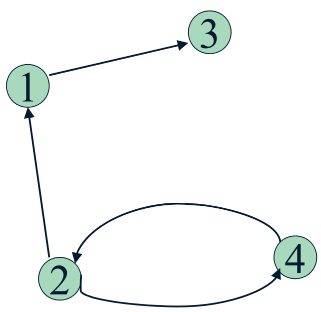
CS 440 Theory of Algorithms / CS 468 Algorithms in Bioinformatics

Dynamic Programming

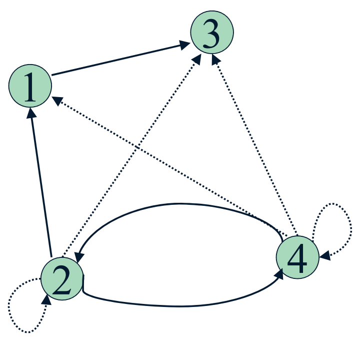
Part II

Warshall's Algorithm: Transitive Closure

- **Computes the transitive closure of a relation**
- **(Alternatively: all paths in a directed graph)**
- **Example of transitive closure:**



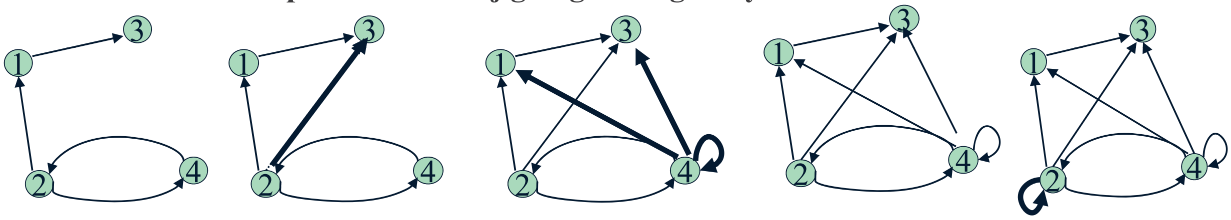
$$\begin{array}{cccc} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}$$



$$\begin{array}{cccc} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{array}$$

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

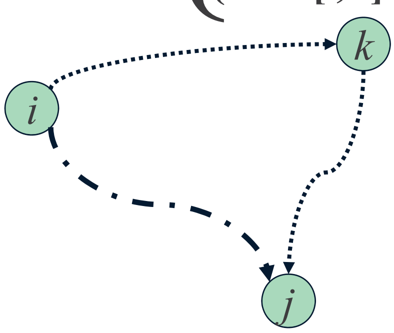


$$R_0 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix} \quad R_1 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix} \quad R_2 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix} \quad R_3 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix} \quad R_4 = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

Warshall's Algorithm

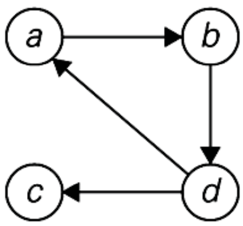
- On the k^{th} iteration, the algorithm determine if a path exists between two vertices *i, j* using just vertices among 1,…,*k* allowed as intermediate

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



k^{th} iteration

Warshall's Algorithm: Transitive Closure



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \qquad T = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

(a) (b) (c)

FIGURE 8.2 (a) Digraph. (b) Its adjacency matrix. (c) Its 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 $R^{(k-1)}$

Warshall's Algorithm: Transitive Closure

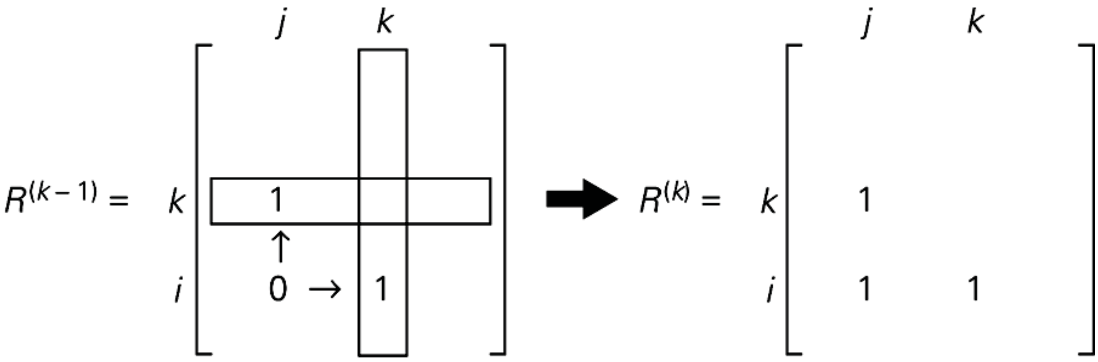
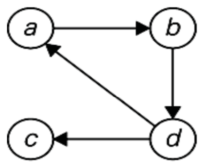


FIGURE 8.3 Rule for changing zeros in Warshall's algorithm



$$R^{(0)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 \end{array}$$

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 $R^{(1)}$.

$$R^{(1)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 0 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & \mathbf{1} & 1 & 0 \end{array}$$

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 $R^{(2)}$.

$$R^{(2)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & \mathbf{1} \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & \mathbf{1} \end{array}$$

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 $R^{(3)}$.

$$R^{(3)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & 0 & 1 & 0 & 1 \\ b & 0 & 0 & 0 & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

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 $R^{(4)}$.

$$R^{(4)} = \begin{array}{c|cccc} & a & b & c & d \\ \hline a & \mathbf{1} & 1 & \mathbf{1} & 1 \\ b & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ c & 0 & 0 & 0 & 0 \\ d & 1 & 1 & 1 & 1 \end{array}$$

Ones reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a, b, c,* and *d* (note five new paths).

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

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

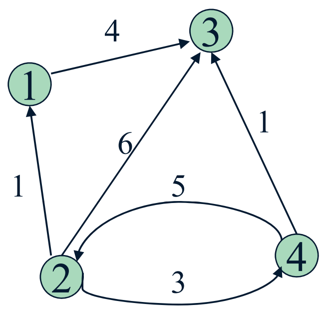
Space efficiency: Matrices can be written over their predecessors

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 $D^{(0)}$, …, $D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

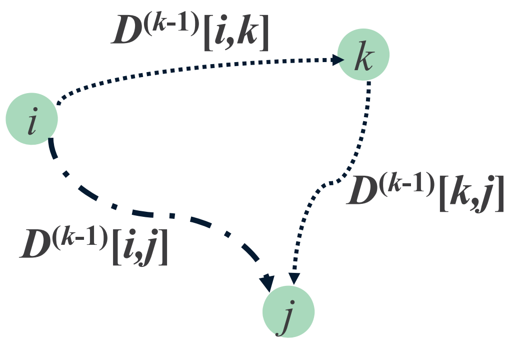
- **Example:**



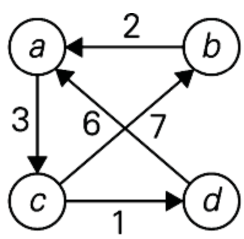
Floyd's Algorithm (matrix generation)

On the *k*-th iteration, the algorithm determines shortest paths between every pair of vertices *i, j* that use only vertices among 1,…,*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



(a)

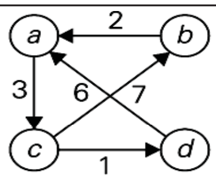
$$W = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

(b)

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

(c)

FIGURE 8.5 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.



$$D^{(0)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$$D^{(1)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \mathbf{5} & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \mathbf{9} & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e. just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$$D^{(2)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e. a and *b* (note a new shortest path from *c* to *a*).

$$D^{(3)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e. a, *b,* and *c* (note four new shortest paths from *a* to *b*, from a to *d*, from *b* to *d*, and from *d* to *b*).

$$D^{(4)} = \begin{array}{cc} & \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{array}{c} a \\ b \\ c \\ d \end{array} & \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e. a, *b, c,* and *d* (note a new shortest path from *c* to *a*).

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
$D \leftarrow W$ //is not necessary if W can be overwritten
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$
return D

Time efficiency: Θ(n^3)

Space efficiency: Matrices can be written over their predecessors

Knapsack Problem by DP

Given n items of

integer weights: $w_1 \quad w_2 \; \dots \; w_n$

values: $v_1 \quad v_2 \; \dots \; v_n$

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 ($j \le W$).

Let $V[i,j]$ be optimal value of such instance. Then

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

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

Knapsack Problem by DP (example)

Example: Knapsack of capacity $W = 5$

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity j

	i	0	1	2	3	4	5
	0						
$w_1 = 2, v_1 = 12$	1						
$w_2 = 1, v_2 = 10$	2						
$w_3 = 3, v_3 = 20$	3						
$w_4 = 2, v_4 = 15$	4						?

Knapsack Problem

- $V[i, j] = \max(\underbrace{V[i-1, j]}_{\text{object}_i \text{ not used}}, \underbrace{V[i-1, j-w_i] + v_i}_{\text{object}_i \text{ used}})$

		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			goal

FIGURE 8.12 Table for solving the knapsack problem by dynamic programming

Knapsack Problem

- $V[i, j] = \max(\underbrace{V[i-1, j]}_{\text{object}_i \text{ not used}}, \underbrace{V[i-1, j-w_i] + v_i}_{\text{object}_i \text{ used}})$

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	37

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]
```


Knapsack Problem – Memory Function

	i	capacity j: 0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	–	12	22	–	22
$w_3 = 3, v_3 = 20$	3	0	–	–	22	–	32
$w_4 = 2, v_4 = 15$	4	0	–	–	–	–	37

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