All-Pairs Shortest Paths

Version of November 5, 2014
A third example of dynamic programming
Will see **two** different dynamic programming formulations for same problem.

Outline

- The all-pairs shortest path problem.
- A first dynamic programming solution.
- The Floyd-Warshall algorithm
- Extracting shortest paths.
The All-Pairs Shortest Paths Problem

**Input:** weighted digraph \( G = (V, E) \) with weight function \( w : E \to \mathbb{R} \)

**Find:** lengths of the shortest paths (i.e., distance) between all pairs of vertices in \( G \).

- we assume that there are no cycles with zero or negative cost.
The All-Pairs Shortest Paths Problem

Input: weighted digraph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$

Find: lengths of the shortest paths (i.e., distance) between all pairs of vertices in $G$.

- 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

- Where there are no negative cost edges.
  - Apply Dijkstra’s algorithm $n$ times, once with each vertex (as the source) of the shortest path tree.
  - Recall that Dijkstra algorithm runs in $\Theta(e \log n)$
    - $n = |V|$ and $e = |E|$.
  - This gives a $\Theta(ne \log n)$ time algorithm
  - If the digraph is dense, this is a $\Theta(n^3 \log n)$ algorithm.

- When negative-weight edges are present:
  - Run the Bellman-Ford algorithm from each vertex.
  - $O(n^2 e)$, which is $O(n^4)$ for dense graphs.
  - We don’t learn Bellman-Ford in this class.
The all-pairs shortest path problem.
The all-pairs shortest path problem.

A first dynamic programming solution.
The all-pairs shortest path problem.
A first dynamic programming solution.
The Floyd-Warshall algorithm
Extracting shortest paths.
Input and Output Formats

Input Format:

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

- Adjacency matrix: graph is represented by an $n \times n$ matrix containing edge weights

$$w_{ij} = \begin{cases} 
0 & \text{if } i = j, \\
\infty & \text{if } i \neq j \text{ and } (i, j) \notin E.
\end{cases}$$
Input Format:
- To simplify the notation, we assume that $V = \{1, 2, \ldots, n\}$.
- Adjacency matrix: graph is represented by an $n \times n$ matrix containing edge weights

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

$$w(i, j) \quad \text{if } i \neq j \text{ and } (i, j) \in E,$$
Input Format:

- To simplify the notation, we assume that $V = \{1, 2, \ldots, n\}$.
- Adjacency matrix: graph is represented by an $n \times n$ matrix containing edge weights

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

Input Format:
- To simplify the notation, we assume that $V = \{1, 2, \ldots, n\}$.
- Adjacency matrix: graph is represented by an $n \times n$ matrix containing edge weights

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

Output Format: an $n \times n$ matrix $D = [d_{ij}]$ in which $d_{ij}$ is the length of the shortest path from vertex $i$ to $j$. 
For $m = 1, 2, 3 \ldots$, 

- Define $d_{ij}^{(m)}$ to be the length of the **shortest path** from $i$ to $j$ that **contains at most** $m$ edges.
Step 1: Space of Subproblems

For $m = 1, 2, 3 \ldots$,

- 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 $[d_{ij}^{(m)}]$. 

We will see (next page) that solution $D$ satisfies $D = D^{n-1}$. 

Subproblems: (Iteratively) compute $D^{(m)}$ for $m = 1, \ldots, n-1$. 

Version of November 5, 2014

All-Pairs Shortest Paths
Step 1: Space of Subproblems

For $m = 1, 2, 3 \ldots$,

- 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 $[d_{ij}^{(m)}]$.

We will see (next page) that solution $D$ satisfies $D = D^{n-1}$. 
For $m = 1, 2, 3 \ldots$, 

- 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 $[d_{ij}^{(m)}]$.

We will see (next page) that solution $D$ satisfies $D = D^{n-1}$.

**Subproblems:** (Iteratively) compute $D^{(m)}$ for $m = 1, \ldots, n - 1$. 
Step 1: Space of Subproblems

For $m = 1, 2, 3 \ldots$,

- 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 $[d_{ij}^{(m)}]$.

We will see (next page) that solution $D$ satisfies $D = D^{n-1}$.

**Subproblems:** (Iteratively) compute $D^{(m)}$ for $m = 1, \ldots, n - 1$. 
Step 1: Space of Subproblems

Lemma

\[ D^{(n-1)} = D \]
Step 1: Space of Subproblems

Lemma

\[ D^{(n-1)} = D, \text{ i.e.} \]
\[ d_{ij}^{(n-1)} = \text{true distance from } i \text{ to } j \]
Lemma

\[ D^{(n-1)} = D, \text{ i.e.} \]
\[ d_{ij}^{(n-1)} = \text{true distance from } i \text{ to } j \]

Proof.

We prove that any shortest path \( P \) from \( i \) to \( j \) contains at most \( n - 1 \) edges.
Step 1: Space of Subproblems

Lemma

\[ D^{(n-1)} = D, \text{ i.e.} \]
\[ d^{(n-1)}_{ij} = \text{true distance from } i \text{ to } j \]

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).
Step 1: Space of Subproblems

Lemma

\[ D^{(n-1)} = D, \text{ i.e.} \]
\[ d_{ij}^{(n-1)} = \text{true distance from } i \text{ to } j \]

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).
Step 2: Building $D^{(m)}$ from $D^{(m-1)}$.

Consider a shortest path from $i$ to $j$ that contains at most $m$ edges.

Let $k$ be the vertex immediately before $j$ on the shortest path ($k$ could be $i$).
Consider a shortest path from $i$ to $j$ that contains at most $m$ edges.

Let $k$ be the vertex immediately before $j$ on the shortest path ($k$ could be $i$).

The sub-path from $i$ to $k$ must be the shortest $1$-$k$ path with at most $m - 1$ edges. Then

$$d_{ij}^{(m)} = d_{ik}^{(m-1)} + w_{kj}.$$
Step 2: Building $D^{(m)}$ from $D^{(m-1)}$.

Consider a **shortest path** from $i$ to $j$ that contains at most $m$ edges.

![Diagram of shortest path](image)

Let $k$ be the vertex immediately before $j$ on the shortest path ($k$ could be $i$).

The sub-path from $i$ to $k$ must be the shortest 1-$k$ path with at most $m - 1$ edges. Then

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

Since we don’t know $k$, we try all possible choices: $1 \leq k \leq n$. 
Step 2: Building $D^{(m)}$ from $D^{(m-1)}$.

Consider a shortest path from $i$ to $j$ that contains at most $m$ edges.

Let $k$ be the vertex immediately before $j$ on the shortest path ($k$ could be $i$).

The sub-path from $i$ to $k$ must be the shortest $1$-$k$ path with at most $m-1$ edges. Then

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

Since we don’t know $k$, we try all possible choices: $1 \leq k \leq n$.

$$d_{ij}^{(m)} = \min_{1 \leq k \leq n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}$$
Step 3: Bottom-up Computation of $D^{(n-1)}$

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

$$d^{(m)}_{ij} = \min_{1 \leq k \leq n} \left\{ d^{(m-1)}_{ik} + w_{kj} \right\}$$
Example: Bottom-up Computation of $D^{(n-1)}$

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

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

$D^{(2)}$ gives the distances between any pair of vertices.
Example: Bottom-up Computation of $D^{(n-1)}$

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

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

$$d^{(2)}_{ij} = \min_{1 \leq k \leq 4} \left\{ d^{(1)}_{ik} + w_{kj} \right\}$$
Example: Bottom-up Computation of $D^{(n-1)}$

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

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

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

$$D^{(2)} = \begin{bmatrix}
0 & 3 & 7 & 14 \\
15 & 0 & 4 & 11 \\
11 & \infty & 0 & 7 \\
4 & 7 & 12 & 0
\end{bmatrix}$$
Example: Bottom-up Computation of $D^{(n-1)}$

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

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

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

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

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

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

$$D^{(3)} = \begin{bmatrix}
0 & 3 & 7 & 14 \\
15 & 0 & 4 & 11 \\
11 & 14 & 0 & 7 \\
4 & 7 & 11 & 0
\end{bmatrix}$$
Example: Bottom-up Computation of $D^{(n-1)}$

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

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

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

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

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

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

$D^{(3)}$ gives the distances between any pair of vertices.
\[ d_{ij}^{(m)} = \min_{1 \leq k \leq n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\} \]

```
for m = 1 to n - 1 do
    for i = 1 to n do
        for j = 1 to n do
            min = \infty;
            for k = 1 to n do
                new = d_{ik}^{(m-1)} + w_{kj};
                if new < min then
                    min = new
                end
            end
            d_{ij}^{(m)} = min;
        end
    end
end
```
Running time $O(n^4)$, much worse than the solution using Dijkstra’s algorithm.
Running time $O(n^4)$, much worse than the solution using Dijkstra’s algorithm.

**Question**

Can we improve this?
Repeated Squaring

We use the recurrence relation:

- Initialization: $D^{(1)} = [w_{ij}]$, the weight matrix.
Repeated Squaring

We use the recurrence relation:

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

\[
d^{(2s)}_{ij} = \min_{1 \leq k \leq n} \left\{ d^{(s)}_{ik} + d^{(s)}_{kj} \right\}
\]

From equation, can calculate $D^{(2i)}$ from $D^{(2i-1)}$ in $O(n^3)$ time.

We have shown earlier that the shortest path between any two vertices contains no more than $n - 1$ edges. So $D_i = D^{(n-1)}$ for all $i \geq n$.

We can therefore calculate all of $D^{(2)}, D^{(4)}, D^{(8)}, \ldots, D^{(2\lceil \log_2 n \rceil)} = D^{(n-1)}$ in $O(n^3 \log n)$ time, improving our running time.
Repeated Squaring

We use the recurrence relation:

- Initialization: $D^{(1)} = [w_{ij}]$, the weight matrix.
- For $s \geq 1$ compute $D^{(2s)}$ using
  \[
  d^{(2s)}_{ij} = \min_{1 \leq k \leq n} \left\{ d^{(s)}_{ik} + d^{(s)}_{kj} \right\}
  \]
- From equation, can calculate $D^{(2^i)}$ from $D^{(2^{i-1})}$ in $O(n^3)$ time.
Recurrent Squaring

We use the recurrence relation:

- **Initialization**: \( D^{(1)} = [w_{ij}] \), the weight matrix.
- For \( s \geq 1 \) compute \( D^{(2s)} \) using
  
  \[
  d^{(2s)}_{ij} = \min_{1 \leq k \leq n} \left\{ d^{(s)}_{ik} + d^{(s)}_{kj} \right\}
  \]

- From equation, can calculate \( D^{(2^i)} \) from \( D^{(2^{i-1})} \) in \( O(n^3) \) time.
- have shown earlier that the shortest path between any two vertices contains no more than \( n - 1 \) edges. So
  
  \[
  D^i = D^{(n-1)} \text{ for all } i \geq n.
  \]
Repeated Squaring

We use the recurrence relation:

- Initialization: \( D^{(1)} = [w_{ij}] \), the weight matrix.
- For \( s \geq 1 \) compute \( D^{(2s)} \) using
  \[
  d_{ij}^{(2s)} = \min_{1 \leq k \leq n} \left\{ d_{ik}^{(s)} + d_{kj}^{(s)} \right\}
  \]
- From equation, can calculate \( D^{(2^i)} \) from \( D^{(2^{i-1})} \) in \( O(n^3) \) time.
- have shown earlier that the shortest path between any two vertices contains no more than \( n - 1 \) edges. So
  \[
  D^i = D^{(n-1)} \quad \text{for all} \quad i \geq n
  \]

We can therefore calculate all of

\[
D^{(2)}, D^{(4)}, D^{(8)}, \ldots, D^{(2^{\lceil \log_2 n \rceil})} = D^{(n-1)}
\]

in \( O(n^3 \log n) \) time, improving our running time.
The all-pairs shortest path problem.
- The all-pairs shortest path problem.
- A first dynamic programming solution.
The all-pairs shortest path problem.
A first dynamic programming solution.
The Floyd-Warshall algorithm
Extracting shortest paths.
Step 1: Space of subproblems

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

For any $k=0, 1, \ldots, n$, 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 the set $\{1, 2, \ldots, k\}$.
Step 1: Space of subproblems

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

- For any $k=0, 1, \ldots, n$, 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 the set $\{1, 2, \ldots, k\}$. 

![Graph with shortest path lengths]
Step 1: Space of subproblems

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

\( \)
Step 1: Space of subproblems

- $d_{ij}^{(k)}$ is the length of the shortest path from $i$ to $j$ such that all intermediate vertices on the path (if any) are in the set $\{1, 2, \ldots, k\}$.
- Let $D^{(k)}$ be the $n \times n$ matrix $[d_{ij}^{(k)}]$. 

Version of November 5, 2014
Step 1: Space of subproblems

- $d_{ij}^{(k)}$ is the length of the shortest path from $i$ to $j$ such that all intermediate vertices on the path (if any) are in the set $\{1, 2, \ldots, k\}$.
- Let $D^{(k)}$ be the $n \times n$ matrix $[d_{ij}^{(k)}]$.
- **Subproblems:** compute $D^{(k)}$ for $k = 0, 1, \ldots, n$. 
Step 1: Space of subproblems

- \(d_{ij}^{(k)}\) is the **length of the shortest path** from \(i\) to \(j\) such that all intermediate vertices on the path (if any) are in the set \(\{1, 2, \ldots, k\}\).
- Let \(D^{(k)}\) be the \(n \times n\) matrix \([d_{ij}^{(k)}]\).
- **Subproblems**: compute \(D^{(k)}\) for \(k = 0, 1, \ldots, n\).
- **Original Problem**: \(D = D^{(n)}\), i.e. \(d_{ij}^{(n)}\) is the shortest distance from \(i\) to \(j\).
Step 2: Relating Subproblems

Observation: For a shortest path from \( i \) to \( j \) with intermediate vertices from the set \( \{1, 2, \ldots, k\} \), there are two possibilities:

1. \( k \) is not a vertex on the path: 
   \[
   d(k)_{ij} = d(k-1)_{ij}
   \]

2. \( k \) is a vertex on the path: 
   \[
   d(k)_{ij} = d(k-1)_{ik} + d(k-1)_{kj}
   \] (Impossible for \( k \) to appear in path twice. Why?) So: 
   \[
   d(k)_{ij} = \min\{d(k-1)_{ij}, d(k-1)_{ik} + d(k-1)_{kj}\} \]
Observation: For a shortest path from $i$ to $j$ with intermediate vertices from the set \{1, 2, \ldots, k\}, there are two possibilities:

1. $k$ is not a vertex on the path:
Observation: For a shortest path from $i$ to $j$ with intermediate vertices from the set $\{1, 2, \ldots, k\}$, there are two possibilities:

1. $k$ is not a vertex on the path: $d^{(k)}_{ij} = d^{(k-1)}_{ij}$
Observation: For a shortest path from $i$ to $j$ with intermediate vertices from the set \{1, 2, \ldots, k\}, there are two possibilities:

1. $k$ is not a vertex on the path: \[ d^{(k)}_{ij} = d^{(k-1)}_{ij} \]

2. $k$ is a vertex on the path: \[ d^{(k)}_{ij} = d^{(k-1)}_{ik} + d^{(k-1)}_{kj} \]
Step 2: Relating Subproblems

Observation: For a shortest path from $i$ to $j$ with intermediate vertices from the set $\{1, 2, \ldots, k\}$, there are two possibilities:

1. $k$ is not a vertex on the path: $d_{ij}^{(k)} = d_{ij}^{(k-1)}$

2. $k$ is a vertex on the path: $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

(Impossible for $k$ to appear in path twice. Why?) So:

$$d_{ij}^{(k)} = \min \left\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right\}$$
Step 2: Relating Subproblems

Proof.

- Consider a shortest path from $i$ to $j$ with intermediate vertices from the set $\{1, 2, \ldots, k\}$. Either it contains vertex $k$ or it does not.
- If it does not contain vertex $k$, then its length must be $d_{ij}^{(k-1)}$.
- Otherwise, it contains vertex $k$, and we can decompose it into a subpath from $i$ to $k$ and a subpath from $k$ to $j$.
- Each subpath can only contain intermediate vertices in
Step 2: Relating Subproblems

Proof.

Consider a **shortest path** from $i$ to $j$ with intermediate vertices from the set $\{1, 2, \ldots, k\}$. Either it contains vertex $k$ or it does not.

- If it does not contain vertex $k$, then its length must be $d_{ij}^{(k-1)}$.
- Otherwise, it contains vertex $k$, and we can decompose it into a subpath from $i$ to $k$ and a subpath from $k$ to $j$.

Each subpath can only contain intermediate vertices in $\{1, \ldots, k - 1\}$, and must be as short as possible. Hence they have lengths
Proof.

- Consider a shortest path from $i$ to $j$ with intermediate vertices from the set \{1, 2, \ldots, k\}. Either it contains vertex $k$ or it does not.
- If it does not contain vertex $k$, then its length must be $d_{ij}^{(k-1)}$.
- Otherwise, it contains vertex $k$, and we can decompose it into a subpath from $i$ to $k$ and a subpath from $k$ to $j$.
- Each subpath can only contain intermediate vertices in \{1, \ldots, k - 1\}, and must be as short as possible. Hence they have lengths $d_{ik}^{(k-1)}$ and $d_{kj}^{(k-1)}$.
- Hence the shortest path from $i$ to $j$ has length $\min \left\{ d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right\}$. 
Step 3: Bottom-up Computation

- Initialization: $D^{(0)} = [w_{ij}]$, the weight matrix.

- Compute $D^{(k)}$ from $D^{(k-1)}$ using

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

  for $k = 1, \ldots, n$. 
The Floyd-Warshall Algorithm: Version 1

Floyd-Warshall\((w, n)\): \(d^{(k)}_{ij} = \)

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

\[
\text{for } i = 1 \text{ to } n
\]
\[
\text{for } j = 1 \text{ to } n
\]
\[
\text{pred}_{ij}^{(k)} = \text{nil}
\]

\[
\text{// initialize}
\]
\[
\text{end for}
\]
\[
\text{end for}
\]

\[
\text{for } k = 1 \text{ to } n
\]
\[
\text{for } i = 1 \text{ to } n
\]
\[
\text{for } j = 1 \text{ to } n
\]
\[
\text{if } d_{ik}^{(k-1)} + d_{kj}^{(k-1)} < d_{ij}^{(k-1)}
\]
\[
\text{then}
\]
\[
\text{pred}_{ij}^{(k)} = k
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
\[
\text{end for}
\]

\[
\text{end for}
\]

\[
\text{return } d_{ij}^{(n)}
\]
The Floyd-Warshall Algorithm: Version 1

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

for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $d^{(0)}[i,j] = w[i,j];$ $\text{pred}[i,j] = \text{nil};$ // initialize
    end
end

Version of November 5, 2014
The Floyd-Warshall Algorithm: Version 1

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

for $i = 1$ to $n$ do
  for $j = 1$ to $n$ do
    $d^{(0)}[i, j] = w[i, j]$; $\text{pred}[i, j] = \text{nil}$; // initialize
  end
end

// dynamic programming
for $k = 1$ to $n$ do
  for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
The Floyd-Warshall Algorithm: Version 1

**Floyd-Warshall**\((w, n)\): \(d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)\)

\[
\text{for } i = 1 \text{ to } n \text{ do}
\quad \text{for } j = 1 \text{ to } n \text{ do}
\quad \quad d^{(0)}[i, j] = w[i, j]; \ pred[i, j] = nil; // initialize
\quad \text{end}
\text{end}
\]

// dynamic programming
\[
\text{for } k = 1 \text{ to } n \text{ do}
\quad \text{for } i = 1 \text{ to } n \text{ do}
\quad \quad \text{for } j = 1 \text{ to } n \text{ do}
\quad \quad \quad \text{if } \left(d^{(k-1)}[i, k] + d^{(k-1)}[k, j] < d^{(k-1)}[i, j]\right) \text{ then}
\quad \quad \quad \quad d^{(k)}[i, j] = d^{(k-1)}[i, k] + d^{(k-1)}[k, j];
\quad \quad \quad \quad pred[i, j] = k;
\quad \quad \text{else}
\quad \quad \text{end}
\quad \text{end}
\text{end}
The Floyd-Warshall Algorithm: Version 1

Floyd-Warshall\((w, n)\):
\[
d_{ij}^{(k)} = \min \left( d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right)
\]

for \(i = 1\) to \(n\) do
  for \(j = 1\) to \(n\) do
    \(d^{(0)}[i, j] = w[i, j];\) pred\([i, j]\) = nil; // initialize
  end
end

// dynamic programming
for \(k = 1\) to \(n\) do
  for \(i = 1\) to \(n\) do
    for \(j = 1\) to \(n\) do
      if \(d^{(k-1)}[i, k] + d^{(k-1)}[k, j] < d^{(k-1)}[i, j]\) then
        \(d^{(k)}[i, j] = d^{(k-1)}[i, k] + d^{(k-1)}[k, j];\)
        pred\([i, j]\) = \(k\);
      else
        end
      \(d^{(k)}[i, j] = d^{(k-1)}[i, j];\)
    end
  end
end
return \(d^{(n)}\)
The Floyd-Warshall Algorithm: Version 1

Floyd-Warshall\((w, n)\): 
\[
\begin{align*}
    d_{ij}^{(k)} &= \min \left( d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right)
\end{align*}
\]

for \(i = 1\) to \(n\) do
  for \(j = 1\) to \(n\) do
    \(d^{(0)}[i, j] = w[i, j];\) \(\text{pred}[i, j] = \text{nil};\) // initialize
  end
end

// dynamic programming
for \(k = 1\) to \(n\) do
  for \(i = 1\) to \(n\) do
    for \(j = 1\) to \(n\) do
      if \(d^{(k-1)}[i, k] + d^{(k-1)}[k, j] < d^{(k-1)}[i, j]\) then
        \(d^{(k)}[i, j] = d^{(k-1)}[i, k] + d^{(k-1)}[k, j];\)
        \(\text{pred}[i, j] = k;\)
      else
        \(\text{end}\)
      end
      \(d^{(k)}[i, j] = d^{(k-1)}[i, j];\)
    end
  end
end

return \(d^{(n)}[1..n, 1..n]\)
The algorithm’s running time is clearly $\Theta(n^3)$. 
The algorithm’s running time is clearly $\Theta(n^3)$.

The predecessor pointer $\text{pred}[i, j]$ can be used to extract the shortest paths.
The algorithm’s running time is clearly $\Theta(n^3)$.

The predecessor pointer $\text{pred}[i, j]$ can be used to extract the shortest paths (see later).
The algorithm’s running time is clearly $\Theta(n^3)$.

The predecessor pointer $\text{pred}[i, j]$ can be used to extract the shortest paths (see later).

**Problem**: the algorithm uses $\Theta(n^3)$ space.
Comments on the Floyd-Warshall Algorithm

- The algorithm’s running time is clearly $\Theta(n^3)$.
- The predecessor pointer $\text{pred}[i, j]$ can be used to extract the shortest paths (see later).

**Problem:** the algorithm uses $\Theta(n^3)$ space.

- It is possible to reduce this to $\Theta(n^2)$ space by keeping only one matrix instead of $n$. 
• The algorithm’s running time is clearly $\Theta(n^3)$.
• The predecessor pointer $\text{pred}[i, j]$ can be used to extract the shortest paths (see later).

Problem: the algorithm uses $\Theta(n^3)$ space.
• It is possible to reduce this 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

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

Floyd-Warshall \((w, n)\)

\begin{verbatim}
for i = 1 to n do
    for j = 1 to n do
        d[i,j] = w[i,j]; pred[i,j] = nil; // initialize
    end
end
\end{verbatim}
The Floyd-Warshall Algorithm: Version 2

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

Floyd-Warshall\((w, n)\)

\[
\text{for } i = 1 \text{ to } n \text{ do}
\]
\[
\quad \text{for } j = 1 \text{ to } n \text{ do}
\]
\[
\quad \quad d[i, j] = w[i, j]; \ pred[i, j] = \text{nil}; // initialize
\]
\[
\quad \text{end}
\]
\[
\text{end}
\]

\// dynamic programming

\[
\text{for } k = 1 \text{ to } n \text{ do}
\]
\[
\quad \text{for } i = 1 \text{ to } n \text{ do}
\]
\[
\quad \quad \text{for } j = 1 \text{ to } n \text{ do}
\]
Floyd-Warshall Algorithm: Version 2

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

Floyd-Warshall(w, n)

\[
\text{for} \ i = 1 \ \text{to} \ n \ \text{do} \\
\quad \text{for} \ j = 1 \ \text{to} \ n \ \text{do} \\
\quad \quad d[i, j] = w[i, j]; \ pred[i, j] = \text{nil}; \ // \ initialize \\
\quad \text{end} \\
\text{end} \\
// \ dynamic \ programming \\
\text{for} \ k = 1 \ \text{to} \ n \ \text{do} \\
\quad \text{for} \ i = 1 \ \text{to} \ n \ \text{do} \\
\quad \quad \text{for} \ j = 1 \ \text{to} \ n \ \text{do} \\
\quad \quad \quad \text{if} \ (d[i, k] + d[k, j] < d[i, j]) \ \text{then} \\
\quad \quad \quad \quad d[i, j] = d[i, k] + d[k, j]; \\
\quad \quad \quad \quad pred[i, j] = k; \\
\quad \quad \text{end} \\
\quad \text{end} \\
\text{end} \\
\text{return} \ d[1..n, 1..n];
The all-pairs shortest path problem.
The all-pairs shortest path problem.
A first dynamic programming solution.
The all-pairs shortest path problem.
A first dynamic programming solution.
The Floyd-Warshall algorithm
Extracting shortest paths.
predecessor pointers $\text{pred}[i, j]$ can be used to extract the shortest paths.

Idea: 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 construct the shortest path from $i$ to $\text{pred}[i, j]$ and the shortest path from $\text{pred}[i, j]$ to $j$. 
predecessor pointers $\text{pred}[i, j]$ can be used to extract the shortest paths.

Idea:

- Whenever we discover that the shortest path from $i$ to $j$ passes through an intermediate vertex $k$, we set $\text{pred}[i, j] = k$. 
predecessor pointers $\text{pred}[i, j]$ can be used to extract the shortest paths.

**Idea:**

- 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 $\text{pred}[i, j] =$
predecessor pointers $\text{pred}[i, j]$ can be used to extract the shortest paths.

Idea:

- 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 $\text{pred}[i, j] = \text{nil}$. 
predecessor pointers \( \text{pred}[i, j] \) can be used to extract the shortest paths.

**Idea:**
- 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 \( \text{pred}[i, j] \).
predecessor pointers $\text{pred}[i, j]$ can be used to extract the shortest paths.

**Idea:**

- 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
predecessor pointers $\text{pred}[i, j]$ can be used to extract the shortest paths.

**Idea:**

- 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 $(i, j)$. 
predecessor pointers $\text{pred}[i, j]$ can be used to extract the shortest paths.

**Idea:**

- 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 $\text{pred}[i, j]$.
  1. If it is nil, then the shortest path is just the edge $(i, j)$.
  2. Otherwise, we recursively construct the shortest path from
predecessor pointers \( \text{pred}[i, j] \) can be used to extract the shortest paths.

**Idea:**

- 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 \( \text{pred}[i, j] \).
  1. If it is nil, then the shortest path is just the edge \((i, j)\).
  2. Otherwise, we recursively construct the shortest path from \( i \) to
predecessor pointers $\text{pred}[i, j]$ can be used to extract the shortest paths.

**Idea:**

- 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 construct the shortest path from $i$ to $\text{pred}[i, j]$ and the shortest path from
predecessor pointers \( \text{pred}[i, j] \) can be used to extract the shortest paths.

**Idea:**

- 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 \( \text{pred}[i, j] \).
  1. If it is nil, then the shortest path is just the edge \((i, j)\).
  2. Otherwise, we **recursively** construct the shortest path from \( i \) to \( \text{pred}[i, j] \) and the shortest path from \( \text{pred}[i, j] \) to
predecessor pointers $\text{pred}[i, j]$ can be used to extract the shortest paths.

Idea:

- 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 $\text{pred}[i, j]$.
  1. If it is nil, then the shortest path is just the edge $(i, j)$.
  2. Otherwise, we recursively construct 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($i,j$)

\[
\begin{align*}
\text{if } \text{pred}[i,j] = \text{nil} & \text{ then} \\
& \quad \text{// single edge} \\
& \quad \text{output } (i,j); \\
\text{else} \\
& \quad \text{// compute the two parts of the path} \\
& \quad \text{Path}(i, \text{pred}[i,j]); \\
& \quad \text{Path}( \text{pred}[i,j],j); \\
\text{end}
\end{align*}
\]
Find the shortest path from vertex 2 to vertex 3.
Example of Extracting the Shortest Paths

Find the shortest path from vertex 2 to vertex 3.

2..3 Path(2, 3) \( \text{pred}[2, 3] = 6 \)
Example of Extracting the Shortest Paths

Find the shortest path from vertex 2 to vertex 3.

2..3 Path(2, 3)  \( \text{pred}[2, 3] = 6 \)

2..6..3 Path(2, 6)  \( \text{pred}[2, 6] = 5 \)
Find the shortest path from vertex 2 to vertex 3.

2..3  Path(2, 3) $pred[2, 3] = 6$
2..6..3 Path(2, 6) $pred[2, 6] = 5$
2..5..6..3 Path(2, 5) $pred[2, 5] = \text{nil}$  Output
Find the shortest path from vertex 2 to vertex 3.

2..3 \hspace{1cm} \text{Path}(2, 3) \hspace{1cm} \text{pred}[2, 3] = 6

2..6..3 \hspace{1cm} \text{Path}(2, 6) \hspace{1cm} \text{pred}[2, 6] = 5

2..5..6..3 \hspace{1cm} \text{Path}(2, 5) \hspace{1cm} \text{pred}[2, 5] = \text{nil} \hspace{1cm} \text{Output}(2, 5)
Find the shortest path from vertex 2 to vertex 3.

2..3  Path(2, 3)  $\text{pred}[2, 3] = 6$
2..6..3 Path(2, 6)  $\text{pred}[2, 6] = 5$
2..5..6..3 Path(2, 5)  $\text{pred}[2, 5] = \text{nil}$  Output(2,5)
25..6..3 Path(5, 6)  $\text{pred}[5, 6] = \text{nil}$  Output
Example of Extracting the Shortest Paths

Find the shortest path from vertex 2 to vertex 3.

- 2..3 Path(2, 3) \( \text{pred}[2, 3] = 6 \)
- 2..6..3 Path(2, 6) \( \text{pred}[2, 6] = 5 \)
- 2..5..6..3 Path(2, 5) \( \text{pred}[2, 5] = \text{nil} \) Output(2,5)
- 25..6..3 Path(5, 6) \( \text{pred}[5, 6] = \text{nil} \) Output(5,6)
Example of Extracting the Shortest Paths

Find the shortest path from vertex 2 to vertex 3.

- Path(2, 3) \( \text{pred}[2, 3] = 6 \)
- Path(2, 6) \( \text{pred}[2, 6] = 5 \)
- Path(2, 5) \( \text{pred}[2, 5] = \text{nil} \) \( \text{Output}(2, 5) \)
- Path(5, 6) \( \text{pred}[5, 6] = \text{nil} \) \( \text{Output}(5, 6) \)
- Path(6, 3) \( \text{pred}[6, 3] = 4 \)
Example of Extracting the Shortest Paths

Find the shortest path from vertex 2 to vertex 3.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2..3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2..6..3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2..5..6..3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25..6..3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256..3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>256..4..3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

All-Pairs Shortest Paths
Find the shortest path from vertex 2 to vertex 3.

- Path(2, 3) \( \text{pred}[2, 3] = 6 \)
- Path(2, 6) \( \text{pred}[2, 6] = 5 \)
- Path(2, 5) \( \text{pred}[2, 5] = \text{nil} \) Output(2,5)
- Path(5, 6) \( \text{pred}[5, 6] = \text{nil} \) Output(5,6)
- Path(6, 3) \( \text{pred}[6, 3] = 4 \)
- Path(6, 4) \( \text{pred}[6, 4] = \text{nil} \) Output(6,4)
Find the shortest path from vertex 2 to vertex 3.

2..3 Path(2, 3) \( \text{pred}[2, 3] = 6 \)
2..6..3 Path(2, 6) \( \text{pred}[2, 6] = 5 \)
2..5..6..3 Path(2, 5) \( \text{pred}[2, 5] = \text{nil} \)  \( \text{Output}(2,5) \)
25..6..3 Path(5, 6) \( \text{pred}[5, 6] = \text{nil} \)  \( \text{Output}(5,6) \)
256..3 Path(6, 3) \( \text{pred}[6, 3] = 4 \)
256..4..3 Path(6, 4) \( \text{pred}[6, 4] = \text{nil} \)  \( \text{Output}(6,4) \)
2564..3 Path(4, 3) \( \text{pred}[4, 3] = \text{nil} \)  \( \text{Output} \)
Example of Extracting the Shortest Paths

Find the shortest path from vertex 2 to vertex 3.

2..3 Path(2, 3) \( \text{pred}[2, 3] = 6 \)

2..6..3 Path(2, 6) \( \text{pred}[2, 6] = 5 \)

2..5..6..3 Path(2, 5) \( \text{pred}[2, 5] = \text{nil} \) Output(2,5)

25..6..3 Path(5, 6) \( \text{pred}[5, 6] = \text{nil} \) Output(5,6)

256..3 Path(6, 3) \( \text{pred}[6, 3] = 4 \)

256..4..3 Path(6, 4) \( \text{pred}[6, 4] = \text{nil} \) Output(6,4)

2564..3 Path(4, 3) \( \text{pred}[4, 3] = \text{nil} \) Output(4,3)
Find the shortest path from vertex 2 to vertex 3.

2..3 Path(2, 3) \( \text{pred}[2, 3] = 6 \)
2..6..3 Path(2, 6) \( \text{pred}[2, 6] = 5 \)
2..5..6..3 Path(2, 5) \( \text{pred}[2, 5] = \text{nil} \) Output(2,5)
25..6..3 Path(5, 6) \( \text{pred}[5, 6] = \text{nil} \) Output(5,6)
256..3 Path(6, 3) \( \text{pred}[6, 3] = 4 \)
256..4..3 Path(6, 4) \( \text{pred}[6, 4] = \text{nil} \) Output(6,4)
2564..3 Path(4, 3) \( \text{pred}[4, 3] = \text{nil} \) Output(4,3)
25643