INT102
Algorithmic Foundations And Problem Solving

More Shortest paths: Bellman-ford Algorithm, Floyd's Algorithm

Dr Pengfei Fan
Department of Intelligent Science

西交利物浦大學
Xi'an Jiaotong-Liverpool University

Learning outcomes

- Bellman-Ford Algorithm, to find the shortest paths in a graph (edges may have a negative weight) or detect a negative weighted cycle in a graph
- Floyd's Algorithm, to find all pair-shortest paths
- Warshall's Algorithm, to find transitive closure of a directed graph (Self-Study)

Single-source shortest-paths

- Consider a (un)directed connected graph G with the edges labelled by weight
- Given a particular vertex called the ***source***
  - Find **shortest paths** from the source to all other vertices (shortest path means the total weight of the path is the smallest)

Shortest Paths

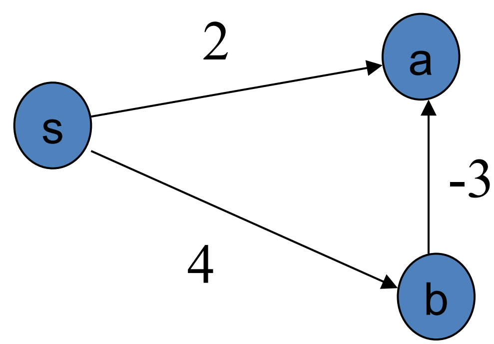
- **Single-source shortest paths:** Find shortest paths from the source to all other vertices (shortest path means the total weight of the path is the smallest)
  - Only nonnegative edge weights: Dijkstra's algorithm
  - Allow negative edge weights: Bellman-Ford algorithm (Dynamic Programming)

- **All-pairs shortest paths:** Find shortest path between each pair of vertices.
  - Floyd's algorithm (Dynamic Programming)

Bellman-Ford Algorithm

Negative Weighted Edges

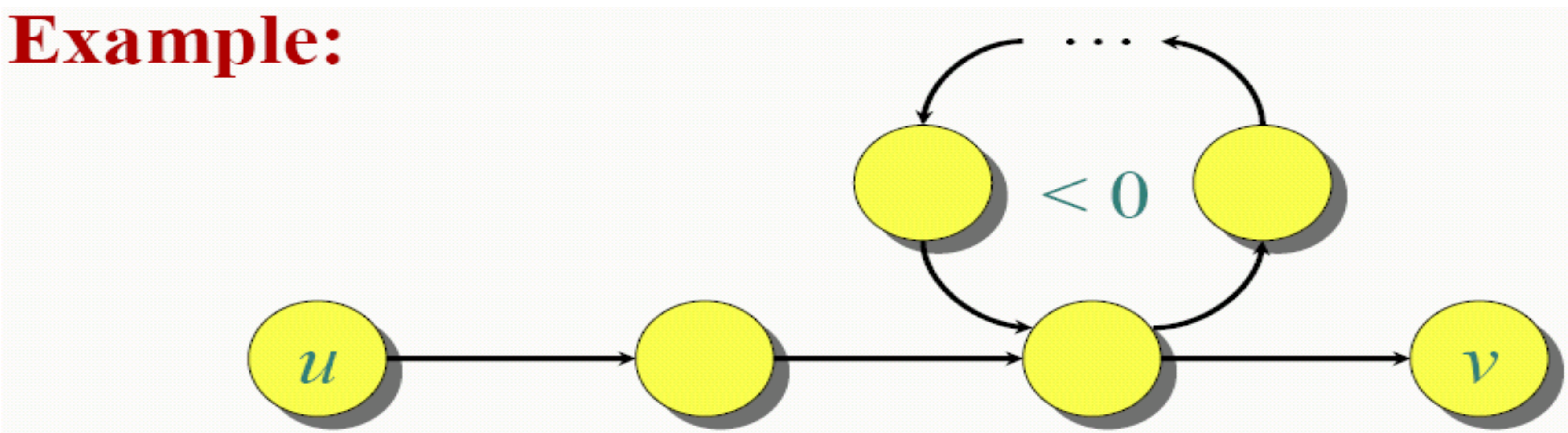
- Dijlstra's Algorithm does not work with graphs with negative weighted edges.
  - Consider the digraph consists of V = {s, a, b} and E = {(s, a), (s, b), (b, a)}, where *w*(s, a) = 2, *w*(s, b) = 4, and *w*(b, a) = -3.



Dijkstra's algorithm gives d[a] = 2, d[b] = 4. But due to the negative-edge weight *w*(b, a), the shortest distance from vertex s to vertex a is 4-3 = 1.

Negative-weight Cycles

Recall: If a graph $G = (V, E)$ contains a negative-weight cycle, then some shortest paths may not exist.



Bellman-Ford algorithm

- *Bellman-Ford algorithm:* Finds all shortest-path lengths from a *source* $s \in V$ to all $v \in V$ or determines that a negative-weight cycle exists.

Algorithm Bellman-Ford(G=(V, E), s)
//input: a graph G=(V,E) with a source vertex s
//output: an array d[0..|V|-1], indexed with V, $d[v]$ is the
//length of shortest path from s to v

$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
  do $d[v] \leftarrow \infty$

for $i \leftarrow 1$ **to** $|V| - 1$
  do for each edge $(u, v) \in E$
    do if $d[v] > d[u] + w(u, v)$
      then $d[v] \leftarrow d[u] + w(u, v)$

for each edge $(u, v) \in E$
  do if $d[v] > d[u] + w(u, v)$
    then report that a negative-weight cycle exists

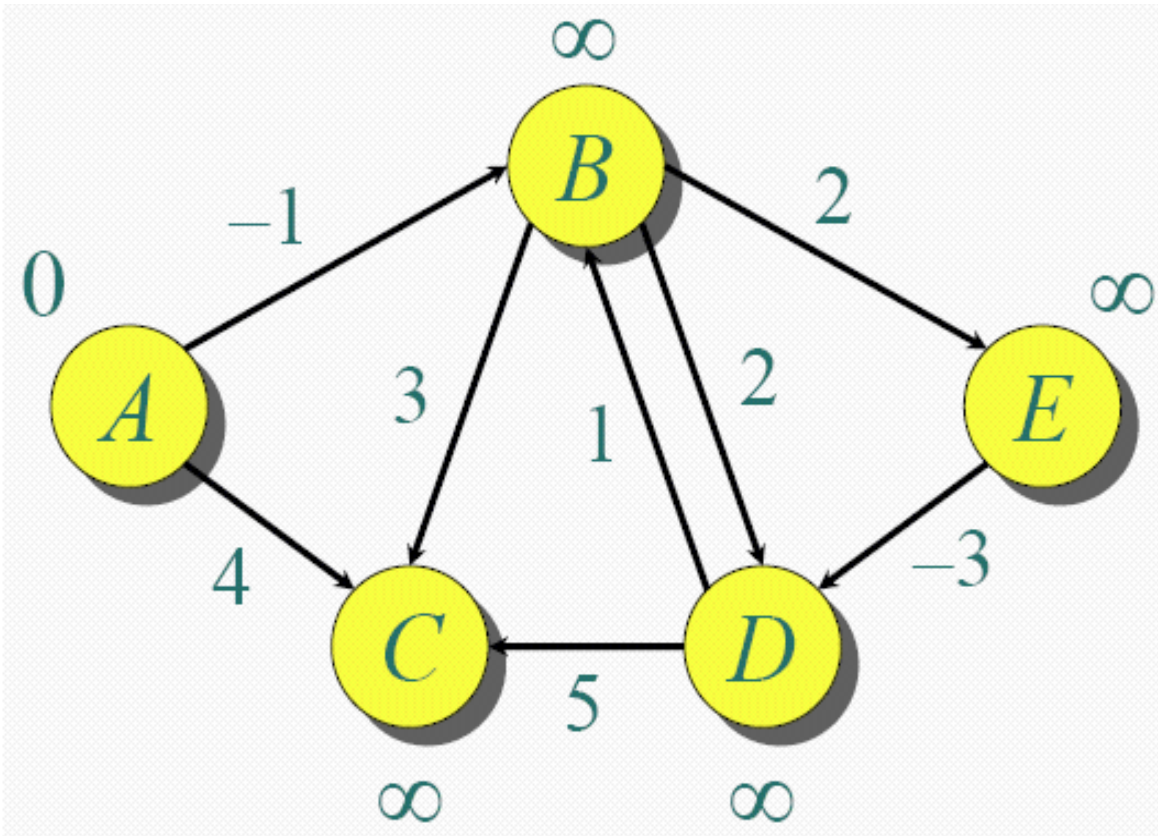
Time complexity: O(VE)

Dynamic programming

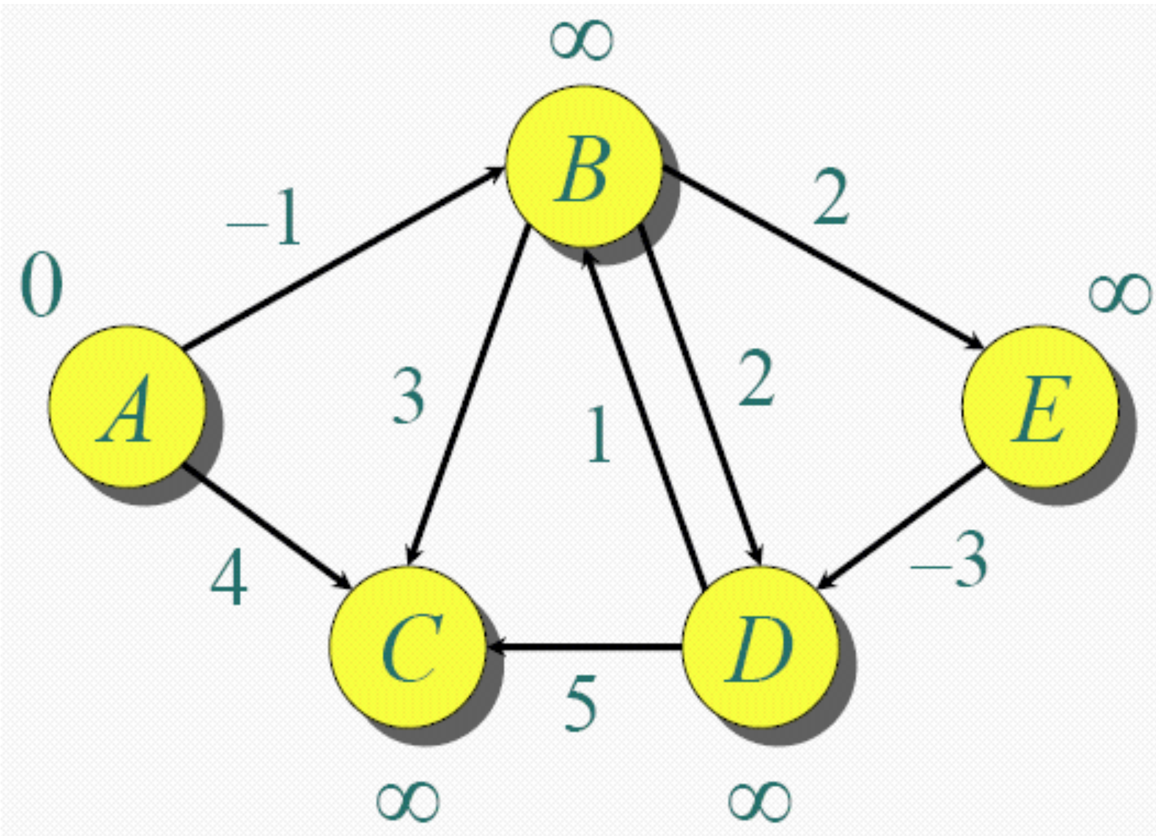
- Write down a formula that relates a solution of a problem with those of sub-problems.
  E.g. $F(n) = F(n-1) + F(n-2)$.
- Index the sub-problems so that they can be **stored** and **retrieved** easily in a table (i.e., array)
- Fill the table in some **bottom-up** manner; start filling the solution of the smallest problem.
  - This ensures that when we solve a particular sub-problem, the solutions of all the smaller sub-problems that it depends are available.

An Example

G is as following and A is the source vertex.



An Example



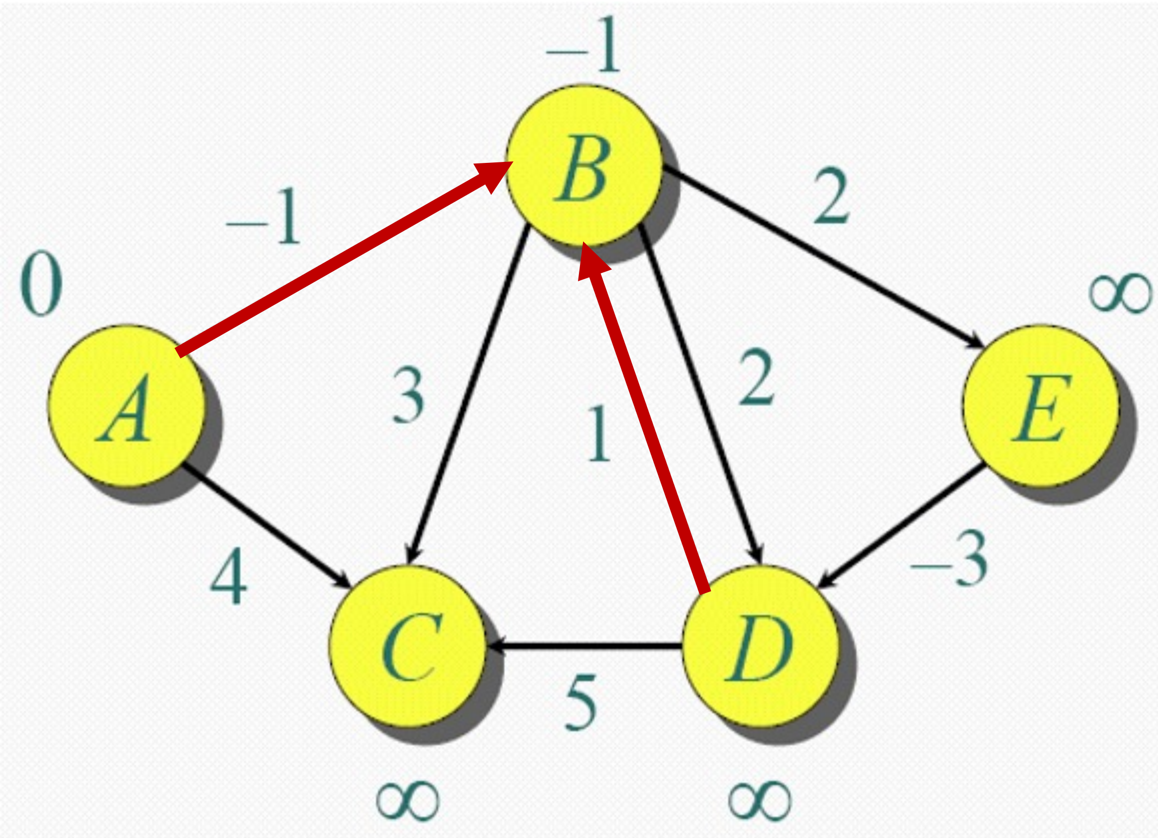
A	B	C	D	E
0	∞	∞	∞	∞

An Example

Note: **for** $i \leftarrow 1$ **to** $|V| - 1$:

The first iteration (i=1)

Update B, w.r.t AB, DB.



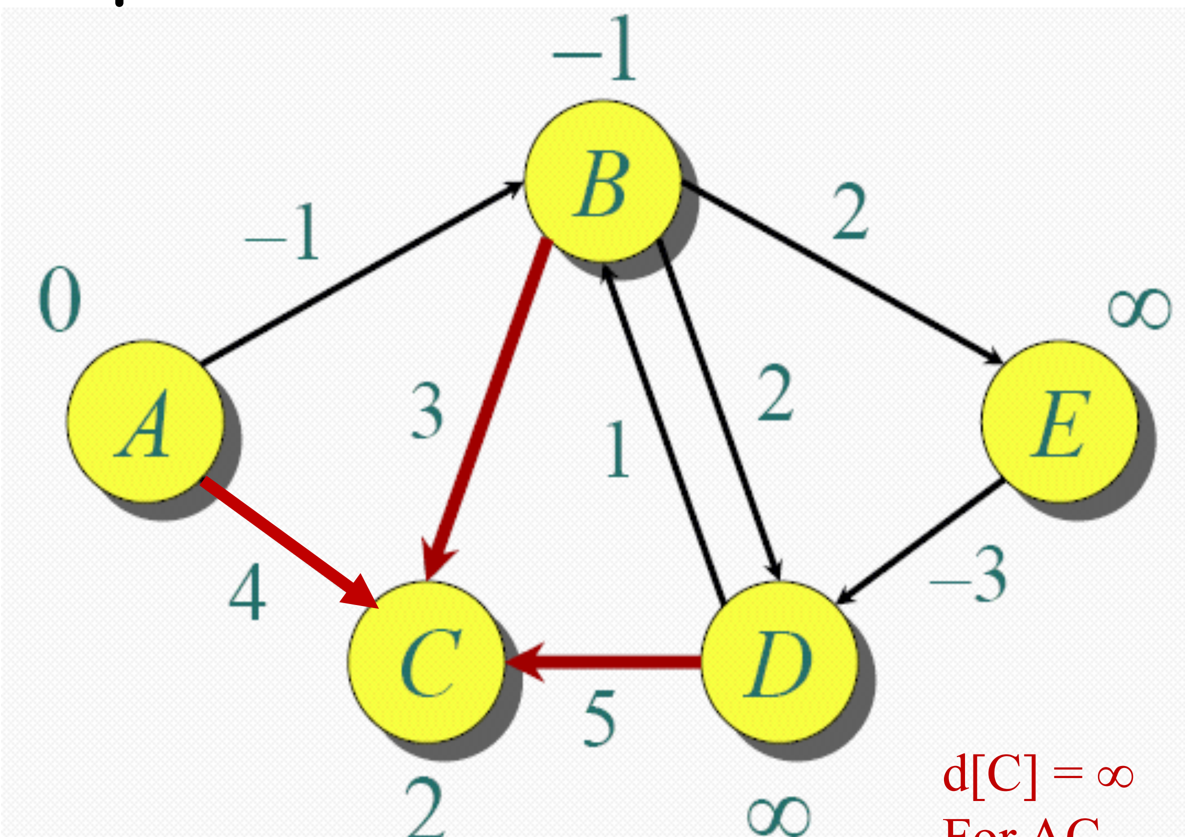
A	*B*	*C*	*D*	*E*
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	∞	∞	∞	∞

d[B] = ∞
For AB
d[A]+w(A,B)=0+-1=-1
For DB
d[D]+w(D,B)=∞+1=∞

do if $d[v] > d[u] + w(u, v)$
then $d[v] \leftarrow d[u] + w(u, v)$

An Example

Update C, w.r.t AC, BC, DC.



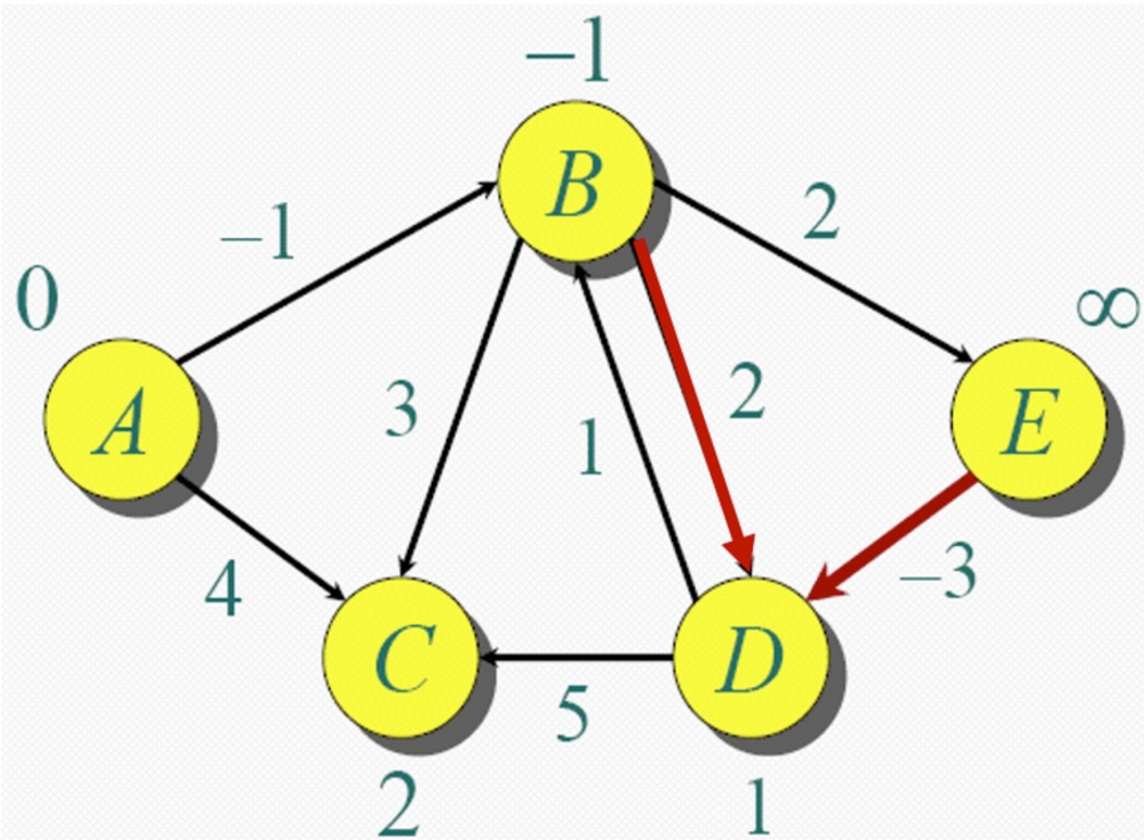
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	∞	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	∞	∞	∞

d[C] = ∞
For AC
d[A]+w(A,C)=0+4=4
For BC
d[B]+w(B,C)=-1+3=2
For DC
d[D]+w(D,C)=∞+5=∞

do if $d[v] > d[u] + w(u, v)$
then $d[v] \leftarrow d[u] + w(u, v)$

An Example

Update D, w.r.t BD, ED.



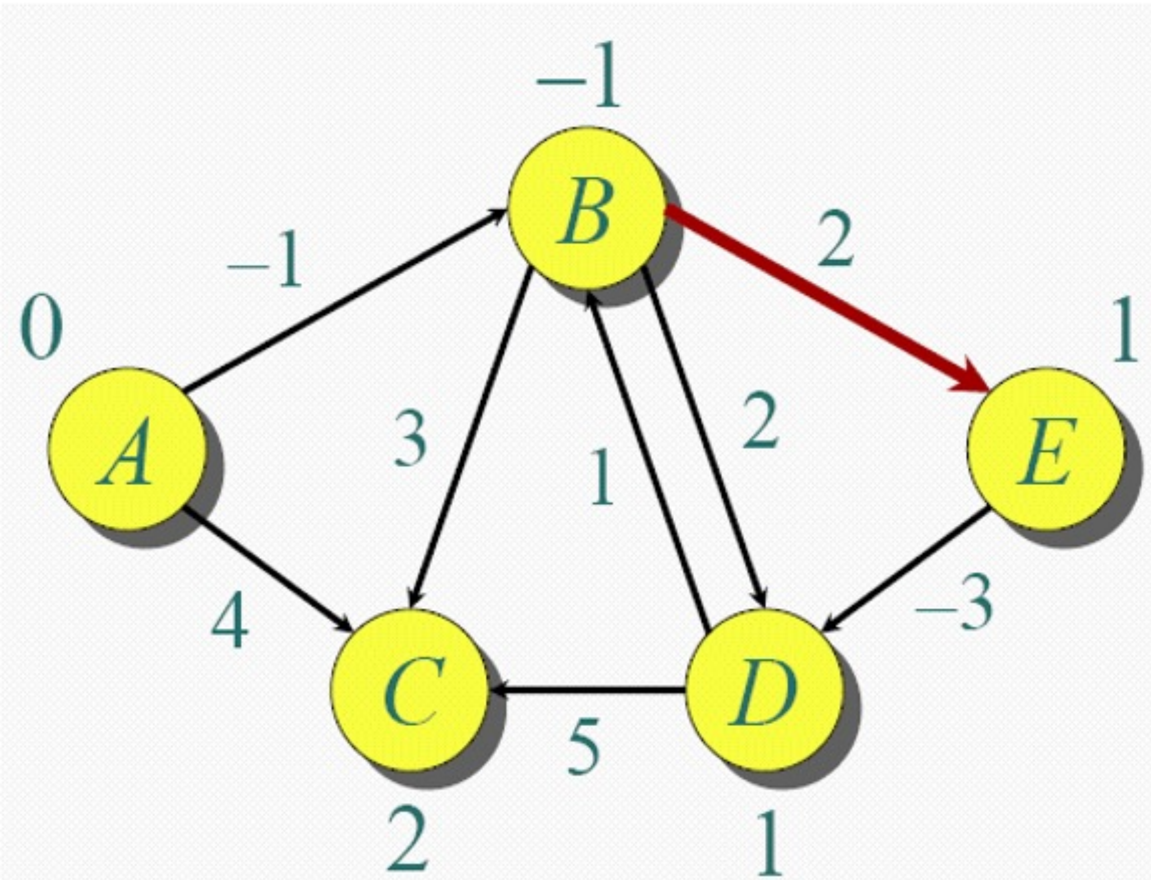
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	∞	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	∞	∞	∞
0	−1	2	1	∞
0	−1	2	∞	∞

d[D] = ∞
For BD
d[B]+w(B,D)=-1+2=1
For ED
d[E]+w(E,D)=∞+-3=∞

do if $d[v] > d[u] + w(u, v)$
then $d[v] \leftarrow d[u] + w(u, v)$

An Example

Update E, w.r.t BE.



A	B	C	D	E
0	∞	∞	∞	∞
0	–1	∞	∞	∞
0	∞	∞	∞	∞
0	–1	4	∞	∞
0	–1	2	∞	∞
0	–1	∞	∞	∞
0	–1	2	1	∞
0	–1	2	∞	∞
0	–1	2	1	1

do if $d[v] > d[u] + w(u, v)$
then $d[v] \leftarrow d[u] + w(u, v)$

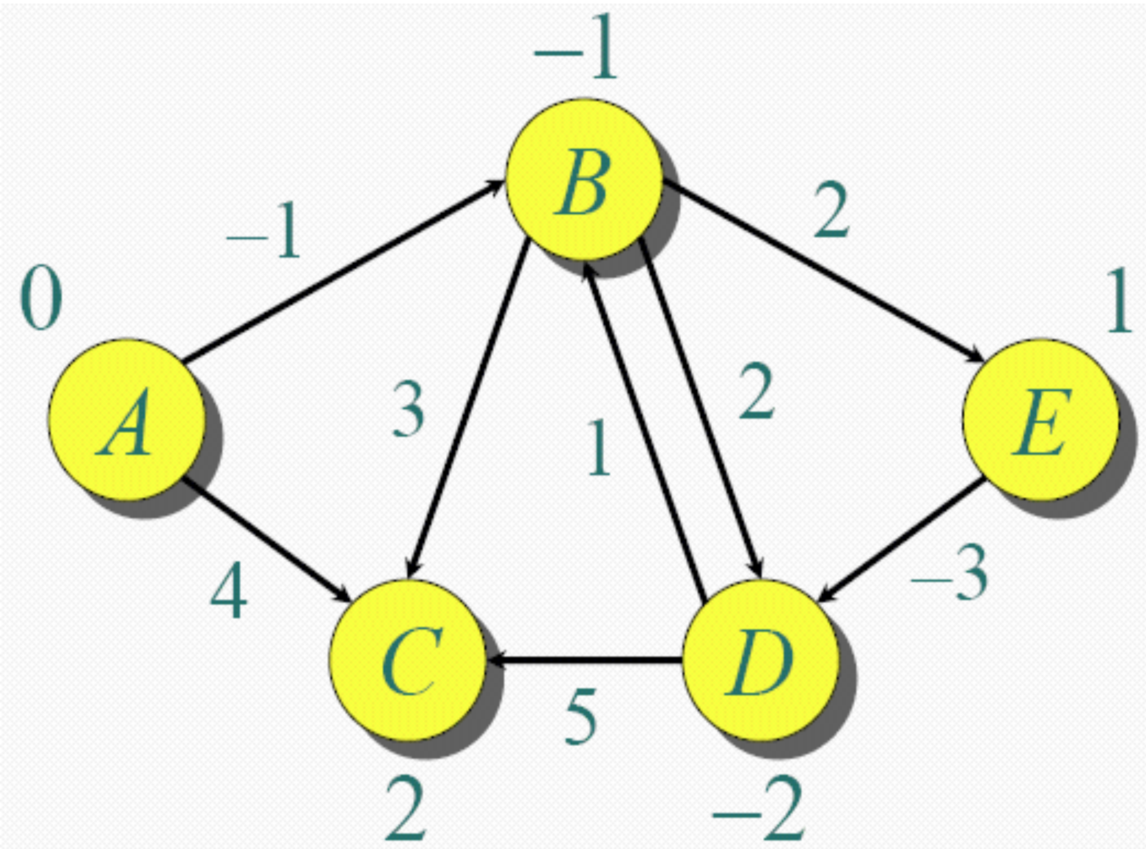
$d[E] = \infty$
For BE
$d[B]+w(B,E)=-1+2=1$

An Example

Note: **for** $i \leftarrow 1$ **to** $|V| - 1$:

The second iteration (i=2)

A	B	C	D	E
0	–1	2	1	1
0	–1	2	–2	1



The third iteration (i=3)

The fourth iteration (i=4)

do if $d[v] > d[u] + w(u, v)$

then $d[v] \leftarrow d[u] + w(u, v)$

See how to maintain the array of predecessors in Q1 from Tutorial Week 8

Floyd's Algorithm

All-pairs shortest paths

Problem: *A* weighted (di)graph $G = (V, E)$, find a $|V| \times |V|$ matrix, it's entry d_{ij} is the shortest-path length between vertices i, j, where $i, j \in V$.

Naïve solution:

- Run Bellman-Ford algorithm once for each vertex.
- Time = $O(V^2 E)$.
- Dense graph $\Rightarrow O(V^4)$ time, e.g., for a complete graph, $|E|=|V||(V-1)|/2$

Dynamic programming

- Write down a formula that relates a solution of a problem with those of sub-problems.
  E.g. $F(n) = F(n-1) + F(n-2)$.
- Index the sub-problems so that they can be **stored** and **retrieved** easily in a table (i.e., array)
- Fill the table in some **bottom-up** manner; start filling the solution of the smallest problem.
  - This ensures that when we solve a particular sub-problem, the solutions of all the smaller sub-problems that it depends are available.

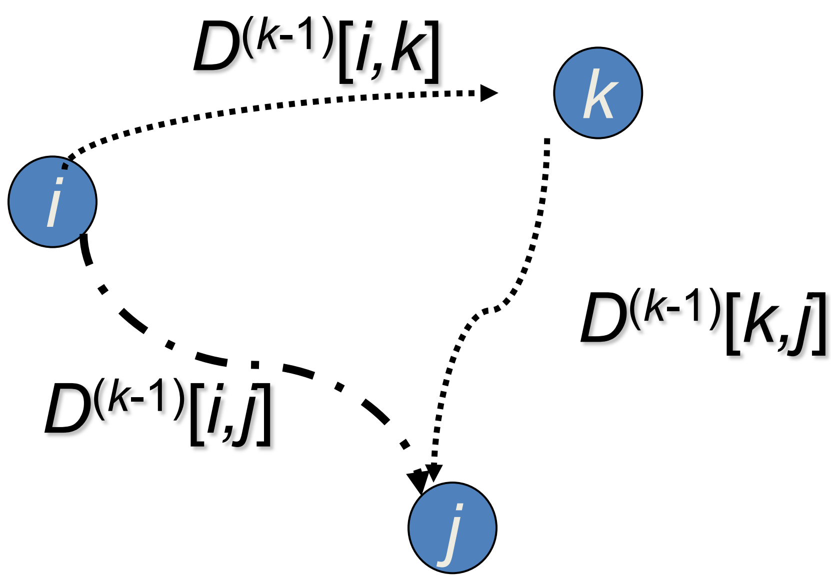
Floyd's Algorithm: All Pairs Shortest Paths

- Problem: in a weighted (di)graph, find shortest paths between every pair of vertices
- Weight Matrix: the entry w_{ij} is the weight on edge $(v_{i,}\ v_j)$.
- Distance Matrix: the entry d_{ij} is the distance between v_i and v_j.
- Idea: construct solutions through a series of matrices $D^{(0)}, \ldots, D^{(n)}$ using increasing subsets of the vertices allowed as intermediate

Floyd's Algorithm

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

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



Recursive & Non-Recursive:
https://www.youtube.com/watch?v=NdBHw5mqIZE&ab_channel=arisaif

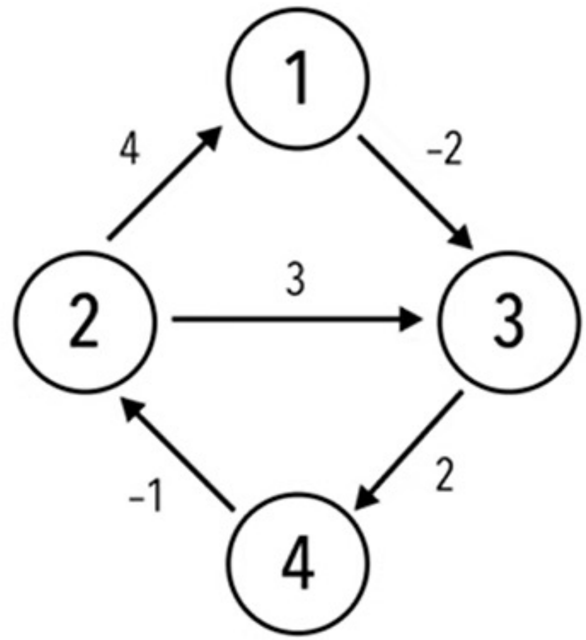
Floyd's Algorithm (pseudocode)

```
let V = number of vertices in graph
let dist = V × V array of minimum distances initialized to ∞
for each vertex v
   dist [v][v] ← 0
for each edge (u,v)
   dist [u][v] ← weight(u,v)
for k from 1 to V
   for i from 1 to V
      for j from 1 to V
         if dist [i][j] > dist [i][k] + dist [k][j]
            dist [i][j] ← dist [i][k] + dist [k][j]
         end if
```

Floyd's Algorithm (example)

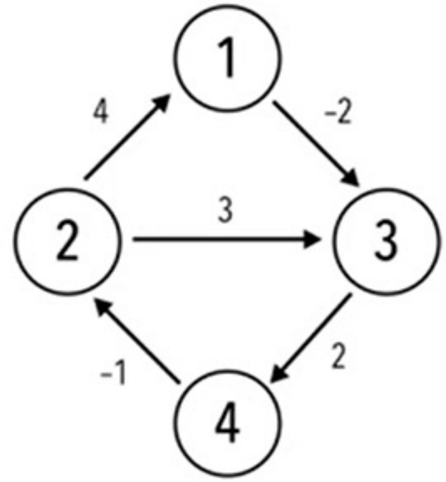
```
let V = number of vertices in graph
let dist = V × V array of minimum distances
for each vertex v
  dist [v][v] ← 0
for each edge (u,v)
  dist [u][v] ← weight(u,v)
for k from 1 to V
  for i from 1 to V
    for j from 1 to V
      if dist [i][j] > dist [i][k] + dist [k][j]
        dist [i][j] ← dist [i][k] + dist [k][j]
      end if
```

V = 4



Floyd's Algorithm (example)

```
let V = number of vertices in graph
let dist = V × V array of minimum distances
for each vertex v
  dist [v][v] ← 0
for each edge (u,v)
  dist [u][v] ← weight(u,v)
for k from 1 to V
  for i from 1 to V
    for j from 1 to V
      if dist [i][j] > dist [i][k] + dist [k][j]
        dist [i][j] ← dist [i][k] + dist [k][j]
      end if
```



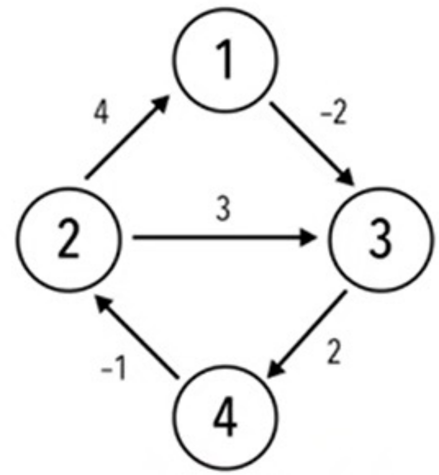
	1	2	3	4
1				
2				
3				
4				

dist

empty cells = ∞

Floyd’s Algorithm (example)

```
let V = number of vertices in graph
let dist = V × V array of minimum distances
for each vertex v
  dist [v][v] ← 0
for each edge (u,v)
  dist [u][v] ← weight(u,v)
for k from 1 to V
  for i from 1 to V
    for j from 1 to V
      if dist [i][j] > dist [i][k] + dist [k][j]
        dist [i][j] ← dist [i][k] + dist [k][j]
      end if
```



to vertex

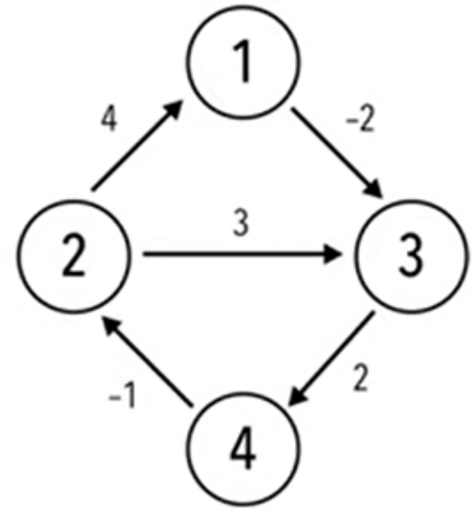
from vertex

	1	2	3	4
1				
2				
3				[3,4]
4				

dist

Floyd’s Algorithm (example)

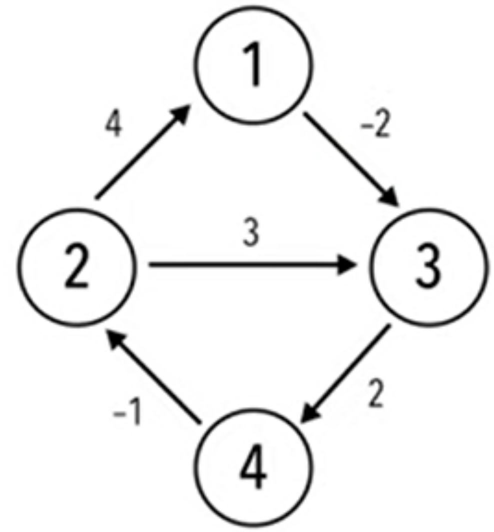
```
let V = number of vertices in graph
let dist = V × V array of minimum distances
for each vertex v
  dist [v][v] ← 0
for each edge (u,v)
  dist [u][v] ← weight(u,v)
for k from 1 to V
  for i from 1 to V
    for j from 1 to V
      if dist [i][j] > dist [i][k] + dist [k][j]
        dist [i][j] ← dist [i][k] + dist [k][j]
      end if
```



	1	2	3	4
1	0			
2		0		
3			0	
4				0

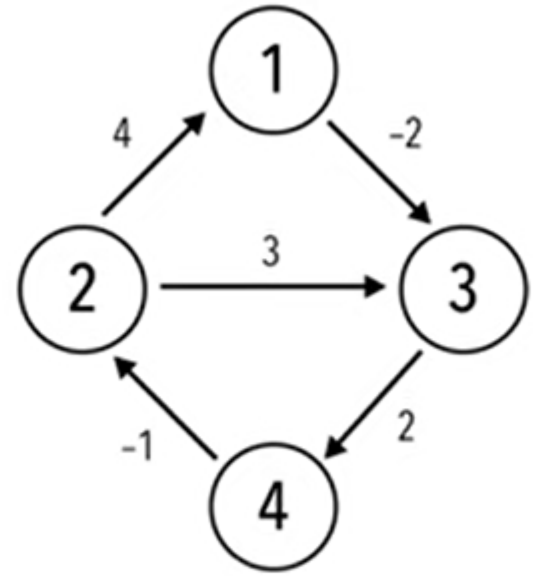
Floyd’s Algorithm (example)

```
let V = number of vertices in graph
let dist = V × V array of minimum distances
for each vertex v
  dist [v][v] ← 0
for each edge (u,v)
  dist [u][v] ← weight(u,v)
for k from 1 to V
  for i from 1 to V
    for j from 1 to V
      if dist [i][j] > dist [i][k] + dist [k][j]
        dist [i][j] ← dist [i][k] + dist [k][j]
      end if
```



	1	2	3	4
1	0		–2	
2	4	0	3	
3			0	2
4		-1		0

Floyd’s Algorithm (example)

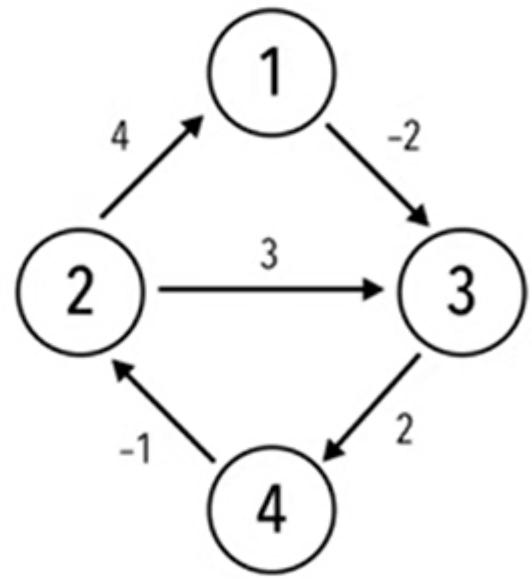


k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

if dist [i][j] > dist [i][k] + dist [k][j]
    dist [i][j] ← dist [i][k] + dist [k][j]

	1	2	3	4
1	0		-2	
2	4	0	3	
3			0	2
4		-1		0

Floyd’s Algorithm (example)

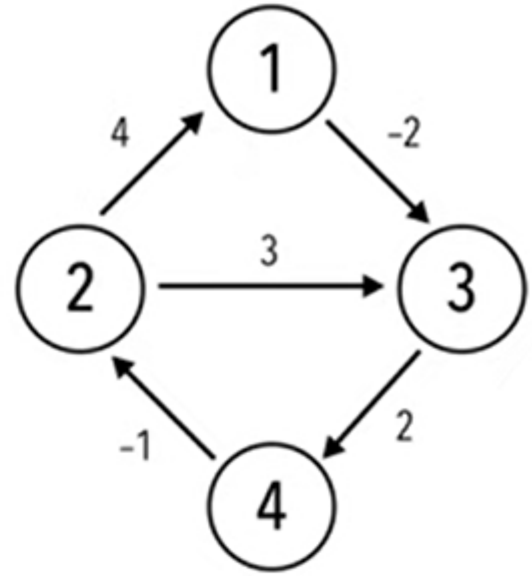


k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]
dist [1][1] > dist [1][1] + dist [1][1]
0 > 0 + 0
X 0 > 0

	1	2	3	4
1	0		-2	
2	4	0	3	
3			0	2
4		-1		0

Floyd's Algorithm (example)



k = 1 2 3 4

i = 1 2 3 4

j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]

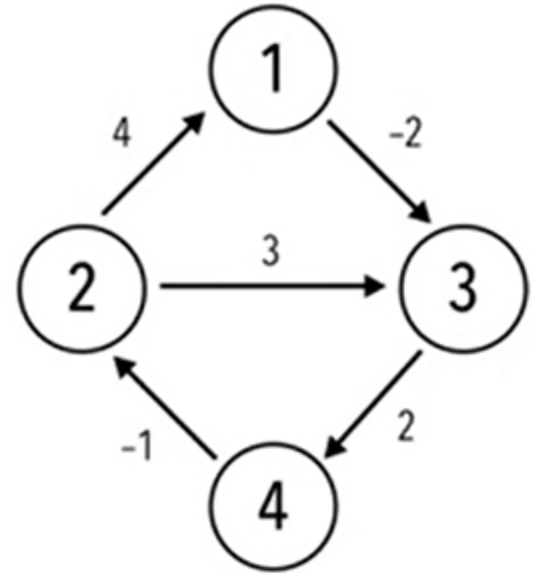
dist [1][2] > dist [1][1] + dist [1][2]

$\infty > 0 + \infty$

X $\infty > \infty$

	1	2	3	4
1	0		-2	
2	4	0	3	
3			0	2
4		-1		0

Floyd's Algorithm (example)



k = 1 2 3 4

i = 1 2 3 4

j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]

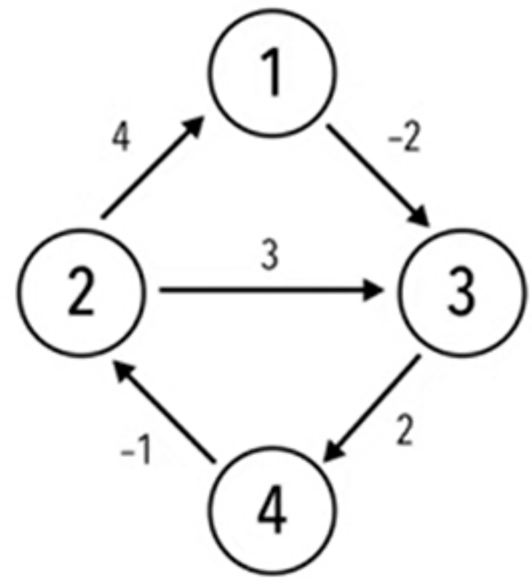
dist [1][3] > dist [1][1] + dist [1][3]

−2 > 0 + −2

X −2 > −2

	1	2	3	4
1	0		-2	
2	4	0	3	
3			0	2
4		-1		0

Floyd's Algorithm (example)



k = 1 2 3 4

i = 1 2 3 4

j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]

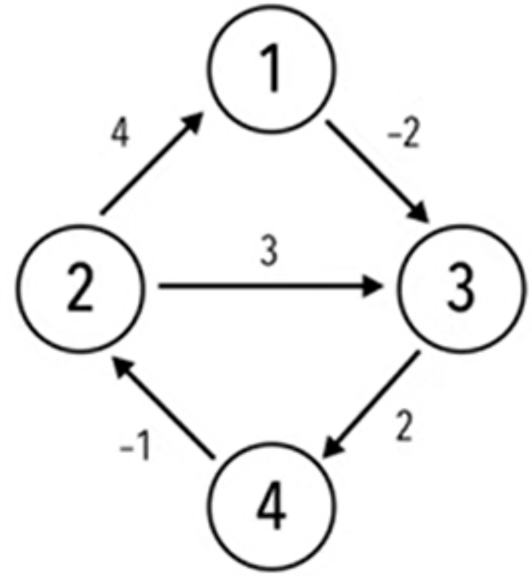
dist [1][4] > dist [1][1] + dist [1][4]

$\infty > 0 + \infty$

X $\infty > \infty$

	1	2	3	4
1	0		-2	
2	4	0	3	
3			0	2
4		-1		0

Floyd’s Algorithm (example)

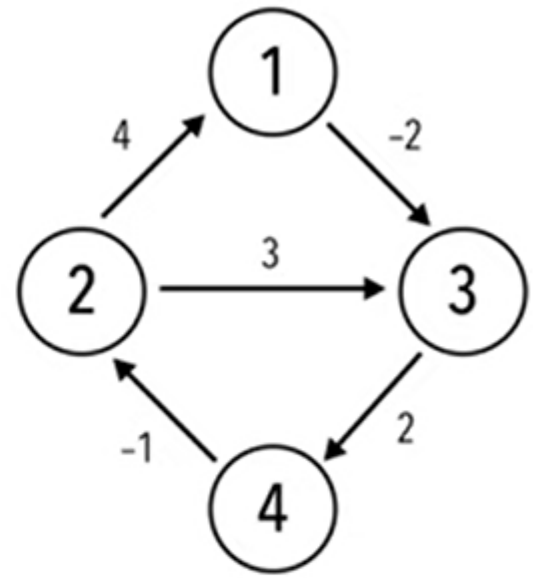


k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]
dist [2][3] > dist [2][1] + dist [1][3]
3 > 4 + −2
3 > 2

	1	2	3	4
1	0		−2	
2	4	0	2	
3			0	2
4		-1		0

Floyd’s Algorithm (example)

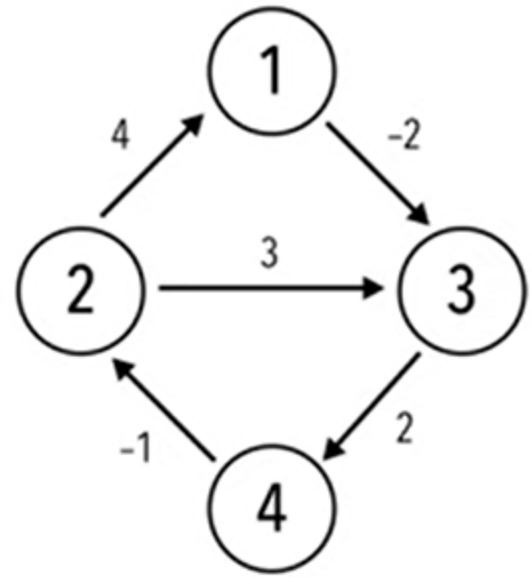


k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]
dist [4][1] > dist [4][2] + dist [2][1]
$\infty > -1 + 4$
$\infty > 3$

	1	2	3	4
1	0		–2	
2	4	0	2	
3			0	2
4	3	-1		0

Floyd’s Algorithm (example)



k = 1 2 3 4

i = 1 2 3 4

j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]

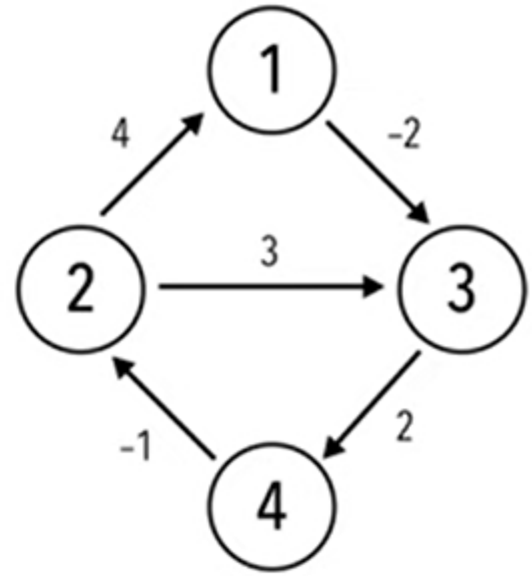
dist [4][3] > dist [4][2] + dist [2][3]

$\infty > -1 + 2$

$\infty > 1$

	1	2	3	4
1	0		–2	
2	4	0	2	
3			0	2
4	3	-1	1	0

Floyd’s Algorithm (example)



k = 1 2 3 4

i = 1 2 3 4

j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]

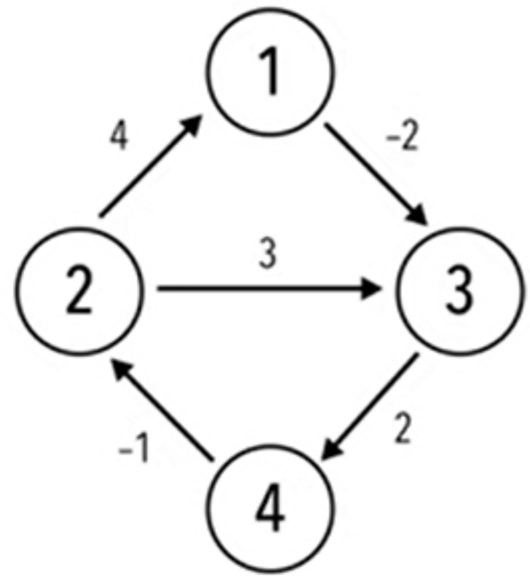
dist [3][1] > dist [3][4] + dist [4][1]

$\infty > 2 + 3$

$\infty > 5$

	1	2	3	4
1	0	-1	-2	0
2	4	0	2	4
3	5		0	2
4	3	-1	1	0

Floyd's Algorithm (example)



k = 1 2 3 4

i = 1 2 3 4

j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]

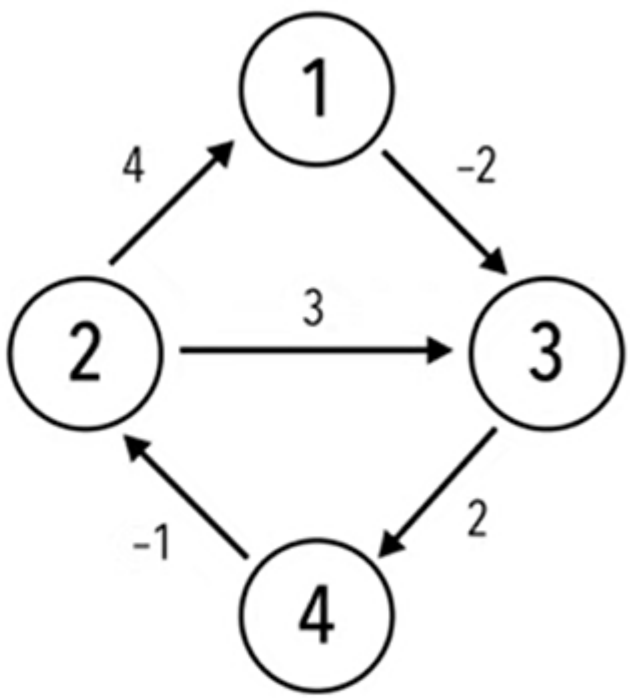
dist [3][2] > dist [3][4] + dist [4][2]

$\infty > 2 + -1$

$\infty > 1$

	1	2	3	4
1	0	-1	-2	0
2	4	0	2	4
3	5	1	0	2
4	3	-1	1	0

Floyd’s Algorithm (example)



	1	2	3	4
1	0	-1	-2	0
2	4	0	2	4
3	5	1	0	2
4	3	-1	1	0

Distance Table & Route Table:
https://www.youtube.com/watch?v=3wmsXtSUm8Y&ab_channel=MathsHelpwithMrOrys

Floyd's Algorithm (Complexity)

Time efficiency: $O(n^3)$

Warshall's Algorithm

Transitive closure of a directed graph (Self-Study)

Input: Digraph $G = (V, E)$, where $|V| = n$.

Output: $n \times n$ matrix R = (R_{ij}) , where

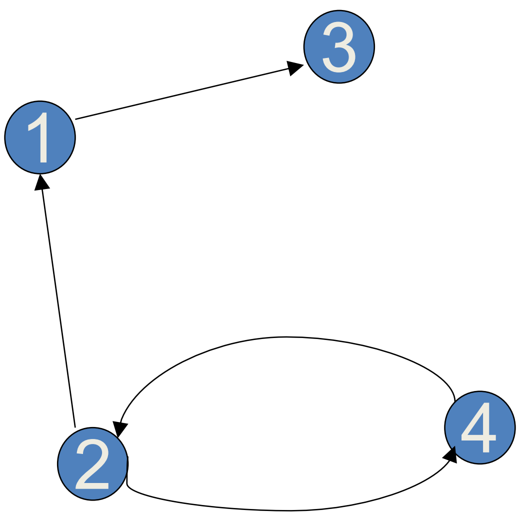
$$R_{ij} = \begin{cases} 1 & \text{if there exists a path from } i \text{ to } j, \\ 0 & \text{otherwise.} \end{cases}$$

IDEA: Similar to Floyd' algorithm, but with $(\vee, \wedge)$ instead of $(\min, +)$:

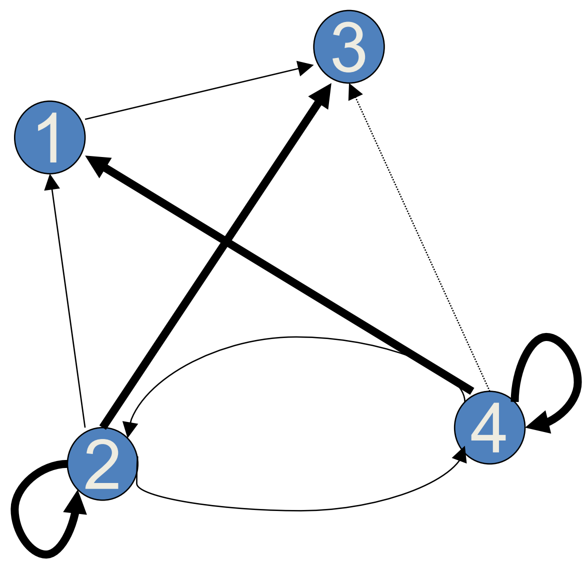
$$R_{ij}^{(k)} = R_{ij}^{(k-1)} \vee (R_{ik}^{(k-1)} \wedge R_{kj}^{(k-1)}).$$

Warshall's Algorithm: Transitive Closure

- Computes the transitive closure of a relation
- Alternatively: existence of all nontrivial paths in a digraph
- Example of transitive closure:



$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$



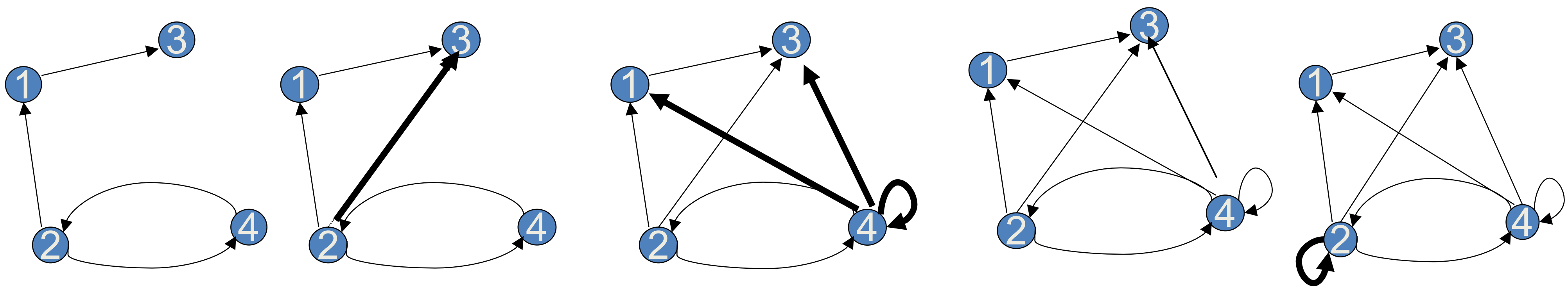
$$\begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

Warshall's Algorithm

Constructs transitive closure T as the last matrix in the sequence of n-by-n matrices $R^{(0)}, \ldots, R^{(k)}, \ldots, R^{(n)}$ where

$R^{(k)}[i,j] = 1$ iff there is nontrivial path from i to j with only first k vertices allowed as intermediate

Note that $R^{(0)} = A$ (adjacency matrix), $R^{(n)} = T$ (transitive closure)

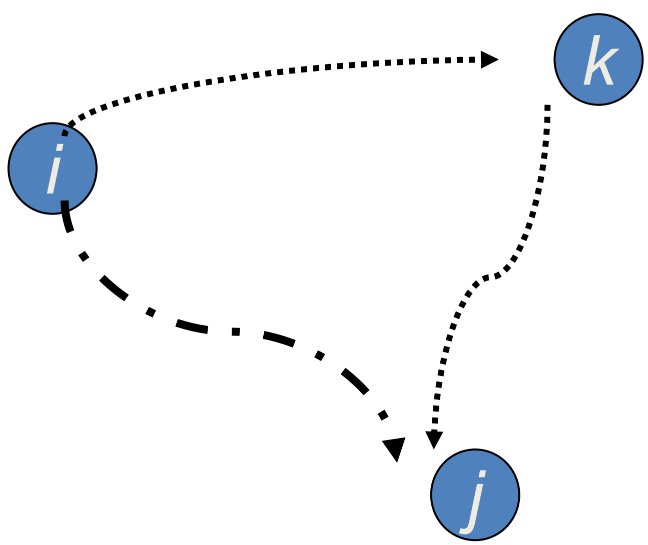


$$R^{(0)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix} \qquad R^{(1)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix} \qquad R^{(2)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix} \qquad R^{(3)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix} \qquad R^{(4)} \quad \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

Warshall's Algorithm (recurrence)

On the k-th iteration, the algorithm determines for every pair of vertices i, j if a path exists from i and j with just vertices 1,…,k allowed as intermediate

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



Warshall's Algorithm (matrix generation)

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

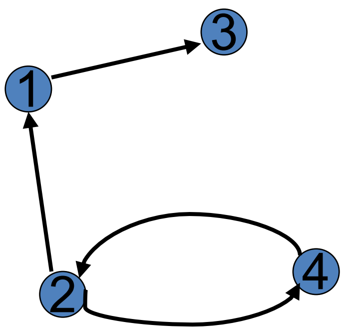
$$R^{(k)}[i,j] = R^{(k-1)}[i,j] \text{ or } (R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j])$$

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 (example)

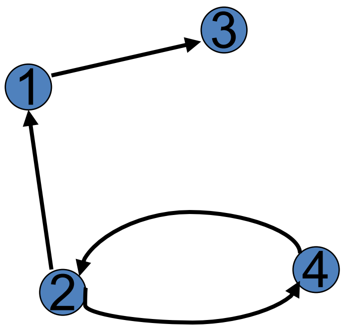


$$R^{(0)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix} \qquad R^{(1)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

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

Rule 2 If an element in row i and column j is 0 in $R^{(k\text{-}1)}$,
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 $R^{(k\text{-}1)}$

Warshall's Algorithm (example)



$$R^{(0)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

$$R^{(1)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{matrix}$$

$$R^{(2)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{matrix}$$

$$R^{(3)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

$$R^{(4)} = \begin{matrix} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix}$$

Warshall's Algorithm (pseudocode)

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 $R^{(n)}$

Warshall's Algorithm (Complexity)

Time efficiency: $O(n^3)$

Space efficiency: Matrices can be written over their predecessors

Learning outcomes

- Bellman-Ford Algorithm, to find the shortest paths in a graph (edges may have a negative weight) or detect a negative weighted cycle in a graph
- Floyd's Algorithm, to find all pair-shortest paths
- Warshall's Algorithm, to find transitive closure of a directed graph (Self-Study)