All-Pairs Shortest Paths

Version of November 5, 2014

Outline

- 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 $w : E \to \mathbb{R}$

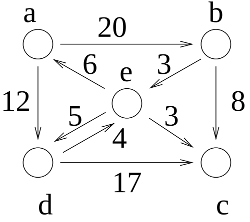
Find: lengths of the shortest paths (i.e., distance) between all pairs of vertices in G.

- we assume that there are no cycles with zero or negative cost.

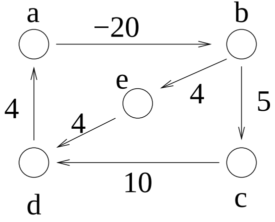
Input: weighted digraph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$

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.

Outline

- The all-pairs shortest path problem.

Outline

- The all-pairs shortest path problem.
- A first dynamic programming solution.

Outline

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

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

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ & \\ & \end{cases}$$

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

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

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

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

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.

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.

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

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 $m = 1, \ldots, n-1$.

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

$D^{(n-1)} = D$, *i.e.*
$d_{ij}^{(n-1)} =$ *true distance from* i *to* j

Step 1: Space of Subproblems

Lemma

$D^{(n-1)} = D$, *i.e.*
$d_{ij}^{(n-1)} =$ *true distance from i 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

$D^{(n-1)} = D$, *i.e.*
$d_{ij}^{(n-1)} =$ *true distance from* i *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

$D^{(n-1)} = D$, *i.e.*
$d_{ij}^{(n-1)} =$ *true distance from* i *to* j

Proof.

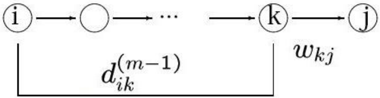
We prove that any shortest path P from i to j contains at most $n-1$ edges.

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). $\square$

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

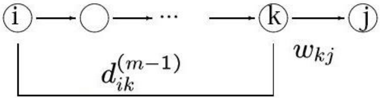
Consider a shortest path from i to j that contains at most m edges.



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

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

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

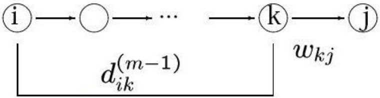


Let k be the vertex immediately before j on the shortest path (k could be i).

The sub-path from i to k must be the shortest1-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.

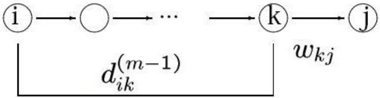


Let k be the vertex immediately before j on the shortest path (k could be i).

The sub-path from i to k must be the shortest1-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: $1 \le k \le n$.

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 shortest1-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: $1 \le k \le n$.

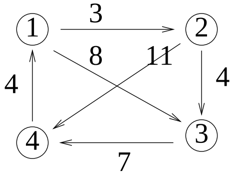
$$d_{ij}^{(m)} = \min_{1 \le k \le n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}$$

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

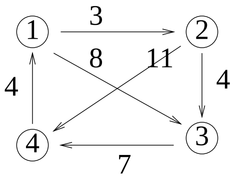
$$\boxed{d_{ij}^{(m)} = \min_{1 \le k \le n} \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}}$$

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



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

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

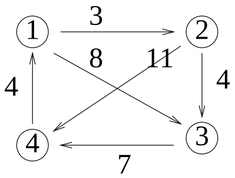


$$d_{ij}^{(2)} = \min_{1 \le k \le 4} \left\{ d_{ik}^{(1)} + w_{kj} \right\}$$

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

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

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

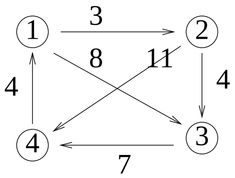


$$\color{red}{d_{ij}^{(2)}} = \min_{1 \le k \le 4} \left\{ d_{ik}^{(1)} + w_{kj} \right\}$$

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

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

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



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

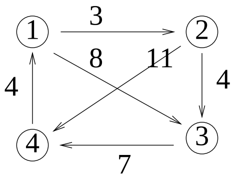
$$d_{ij}^{(2)} = \min_{1 \le k \le 4} \left\{ d_{ik}^{(1)} + w_{kj} \right\}$$

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

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

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

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



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

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

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

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

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

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

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

```
for m = 1 to n − 1 do
    for i = 1 to n do
        for j = 1 to n do
            min = ∞;
            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
```

Notes

Running time $O(n^4)$, much worse than the solution using Dijkstra's algorithm.

Notes

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

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

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 $D^{\left(2^i\right)}$ from $D^{\left(2^{i-1}\right)}$ in $O(n^3)$ time.

Repeated Squaring

We use the recurrence relation:

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

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

- From equation, can calculate $D^{\left(2^i\right)}$ from $D^{\left(2^{i-1}\right)}$ 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 $D^{\left(2^i\right)}$ from $D^{\left(2^{i-1}\right)}$ in $O(n^3)$ time.
- have shown earlier that the shortest path between any two vertices contains no more than $n-1$ edges. So

$$D^i = D^{(n-1)} \text{ for all } i \geq n.$$

We can therefore calculate all of

$$D^{(2)}, D^{(4)}, D^{(8)}, \ldots, D^{\left(2^{\lceil \log_2 n \rceil}\right)} = D^{(n-1)}$$

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

Outline

- The all-pairs shortest path problem.

Outline

- The all-pairs shortest path problem.
- A first dynamic programming solution.

Outline

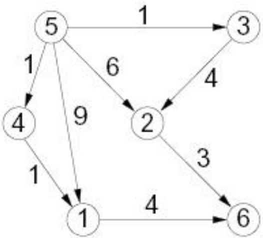
- The all-pairs shortest path problem.
- A first dynamic programming solution.
- The Floyd-Warshall algorithm
- Extracting shortest paths.

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

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, . . . , 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 $\{1, 2, \ldots, k\}$.

- The vertices $v_2, v_3, ..., v_{l-1}$ are called the intermediate vertices of the path $p = \langle v_1, v_2, ..., v_{l-1}, v_l \rangle$.
- For any k=0, 1, . . . , 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 $\{1, 2, \ldots, k\}$.



$d_{5,6}^{(0)}$ = INF (no path)

$d_{5,6}^{(1)}$ = 13 (5,1,6)

$d_{5,6}^{(2)}$ = 9 (5,2,6)

$d_{5,6}^{(3)}$ = 8 (5,3,2,6)

$d_{5,6}^{(4)}$ = 6 (5,4,1,6)

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 $\{1, 2, \ldots, k\}$.

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

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 $k = 0, 1, \cdots, n$.

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 $k = 0, 1, \cdots, n$.
- 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:

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:

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: $d_{ij}^{(k)} = d_{ij}^{(k-1)}$

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:

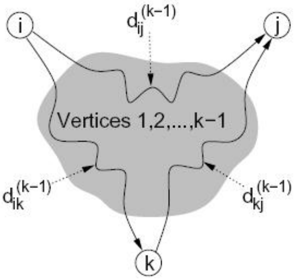
1. k is not a vertex on the path: $d_{ij}^{(k)} = d_{ij}^{(k-1)}$
2. k is a vertex on the path.: $d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

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

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 $\{1, ..., k-1\}$, and must be as short as possible. Hence they have lengths

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 $\{1, ..., k-1\}$, 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

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

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

The Floyd-Warshall Algorithm: Version 1

Floyd-Warshall(w, n): $d_{ij}^{(k)} =$

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
```

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**

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
```

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

```

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)[1..n, 1..n]
```

- The algorithm's running time is clearly $\Theta(n^3)$.

Comments on the Floyd-Warshall Algorithm

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

Comments on the Floyd-Warshall Algorithm

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

Comments on the Floyd-Warshall Algorithm

- The algorithm's running time is clearly $\Theta(n^3)$.
- The predecessor pointer $\texttt{pred}[\texttt{i}, \texttt{j}]$ 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 $\texttt{pred}[\texttt{i}, \texttt{j}]$ 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 $\texttt{pred}[\texttt{i}, \texttt{j}]$ 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.

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

Floyd-Warshall(w, n)

```
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
```

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

Floyd-Warshall(w, n)

```
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
// dynamic programming
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
```

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

Floyd-Warshall(w, n)

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

Outline

- The all-pairs shortest path problem.

Outline

- The all-pairs shortest path problem.
- A first dynamic programming solution.

Outline

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

Extracting the Shortest Paths

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

Extracting the Shortest Paths

predecessor pointers $\mathtt{pred[i,j]}$ 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 $pred[i,j] = k$.

predecessor pointers $\texttt{pred}[\texttt{i}, \texttt{j}]$ 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 $pred[i, j] = k$.
- If the shortest path does not pass through any intermediate vertex, then $pred[i, j] =$

Extracting the Shortest Paths

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

Extracting the Shortest Paths

predecessor pointers $\texttt{pred}[\texttt{i}, \texttt{j}]$ 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 $pred[i, j] = k$.
- If the shortest path does not pass through any intermediate vertex, then $pred[i, j] = nil$.
- To find the shortest path from i to j, we consult $pred[i, j]$.

Extracting the Shortest Paths

predecessor pointers $\texttt{pred}[\texttt{i}, \texttt{j}]$ 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 $pred[i,j] = k$.
- If the shortest path does not pass through any intermediate vertex, then $pred[i,j] = nil$.
- To find the shortest path from i to j, we consult $pred[i,j]$.
  1. If it is nil, then the shortest path is

Extracting the Shortest Paths

predecessor pointers $\texttt{pred}[\texttt{i},\texttt{j}]$ 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 $pred[i,j] = k$.
- If the shortest path does not pass through any intermediate vertex, then $pred[i,j] = nil$.
- To find the shortest path from i to j, we consult $pred[i,j]$.
  1. If it is nil, then the shortest path is just the edge (i,j).

predecessor pointers `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 $pred[i, j] = k$.
- If the shortest path does not pass through any intermediate vertex, then $pred[i, j] = nil$.
- To find the shortest path from i to j, we consult $pred[i, j]$.
  1. If it is nil, then the shortest path is just the edge (i, j).
  2. Otherwise, we recursively construct the shortest path from

Extracting the Shortest Paths

predecessor pointers `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 $pred[i, j] = k$.
- If the shortest path does not pass through any intermediate vertex, then $pred[i, j] = nil$.
- To find the shortest path from i to j, we consult $pred[i, j]$.
  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

Extracting the Shortest Paths

predecessor pointers $\texttt{pred}[\texttt{i}, \texttt{j}]$ 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 $pred[i, j] = k$.
- If the shortest path does not pass through any intermediate vertex, then $pred[i, j] = nil$.
- To find the shortest path from i to j, we consult $pred[i, j]$.
    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 $pred[i, j]$ and the shortest path from

Extracting the Shortest Paths

predecessor pointers $\texttt{pred}[\texttt{i},\texttt{j}]$ 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 $pred[i,j] = k$.
- If the shortest path does not pass through any intermediate vertex, then $pred[i,j] = nil$.
- To find the shortest path from i to j, we consult $pred[i,j]$.
  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 $pred[i,j]$ and the shortest path from $pred[i,j]$ to

predecessor pointers $\texttt{pred}[\texttt{i},\texttt{j}]$ 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 $pred[i,j] = k$.
- If the shortest path does not pass through any intermediate vertex, then $pred[i,j] = nil$.
- To find the shortest path from i to j, we consult $pred[i,j]$.
  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 $pred[i,j]$ and the shortest path from $pred[i,j]$ to j.

Path(i, j)

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

Example of Extracting the Shortest Paths

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 $\quad\quad$ Path$(2,3)$ $\quad$ $pred[2,3] = 6$

Example of Extracting the Shortest Paths

Find the shortest path from vertex 2 to vertex 3.

2..3	Path$(2, 3)$	$pred[2, 3] = 6$
2..6..3	Path$(2, 6)$	$pred[2, 6] = 5$

Example of Extracting the Shortest Paths

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$(2,5)$	$pred[2,5] = nil$	*Output*

Example of Extracting the Shortest Paths

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)	$pred[2, 5] = nil$	$Output$(2,5)

Example of Extracting the Shortest Paths

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$(2,5)$	$pred[2,5] = nil$	*Output*(2,5)
25..6..3	Path$(5,6)$	$pred[5,6] = nil$	*Output*

Example of Extracting the Shortest Paths

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)	$pred[2,5] = nil$	*Output*(2,5)
25..6..3	Path(5, 6)	$pred[5,6] = nil$	*Output*(5,6)

Example of Extracting the Shortest Paths

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)	$pred[2, 5] = nil$	$Output(2,5)$
25..6..3	Path(5, 6)	$pred[5, 6] = nil$	$Output(5,6)$
256..3	Path(6, 3)	$pred[6, 3] = 4$	

Example of Extracting the Shortest Paths

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)	$pred[2,5] = nil$	*Output*(2,5)
25..6..3	Path(5, 6)	$pred[5,6] = nil$	*Output*(5,6)
256..3	Path(6, 3)	$pred[6,3] = 4$	
256..4..3	Path(6, 4)	$pred[6,4] = nil$	*Output*

Example of Extracting the Shortest Paths

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)	$pred[2, 5] = nil$	$Output(2,5)$
25..6..3	Path(5, 6)	$pred[5, 6] = nil$	$Output(5,6)$
256..3	Path(6, 3)	$pred[6, 3] = 4$	
256..4..3	Path(6, 4)	$pred[6, 4] = nil$	$Output(6,4)$

Example of Extracting the Shortest Paths

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)	$pred[2,5] = nil$	*Output*(2,5)
25..6..3	Path(5, 6)	$pred[5,6] = nil$	*Output*(5,6)
256..3	Path(6, 3)	$pred[6,3] = 4$	
256..4..3	Path(6, 4)	$pred[6,4] = nil$	*Output*(6,4)
2564..3	Path(4, 3)	$pred[4,3] = nil$	*Output*

Example of Extracting the Shortest Paths

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)	$pred[2,5] = nil$	$Output(2,5)$
25..6..3	Path(5, 6)	$pred[5,6] = nil$	$Output(5,6)$
256..3	Path(6, 3)	$pred[6,3] = 4$	
256..4..3	Path(6, 4)	$pred[6,4] = nil$	$Output(6,4)$
2564..3	Path(4, 3)	$pred[4,3] = nil$	$Output(4,3)$

Example of Extracting the Shortest Paths

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