CS60020: Foundations of Algorithm Design and Machine Learning

Sourangshu Bhattacharya

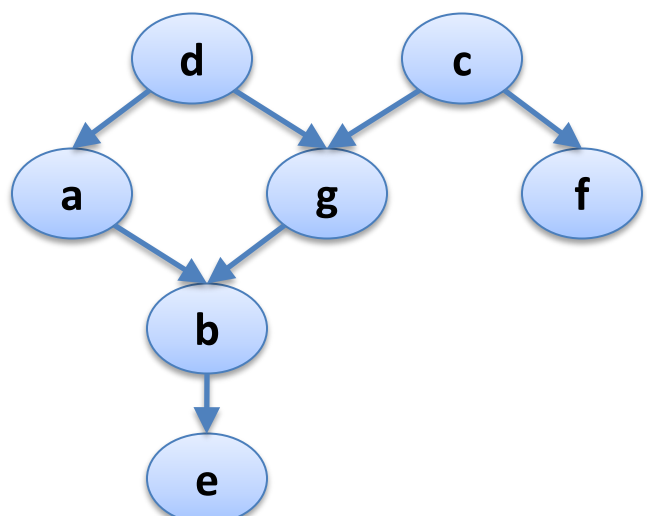
TOPOLOGICAL SORT

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 (the most efficient scheduling is an NP-complete problem)
- Or during compilation to order modules/libraries



Examples

- **Resolving dependencies**: *apt-get* uses topological sorting to obtain the admissible sequence in which a set of Debian packages can be installed/removed

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**

DFS Algorithm

```
DFS-VISIT(G, u)
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK
time = time + 1
u.f = time
```

```
DFS(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for each vertex u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

u.d : Discovery Time
u.F : Finishing Time

u.color : tracks the visited status.

Algorithm for TS

- 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

- Note that the result is just a list of vertices in order of **decreasing** finish times **f**[]

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:

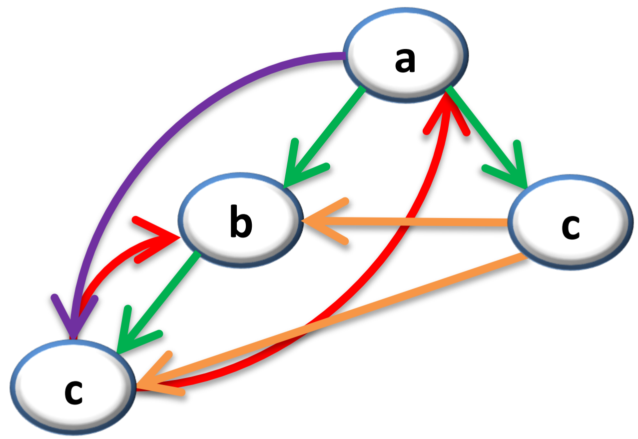
(2) **Forward edge** iff **u** is an ancestor of **v** in the DFS tree

(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

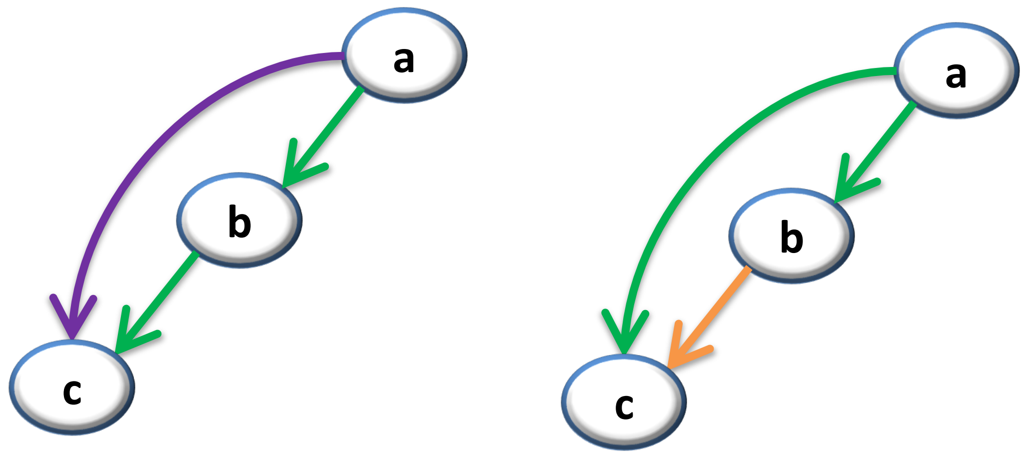


The edge classification depends on the particular DFS tree!

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**)

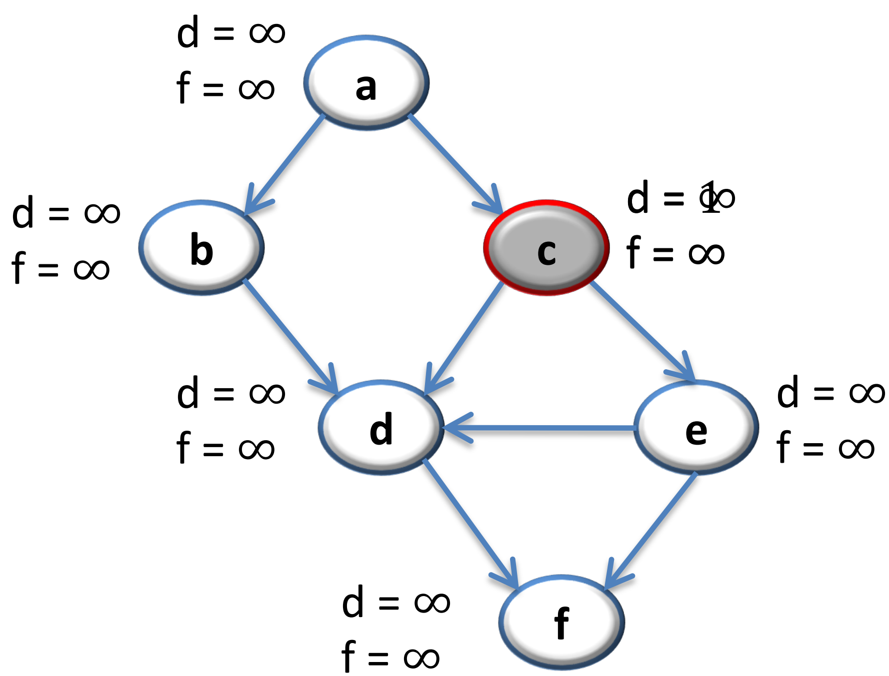
Back to topological sort

- 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

Topological sort

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

Time = 2



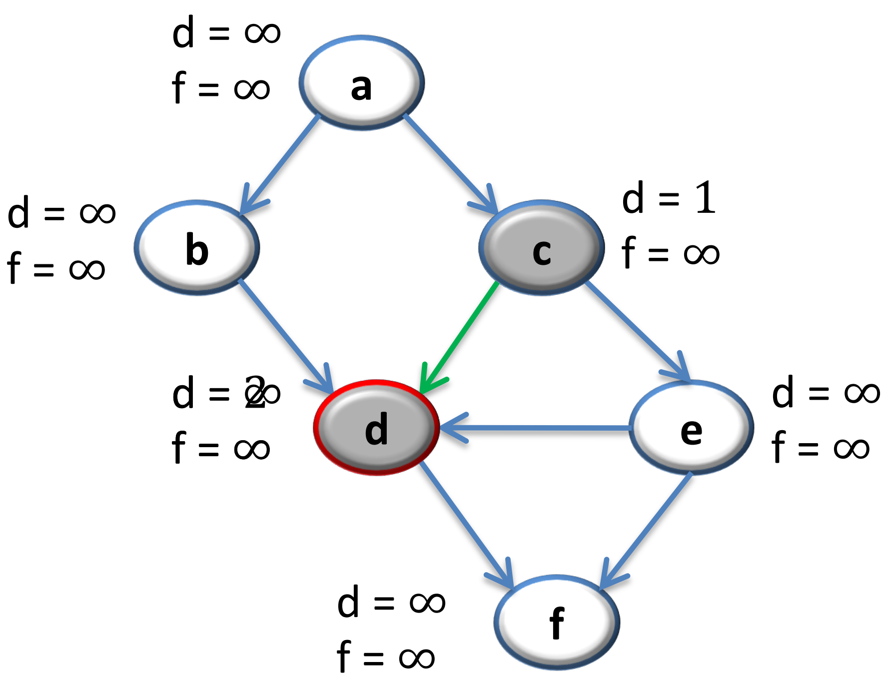
Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Topological sort

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

Time = 3



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

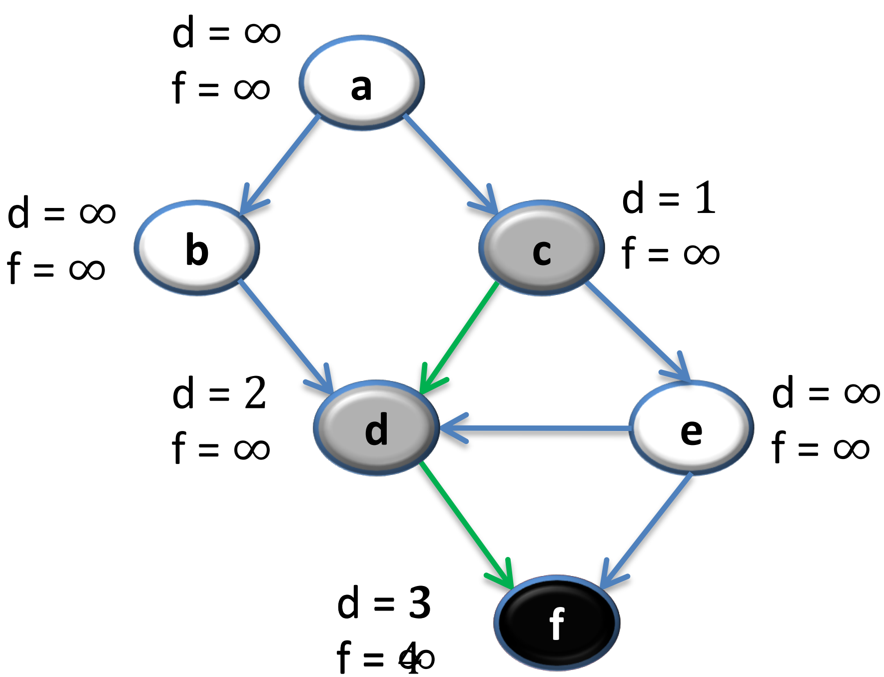
Next we discover the vertex **d**

Topological sort

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

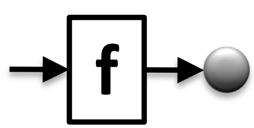
2) as each vertex is finished, insert it onto the **front** of a linked list

Time = 4



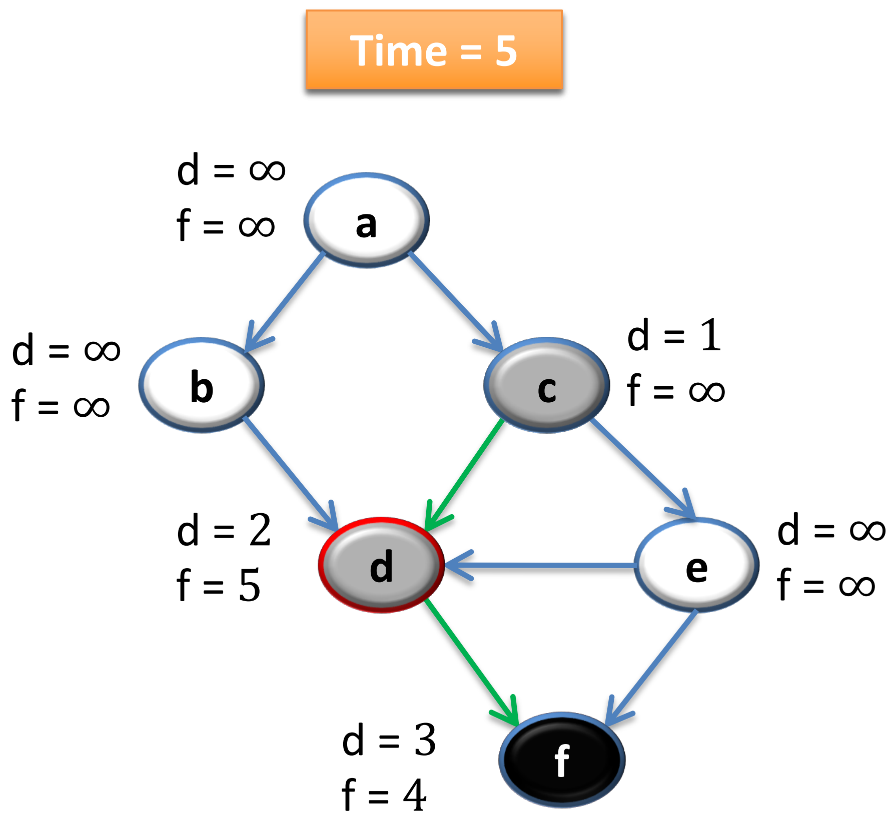
Next we discover the vertex **f**

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



Topological sort

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



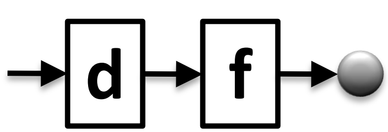
Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Next we discover the vertex **f**

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

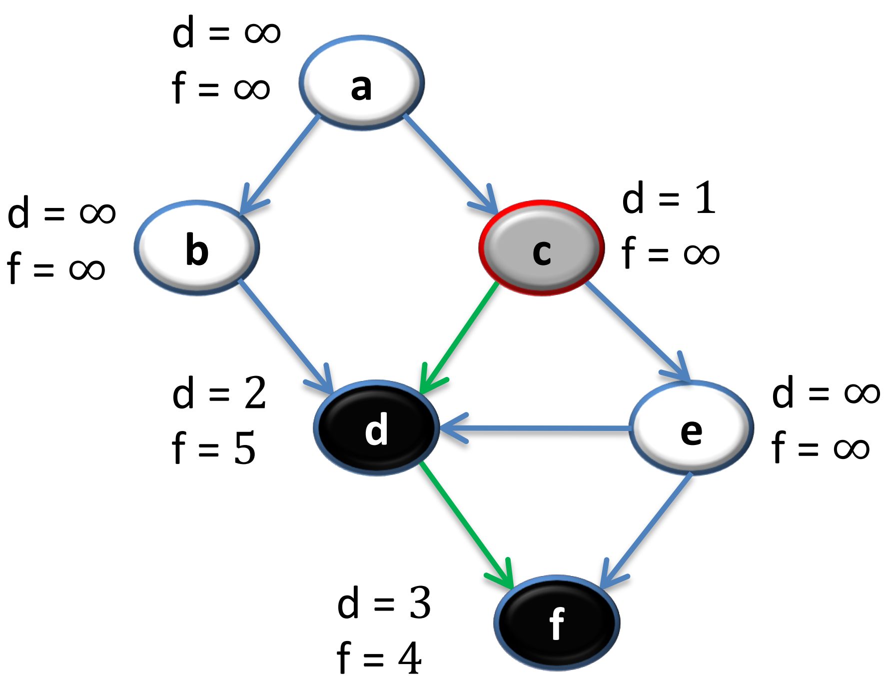
d is done, move back to **c**

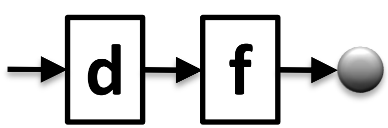


Topological sort

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**

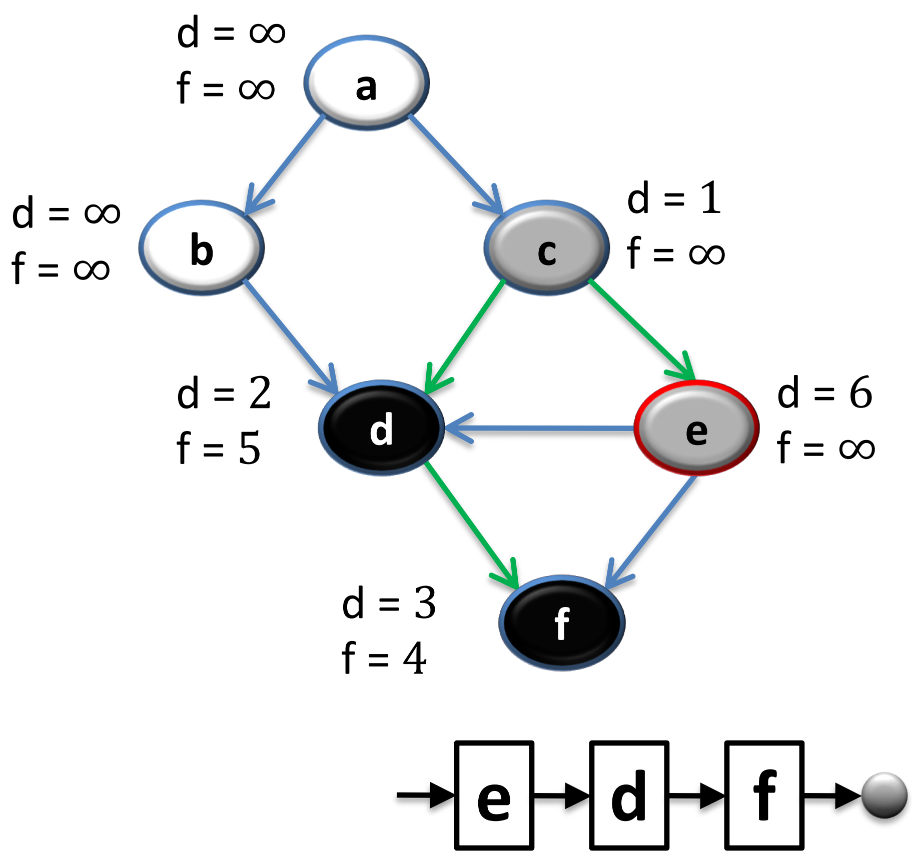
f is done, move back to **d**

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

Next we discover the vertex **e**

Topological sort

Time = 7



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**

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

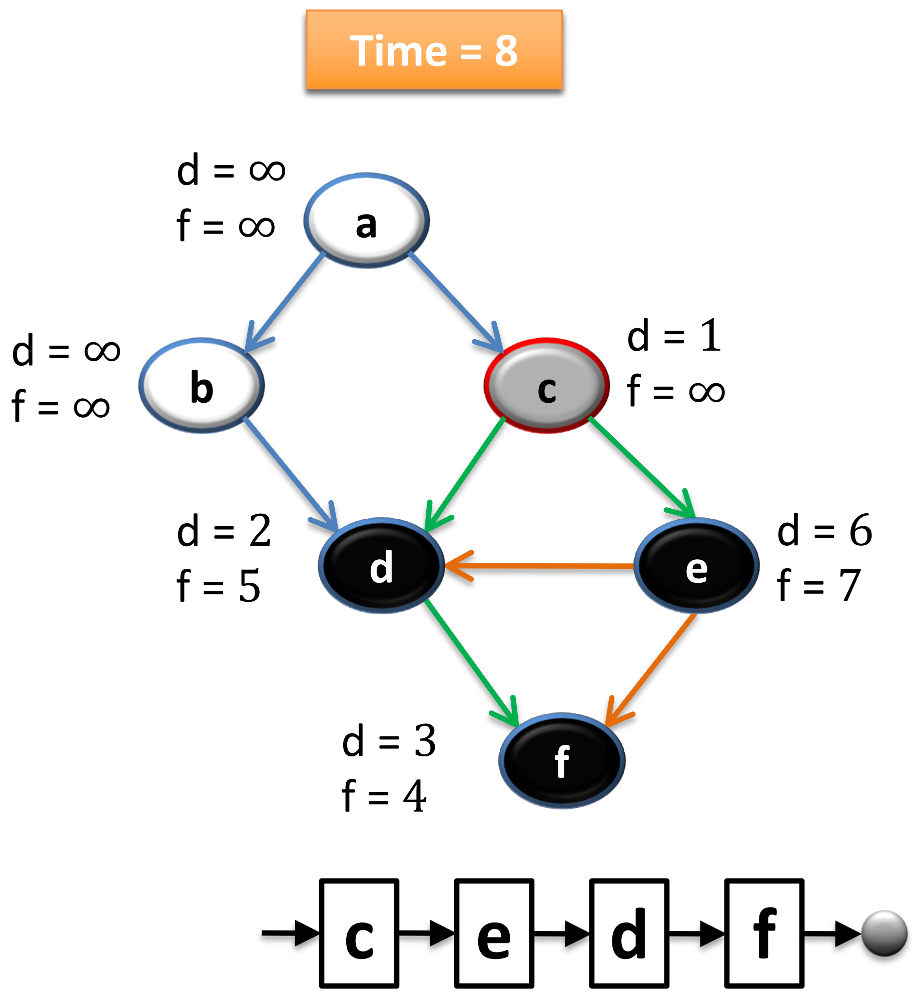
d is done, move back to **c**

Next we discover the vertex **e**

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

Topological sort

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



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**

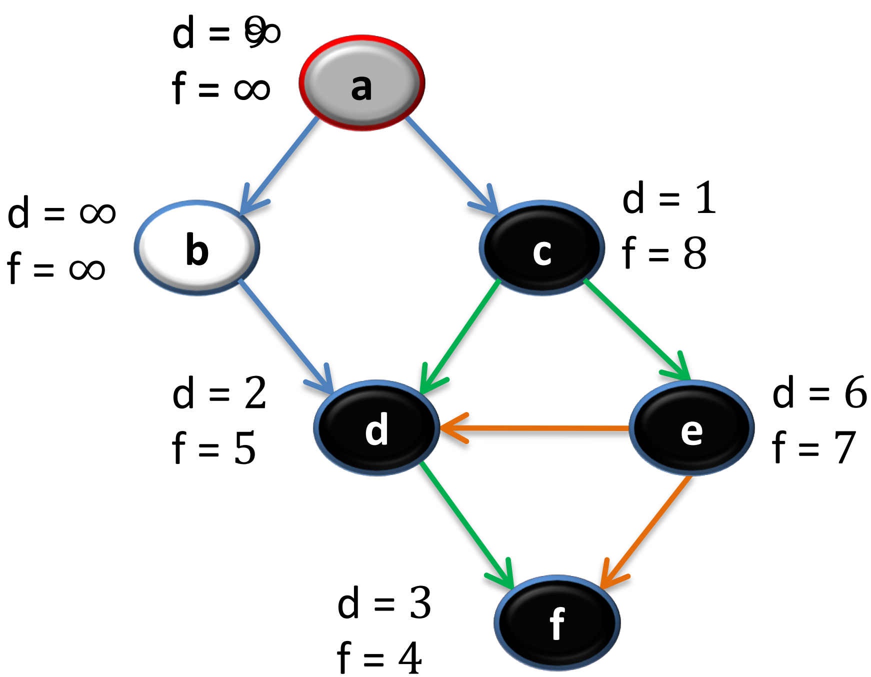
e is done, move back to **c**

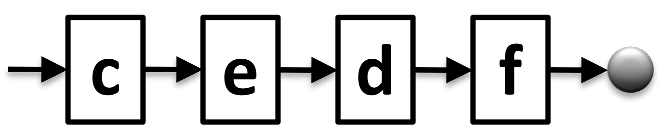
c is done as well

Topological sort

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

Time = 10





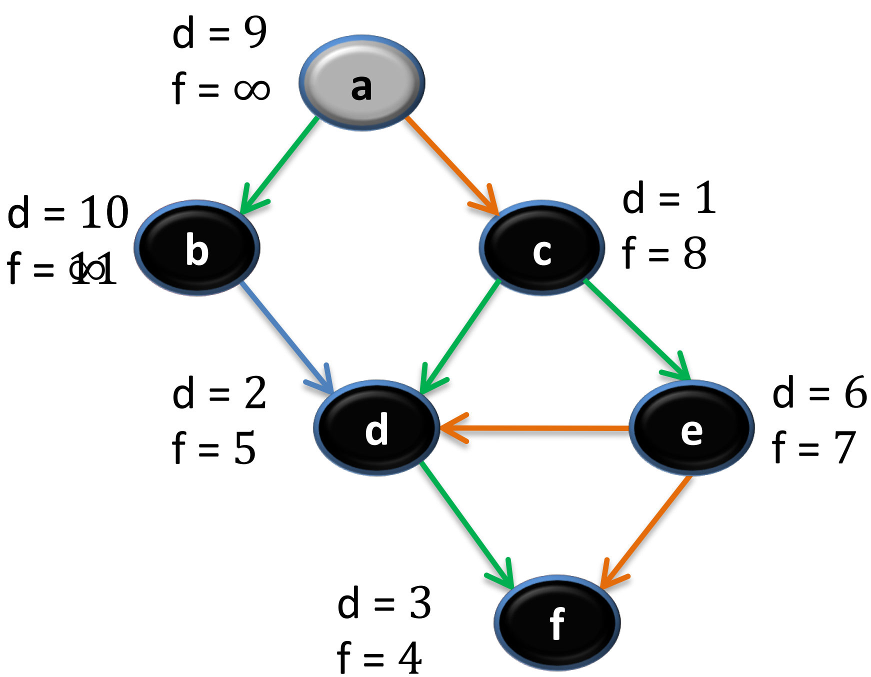
Let’s now call DFS visit from the vertex **a**

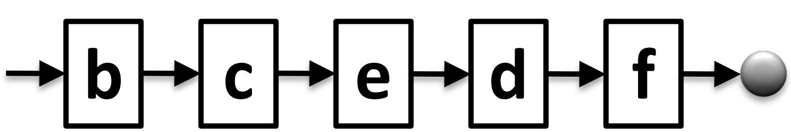
Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

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**

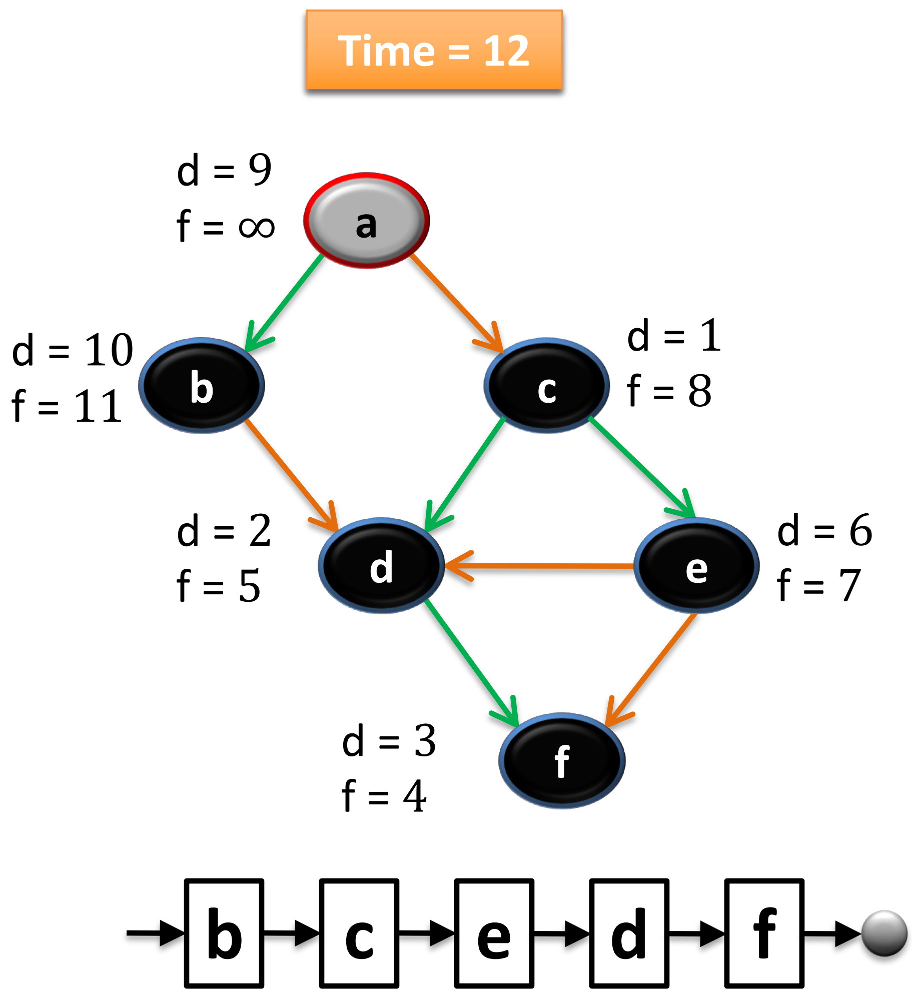
Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

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**

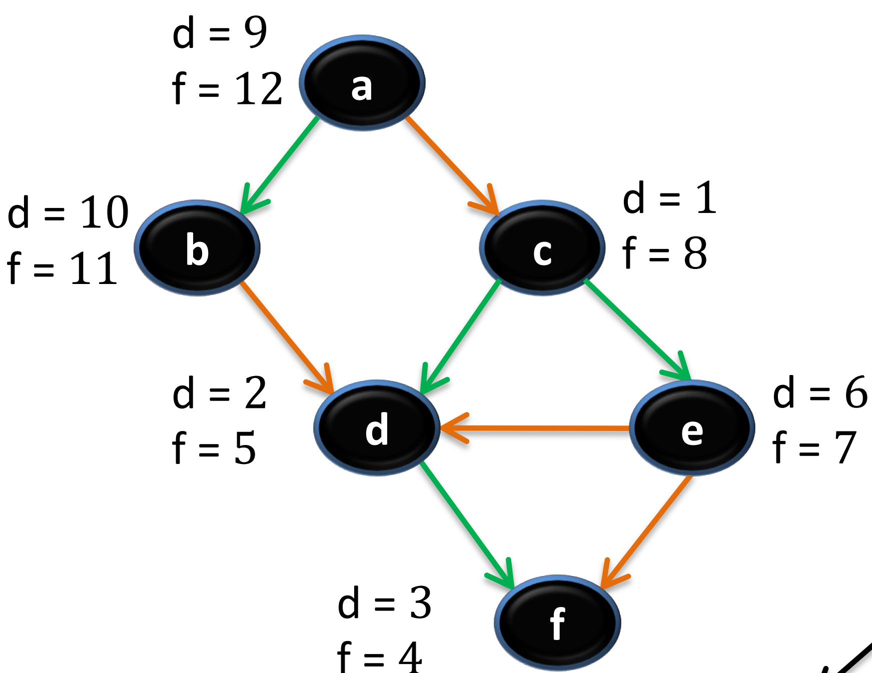
b is done as (**b**,**d**) is a cross edge => now move back to **c**

a is done as well

Topological sort

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

Time = 13



WE HAVE THE RESULT!

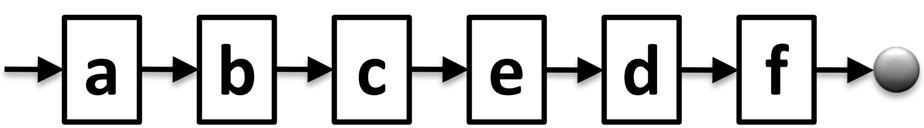
3) return the linked list of vertices

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

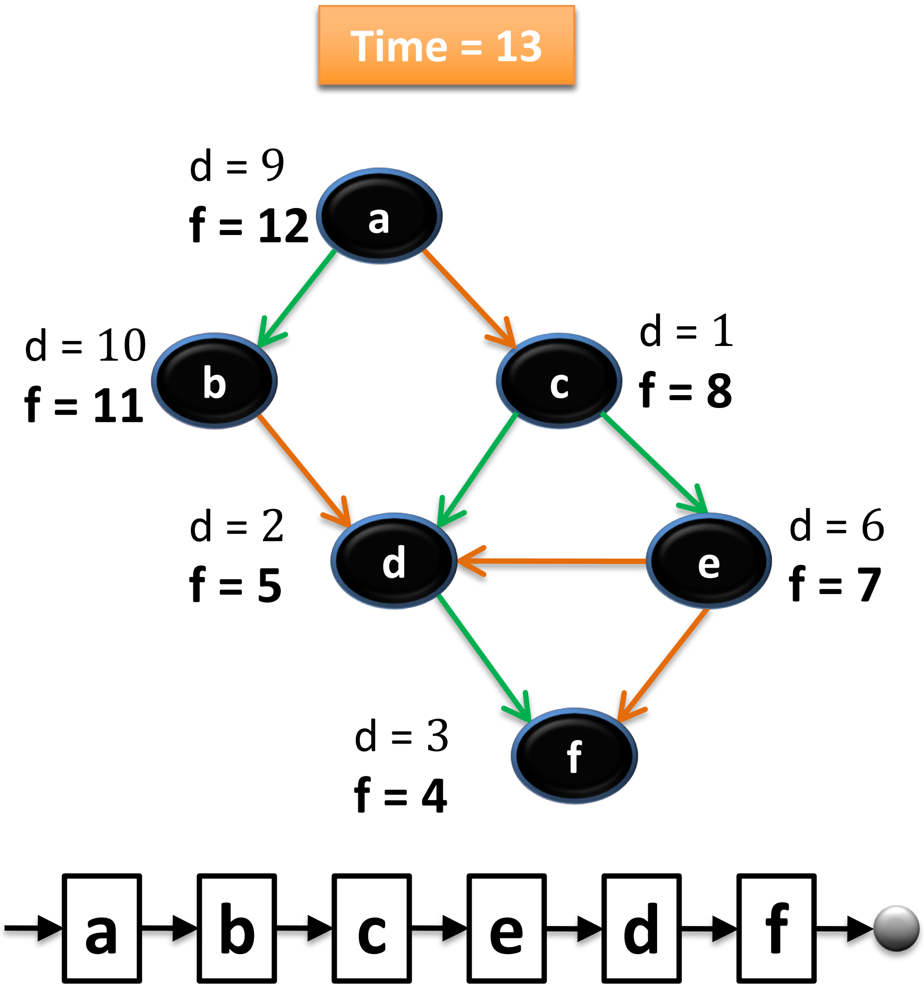
Next we discover the vertex **b**

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“**

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"

- The DFS classifies (**u**,**v**) as a **tree edge**, a **forward edge** or a **cross-edge** (it cannot be a back-edge since **G** has no cycles):
  i. If (**u**,**v**) is a **tree** or a **forward edge** $\Rightarrow$ **v** is a descendant of **u** $\Rightarrow$ **f**[**v**] < **f**[**u**]
  ii. If (**u**,**v**) is a **cross-edge**

Proof of correctness

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

Q.E.D. of Claim

ii. If (**u**,**v**) is a **cross-edge**:

- as (**u**,**v**) is a cross-edge, by definition, neither **u** is a descendant of **v** nor **v** is a descendant of **u:**

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

or

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

f[v] < f[u]

since (**u**,**v**) is an edge, **v** is surely discovered **before** **u**'s exploration completes

Proof of correctness

- TOPOLOGICAL-SORT(G) lists the nodes of G from highest to lowest finishing times

- By the **Claim**, for every edge (**u**,**v**) of **G**:

$$\mathbf{f}[\mathbf{v}] < \mathbf{f}[\mathbf{u}]$$

⇒ **u** will be before **v** in the algorithm's list

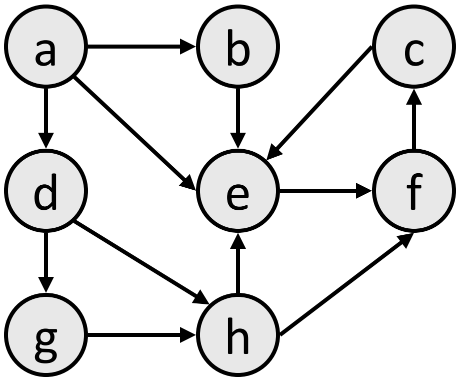
- Q.E.D of **Theorem**

BREADTH FIRST SEARCH

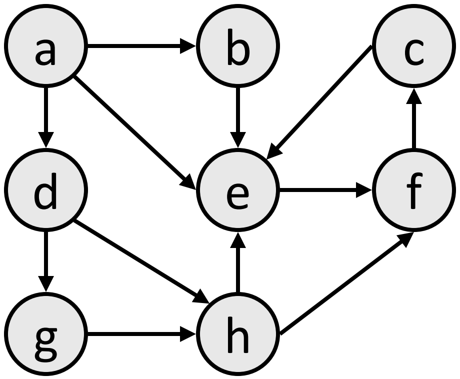
Breadth-first search

- **breadth-first search** (BFS): Finds a path between two nodes by taking one step down all paths and then immediately backtracking.
  - Often implemented by maintaining a queue of vertices to visit.

- BFS always returns the shortest path (the one with the fewest edges) between the start and the end vertices.
  - to b: {a, b}
  - to c: **{a, e, f, c}**
  - to d: {a, d}
  - to e: **{a, e}**
  - to f: **{a, e, f}**
  - to g: {a, d, g}
  - to h: **{a, d, h}**



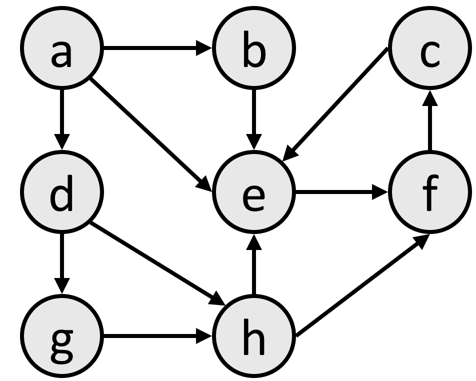
BFS pseudocode




```
function bfs(v1, v2):
    queue := {v1}.
    mark v1 as visited.

    while queue is not empty:
        v := queue.removeFirst().
        if v is v2:
            a path is found!

        for each unvisited neighbor n of v:
                mark n as visited.
            queue.addLast(n).

    // path is not found.
```

- Trace bfs(*a*, *f*) in the above graph.

BFS observations



- *optimality*:
  - always finds the shortest path (fewest edges).
  - in unweighted graphs, finds optimal cost path.
  - In weighted graphs, *not* always optimal cost.
- *retrieval*: harder to reconstruct the actual sequence of vertices or edges in the path once you find it
  - conceptually, BFS is exploring many possible paths in parallel, so it's not easy to store a path array/list in progress
  - solution: We can keep track of the path by storing predecessors for each vertex (each vertex can store a reference to a *previous* vertex).
- DFS uses less memory than BFS, easier to reconstruct the path once found; but DFS does not always find shortest path. BFS does.

DFS, BFS runtime

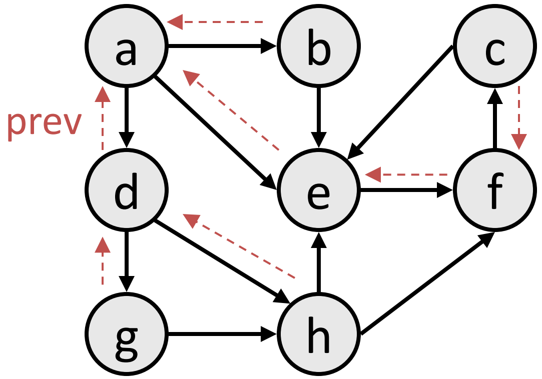
- What is the expected runtime of DFS and BFS, in terms of the number of vertices V and the number of edges E ?

- Answer: O(|V| + |E|)
  - where |V| = number of vertices, |E| = number of edges
  - Must potentially visit every node and/or examine every edge once.

  - why not O(|V| * |E|) ?

- What is the space complexity of each algorithm?
  - (How much memory does each algorithm require?)

BFS that finds path



function **bfs**(v_1, v_2):
  queue := {v_1}.
  mark v_1 as visited.

  while *queue* is not empty:
    v := *queue*.removeFirst().
    if v is v_2:
      a path is found! *(reconstruct it by following .prev back to v_1.)*

    for each unvisited neighbor n of v:
      mark n as visited. *(set n.prev = v.)*
      queue.addLast(n).

  // path is not found.

– By storing some kind of "previous" reference associated with each vertex, you can reconstruct your path back once you find v_2.