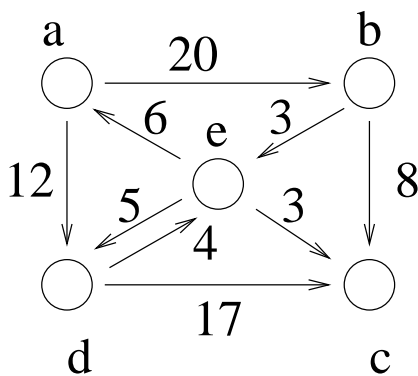
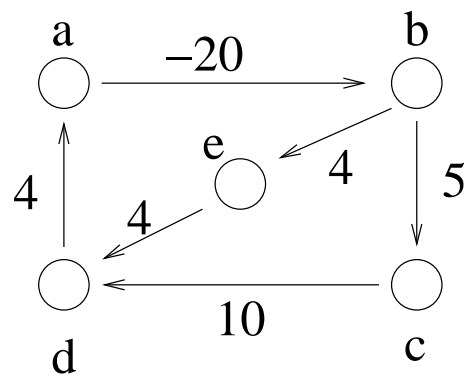


The All-Pairs Shortest Paths Problem

Given a weighted digraph $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}$, (\mathbb{R} is the set of real numbers), determine the **length of the shortest path** (i.e., **distance**) between all pairs of vertices in G . Here we assume that there are no cycles with **zero or negative cost**.



without negative cost cycle



with negative cost cycle

Solution 1: Using Dijkstra's Algorithm

If there are no negative cost edges apply Dijkstra's algorithm to each vertex (as the source) of the digraph.

- Recall that D's algorithm runs in $\Theta((n+e) \log n)$. This gives a

$$\Theta(n(n+e) \log n) = \Theta(n^2 \log n + ne \log n)$$

time algorithm, where $n = |V|$ and $e = |E|$.

- If the digraph is dense, this is an $\Theta(n^3 \log n)$ algorithm.
- With more advanced (complicated) data structures D's algorithm runs in $\Theta(n \log n + e)$ time yielding a $\Theta(n^2 \log n + ne)$ final algorithm. For dense graphs this is $\Theta(n^3)$ time.

Solution 2: Dynamic Programming

- (1)** How do we decompose the all-pairs shortest paths problem into subproblems?
- (2)** How do we express the optimal solution of a subproblem in terms of optimal solutions to some subsubproblems?
- (3)** How do we use the recursive relation from (2) to compute the optimal solution in a bottom-up fashion?
- (4)** How do we construct all the shortest paths?

Solution 2: Input and Output Formats

To simplify the notation, we assume that $V = \{1, 2, \dots, n\}$.

Assume that the graph is represented by an $n \times n$ matrix with the weights of the edges:

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

Output Format: an $n \times n$ matrix $D = [d_{ij}]$ where d_{ij} is the length of the shortest path from vertex i to j .

Step 1: How to Decompose the Original Problem

- Subproblems with smaller sizes should be easier to solve.
- An optimal solution to a subproblem should be expressed in terms of the optimal solutions to subproblems with smaller sizes.

These are guidelines ONLY.

Step 1: Decompose in a Natural Way

- Define $d_{ij}^{(m)}$ to be the length of the **shortest path** from i to j that **contains at most m edges**.
Let $D^{(m)}$ be the $n \times n$ matrix $[d_{ij}^{(m)}]$.
- $d_{ij}^{(n-1)}$ is the **true distance** from i to j (see next page for a proof this conclusion).
- **Subproblems:** compute $D^{(m)}$ for $m = 1, \dots, n-1$.

Question: Which $D^{(m)}$ is easiest to compute?

$$d_{ij}^{(n-1)} = \text{True Distance from } i \text{ to } j$$

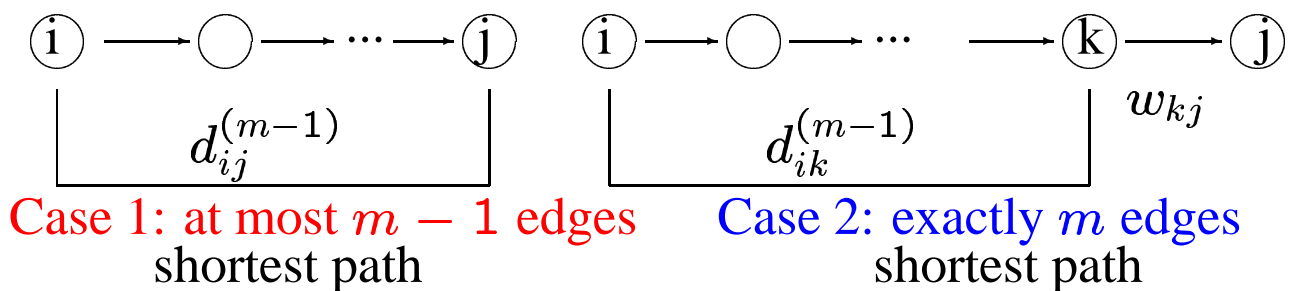
Proof: We prove that any shortest path P from i to j contains at most $n - 1$ edges.

First note that since all cycles have positive weight, **a shortest path can have no cycles** (if there were a cycle, we could remove it and lower the length of the path).

A path without cycles can have length at most $n - 1$ (since a longer path must contain some vertex twice, that is, contain a cycle).

A Recursive Formula

Consider a **shortest path** from i to j of length $d_{ij}^{(m)}$.



Case 1: It has at most $m - 1$ edges.

Then $d_{ij}^{(m)} = d_{ij}^{(m-1)} = d_{ij}^{(m-1)} + w_{jj}$.

Case 2: It has m edges. Let k be the vertex before j on a shortest path.

Then $d_{ij}^{(m)} = d_{ik}^{(m-1)} + w_{kj}$.

Combining the two cases,

$$d_{ij}^{(m)} = \min_{1 \leq k \leq n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}.$$

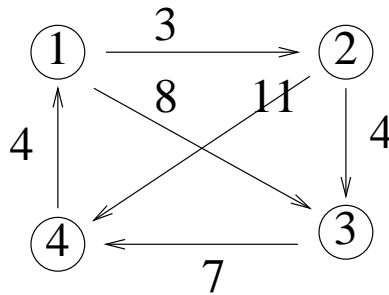
Step 3: Bottom-up Computation of $D^{(n-1)}$

- Bottom: $D^{(1)} = [w_{ij}]$, the weight matrix.
- Compute $D^{(m)}$ from $D^{(m-1)}$, for $m = 2, \dots, n-1$, using

$$d_{ij}^{(m)} = \min_{1 \leq k \leq n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}.$$

Example: Bottom-up Computation of $D^{(n-1)}$

Example

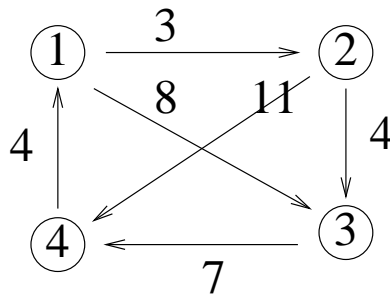


$D^{(1)} = [w_{ij}]$ is just the weight matrix:

$$D^{(1)} = \begin{bmatrix} 0 & 3 & 8 & \infty \\ \infty & 0 & 4 & 11 \\ \infty & \infty & 0 & 7 \\ 4 & \infty & \infty & 0 \end{bmatrix}$$

Example: Computing $D^{(2)}$ from $D^{(1)}$

$$d_{ij}^{(2)} = \min_{1 \leq k \leq 4} \left\{ d_{ik}^{(1)} + w_{kj} \right\}.$$

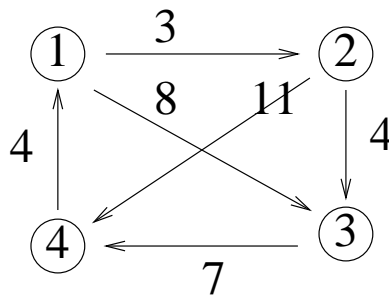


With $D^{(1)}$ given earlier and the recursive formula,

$$D^{(2)} = \begin{bmatrix} 0 & 3 & 7 & 14 \\ 15 & 0 & 4 & 11 \\ 11 & \infty & 0 & 7 \\ 4 & 7 & 12 & 0 \end{bmatrix}$$

Example: Computing $D^{(3)}$ from $D^{(2)}$

$$d_{ij}^{(3)} = \min_{1 \leq k \leq 4} \left\{ d_{ik}^{(2)} + w_{kj} \right\}$$



With $D^{(2)}$ given earlier and the recursive formula,

$$D^{(3)} = \begin{bmatrix} 0 & 3 & 7 & 14 \\ 15 & 0 & 4 & 11 \\ 11 & 14 & 0 & 7 \\ 4 & 7 & 11 & 0 \end{bmatrix}$$

$D^{(3)}$ gives the distances between any pair of vertices.

The Algorithm for Computing $D^{(n-1)}$

```
for  $m = 1$  to  $n - 1$ 
  for  $i = 1$  to  $n$ 
    for  $j = 1$  to  $n$ 
      {
         $min = \infty$ ;
        for  $k = 1$  to  $n$ 
          {
             $new = d_{ik}^{(m-1)} + w_{kj}$ ;
            if ( $new < min$ )  $min = new$ ;
          }
         $d_{ij}^{(m)} = min$ ;
      }
```

Comments on Solution 2

- Algorithm uses $\Theta(n^3)$ space; how can this be reduced down to $\Theta(n^2)$?
- How can we extract the actual shortest paths from the solution?
- Running time $O(n^4)$, much worse than the solution using Dijkstra's algorithm. Can we improve this?

Repeated Squaring

Observe that we are only interested to find $D^{(n-1)}$, all others D^i , $1 \leq i \leq n-2$ are only auxiliary. Furthermore, since the graph does not have negative cycle, we have $D^{(n-1)} = D^i$, for all $i \geq n$.

In particular, this implies that $D\left(2^{\lceil \log_2 n \rceil}\right) = D^{(n-1)}$.

We can calculate $D\left(2^{\lceil \log_2 n \rceil}\right)$ using “repeated squaring” to find

$$D^{(2)}, D^{(4)}, D^{(8)}, \dots, D\left(2^{\lceil \log_2 n \rceil}\right)$$

We use the recurrence relation:

- Bottom: $D^{(1)} = [w_{ij}]$, the weight matrix.
- For $s \geq 1$ compute $D^{(2s)}$ using

$$d_{ij}^{(2s)} = \min_{1 \leq k \leq n} \left\{ d_{ik}^{(s)} + d_{kj}^{(s)} \right\}.$$

Given this relation we can calculate $D^{(2^i)}$ from $D^{(2^{i-1})}$ in $O(n^3)$ time. We can therefore calculate **all** of

$$D^{(2)}, D^{(4)}, D^{(8)}, \dots, D^{(2^{\lceil \log_2 n \rceil})} = D^{(n)}$$

in $O(n^3 \log n)$ time, improving our running time.

The Floyd-Warshall Algorithm

Step 1 : Decomposition

Definition: The vertices v_2, v_3, \dots, v_{l-1} are called the *intermediate vertices* of the path $p = \langle v_1, v_2, \dots, v_{l-1}, v_l \rangle$.

- Let $d_{ij}^{(k)}$ be the **length of the shortest path** from i to j such that *all* intermediate vertices on the path (**if any**) are in set $\{1, 2, \dots, k\}$.

$d_{ij}^{(0)}$ is set to be w_{ij} , i.e., no intermediate vertex.

Let $D^{(k)}$ be the $n \times n$ matrix $[d_{ij}^{(k)}]$.

- Claim: $d_{ij}^{(n)}$ is the distance from i to j . So our aim is to compute $D^{(n)}$.
- **Subproblems:** compute $D^{(k)}$ for $k = 0, 1, \dots, n$.

Step 2: Structure of shortest paths

Observation 1: A shortest path does not contain the same vertex twice. Proof: A path containing the same vertex twice contains a cycle. Removing cycle gives a shorter path.

Observation 2: For a shortest path from i to j such that any intermediate vertices on the path are chosen from the set $\{1, 2, \dots, k\}$, there are two possibilities:

1. k is not a vertex on the path,

The shortest such path has length $d_{ij}^{(k-1)}$.

2. k is a vertex on the path.

The shortest such path has length $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

Step 2: Structure of shortest paths

Consider a **shortest path** from i to j containing the vertex k . It consists of a subpath from i to k and a subpath from k to j .

Each subpath can only contain intermediate vertices in $\{1, \dots, k-1\}$, and must be as short as possible, namely they have lengths $d_{ik}^{(k-1)}$ and $d_{kj}^{(k-1)}$.

Hence the path has length $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

Combining the two cases we get

$$d_{ij}^{(k)} = \min \left\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right\}.$$

Step 3: the Bottom-up Computation

- Bottom: $D^{(0)} = [w_{ij}]$, the weight matrix.
- Compute $D^{(k)}$ from $D^{(k-1)}$ using

$$d_{ij}^{(k)} = \min \left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right)$$

for $k = 1, \dots, n$.

The Floyd-Warshall Algorithm: Version 1

Floyd-Warshall(w, n)

```
{ for  $i = 1$  to  $n$  do                                initialize
    for  $j = 1$  to  $n$  do
        {  $D^0[i, j] = w[i, j];$ 
           $pred[i, j] = nil;$ 
        }

    for  $k = 1$  to  $n$  do                                dynamic programming
        for  $i = 1$  to  $n$  do
            for  $j = 1$  to  $n$  do
                if  $(d^{(k-1)}[i, k] + d^{(k-1)}[k, j] < d^{(k)}[i, j])$ 
                    {  $d^{(k)}[i, j] = d^{(k-1)}[i, k] + d^{(k-1)}[k, j];$ 
                       $pred[i, j] = k;$  }
                else  $d^{(k)}[i, j] = d^{(k-1)}[i, j];$ 
    return  $d^{(n)}[1..n, 1..n];$ 
}
```

Comments on the Floyd-Warshall Algorithm

- The algorithm's running time is clearly $\Theta(n^3)$.
- The predecessor pointer `pred[i, j]` can be used to extract the final path (see later).
- Problem: the algorithm uses $\Theta(n^3)$ space.
It is possible to reduce this down to $\Theta(n^2)$ space by keeping only one matrix instead of n .
Algorithm is on next page. Convince yourself that it works.

The Floyd-Warshall Algorithm: Version 2

Floyd-Warshall(w, n)

```
{ for  $i = 1$  to  $n$  do                                initialize
    for  $j = 1$  to  $n$  do
        {  $d[i, j] = w[i, j];$ 
           $pred[i, j] = nil;$ 
        }

    for  $k = 1$  to  $n$  do                                dynamic programming
        for  $i = 1$  to  $n$  do
            for  $j = 1$  to  $n$  do
                if ( $d[i, k] + d[k, j] < d[i, j]$ )
                    { $d[i, j] = d[i, k] + d[k, j];$ 
                      $pred[i, j] = k;$ }
    return  $d[1..n, 1..n];$ 
}
```

Extracting the Shortest Paths

The predecessor pointers $\text{pred}[i, j]$ can be used to extract the final path. The idea is as follows.

Whenever we discover that the shortest path from i to j passes through an intermediate vertex k , we set $\text{pred}[i, j] = k$.

If the shortest path does not pass through any intermediate vertex, then $\text{pred}[i, j] = \text{nil}$.

To find the shortest path from i to j , we consult $\text{pred}[i, j]$. If it is nil , then the shortest path is just the edge (i, j) . Otherwise, we recursively compute the shortest path from i to $\text{pred}[i, j]$ and the shortest path from $\text{pred}[i, j]$ to j .

The Algorithm for Extracting the Shortest Paths

```
Path( $i, j$ )
{
    if ( $pred[i, j] = nil$ )    single edge
        output ( $i, j$ );
    else    compute the two parts of the path
    {
        Path( $i, pred[i, j]$ );
        Path( $pred[i, j], j$ );
    }
}
```

Example of Extracting the Shortest Paths

Find the shortest path from vertex 2 to vertex 3.

2..3	Path(2, 3)	$pred[2, 3] = 4$	
2..4..3	Path(2, 4)	$pred[2, 4] = 5$	
2..5..4..3	Path(2, 5)	$pred[2, 5] = nil$	<i>Output(2,5)</i>
25..4..3	Path(5, 4)	$pred[5, 4] = nil$	<i>Output(5,4)</i>
254..3	Path(4, 3)	$pred[4, 3] = 6$	
254..6..3	Path(4, 6)	$pred[4, 6] = nil$	<i>Output(4,6)</i>
2546..3	Path(6, 3)	$pred[6, 3] = nil$	<i>Output(6,3)</i>
25463			