**Floyd-Warshall Algorithm:** 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 runs in $\Theta(n^3)$ time.

As with any DP algorithm, the key is reducing a large problem to smaller problems. A natural way of doing this is by limiting the number of edges of the path, but it turns out that this does not lead to the fastest algorithm (but is an approach worthy of consideration). The main feature of the Floyd-Warshall algorithm is in finding a the best formulation for the shortest path subproblem. Rather than limiting the number of edges on the path, they instead limit the set of vertices through which the path is allowed to pass. In particular, for a path $p = (v_1, v_2, \ldots, v_k)$ we say that the vertices $v_2, v_3, \ldots, v_{k-1}$ are the intermediate vertices of this path. Note that a path consisting of a single edge has no intermediate vertices.

**Formulation:** 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 among $\{1, 2, \ldots, k\}$. 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. 41(a), notice how the value of $d_{5,6}^{(k)}$ changes as $k$ varies.

![Fig. 41: Limiting intermediate vertices. For example $d_{5,6}^{(3)}$ can go through any combination of the intermediate vertices $\{1, 2, 3\}$, of which $(5, 3, 2, 6)$ has the lowest cost of 8.](image)

**Floyd-Warshall Update Rule:** How do we compute $d_{ij}^{(k)}$ assuming that we have already computed the previous matrix $d^{(k-1)}$? 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 $\{1, 2, \ldots, k\}$:

- **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 $d_{ij}^{(k-1)}$.

- **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 $j$. Since of these paths uses intermediate vertices only in $\{1, 2, \ldots, k-1\}$, the length of the path is $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

© David M. Mount, 2004, Dept. of Computer Science, University of Maryland, College Park, MD, 20742. These lecture notes were prepared by David Mount for the course CMSC 451, Design and Analysis of Computer Algorithms, at the University of Maryland. Permission to use, copy, modify, and distribute these notes for educational purposes and without fee is hereby granted, provided that this copyright notice appear in all copies.
This suggests the following recursive rule (the DP formulation) for computing \( d^{(k)} \), which is illustrated in Fig. 41(b).

\[
\begin{align*}
    d^{(0)}_{ij} &= w_{ij}, \\
    d^{(k)}_{ij} &= \min\left(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj}\right) \quad \text{for } k \geq 1.
\end{align*}
\]

The final answer is \( d^{(n)}_{ij} \) because this allows all possible vertices as intermediate vertices. We could write a recursive program to compute \( d^{(k)}_{ij} \), but this will be prohibitively slow because the same value may be reevaluated many times. 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 mid-vertex pointers, \( \text{mid}[i,j] \) for extracting the final shortest paths. We will leave the extraction of the shortest path as an exercise.

---

**Floyd-Warshall Algorithm**

```java
FloydWarshall(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]
            mid[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] { // new shorter path length
                    d[i, j] = d[i, k] + d[k, j]
                    mid[i, j] = k
                }
            }
        }
    }
    return d // matrix of distances
}
```

---

An example of the algorithm's execution is shown in Fig. 42.

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. (Hint: The danger is that values may be overwritten and then reused later in the same phase. Consider which entries might be overwritten and then reused, they occur in row \( k \) and column \( k \). It can be shown that the overwritten values are equal to their original values.)
Fig. 42: Floyd-Warshall Example. Newly updates entries are circled.