

# Algorithm and Data Structure Analysis (ADSA)

Shortest Paths2

# Bellman-Ford Algorithm

- **Dijkstra's algorithm** works for acyclic graphs and for non-negative edge costs.
- **Bellman-Ford algorithm** solves the problem for arbitrary edge costs.
- It uses  **$n-1$  rounds** and **relaxes all edges in each round**.
- This works **as simple paths have at most  $n-1$  edges**.
- **After the relaxations are complete, we have all shortest paths to nodes with non-negative cycles**.
- **to identify the nodes that can be reached by using negative cycles**.

# Finding Nodes with Negative Cycles

- **Assume** that there is an edge  $e=(u,v)$  that allows to **improve  $d[v]$  after the relaxations are complete.**
- Then the node  **$v$  is reachable by using a negative cycle.**
- Furthermore, **all nodes reachable from  $v$  can also be reached by using a negative cycle.**
- We set  $d[v] = -\infty$  for these nodes  $v$ .
- We **use postprocessing and the function infect** to find nodes reachable by negative cycles.

# Bellman-Ford Algorithm

```
Function BellmanFord(s : NodeId) : NodeArray × NodeArray  
    d =  $\langle \infty, \dots, \infty \rangle$  : NodeArray of  $\mathbb{R} \cup \{-\infty, \infty\}$            // distance from root  
    parent =  $\langle \perp, \dots, \perp \rangle$  : NodeArray of NodeId  
    d[s] := 0;   parent[s] := s           // self-loop signals root  
    for i := 1 to n − 1 do  
        forall e ∈ E do relax(e)           // round i  
    forall e = (u, v) ∈ E do           // postprocessing  
        if d[u] + c(e) < d[v] then infect(v)  
    return (d, parent)
```

```
Procedure infect(v)  
    if d[v] >  $-\infty$  then  
        d[v] :=  $-\infty$   
        foreach (v, w) ∈ E do infect(w)
```

Fig 10.9 Mehlhorn/Sanders

Runtime  $O(nm)$

# All-pairs-shortest-paths (APSP)

Given a directed graph  $G=(V,E)$  and a cost function

$c : E \rightarrow R_{\geq 0}$  on the edges.

Compute for each pair of nodes  $i$  and  $j$  a shortest path.

- We assume **edge costs are non-negative**.
- The nodes are labeled  $1, \dots, n$ .
- We set  $c(i,j) = \infty$  if there is no edge from  $i$  to  $j$  in  $G$ .
- Otherwise,  $c(i,j)$  is the cost of the edge from  $i$  to  $j$ .

# Dynamic Programming

**Dynamic Programming** is a powerful approach to solve problems of special structure.

**Approach:**

- Define and solve **subproblems**
- **Combine solutions**
- **Subproblems are not independent** (they share subsubproblems)
- **Solve every subsubproblem just once** and store the answer
- **Avoid recomputation**

# Dynamic Programming

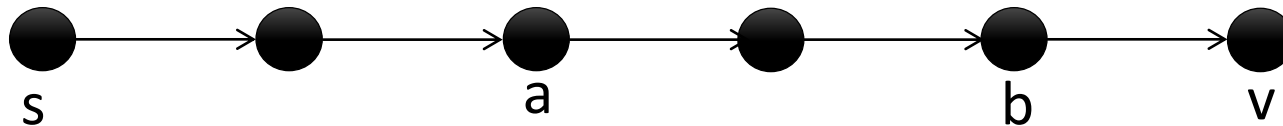
1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution bottom up.
4. Construct an optimal solution from computed information.

# Properties of subpaths

**Lemma:** Subpaths of a shortest path are also shortest paths.

**Proof (by contradiction):**

- Assume that the path  $P$  is a shortest path from  $s$  to  $v$ .
- Assume that a subpath from  $a$  to  $b$  is not a shortest path from  $a$  to  $b$



- This implies that there is a shorter path from  $a$  to  $b$
- We can use this path to obtain a shorter path from  $s$  to  $v$ .
- Contradiction to  $P$  is shortest path from  $s$  to  $v$ .





# Floyd-Warshall Algorithm

**Idea:** Compute the shortest path from  $i$  to  $j$  using only the intermediate nodes  $1, \dots, k$ .

**Do this for every  $k$ ,  $0 \leq k \leq n$ .**

**Notation:**

- $d_k(i,j)$  is the cost of a shortest path from  $i$  to  $j$  if only nodes from the set  $\{1, \dots, k\}$  are used.
- $N_k(i,j)$ : denotes the successor of  $i$  in such a path.

**Compute the entries of  $d$  and  $N$  for the different values of  $i,j,k$**

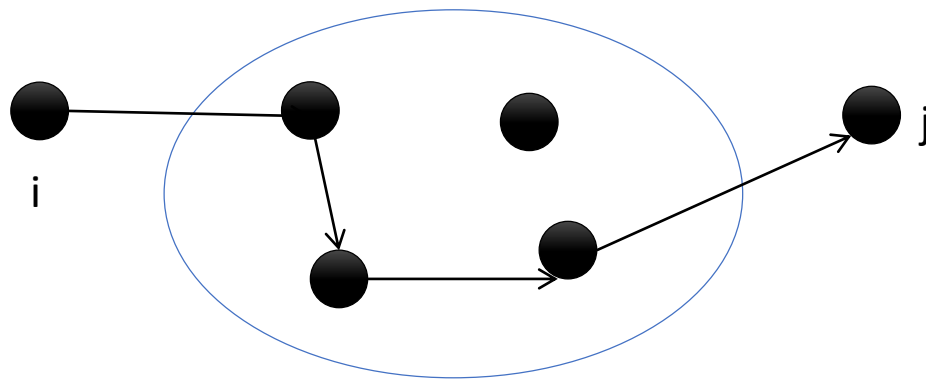
- **k=0**: no intermediate node is allowed and we have

$$d_0(i, j) = c(i, j) \text{ and}$$

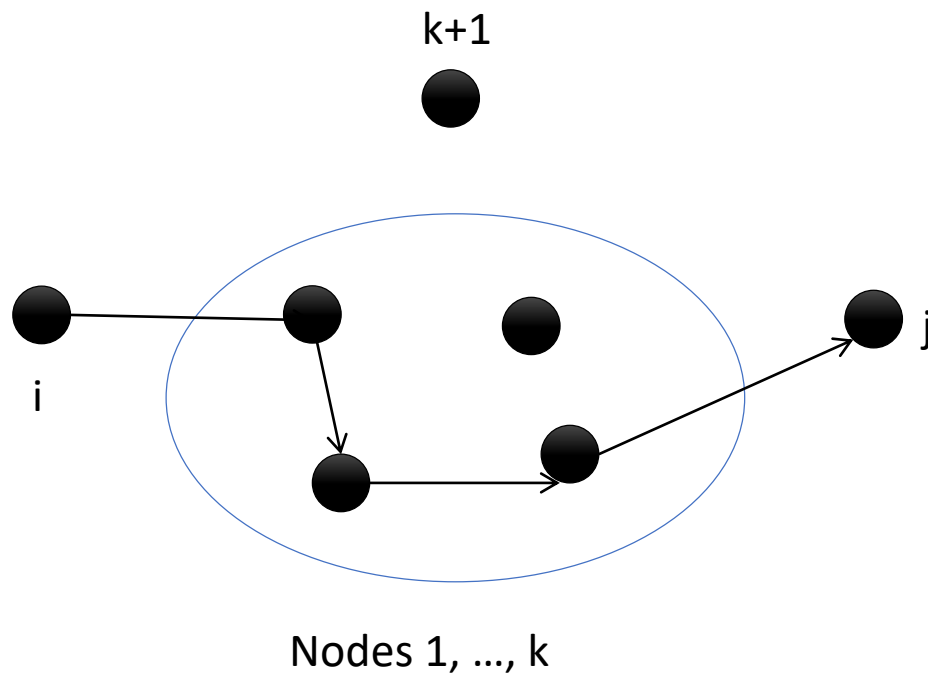
$$N_0(i, j) = j$$

- **k+1**: we have to check whether node k+1 can be used for a shorter path.

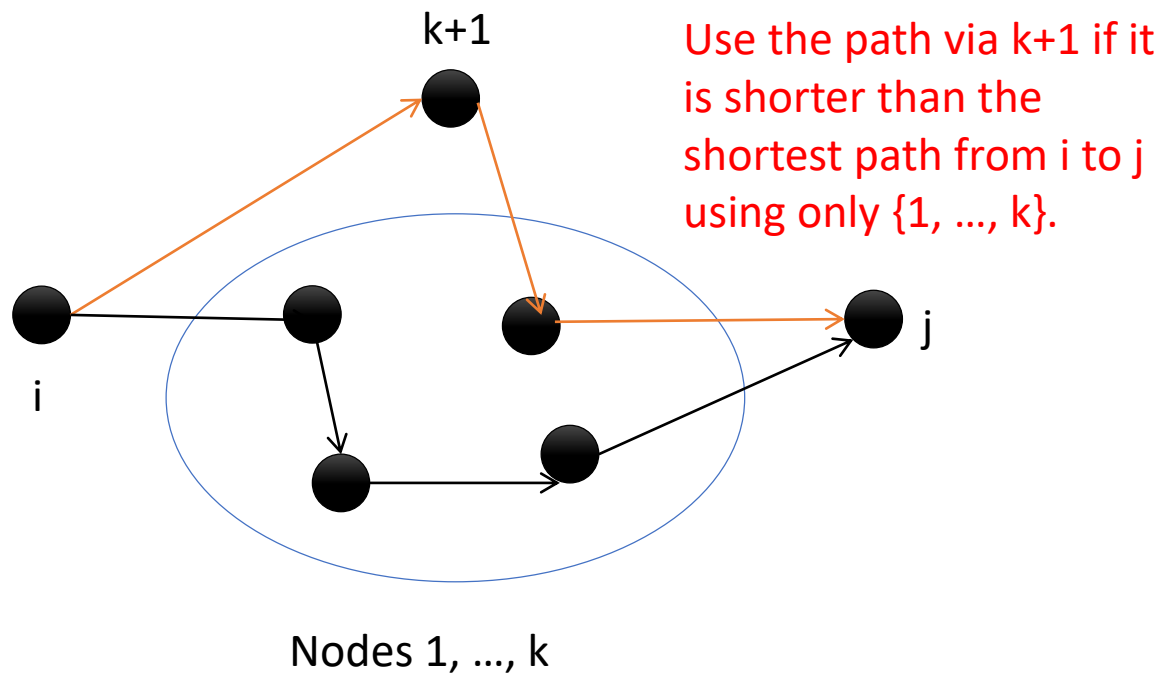
Shortest path from  $i$  to  $j$  using the nodes  $1, \dots, k$



Nodes  $1, \dots, k$



Adding the  
node  $k+1$  may  
lead to a  
shorter path  
from  $i$  to  $j$ .



Two possibilities:

- The **node  $k+1$  does not improve** the shortest path from  $i$  to  $j$ .
- We get a **shorter path by going from  $i$  to  $k+1$  and from  $k+1$  to  $j$ .**

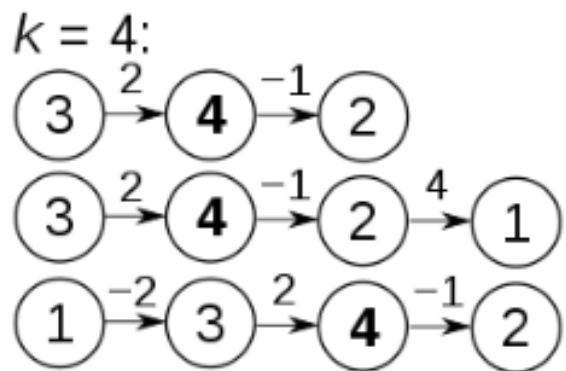
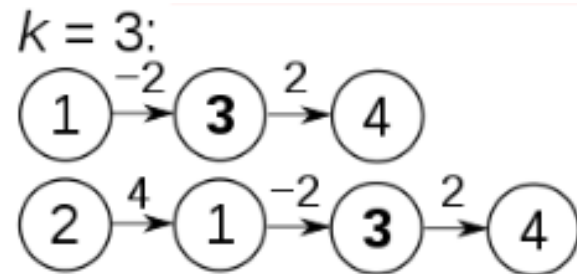
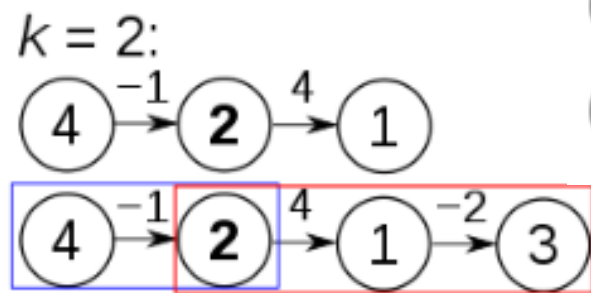
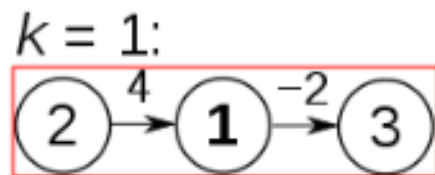
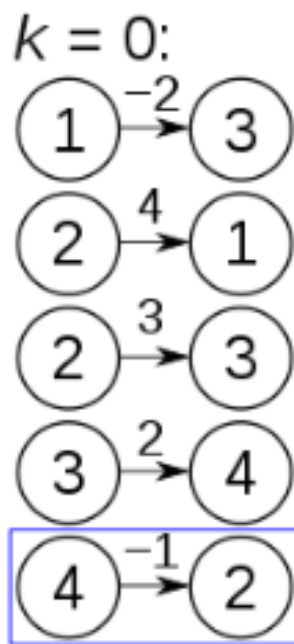
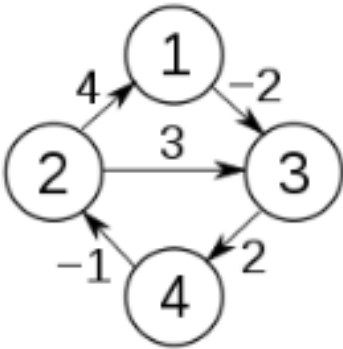
**Taking the best option,** we get

$$d_{k+1}(i, j) = \min\{d_k(i, j), d_k(i, k + 1) + d_k(k + 1, j)\}$$

- The **length of a shortest path from  $i$  to  $j$**  in the given graph  $G$  is  $d_n(i, j)$

# Successors

- If  $d_k(i, j) \leq d_k(i, k + 1) + d_k(k + 1, j)$   
we set  $N_{k+1}(i, j) = N_k(i, j)$
- Else we set  $N_{k+1}(i, j) = N_k(i, k + 1)$





The distance matrix at each iteration of  $k$ , with the updated distances in **bold**, will be:

$k = 0$		$j$			
		1	2	3	4
$i$	1	0	$\infty$	-2	$\infty$
	2	4	0	3	$\infty$
	3	$\infty$	$\infty$	0	2
	4	$\infty$	-1	$\infty$	0

$k = 1$		$j$			
		1	2	3	4
$i$	1	0	$\infty$	-2	$\infty$
	2	4	0	<b>2</b>	$\infty$
	3	$\infty$	$\infty$	0	2
	4	$\infty$	-1	$\infty$	0

$k = 2$		$j$			
		1	2	3	4
$i$	1	0	$\infty$	-2	$\infty$
	2	4	0	2	$\infty$
	3	$\infty$	$\infty$	0	2
	4	<b>3</b>	-1	<b>1</b>	0

$k = 3$		$j$			
		1	2	3	4
$i$	1	0	$\infty$	-2	<b>0</b>
	2	4	0	2	<b>4</b>
	3	$\infty$	$\infty$	0	2
	4	<b>3</b>	-1	<b>1</b>	0

$k = 4$		$j$			
		1	2	3	4
$i$	1	0	<b>-1</b>	-2	0
	2	4	0	2	4
	3	<b>5</b>	<b>1</b>	0	2
	4	3	-1	1	0

# Pseudocode

```
1 let dist be a  $|V| \times |V|$  array of minimum distances initialized to  $\infty$  (infinity)
2 for each edge  $(u,v)$ 
3    $\text{dist}[u][v] \leftarrow w(u,v)$  // the weight of the edge  $(u,v)$ 
4 for each vertex  $v$ 
5    $\text{dist}[v][v] \leftarrow 0$ 
6 for  $k$  from 1 to  $|V|$ 
7   for  $i$  from 1 to  $|V|$ 
8     for  $j$  from 1 to  $|V|$ 
9       if  $\text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j]$ 
10          $\text{dist}[i][j] \leftarrow \text{dist}[i][k] + \text{dist}[k][j]$ 
11       end if
```

# Runtime

- We need to compute all the entries

$$d_k(i, j), 1 \leq i, j, k \leq n$$

$$N_k(i, j), 1 \leq i, j, k \leq n$$

- For  $k=0$ , we can set the values directly.
- Each entry for  $k+1$  can be computed in constant time if we have already all entries for  $k$ .
- There are  $O(n^3)$  entries that have to be computed.
- Total runtime is  $O(n^3)$ .