Faculty of Information Technology, Monash University

FIT2004: Algorithms and Data Structures

Week 9: Bellman-Ford and Floyd-Warshall Algorithms

These slides are prepared by M. A. Cheema and are based on the material developed by Arun Konagurthu and Lloyd Allison.

Recommended reading

- Unit notes: Chapter 13
- Cormen et al. Introduction to Algorithms.
  - Section 24.1: Bellman-Ford algorithm
  - Section 25.2: Floyd-Warshall algorithm

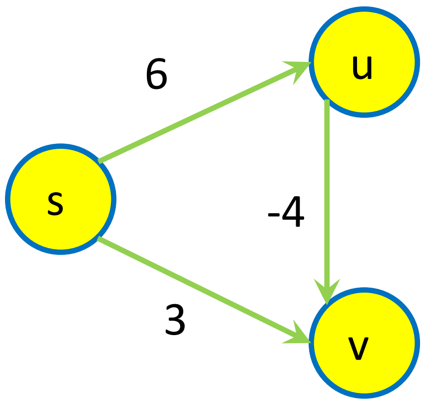
- Student Evaluation of Teaching and Units is now open

Outline

1. Shortest path in graphs with negative weights
2. All-pairs shortest paths
3. Transitive Closure

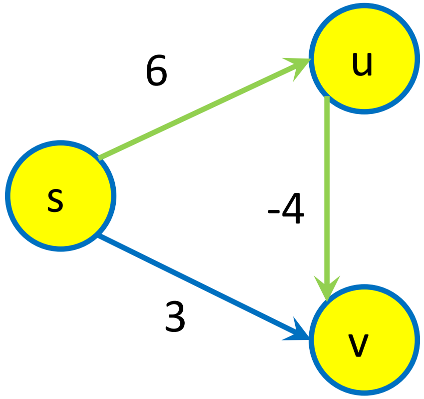
Shortest path (negative weights)

- What is the shortest distance from s to v in this graph?
- If Dijkstra's algorithm is used on this graph, what will it output as being the shortest path from s to v?
- Dijkstra's algorithm is not guaranteed to output the correct answer when there are negative weights.



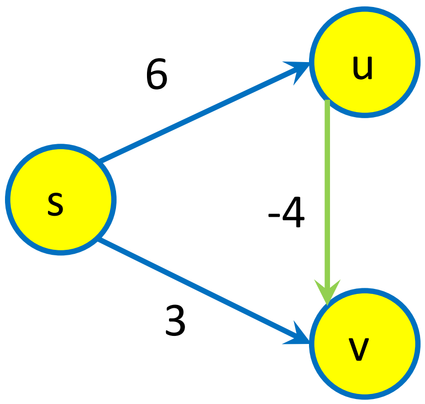
Shortest path (negative weights)

- What is the shortest distance from s to v in this graph?
- If Dijkstra's algorithm is used on this graph, what will it output as being the shortest path from s to v?
- Dijkstra's algorithm is not guaranteed to output the correct answer when there are negative weights.



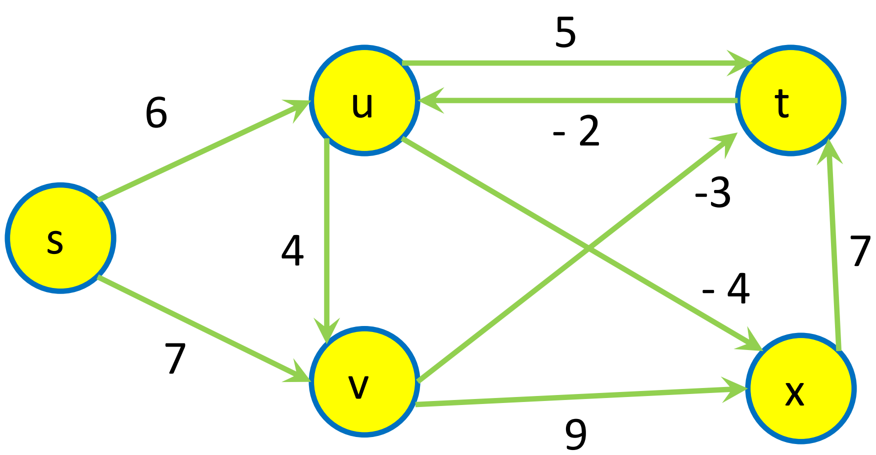
Shortest path (negative weights)

- What is the shortest distance from s to v in this graph?
- If Dijkstra's algorithm is used on this graph, what will it output as being the shortest path from s to v?
- Dijkstra's algorithm is not guaranteed to output the correct answer when there are negative weights.



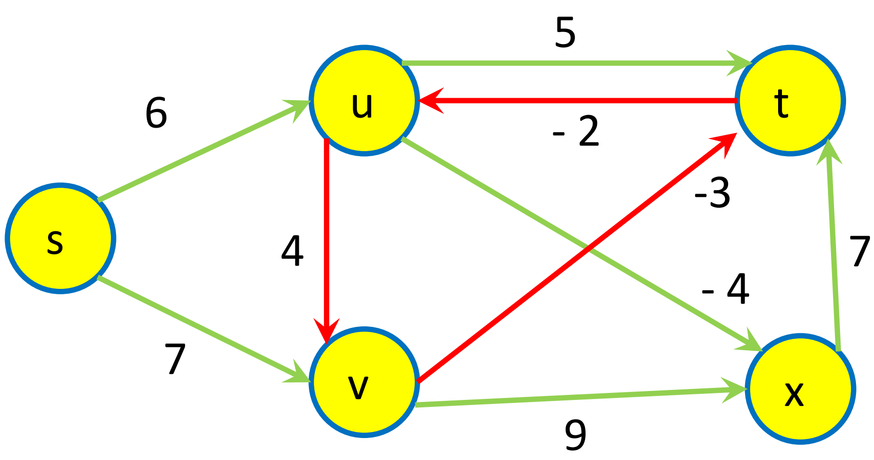
Shortest path (negative weights)

- What is the shortest distance from s to x in this graph?



Shortest path (negative weights)

- What is the shortest distance from s to x in this graph?
- Not well-defined:
  - From s, it is possible to reach the negative cycle u-->v-->t, and from this cycle it is possible to reach x.
  - Given any path P, it is possible to obtain an alternative path P’ with smaller total weight than P: P’ goes from s to the negative cycle, include as many repetitions of the negative cycle as necessary, and then reaches x from the negative cycle.



Bellman-Ford Algorithm

- Bellman-Ford algorithm returns:
  - shortest distances from s to all vertices in the graph if there are no negative cycles that are reachable from s.
  - an error if there is a negative cycle reachable from s (i.e., can be used to detect negative cycles).

- Can be modified to return all valid shortest distances, and minus ∞ for vertices which are affected by the negative cycle.

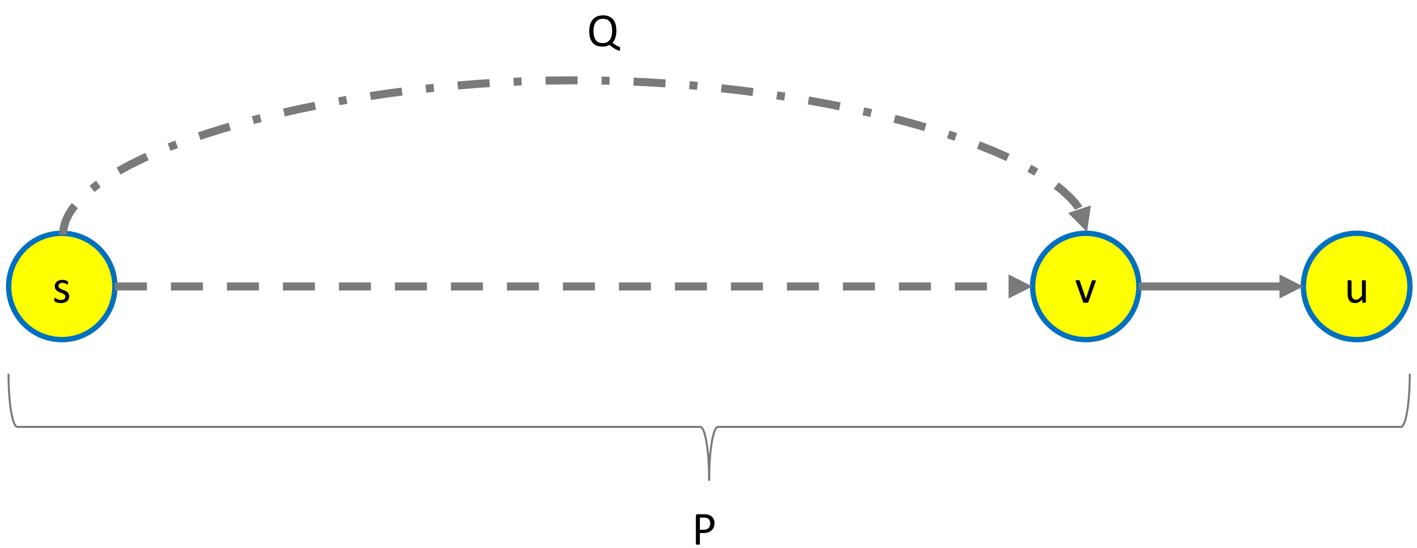
Bellman-Ford Algorithm

- Idea: If no negative cycles are reachable from node s, then for every node t that is reachable from s there is a shortest path from s to t that is simple (i.e., no nodes are repeated).
  - Cycles with positive weight cannot be part of a shortest path.
  - Given a shortest path that contains cycles of weight 0, the cycles can be removed to obtain an alternative shortest path that is simple.

- Note that any simple path has at most V-1 edges.

Bellman-Ford Algorithm

- A fact from last week: If P is a shortest path from s to u, and v is the last vertex on P before u, then the part of P from s to v is also a shortest path.
- Suppose there was a shorter path from s to v, say Q.
- weight(Q) + w(v,u) < weight(P)
- But P is the shortest path from s to u.
- Contradiction



Bellman-Ford Algorithm

- Bellman-Ford was one of the first applications of dynamic programming.

- For a source node s, let OPT(i,v) denote the minimum weight of a s-->v path with at most i edges.

- Let P be an optimal path with at most i edges that achieves total weight OPT(i,v):
  - If P has at most i-1 edges, then OPT(i,v)=OPT(i-1,v).
  - If P has exactly i edges and (u,v) is the last edge of P, then OPT(i,v)=OPT(i-1,u)+w(u,v), where w(u,v) denotes the weight of edge (u,v).

- Recursive formula for dynamic programming:

$$OPT(i,v) = \min\left(OPT(i-1,v), \min_{u:(u,v)\in E}\left(OPT(i-1,u) + w(u,v)\right)\right)$$

Bellman-Ford Algorithm

```
Uses array M[0...V-1,1...V]
Initialize M[0,s] = 0, for all other vertices M[0,v] = infinity
for i = 1 to V-1:
        for each vertex v:
                Compute M[i,v] using the recurrence
return M[V-1,1...V]
```

Time Complexity:

O(VE)

Bellman-Ford Algorithm

- Commonly, a more space-efficient version of Bellman-Ford algorithm is implemented.
- V-1 iterations are performed, but the value i is used just as a counter, and in each iteration, for each node v, we use the update rule

$$M[v] = \min(M[v], \min_{u:(u,v)\in E}(M[u] + w(u,v)))$$

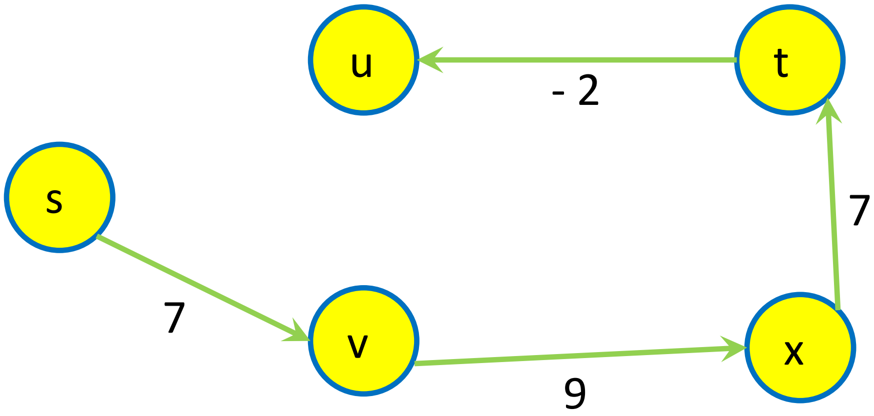
- In some cases, this version also provides a speed-up (but no improvement in the worst-case time complexity).

Bellman-Ford Algorithm

- V-1 iterations are performed, but the value i is used just as a counter, and in each iteration, for each node v, we use following update rule for the distance:

$$dist[v] = \min(dist[v], \min_{u:(u,v)\in E}(dist[u] + w(u,v)))$$

- If vertices are updated in the order s, v, x, t, u, then we are done after 1 iteration.
- On the other hand, if vertices are updated in the order u, t, x, v, s, then we need 4 iterations to get the right result.
- We will analyse the early stopping condition later on.



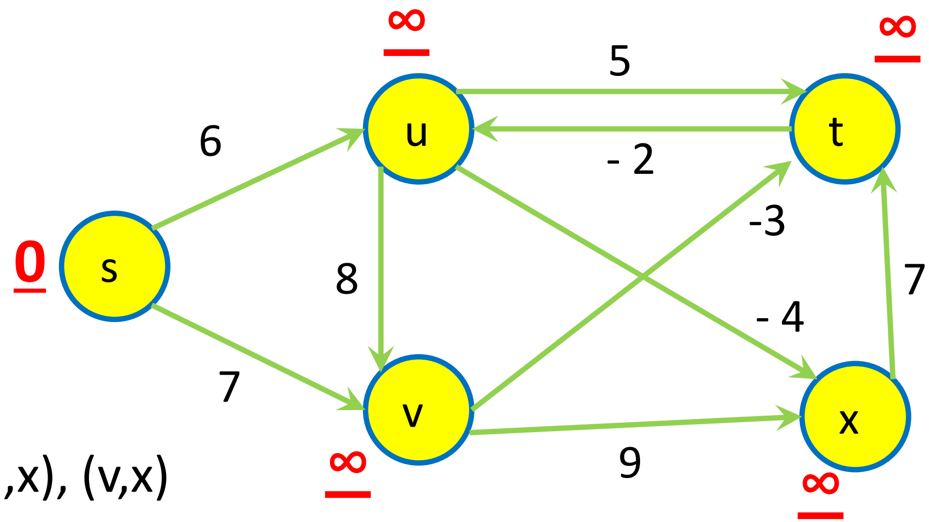
Bellman-Ford Algorithm

Initialize:

- For each vertex a in the graph
  - dist(s,a) = ∞
- dist(s,s) = 0

Consider the following operation (relaxation):

- For each edge (a, b) in the graph
  - dist(s, b) = min(dist(s,b) , dist(s,a) + w(a,b))



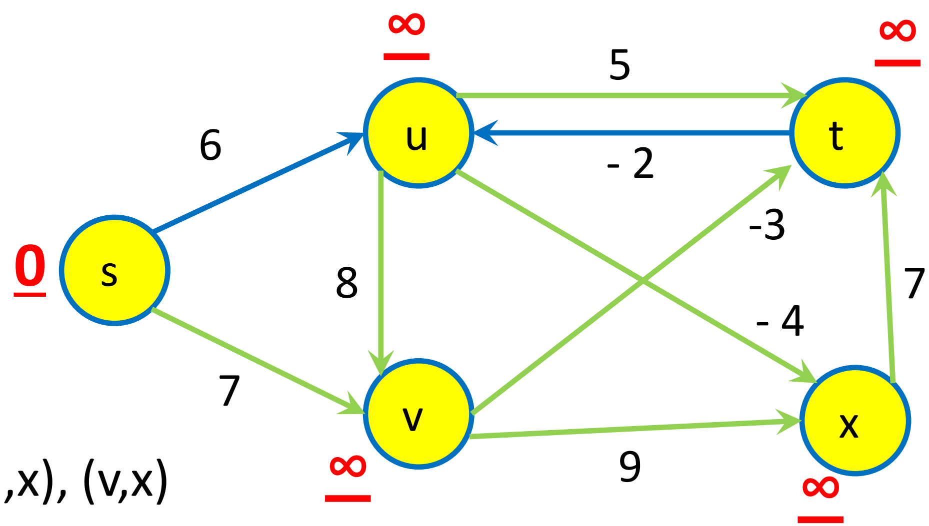
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

First iteration:

Relaxing incoming edges of node u



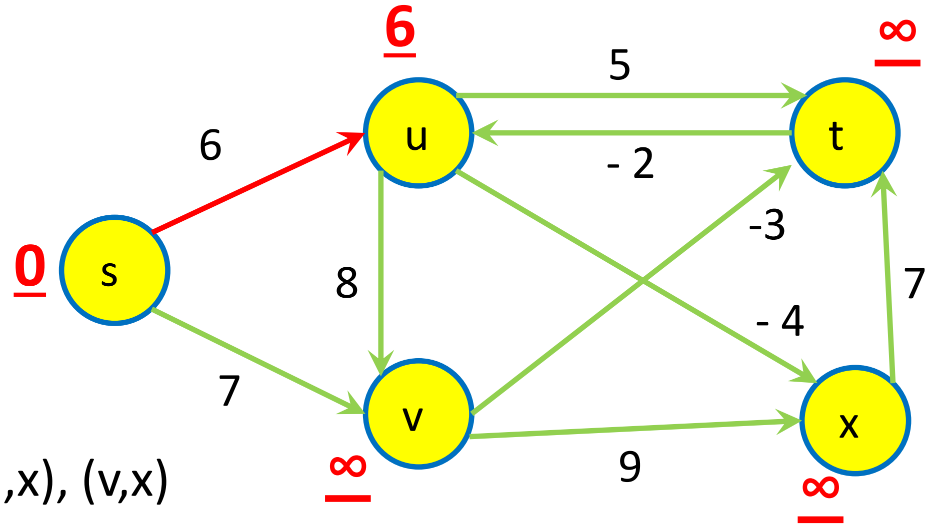
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

First iteration:

Done relaxing incoming edges of node u



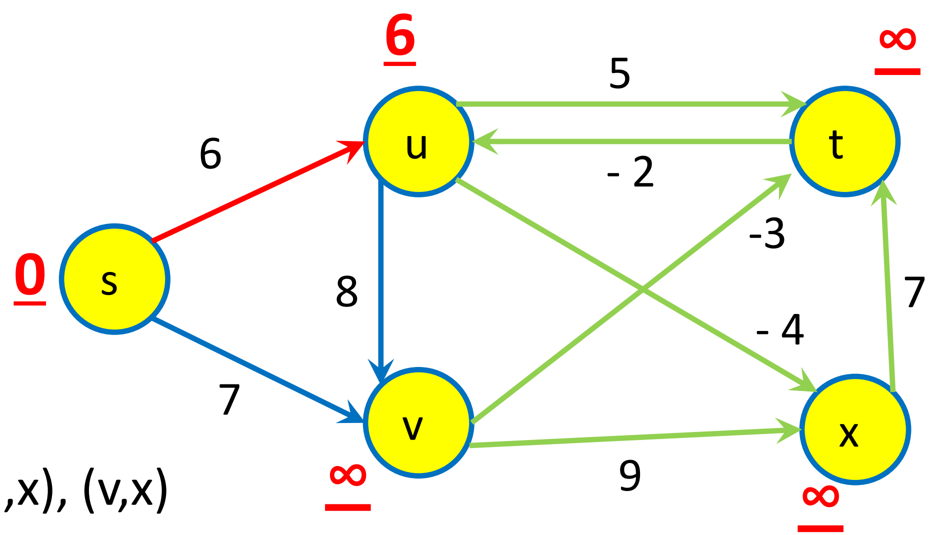
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

First iteration:

Relaxing incoming edges of node v



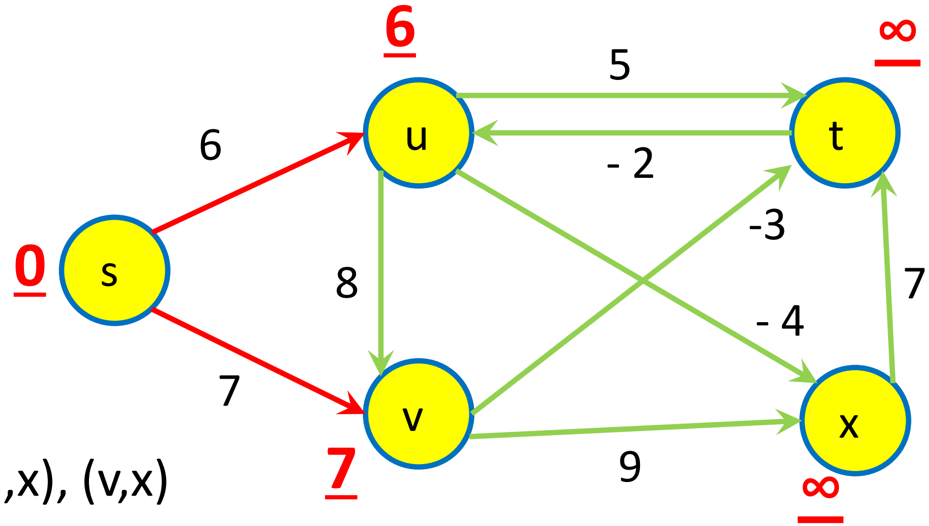
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

First iteration:

Done relaxing incoming edges of node v



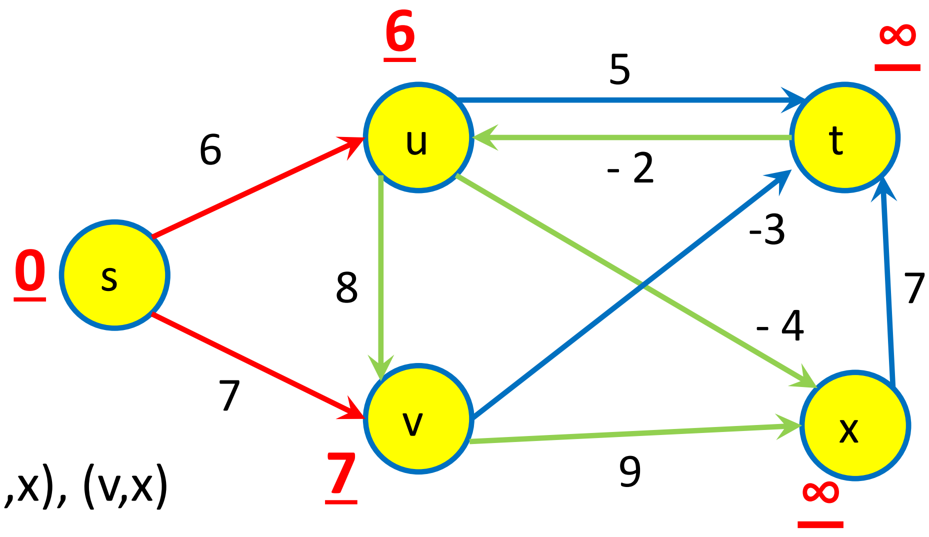
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

First iteration:

Relaxing incoming edges of node t



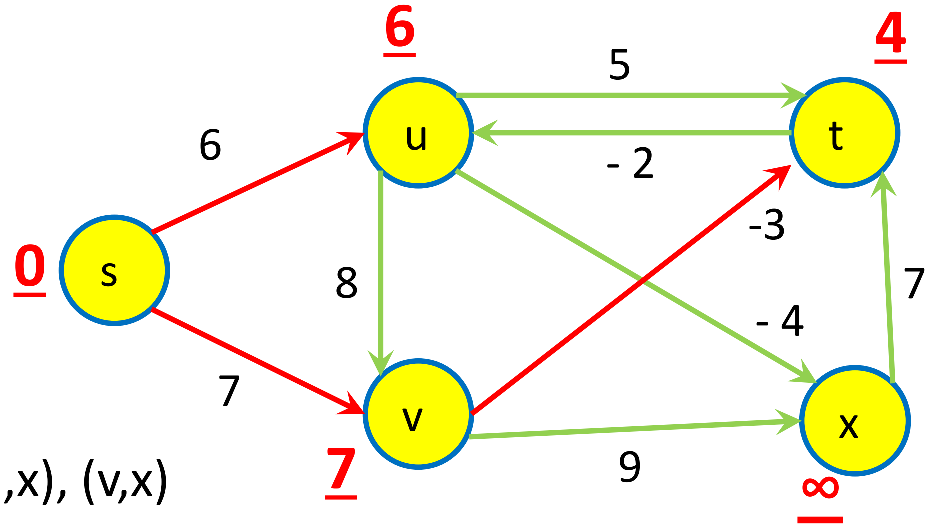
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

First iteration:

Done relaxing incoming edges of node t



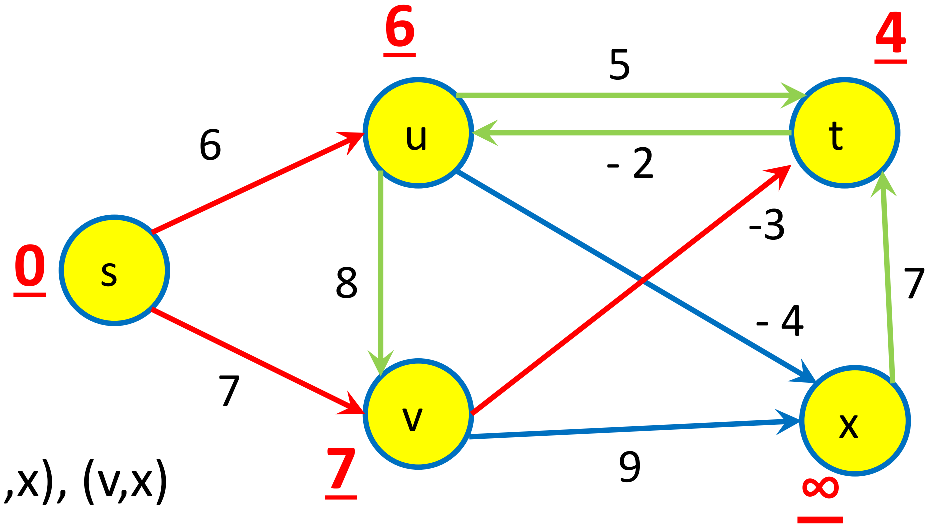
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

First iteration:

Relaxing incoming edges of node x



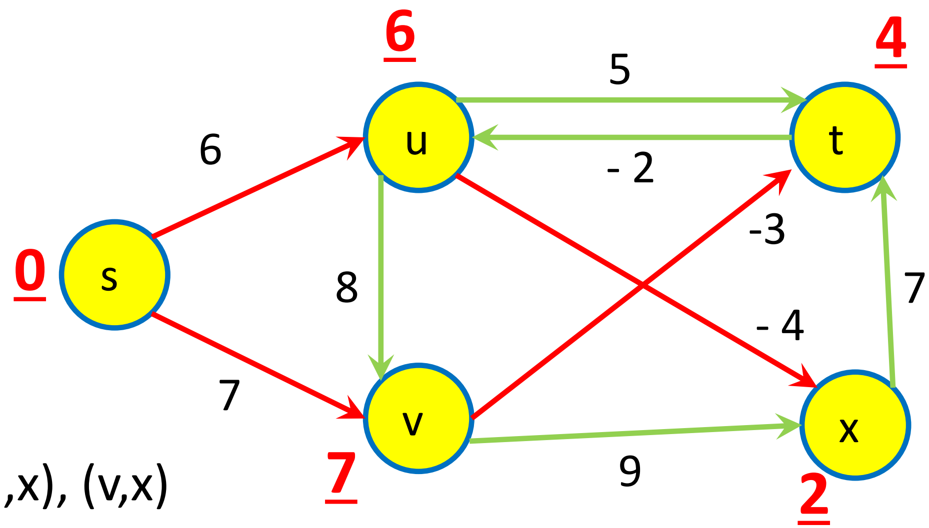
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

First iteration:

Done relaxing incoming edges of node x

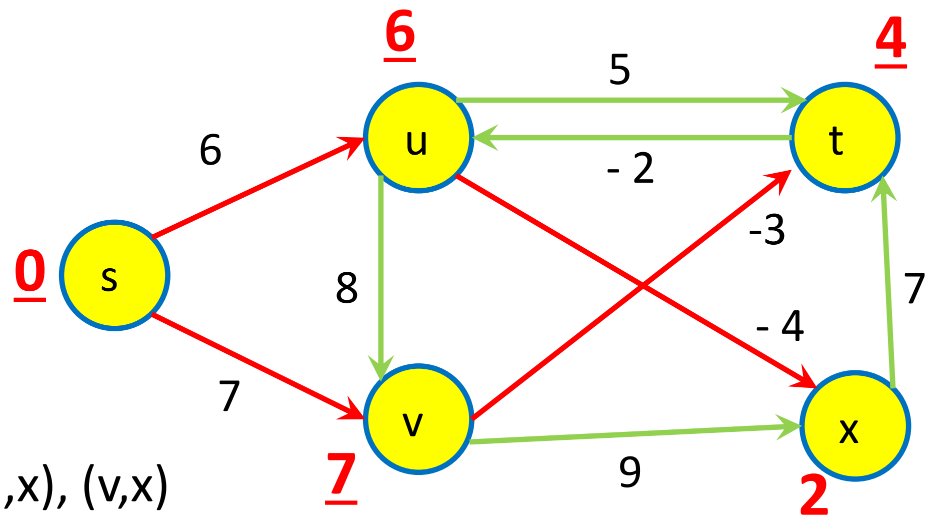


Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

First iteration finished:



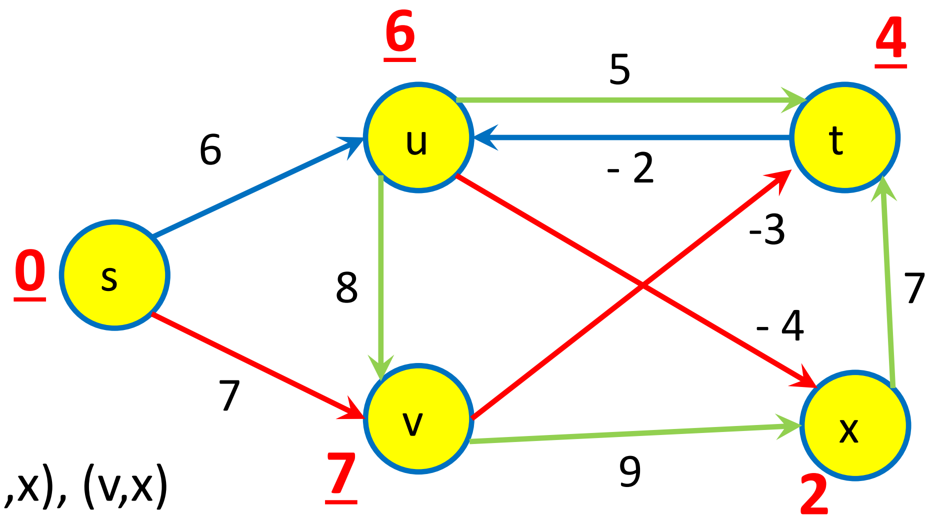
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

Second iteration:

Relaxing incoming edges of node u



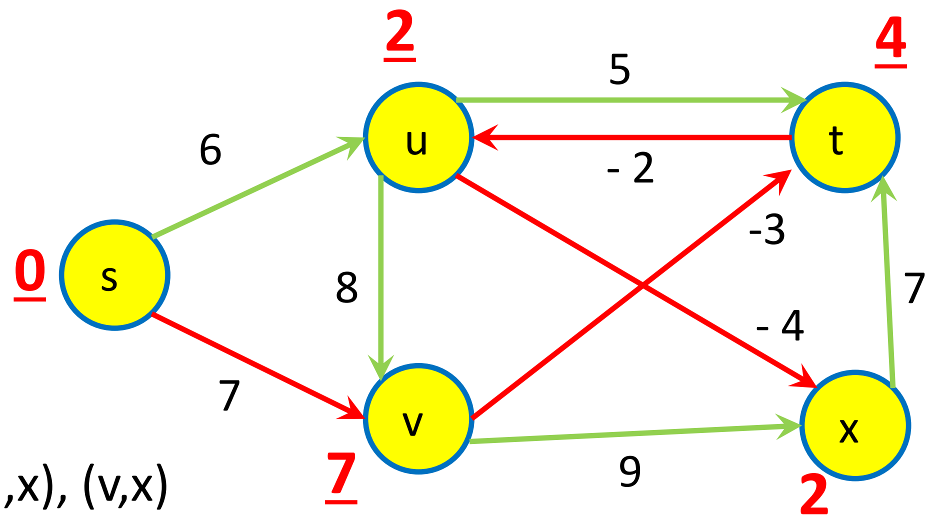
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

Second iteration:

Done relaxing incoming edges of node u



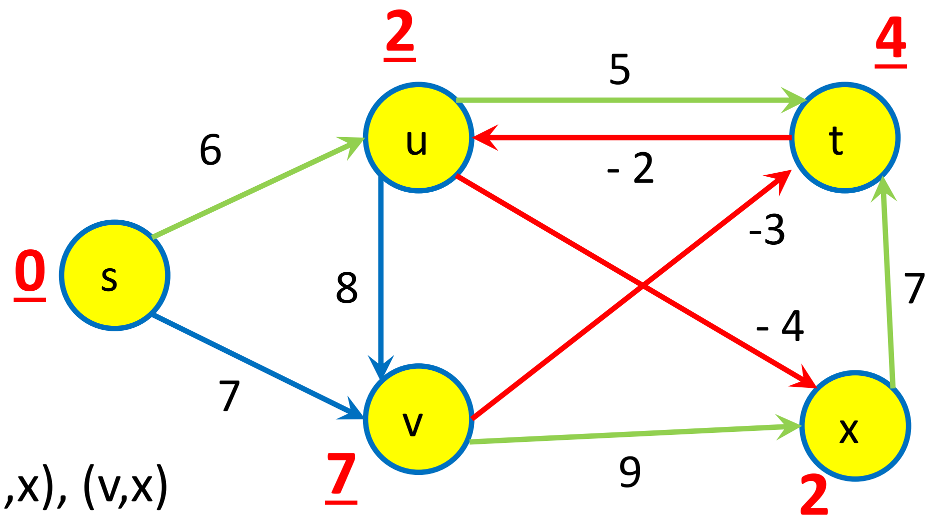
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

Second iteration:

Relaxing incoming edges of node v



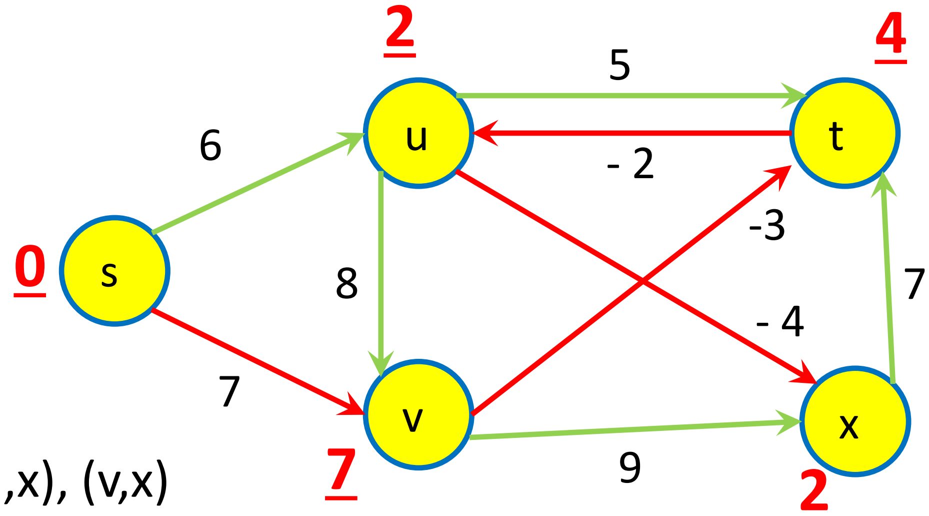
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

Second iteration:

Done relaxing incoming edges of node v



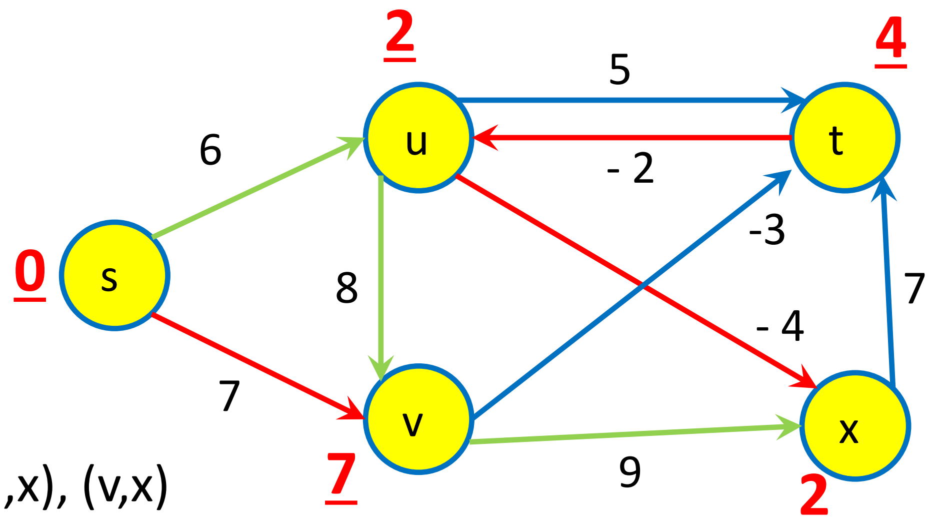
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

Second iteration:

Relaxing incoming edges of node t



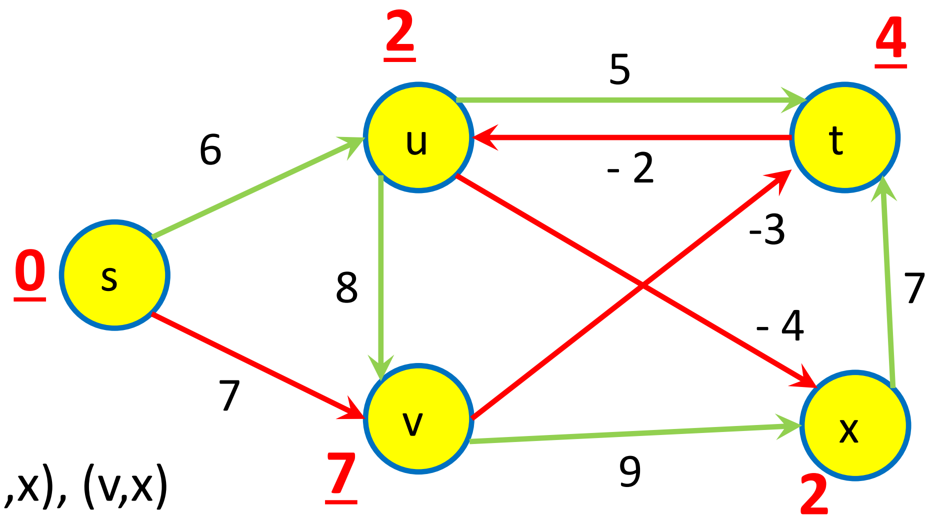
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

Second iteration:

Done Relaxing incoming edges of node t



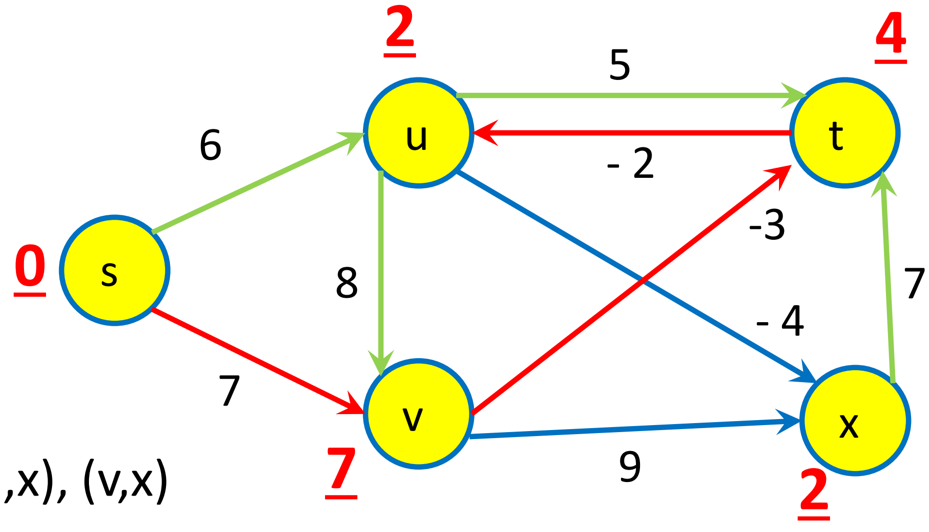
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

Second iteration:

Relaxing incoming edges of node x



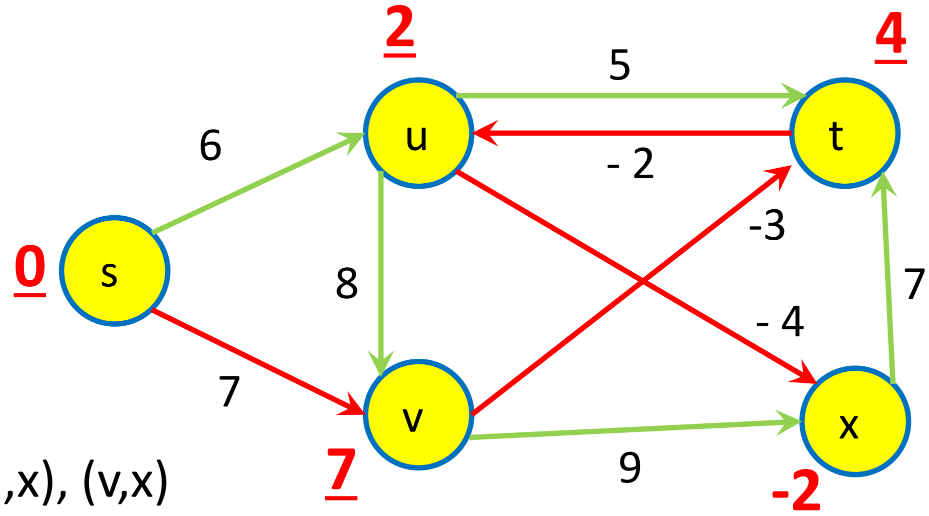
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

Second iteration:

Done Relaxing incoming edges of node x

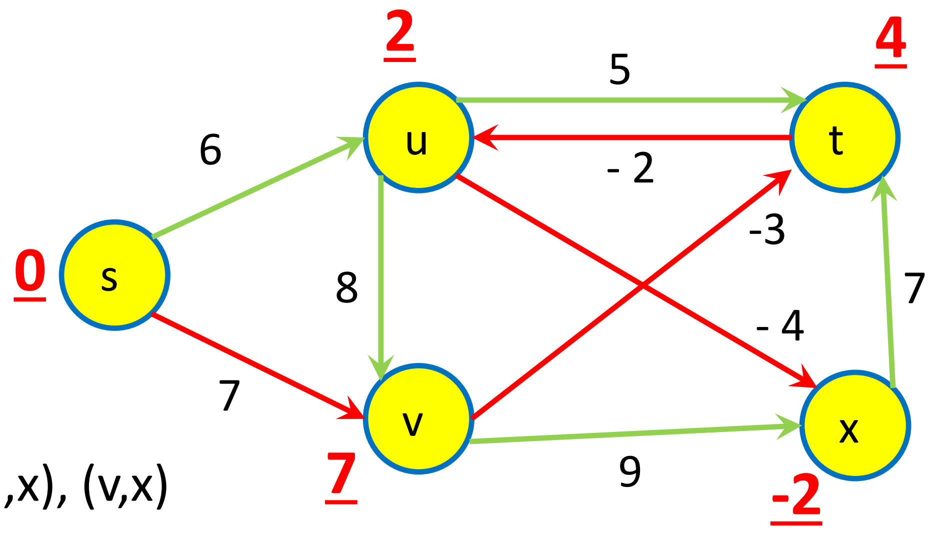


Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

Second iteration finished:



Assume the following order:

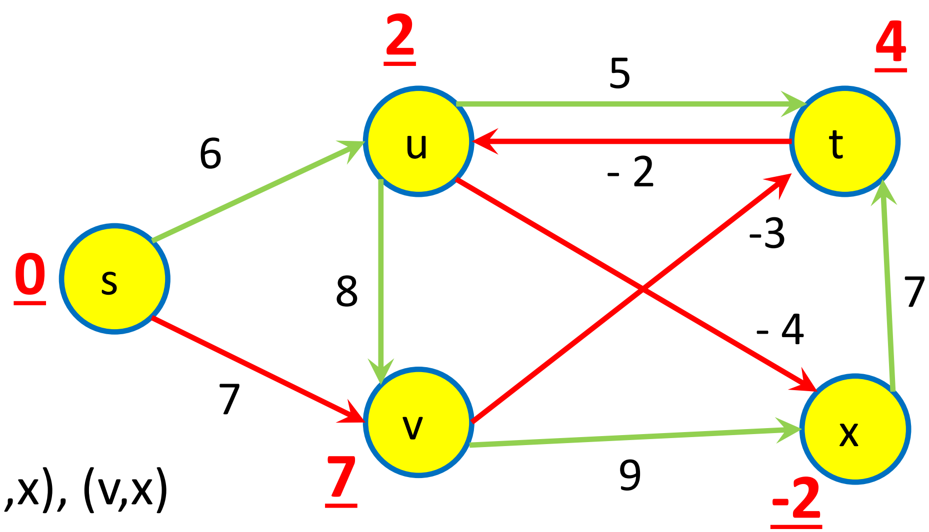
(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm

Third iteration:

Speeding things up: All edges relaxation in the third iteration do not change anything.

Early Stop Condition: If nothing changes in one iteration, it is possible to stop the execution of the Bellman-Ford algorithm and output the current values.

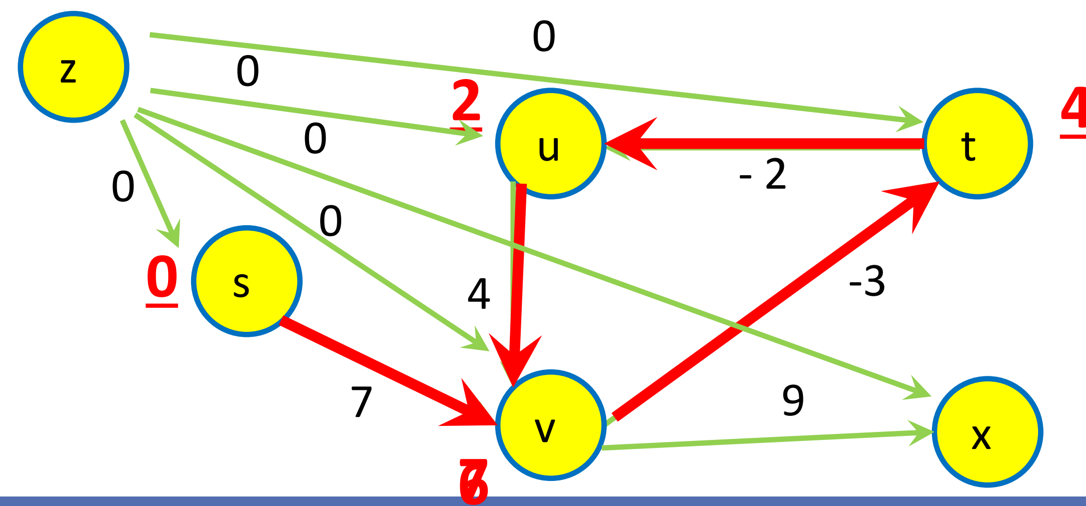


Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm: Negative Cycles

- If V-th iteration reduces the distance of a vertex, this means that there is a shorter path with at least V edges which implies that there is a negative cycle.
- Consider the graph with vertices s, u, v, and t and assume we have run (V-1 = 3) iterations.
- In the 4th iteration, the weight of at least one vertex will be reduced (due to the presence of a negative cycle).
- Important: Bellman-Ford Algorithm finds negative cycles only if such cycle is reachable from the source vertex
  - E.g., if x is the source vertex, the algorithm will not detect the negative cycle
- Detecting if a graph G has a negative cycle: just add one extra node to G and edges from it to every other node, and run Bellman-Ford on the added node.



Bellman-Ford Algorithm

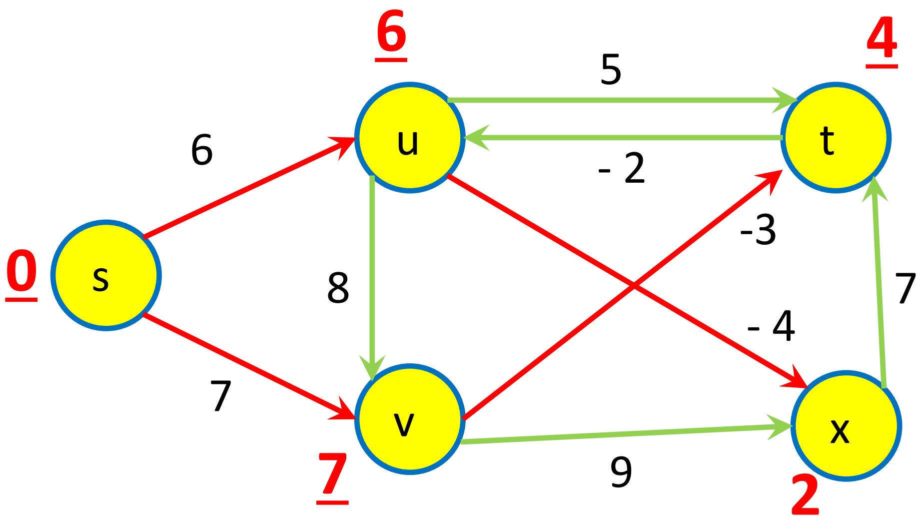
```
# STEP 1: Initializations
dist[1...V] = infinity
pred[1...V] = Null
dist[s] = 0
# STEP 2: Iteratively estimate dist[v] (from source s)
for i = 1 to V-1:
        for each edge <u,v> in the whole graph:
                est = dist[u] + w(u,v)
                if est < dist[v]:
                        dist[v] = est
                        pred[v] = u

# STEP 3: Checks and returns false if a negative weight cycle
# is along the path from s to any other vertex
for each edge <u,v> in the whole graph:
        if dist[u]+w(u,v) < dist[v] :
                return error; # negative edge cylce found in this graph

return dist[...], pred[...]
```

Time Complexity:

O(VE)

Bellman-Ford Algorithm

- For this space-efficient version of Bellman-Ford algorithm, there is a guarantee that after i iterations dist[v] is no larger than the total weight of the shortest path from s to v that uses at most i edges.
- But there is no guarantee that these two values are equal after i iterations: depending on the order in which the edges are relaxed, the path P from s to v that has weight dist[v] could already contain more than i edges after the i-th iteration.
  - e.g., in the graph that we followed a detailed execution of Bellman-Ford, the path from s to t already has two edges after just one iteration.



Bellman-Ford Algorithm: Negative Cycles

- How could we modify Bellman-Ford to determine **which** vertices have valid distances, and which are affected by the negative cycle?

- Execute the V^{th} iteration, and for each node whose distance would be updated, just mark its distance as $-\infty$.

Outline

1. Shortest path in a graph with negative weights
2. All-pairs shortest paths
3. Transitive Closure

All-Pairs Shortest Paths

Problem

- Return shortest distances between **all** pairs of vertices in a connected graph.

For unweighted graphs:

- For each vertex v in the graph
  - Call Breadth-First Search for v

Time complexity:

$O(V(V+E)) = O(V^2 + EV) \rightarrow O(EV)$ [for connected graphs $O(V) \leq O(E)$]

For dense graphs: E is $O(V^2)$, therefore total cost is $O(V^3)$ for dense graphs

All-Pairs Shortest Paths

For weighted graphs (with non-negative weights):

- For each vertex v in the graph
  - Call Dijkstra's algorithm for v

Time complexity:

$$O(V(E \log V)) = O(EV \log V)$$

For dense graphs: $O(V^3 \log V)$

All-Pairs Shortest Paths

For weighted graphs (allowing negative weights):

- For each vertex v in the graph
  - Call Bellman-Ford algorithm for v

Time complexity:

$$O(V(VE)) = O(V^2 E)$$

For dense graphs: $O(V^4)$

Can we do better?

- Yes, Floyd-Warshall Algorithm returns all-pairs shortest distances in $O(V^3)$ for graphs allowing negative weights.

Floyd-Warshall Algorithm

- Algorithm based on dynamic programming.
- If the graph has a negative cycle, it will always be detected.
- For a graph without negative cycles, after the k-th iteration, dist[i][j] contains the weight of the shortest path from node i to node j that only uses intermediate nodes from the set {1,…, k}.

Floyd-Warshall Algorithm

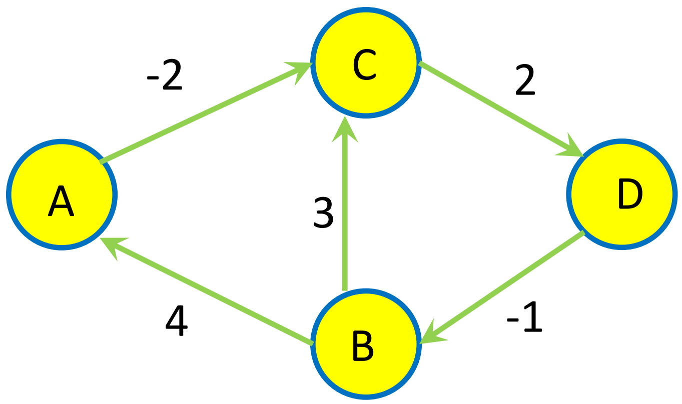
- Initialize adjacency matrix called dist[][] considering adjacent edges only
- For each vertex k in the graph
  - For each pair of vertices i and j in the graph
    - If dist(i → k → j) is smaller than the current dist(i→j)
      - Update/create shortcut i →j with weight equal to dist(i→k→j)

        i.e., update dist[i][j] = dist[i][k] + dist[k][j]

Assume that the outer for-loop will access vertices in the order A, B, C, D

First iteration of outer loop (i.e., k is 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 Algorithm

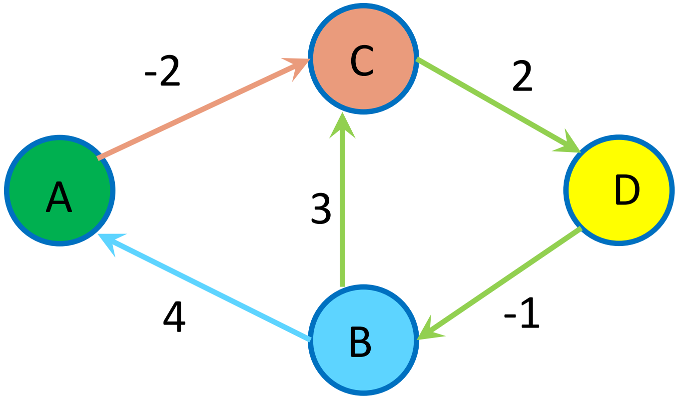
- Initialize adjacency matrix called dist[][] considering adjacent edges only
- For each vertex k in the graph
  - For each pair of vertices i and j in the graph
    - If dist(i → k → j) is smaller than the current dist(i→j)
      - Update/create shortcut i →j with weight equal to dist(i→k→j)

        i.e., update dist[i][j] = dist[i][k] + dist[k][j]

Assume that the outer for-loop will access vertices in the order A, B, C, D

First iteration of outer loop (i.e., k is 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



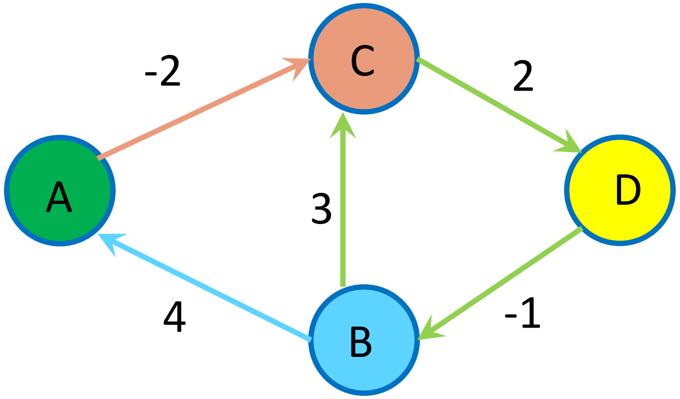
Floyd-Warshall Algorithm

- Initialize adjacency matrix called dist[][] considering adjacent edges only
- For each vertex k in the graph
  - For each pair of vertices i and j in the graph
    - If dist(i → k → j) is smaller than the current dist(i→j)
      - Update/create shortcut i →j with weight equal to dist(i→k→j)
        i.e., update dist[i][j] = dist[i][k] + dist[k][j]

Assume that the outer for-loop will access vertices in the order A, B, C, D

First iteration of outer loop (i.e., k is 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 Algorithm

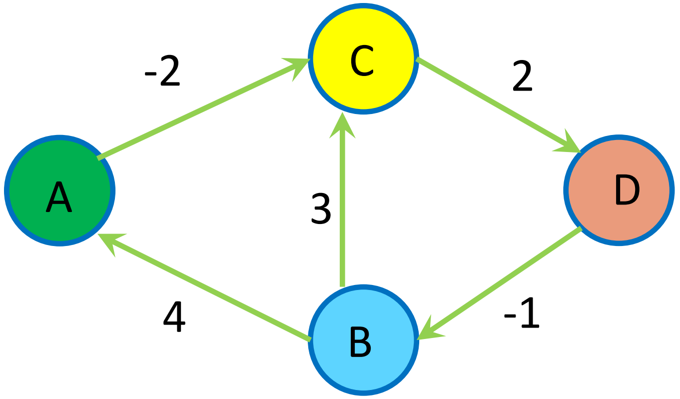
- Initialize adjacency matrix called dist[][] considering adjacent edges only
- For each vertex k in the graph
  - For each pair of vertices i and j in the graph
    - If dist(i → k → j) is smaller than the current dist(i→j)
      - Update/create shortcut i →j with weight equal to dist(i→k→j)

        i.e., update dist[i][j] = dist[i][k] + dist[k][j]

Assume that the outer for-loop will access vertices in the order A, B, C, D

First iteration of outer loop (i.e., k is 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 Algorithm

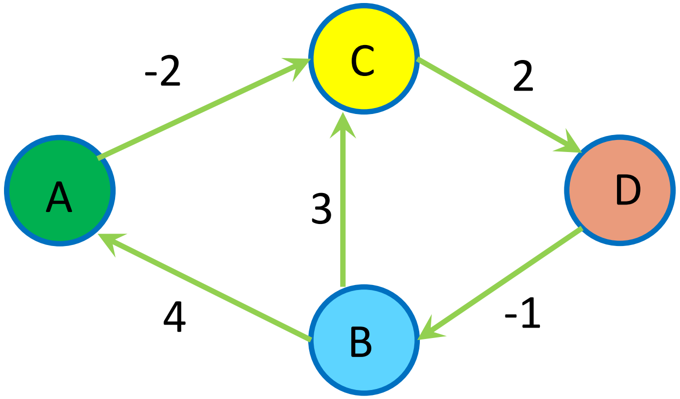
- Initialize adjacency matrix called dist[][] considering adjacent edges only
- For each vertex k in the graph
  - For each pair of vertices i and j in the graph
    - If dist(i → k → j) is smaller than the current dist(i→j)
      - Update/create shortcut i →j with weight equal to dist(i→k→j)
        i.e., update dist[i][j] = dist[i][k] + dist[k][j]

Assume that the outer for-loop will access vertices in the order A, B, C, D

First iteration of outer loop (i.e., k is A):

BA exists, but AD is currently inf, so we cannot update BD

	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 Algorithm

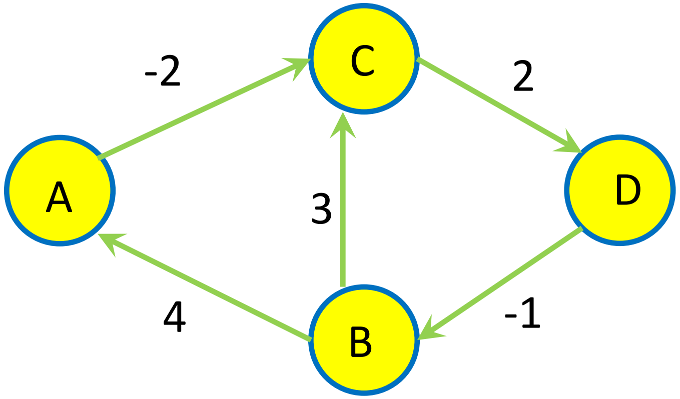
- Initialize adjacency matrix called dist[][] considering adjacent edges only
- For each vertex k in the graph
  - For each pair of vertices i and j in the graph
    - If dist(i → k → j) is smaller than the current dist(i→j)
      - Update/create shortcut i →j with weight equal to dist(i→k→j)
        i.e., update dist[i][j] = dist[i][k] + dist[k][j]

Assume that the outer for-loop will access vertices in the order A, B, C, D

First iteration of outer loop (i.e., k is A):

It is not possible to improve the distance between any other pair of nodes using only A as intermediate.

	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 Algorithm

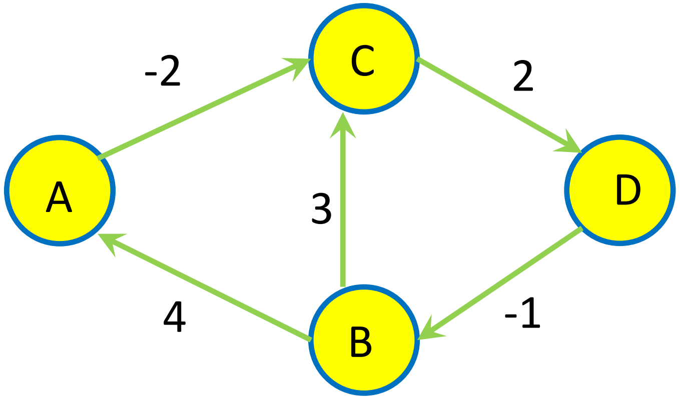
- Initialize adjacency matrix called dist[][] considering adjacent edges only
- For each vertex k in the graph
  - For each pair of vertices i and j in the graph
    - If dist(i → k → j) is smaller than the current dist(i→j)
      - Update/create shortcut i →j with weight equal to dist(i→k→j)

        i.e., update dist[i][j] = dist[i][k] + dist[k][j]

Assume that the outer for-loop will access vertices in the order A, B, C, D

Using nodes from {A, B} as intermediates, it is possible to update the following distances:

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



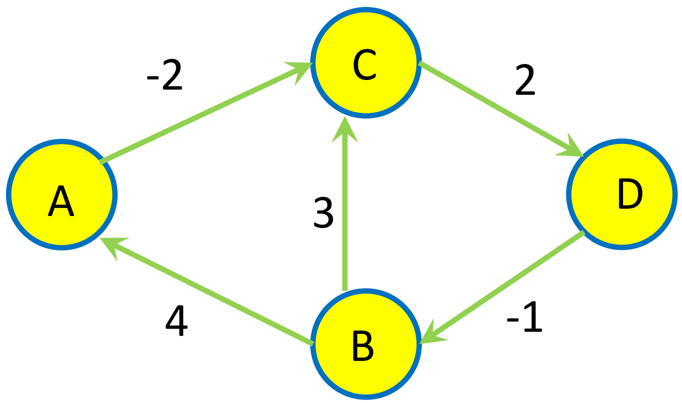
Floyd-Warshall Algorithm

- Initialize adjacency matrix called dist[][] considering adjacent edges only
- For each vertex k in the graph
  - For each pair of vertices i and j in the graph
    - If dist(i → k → j) is smaller than the current dist(i→j)
      - Update/create shortcut i →j with weight equal to dist(i→k→j)
        i.e., update dist[i][j] = dist[i][k] + dist[k][j]

Assume that the outer for-loop will access vertices in the order A, B, C, D

Using nodes from {A, B, C} as intermediates, it is possible to update the following distances:

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



Floyd-Warshall Algorithm

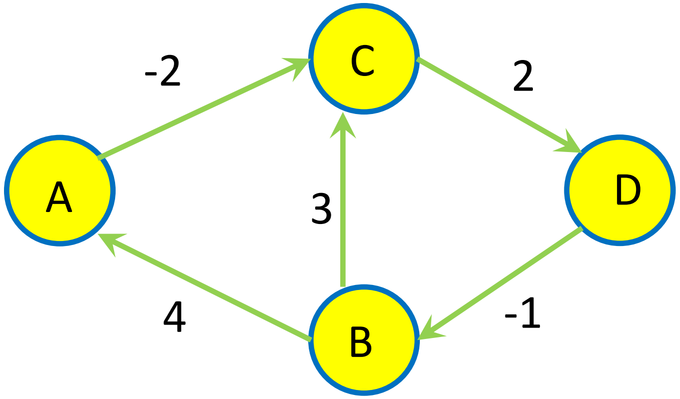
- Initialize adjacency matrix called dist[][] considering adjacent edges only
- For each vertex k in the graph
  - For each pair of vertices i and j in the graph
    - If dist(i → k → j) is smaller than the current dist(i→j)
      - Update/create shortcut i →j with weight equal to dist(i→k→j)

        i.e., update dist[i][j] = dist[i][k] + dist[k][j]

Assume that the outer for-loop will access vertices in the order A, B, C, D

Using nodes from {A, B, C, D} as intermediates, it is possible to update the following distances:

	A	B	C	D
A	0	-1	-2	0
B	4	0	2	4
C	5	1	0	2
D	3	-1	1	0



Floyd-Warshall Algorithm

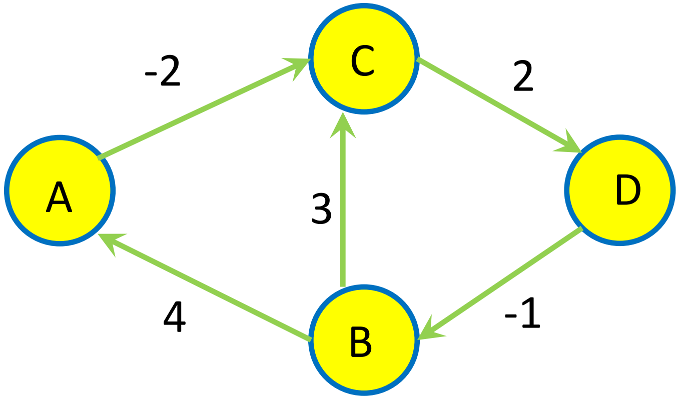
- Initialize adjacency matrix called dist[][] considering adjacent edges only
- For each vertex k in the graph
  - For each pair of vertices i and j in the graph
    - If dist(i → k → j) is smaller than the current dist(i→j)
      - Update/create shortcut i →j with weight equal to dist(i→k→j)

        i.e., update dist[i][j] = dist[i][k] + dist[k][j]

Assume that the outer for-loop will access vertices in the order A, B, C, D

Final Solution:

	A	**B**	**C**	**D**
A	0	-1	-2	0
B	4	0	2	4
C	5	1	0	2
D	3	-1	1	0



Floyd-Warshall Algorithm

```
dist[][] = E #  Initialize adjacency matrix using E
for vertex k in 1..V:
   #Invariant: dist[i][j] corresponds to the shortest path from i
to j considering the intermediate vertices 1 to k-1
   for vertex i in 1..V:
      for vertex j in 1..V:
         dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
```

Time Complexity:

$O(V^3)$

Space Complexity:

$O(V^2)$

Floyd-Warshall Algorithm: Correctness

```
Invariant: dist[i][j] corresponds to the shortest path from
i to j considering only intermediate vertices 1 to k-1
```

Base Case k = 1 (i.e. there are no intermediate vertices yet):

- It is true because dist[][] is initialized based only on the adjacent edges

Inductive Step:

- Assume dist[i][j] is the shortest path from i to j detouring through only vertices 1 to k-1
- Adding the k-th vertex to the “detour pool” can only help if the best path detours through k
- Thus, minimum of dist(i→k→j) and dist(i→j) gives the minimum distance from i to j considering the intermediate vertices 1 to k

Floyd-Warshall Algorithm: Correctness

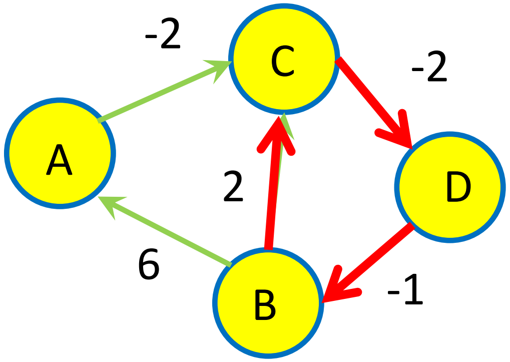
`Invariant: dist[i][j] corresponds to the shortest path from i to j considering only intermediate vertices 1 to k-1`

- Adding the k-th vertex to the “detour pool” can only help if the best path detours through k
- We already know the best way to get from i to k (using only vertices in 1…k-1) and we know the best way to get from j to k (using only vertices in 1…k-1)
- Thus, minimum of dist(i→k→j) and dist(i→j) gives the minimum distance from i to j considering the intermediate vertices 1 to k

Floyd-Warshall Algorithm: Negative Cycles

- If there is a negative cycle, there will be a vertex v such that dist[v][v] is negative.
- Look at the diagonal of the adjacency matrix and return error if a negative value is found
- How could you modify the algorithm to return the **paths**?

	A	B	C	D
A	0	-5	-3	-5
B	5	-1	1	-1
C	3	-3	-1	-3
D	4	-2	0	-2



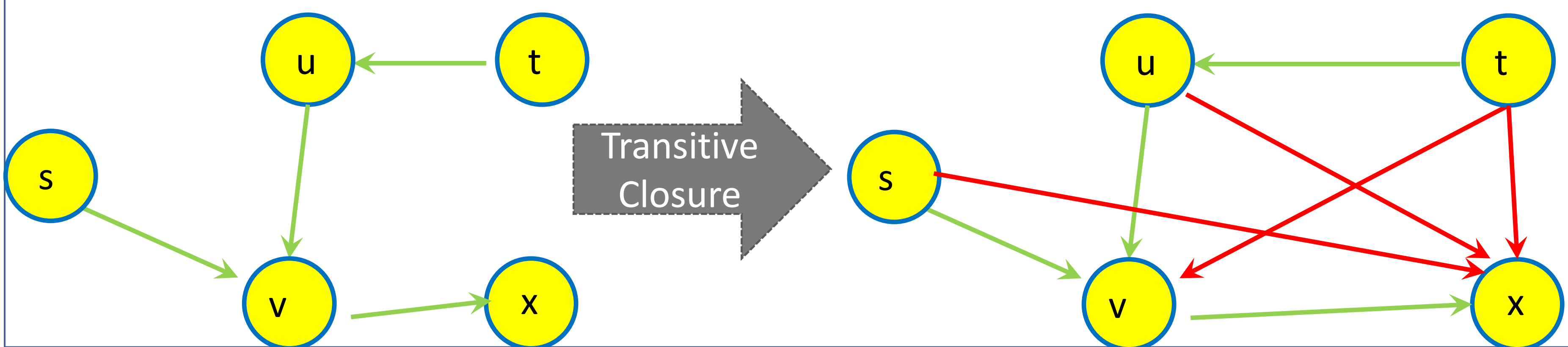
Outline

1. Shortest path in a graph with negative weights
2. All-pairs shortest paths
3. Transitive Closure

Transitive Closure of a Graph

- Given a graph G = (V,E), its transitive closure is another graph (V,E') that contains the same vertices V but contains an edge from node u to node v if there is a path from u to v in the original graph.

- **Solution:** Assign each edge a weight 1 and then apply Floyd-Warshall algorithm. If dist[i][j] is not infinity, this means there is a path from i to j in the original graph. (Or just maintain True and False as shown next)



Floyd-Warshall Algorithm for Transitive Closure

```
# Modify Floyd-Warshall Algorithm to compute Transitive Closure
# initialization
for vertex i in 1..V:
        for vertex j in 1..V:
            if there is an edge between i and j or i == j:
                TC[i][j] = True
            else:
                TC[i][j] = False
for vertex k in 1..V:
# Invariant: TC[i][j] corresponds to the existence of path from i to j considering the intermediate vertices 1 to k-1
    for vertex i in 1..V:
        for vertex j in 1..V:
            TC[i][j] = TC[i][j] or (TC[i][k] and TC[k][j])
```

Time Complexity:

$O(V^3)$

Space Complexity:

$O(V^2)$

Summary

Take home message

- Dijkstra's algorithm works only for graphs with non-negative weights.
- Bellman-Ford computes shortest paths in graphs with negative weights in O(VE) and can also detect the negative cycles that are reachable.
- Floyd-Warshall Algorithm computes all-pairs shortest paths and transitive closure in $O(V^3)$.

Things to do (this list is not exhaustive)

- Go through recommended reading and make sure you understand why the algorithms are correct.
- Implement Bellman-Ford and Floyd-Warshall Algorithms.

Coming Up Next

- Minimum spanning trees