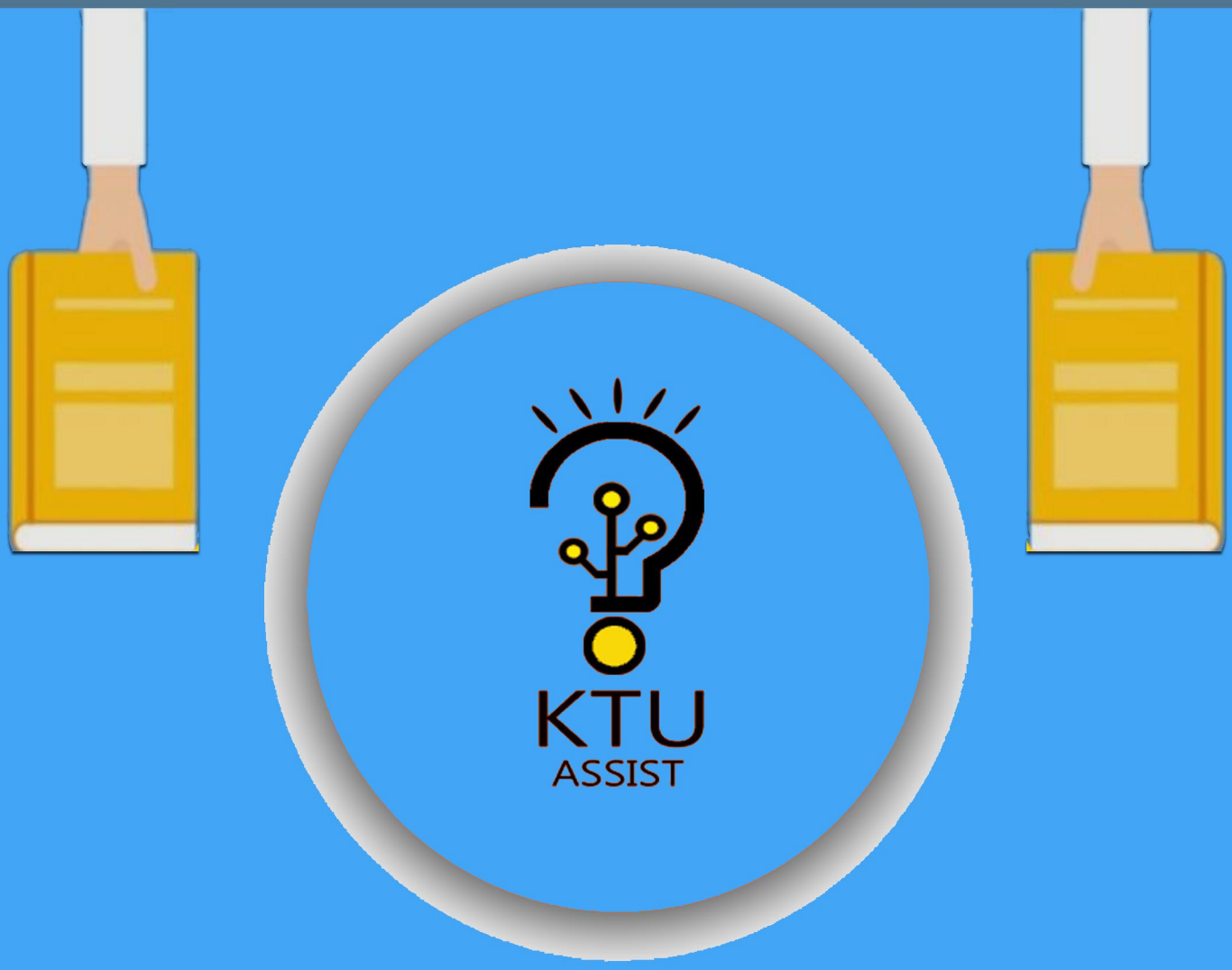
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

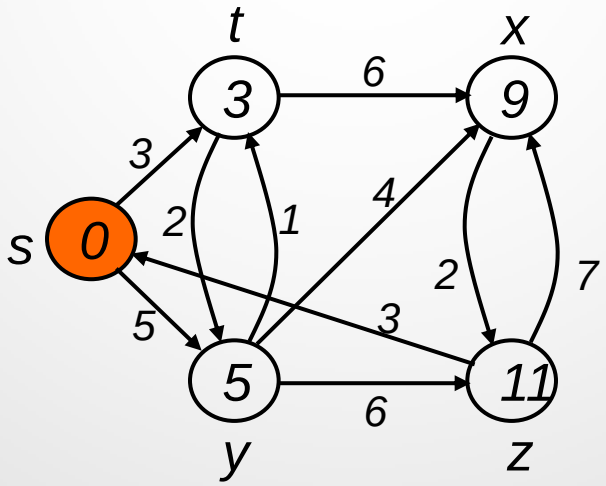
STUDY MATERIALS



Single-Source Shortest Path

Shortest Path Problems

- Directed weighted graph.
- Path length is sum of weights of edges on path.
- The vertex at which the path begins is the source vertex.
- The vertex at which the path ends is the destination vertex.



Shortest Path Variants

- Single-source single-destination (1-1): Find the shortest path from source *s* to destination *v*.
- Single-source all-destination(1-Many): Find the shortest path from *s* to each vertex *v*.
- Single-destination shortest-paths (Many-1): Find a shortest path to a given *destination* vertex *t* from each vertex *v*.
- All-pairs shortest-paths problem (Many-Many): Find a shortest path from *u* to *v* for every pair of vertices *u* and *v*.

Shortest Path Variants

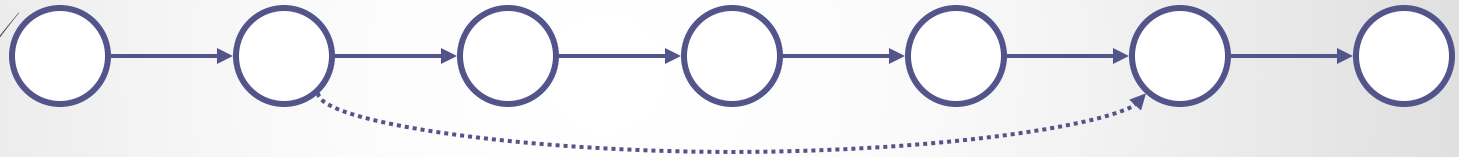
Single-Source Single-Destination (1-1) - No good solution that beats 1-M variant - Thus, this problem is mapped to the 1-M variant	**Single-Source All-Destination (1-M)** - Need to be solved (several algorithms) - We will study this one
All-Sources Single-Destination (M-1) - Reverse all edges in the graph - Thus, it is mapped to the (1-M) variant	**All-Sources All-Destinations (M-M)** - Need to be solved (several algorithms) - We will study it (if time permits)

Single-Source Shortest Path

- Problem: given a weighted directed graph G, find the minimum-weight path from a given source vertex s to another vertex v
  - "Shortest-path" = minimum weight
  - Weight of path is sum of edges
  - E.g., a road map: what is the shortest path from Kunnakulam to Thrissur?

Shortest Path Properties

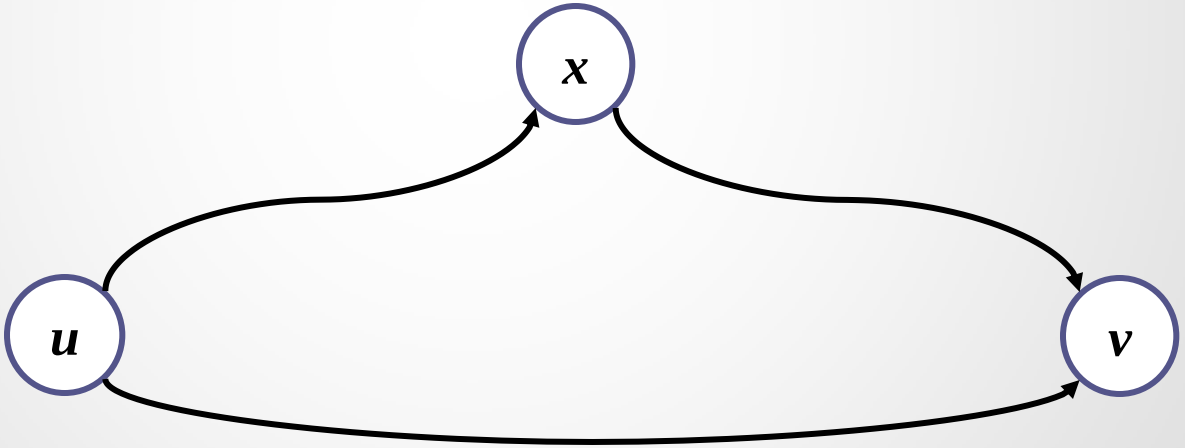
- *Optimal substructure*: the shortest path consists of shortest subpaths:



  - suppose some subpath is not a shortest path
    - There must then exist a shorter subpath
    - Could substitute the shorter subpath for a shorter path
    - But then overall path is not shortest path. Contradiction

Shortest Path Properties

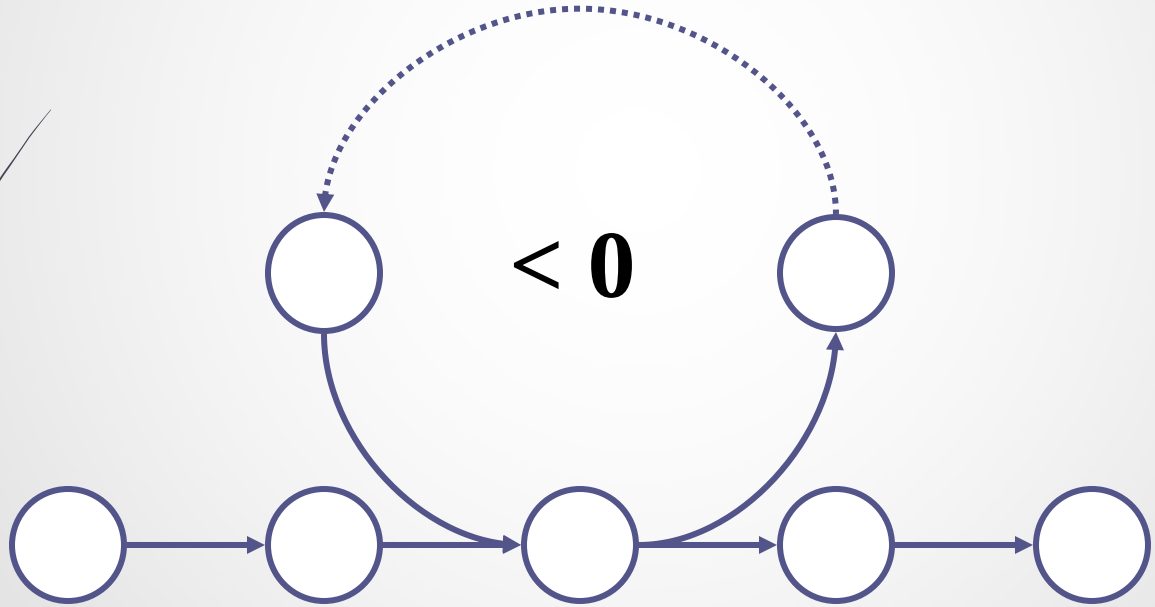
- Define $\delta(u,v)$ to be the weight of the shortest path from u to v
- *Triangle inequality*: $\delta(u,v) \leq \delta(u,x) + \delta(x,v)$



This path is no longer than any other path

Shortest Path Properties

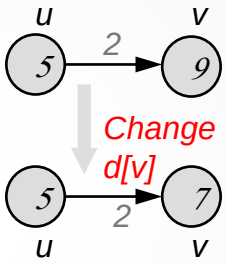
- In graphs with negative weight cycles, some shortest paths will not exist

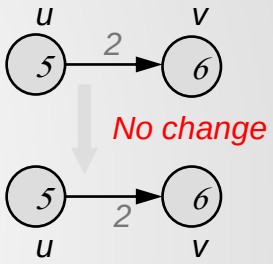


Relaxation

Algorithms keep track of d[v], π[v]. ***Initialized*** *as follows:*

```
Initialize(G, s)
    for each v ∈ V[G] do
        d[v] := ∞;
        π[v] := NIL
    od;
    d[s] := 0
```





These values are changed when an edge (u, v) is ***relaxed****:*

```
Relax(u, v, w)
    if d[v] > d[u] + w(u, v) then
        d[v] := d[u] + w(u, v);
        π[v] := u
    fi
```

Bellman-Ford Algorithm

Can have negative-weight edges.

Will "detect" reachable negative-weight cycles.

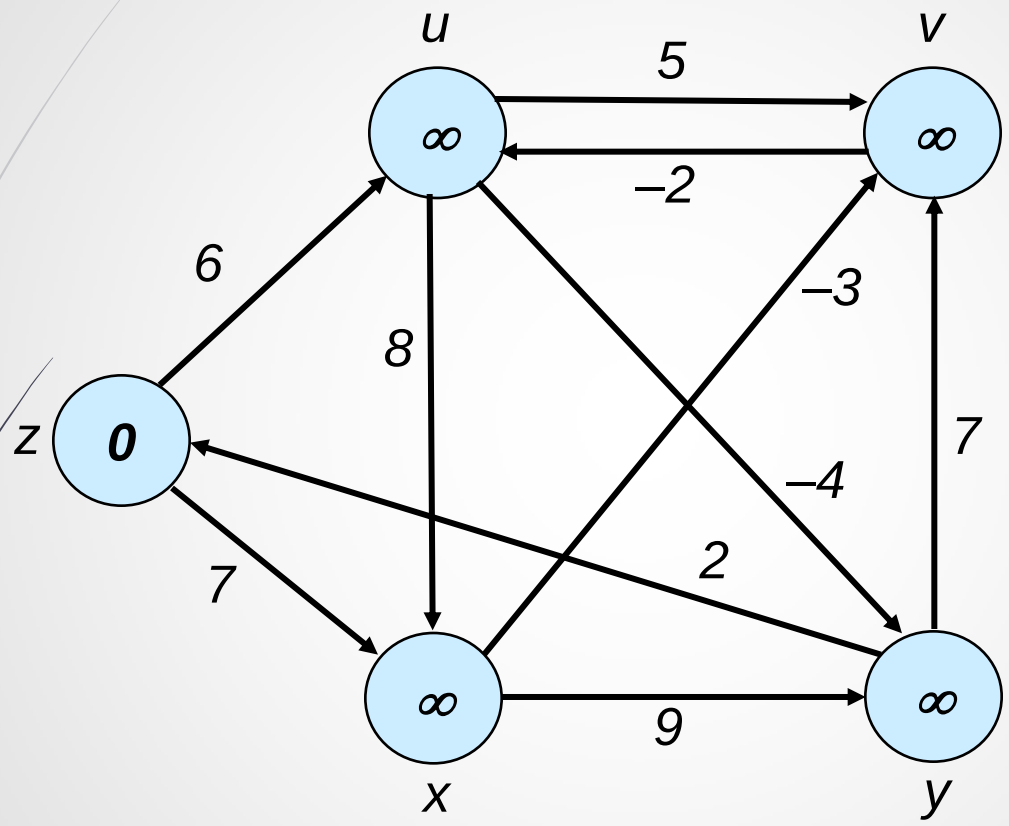
```
Initialize(G, s);
for i := 1 to |V[G]| –1 do
        for each (u, v) in E[G] do
                Relax(u, v, w)
        od
od;
for each (u, v) in E[G] do
        if d[v] > d[u] + w(u, v) then
                return false
        fi
od;
return true
```

Time Complexity is O(VE).

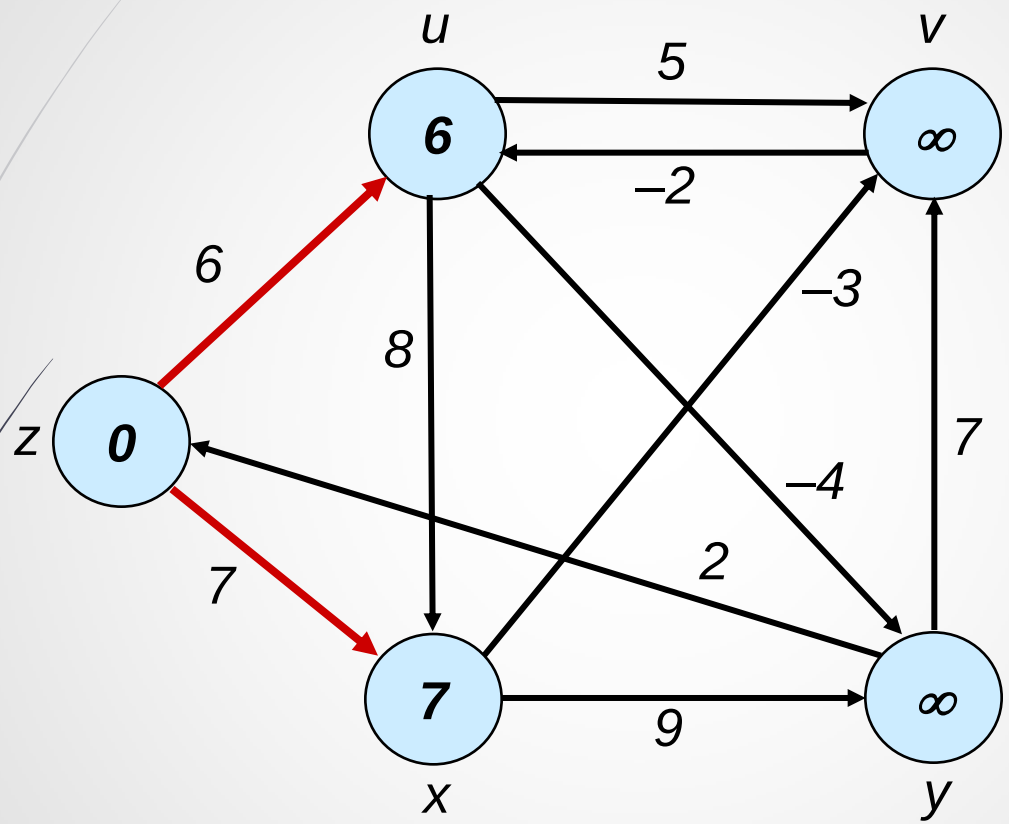
Contd...

- So if Bellman-Ford has not converged after V(G) - 1 iterations, then there cannot be a shortest path tree, so there must be a negative weight cycle.

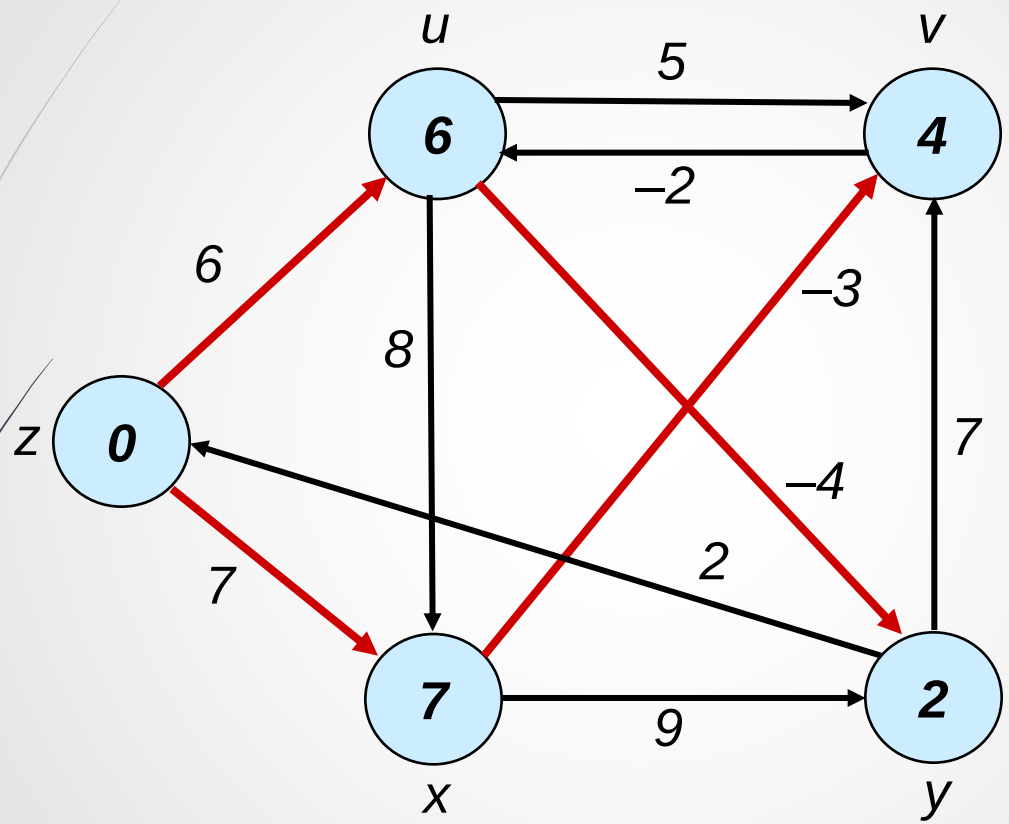
Example



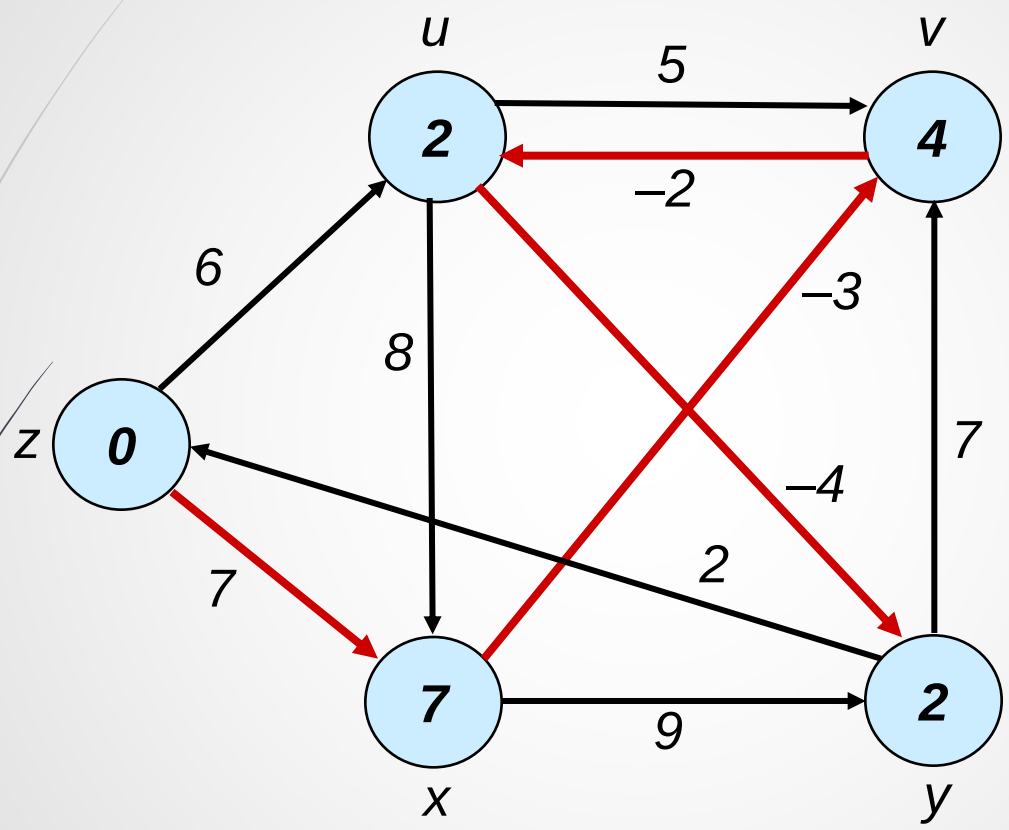
Example



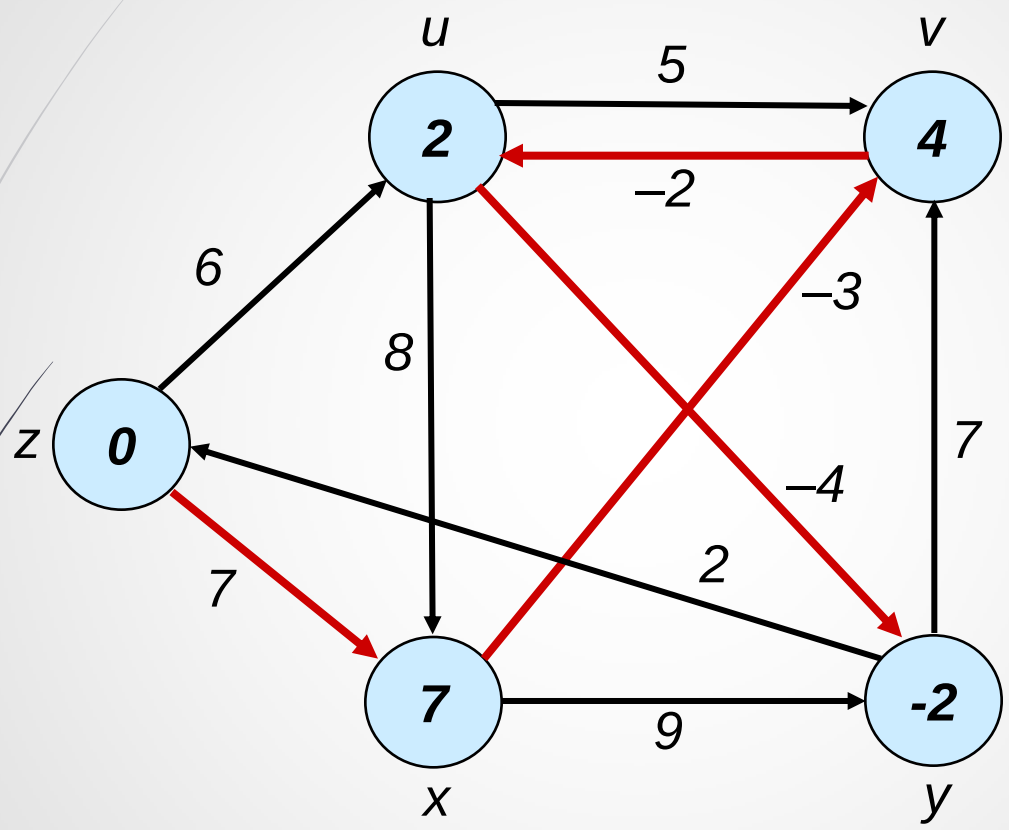
Example



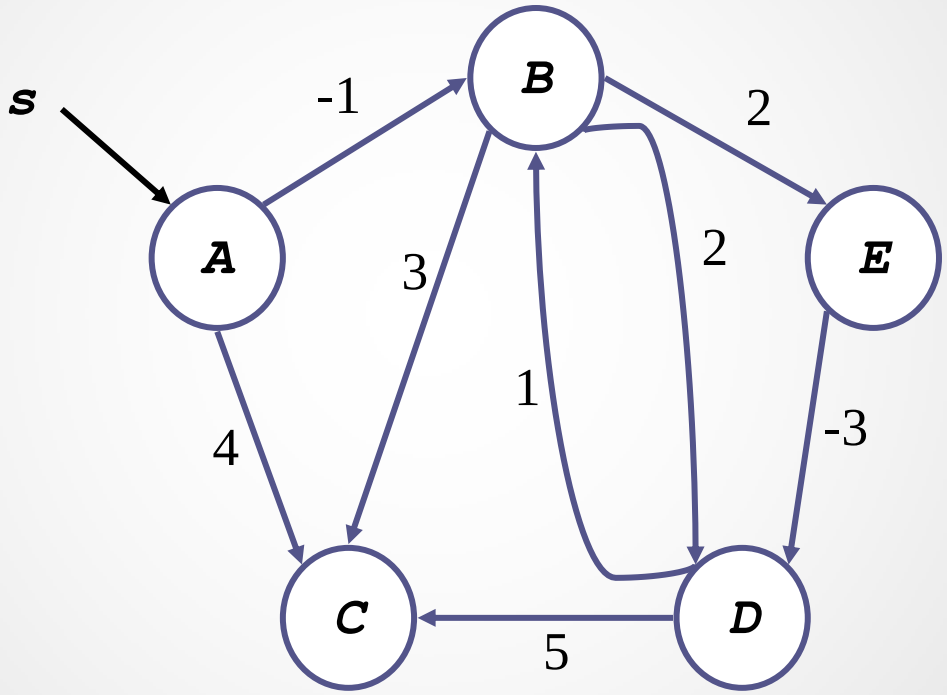
Example



Example



Bellman-Ford Algorithm



Dijkstra's Algorithm For Shortest Paths

- Non-negative edge weight
- Like BFS: If all edge weights are equal, then use BFS, otherwise use this algorithm
- Use Q = min-priority queue keyed on d[v] values

Dijkstra's Algorithm

*Assumes **no negative-weight edges**.*

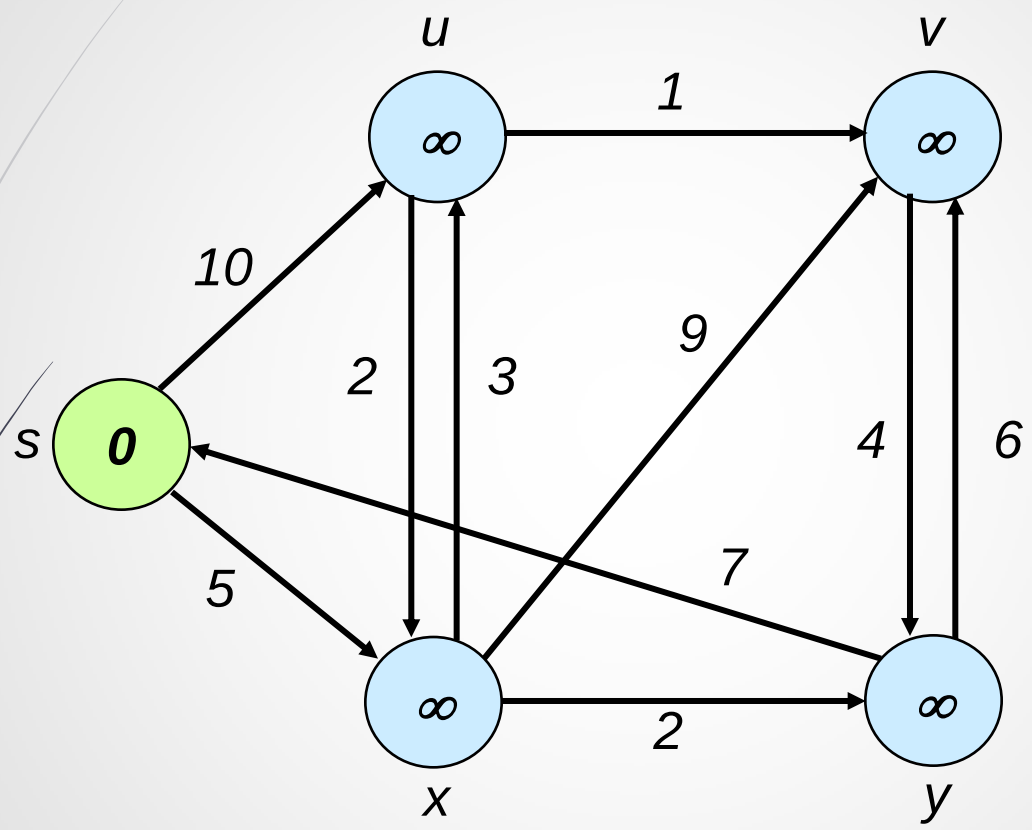
Maintains a set S of vertices whose SP from s has been determined.

Repeatedly selects u in V–S with minimum SP estimate (greedy choice).

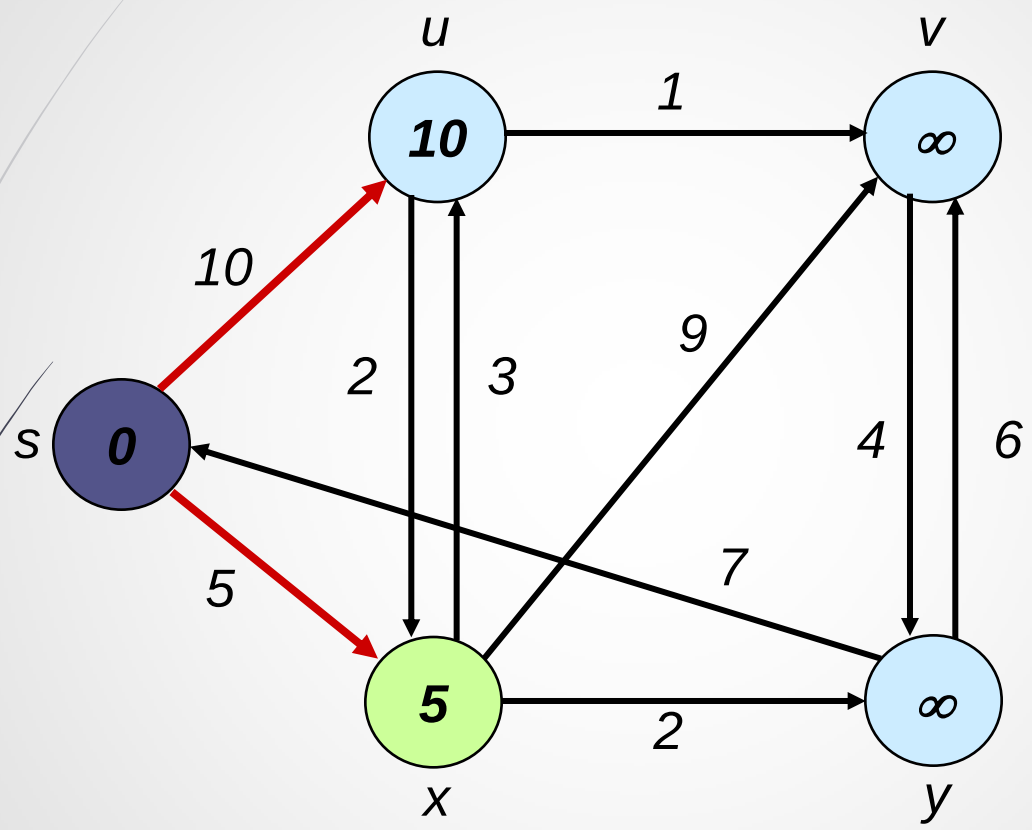
Store V–S in priority queue Q.

```
Initialize(G, s);
S := ∅;
Q := V[G];
while Q ≠ ∅ do
      u := Extract-Min(Q);
      S := S ∪ {u};
      for each v ∈ Adj[u] do
            Relax(u, v, w)
      od
od
```

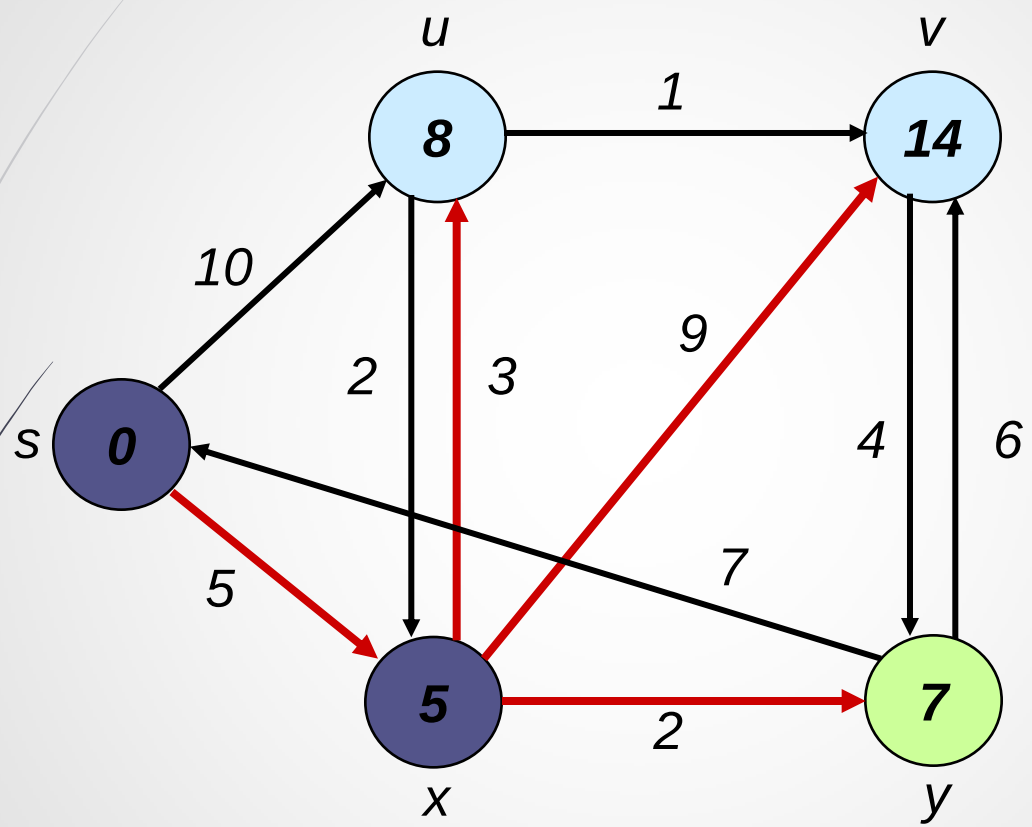
Example



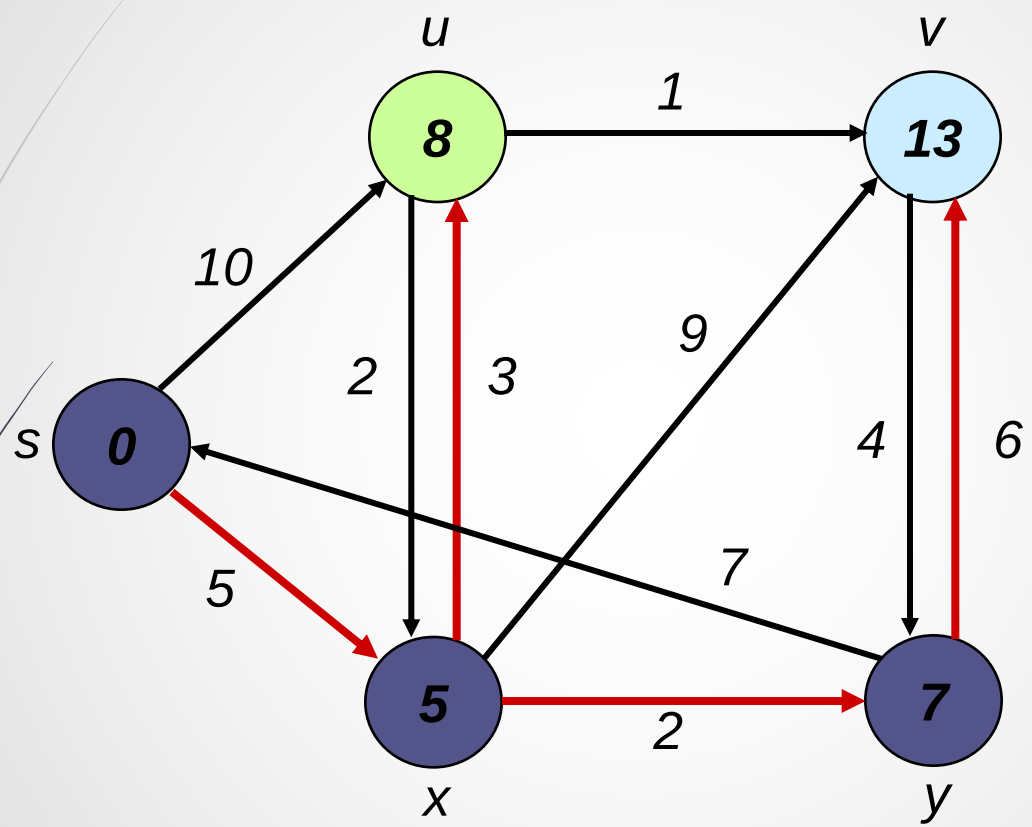
Example



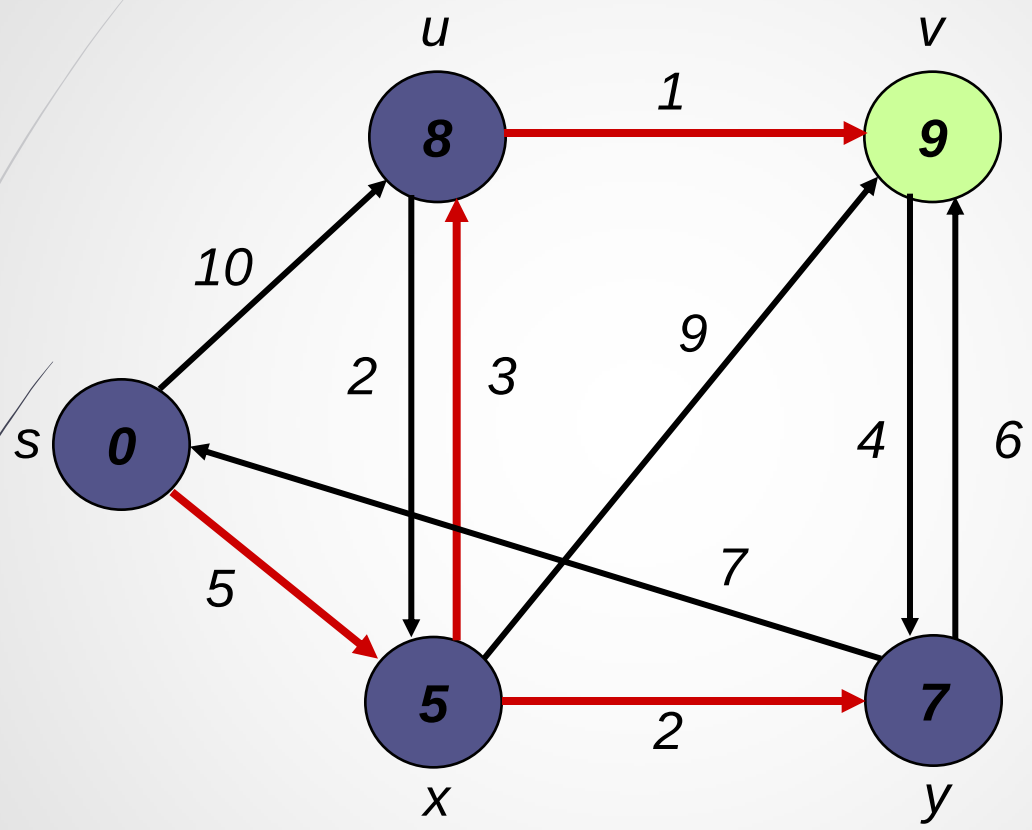
Example



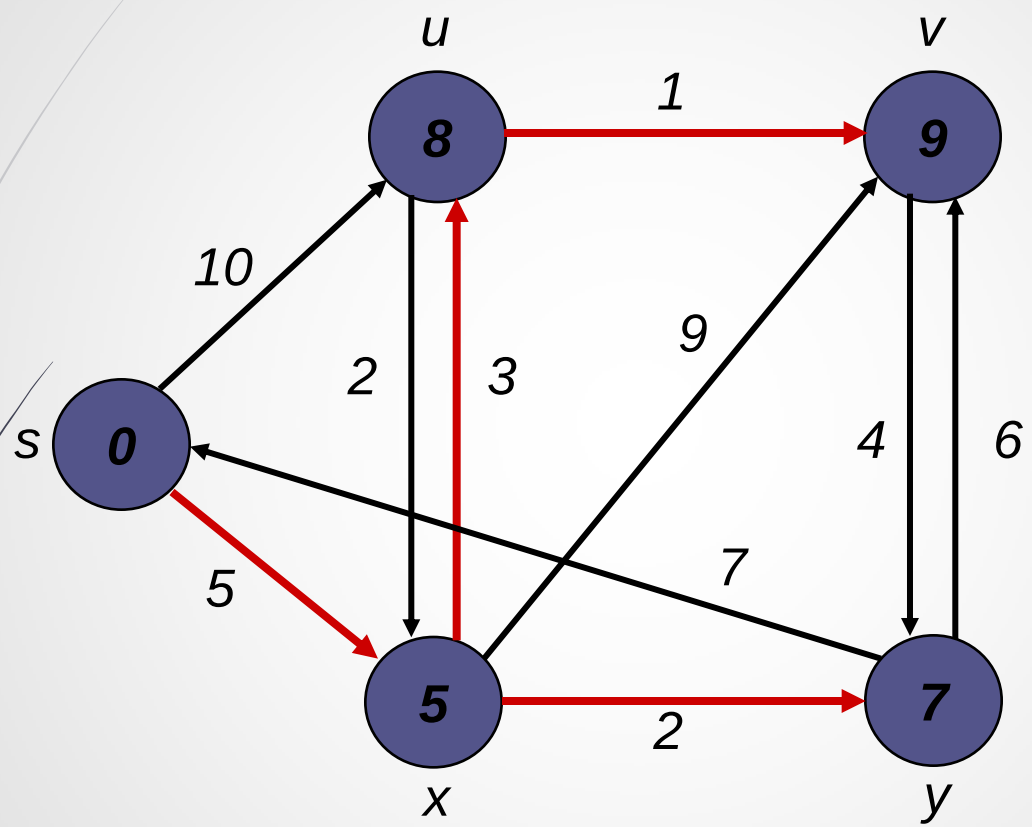
Example



Example

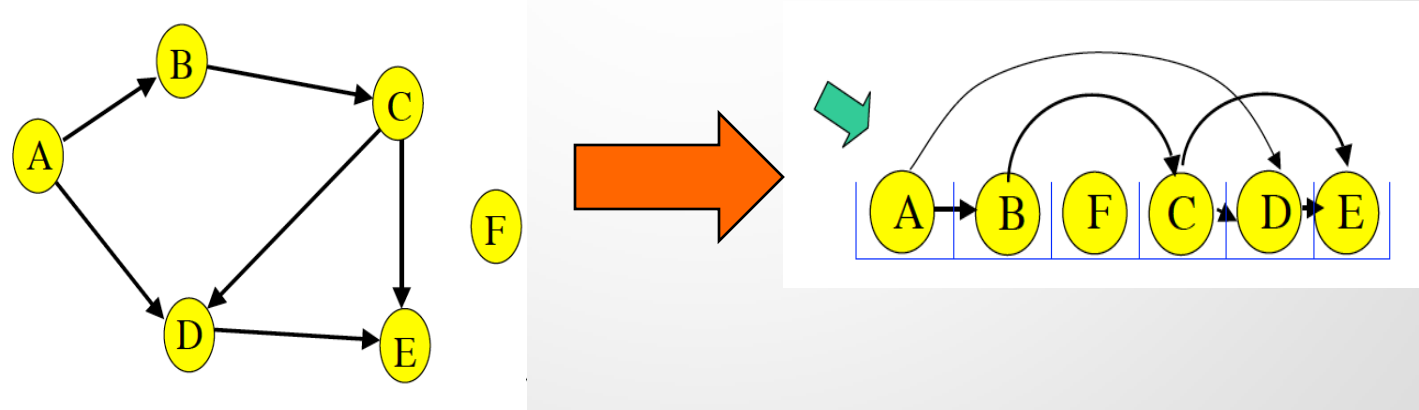


Example



Shortest paths in DAGs

- If there are no cycles ➔ it is called a DAG
- In DAGs, nodes can be sorted in a linear order such that all edges are forward edges
  - Topological sort



Single-Source Shortest Paths in DAGs

- Shortest paths are always *well-defined* in *dags*
  - no cycles => no negative-weight cycles even if there are negative-weight edges
- **Idea:** If we were lucky
  - To process vertices on each shortest path from left to right, we would be done in 1 pass

Single-Source Shortest Paths in DAGs

DAG-SHORTEST PATHS(G, s)

TOPOLOGICALLY-SORT the vertices of G

INIT(G, s)

for each vertex *u* taken in topologically sorted order do

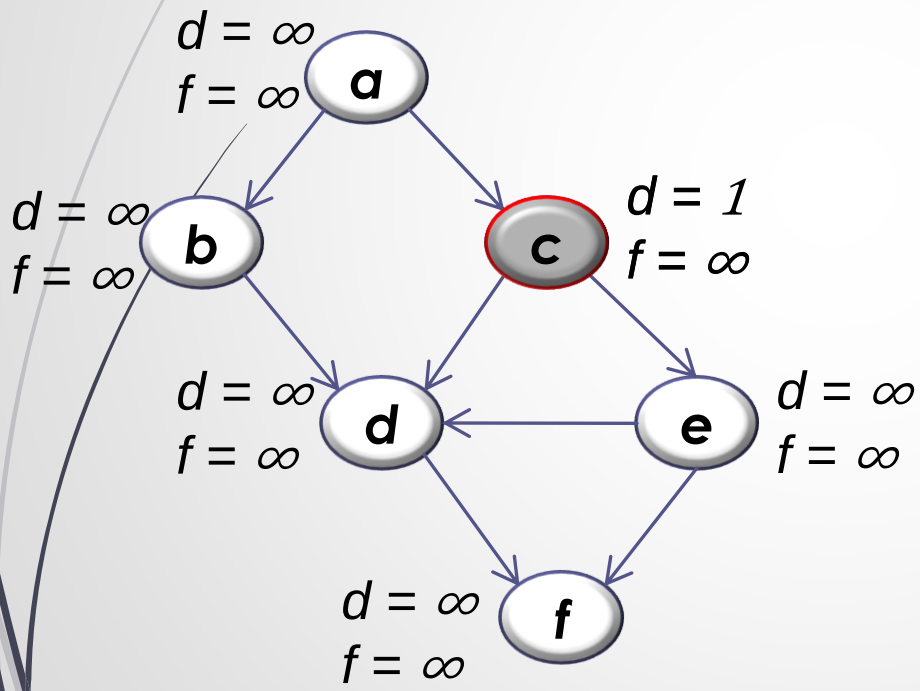
for each *v* in Adj[*u*] do

RELAX(u, v)

Topological sort

1) *Call DFS(**G**) to compute the finishing times **f**[**v**]*

Time = 2

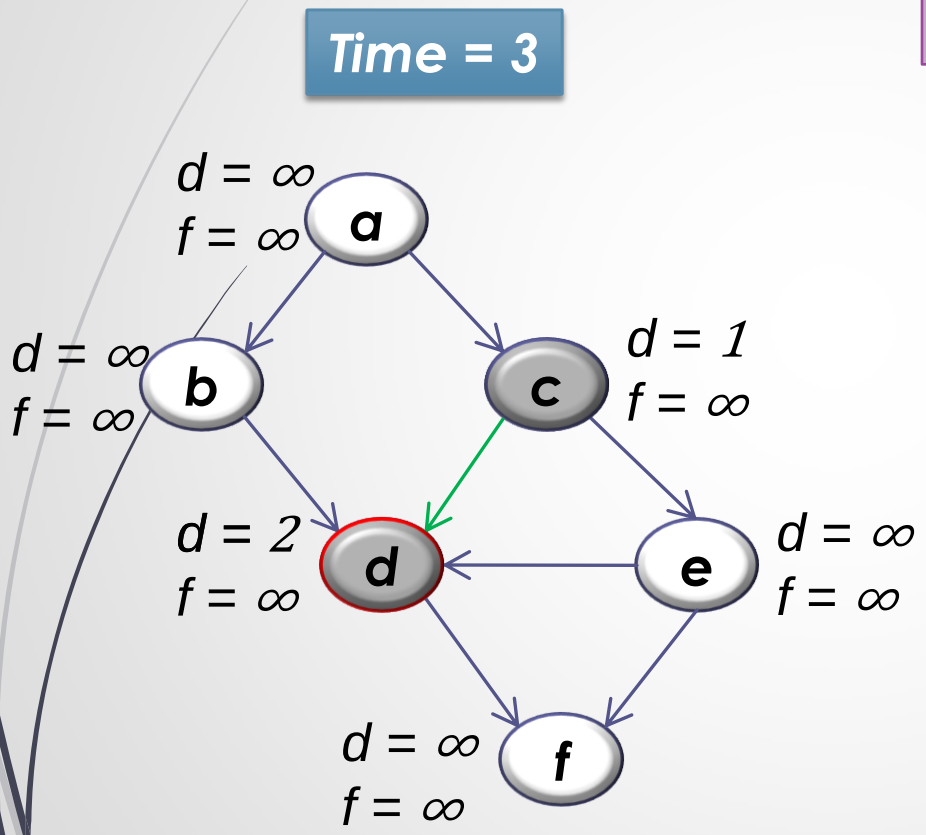


*Let's say we start the DFS from the vertex **c***

*Next we discover the vertex **d***

Topological sort

1) *Call DFS(**G**) to compute the finishing times **f**[**v**]*



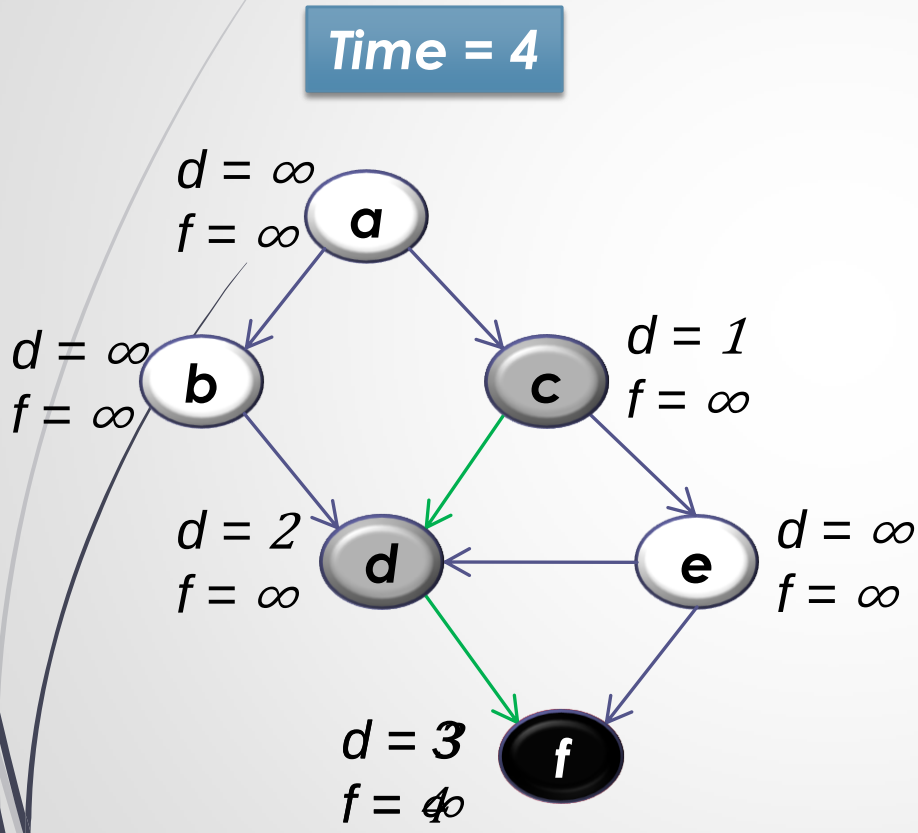
*Let's say we start the DFS from the vertex **c***

*Next we discover the vertex **d***

Topological sort

1) *Call DFS(**G**) to compute the finishing times **f**[**v**]*

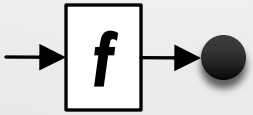
2) *as each vertex is finished, insert it onto the **front** of a linked list*



*Next we discover vertex **d***

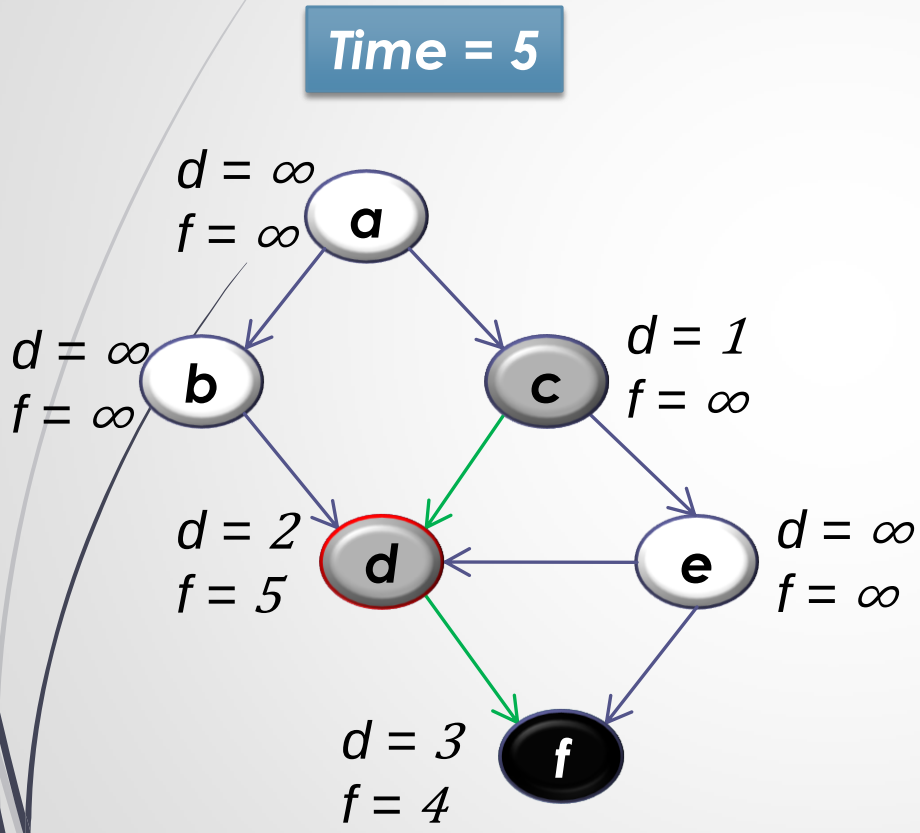
*Next we discover vertex **f***

f** is done, move back to **d



Topological sort

1) *Call DFS(**G**) to compute the finishing times **f**[**v**]*



*Let's say we start DFS from the vertex **c***

*Next we discover vertex **d***

*Next we discover the vertex **f***

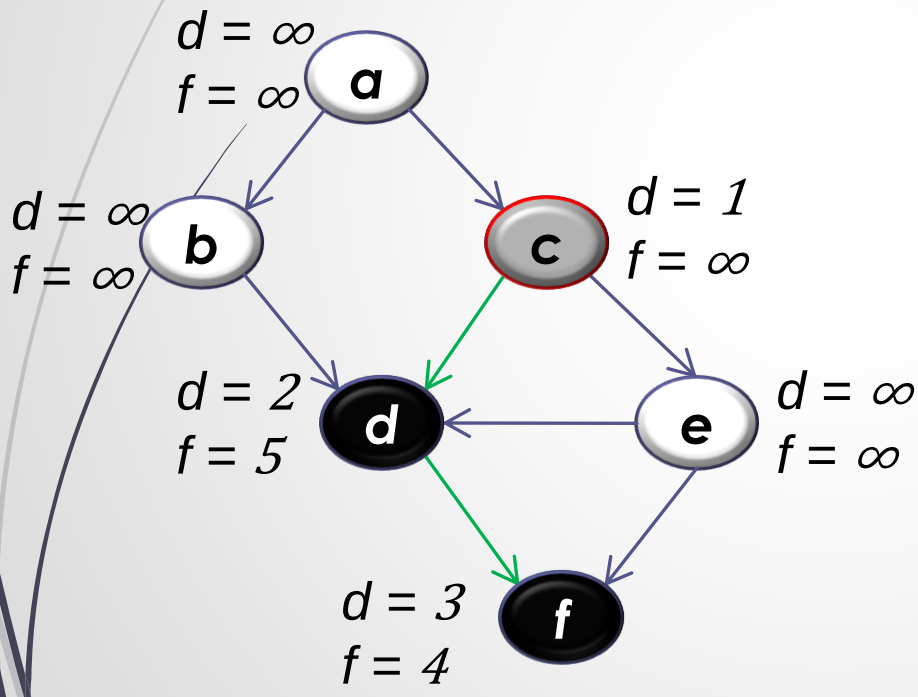
f** is done, move back to **d

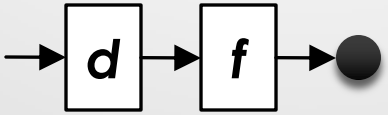
d** is done, move back to **c

Topological sort

1) *Call DFS(**G**) to compute the finishing times **f**[**v**]*

Time = 6





*Let's say we start the DFS from the vertex **c***

*Next we discover vertex **d***

*Next we discover vertex **f***

f** is done, move back to **d

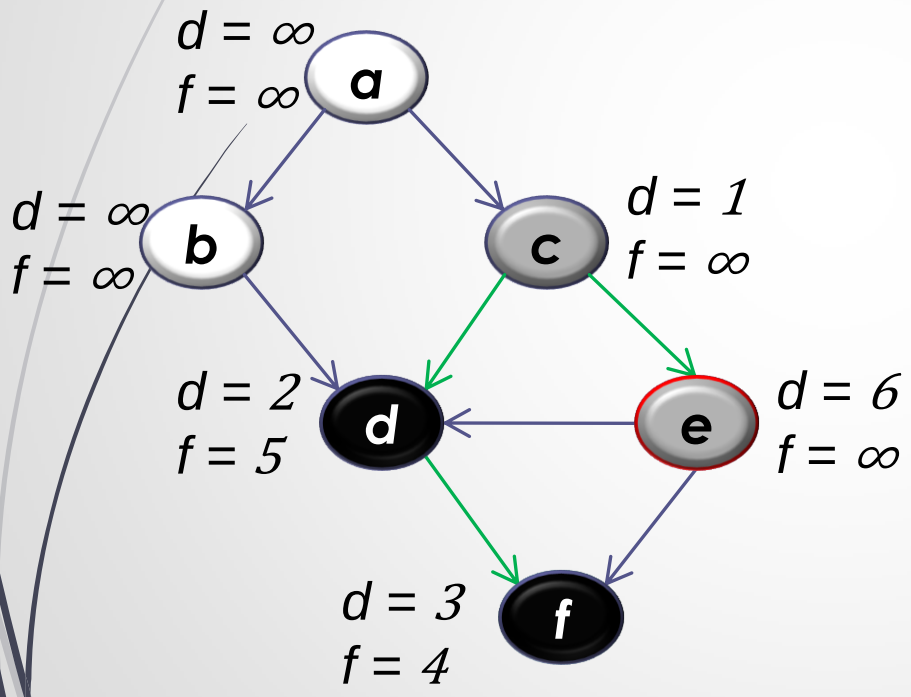
d** is done, move back to **c

*Next we discover vertex **e***

Topological sort

1) *Call DFS(**G**) to compute the finishing times **f**[**v**]*

Time = 7



*Let's say we start the DFS from the vertex **c***

*Next we discover vertex **d***

*Both edges from **e** are **cross edges***

f** is done, move back to **d

d** is done, move back to **c

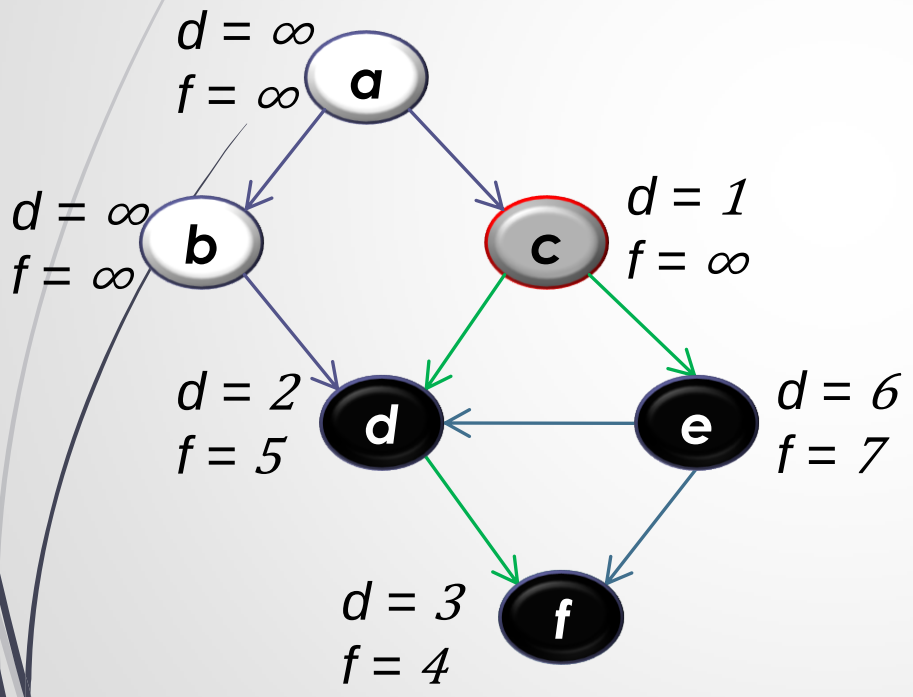
*Next we discover vertex **e***

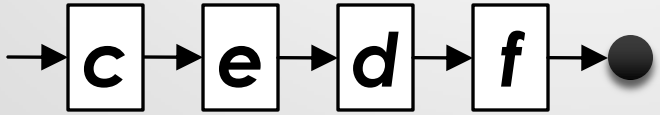
e** is done, move back to **c

Topological sort

1) *Call DFS(**G**) to compute the finishing times **f**[**v**]*

Time = 8





*Let's say we start the DFS from the vertex **c***

*Next we discover the vertex **d***

*Just a note: If there was (**c**,**f**) edge in the graph, it would be classified as a **forward edge** (in this particular DFS run)*

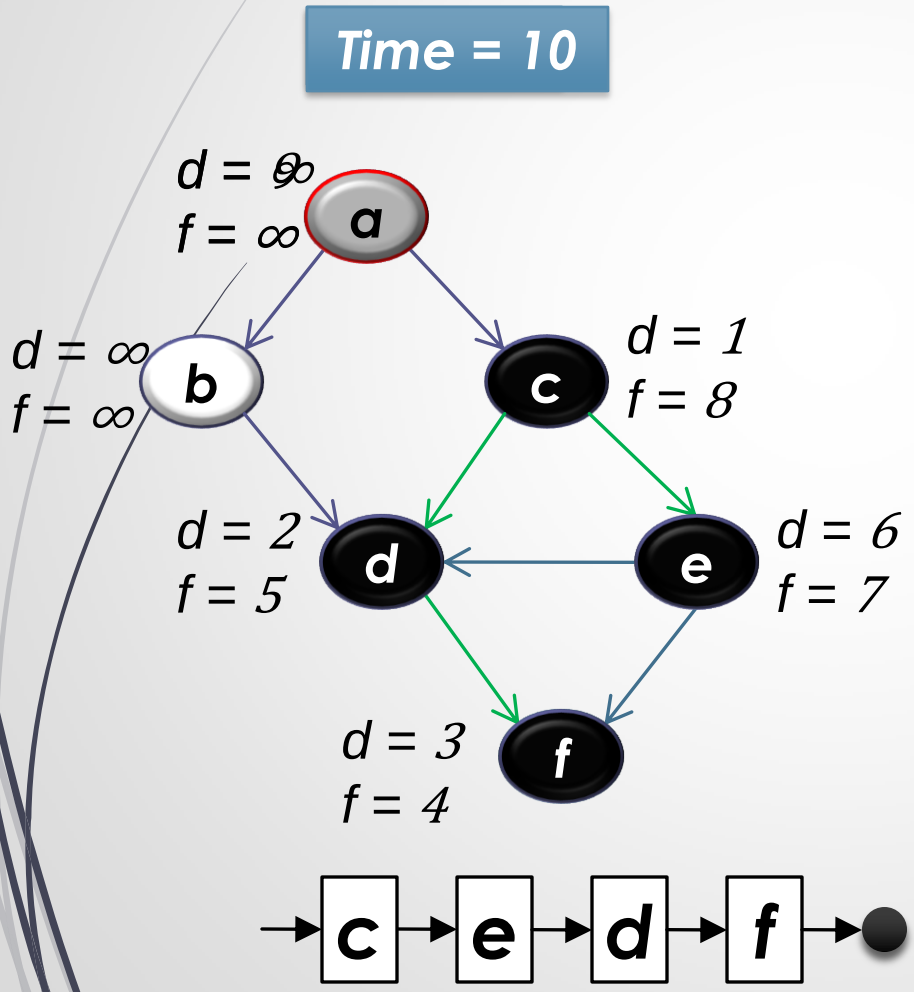
*Next we discover the vertex **e***

e** is done, move back to **c

***c** is done as well*

Topological sort

1) *Call DFS(**G**) to compute the finishing times **f**[**v**]*



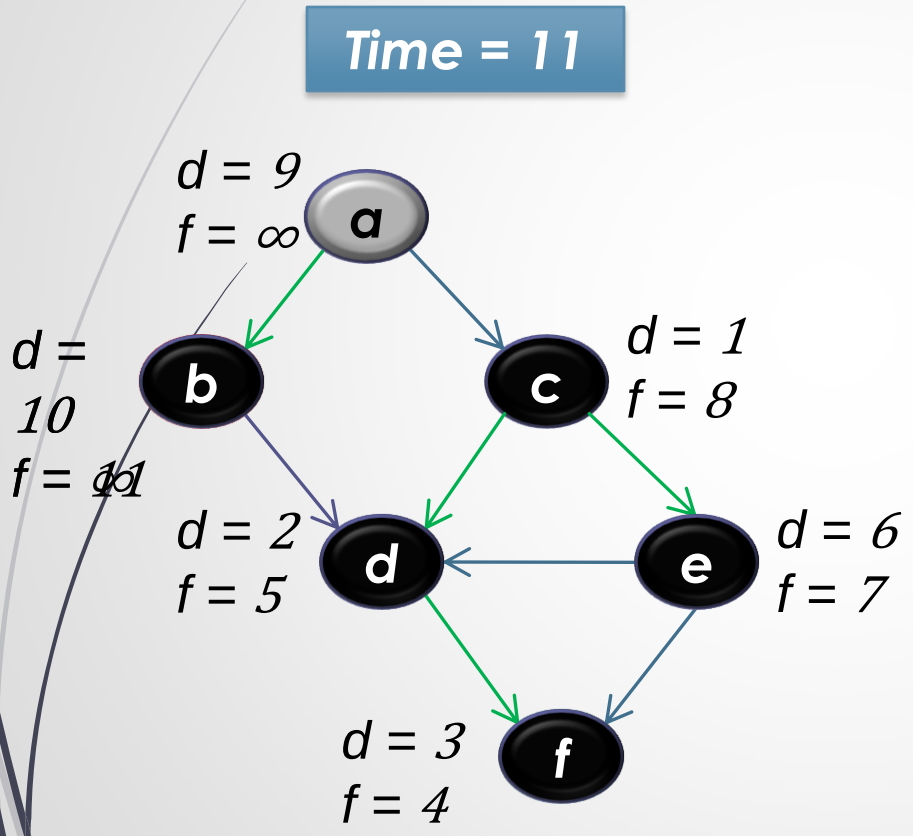
*Let's now call DFS visit from the vertex **a***

*Next we discover vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge*

*Next we discover the vertex **b***

Topological sort

1) *Call DFS(**G**) to compute the finishing times **f**[**v**]*

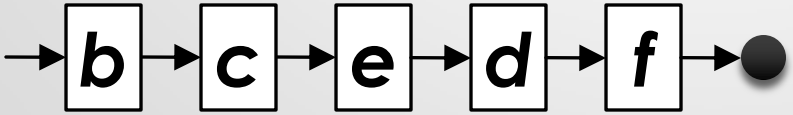


*Let's now call DFS visit from the vertex **a***

*Next we discover vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge*

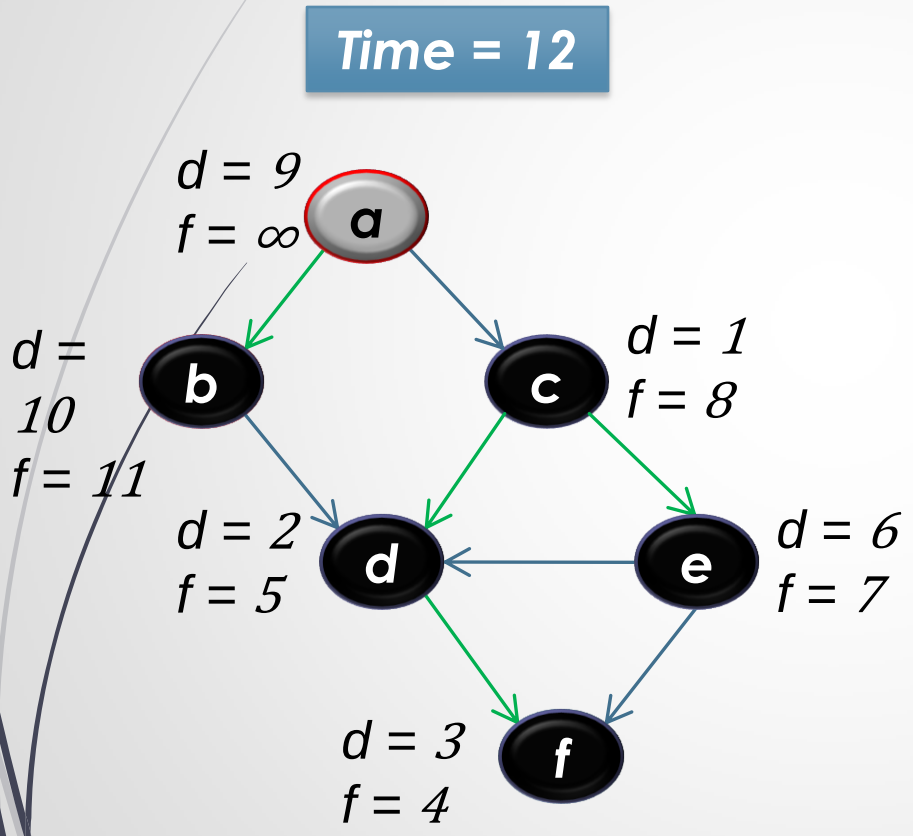
*Next we discover the vertex **b***

b** is done as (**b**,**d**) is a cross edge => now move back to **c



Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

Next we discover the vertex **b**

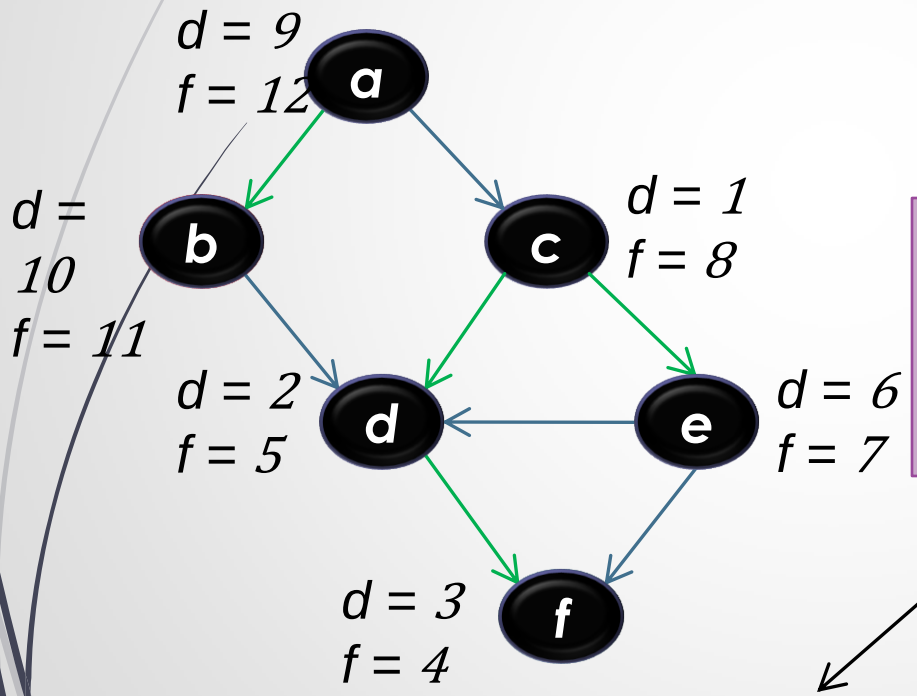
b is done as (**b**,**d**) is a cross edge => now move back to **c**

a is done as well

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 13



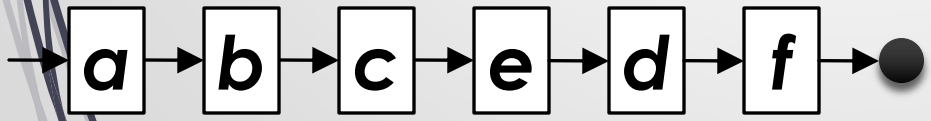
Let's now call DFS visit from the vertex **a**

WE HAVE THE RESULT!

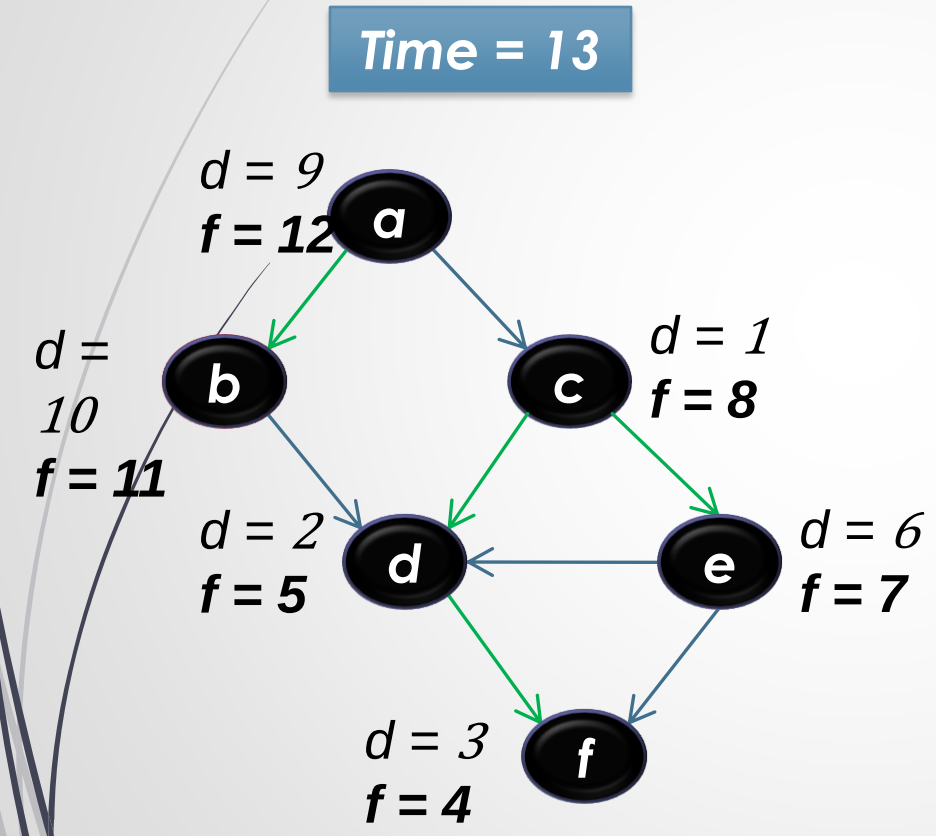
3) return the linked list of vertices

b is done as (**b**,**d**) is a cross edge => now move back to **c**

a is done as well

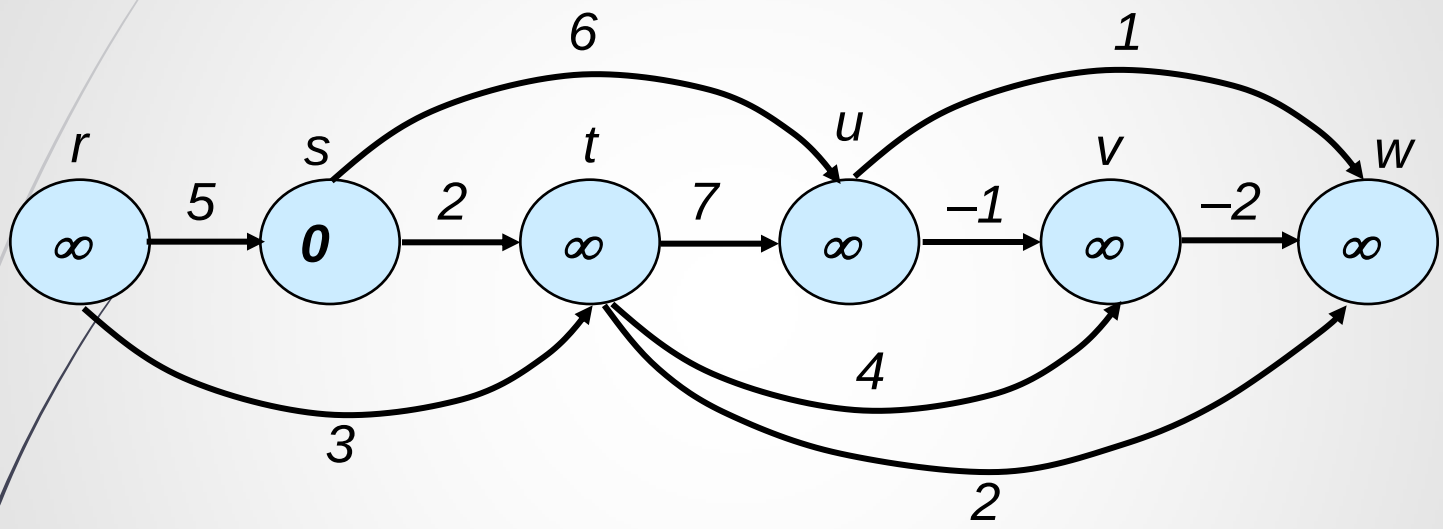


Topological sort

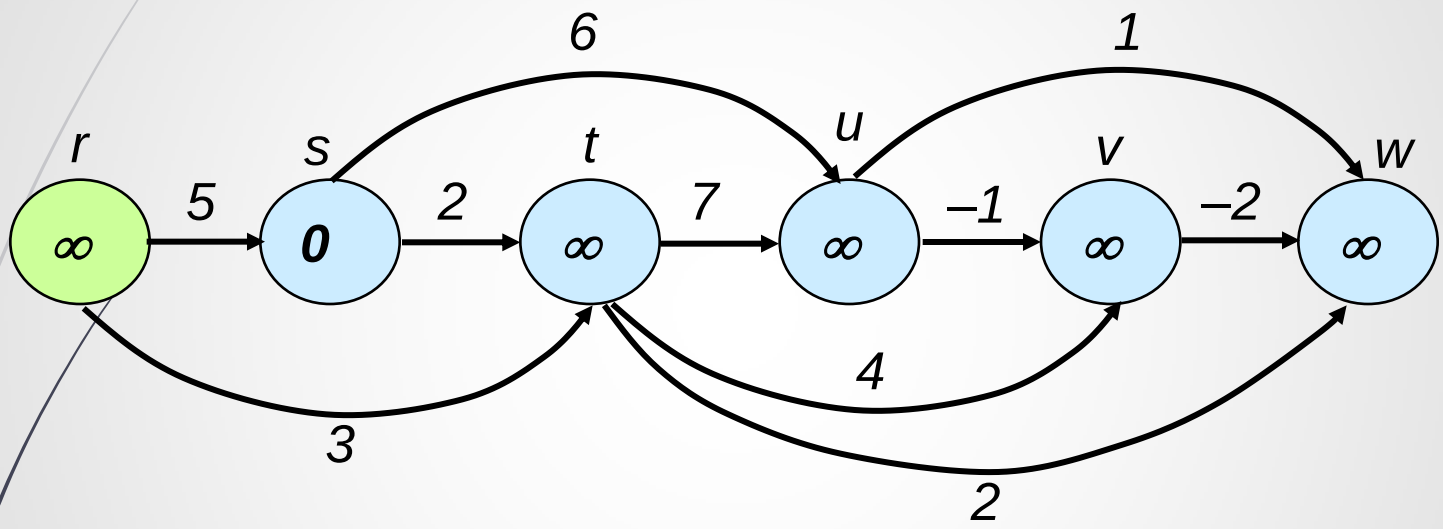


*The linked list is sorted in **decreasing** order of finishing times **f**[]*

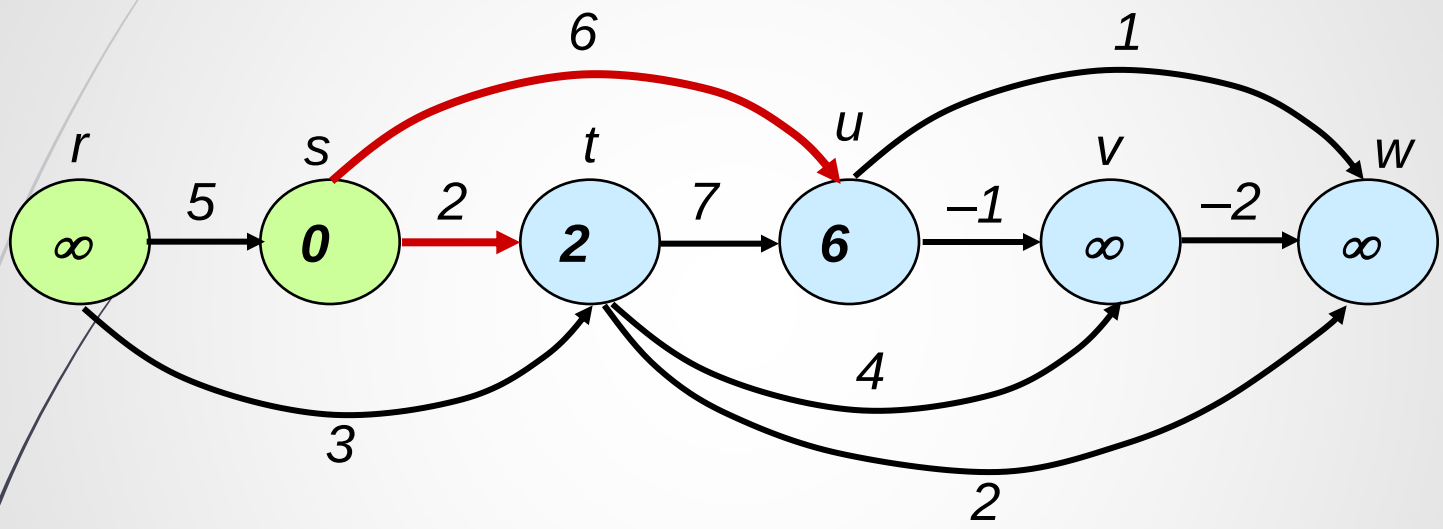
Example



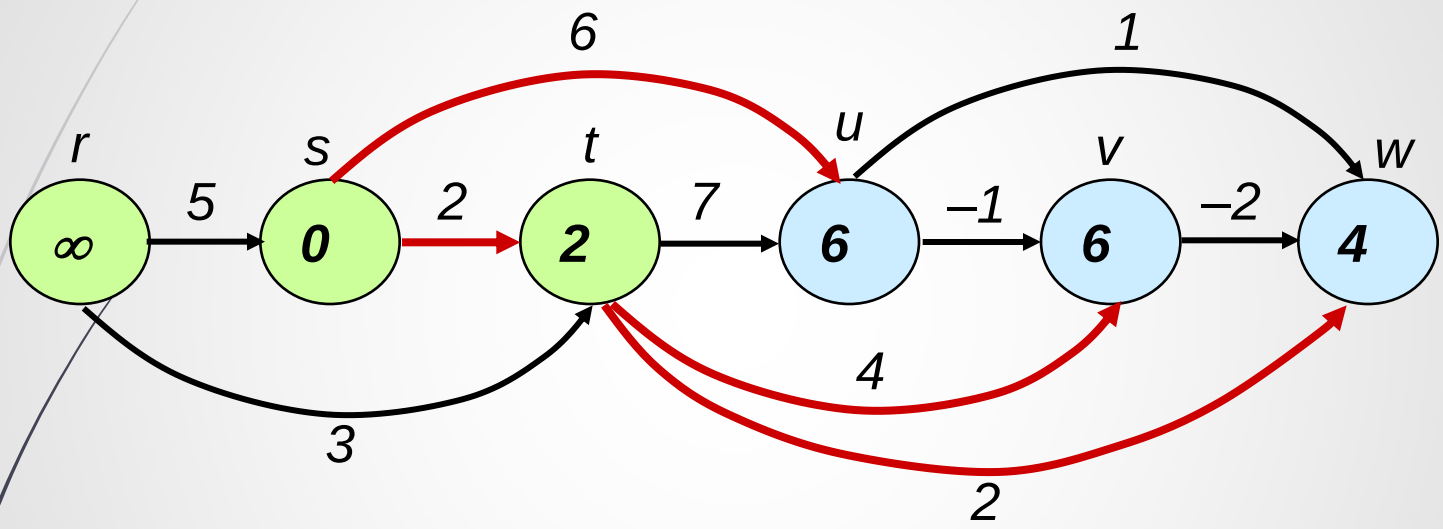
Example



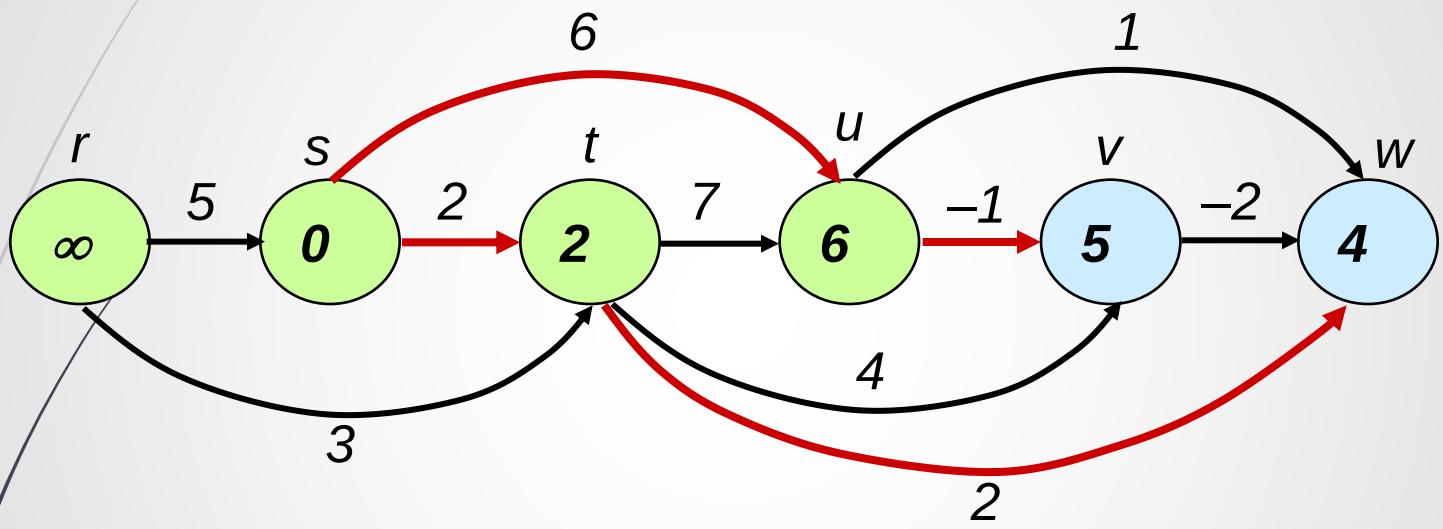
Example



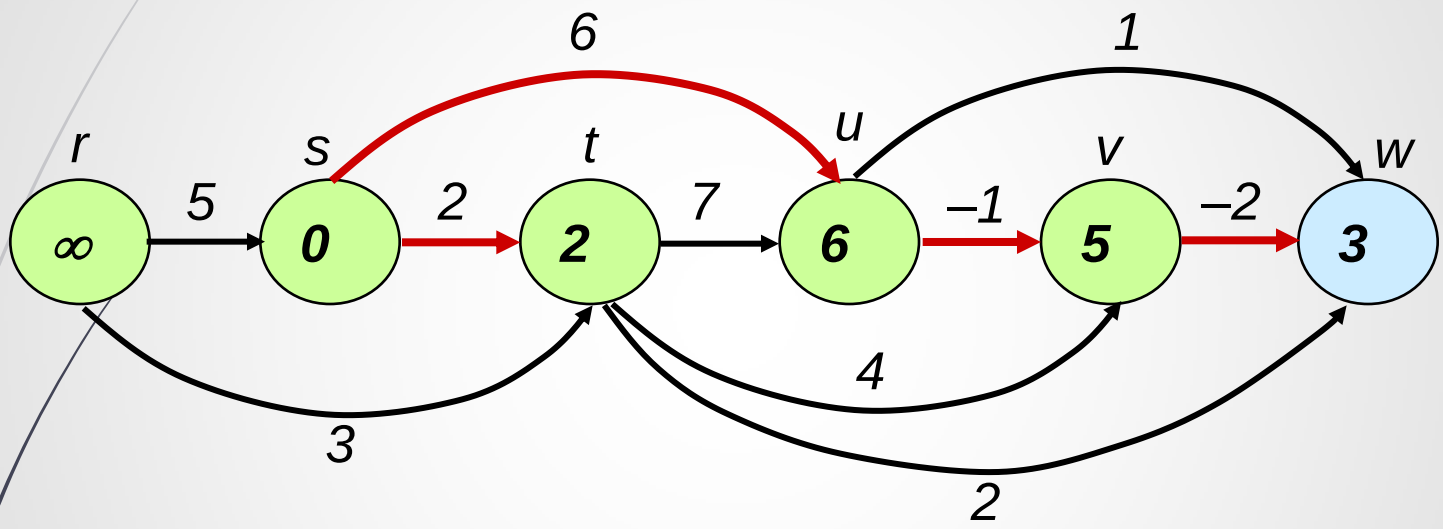
Example



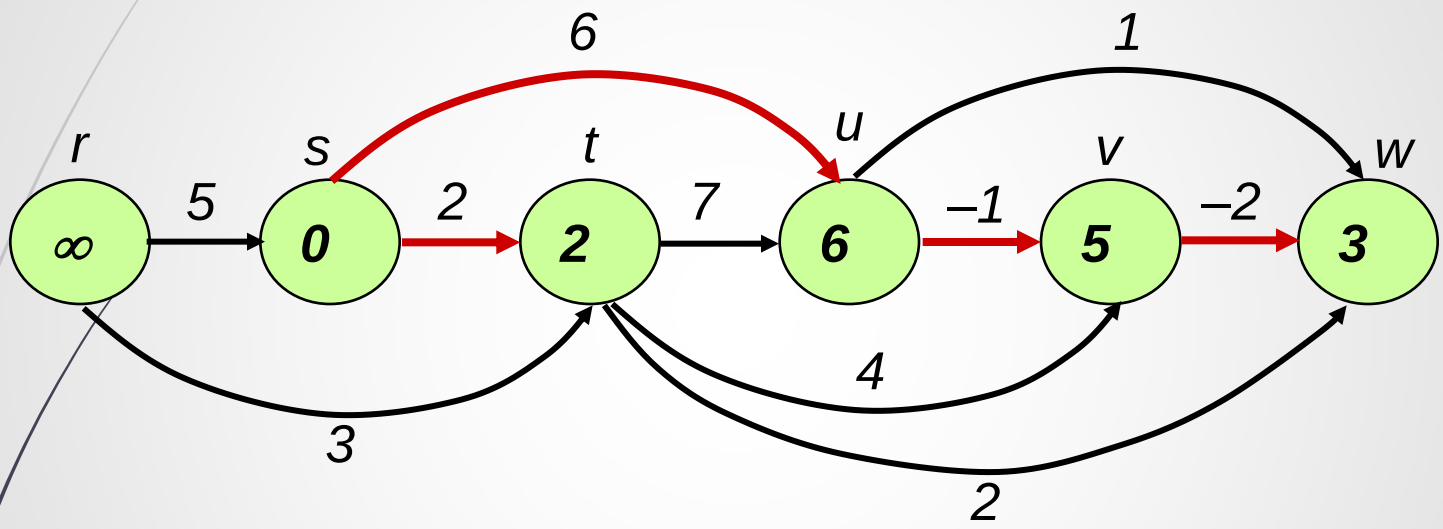
Example



Example



Example



END