Warshall’s Algorithm: Transitive Closure

• Computes the transitive closure of a relation
• (Alternatively: all paths in a directed graph)
• Example of transitive closure:
Warshall’s Algorithm

- Main idea: a path exists between two vertices $i, j$, iff
  - there is an edge from $i$ to $j$; or
  - there is a path from $i$ to $j$ going through vertex 1; or
  - there is a path from $i$ to $j$ going through vertex 1 and/or 2; or
  - there is a path from $i$ to $j$ going through vertex 1, 2, and/or 3; or
  ...
  - there is a path from $i$ to $j$ going through any of the other vertices

Warshall’s Algorithm

- On the $k^{th}$ iteration, the algorithm determine if a path exists between two vertices $i, j$ using just vertices among 1,...,$k$ allowed as intermediate

\[
R^{(k)}[i,j] = \begin{cases} 
R^{(k-1)}[i,j] & \text{(path using just 1 ,...,k-1)} \\
\text{or} & \\
(R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j]) & \text{(path from i to k and from k to i using just 1 ,...,k-1)}
\end{cases}
\]

$k^{th}$ iteration
**Warshall’s Algorithm: Transitive Closure**

**Warshall’s Algorithm (matrix generation)**

Recurrence relating elements $R^{(k)}$ to elements of $R^{(k-1)}$ is:

$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

It implies the following rules for generating $R^{(k)}$ from $R^{(k-1)}$:

**Rule 1** If an element in row $i$ and column $j$ is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.

**Rule 2** If an element in row $i$ and column $j$ is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row $i$ and column $k$ and the element in its column $j$ and row $k$ are both 1’s in $R^{(k-1)}$.

**FIGURE 8.2** (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.
Warshall’s Algorithm: Transitive Closure

\[
R^{(k-1)} = \begin{bmatrix}
& j & k \\
i & 1 & 0 \\
& & 1
\end{bmatrix} \quad \rightarrow \quad R^{(k)} = \begin{bmatrix}
& j & k \\
i & 1 & 1 \\
& & 1
\end{bmatrix}
\]

**FIGURE 8.3** Rule for changing zeros in Warshall’s algorithm

\[
\begin{align*}
R^{(0)} &= \begin{bmatrix}
a & b & c & d \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0
\end{bmatrix} \\
R^{(1)} &= \begin{bmatrix}
a & b & c & d \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0
\end{bmatrix} \\
R^{(2)} &= \begin{bmatrix}
a & b & c & d \\
0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1
\end{bmatrix} \\
R^{(3)} &= \begin{bmatrix}
a & b & c & d \\
0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1
\end{bmatrix} \\
R^{(4)} &= \begin{bmatrix}
a & b & c & d \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1
\end{bmatrix}
\]

Ones reflect the existence of paths with no intermediate vertices \((R^{(0)}\) is just the adjacency matrix); boxed row and column are used for getting \(R^{(1)}\).

Ones reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex \(a\) (note a new path from \(d\) to \(b\)); boxed row and column are used for getting \(R^{(2)}\).

Ones reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., \(a\) and \(b\) (note two new paths); boxed row and column are used for getting \(R^{(3)}\).

Ones reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., \(a, b,\) and \(c\) (no new paths); boxed row and column are used for getting \(R^{(4)}\).

**FIGURE 8.4** Application of Warshall’s algorithm to the digraph shown. New ones are in bold.
Warshall’s Algorithm (pseudocode and analysis)

**ALGORITHM**  \( Warshall(A[1..n, 1..n]) \)

//Implements Warshall’s algorithm for computing the transitive closure
//Input: The adjacency matrix \( A \) of a digraph with \( n \) vertices
//Output: The transitive closure of the digraph

\[
R^{(0)} \leftarrow A \\
\text{for } k \leftarrow 1 \text{ to } n \text{ do} \\
\quad \text{for } i \leftarrow 1 \text{ to } n \text{ do} \\
\quad \quad \text{for } j \leftarrow 1 \text{ to } n \text{ do} \\
\quad \quad \quad R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j] \text{ or } (R^{(k-1)}[i, k] \text{ and } R^{(k-1)}[k, j]) \\
\text{return } R^{(n)}
\]

**Time efficiency:** \( \Theta(n^3) \)

**Space efficiency:** Matrices can be written over their predecessors

Floyd’s Algorithm: All pairs shortest paths

**Problem:** In a weighted (di)graph, find shortest paths between every pair of vertices

**Same idea:** construct solution through series of matrices \( D^{(0)}, \ldots, D^{(n)} \) using increasing subsets of the vertices allowed as intermediate

**Example:**

![Graph Example](image)
**Floyd’s Algorithm (matrix generation)**

On the $k$-th iteration, the algorithm determines shortest paths between every pair of vertices $i, j$ that use only vertices among $1, \ldots, k$ as intermediate.

\[ D^{(k)}[i, j] = \min \{ D^{(k-1)}[i, j], \ D^{(k-1)}[i, k] + D^{(k-1)}[k, j] \} \]

**Floyd’s Algorithm: All pairs shortest paths**

![Diagram](image)

**FIGURE 8.5** (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.
FIGURE 8.7 Application of Floyd’s algorithm to the graph shown. Updated elements are shown in bold.

Floyd’s Algorithm (pseudocode and analysis)

**ALGORITHM** Floyd(W[1..n, 1..n])

//Implements Floyd’s algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths’ lengths
D ← W //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            \[ D[i, j] ← \min\{D[i, j], D[i, k] + D[k, j]\} \]
return D

Time efficiency: \( \Theta(n^3) \)

Space efficiency: Matrices can be written over their predecessors
Knapsack Problem by DP

Given $n$ items of integer weights: $w_1, w_2, \ldots, w_n$
values: $v_1, v_2, \ldots, v_n$
a knapsack of integer capacity $W$
find most valuable subset of the items that fit into the knapsack

Consider instance defined by first $i$ items and capacity $j$ ($j \leq W$).
Let $V[i,j]$ be optimal value of such instance. Then

$$V[i,j] = \begin{cases} 
V[i-1,j], & j - w_i \geq 0 \\
\max \{V[i-1,j], v_i + V[i-1,j-w_i]\} & j - w_i < 0
\end{cases}$$

Initial conditions: $V[0,j] = 0$ and $V[i,0] = 0$

Knapsack Problem by DP (example)

Example: Knapsack of capacity $W = 5$

<table>
<thead>
<tr>
<th>item</th>
<th>weight</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>$12</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>$10</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$20</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>$15</td>
</tr>
</tbody>
</table>

capacity $j$

0 1 2 3 4 5

$w_1 = 2, v_1 = 12$ 1
$w_2 = 1, v_2 = 10$ 2
$w_3 = 3, v_3 = 20$ 3
$w_4 = 2, v_4 = 15$ 4

Copyright © 2007 Pearson Addison-Wesley. All rights reserved
Knapsack Problem

\[ V[i, j] = \max (V[i - 1, j], V[i - 1, j - w_i] + v_i) \]

- object \( i \) not used
- object \( i \) used

\[ \begin{array}{cccccc} & 0 & j - w_i & j & W \\ 0 & 0 & 0 & 0 & 0 \\ i - 1 & 0 & V[i - 1, j - w_i] & V[i - 1, j] & \\ w_i & v_i & i & 0 & V[i, j] \\ n & 0 & \text{goal} \end{array} \]

**FIGURE 8.12** Table for solving the knapsack problem by dynamic programming

---

**FIGURE 8.13** Example of solving an instance of the knapsack problem by the dynamic programming algorithm
Knapsack Problem – Memory Function

- Implement the recurrence recursively
- Do not calculate a value if it is not needed
- Do not recalculate a value
- Row 0 and column 0 of V are initialized to 0, other entries are -1

MFKnapsack(i, j)
if V[i, j] < 0
  if j < w[i]
    value ← MFKnapsack(i – 1, j)
  else
    value ← max (MFKnapsack(i – 1, j),
                  MFKnapsack(i – 1, j – w[i]) + v[i])

V[i, j] ← value
return V[i, j]

Copyright © 2007 Pearson Addison-Wesley. All rights reserved

Knapsack Problem – Memory Function

<table>
<thead>
<tr>
<th></th>
<th>capacity j</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>0</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

FIGURE 8.14 Example of solving an instance of the knapsack problem by the memory function algorithm