Faculty of Information Technology,
Monash University

FIT2004: Algorithms and Data Structures

Week 7: Dynamic Programming Graph Algorithms

Outline

Divide and conquer (W 1-3)	Greedy algorithms (W 4-5)	Dynamic programming (W 6-7)	Network flow (W 8-9)	Data structures (W 10-11)

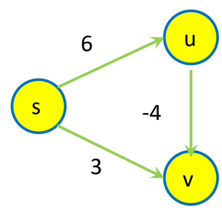
- Last Lecture: DP algorithms
  - Coins Change
  - Knapsack
  - Edit Distance
- Today's Lecture: **DP graph algorithms**
  - Shortest path in graphs with negative weights
  - All-pairs shortest paths
  - Transitive Closure

Outline

1. Shortest path in a graph with negative weights (Bellman-Ford Algorithm)
2. All-pairs shortest paths (Floyd-Warshall Algorithm)
3. Transitive Closure

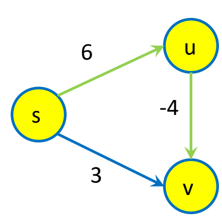
Shortest path (negative weights)

- 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.



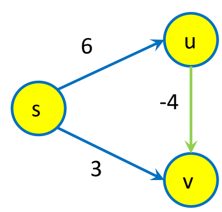
Shortest path (negative weights)

- 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.



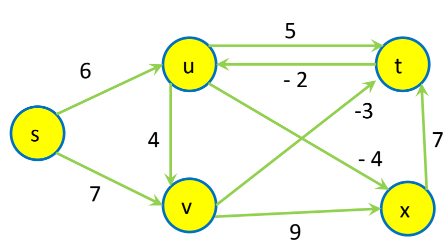
Shortest path (negative weights)

- 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 **<u>not guaranteed</u>** to output the correct answer when there are negative weights.



Shortest path (negative weights)

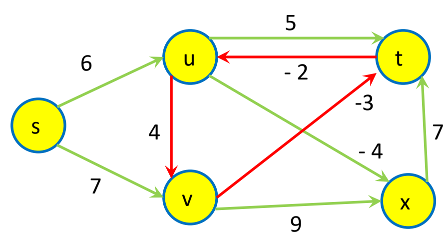
What is the shortest distance from s to x in this graph?



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's algorithm **works correctly even with negative edge weights** (***as long as there are no negative weight cycles***).
- 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).
- It can be modified to return all valid shortest distances, and minus ∞ for vertices which are affected by the negative cycle.

Bellman-Ford Algorithm: Core Idea

- 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).

- Can the shortest path from s to t have a positive cycle?

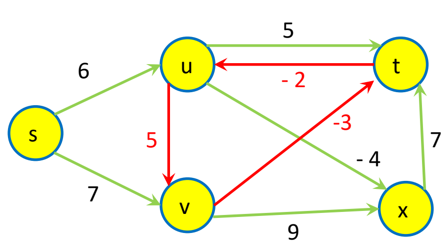
  No, the **shortest path** from s to t will never include a cycle (either positive or negative) if the goal is to minimize the total cost of the path.

  Why is that?

  - A cycle should revisit at least one node.
  - If the cycle is positive (i.e., the total weight of the cycle is > 0), then going through it increases the total cost of the path.
  - So a shorter path can be found by just skipping that cycle.
  - Therefore, the shortest path will always be simple (i.e., no repeated nodes).

Bellman-Ford Algorithm: Core 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 a 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 Week 5: 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.

Proof:

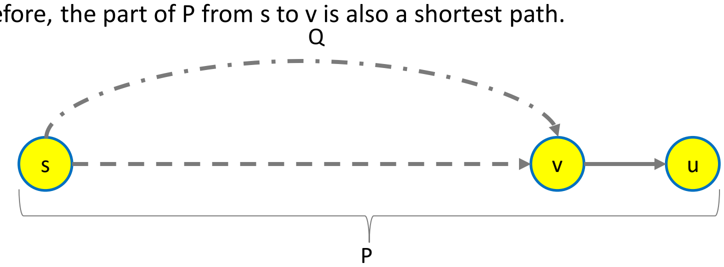
Suppose there was a shorter path from s to v, say Q.

If it's the shortest to reach u,

weight(Q) + w(v,u) < weight(P)

But P is the shortest path from s to u. (Contradiction)

Therefore, the part of P from s to v is also a shortest path.



Bellman-Ford Algorithm

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

Richard Bellman

Lester Ford Jr.

Bellman-Ford Algorithm

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

- 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:

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

FIT2004, Lecture 7 - DP Graph Algorithms

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]
```

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

What is the time complexity of Bellman-ford algorithm?

FIT2004, Lecture 7 - DP Graph Algorithms

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]
```

What is the time complexity of Bellman-ford algorithm?

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 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).

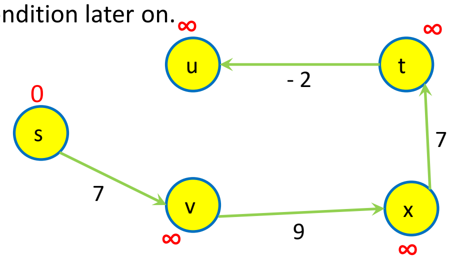
Bellman-Ford Algorithm

- 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.

i	s	v	x	t	u	u	t	x	v	s
1	0	7	16	23	21	∞	∞	∞	7	0
2						∞	∞	16	7	0
3						∞	23	16	7	0
4						21	23	16	7	0



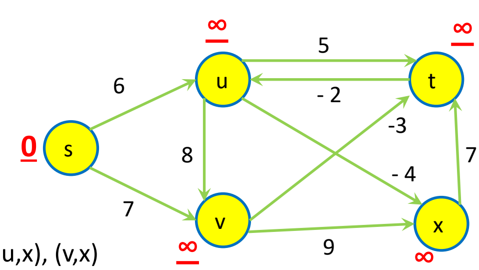
Bellman-Ford Algorithm: Example

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))



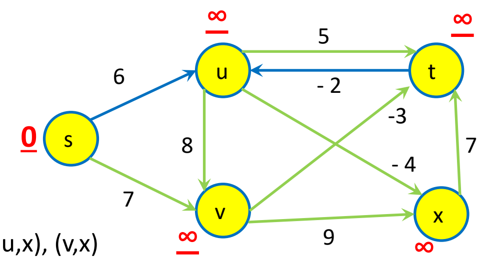
Assume the following order:

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

Bellman-Ford Algorithm: Example

First iteration:

Relaxing incoming edges of node u



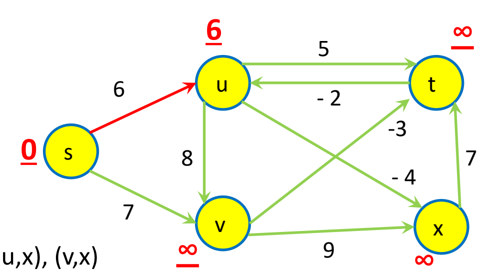
Assume the following order:

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

Bellman-Ford Algorithm: Example

First iteration:

Done relaxing incoming edges of node u



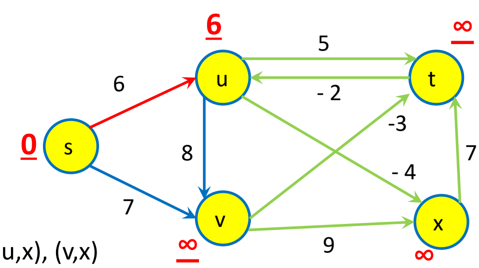
Assume the following order:

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

Bellman-Ford Algorithm: Example

First iteration:

Relaxing incoming edges of node v



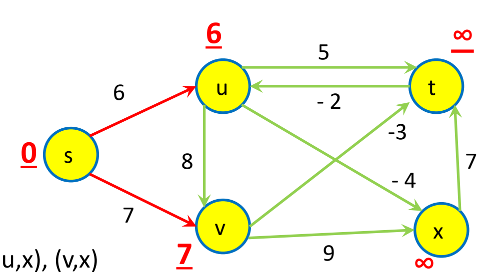
Assume the following order:

(s,u), (t,u), $\boxed{\text{(s,v), (u,v),}}$ (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm: Example

First iteration:

Done relaxing incoming edges of node v



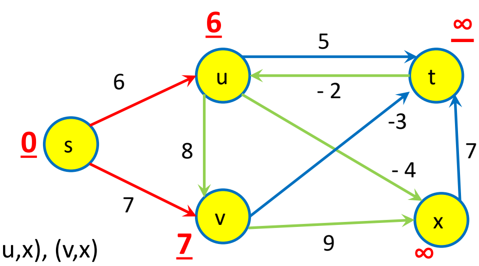
Assume the following order:

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

Bellman-Ford Algorithm: Example

First iteration:

Relaxing incoming edges of node t



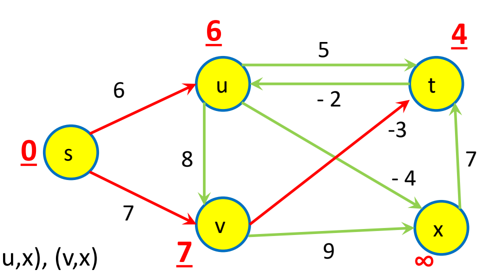
Assume the following order:

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

Bellman-Ford Algorithm: Example

First iteration:

Done relaxing incoming edges of node t



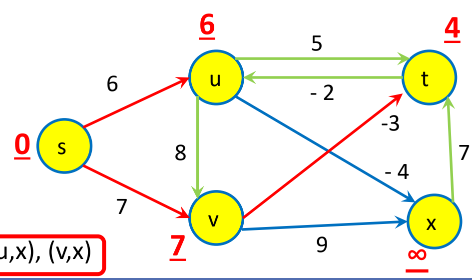
Assume the following order:

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

Bellman-Ford Algorithm: Example

First iteration:

Relaxing incoming edges of node x



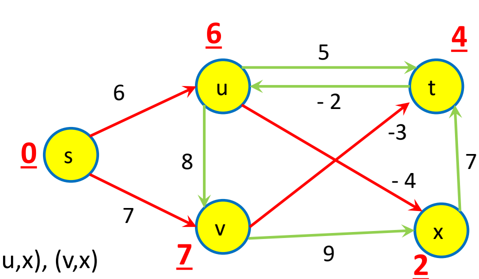
Assume the following order:

(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), $\boxed{\text{(u,x), (v,x)}}$

Bellman-Ford Algorithm: Example

First iteration:

Done relaxing incoming edges of node x

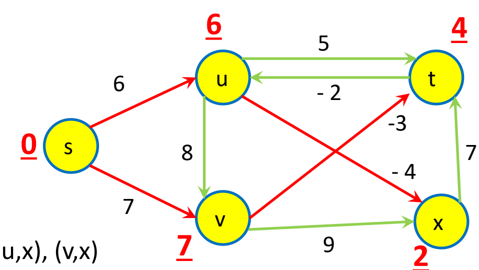


Assume the following order:

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

Bellman-Ford Algorithm: Example

First iteration finished:

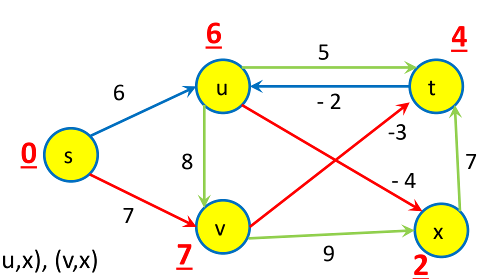


Assume the following order:
(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm: Example

Second iteration:

Relaxing incoming edges of node u

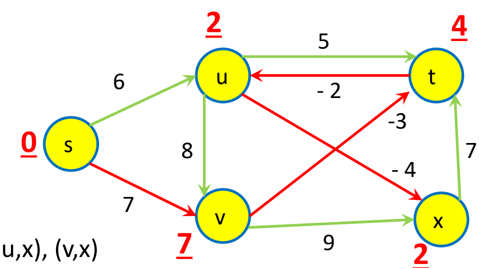


Assume the following order:
(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm: Example

Second iteration:

Done relaxing incoming edges of node u

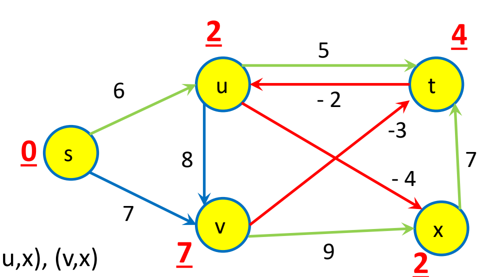


Assume the following order:
(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm: Example

Second iteration:

Relaxing incoming edges of node v

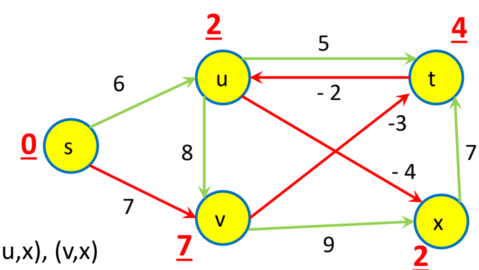


Assume the following order:
(s,u), (t,u), $\boxed{\text{(s,v), (u,v),}}$ (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm: Example

Second iteration:

Done relaxing incoming edges of node v

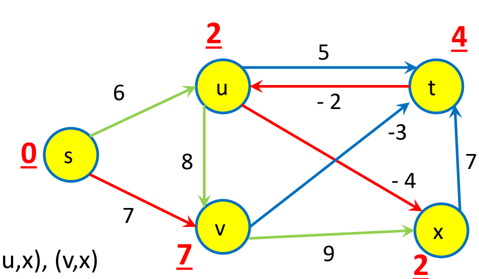


Assume the following order:
(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm: Example

Second iteration:

Relaxing incoming edges of node t

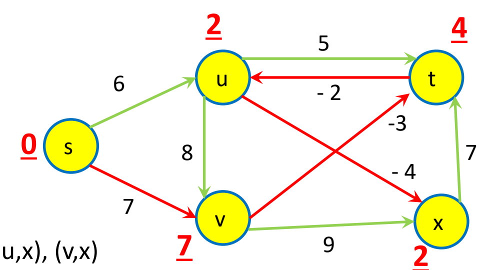


Assume the following order:
(s,u), (t,u), (s,v), (u,v), $\boxed{\text{(u,t), (v,t), (x,t),}}$ (u,x), (v,x)

Bellman-Ford Algorithm: Example

Second iteration:

Done Relaxing incoming edges of node t

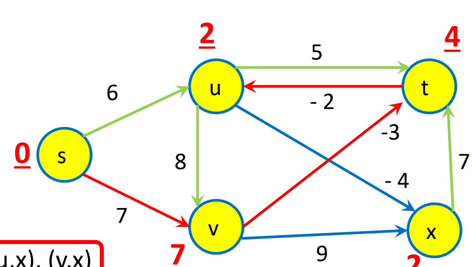


Assume the following order:
(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm: Example

Second iteration:

Relaxing incoming edges of node x

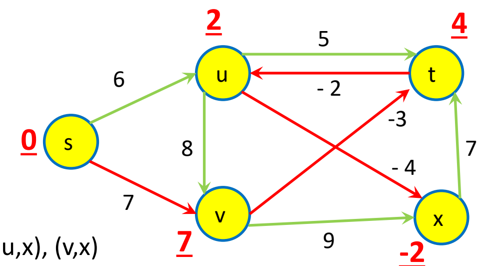


Assume the following order:
(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), $\boxed{\text{(u,x), (v,x)}}$

Bellman-Ford Algorithm: Example

Second iteration:

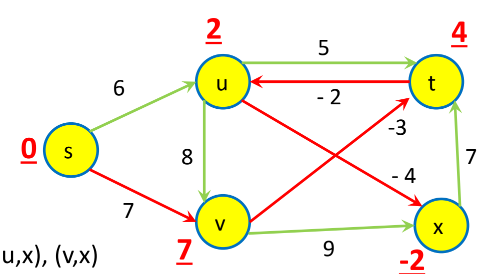
Done Relaxing incoming edges of node x



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: Example

Second iteration finished:



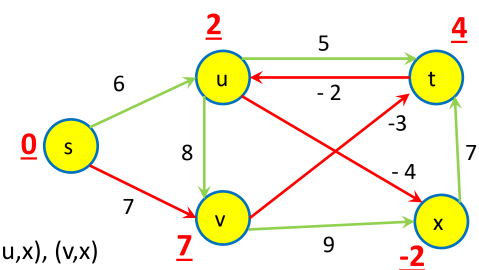
Assume the following order:
(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Bellman-Ford Algorithm: Example

Third iteration:

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

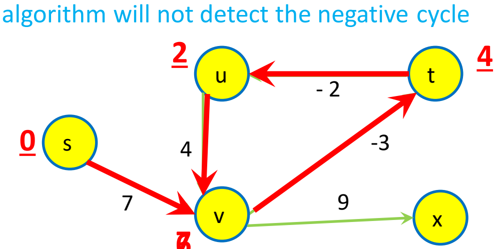
Early Stop Condition: If nothing changes in one iteration, it is possible to stop the execution of the Bellman-Ford algorithm and output the current values.



Assume the following order:
(s,u), (t,u), (s,v), (u,v), (u,t), (v,t), (x,t), (u,x), (v,x)

Finding Negative Cycles

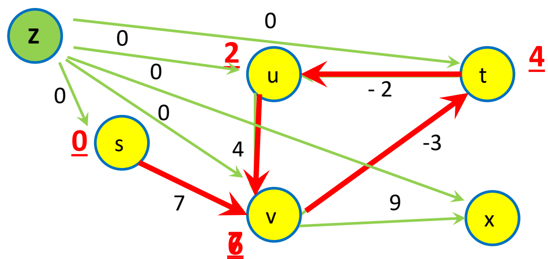
- If V-th iteration reduces the distance of a vertex, it tells us that there is a shorter path with at least V edges.
  - This implies that there is a negative cycle.
- For example, consider the graph with 4 vertices s, u, v, and t and assume we have run (V-1 = 3) iterations.
- In the 4th iteration, 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 Negative Cycles

To detect if a graph G has a negative cycle:

Just add one extra node (z) to G and edges from it to every other node, and run Bellman-Ford on the added node, z.



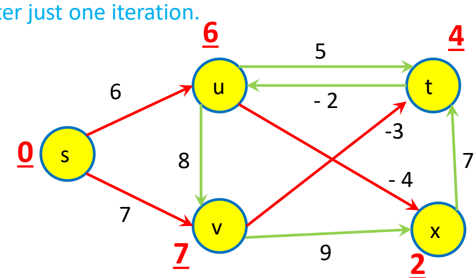
Updated Bellman-Ford Algorithm

```
# 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 cycle 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 of the outer loop.
  - 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.



Handling Negative Cycles

- How could we modify Bellman-Ford to determine **which** vertices have valid distances, and which are affected by the negative cycle?
- Execute V additional iterations, and for each node whose distance would be updated, just mark its distance as -∞.
  - By continuing to relax the graph for V more iterations, the effect of negative cycles is allowed to propagate across the graph.
  - Therefore, any vertex that can be reached from a negative cycle will eventually get -∞ as its distance.

Outline

1. Shortest path in a graph with negative weights (Bellman-Ford Algorithm)
2. All-pairs shortest paths (Floyd-Warshall Algorithm)
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 (BFS) 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: $E \approx O(V^2)$, 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: $O(V^3 \log V)$ since $E \approx O(V^2)$

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: $O(V^4)$ since $E \approx O(V^2)$

Can we do better?

- Yes, **Floyd-Warshall algorithm** returns all-pairs shortest distances in $O(V^3)$ (even for graphs with negative weights).

Floyd-Warshall Algorithm

- Algorithm based on dynamic programming.

Robert W. Floyd
(Turing Award 1978)

Stephen Warshall

Floyd-Warshall Algorithm

- Algorithm based on dynamic programming.
- Computes the shortest distance between every pair of nodes as long as there are no negative weight cycles.
- If the graph has a negative cycle, it will always be detected.
  - Is this similar to Bellman-Ford algorithm?
    - No.
    - Bellman-Ford **only** detects negative cycles that are reachable from the source node.
- 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}.

Key idea behind Floyd-Warshall Algorithm

- For each pair of vertices `(i,j)`, find whether there is a shorter path from `i` to `j` by going through an intermediate vertex `k`.
- All variables `i,j,k` refer to vertices.
- At each step, it updates:

```
dist[i][j] = min(dist[i][j],
                 dist[i][k]+dist[k][j])
```

- It does this for every possible value of `k` from `0` to `V - 1`.
- After the algorithm finishes, check for all vertices:

`dist[i][i] < 0` which tells that a Negative weight cycle is detected.

Floyd-Warshall Algorithm

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→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 to A

To vertex

From vertex	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 = A B C D

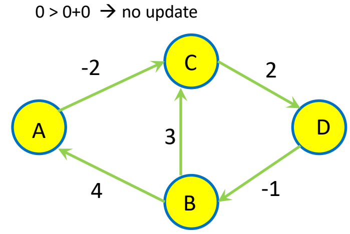
i = A B C D

j = A B C D

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

dist[A][A] > dist[A][A] + dist[A][A]

0 > 0+0 → no update



Floyd-Warshall Algorithm

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→k→j)

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 to B

	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 = A B C D

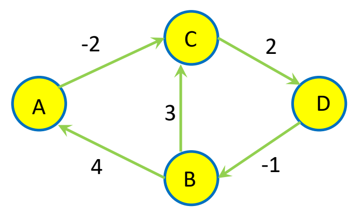
i = A B C D

j = A B C D

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

dist[A][B] > dist[A][A] + dist[A][B]

∞ > 0+∞ → no update



Floyd-Warshall Algorithm

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→k→j)

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 to 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

k = A B C D

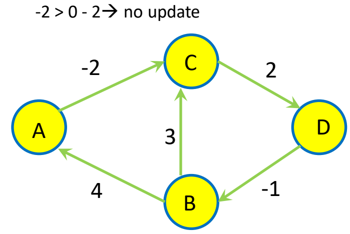
i = A B C D

j = A B C D

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

dist[A][C] > dist[A][A] + dist[A][C]

-2 > 0 - 2→ no update



Floyd-Warshall Algorithm

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→k→j)

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 to D

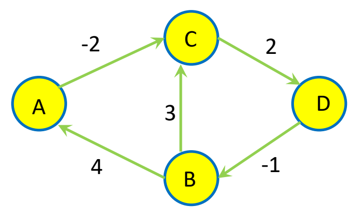
k = A B C D

i = A B C D

j = A B C D

dist[i][j] > dist[i][k] + dist[k][j]
dist[A][D] > dist[A][A] + dist[A][D]
∞ > 0+∞ → no update

	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

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→k→j)

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

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

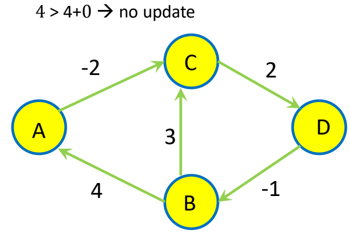
k = A B C D

i = A B C D

j = A B C D

dist[i][j] > dist[i][k] + dist[k][j]
dist[B][A] > dist[B][A] + dist[A][A]
4 > 4+0 → no update

	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

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→k→j)

k = A B C D

i = A B C D Let's speed up

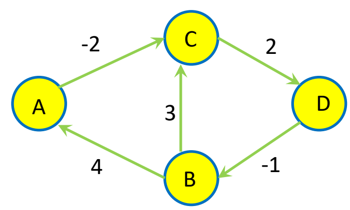
j = A B C D

dist[i][j] > dist[i][k] + dist[k][j]
dist[B][C] > dist[B][A] + dist[A][C]
3 > 4-2 → yes, then update

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

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



Floyd-Warshall Algorithm

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→k→j)

k = A B C D

i = A B C D Let's speed up

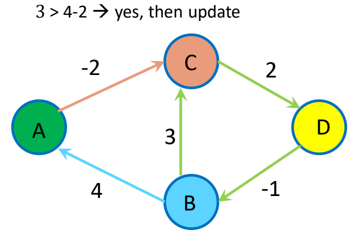
j = A B C D

dist[i][j] > dist[i][k] + dist[k][j]
dist[B][C] > dist[B][A] + dist[A][C]
3 > 4-2 → yes, then update

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

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



FIT2004, Lecture 7 - DP Graph Algorithms

Floyd-Warshall Algorithm

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→k→j)

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

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

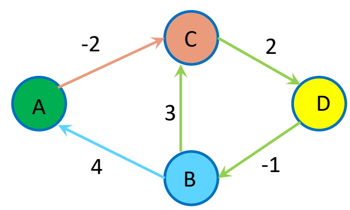
	A	B	C	D
A	0	Inf	-2	Inf
B	4	0	2	Inf
C	Inf	Inf	0	2
D	Inf	-1	Inf	0

k = A B C D

i = A B C D Let's speed up

j = A B C D

dist[i][j] > dist[i][k] + dist[k][j]
dist[B][C] > dist[B][A] + dist[A][C]
3 > 4-2 → yes, then update



Floyd-Warshall Algorithm

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→k→j)

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

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

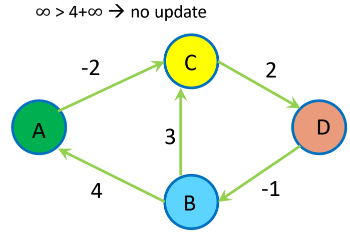
	A	B	C	D
A	0	Inf	-2	Inf
B	4	0	2	Inf
C	Inf	Inf	0	2
D	Inf	-1	Inf	0

k = A B C D

i = A B C D Let's speed up

j = A B C D

dist[i][j] > dist[i][k] + dist[k][j]
dist[B][D] > dist[B][A] + dist[A][D]
∞ > 4+∞ → no update



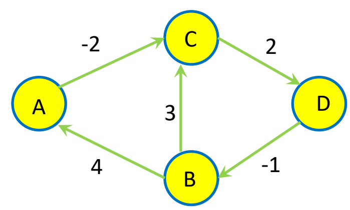
Floyd-Warshall Algorithm

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→k→j)

First iteration of outer loop (i.e., k is A):
The remaining pairs are (C,A), (C,B), (C,C), (C,D), (D,A), (D,B), (D,C) and (D,D).
It is not possible to improve the distance between any of these pairs of nodes using only A as intermediate.
Hence, we move to next iteration.

k = A B C D
i = A B C D
j = A B C D
dist[i][j] > dist[i][k] + dist[k][j]

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



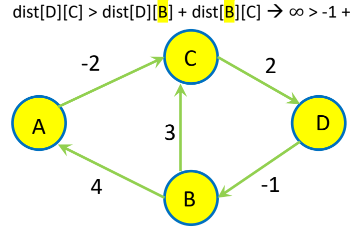
Floyd-Warshall Algorithm

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→k→j)

The outer for-loop will access vertices in the order A, B, C, D
Second iteration of outer loop (i.e., k is B): D to A and D to C

k = A B C D
i = A B C D
j = A B C D
dist[i][j] > dist[i][k] + dist[k][j]
dist[D][A] > dist[D][B] + dist[B][A] → ∞ > -1 + 4, update
dist[D][C] > dist[D][B] + dist[B][C] → ∞ > -1 + 2, update

	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

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→k→j)

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

Second iteration of outer loop (i.e., k is B): D to A and D to C

k = A B C D

i = A B C D

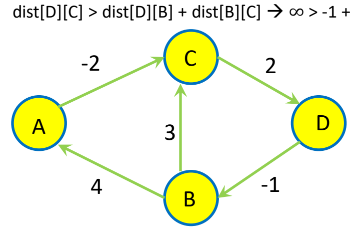
j = A B C D

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

dist[D][A] > dist[D][B] + dist[B][A] → ∞ > -1 + 4 = 3

dist[D][C] > dist[D][B] + dist[B][C] → ∞ > -1 + 2 = 1

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



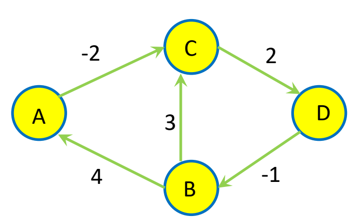
Floyd-Warshall Algorithm

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→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



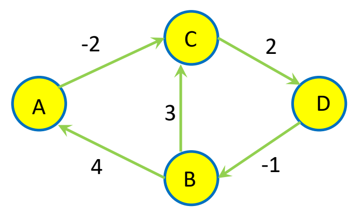
Floyd-Warshall Algorithm

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→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



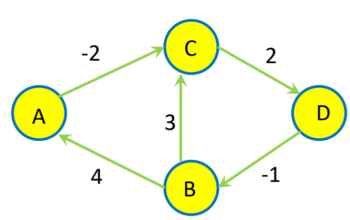
Floyd-Warshall Algorithm

- Initialise 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] + dist[k][j] < dist[i][j] // i.e., dist(i → k → j) is smaller than the current dist(i→j)
      - Update dist[i][j] = dist[i][k] + dist[k][j] // create shortcut i →j with weight equal to dist(i→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

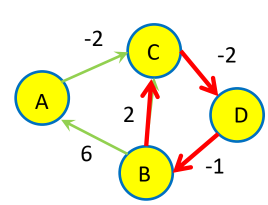
Invariant: dist[i][j] corresponds to the shortest path from i to j considering only intermediate vertices 1 to k-1

- 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 k to j (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 final matrix and **return error** if a negative value is found.
- How could you modify the algorithm to return the **paths**?
  - Add a "predecessor" matrix that stores the path information when the shortest distances are updated.

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



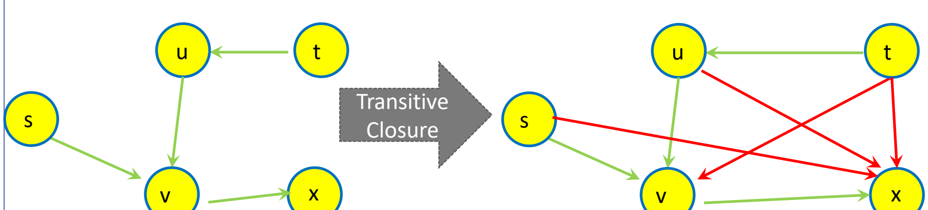
Outline

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 that 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:
$O(V^2)$

Reading

- Course Notes: Chapter 8
- You can also check algorithms' textbooks for contents related to this lecture, e.g.:
  - CLRS: Sections 24.1 and 25.2
  - KT: Sections 6.8, 6.9 and 6.10
  - Rou: Chapter 18

Concluding Remarks

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

- Network Flow