MONASH University

MONASH
INFORMATION
TECHNOLOGY

FIT2004
Algorithms and Data Structures

Ian Wern Han Lim
lim.wern.han@monash.edu

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

GROUP OF EIGHT AUSTRALIA

Faculty of Information Technology, Monash University

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

Monash University

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?

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'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…

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

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

MONASH University

- 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

MONASH University

- 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?

Another shortest distance…

- Negative cycle is bad…

Questions?

Bellman Ford

Another shortest distance…

- 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?

Another shortest distance…

- 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?

Bellman Ford

Another shortest distance…

- 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?

Bellman Ford

Another shortest distance…

- 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?

Bellman Ford

Another shortest distance…

- 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

Bellman Ford

Another shortest distance…

- 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?

Bellman Ford

Another shortest distance…

- 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[]
```

Bellman Ford

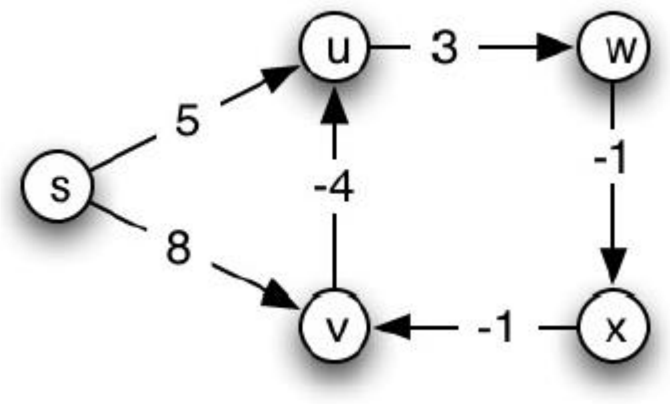
Another shortest distance…

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

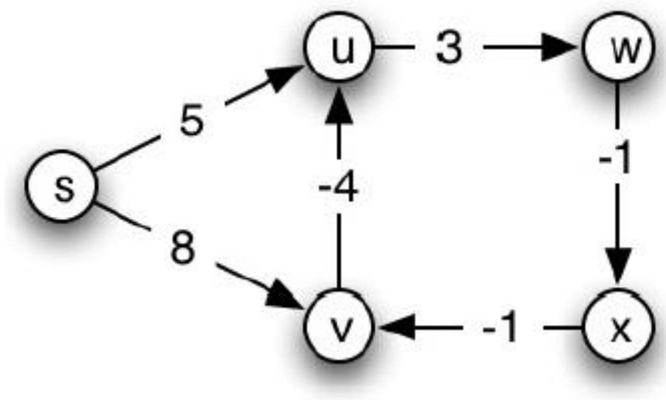
- Let us try with an example first…

Bellman Ford

Another shortest distance…



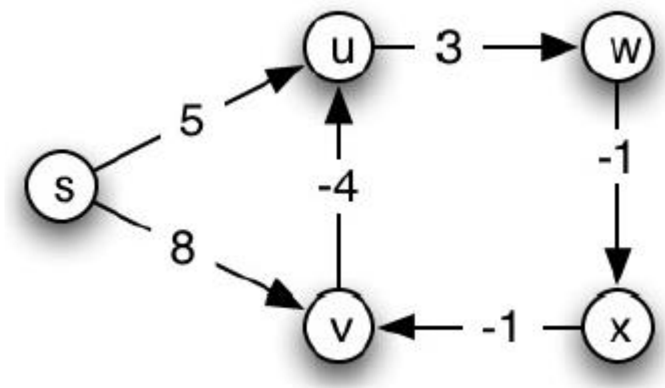
Another shortest distance…



s
u
v
w
x

Bellman Ford

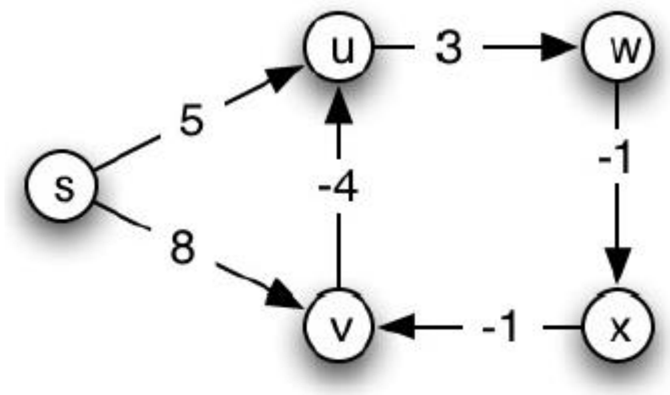
Another shortest distance…



s
u
v
w
x

Bellman Ford

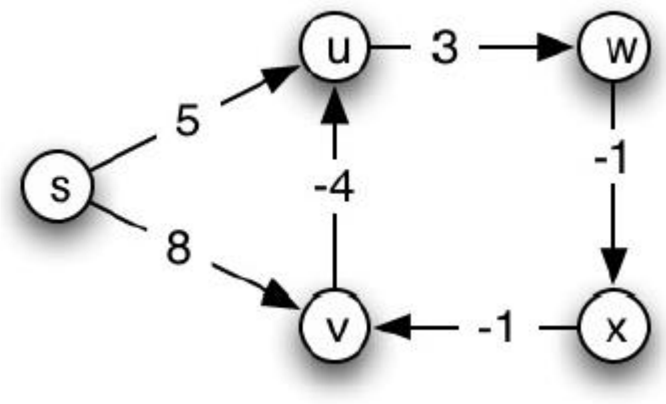
Another shortest distance…



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

Bellman Ford

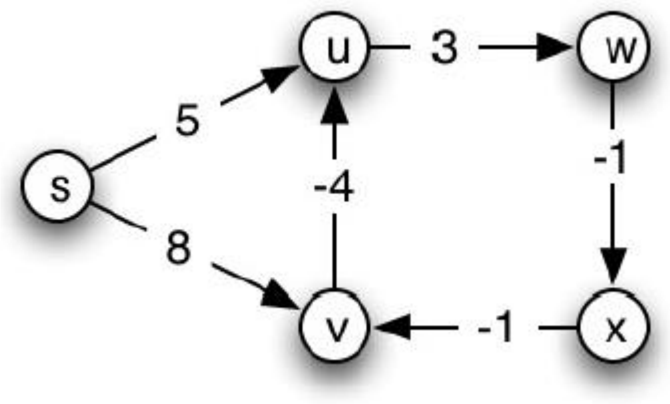
Another shortest distance…



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

Bellman Ford

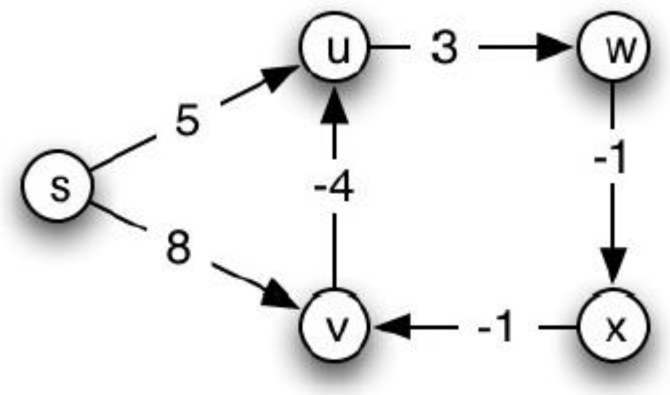
Another shortest distance…



	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

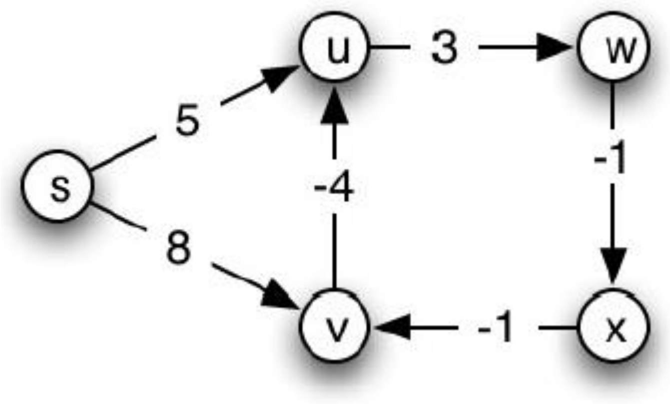
Another shortest distance…



	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

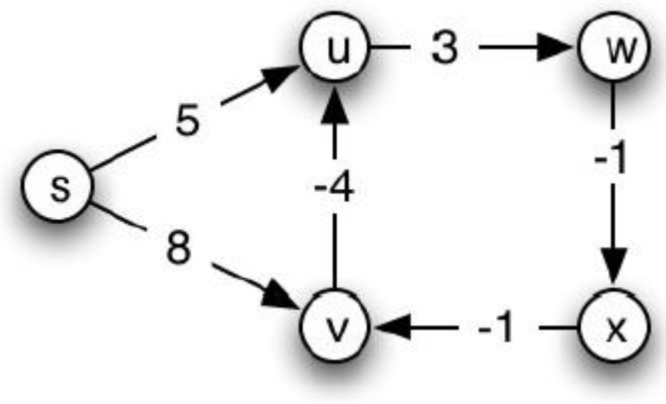
Another shortest distance…



	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

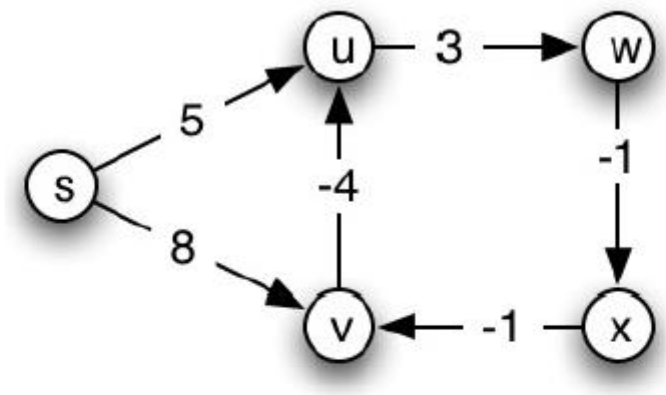
Another shortest distance…



	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

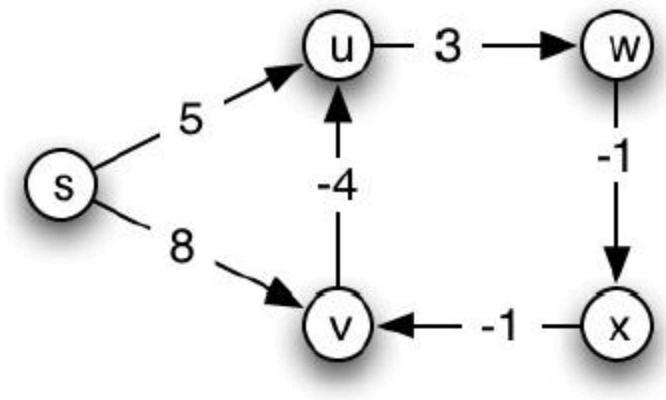
Another shortest distance…



	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)

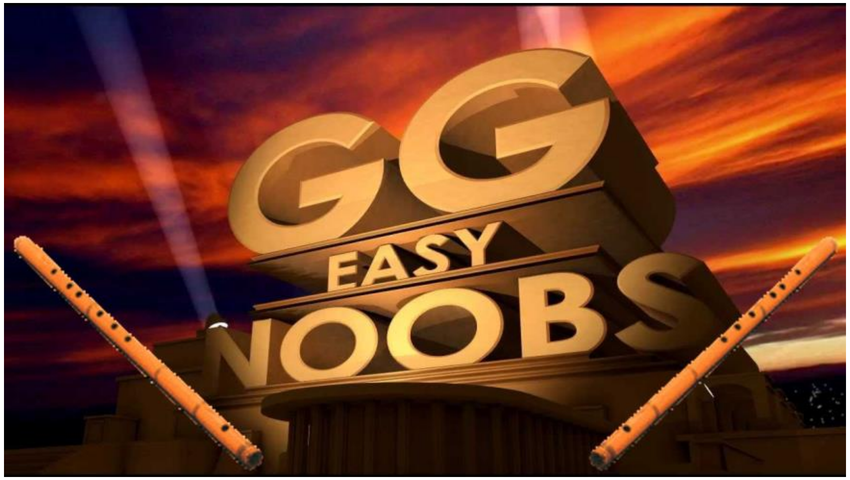
```
function BellmanFord(list vertices, list edges, vertex source)
   ::distance[],predecessor[]

   // 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)

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

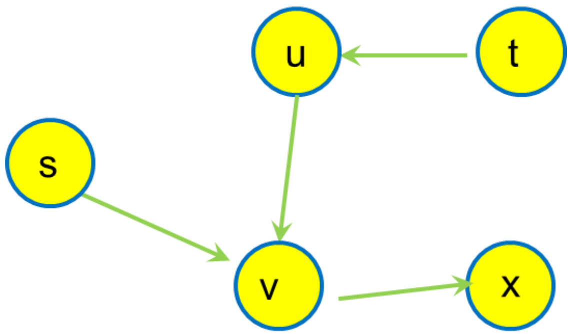
- Given a graph G=(V,E)
- Transitive closure is another graph G’=(V,E’)
  - Same vertices V

- 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 -> B, B -> C therefore A -> C

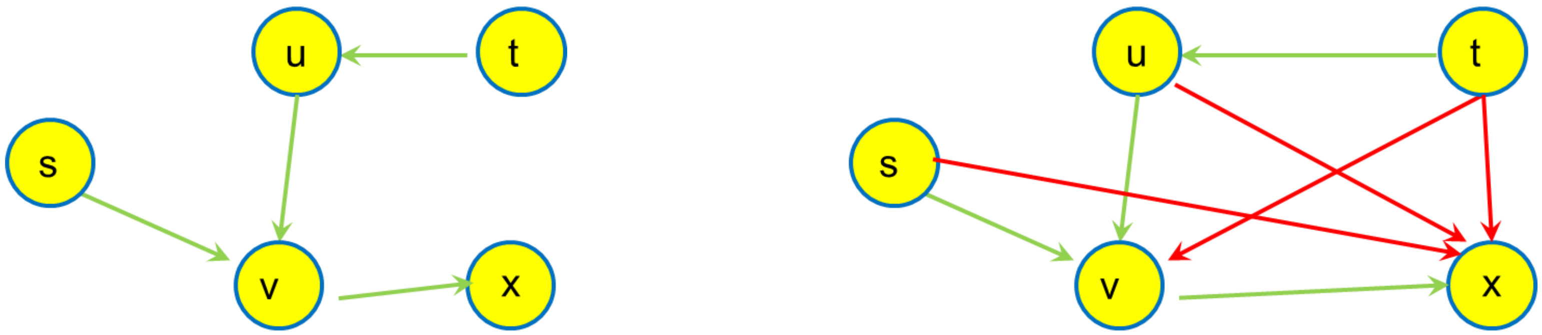
- Given a graph G=(V,E)
- Transitive closure is another graph G'=(V,E')
  - Same vertices V
  - Additional edges between vertex u and vertex v if there's a path between them
  - Concept of transitivity
    - A -> B, B -> C therefore A -> C



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 -> B, B -> C therefore A -> 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 -> B, B -> C therefore A -> 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 -> B, B -> C therefore A -> C

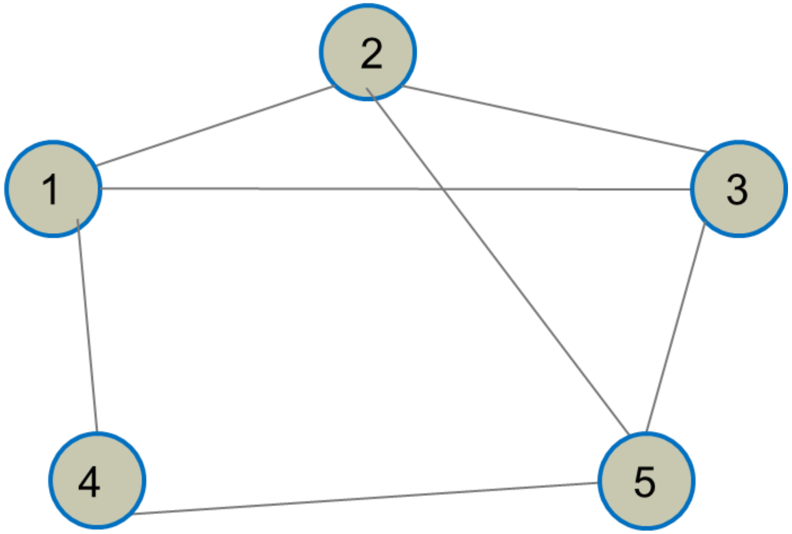
- So how do you do this?
  - Recall our adjacency matrix

Transitive Closure

Reachability

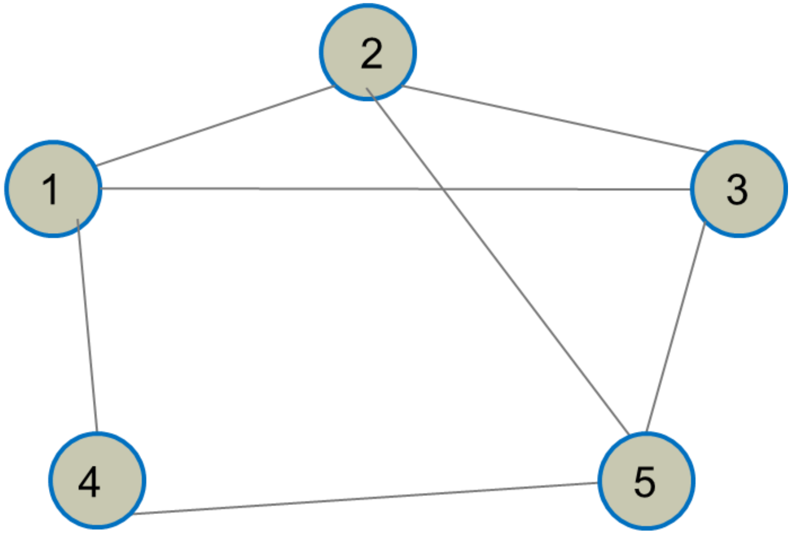
- So how do you do this?
  - Recall our adjacency matrix

	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



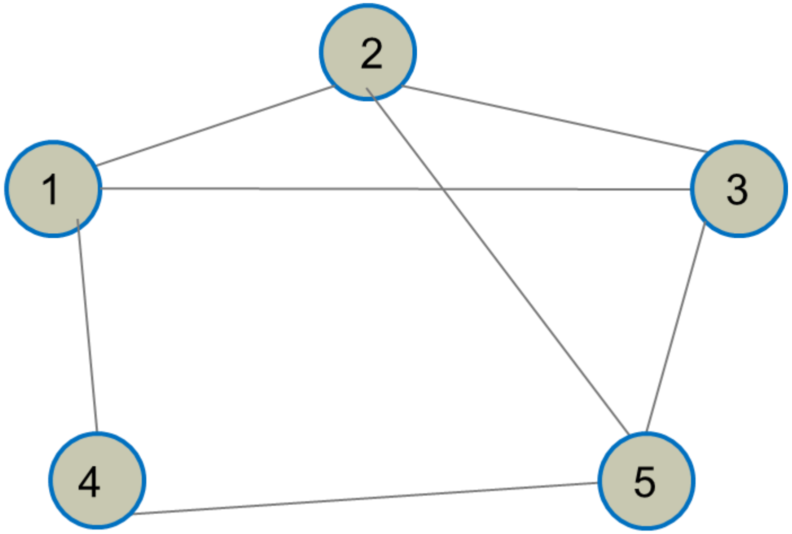
- So how do you do this?
  - Recall our adjacency matrix
  - If matrix[1,2] = True and matrix [2,5] = True

	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



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

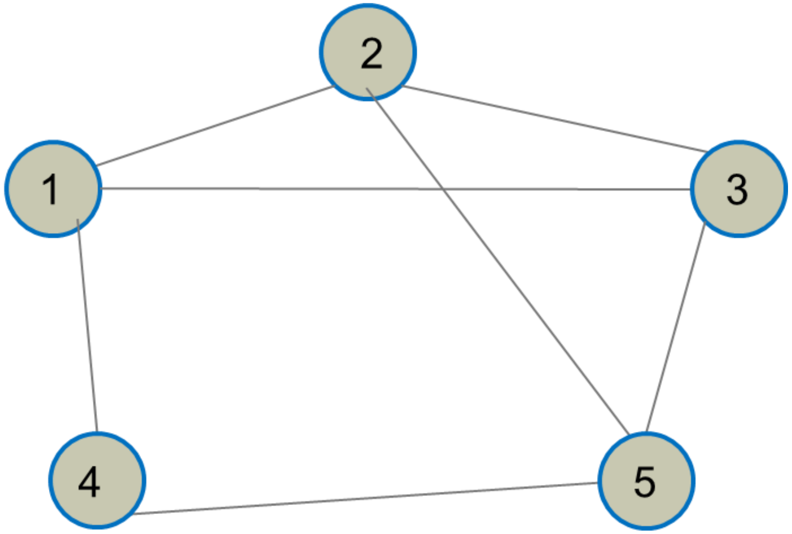
	1	2	3	4	5
1	F	T	T	T	F
2	T	F	T	F	T
3	T	T	F	F	T
4	T	F	F	F	T
5	F	T	T	T	F



Transitive Closure
Reachability

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



MONASH University

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

Transitive Closure

Reachability

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

- Can you code it?

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

Questions?

Transitive Closure

Reachability

- Can you code it?

```python
# warshall's
for k in range(count_vertex):
    for i in range(count_vertex):
        for j in range(count_vertex):
            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?

```
# warshall's
for k in range(count_vertex):
    for i in range(count_vertex):
        for j in range(count_vertex):
            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?

```
# warshall's
for k in range(count_vertex):
    for i in range(count_vertex):
        for j in range(count_vertex):
            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

MONASH University

Questions?

Floyd-Warshall

Shortest distance

- So we know the Warshall's algorithm

- So we know the Warshall's algorithm
  - A -> B and B -> C give us A -> C

- So we know the Warshall's algorithm
  - A -> B and B -> C give us A -> C

- Thus
  - Distance(A -> B) + Distance(B -> C) = Distance(A -> C)

MONASH University

- So we know the Warshall's algorithm
  - A -> B and B -> C give us A -> C

- Thus
  - Distance(A -> B) + Distance(B -> C) = Distance(A -> C)
  - So we can have the shortest distance from A to C, going through B

- So we know the Warshall’s algorithm
  - A -> B and B -> C give us A -> C
- Thus
  - Distance(A -> B) + Distance(B -> C) = Distance(A -> 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

- So we know the Warshall's algorithm
  - A -> B and B -> C give us A -> C
- Thus
  - Distance(A -> B) + Distance(B -> C) = Distance(A -> 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!

All-Pair shortest distance

- So we know the Warshall's algorithm
  - A -> B and B -> C give us A -> C

- Thus
  - Distance(A -> B) + Distance(B -> C) = Distance(A -> 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!

- Why not Dijkstra?

Floyd-Warshall
All-Pair shortest distance

- 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)

MONASH University

- 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)

Floyd-Warshall

All-Pair shortest distance

- 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

Floyd-Warshall

All-Pair shortest distance

- 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

All-Pair shortest distance

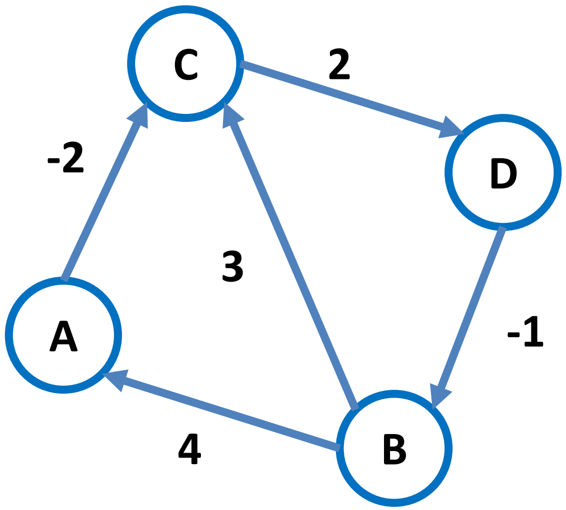
- Why not Dijkstra?
  - Need to Dijkstra from every vertex.
    O(V) * O(E log V) = O(EV log V) = O(V^3 log V)

- 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?

- Now let us do it manually…

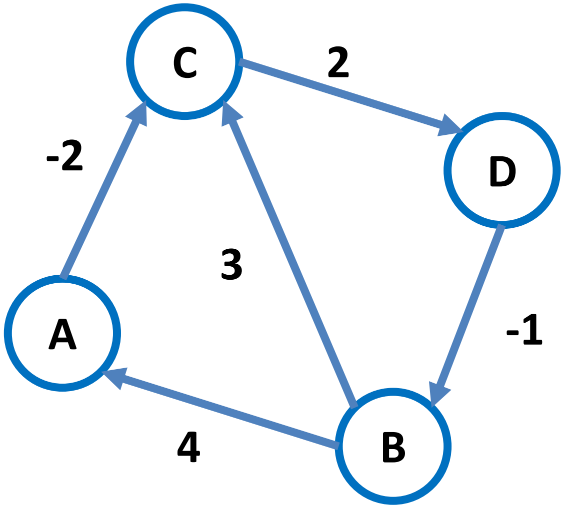


Floyd-Warshall

All-Pair shortest distance

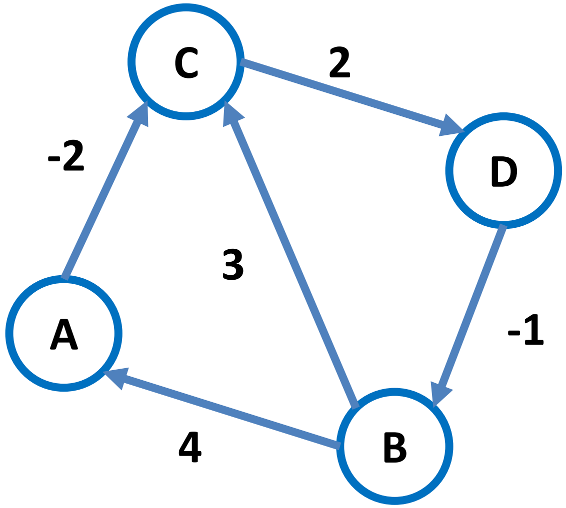
- Now let us do it manually…

	A	B	C	D
A				
B				
C				
D				



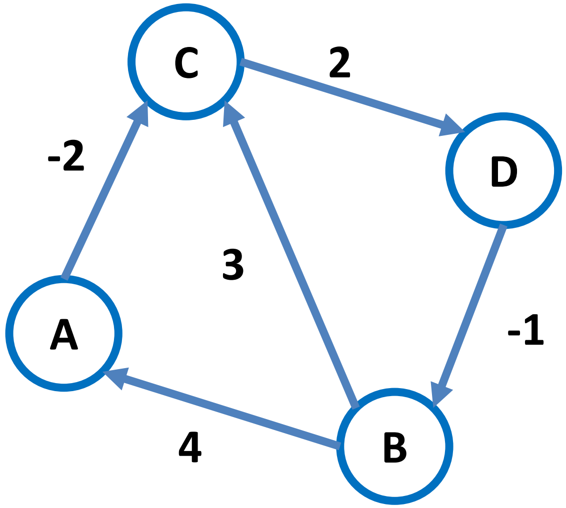
- Now let us do it manually…
  - From itself back to itself is 0

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



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

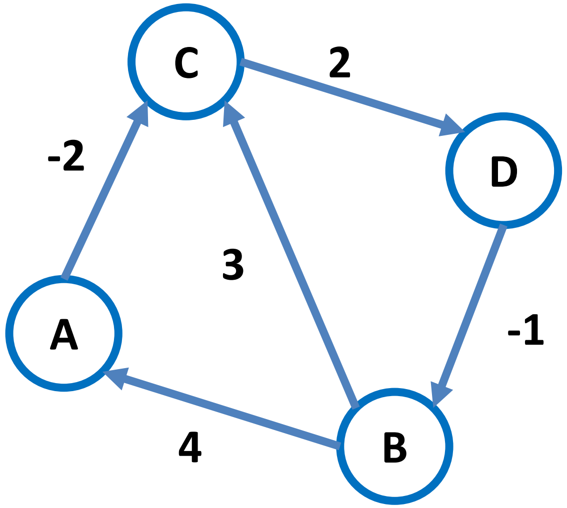
	A	B	C	D
A	0		-2	
B	4	0	3	
C			0	2
D		-1		0



MONASH University

- 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

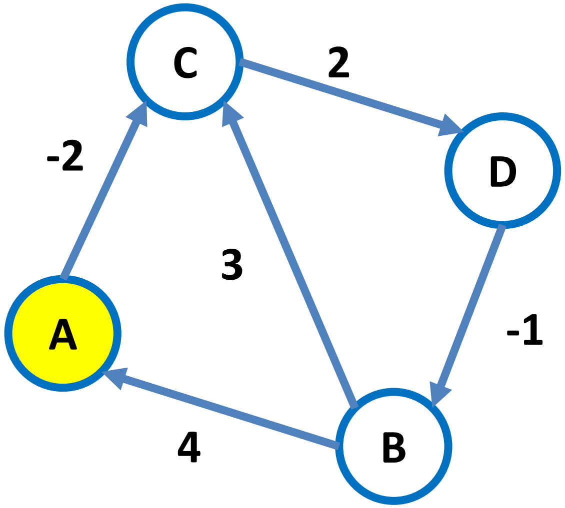


Questions?

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?

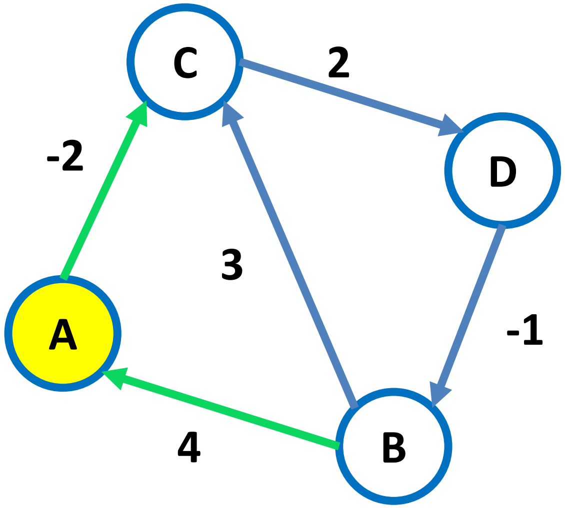
	A	B	C	D
A	0	inf	-2	inf
B	4	0	3	inf
C	inf	inf	0	2
D	inf	-1	inf	0



Floyd-Warshall
All-Pair shortest distance

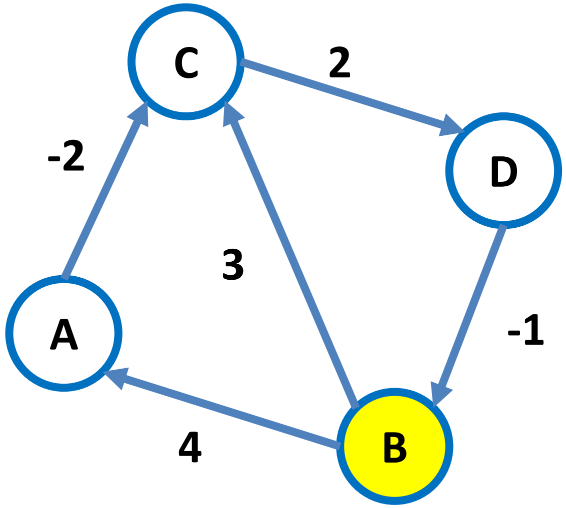
- What is the algorithm?
  - Begin with vertex A, can we update the shortest distance of all vertices going through A?

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



- 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?

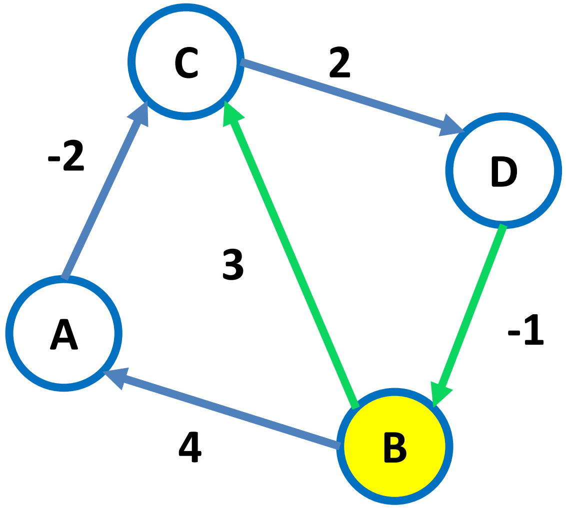
	A	B	C	D
A	0	inf	-2	inf
B	4	0	2	inf
C	inf	inf	0	2
D	inf	-1	inf	0



MONASH University

- 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?

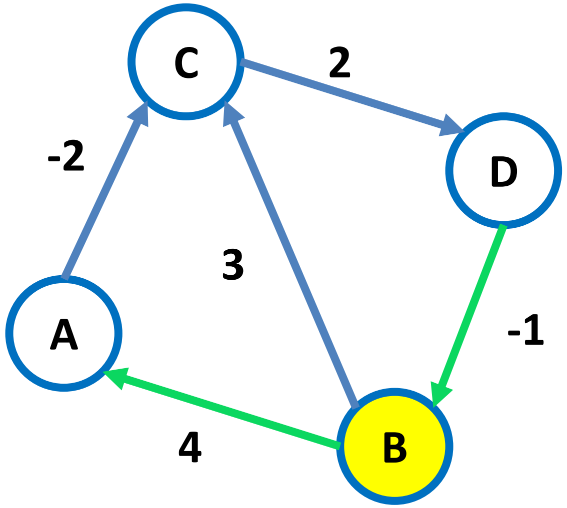
	A	B	C	D
A	0	inf	-2	inf
B	4	0	2	inf
C	inf	inf	0	2
D	inf	-1	2	0



MONASH University

- 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



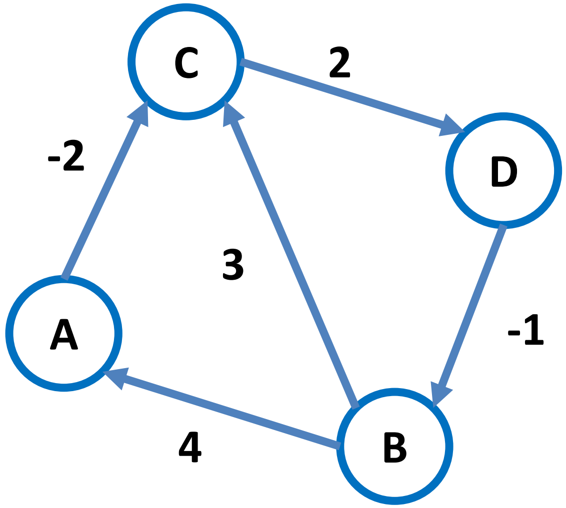
Questions?

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

Floyd-Warshall

All-Pair shortest distance

- 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 distance[i][j] > distance[i][k] + distance[k][j]
    then distance [i][j] = distance[i][k] + distance[k][j]

Floyd-Warshall

All-Pair shortest distance

- What is the algorithm?
  - Same as the Warshall's
  - Except we are now doing distance…
  - If distance[i][j] > distance[i][k] + distance[k][j]
    then distance [i][j] = distance[i][k] + distance[k][j]

```python
# floyd warshall's
for k in range(count_vertex):
    for i in range(count_vertex):
        for j in range(count_vertex):
            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 distance[i][j] > distance[i][k] + distance[k][j]
    then distance [i][j] = distance[i][k] + distance[k][j]

```
# floyd warshall's
for k in range(count_vertex):
    for i in range(count_vertex):
        for j in range(count_vertex):
            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 distance[i][j] > distance[i][k] + distance[k][j]
    then distance [i][j] = distance[i][k] + distance[k][j]

```
# floyd warshall's
for k in range(count_vertex):
    for i in range(count_vertex):
        for j in range(count_vertex):
            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?

```
# floyd warshall's
for k in range(count_vertex):
    for i in range(count_vertex):
        for j in range(count_vertex):
            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

```
# floyd warshall's
for k in range(count_vertex):
    for i in range(count_vertex):
        for j in range(count_vertex):
            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?

- Can you summarize it up?

- Can you summarize it up?

- Un-weighted graph?
- Weighted graph?

- Can you summarize it up?

- Un-weighted graph?
- Weighted graph?
  – Handle negative edge?

- Can you summarize it up?

- Un-weighted graph?
  - BFS
  - Dijkstra
- Weighted graph?
  - Handle negative edge?

- Can you summarize it up?

- Un-weighted graph?
  - BFS
  - Dijkstra
- Weighted graph?
  - Dijkstra
  - Handle negative edge?

MONASH University

- Can you summarize it up?

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

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

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

Questions?

MONASH University

Thank You