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

Faculty of Information Technology, Monash University

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

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)

store the distance from the source
to target vertex by v.distance = u.distance + edge.w

- Dijkstra is a…
  - Dynamic programming algorithm
  - Greedy algorithm
    - Might not work when there is a negative edge

Questions?

cause negative edge would reduce overall distance due to the greedinnes of Dijkstra whihc would prefer the local optimum
assuming no negaive edge (that make v.distance lower than the u.distance) As A-> B shortes, B -> D shortes
then A -> D shortest However in another path, A -> C is not shorest, but duev to the C-> D is a negative edge
overall A -> C -> D is actually shortest not A -> B -> D from Dijkstra

Dijkstra would not consider C -> D if A -> C -> D has distance lower than A -> C , Dijkstra terminate due to reach the D which is destination
However, since C -> D is negative edge, A -> C -> D would actually give shortest path

Bellman Ford

Another shortest distance…

- Bellman-Ford isn’t greedy

- 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

Another shortest distance…

- 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

Another shortest distance…

- 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…

Bellman Ford

Another shortest distance…

- 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?

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?

- 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?

Another shortest distance…

MONASH University

- 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

- 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…

negative circle: overal distance of the edges in cycle is negative

- 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[]
```

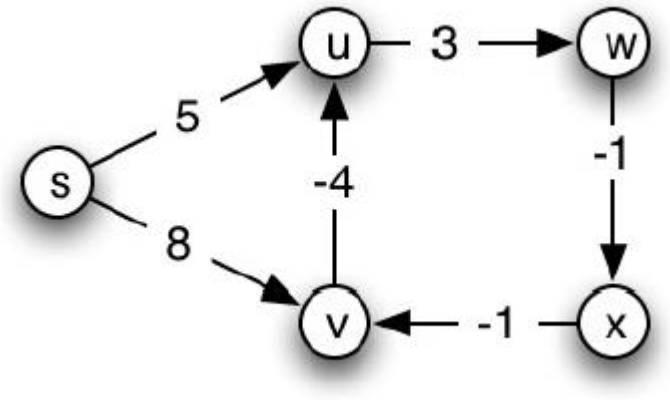
Bellman Ford

Another shortest distance…

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

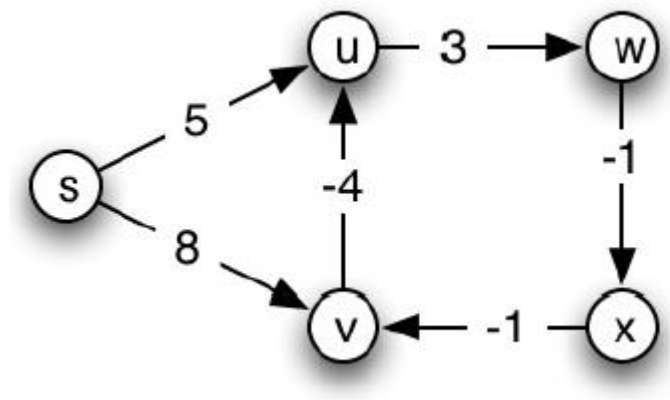
Bellman Ford

Another shortest distance…



Bellman Ford

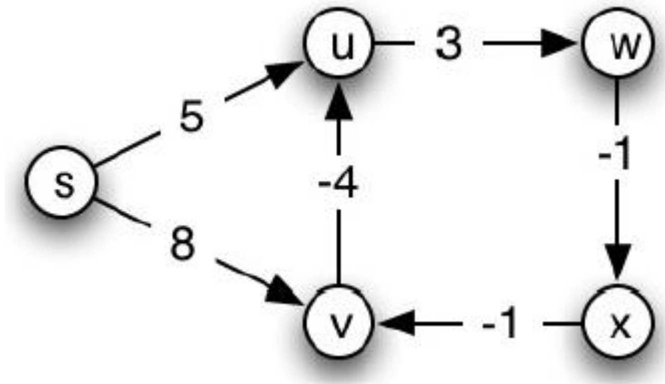
Another shortest distance…



s
u
v
w
x

Bellman Ford

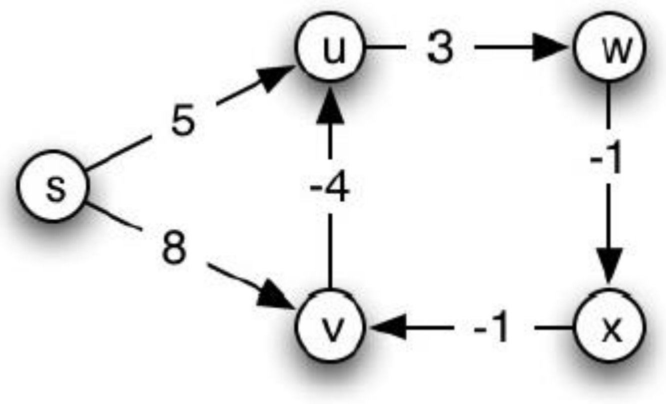
Another shortest distance…



s
u
v
w
x

Bellman Ford

Another shortest distance…

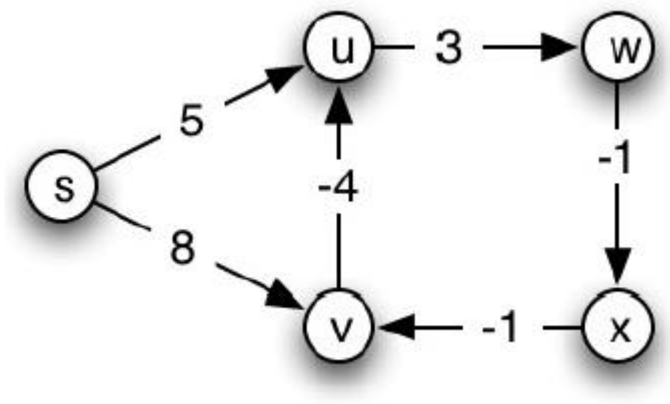


i: numbee of egdes

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

Bellman Ford

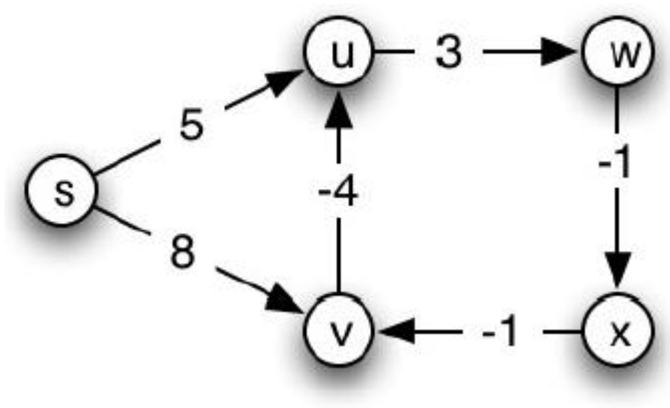
Another shortest distance…



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

Bellman Ford

Another shortest distance…

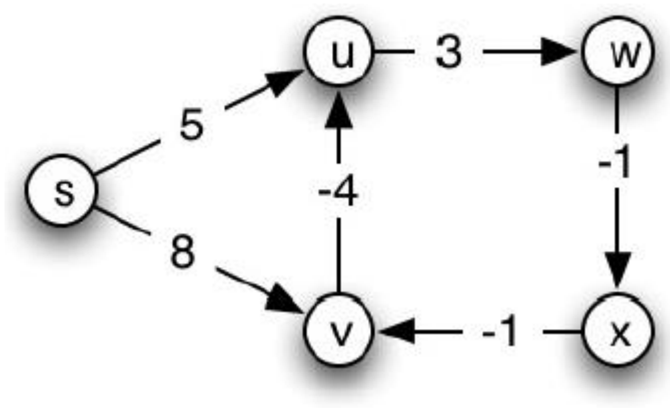


allow 2 edges, u = 4v as: s -> v, v -> u

	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

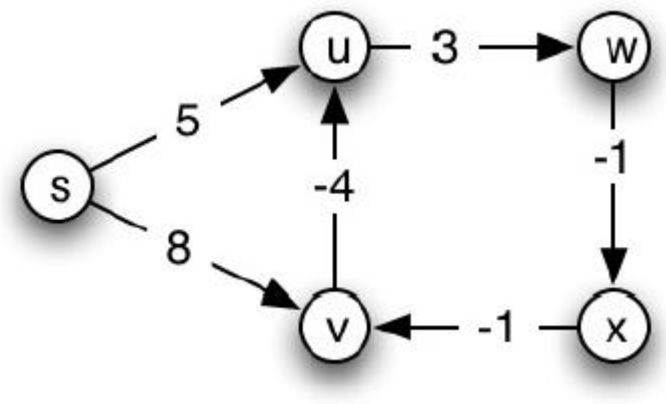
Another shortest distance…



	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

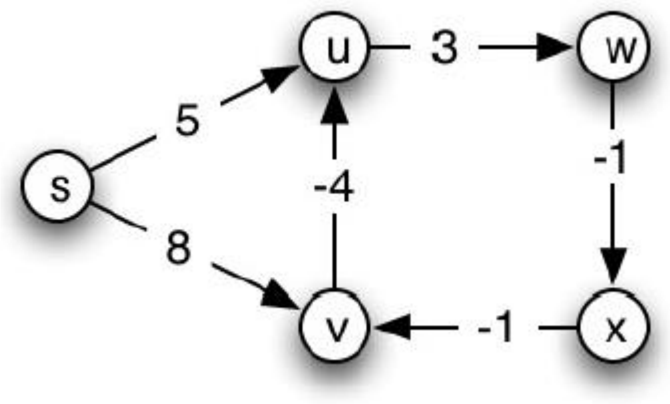
Another shortest distance…



	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…

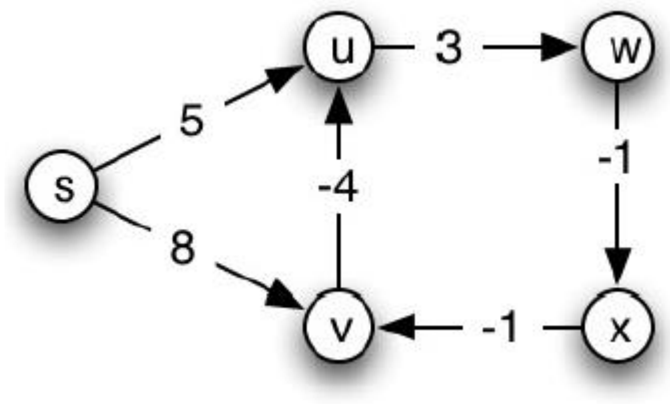


v - 1

	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

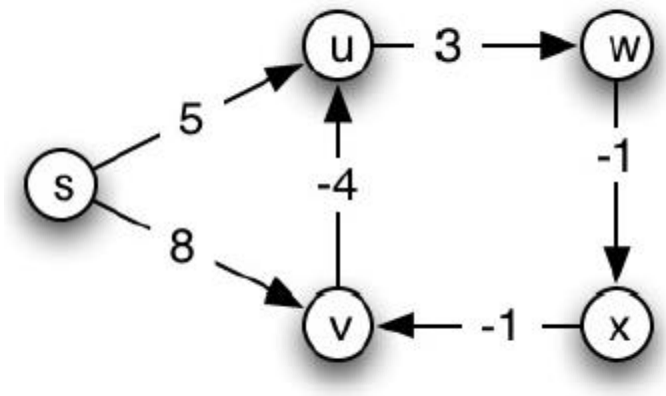
Bellman Ford

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…



only use V - 1 edge at most, tree would only has traversal V - 1, so only V-1 edge in tree, otherwise form cycle   i = V

	i=0	i=1	i=2	i=3	i=4	Checking	
s	0	0	0	0	0	0	
u	inf	5s	4v	4v	4v	decrease negative cycle exist 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

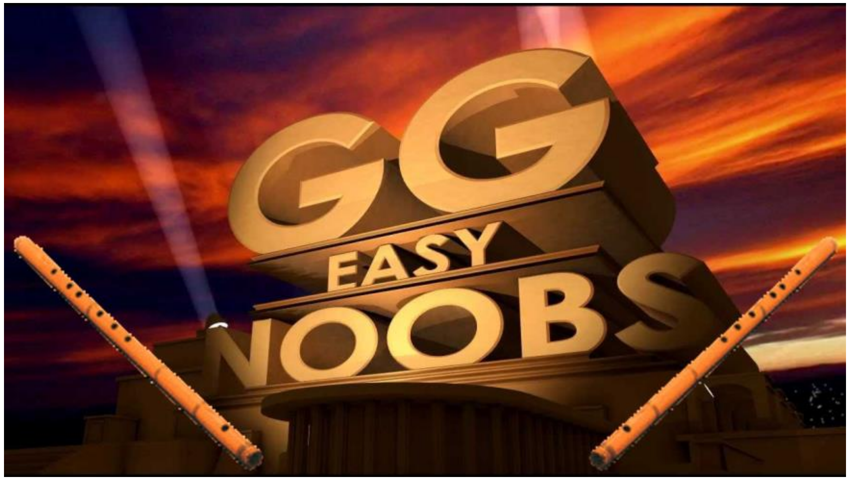
- Total? O(VE)

```
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
         only run v -1 times
   // 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  each edge in vertex update weigh of vertex
                                    with update v.weight with updated u.weight
                                    then updatee previous pmpevious round
   // 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

- 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

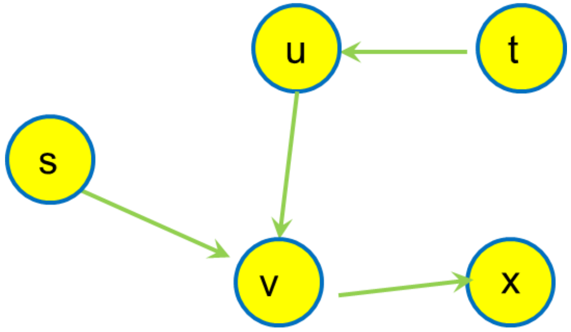
Transitive Closure
Reachability

- 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

Transitive Closure

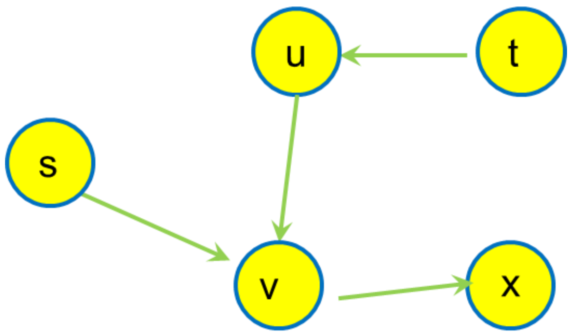
Reachability

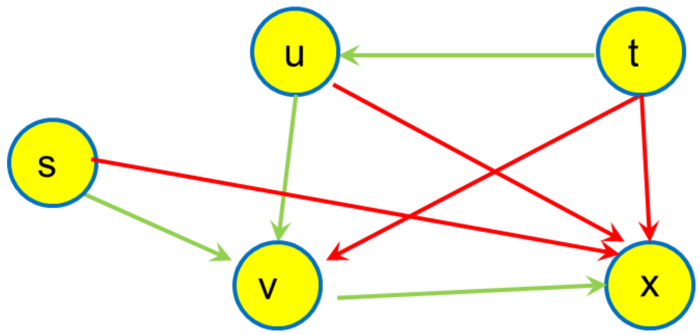
- Given a graph G=(V,E)
- Transitive closure is another graph G'=(V,E')
  - Same vertices V
  - Additional edges between vertex u and vertex v if there's a path between them
  - Concept of transitivity
    - A -> B, B -> C therefore A -> C



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
    virtual link
  - Concept of transitivity
    - A -> B, B -> C therefore A -> C virtual link





Questions?

- 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

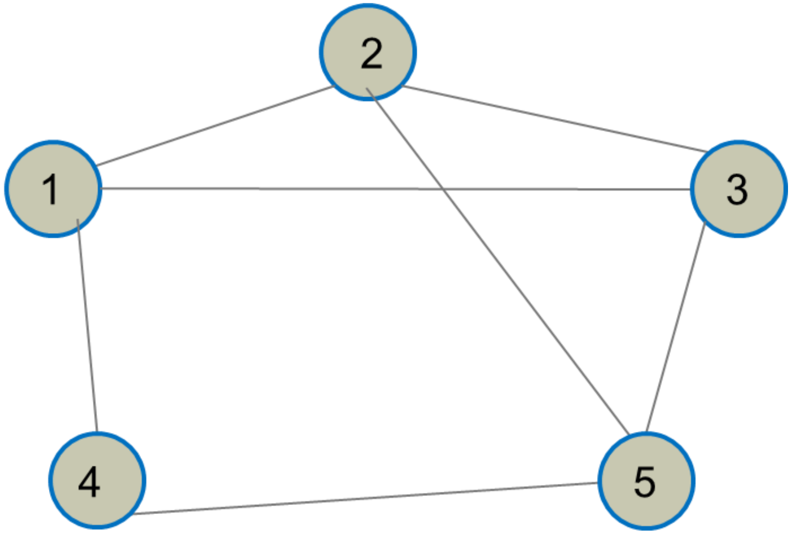
Reachability

- 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

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

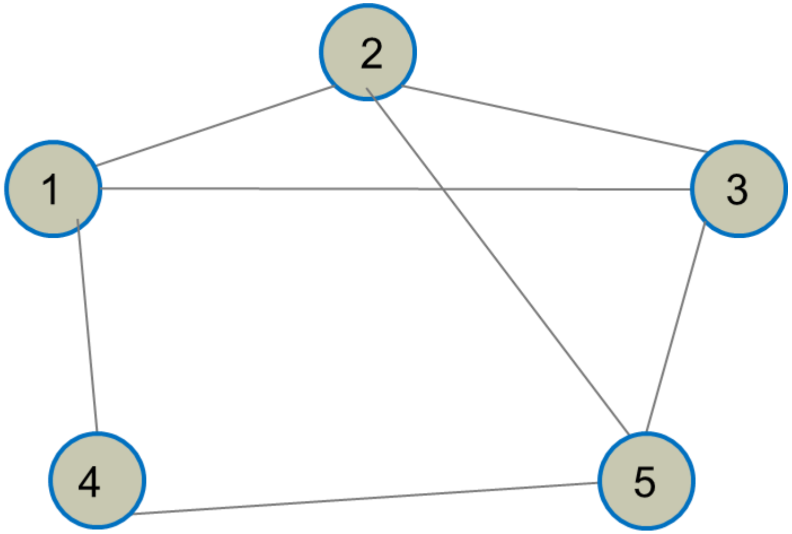
	1	2	3	4	5
1	F	T	T	T	F
2	T	F	T	F	T
3	T	T	F	F	T
4	T	F	F	F	T
5	F	T	T	T	F



Transitive Closure
Reachability

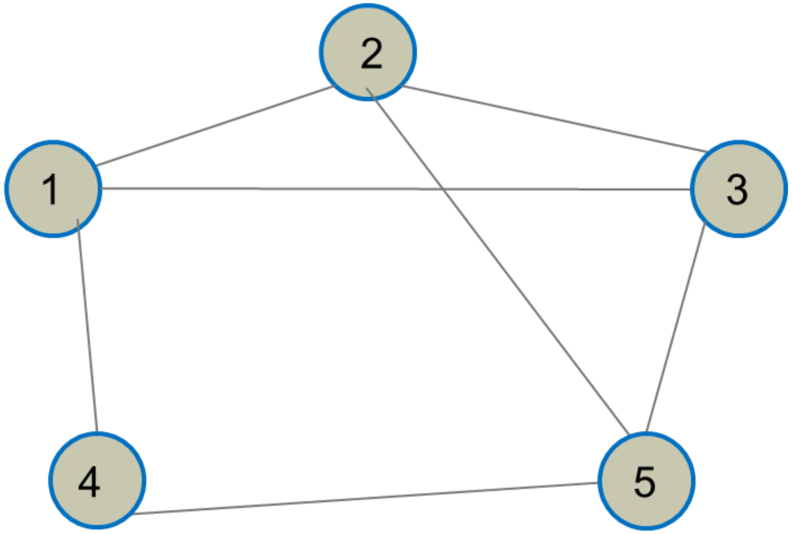
- So how do you do this?
  - Recall our adjacency matrix
  - If matrix[1,2] = True and matrix [2,5] = True

	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

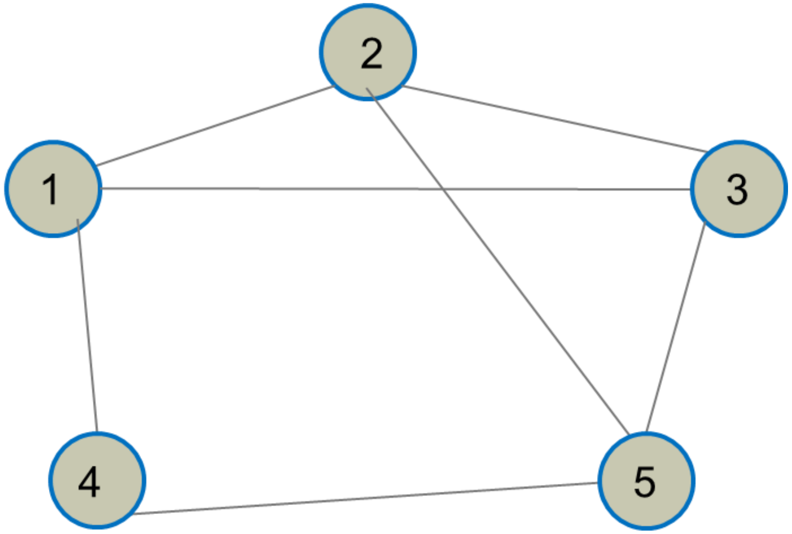
	1	2	3	4	5
1	F	T	T	T	F
2	T	F	T	F	T
3	T	T	F	F	T
4	T	F	F	F	T
5	F	T	T	T	F



- 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

i -> j, j -> k, i -> k

	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
  - 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])
```

either i,k (itself is already True) or Both (i,k = True and k,j = True)

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])
```

either i,k (itself is already True) or Both (i,k = True and k,j = True)

- 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

Questions?

- So we know the Warshall’s algorithm

Floyd-Warshall

Shortest distance

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

Floyd-Warshall
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 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

Floyd-Warshall

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

Floyd-Warshall
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)

    Bellman-Ford and Dijkstra always need source and destination

    Floryd-Warshall only find shortest distance of pairs of verticesa
  - 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 -> 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!

Floyd-Warshall

All-Pair shortest distance

- Why not Dijkstra?

Floyd-Warshall

All-Pair shortest distance

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

Floyd-Warshall
All-Pair shortest distance

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

Floyd-Warshall

All-Pair shortest distance

- 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)

Floyd-Warshall
All-Pair shortest distance

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

    if imutate the FW with Dijkstra, need to run Dijkstra on every vertex

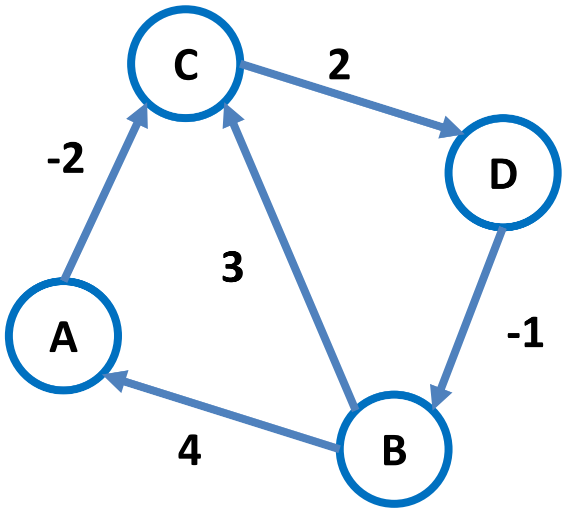
- Why not Bellman-Ford? single-source algo
  - Need to Bellman-Ford from every vertex.
    O(V) * O(VE) = O(V^2 E) = O(V^4)

- Floyd-Warshall can do it quicker!

  O(V^3)

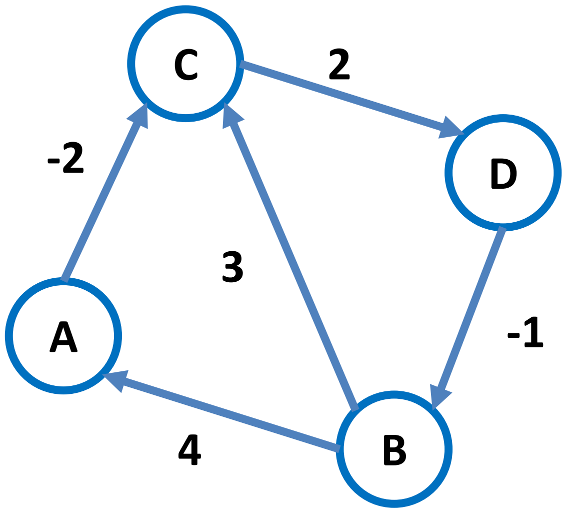
Questions?

- Now let us do it manually…



- Now let us do it manually…

	A	B	C	D
A				
B				
C				
D				

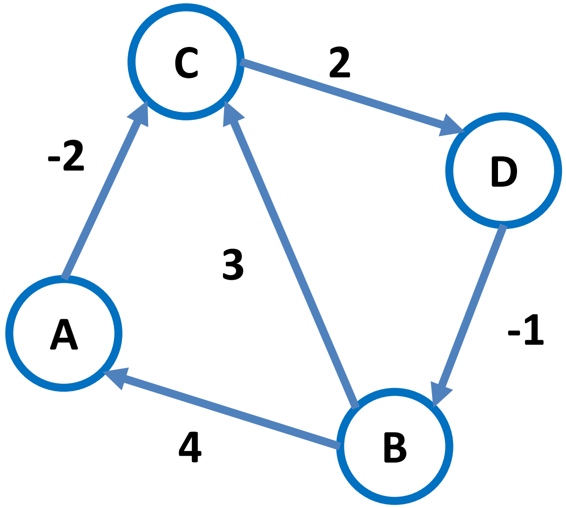


Floyd-Warshall

All-Pair shortest distance

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

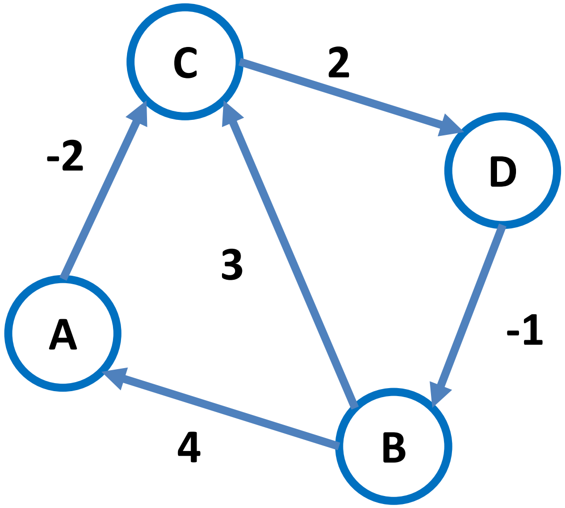
	A	B	C	D
A	0			
B		0		
C			0	
D				0



MONASH University

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

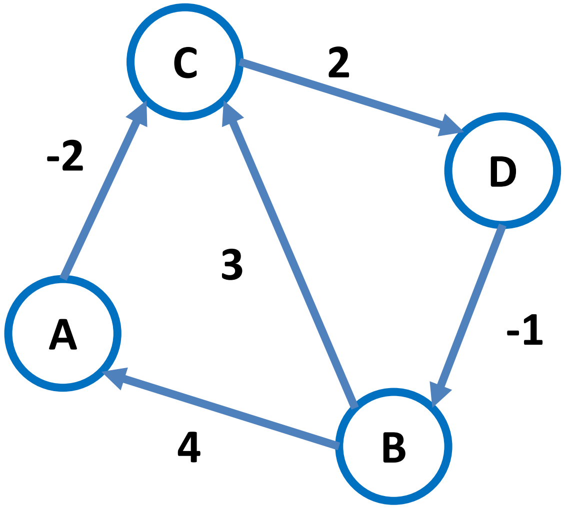
	A	B	C	D
A	0		-2	
B	4	0	3	
C			0	2
D		-1		0



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



k: – 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?

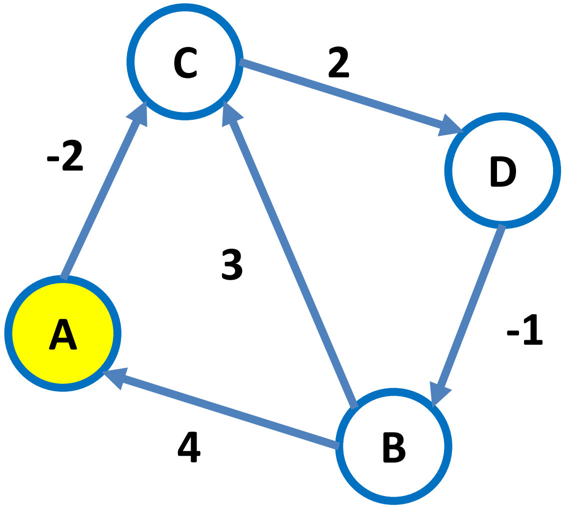
i:

Questions?

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

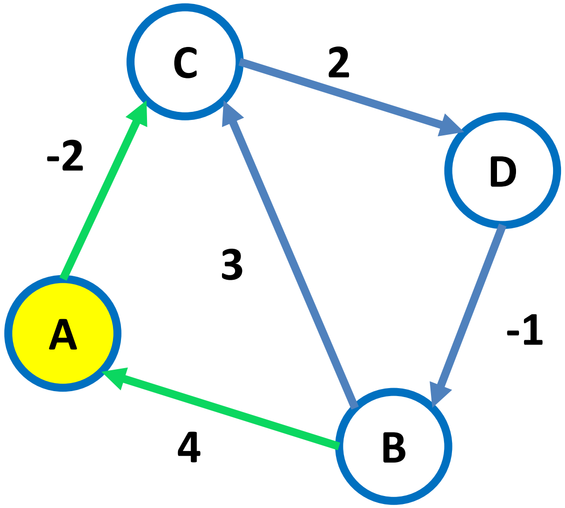
do A column first with all rows, then B, C

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



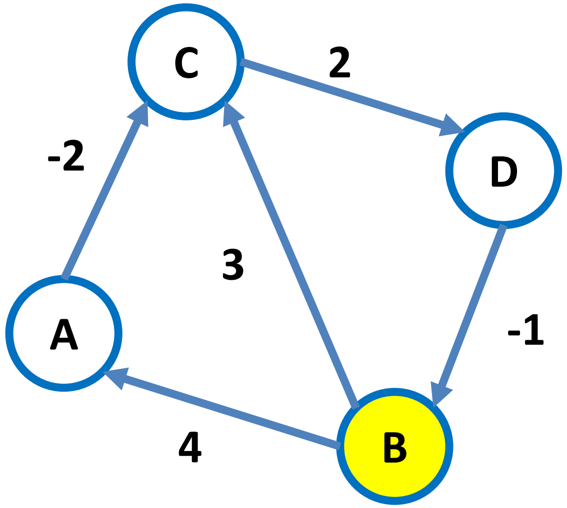
- What is the algorithm?
  – Begin with vertex A, can we update the shortest distance of all vertices going through A?

	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



Floyd-Warshall

All-Pair shortest distance

i -> k -> j   B -> A -> C

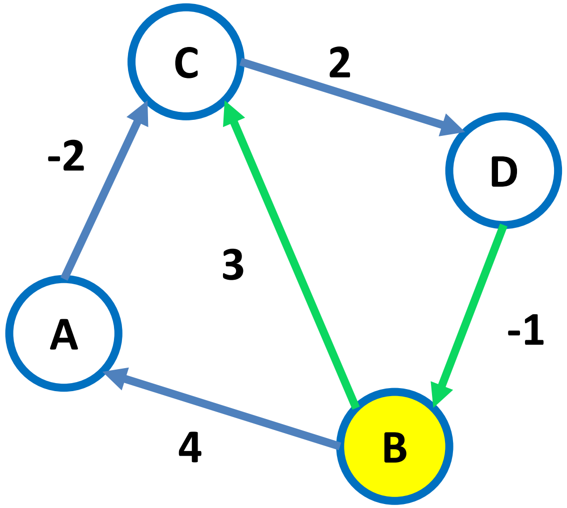
k: – Begin with vertex A, can we update the shortest distance of all vertices going through A

i: B in this case, loop through the rows for column

j: C

- 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?

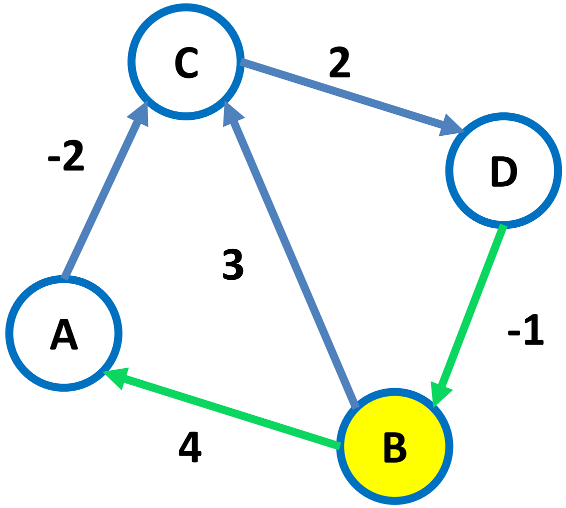
	A	B	C	D
A	0	inf	-2	inf
B	4	0	2	inf
C	inf	inf	0	2
D	inf	-1	2	0



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



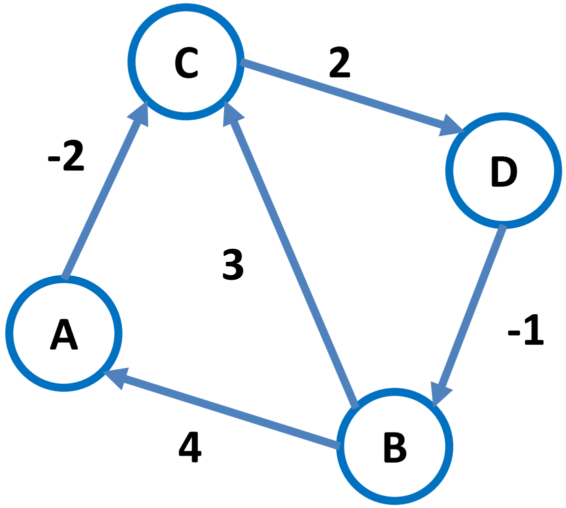
Questions?

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

- 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])
```

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

MONASH University

B -> A -> C,
Even there is no edge from B to C
(Inf the matrix) Flord-Warshall would
give a B -> C vertual edge by assgin a
value = edge.w(B,A) + edge.w(A, C)
which value must < Inf So matrix[A][A] can be changed
by matrix[B][A] + matrix[A][C] and more as there is negative cycle

i -> k -> j   B -> A -> C
k: – Begin with vertex A, can we update the shortest distance of all vertices going through A

i: B in this case, loop through the rows for column A, take the value - matrix[i][k] = matrix[B][A], value is maintained for possible edge from B
(next loop to search for v of the outgoing edge from A)
j: C in this case, all possible edge from A
Finally update matrix[i][j] = matrix[B][C]

- 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]

choose minimum one between (existing one and the updated one)

```
# 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?

going for vertex column

    - 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?

- What is the complexity?

```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 complexity?
  - Same as Warshall's

```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])
```

choose minimum one between (existing one and the updated one)

Questions?

- 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

MONASH University

- 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 =(

if dagonal value is negative -> negative cycle
all algori not useable, as forever going round the negative cycle

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?

Shortest Distance

Shortest Paths

- Can you summarize it up?

k: – 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?

i: B

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

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

i -> k -> j   B -> A -> C

k: – Begin with vertex A, can we update the shortest distance of all vertices going through A

i: B in this case, loop through the rows for column
j: C

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

k: – Begin with vertex A, can we update the shortest distance of all vertices going through A?

i: B

- 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