

# **FIT2004**

## **Algorithms and Data Structures**

Ian Wern Han Lim  
[lim.wern.han@monash.edu](mailto:lim.wern.han@monash.edu)

Referencing materials by  
Nathan Companeze, Aamir Cheema, Arun Konagurthu and Lloyd Allison



# Faculty of Information Technology, Monash University

---

## COMMONWEALTH OF AUSTRALIA

### *Copyright Regulations 1969*

This material has been reproduced and communicated to you by or on behalf of Monash University pursuant to Part VB of the Copyright Act 1968 (the Act). The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act. Do not remove this notice

Ready?

# Agenda

- More shortest distance algorithms

# Agenda

- More shortest distance algorithms
  - Remember we can get the path through back tracking

# Agenda

- More shortest distance algorithms
  - Remember we can get the path through back tracking
- Bellman-Ford
- Floyd-Warshall

# Agenda

- More shortest distance algorithms
  - Remember we can get the path through back tracking
- Bellman-Ford
- Floyd-Warshall
  - Warshall's algorithm for transitive closure

Let us begin...



# Dijkstra's Algorithm

## A recap

- Let us recap Dijkstra

# Dijkstra's Algorithm

## A recap

- Shortest distance algorithm

# Dijkstra's Algorithm

## A recap

- Shortest distance algorithm
  - Similar to BFS
    - Is BFS when the graph is unweighted

# Dijkstra's Algorithm

## A recap

- Shortest distance algorithm
  - Similar to BFS
    - Is BFS when the graph is unweighted
  - Uses a priority queue
    - Implemented with a min-heap

# Dijkstra's Algorithm

## A recap

- Shortest distance algorithm
  - Similar to BFS
    - Is BFS when the graph is unweighted
  - Uses a priority queue
    - Implemented with a min-heap
  - What is the complexity?

- Shortest distance algorithm
  - Similar to BFS
    - Is BFS when the graph is unweighted
  - Uses a priority queue
    - Implemented with a min-heap
  - What is the complexity?  $O(E \log V)$

- Shortest distance algorithm
  - Similar to BFS
    - Is BFS when the graph is unweighted
  - Uses a priority queue
    - Implemented with a min-heap
  - What is the complexity?  $O(E \log V)$
  
- Dijkstra is a...

- Shortest distance algorithm
  - Similar to BFS
    - Is BFS when the graph is unweighted
  - Uses a priority queue
    - Implemented with a min-heap
  - What is the complexity?  $O(E \log V)$
  
- Dijkstra is a...
  - Dynamic programming algorithm
  - Greedy algorithm



# Dijkstra's Algorithm

## A recap

- Shortest distance algorithm
  - Similar to BFS
    - Is BFS when the graph is unweighted
  - Uses a priority queue
    - Implemented with a min-heap
  - What is the complexity?  $O(E \log V)$
  
- Dijkstra is a...
  - Dynamic programming algorithm
  - Greedy algorithm

# Dijkstra's Algorithm

## A recap

- Shortest distance algorithm
  - Similar to BFS
    - Is BFS when the graph is unweighted
  - Uses a priority queue
    - Implemented with a min-heap
  - What is the complexity?  $O(E \log V)$
- Dijkstra is a...
  - Dynamic programming algorithm
  - Greedy algorithm
    - Might not work when there is a negative edge



Questions?

# Bellman Ford

Another shortest distance...

- Bellman-Ford isn't greedy

# Bellman Ford

Another shortest distance...

- Bellman-Ford isn't greedy
  - Does that mean it will work for negative edges?

- Bellman-Ford isn't greedy
  - Does that mean it will work for negative edges?
  - Does that mean the complexity will increase?

- Bellman-Ford isn't greedy
  - Does that mean it will work for negative edges?
  - Does that mean the complexity will increase?
    - Yes, we consider all edges
    - But we overcome it via Dynamic Programming

- Bellman-Ford isn't greedy
  - Does that mean it will work for negative edges?
  - Does that mean the complexity will increase?
    - Yes, we consider all edges
    - But we overcome it via Dynamic Programming
- 2 main components



- Bellman-Ford isn't greedy
  - Does that mean it will work for negative edges?
  - Does that mean the complexity will increase?
    - Yes, we consider all edges
    - But we overcome it via Dynamic Programming
- 2 main components
  - Distance calculation
  - Check for negative cycle

# Bellman Ford

Another shortest distance...

- Distance calculation
- Check for negative cycle

- Distance calculation
- Check for negative cycle
  - Check if there's a negative cycle...

- Distance calculation
- Check for negative cycle
  - Check if there's a negative cycle... If we have a negative cycle, there won't be a shortest distance... Why?

# Bellman Ford

Another shortest distance...

- Negative cycle is bad...



Questions?

- Distance calculation
  - Here we loop  $|V| - 1$  times
  
- Check for negative cycle
  - Check if there's a negative cycle... If we have a negative cycle, there won't be a shortest distance... Why?

- Distance calculation
  - Here we loop  $|V| - 1$  times. Why?
- Check for negative cycle
  - Check if there's a negative cycle... If we have a negative cycle, there won't be a shortest distance... Why?



- Distance calculation
  - Here we loop  $|V| - 1$  times. Why?  
This is the maximum number of jumps without a cycle in a graph.
- Check for negative cycle
  - Check if there's a negative cycle... If we have a negative cycle, there won't be a shortest distance... Why?

- Distance calculation

- Here we loop  $|V| - 1$  times. Why?

This is the maximum number of jumps without a cycle in a graph.

Going from vertex  $u$  to vertex  $v$ , you can only go through a maximum of  $|V| - 1$  edges without a cycle...

- Check for negative cycle

- Check if there's a negative cycle... If we have a negative cycle, there won't be a shortest distance... Why?

- Distance calculation

- Here we loop  $|V| - 1$  times. Why?

This is the maximum number of jumps without a cycle in a graph.

Going from vertex  $u$  to vertex  $v$ , you can only go through a maximum of  $|V| - 1$  edges without a cycle...

- Check for negative cycle

- Check if there's a negative cycle... If we have a negative cycle, there won't be a shortest distance... Why?
  - Then here, we repeat the process **ONE MORE TIME**

- Distance calculation

- Here we loop  $|V| - 1$  times. Why?

This is the maximum number of jumps without a cycle in a graph.

Going from vertex  $u$  to vertex  $v$ , you can only go through a maximum of  $|V| - 1$  edges without a cycle...

- Check for negative cycle

- Check if there's a negative cycle... If we have a negative cycle, there won't be a shortest distance... Why?
  - Then here, we repeat the process **ONE MORE TIME. Why?**

- Distance calculation

- Here we loop  $|V| - 1$  times. Why?

This is the maximum number of jumps without a cycle in a graph.

Going from vertex  $u$  to vertex  $v$ , you can only go through a maximum of  $|V| - 1$  edges without a cycle...

- Check for negative cycle

- Check if there's a negative cycle... If we have a negative cycle, there won't be a shortest distance... Why?

- Then here, we repeat the process **ONE MORE TIME. Why?**

If a cycle exist, this additional traversal will form the biggest cycle!

Questions?

# Bellman Ford

Another shortest distance...

- Let's look at the algorithm first, then I will explain from there...

# Bellman Ford

## Another shortest distance...

- Let's look at the algorithm first, then I will explain from there...

```
function BellmanFord(list vertices, list edges, vertex source)
    ::distance[],predecessor[]

    // This implementation takes in a graph, represented as
    // lists of vertices and edges, and fills two arrays
    // (distance and predecessor) with shortest-path
    // (less cost/distance/metric) information

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then distance[v] := 0
        else distance[v] := inf
        predecessor[v] := null

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge (u, v) with weight w in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]
```



```

function BellmanFord(list vertices, list edges, vertex source)
    ::distance[],predecessor[]

    // This implementation takes in a graph, represented as
    // lists of vertices and edges, and fills two arrays
    // (distance and predecessor) with shortest-path
    // (less cost/distance/metric) information

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then distance[v] := 0
        else distance[v] := inf
        predecessor[v] := null

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge (u, v) with weight w in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

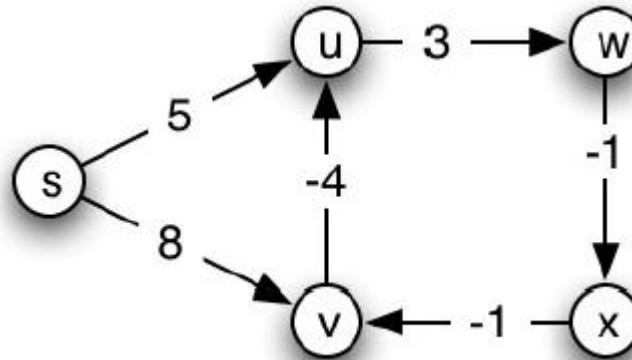
    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]

```

- Let's look at the algorithm first, then I will explain from there...
- Let us try with an example first...

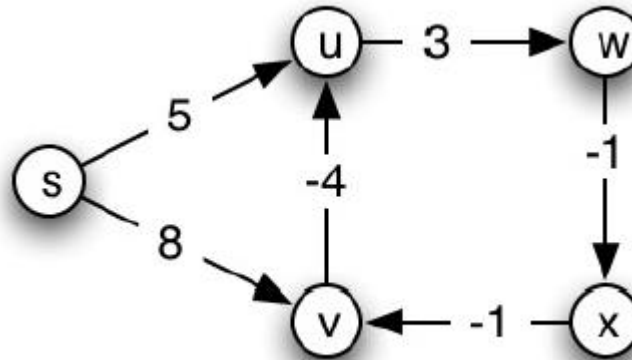
# Bellman Ford

Another shortest distance...



# Bellman Ford

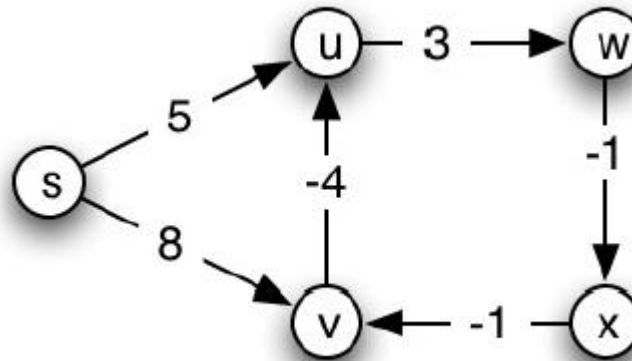
Another shortest distance...



s	
u	
v	
w	
x	

# Bellman Ford

Another shortest distance...



s	
u	
v	
w	
x	

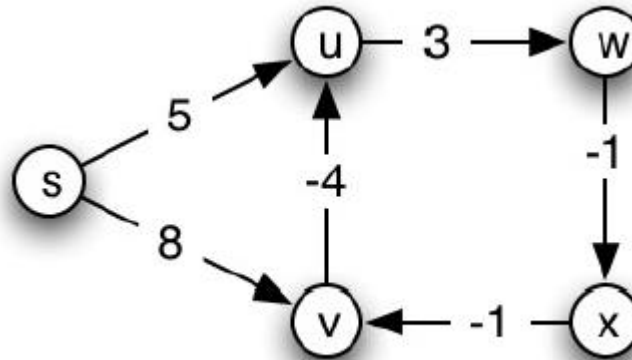
**VERTEX S IS  
WHERE WE BEGIN**

**But what's that  
shadowy place over there?**

**THERE IS  
WHERE WE CAN'T  
REACH... YET...**

# Bellman Ford

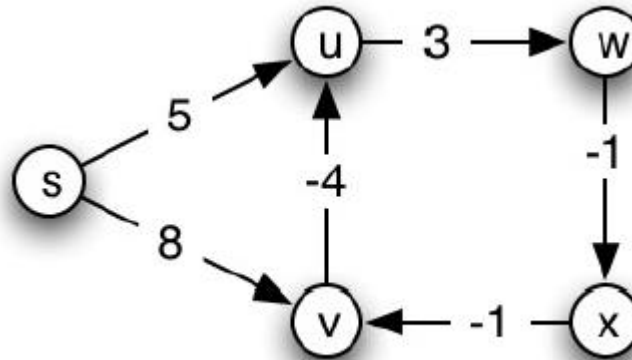
Another shortest distance...



	i=0		
s	0		
u	inf		
v	inf		
w	inf		
x	inf		

# Bellman Ford

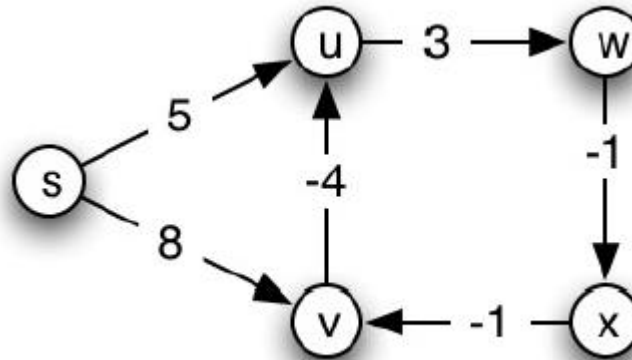
Another shortest distance...



	i=0	i=1
s	0	0
u	inf	5s
v	inf	8s
w	inf	inf
x	inf	inf

# Bellman Ford

Another shortest distance...

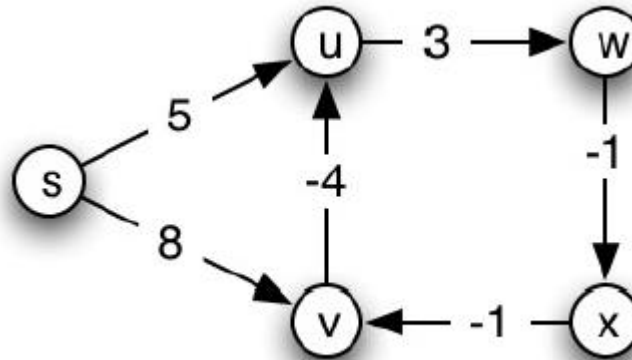


	i=0	i=1	i=2
s	0	0	0
u	inf	5s	4v
v	inf	8s	8s
w	inf	inf	8u
x	inf	inf	inf



# Bellman Ford

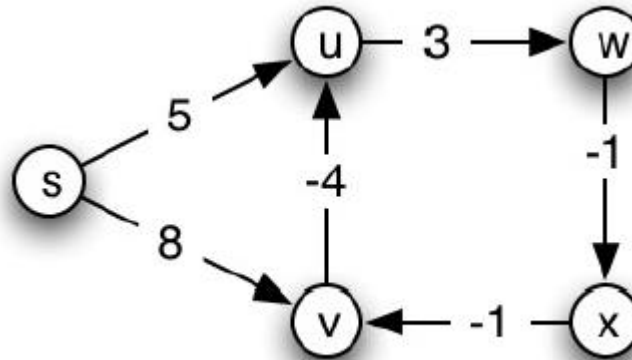
Another shortest distance...



	i=0	i=1	i=2	i=3
s	0	0	0	0
u	inf	5s	4v	4v
v	inf	8s	8s	8s
w	inf	inf	8u	7u
x	inf	inf	inf	7w

# Bellman Ford

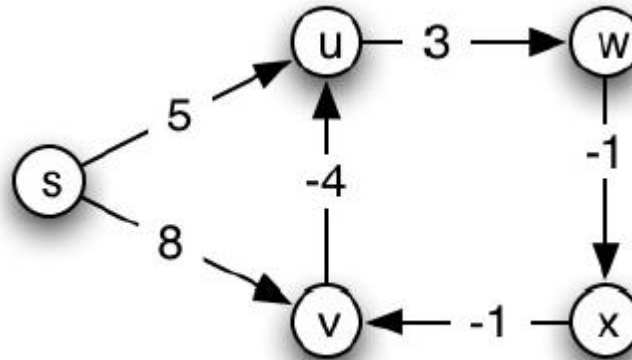
Another shortest distance...



	i=0	i=1	i=2	i=3	i=4
s	0	0	0	0	0
u	inf	5s	4v	4v	4v
v	inf	8s	8s	8s	6x
w	inf	inf	8u	7u	7u
x	inf	inf	inf	7w	6w

# Bellman Ford

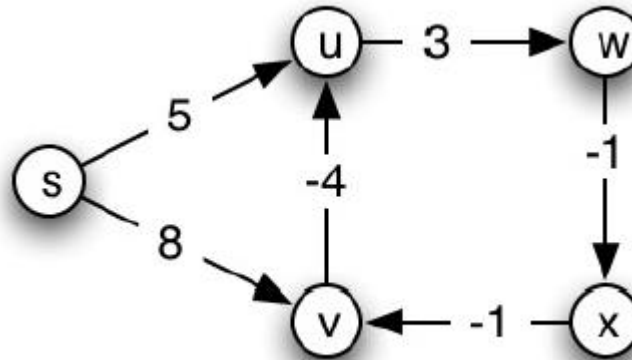
Another shortest distance...



	i=0	i=1	i=2	i=3	i=4
s	0	0	0	0	0
u	inf	5s	4v	4v	4v
v	inf	8s	8s	8s	6x
w	inf	inf	8u	7u	7u
x	inf	inf	inf	7w	6w

# Bellman Ford

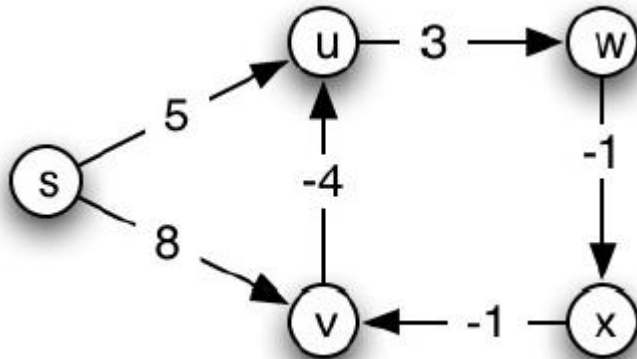
Another shortest distance...



	i=0	i=1	i=2	i=3	i=4	Checking	
s	0	0	0	0	0	0	
u	inf	5s	4v	4v	4v	2v	Break
v	inf	8s	8s	8s	6x	5x	Break too
w	inf	inf	8u	7u	7u	7u	
x	inf	inf	inf	7w	6w	6w	

# Bellman Ford

Another shortest distance...



	i=0	i=1	i=2	i=3	i=4	Checking	
s	0	0	0	0	0	0	
u	inf	5s	4v	4v	4v	2v	Break
v	inf	8s	8s	8s	6x	5x	Break too
w	inf	inf	8u	7u	7u	7u	
x	inf	inf	inf	7w	6w	6w	

Questions?

# Bellman Ford

Another shortest distance...

- What is our complexity here?

# Bellman Ford

## Another shortest distance...

- What is our complexity here?

```
function BellmanFord(List vertices, List edges, vertex source)
::distance[],predecessor[]

// This implementation takes in a graph, represented as
// lists of vertices and edges, and fills two arrays
// (distance and predecessor) with shortest-path
// (less cost/distance/metric) information

// Step 1: initialize graph
for each vertex v in vertices:
    if v is source then distance[v] := 0
    else distance[v] := inf
    predecessor[v] := null

// Step 2: relax edges repeatedly
for i from 1 to size(vertices)-1:
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            distance[v] := distance[u] + w
            predecessor[v] := u

// Step 3: check for negative-weight cycles
for each edge (u, v) with weight w in edges:
    if distance[u] + w < distance[v]:
        error "Graph contains a negative-weight cycle"
return distance[], predecessor[]
```



# Bellman Ford

## Another shortest distance...

- What is our complexity here?
- Initialize
- Calculate distance
- Check negative cycle

```
function BellmanFord(List vertices, List edges, vertex source)
    ::distance[],predecessor[]

    // This implementation takes in a graph, represented as
    // lists of vertices and edges, and fills two arrays
    // (distance and predecessor) with shortest-path
    // (less cost/distance/metric) information

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then distance[v] := 0
        else distance[v] := inf
        predecessor[v] := null

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge (u, v) with weight w in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]
```

# Bellman Ford

## Another shortest distance...

- What is our complexity here?
- Initialize
  - $O(V)$
- Calculate distance
- Check negative cycle

```
function BellmanFord(list vertices, list edges, vertex source)
    ::distance[],predecessor[]

    // This implementation takes in a graph, represented as
    // lists of vertices and edges, and fills two arrays
    // (distance and predecessor) with shortest-path
    // (less cost/distance/metric) information

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then distance[v] := 0
        else distance[v] := inf
        predecessor[v] := null

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge (u, v) with weight w in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]
```

# Bellman Ford

## Another shortest distance...

- What is our complexity here?
- Initialize
  - $O(V)$
- Calculate distance
  - $O(V)$  outer loop
- Check negative cycle

```
function BellmanFord(List vertices, List edges, vertex source)
    ::distance[],predecessor[]

    // This implementation takes in a graph, represented as
    // lists of vertices and edges, and fills two arrays
    // (distance and predecessor) with shortest-path
    // (less cost/distance/metric) information

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then distance[v] := 0
        else distance[v] := inf
        predecessor[v] := null

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge (u, v) with weight w in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]
```

# Bellman Ford

## Another shortest distance...

- What is our complexity here?
- Initialize
  - $O(V)$
- Calculate distance
  - $O(V)$  outer loop
  - $O(E)$  inner loop
- Check negative cycle

```
function BellmanFord(List vertices, List edges, vertex source)
    ::distance[],predecessor[]

    // This implementation takes in a graph, represented as
    // lists of vertices and edges, and fills two arrays
    // (distance and predecessor) with shortest-path
    // (less cost/distance/metric) information

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then distance[v] := 0
        else distance[v] := inf
        predecessor[v] := null

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge (u, v) with weight w in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]
```

# Bellman Ford

## Another shortest distance...

- What is our complexity here?
- Initialize
  - $O(V)$
- Calculate distance
  - $O(V)$  outer loop
  - $O(E)$  inner loop
- Check negative cycle
  - $O(E)$  again one last time

```
function BellmanFord(List vertices, List edges, vertex source)
    ::distance[],predecessor[]

    // This implementation takes in a graph, represented as
    // lists of vertices and edges, and fills two arrays
    // (distance and predecessor) with shortest-path
    // (less cost/distance/metric) information

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then distance[v] := 0
        else distance[v] := inf
        predecessor[v] := null

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge (u, v) with weight w in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]
```

# Bellman Ford

## Another shortest distance...

- What is our complexity here?
- Initialize
  - $O(V)$
- Calculate distance
  - $O(V)$  outer loop
  - $O(E)$  inner loop
- Check negative cycle
  - $O(E)$  again one last time
- Total?

```
function BellmanFord(List vertices, List edges, vertex source)
    ::distance[],predecessor[]

    // This implementation takes in a graph, represented as
    // lists of vertices and edges, and fills two arrays
    // (distance and predecessor) with shortest-path
    // (less cost/distance/metric) information

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then distance[v] := 0
        else distance[v] := inf
        predecessor[v] := null

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge (u, v) with weight w in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]
```

# Bellman Ford

## Another shortest distance...

- What is our complexity here?
- Initialize
  - $O(V)$
- Calculate distance
  - $O(V)$  outer loop
  - $O(E)$  inner loop
- Check negative cycle
  - $O(E)$  again one last time
- Total?  $O(VE)$

```
function BellmanFord(List vertices, List edges, vertex source)
    ::distance[],predecessor[]

    // This implementation takes in a graph, represented as
    // lists of vertices and edges, and fills two arrays
    // (distance and predecessor) with shortest-path
    // (less cost/distance/metric) information

    // Step 1: initialize graph
    for each vertex v in vertices:
        if v is source then distance[v] := 0
        else distance[v] := inf
        predecessor[v] := null

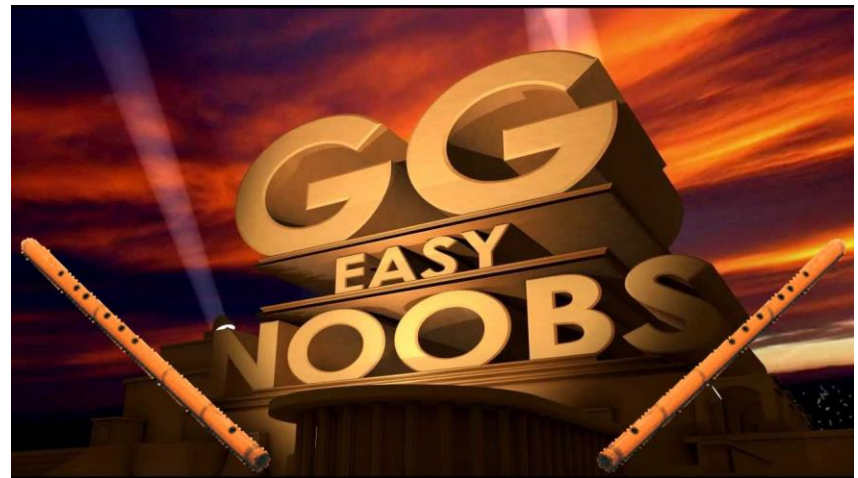
    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1:
        for each edge (u, v) with weight w in edges:
            if distance[u] + w < distance[v]:
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            error "Graph contains a negative-weight cycle"
    return distance[], predecessor[]
```

# Bellman Ford

Another shortest distance...

- What is our complexity here?
- Initialize
  - $O(V)$
- Calculate distance
  - $O(V)$  outer loop
  - $O(E)$  inner loop
- Check negative cycle
  - $O(E)$  again one last time
- Total?  $O(VE)$





Questions?

# Transitive Closure

## Reachability

- Given a graph  $G=(V,E)$

# Transitive Closure

## Reachability

- Given a graph  $G=(V,E)$
- Transitive closure is another graph  $G'=(V,E')$

# Transitive Closure

## Reachability

- Given a graph  $G=(V,E)$
- Transitive closure is another graph  $G'=(V,E')$ 
  - Same vertices  $V$

# Transitive Closure

## Reachability

- Given a graph  $G=(V,E)$
- Transitive closure is another graph  $G'=(V,E')$ 
  - Same vertices  $V$
  - Additional edges between vertex  $u$  and vertex  $v$  if there's a path between them

# Transitive Closure

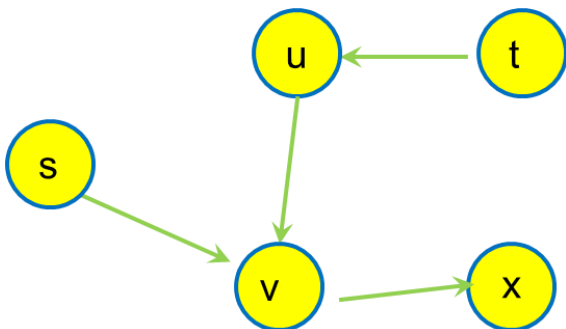
## Reachability

- Given a graph  $G=(V,E)$
- Transitive closure is another graph  $G'=(V,E')$ 
  - Same vertices  $V$
  - Additional edges between vertex  $u$  and vertex  $v$  if there's a path between them
  - Concept of transitivity
    - $A \rightarrow B, B \rightarrow C$  therefore  $A \rightarrow C$

# Transitive Closure

## Reachability

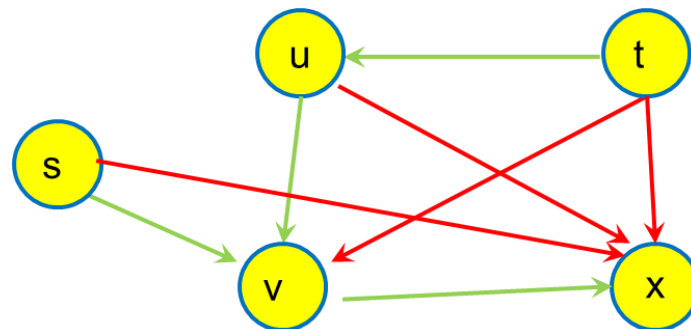
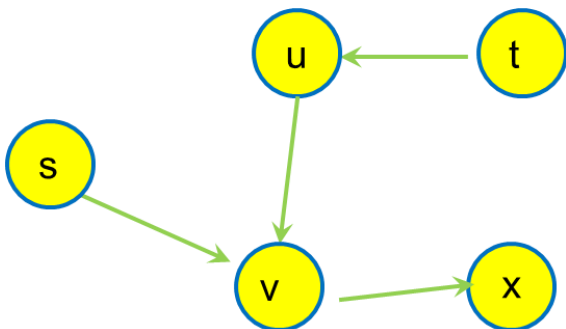
- Given a graph  $G=(V,E)$
- Transitive closure is another graph  $G'=(V,E')$ 
  - Same vertices  $V$
  - Additional edges between vertex  $u$  and vertex  $v$  if there's a path between them
  - Concept of transitivity
    - $A \rightarrow B, B \rightarrow C$  therefore  $A \rightarrow C$



# Transitive Closure

## Reachability

- Given a graph  $G=(V,E)$
- Transitive closure is another graph  $G'=(V,E')$ 
  - Same vertices  $V$
  - Additional edges between vertex  $u$  and vertex  $v$  if there's a path between them
  - Concept of transitivity
    - $A \rightarrow B, B \rightarrow C$  therefore  $A \rightarrow C$





Questions?

# Transitive Closure

## Reachability

- Given a graph  $G=(V,E)$
- Transitive closure is another graph  $G'=(V,E')$ 
  - Same vertices  $V$
  - Additional edges between vertex  $u$  and vertex  $v$  if there's a path between them
  - Concept of transitivity
    - $A \rightarrow B, B \rightarrow C$  therefore  $A \rightarrow C$
- So how do you do this?

# Transitive Closure

## Reachability

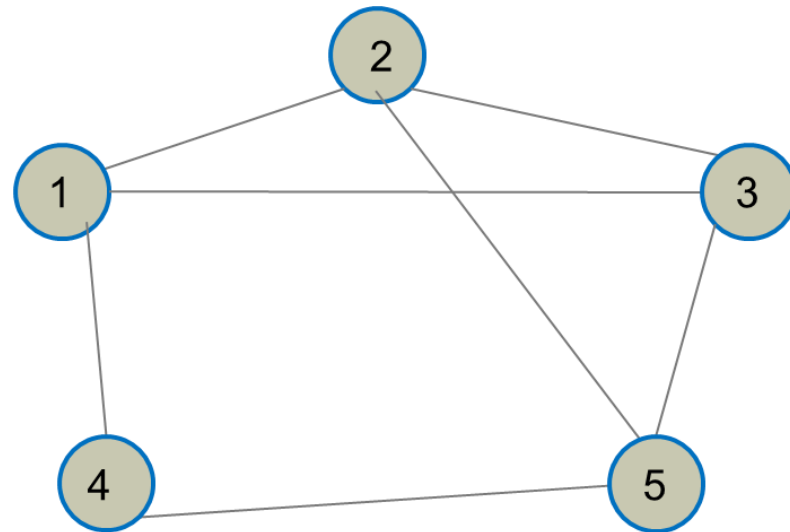
- Given a graph  $G=(V,E)$
- Transitive closure is another graph  $G'=(V,E')$ 
  - Same vertices  $V$
  - Additional edges between vertex  $u$  and vertex  $v$  if there's a path between them
  - Concept of transitivity
    - $A \rightarrow B, B \rightarrow C$  therefore  $A \rightarrow C$
- So how do you do this?
  - Recall our adjacency matrix

# Transitive Closure

## Reachability

- So how do you do this?
  - Recall our adjacency matrix

	1	2	3	4	5
1	F	T	T	T	F
2	T	F	T	F	T
3	T	T	F	F	T
4	T	F	F	F	T
5	F	T	T	T	F

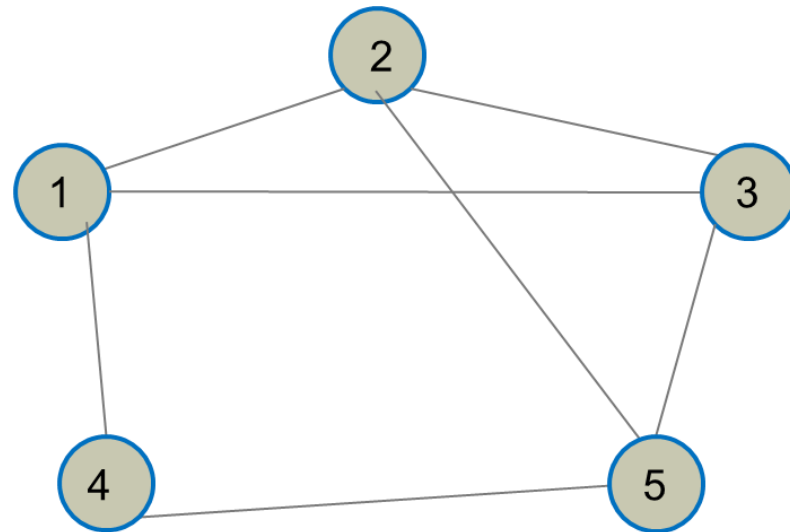


# Transitive Closure

## Reachability

- So how do you do this?
  - Recall our adjacency matrix
  - If  $\text{matrix}[1,2] = \text{True}$  and  $\text{matrix}[2,5] = \text{True}$

	1	2	3	4	5
1	F	T	T	T	F
2	T	F	T	F	T
3	T	T	F	F	T
4	T	F	F	F	T
5	F	T	T	T	F

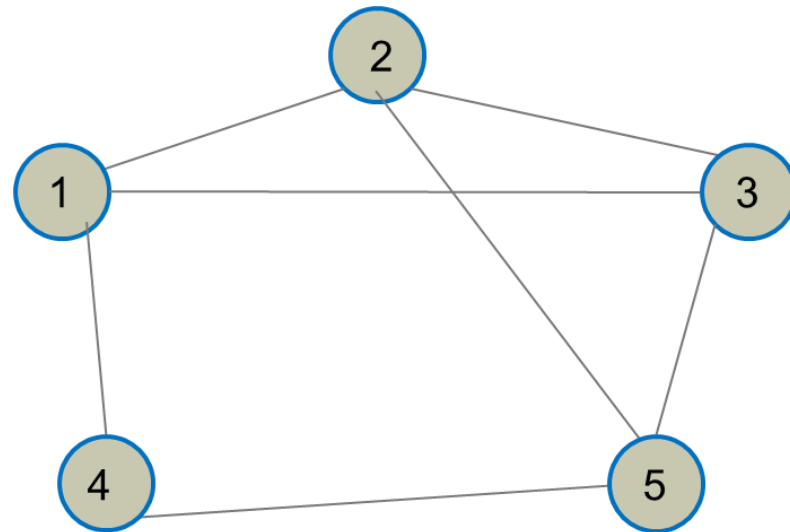


# Transitive Closure

## Reachability

- So how do you do this?
  - Recall our adjacency matrix
  - If  $\text{matrix}[1,2] = \text{True}$  and  $\text{matrix}[2,5] = \text{True}$
  - Then  $\text{matrix}[1,5] = \text{True}$

	1	2	3	4	5
1	F	T	T	T	F
2	T	F	T	F	T
3	T	T	F	F	T
4	T	F	F	F	T
5	F	T	T	T	F

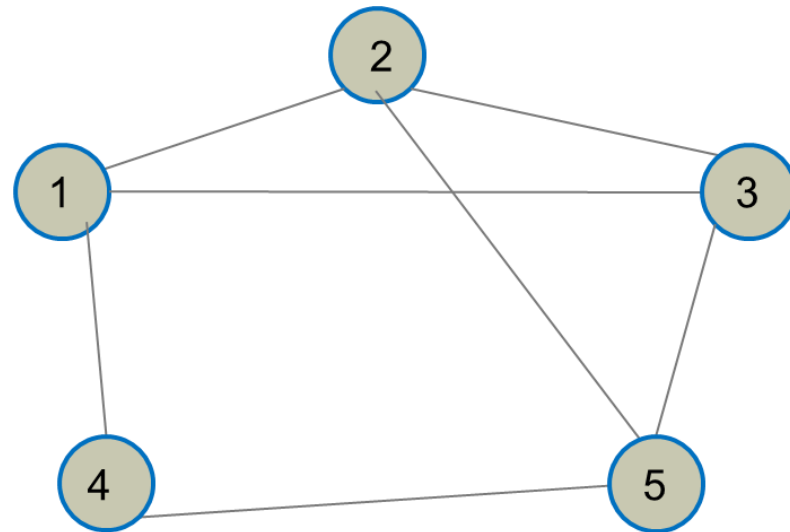


# Transitive Closure

## Reachability

- So how do you do this?
  - Recall our adjacency matrix
  - If  $\text{matrix}[1,2] = \text{True}$  and  $\text{matrix}[2,5] = \text{True}$
  - Then  $\text{matrix}[1,5] = \text{True}$
  - If  $\text{matrix}[i,j] = \text{True}$  and  $\text{matrix}[j,k] = \text{True}$  then  $\text{matrix}[i,k] = \text{True}$

	1	2	3	4	5
1	F	T	T	T	F
2	T	F	T	F	T
3	T	T	F	F	T
4	T	F	F	F	T
5	F	T	T	T	F



# Transitive Closure

## Reachability

- So how do you do this?
  - Recall our adjacency matrix
  - If  $\text{matrix}[1,2] = \text{True}$  and  $\text{matrix}[2,5] = \text{True}$
  - Then  $\text{matrix}[1,5] = \text{True}$
  - If  $\text{matrix}[i,j] = \text{True}$  and  $\text{matrix}[j,k] = \text{True}$  then  $\text{matrix}[i,k] = \text{True}$
- Can you code it?



# Transitive Closure

## Reachability

- So how do you do this?
  - Recall our adjacency matrix
  - If  $\text{matrix}[1,2] = \text{True}$  and  $\text{matrix}[2,5] = \text{True}$
  - Then  $\text{matrix}[1,5] = \text{True}$
  - If  $\text{matrix}[i,j] = \text{True}$  and  $\text{matrix}[j,k] = \text{True}$  then  $\text{matrix}[i,k] = \text{True}$
- Can you code it?

```
17 # warshall's
18 for k in range(count_vertex):
19     for i in range(count_vertex):
20         for j in range(count_vertex):
21             matrix[i][j] = matrix[i][j] or (matrix[i][k] and matrix[k][j])
```

Questions?

# Transitive Closure

## Reachability

- Can you code it?

```
17 # warshall's
18 for k in range(count_vertex):
19     for i in range(count_vertex):
20         for j in range(count_vertex):
21             matrix[i][j] = matrix[i][j] or (matrix[i][k] and matrix[k][j])
```

- What is the complexity?
  - Time =
  - Space =

# Transitive Closure

## Reachability

- Can you code it?

```
17 # warshall's
18 for k in range(count_vertex):
19     for i in range(count_vertex):
20         for j in range(count_vertex):
21             matrix[i][j] = matrix[i][j] or (matrix[i][k] and matrix[k][j])
```

- What is the complexity?
  - Time =  $O(V^3)$
  - Space =

# Transitive Closure

## Reachability

- Can you code it?

```
17 # warshall's
18 for k in range(count_vertex):
19     for i in range(count_vertex):
20         for j in range(count_vertex):
21             matrix[i][j] = matrix[i][j] or (matrix[i][k] and matrix[k][j])
```

- What is the complexity?
  - Time =  $O(V^3)$
  - Space =  $O(V^2)$  for the matrix

Questions?

# Floyd-Warshall

## Shortest distance

- So we know the Warshall's algorithm

# Floyd-Warshall

## Shortest distance

- So we know the Warshall's algorithm
  - $A \rightarrow B$  and  $B \rightarrow C$  give us  $A \rightarrow C$



# Floyd-Warshall

## Shortest distance

- So we know the Warshall's algorithm
  - $A \rightarrow B$  and  $B \rightarrow C$  give us  $A \rightarrow C$
- Thus
  - $\text{Distance}(A \rightarrow B) + \text{Distance}(B \rightarrow C) = \text{Distance}(A \rightarrow C)$

# Floyd-Warshall

## Shortest distance

- So we know the Warshall's algorithm
  - $A \rightarrow B$  and  $B \rightarrow C$  give us  $A \rightarrow C$
- Thus
  - $\text{Distance}(A \rightarrow B) + \text{Distance}(B \rightarrow C) = \text{Distance}(A \rightarrow C)$
  - So we can have the shortest distance from A to C, going through B

# Floyd-Warshall

## Shortest distance

- So we know the Warshall's algorithm
  - $A \rightarrow B$  and  $B \rightarrow C$  give us  $A \rightarrow C$
  
- Thus
  - $\text{Distance}(A \rightarrow B) + \text{Distance}(B \rightarrow C) = \text{Distance}(A \rightarrow C)$
  - So we can have the shortest distance from A to C, going through B
    - And we do this for all vertex u and vertex v

# Floyd-Warshall

## Shortest distance

- So we know the Warshall's algorithm
  - $A \rightarrow B$  and  $B \rightarrow C$  give us  $A \rightarrow C$
  
- Thus
  - $\text{Distance}(A \rightarrow B) + \text{Distance}(B \rightarrow C) = \text{Distance}(A \rightarrow C)$
  - So we can have the shortest distance from A to C, going through B
    - And we do this for all vertex u and vertex v
    - Thus, we have the shortest distance between all pairs!

- So we know the Warshall's algorithm
  - $A \rightarrow B$  and  $B \rightarrow C$  give us  $A \rightarrow C$
  
- Thus
  - $\text{Distance}(A \rightarrow B) + \text{Distance}(B \rightarrow C) = \text{Distance}(A \rightarrow C)$
  - So we can have the shortest distance from A to C, going through B
    - And we do this for all vertex u and vertex v
    - Thus, we have the shortest distance between all pairs!

# Floyd-Warshall

All-**Pair** shortest distance

- Why not Dijkstra?

- Why not Dijkstra?
  - Need to Dijkstra from every vertex. Complexity is?

- Why not Dijkstra?
  - Need to Dijkstra from every vertex. Complexity is?  
 $O(V) * O(E \log V) = O(EV \log V)$



- Why not Dijkstra?
  - Need to Dijkstra from every vertex. Complexity is?  
 $O(V) * O(E \log V) = O(EV \log V) = O(V^3 \log V)$

- Why not Dijkstra?
  - Need to Dijkstra from every vertex.  
 $O(V) * O(E \log V) = O(EV \log V) = O(V^3 \log V)$
- Why not Bellman-Ford?
  - Need to Bellman-Ford from every vertex

- Why not Dijkstra?
  - Need to Dijkstra from every vertex.  
 $O(V) * O(E \log V) = O(EV \log V) = O(V^3 \log V)$
- Why not Bellman-Ford?
  - Need to Bellman-Ford from every vertex.  
 $O(V) * O(VE) = O(V^2 E) = O(V^4)$

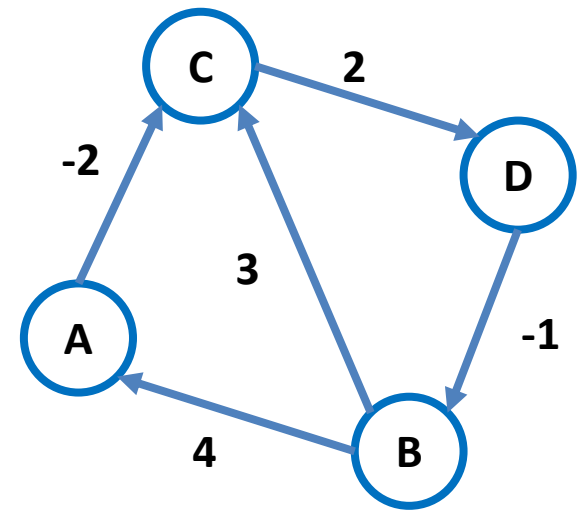
- Why not Dijkstra?
  - Need to Dijkstra from every vertex.  
 $O(V) * O(E \log V) = O(EV \log V) = O(V^3 \log V)$
- Why not Bellman-Ford?
  - Need to Bellman-Ford from every vertex.  
 $O(V) * O(VE) = O(V^2 E) = O(V^4)$
- Floyd-Warshall can do it quicker!

Questions?

# Floyd-Warshall

## All-Pair shortest distance

- Now let us do it manually...

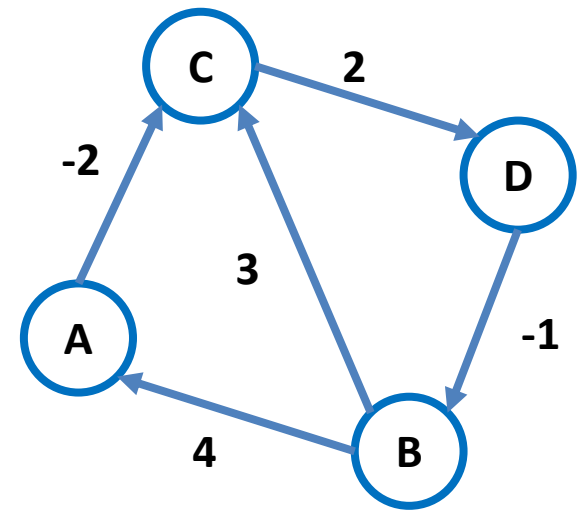


# Floyd-Warshall

## All-Pair shortest distance

- Now let us do it manually...

	A	B	C	D
A				
B				
C				
D				

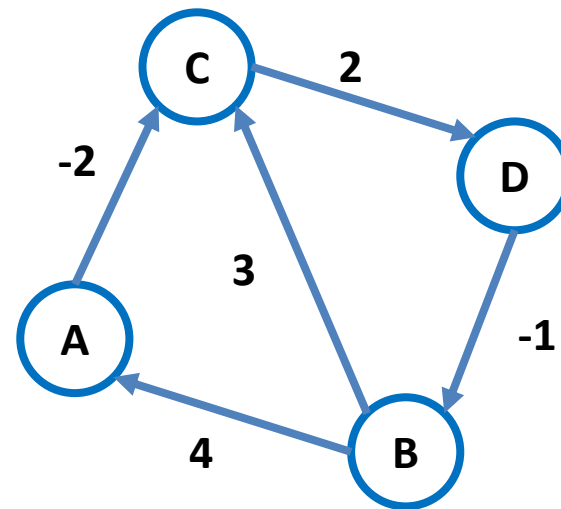


# Floyd-Warshall

## All-Pair shortest distance

- Now let us do it manually...
  - From itself back to itself is 0

	A	B	C	D
A	0			
B		0		
C			0	
D				0



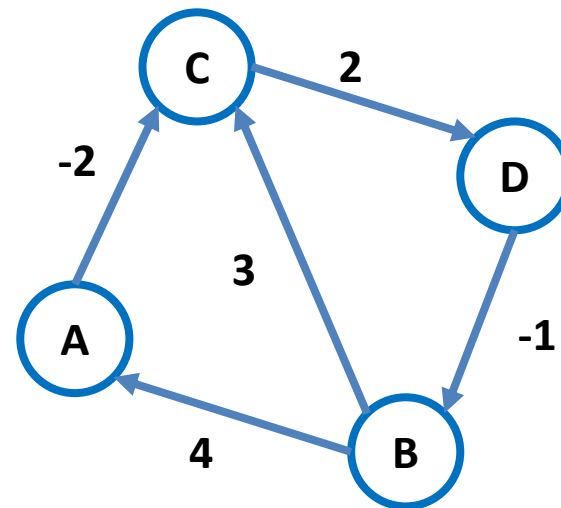


# Floyd-Warshall

## All-Pair shortest distance

- Now let us do it manually...
  - From itself back to itself is 0
  - All of the edges are added

	A	B	C	D
A	0		-2	
B	4	0	3	
C			0	2
D		-1		0

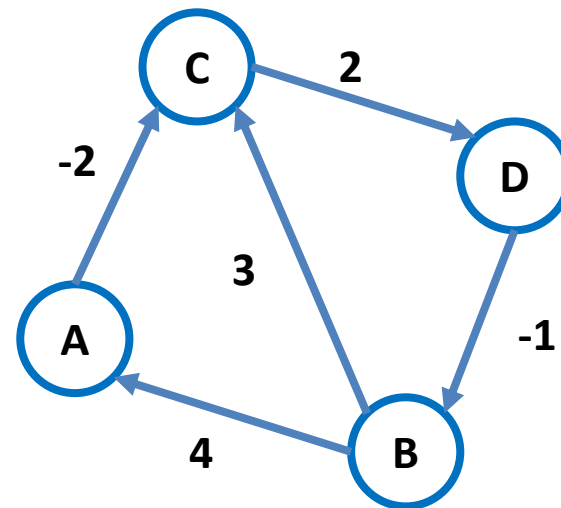


# Floyd-Warshall

## All-Pair shortest distance

- Now let us do it manually...
  - From itself back to itself is 0
  - All of the edges are added
  - Infinity for all of the non-edges

	A	B	C	D
A	0	inf	-2	inf
B	4	0	3	inf
C	inf	inf	0	2
D	inf	-1	inf	0



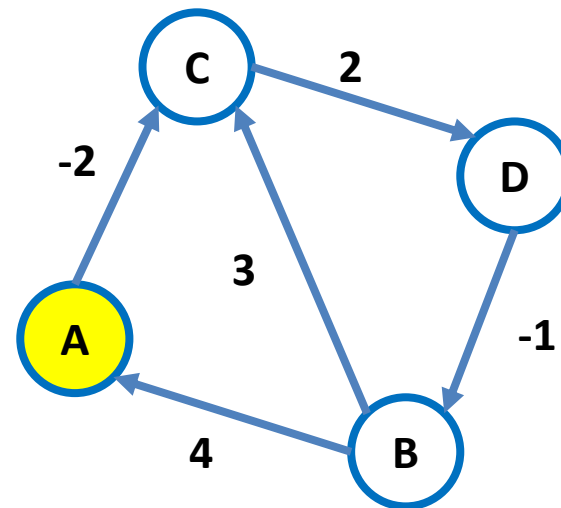
Questions?

# Floyd-Warshall

## All-Pair shortest distance

- What is the algorithm?
  - Begin with vertex A, can we update the shortest distance of all vertices going through A?

	A	B	C	D
A	0	inf	-2	inf
B	4	0	3	inf
C	inf	inf	0	2
D	inf	-1	inf	0

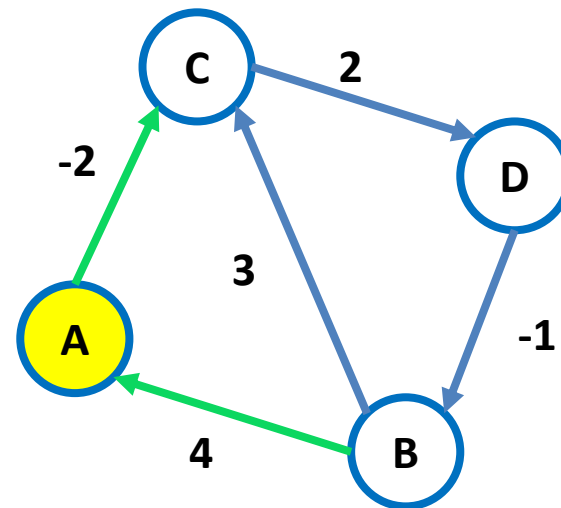


# Floyd-Warshall

## All-Pair shortest distance

- What is the algorithm?
  - Begin with vertex A, can we update the shortest distance of all vertices going through A?

	A	B	C	D
A	0	inf	-2	inf
B	4	0	2	inf
C	inf	inf	0	2
D	inf	-1	inf	0

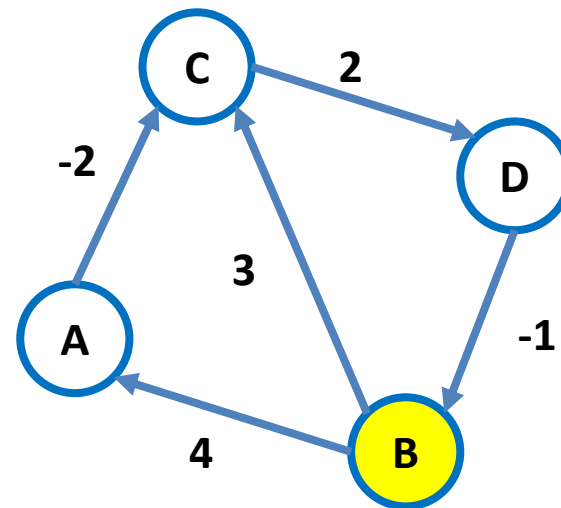


# Floyd-Warshall

## All-Pair shortest distance

- What is the algorithm?
  - Begin with vertex A, can we update the shortest distance of all vertices going through A?
  - Begin with vertex B, can we update the shortest distance of all vertices going through B?

	A	B	C	D
A	0	inf	-2	inf
B	4	0	2	inf
C	inf	inf	0	2
D	inf	-1	inf	0

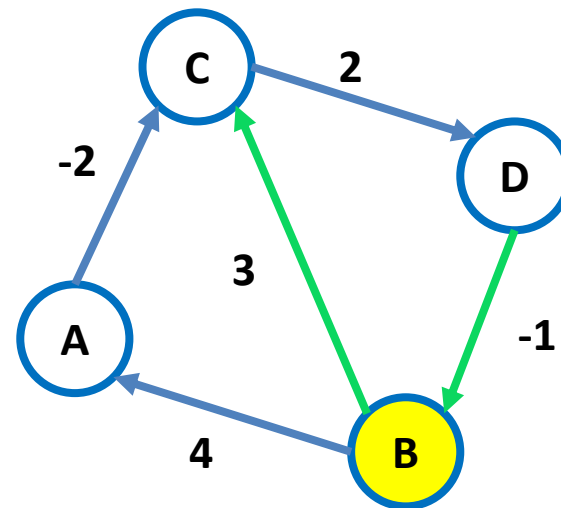


# Floyd-Warshall

## All-Pair shortest distance

- What is the algorithm?
  - Begin with vertex A, can we update the shortest distance of all vertices going through A?
  - Begin with vertex B, can we update the shortest distance of all vertices going through B?

	A	B	C	D
A	0	inf	-2	inf
B	4	0	2	inf
C	inf	inf	0	2
D	inf	-1	2	0

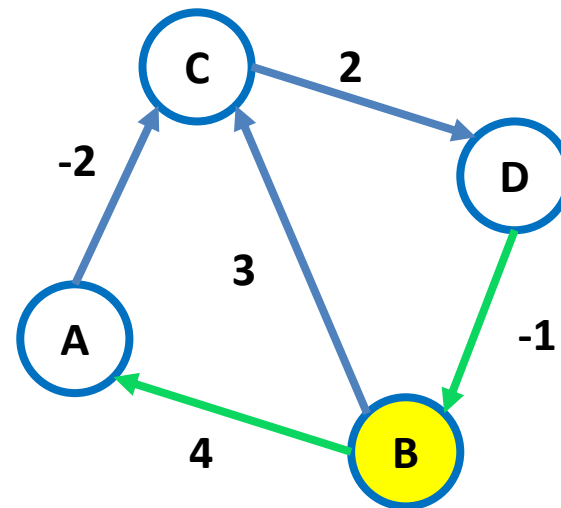


# Floyd-Warshall

## All-Pair shortest distance

- What is the algorithm?
  - Begin with vertex A, can we update the shortest distance of all vertices going through A?
  - Begin with vertex B, can we update the shortest distance of all vertices going through B?

	A	B	C	D
A	0	inf	-2	inf
B	4	0	2	inf
C	inf	inf	0	2
D	3	-1	2	0





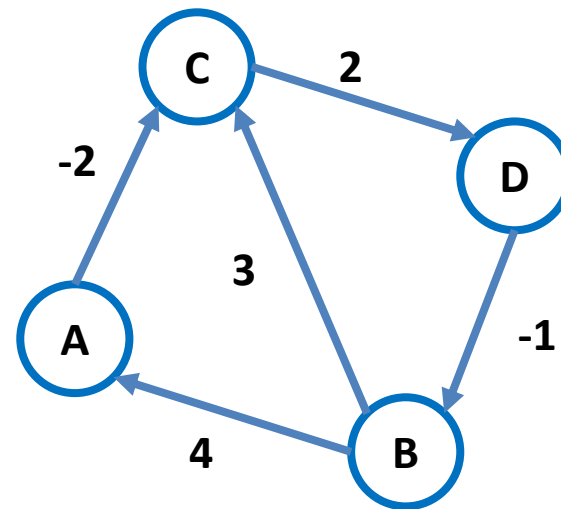
Questions?

# Floyd-Warshall

## All-Pair shortest distance

- What is the algorithm?

	A	B	C	D
A	0	inf	-2	inf
B	4	0	3	inf
C	inf	inf	0	2
D	inf	-1	inf	0



# Floyd-Warshall

## All-Pair shortest distance

- What is the algorithm?
  - Same as the Warshall's

- What is the algorithm?
  - Same as the Warshall's
  - Except we are now doing distance...

- What is the algorithm?
  - Same as the Warshall's
  - Except we are now doing distance...
  - If  $\text{distance}[i][j] > \text{distance}[i][k] + \text{distance}[k][j]$   
then  $\text{distance}[i][j] = \text{distance}[i][k] + \text{distance}[k][j]$

- What is the algorithm?
  - Same as the Warshall's
  - Except we are now doing distance...
  - If  $\text{distance}[i][j] > \text{distance}[i][k] + \text{distance}[k][j]$   
then  $\text{distance}[i][j] = \text{distance}[i][k] + \text{distance}[k][j]$

```
21  # floyd warshall's
22  for k in range(count_vertex):
23      for i in range(count_vertex):
24          for j in range(count_vertex):
25              matrix[i][j] = min(matrix[i][j], matrix[i][k]+matrix[k][j])
```

# Floyd-Warshall

## All-Pair shortest distance

- What is the algorithm?
  - Same as the Warshall's
  - Except we are now doing distance...
  - If  $\text{distance}[i][j] > \text{distance}[i][k] + \text{distance}[k][j]$   
then  $\text{distance}[i][j] = \text{distance}[i][k] + \text{distance}[k][j]$

```
21  # floyd warshall's
22  for k in range(count_vertex):
23      for i in range(count_vertex):
24          for j in range(count_vertex):
25              matrix[i][j] = min(matrix[i][j], matrix[i][k]+matrix[k][j])
```

- What is the meaning of the outer loop?

# Floyd-Warshall

## All-Pair shortest distance

- What is the algorithm?
  - Same as the Warshall's
  - Except we are now doing distance...
  - If  $\text{distance}[i][j] > \text{distance}[i][k] + \text{distance}[k][j]$   
then  $\text{distance}[i][j] = \text{distance}[i][k] + \text{distance}[k][j]$

```
21  # floyd warshall's
22  for k in range(count_vertex):
23      for i in range(count_vertex):
24          for j in range(count_vertex):
25              matrix[i][j] = min(matrix[i][j], matrix[i][k]+matrix[k][j])
```

- What is the meaning of the outer loop?
  - As we increment, we find the minimum distance going through vertex k. Thus, we would have the minimum through every vertex, updating as needed!



Questions?

# Floyd-Warshall

## All-Pair shortest distance

- What is the complexity?

```
21  # floyd warshall's
22  for k in range(count_vertex):
23      for i in range(count_vertex):
24          for j in range(count_vertex):
25              matrix[i][j] = min(matrix[i][j], matrix[i][k]+matrix[k][j])
```

# Floyd-Warshall

## All-Pair shortest distance

- What is the complexity?
  - Same as Warshall's

```
21  # floyd warshall's
22  for k in range(count_vertex):
23      for i in range(count_vertex):
24          for j in range(count_vertex):
25              matrix[i][j] = min(matrix[i][j], matrix[i][k]+matrix[k][j])
```

Questions?

# Floyd-Warshall

## All-Pair shortest distance

- What about negative cycle?

- What about negative cycle?
  - We know about cycle by looking at the diagonal going from vertex  $u$  back to vertex  $u$

- What about negative cycle?
  - We know about cycle by looking at the diagonal going from vertex  $u$  back to vertex  $u$
  - So if we have negative cycle, the diagonal will tell us!  
If the value is negative =(

Questions?



# Shortest Distance

## Shortest Paths

- Can you summarize it up?

# Shortest Distance

## Shortest Paths

- Can you summarize it up?
- Un-weighted graph?
- Weighted graph?

# Shortest Distance

## Shortest Paths

- Can you summarize it up?
- Un-weighted graph?
- Weighted graph?
  - Handle negative edge?

# Shortest Distance

## Shortest Paths

- Can you summarize it up?
- Un-weighted graph?
  - BFS
  - Dijkstra
- Weighted graph?
  - Handle negative edge?

# Shortest Distance

## Shortest Paths

- Can you summarize it up?
- Un-weighted graph?
  - BFS
  - Dijkstra
- Weighted graph?
  - Dijkstra
  - Handle negative edge?

# Shortest Distance

## Shortest Paths

- Can you summarize it up?
- Un-weighted graph?
  - BFS
  - Dijkstra
- Weighted graph?
  - Dijkstra
  - Handle negative edge?
    - Bellman-Ford
    - Floyd-Warshall (also all pairs)

# Shortest Distance

## Shortest Paths

- Un-weighted graph?
  - BFS
  - Dijkstra
- Weighted graph?
  - Dijkstra
  - Handle negative edge?
    - Bellman-Ford
    - Floyd-Warshall (also all pairs)
  - Handle negative cycle?

# Shortest Distance

## Shortest Paths

- Un-weighted graph?
  - BFS
  - Dijkstra
- Weighted graph?
  - Dijkstra
  - Handle negative edge?
    - Bellman-Ford
    - Floyd-Warshall (also all pairs)
  - Handle negative cycle?





Questions?

Thank You