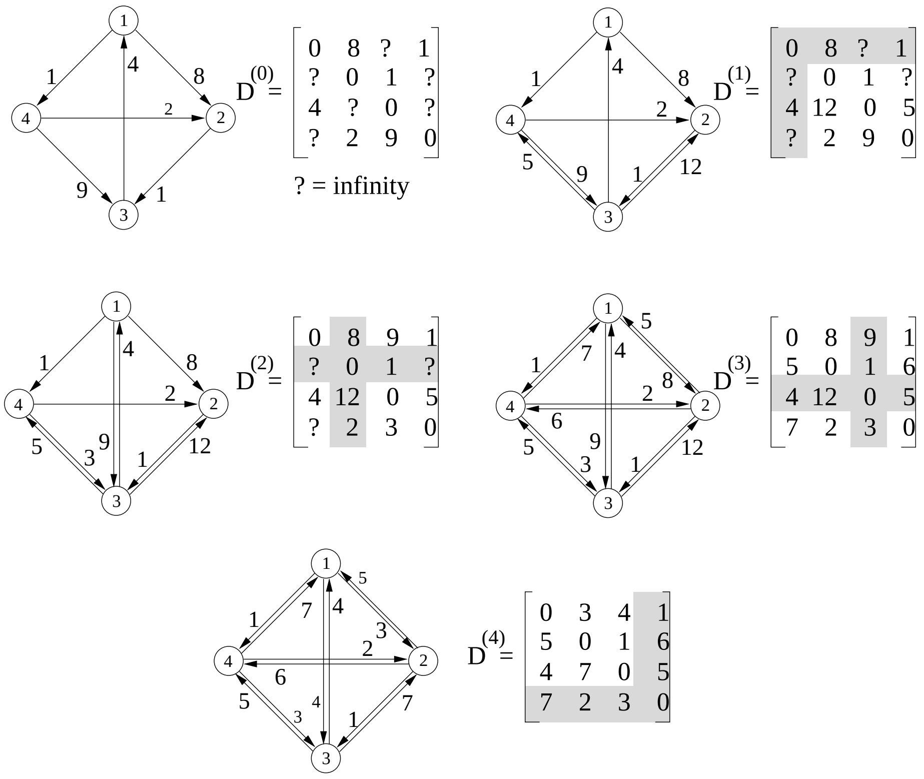
$$\begin{aligned} d_{ij}^{(0)} &= w_{ij}, \\ d_{ij}^{(k)} &= \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) \qquad \text{for } k \geq 1. \end{aligned}$$

The final answer is $d_{ij}^{(n)}$ because this allows all possible vertices as intermediate vertices. Again, we could write a recursive program to compute $d_{ij}^{(k)}$, but this will be prohibitively slow. Instead, we compute it by storing the values in a table, and looking the values up as we need them. Here is the complete algorithm. We have also included predecessor pointers, $pred[i, j]$ for extracting the final shortest paths. We will discuss them later.

Floyd-Warshall Algorithm

```
Floyd_Warshall(int n, int W[1..n, 1..n]) {
    array d[1..n, 1..n]
    for i = 1 to n do {                         // initialize
        for j = 1 to n do {
            d[i,j] = W[i,j]
            pred[i,j] = null
        }
    }
    for k = 1 to n do                           // use intermediates {1..k}
        for i = 1 to n do                       // ...from i
            for j = 1 to n do                   // ...to j
                if (d[i,k] + d[k,j]) < d[i,j]) {
                    d[i,j] = d[i,k] + d[k,j]    // new shorter path length
                    pred[i,j] = k               // new path is through k
                }
    return d                                        // matrix of final distances
}
```

Clearly the algorithm's running time is $\Theta(n^3)$. The space used by the algorithm is $\Theta(n^2)$. Observe that we deleted all references to the superscript (k) in the code. It is left as an exercise that this does not affect the correctness of the algorithm. An example is shown in the following figure.



$D^{(0)} = \begin{bmatrix} 0 & 8 & ? & 1 \\ ? & 0 & 1 & ? \\ 4 & ? & 0 & ? \\ ? & 2 & 9 & 0 \end{bmatrix}$
? = infinity
$D^{(1)} = \begin{bmatrix} 0 & 8 & ? & 1 \\ ? & 0 & 1 & ? \\ 4 & 12 & 0 & 5 \\ ? & 2 & 9 & 0 \end{bmatrix}$
$D^{(2)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ ? & 0 & 1 & ? \\ 4 & 12 & 0 & 5 \\ ? & 2 & 3 & 0 \end{bmatrix}$
$D^{(3)} = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$
$D^{(4)} = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$

Figure 31: Floyd-Warshall Example.

Extracting Shortest Paths: The predecessor pointers $pred[i, j]$ can be used to extract the final path. Here is the idea, whenever we discover that the shortest path from i to j passes through an intermediate vertex k, we set $pred[i, j] = k$. If the shortest path does not pass through any intermediate vertex, then $pred[i, j] = null$. To find the shortest path from i to j, we consult $pred[i, j]$. If it is *null*, then the shortest path is just the edge (i, j). Otherwise, we recursively compute the shortest path from i to $pred[i, j]$ and the shortest path from $pred[i, j]$ to j.

Printing the Shortest Path

```
    Path(i,j) {
        if pred[i,j] = null                     // path is a single edge
            output(i,j)
        else {                                  // path goes through pred
            Path(i, pred[i,j]);                 // print path from i to pred
            Path(pred[i,j], j);                 // print path from pred to j
        }
    }
```

Lecture 25: Longest Common Subsequence

(April 28, 1998)

Read: Section 16.3 in CLR.

Strings: One important area of algorithm design is the study of algorithms for character strings. There are a number of important problems here. Among the most important has to do with efficiently searching for a substring or generally a pattern in large piece of text. (This is what text editors and functions like ”grep” do when you perform a search.) In many instances you do not want to find a piece of text exactly, but rather something that is ”similar”. This arises for example in genetics research. Genetic codes are stored as long DNA molecules. The DNA strands can be broken down into a long sequences each of which is one of four basic types: C, G, T, A.

But exact matches rarely occur in biology because of small changes in DNA replication. Exact substring search will only find exact matches. For this reason, it is of interest to compute similarities between strings that do not match exactly. The method of string similarities should be insensitive to random insertions and deletions of characters from some originating string. There are a number of measures of similarity in strings. The first is the *edit distance*, that is, the minimum number of single character insertions, deletions, or transpositions necessary to convert one string into another. The other, which we will study today, is that of determining the length of the longest common subsequence.

Longest Common Subsequence: Let us think of character strings as sequences of characters. Given two sequences $X = \langle x_1, x_2, \ldots, x_m\rangle$ and $Z = \langle z_1, z_2, \ldots, z_k\rangle$, we say that Z is a *subsequence* of X if there is a strictly increasing sequence of k indices $\langle i_1, i_2, \ldots, i_k\rangle$ ($1 \le i_1 < i_2 < \ldots < i_k \le n$) such that $Z = \langle X_{i_1}, X_{i_2}, \ldots, X_{i_k}\rangle$. For example, let $X = \langle ABRACADABRA\rangle$ and let $Z = \langle AADAA\rangle$, then Z is a subsequence of X.

Given two strings X and Y, the *longest common subsequence* of X and Y is a longest sequence Z which is both a subsequence of X and Y.

For example, let X be as before and let $Y = \langle YABBADABBADOO\rangle$. Then the longest common subsequence is $Z = \langle ABADABA\rangle$.

The Longest Common Subsequence Problem (LCS) is the following. Given two sequences $X = \langle x_1, \ldots, x_m\rangle$ and $Y = \langle y_1, \ldots, y_n\rangle$ determine a longest common subsequence. Note that it is not always unique. For example the LCS of $\langle ABC\rangle$ and $\langle BAC\rangle$ is either $\langle AC\rangle$ or $\langle BC\rangle$.

Dynamic Programming Solution: The simple brute-force solution to the problem would be to try all possible subsequences from one string, and search for matches in the other string, but this is hopelessly inefficient, since there are an exponential number of possible subsequences.

Instead, we will derive a dynamic programming solution. In typical DP fashion, we need to break the problem into smaller pieces. There are many ways to do this for strings, but it turns out for this problem that considering all pairs of *prefixes* will suffice for us. A *prefix* of a sequence is just an initial string of values, $X_i = \langle x_1, x_2, \ldots, x_i\rangle$. X_0 is the empty sequence.

The idea will be to compute the longest common subsequence for every possible pair of prefixes. Let $c[i, j]$ denote the length of the longest common subsequence of X_i and Y_j. Eventually we are interested in $c[m, n]$ since this will be the LCS of the two entire strings. The idea is to compute $c[i, j]$ assuming that we already know the values of $c[i', j']$ for $i' \le i$ and $j' \le j$ (but not both equal). We begin with some observations.

Basis: $c[i, 0] = c[j, 0] = 0$. If either sequence is empty, then the longest common subsequence is empty.