Topological Sort

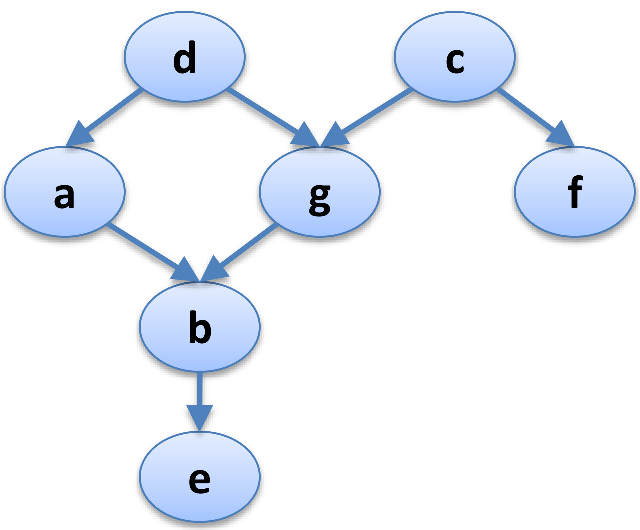
(an application of DFS)

Topological sort

- We have a **set of tasks** and a **set of dependencies (precedence constraints)** of form "task A must be done before task B"
- **Topological sort**: An ordering of the tasks that conforms with the given dependencies
- **Goal**: Find a topological sort of the tasks or decide that there is no such ordering

Examples

- **Scheduling**: When scheduling *task graphs* in distributed systems, usually we first need to <u>sort the tasks topologically</u>
  ...and then assign them to resources
- Or during compilation to order modules/libraries



Topological sort more formally

- Suppose that in a **directed** graph **G = (V, E)** vertices **V** represent tasks, and each edge (**u**, **v**)∈**E** means that task **u** must be done before task **v**

- What is an ordering of vertices 1, ..., |**V**| such that for every edge (**u**, **v**), **u** appears before **v** in the ordering?

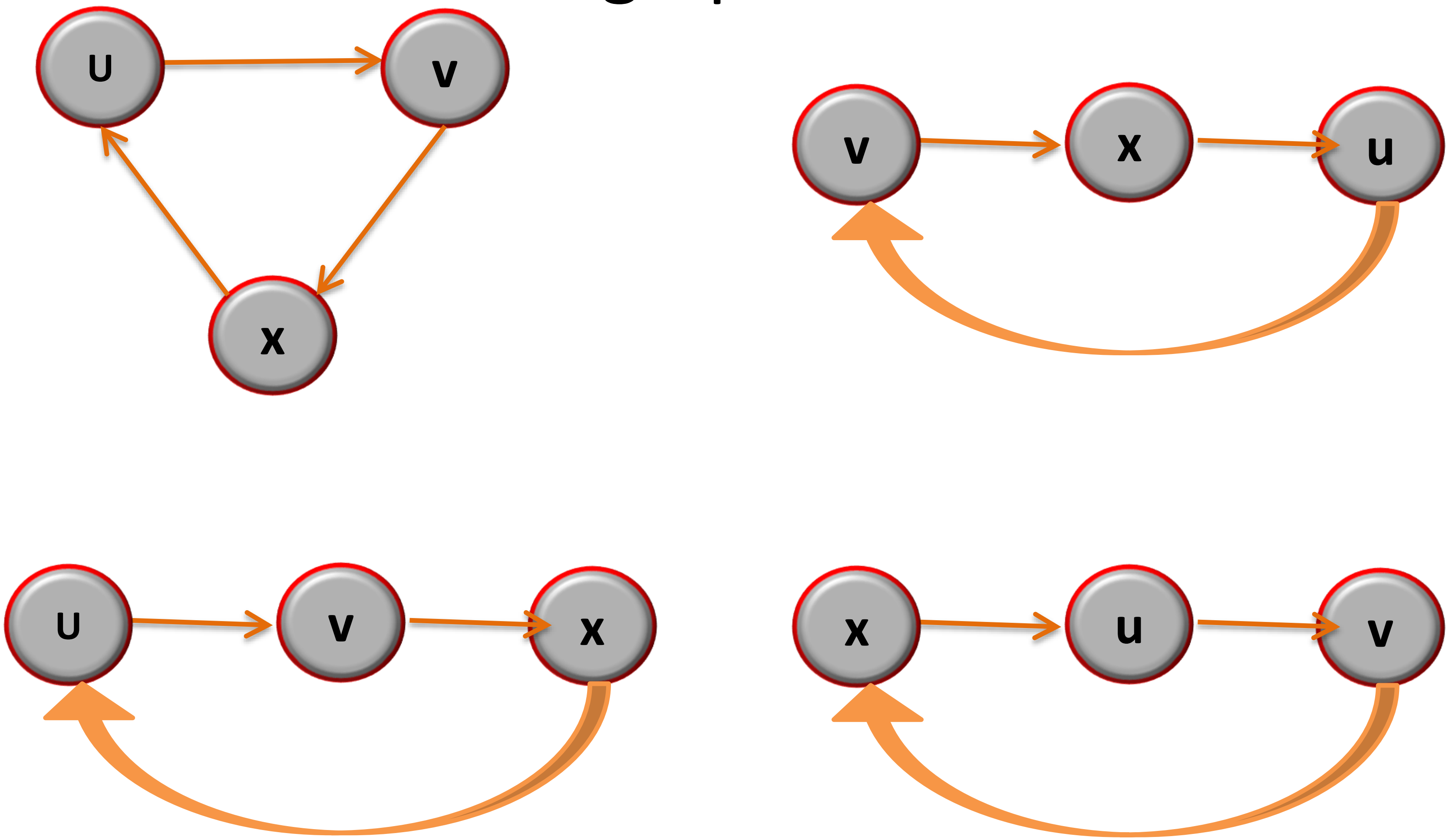
- Such an ordering is called a **topological sort of G**
- Note: there can be multiple topological sorts of G

Topological sort more formally

- Is it possible to execute all the tasks in **G** in an order that respects all the precedence requirements given by the graph edges?
- The answer is "**yes**" *if and only if* the directed graph **G** has **no cycle**!

  (otherwise we have a **deadlock**)
- Such a **G** is called a Directed Acyclic Graph, or just a **DAG**

Topological sort is only possible for Acyclic graph



Cycle Detection in Directed graph using DFS

Cycle Detection in Directed graph using DFS

Cycle Detection in Directed graph using DFS

- We will assign every vertex a color and will use 3 colors- white, gray and black.
- **White Color:** Vertices which are not processed will be assigned white colors. So at the beginning all the vertices will be white.
- **Gray Color:** Vertices will are currently being processed. If DFS is started from a particular vertex will be in gray color till DFS is not completed (means all the descendants in DFS are not processed.)
- **Black Color:** Vertices for which the DFS is completed, means all the processed vertices will be assigned black color.
- **Cycle Detection:** During DFS if we encounter a vertex which is already in Gray color (means this vertex already in processing and in Gray color in the current DFS) then we have detected a Cycle and edge from current vertex to gray vertex will a back edge.

Edge classification by DFS

Edge (**u**,**v**) of **G** is classified as a:

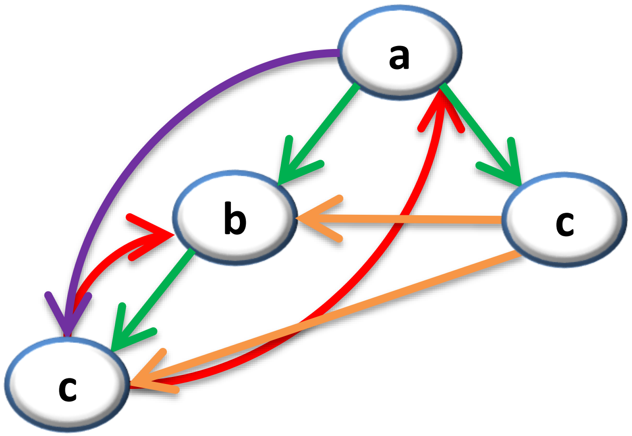
(1) **Tree** edge iff **u** discovers **v** during the DFS: **P**[**v**] = **u**

If (**u**,**v**) is NOT a tree edge then it is a:

- (2) **Forward** edge iff **u** is an ancestor of **v** in the DFS tree
- (3) **Back** edge iff **u** is a descendant of **v** in the DFS tree
- (4) **Cross** edge iff **u** is neither an ancestor nor a descendant of **v**

Edge classification by DFS

Tree edges
Forward edges
Back edges
Cross edges

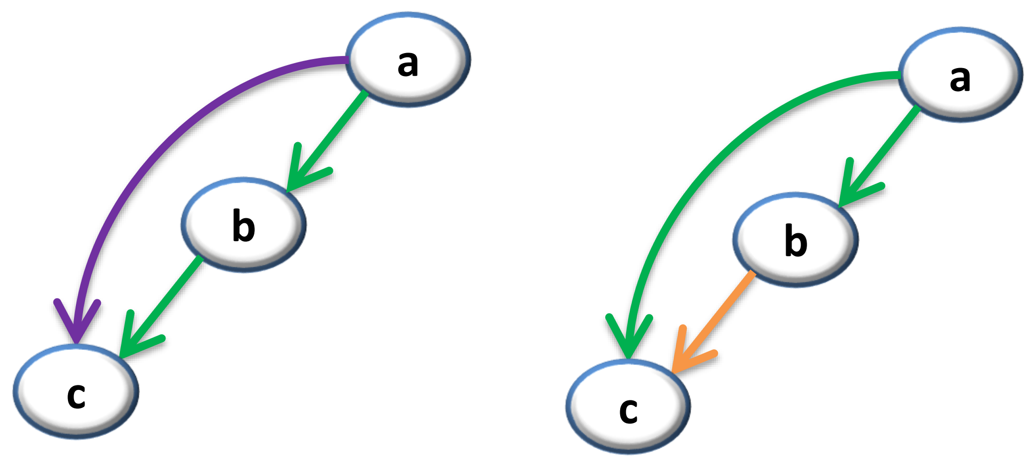


The edge classification depends on the particular DFS tree!

Edge classification by DFS

Tree edges
Forward edges
Back edges
Cross edges

Both are valid

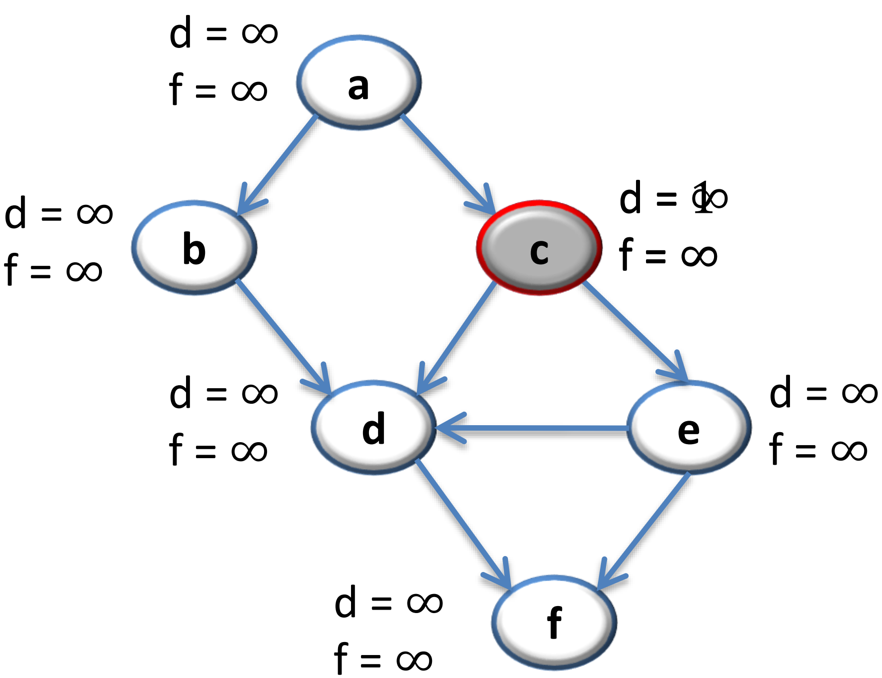


The edge classification depends on the particular DFS tree!

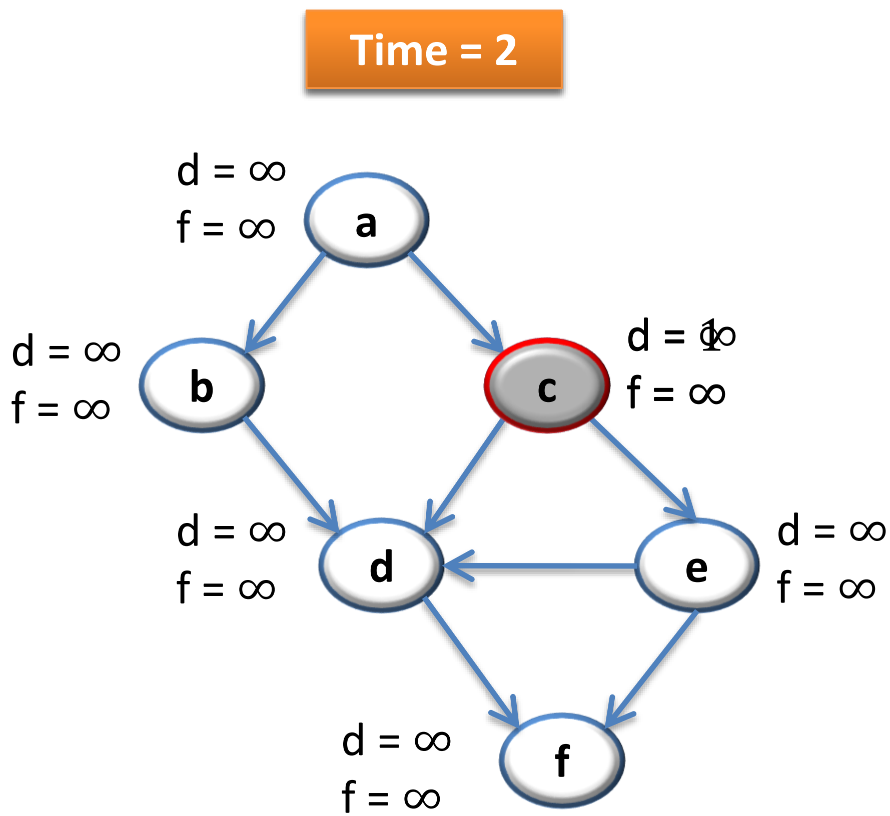
DAGs and back edges

- Can there be a **back** edge in a DFS on a DAG?
- NO! Back edges close a cycle!
- A graph **G** is a DAG <=> there is no back edge classified by DFS(**G**)

Topological sort using DFS



Topological sort



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

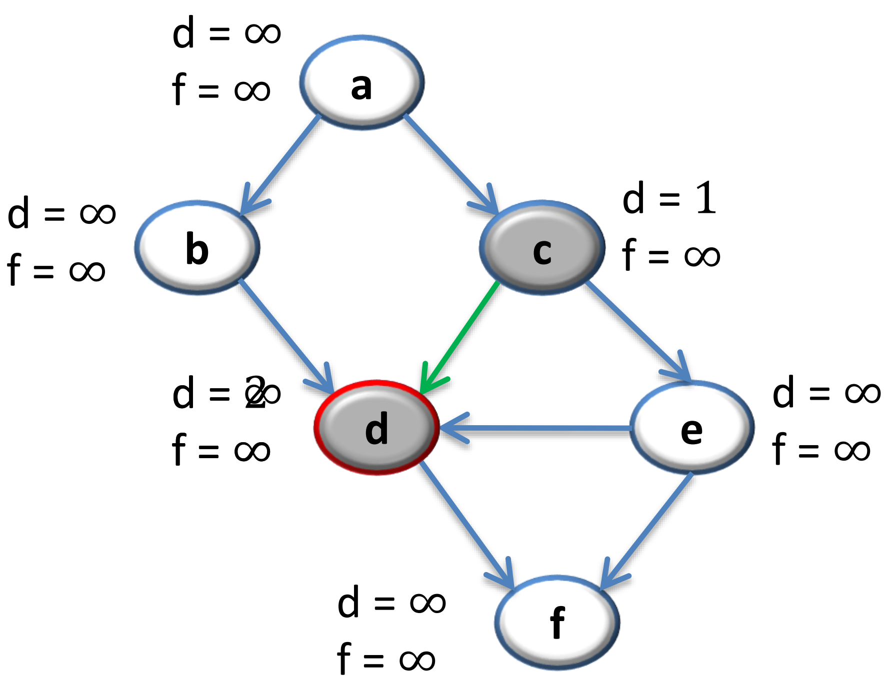
Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Topological sort

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

Time = 3

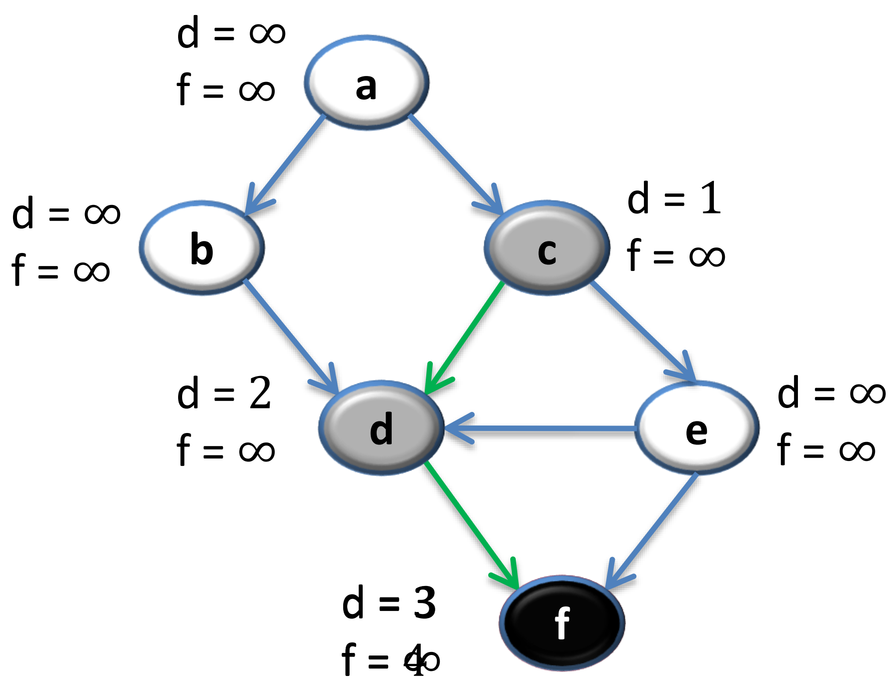


Let’s say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Topological sort

Time = 4

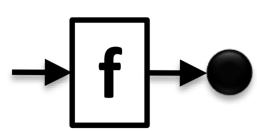


1) Call DFS(**G**) to compute the finishing times **f**[**v**]

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

Next we discover the vertex **f**

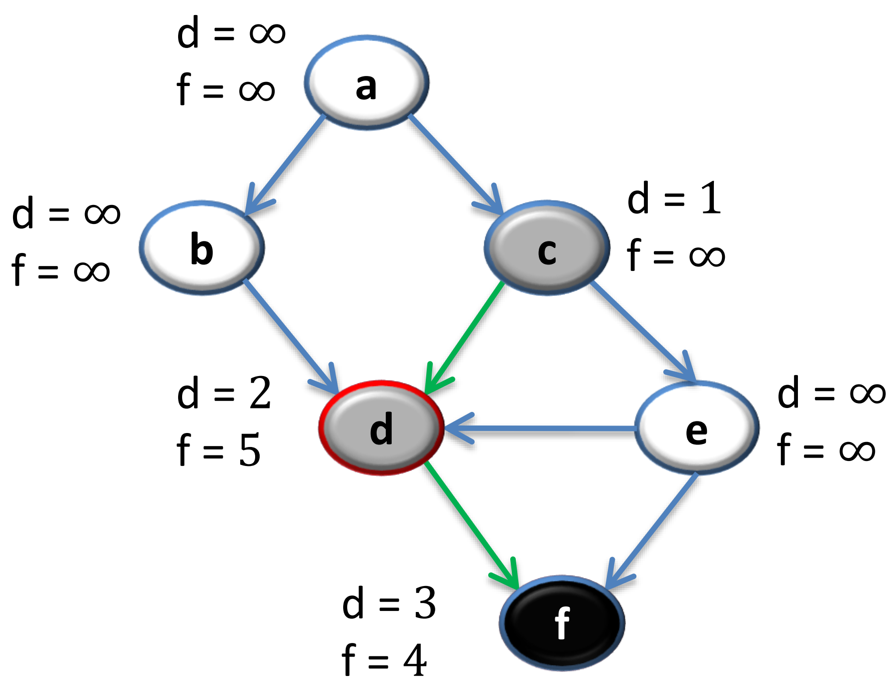
f is done, move back to **d**



Topological sort

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

Time = 5



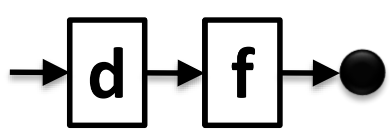
Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Next we discover the vertex **f**

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

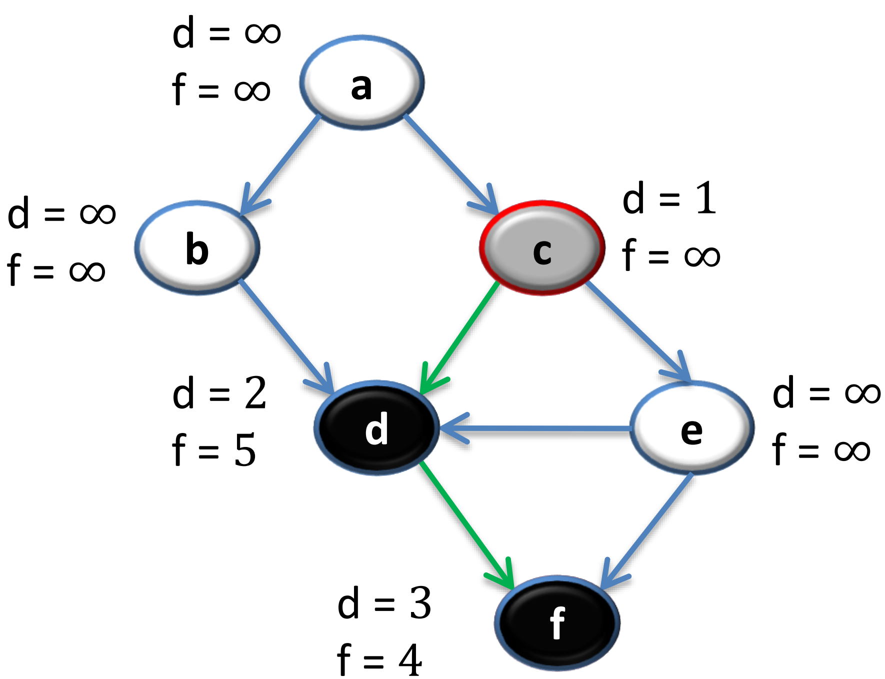
d is done, move back to **c**



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**

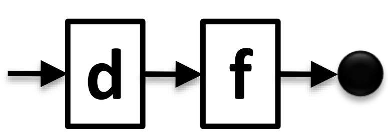
Next we discover the vertex **d**

Next we discover the vertex **f**

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

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

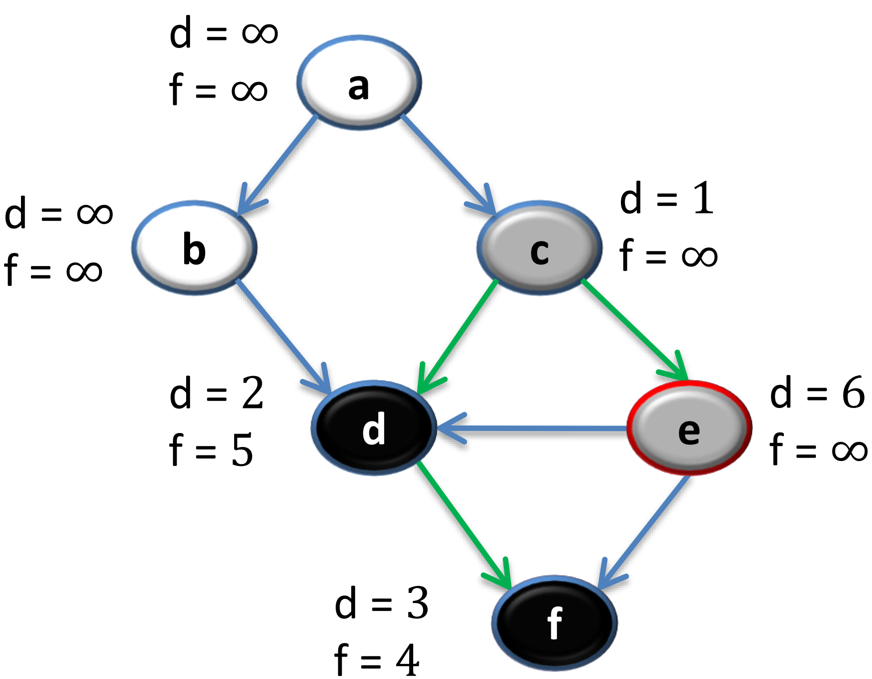
Next we discover the vertex **e**

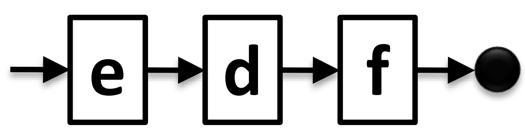


Topological sort

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 the vertex **d**

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

Next we discover the vertex **e**

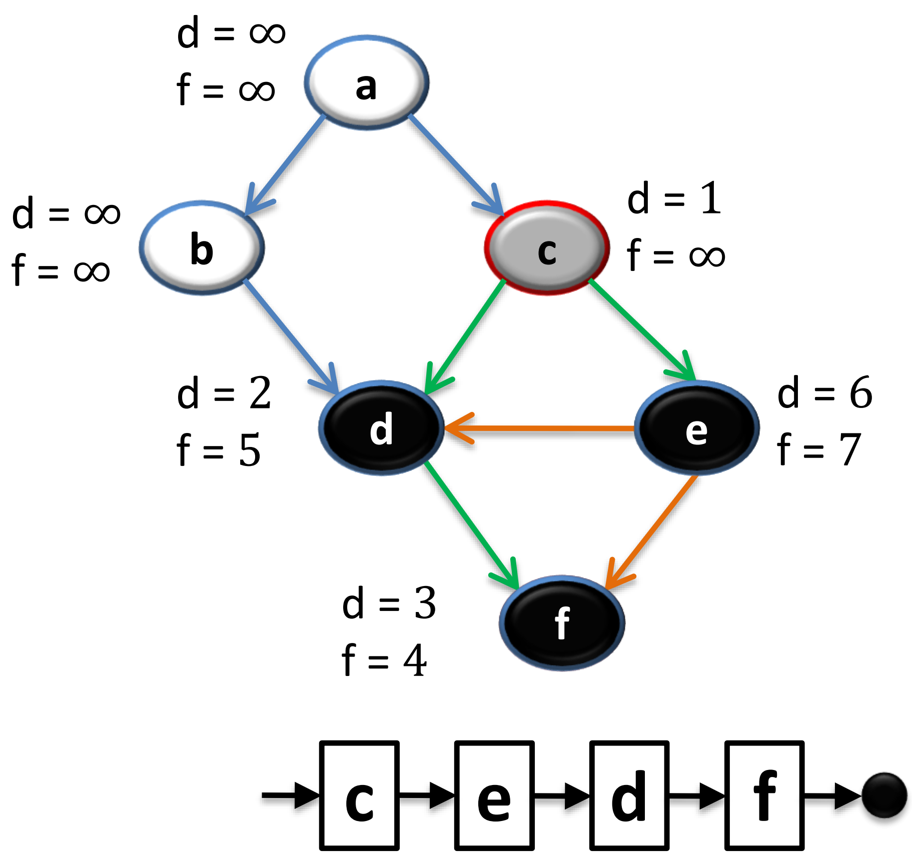
e is done, move back to **c**

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

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**

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

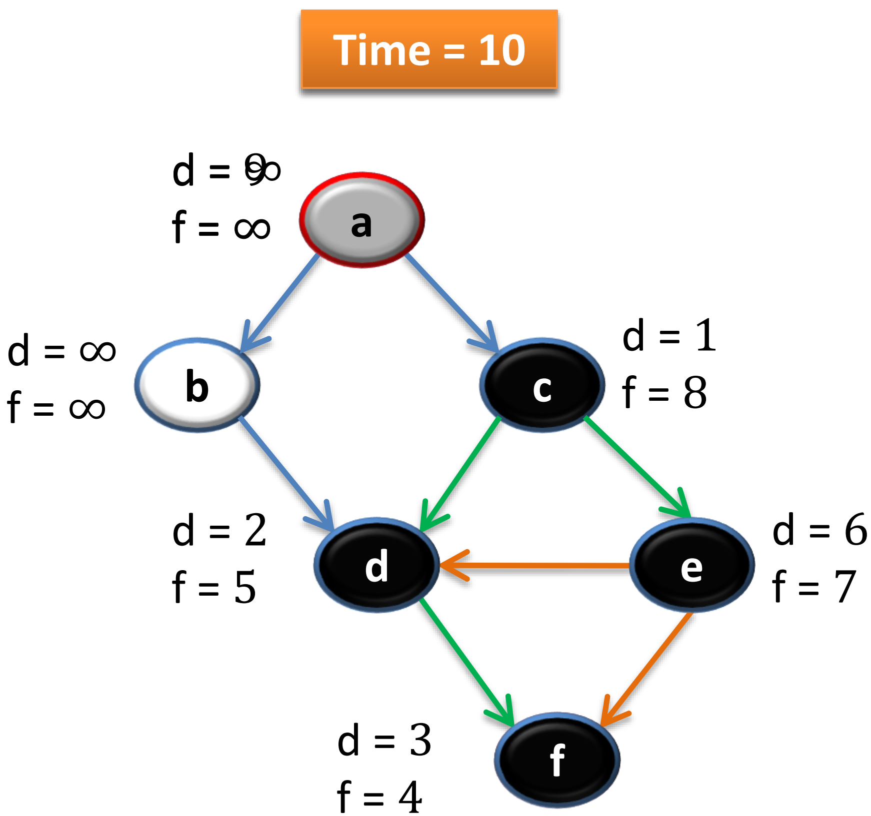
d is done, move back to **c**

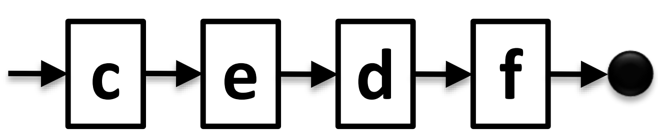
Next we discover the vertex **e**

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**]

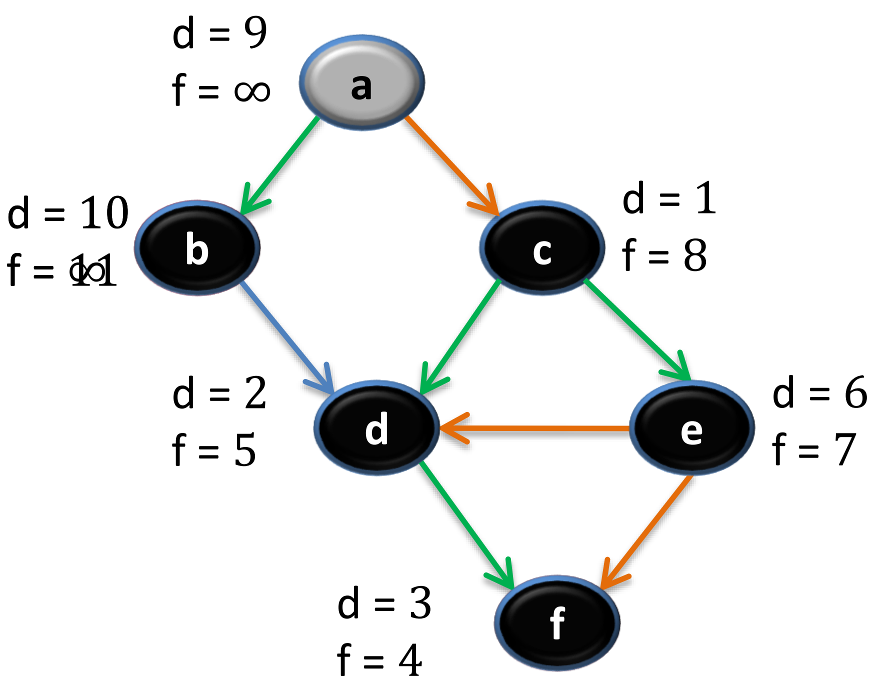
Let's now call DFS visit from the vertex **a**

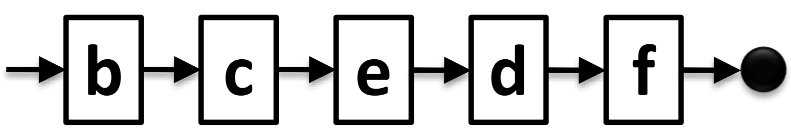
Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

Next we discover the vertex **b**

Topological sort

Time = 11





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

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

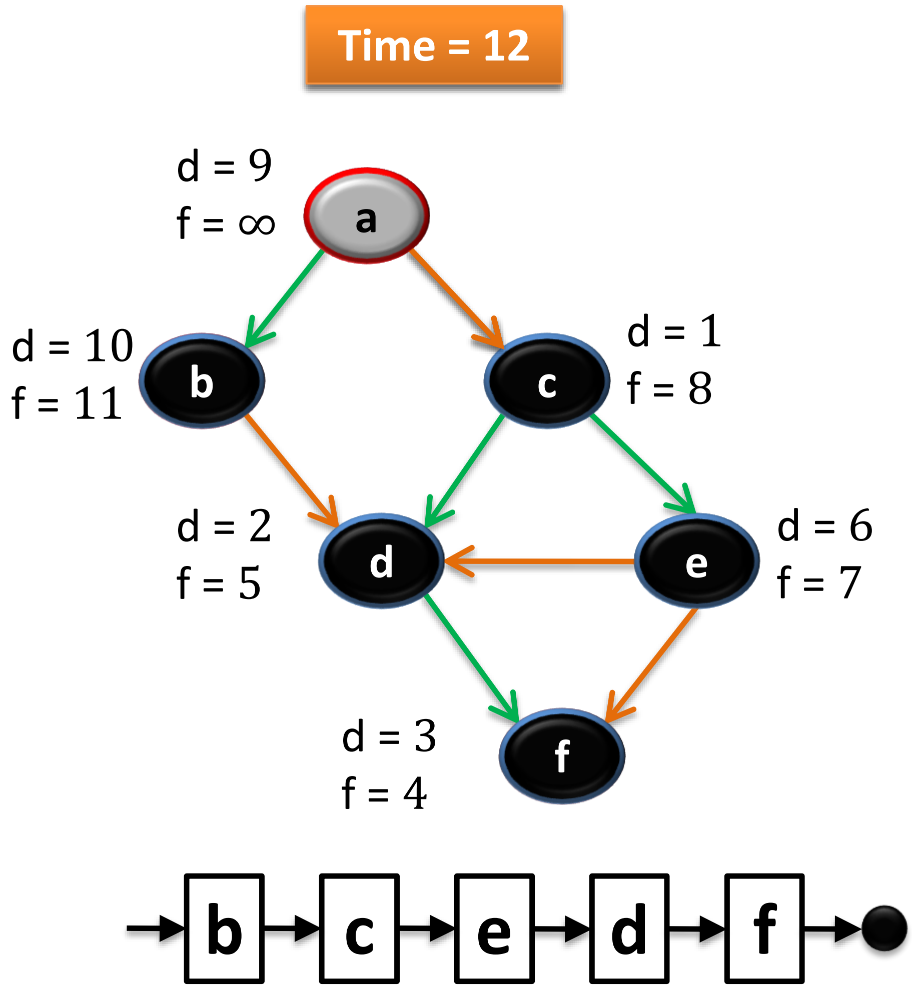
Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

Next we discover the vertex **b**

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

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**

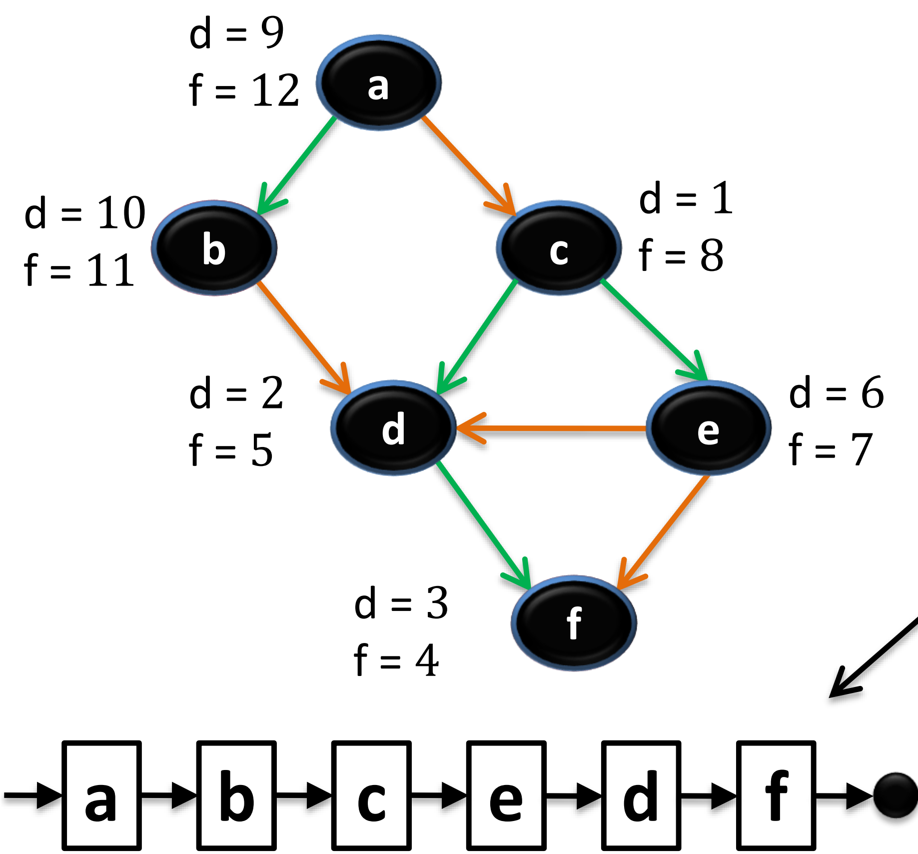
b is done as (**b**,**d**) is a cross edge => now move back to **c**

a is done as well

Topological sort

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

Time = 13



WE HAVE THE RESULT!

3) return the linked list of vertices

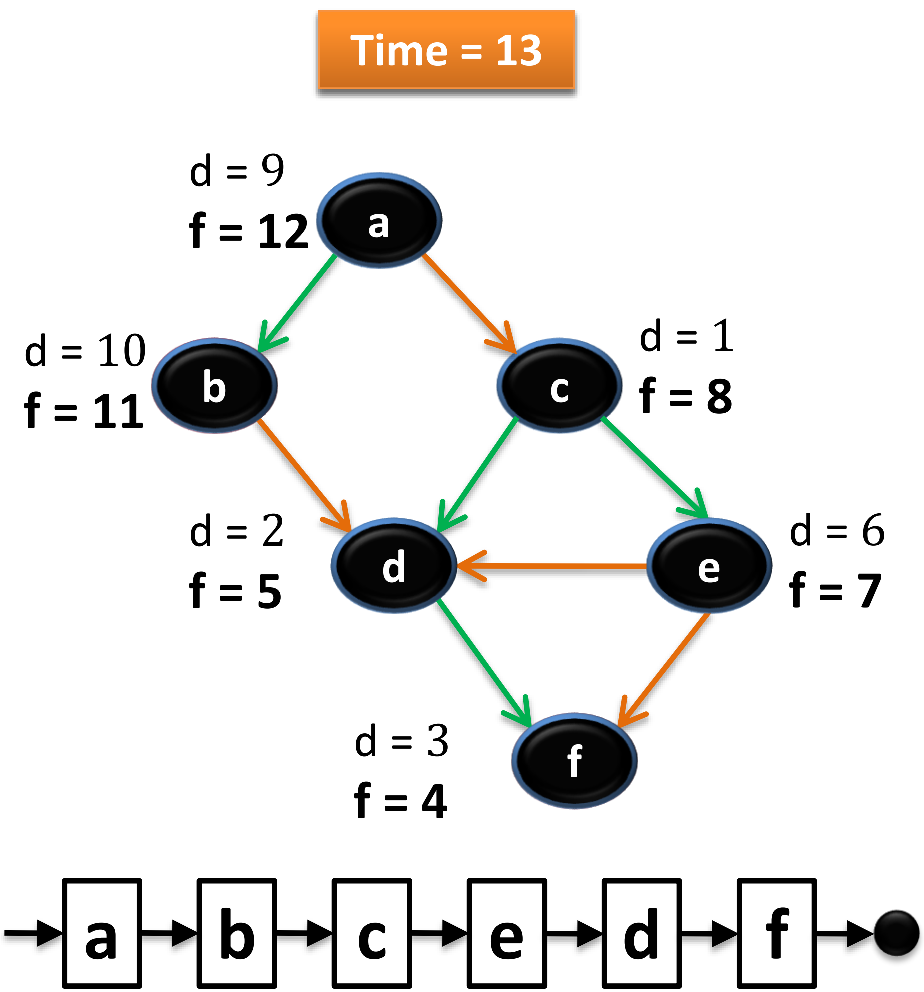
=> (**a**,**c**) is a cross edge

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



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

Try yourself with different vertex order for DFS visit

Note: If you redraw the graph so that all vertices are in a line ordered by a valid topological sort, then all edges point **„from left to right"**

Topological sort

- TOPOLOGICAL-SORT(**G**):
  1) call DFS(G) to compute **finishing** times **f**[**v**] for each vertex **v**
  2) as each vertex is finished, insert it onto the **front** of a linked list
  3) return the linked list of vertices

Time complexity of TS(G)

- Running time of topological sort:

  $$\mathbf{\Theta(n + m)}$$

  where **n**=|**V**| and **m**=|**E**|

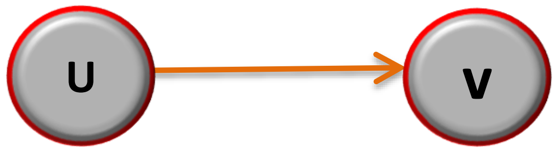
- Why? Depth first search takes **Θ(n + m)** time in the worst case, and inserting into the front of a linked list takes **Θ(1)** time

Proof of correctness

- **Theorem**: TOPOLOGICAL-SORT(**G**) produces a topological sort of a DAG **G**

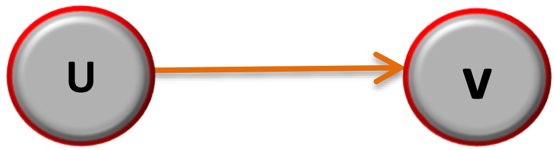
- The TOPOLOGICAL-SORT(**G**) algorithm does a DFS on the DAG **G**, and it lists the nodes of **G** in order of decreasing finish times **f**[]
- We must show that this list satisfies the topological sort property, namely, that for every edge (**u**,**v**) of **G**, **u** appears before **v** in the list
- **Claim**: For every edge (**u**,**v**) of **G**: **f**[**v**] < **f**[**u**] in DFS

Proof of correctness



"For every edge (**u**,**v**) of **G**, **f**[**v**] < **f**[**u**] in this DFS"

Proof of correctness



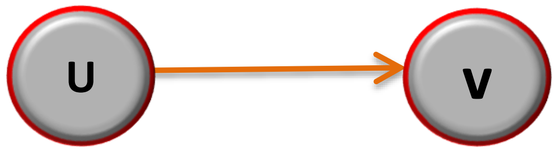
"For every edge (**u**,**v**) of **G**, **f**[**v**] < **f**[**u**] in this DFS"

Case 1: If DFS-Visit is called on v first

f[**v**] < **f**[**u**]

Since the graph has no cycle so there cannot be a path from v to u so **v must finish before u**

Proof of correctness



"For every edge (**u**,**v**) of **G**, **f**[**v**] < **f**[**u**] in this DFS"

Case 2: If DFS-Visit is called on u first

f[**v**] < **f**[**u**]

Proof: Due to the edge (u,v) v will be explored and finished and then finish time of u is updated.

Slide Credits

- Trevor Brown, University of Toronto