Graph Algorithms-II

Topological Sorting and Strongly connected Components

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

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

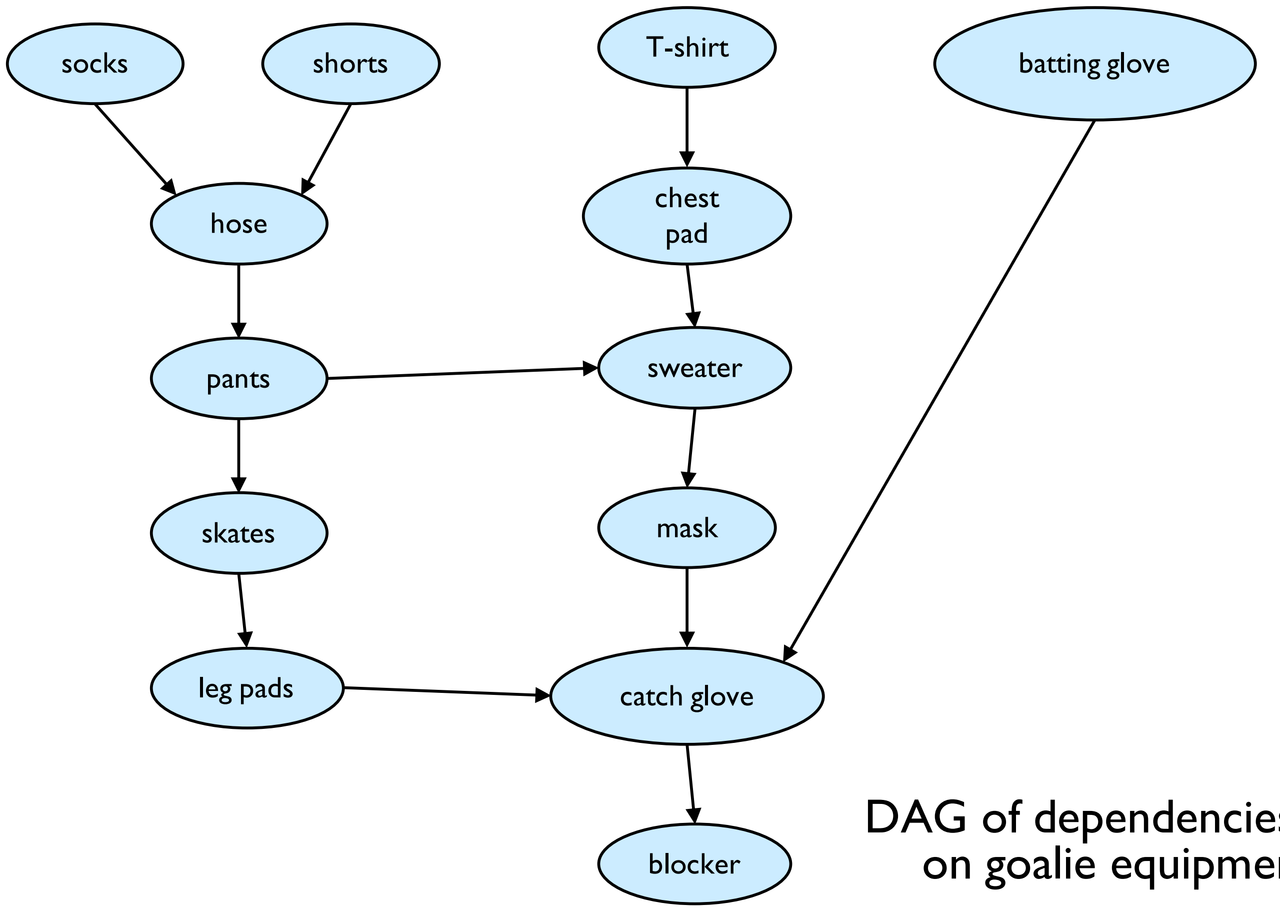
Topological sort more formally

- 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**

Directed Acyclic Graph

- DAG – Directed graph with no cycles.
- Eg:



DAG of dependencies for putting on goalie equipment.

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

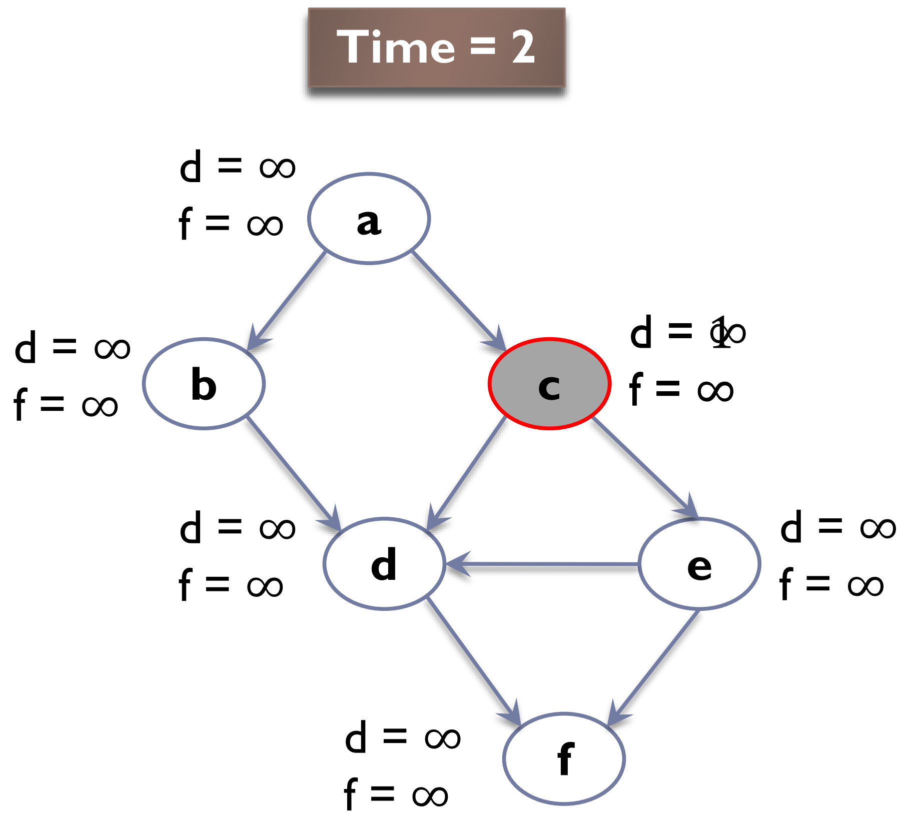
- Performed on a DAG.
- Linear ordering of the vertices of G such that if $(u, v) \in E$, then u appears somewhere before v.

```
Topological-Sort ($G$)
1.   call DFS($G$) to compute finishing times $f[v]$ for all $v \in V$
2.   as each vertex is finished, insert it onto the front of a linked list
3.   **return** the linked list of vertices
```

Time: $\Theta(V + E)$.

Topological sort

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



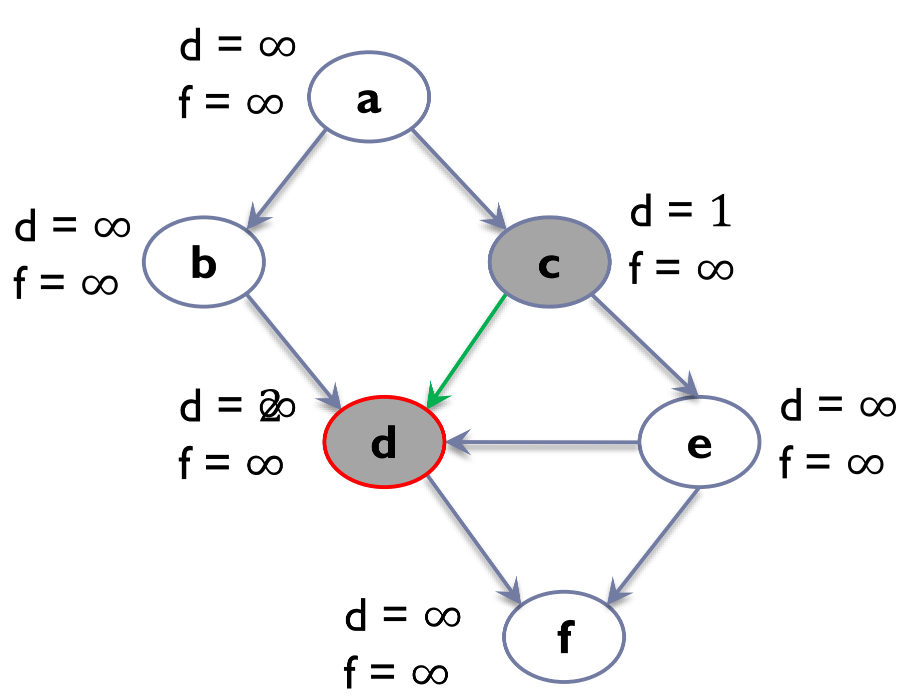
Let's say we start the DFS from the vertex **c**

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**

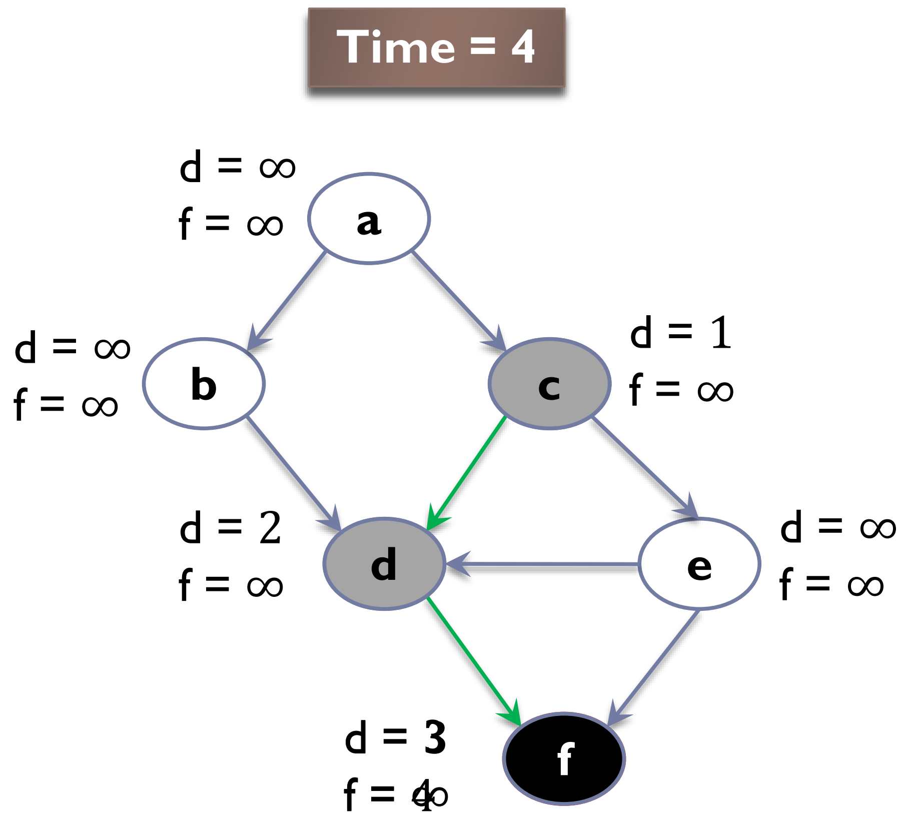
Topological sort

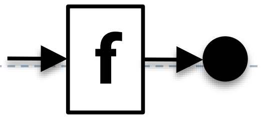
1) Call DFS(**G**) to compute the finishing times **f[v]**

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

Next we discover the vertex **f**

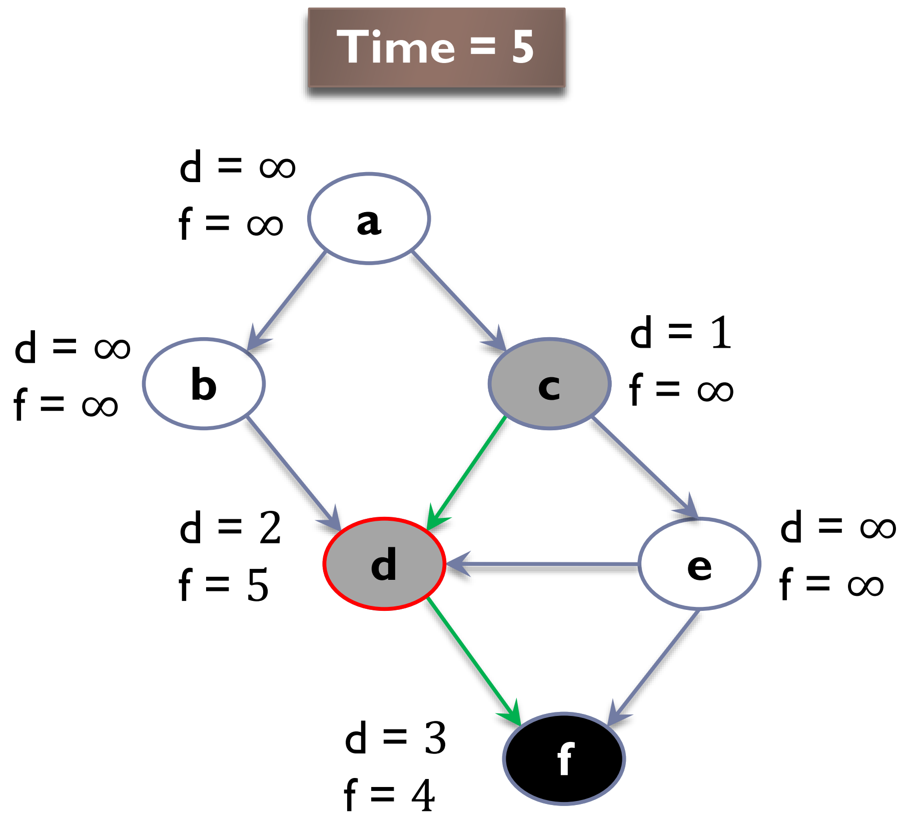
f is done, move back to **d**





Topological sort

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



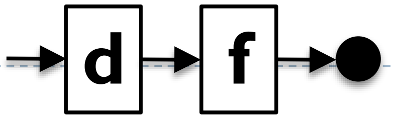
Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Next we discover the vertex **f**

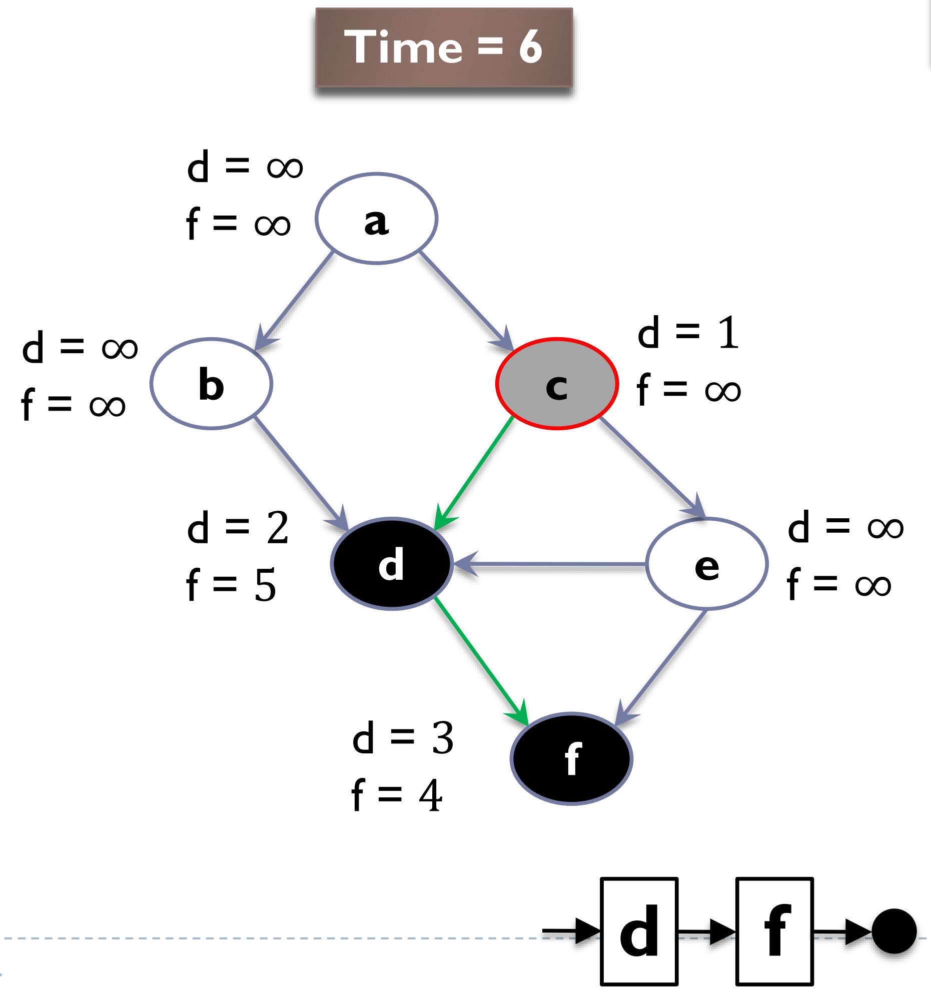
f is done, move back to **d**

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



Topological sort

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



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**

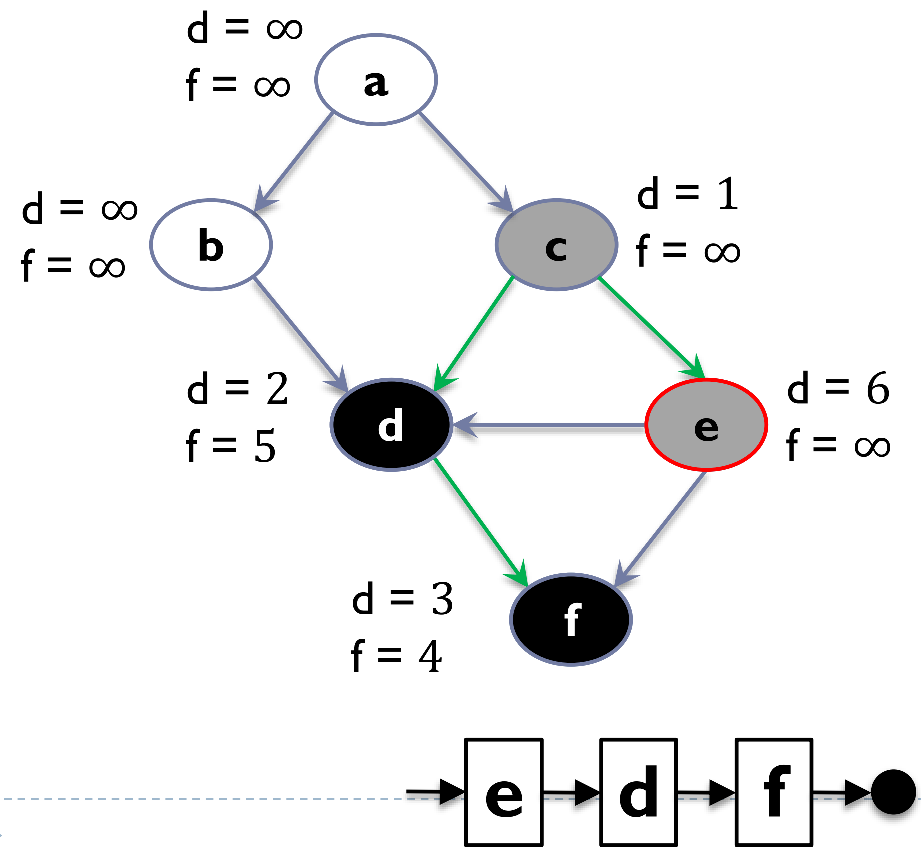
d is done, move back to **c**

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**

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

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

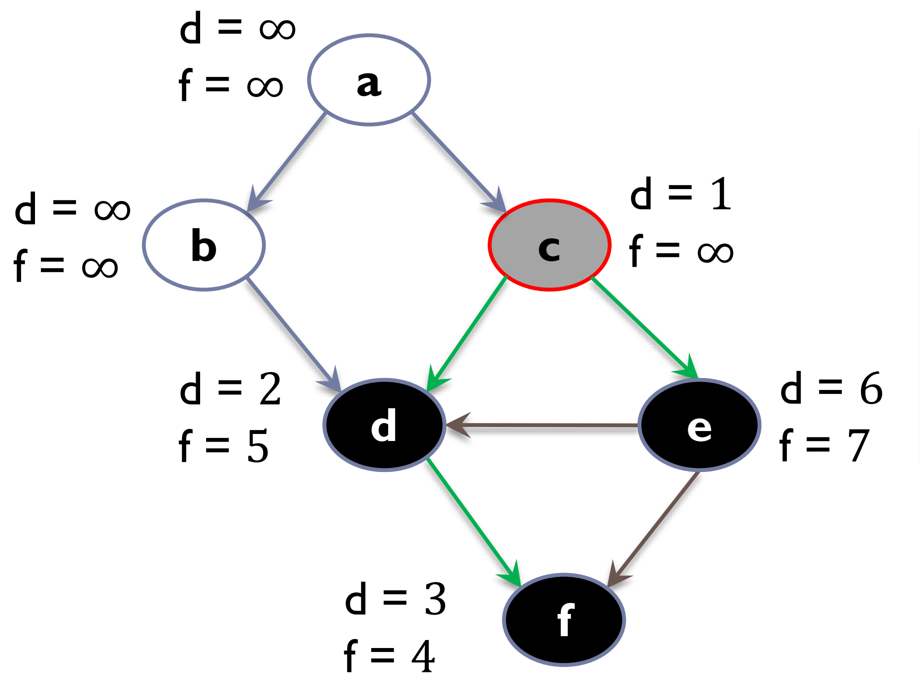
Next we discover the vertex **e**

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

Topological sort

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

Time = 8



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

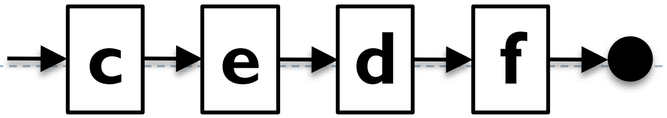
Just a note: If there was (**c**,**f**) edge in the graph, it would be classified as a **forward edge** (in this particular DFS run)

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

Next we discover the vertex **e**

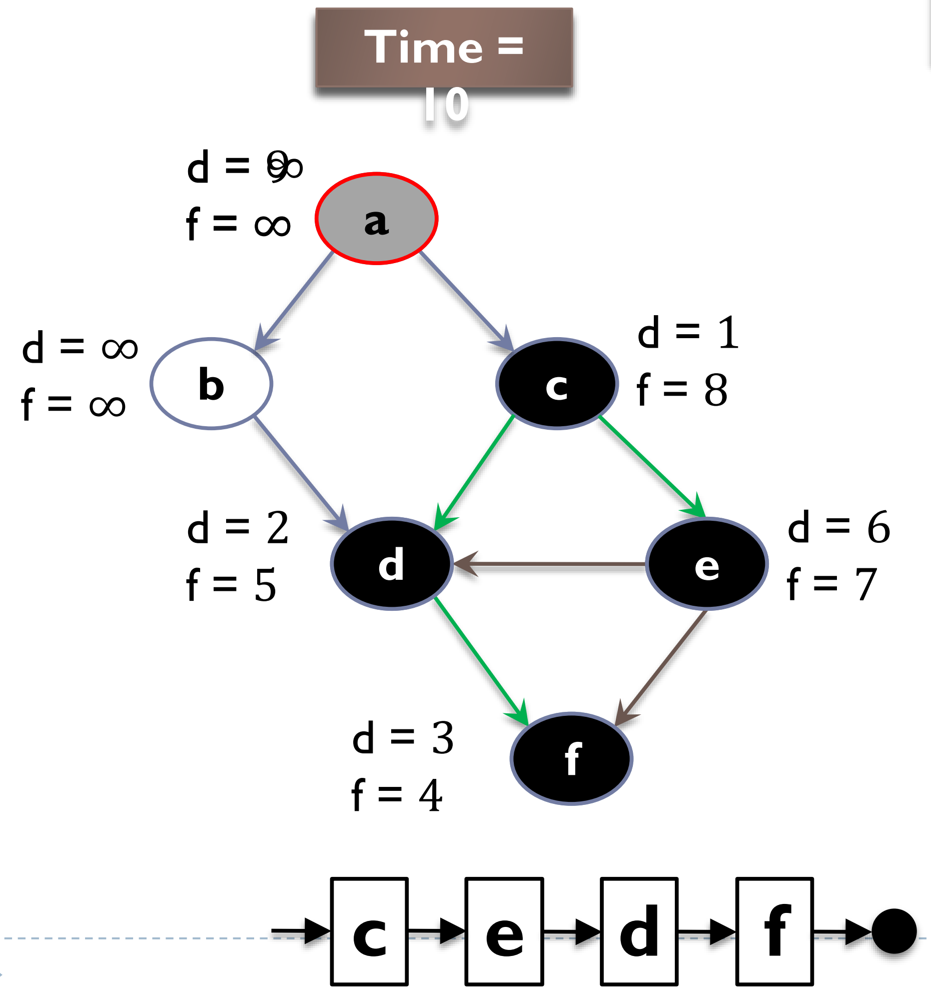
e is done, move back to **c**

c is done as well



Topological sort

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



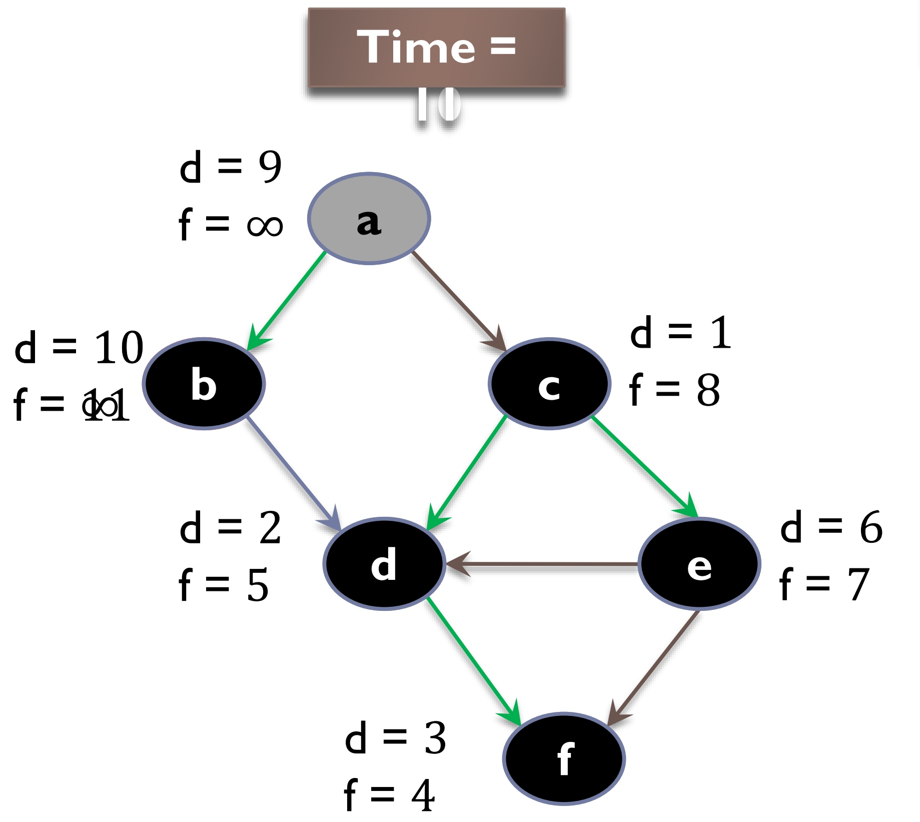
Let's now call DFS visit from the vertex **a**

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

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

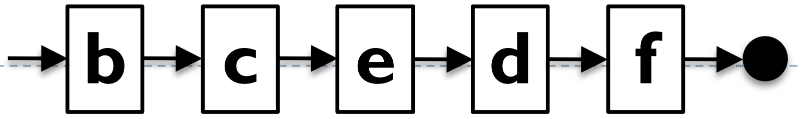


Let's now call DFS visit from the vertex **a**

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

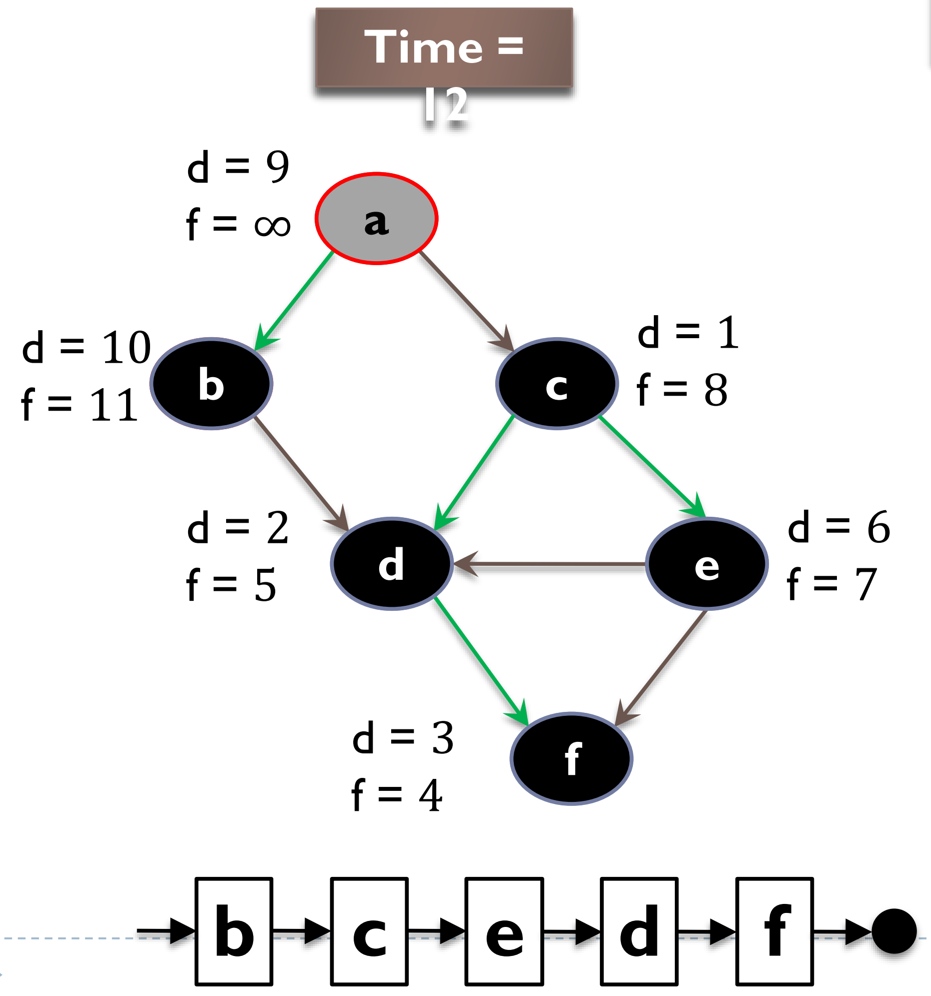
Next we discover the vertex **b**

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]**

WE HAVE THE RESULT!

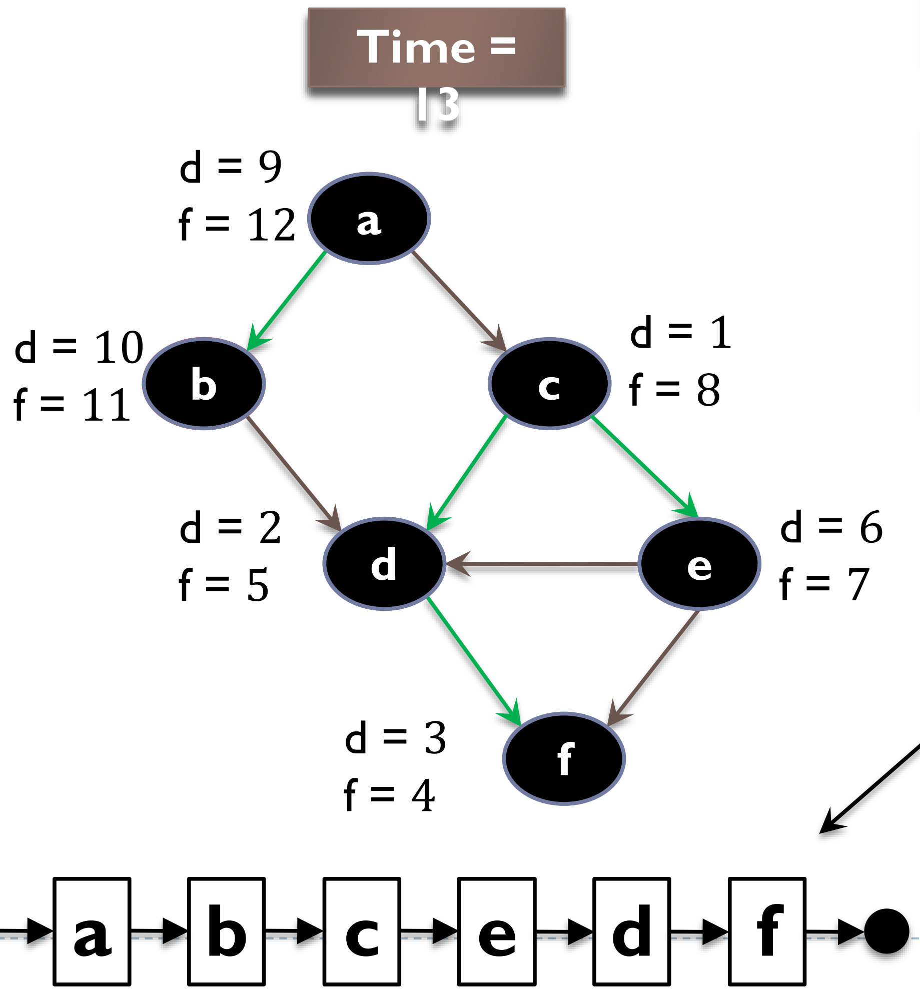
3) return the linked list of vertices

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

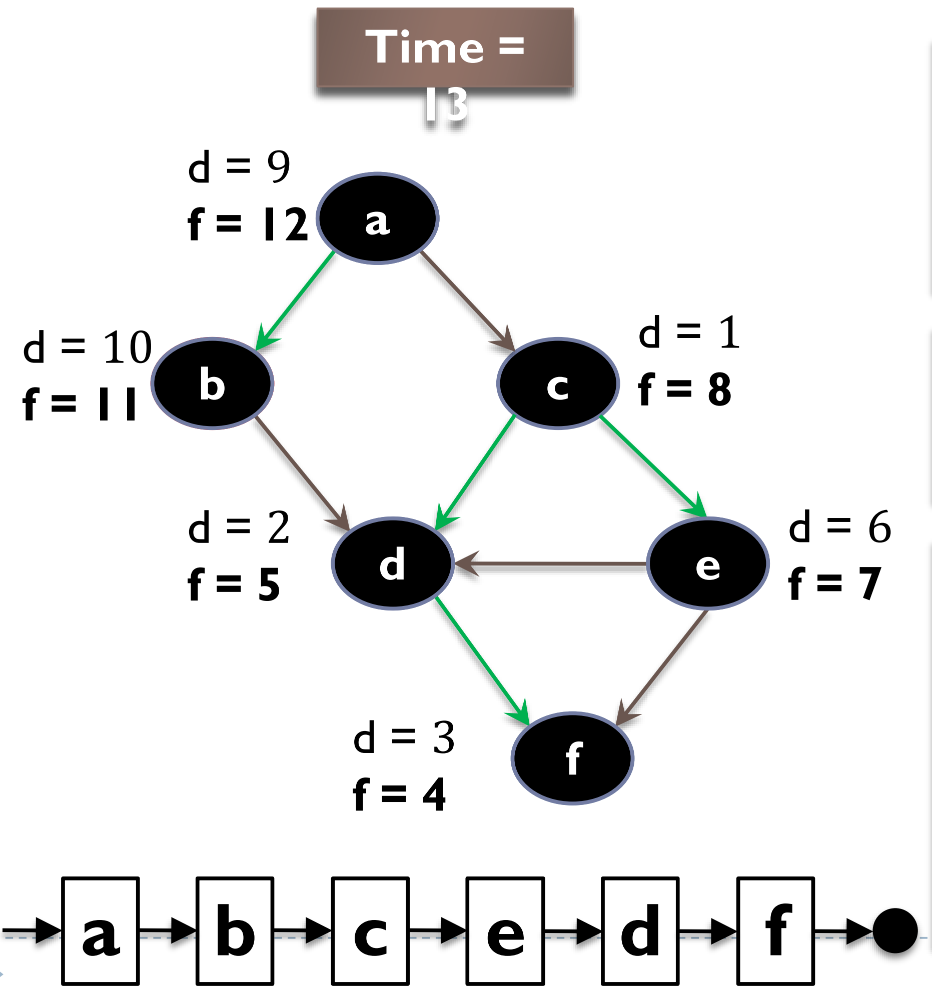
Next we discover the vertex **b**

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

a is done as well



Topological sort

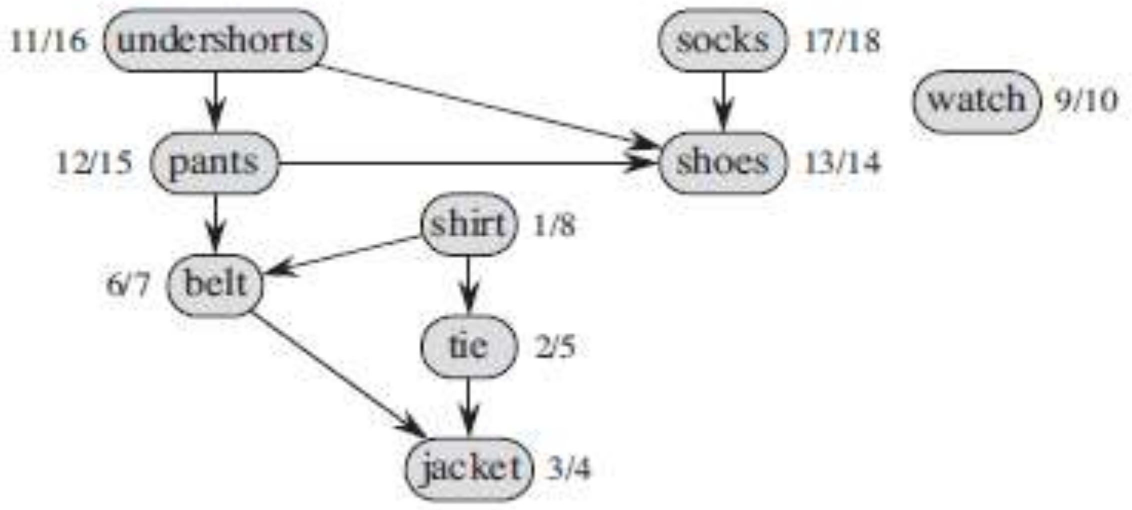


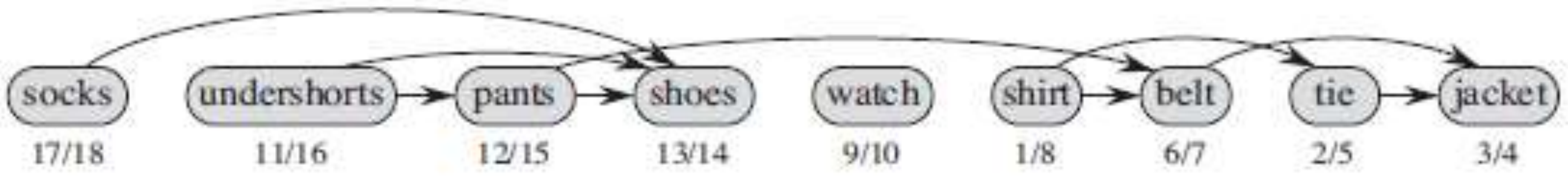
The linked list is sorted in **decreasing** order of finishing times **f[]**

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**“

TS(G)Example:2





Time complexity of TS(G)

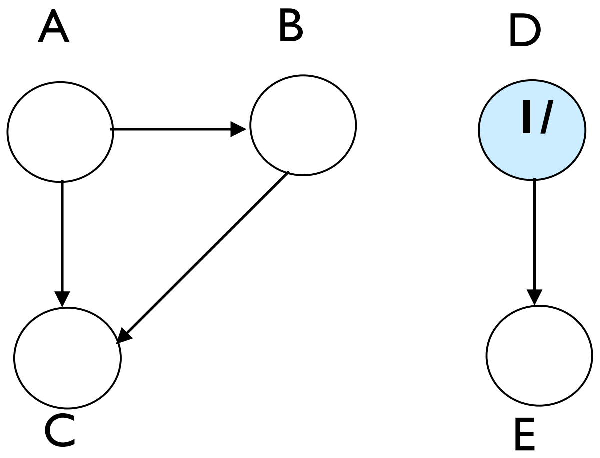
- Running time of topological sort:

  Θ(n + m)

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

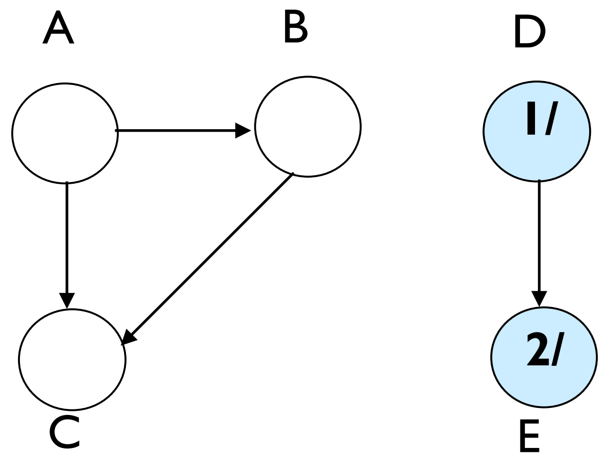
- Why? Depth first search takes **Θ(n + m)** time in the worst case, and inserting into the front of a linked list takes **Θ(1)** time

Example:3



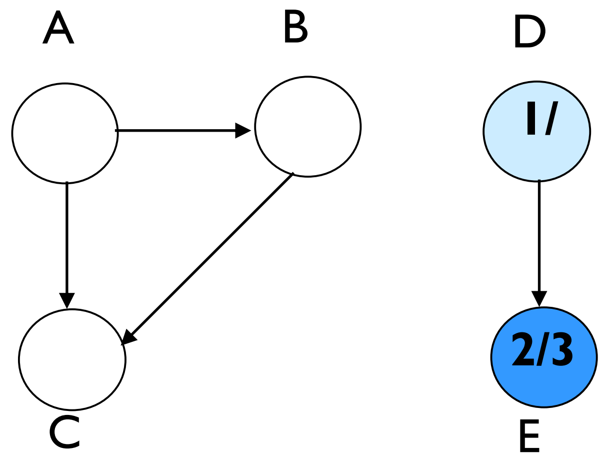
Linked List:

Example

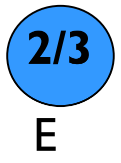


Linked List:

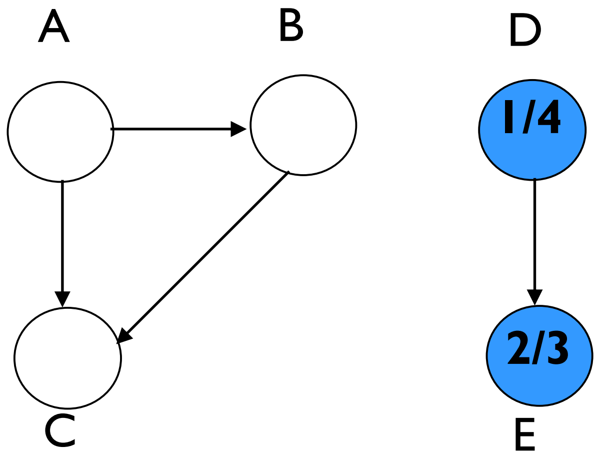
Example



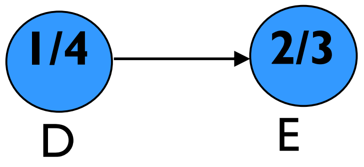
Linked List:



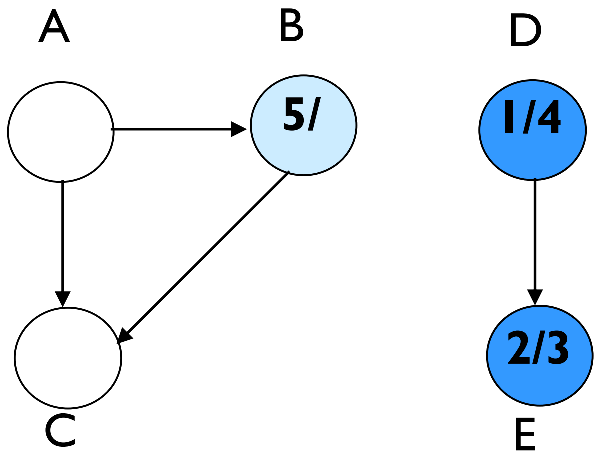
Example



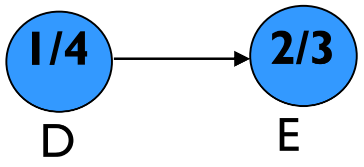
Linked List:



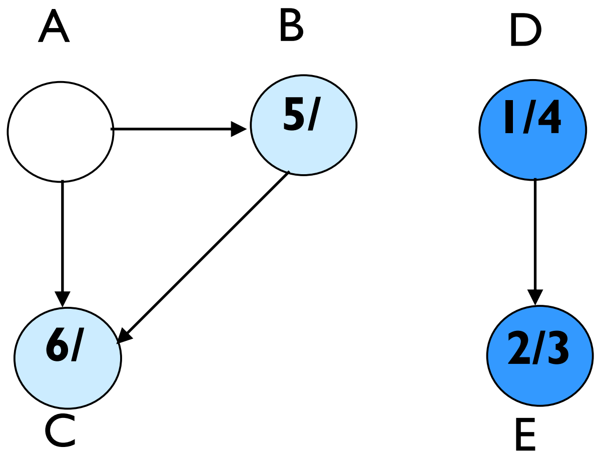
Example



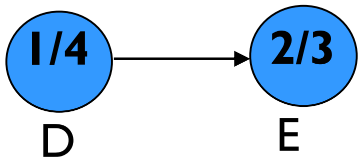
Linked List:



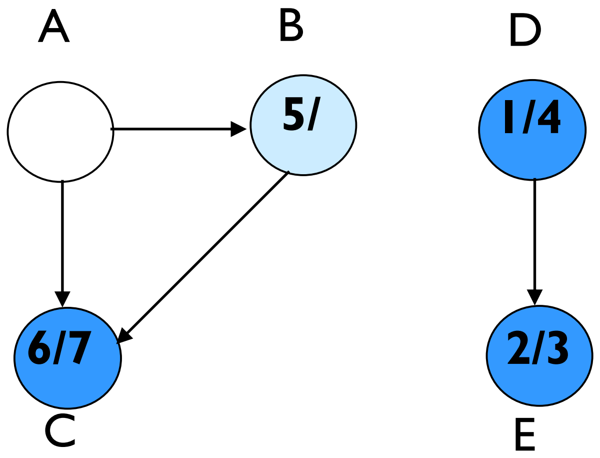
Example



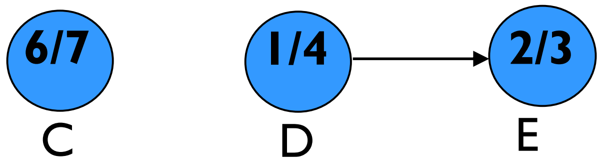
Linked List:



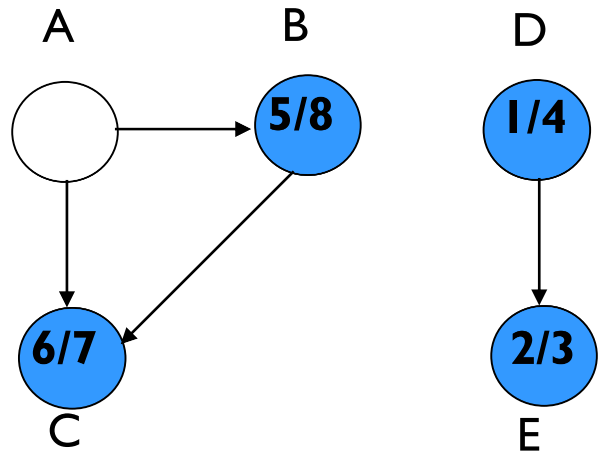
Example



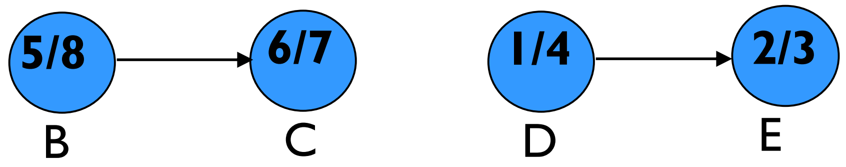
Linked List:



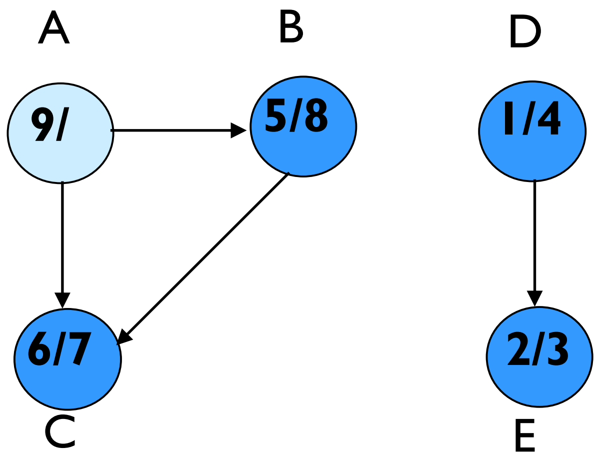
Example



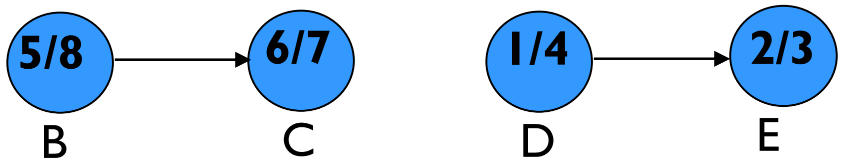
Linked List:



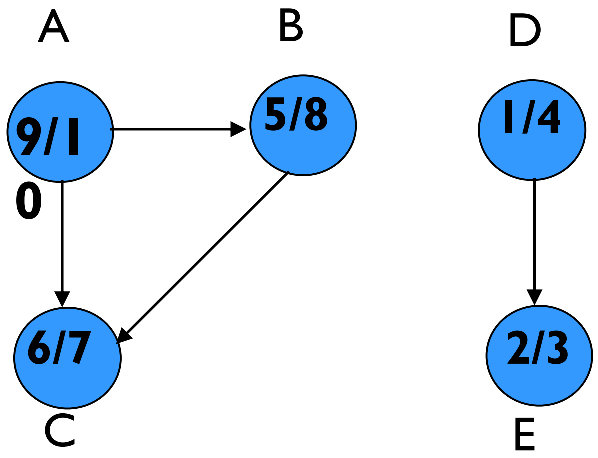
Example



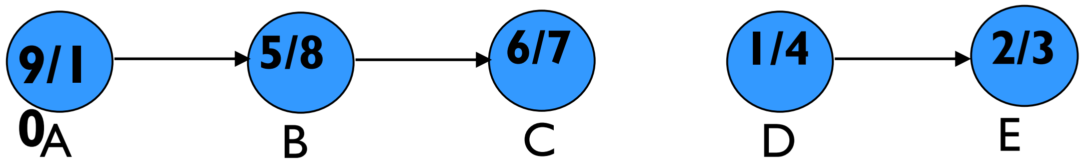
Linked List:



Example

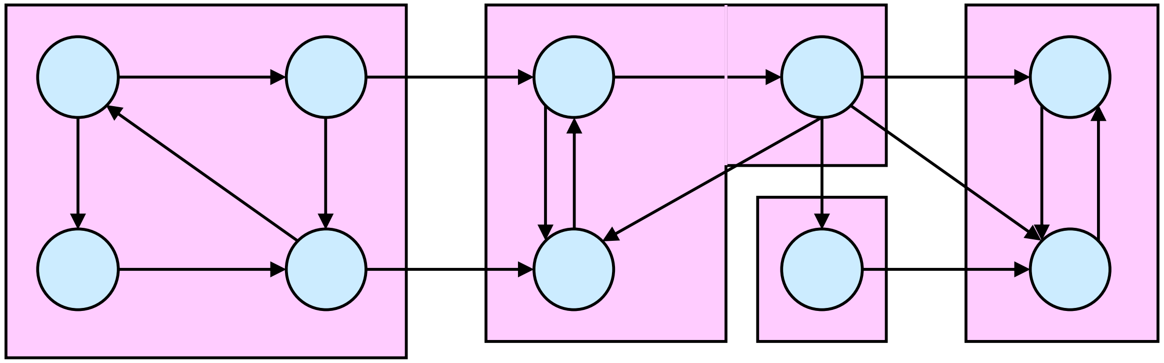


Linked List:



Strongly Connected Components

- *G* is strongly connected if every pair (u, v) of vertices in *G* is reachable from one another.
- A **strongly connected component** (***SCC***) of *G* is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightsquigarrow v$ and $v \rightsquigarrow u$ exist.



Strongly Connected Components

Definition: a strongly connected component (SCC) of a directed graph $G=(V,E)$ is a maximal set of vertices $U \subseteq V$ such that

- For each $u, v \in U$ we have both u □ v and v □ u

i.e., u and v are mutually reachable from each other ($u \leftrightarrows v$)

Let $G^T=(V,E^T)$ be the *transpose* of $G=(V,E)$ where

$$E^T = \{(u,v) : (u,v) \in E\}$$

- i.e., E^T consists of edges of G with their directions reversed

Constructing G^T from G takes $O(V+E)$ time (adjacency list rep)

Note: G and G^T have the same SCCs ($u \leftrightarrows v$ in $G \Leftrightarrow u \leftrightarrows v$ in G^T)

Transpose of a Directed Graph

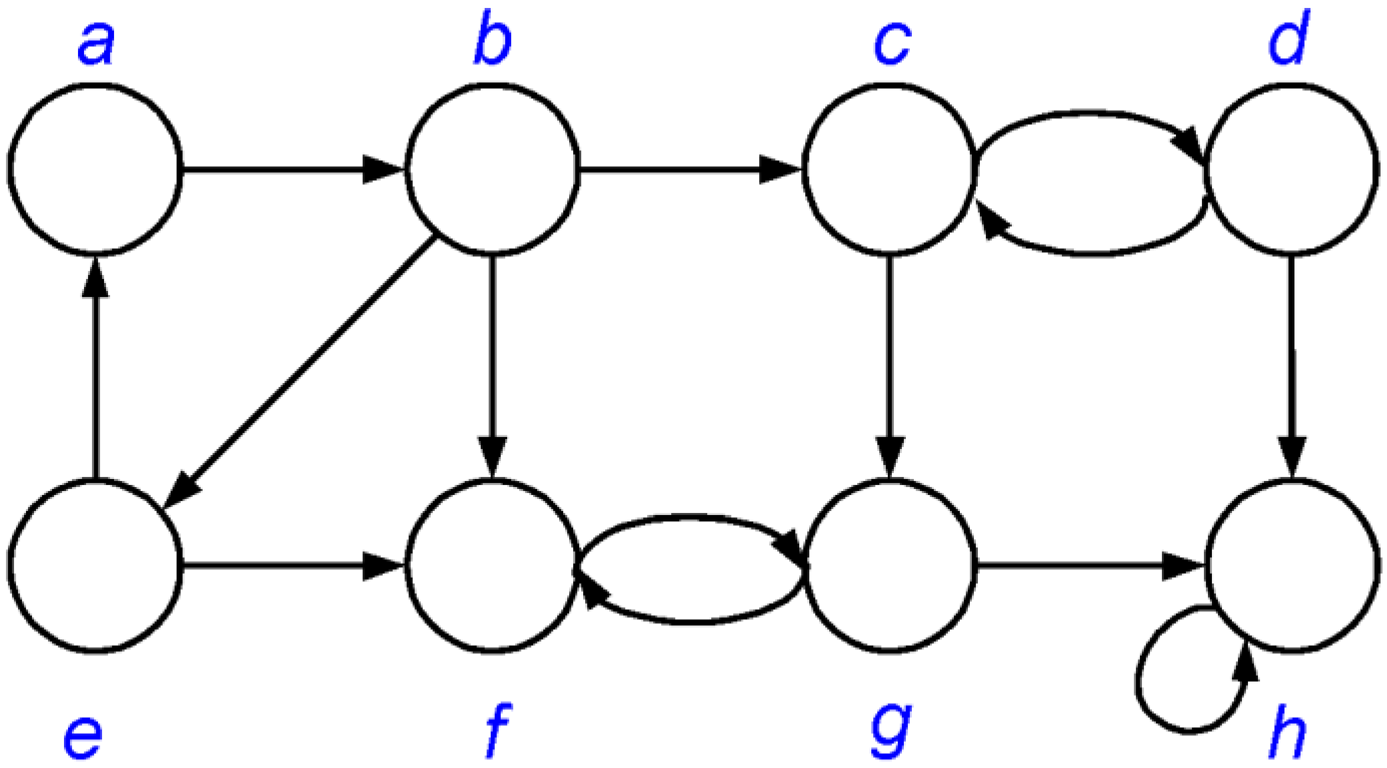
- G^T = **transpose** of directed G.
  - $G^T = (V, E^T)$, $E^T = \{(u, v) : (v, u) \in E\}$.
  - G^T is G with all edges reversed.
- Can create G^T in $\Theta(V + E)$ time if using adjacency lists.
- G and G^T have the *same* SCC's. (u and v are reachable from each other in G if and only if reachable from each other in G^T.)

Algorithm to determine SCCs

```
SCC(G)
1.  call DFS(G) to compute finishing times f[u] for all u
2.  compute G^T
3.  call DFS(G^T), but in the main loop, consider vertices in order of
    decreasing f[u] (as computed in first DFS)
4.  output the vertices in each tree of the depth-first forest formed
    in second DFS as a separate SCC
```

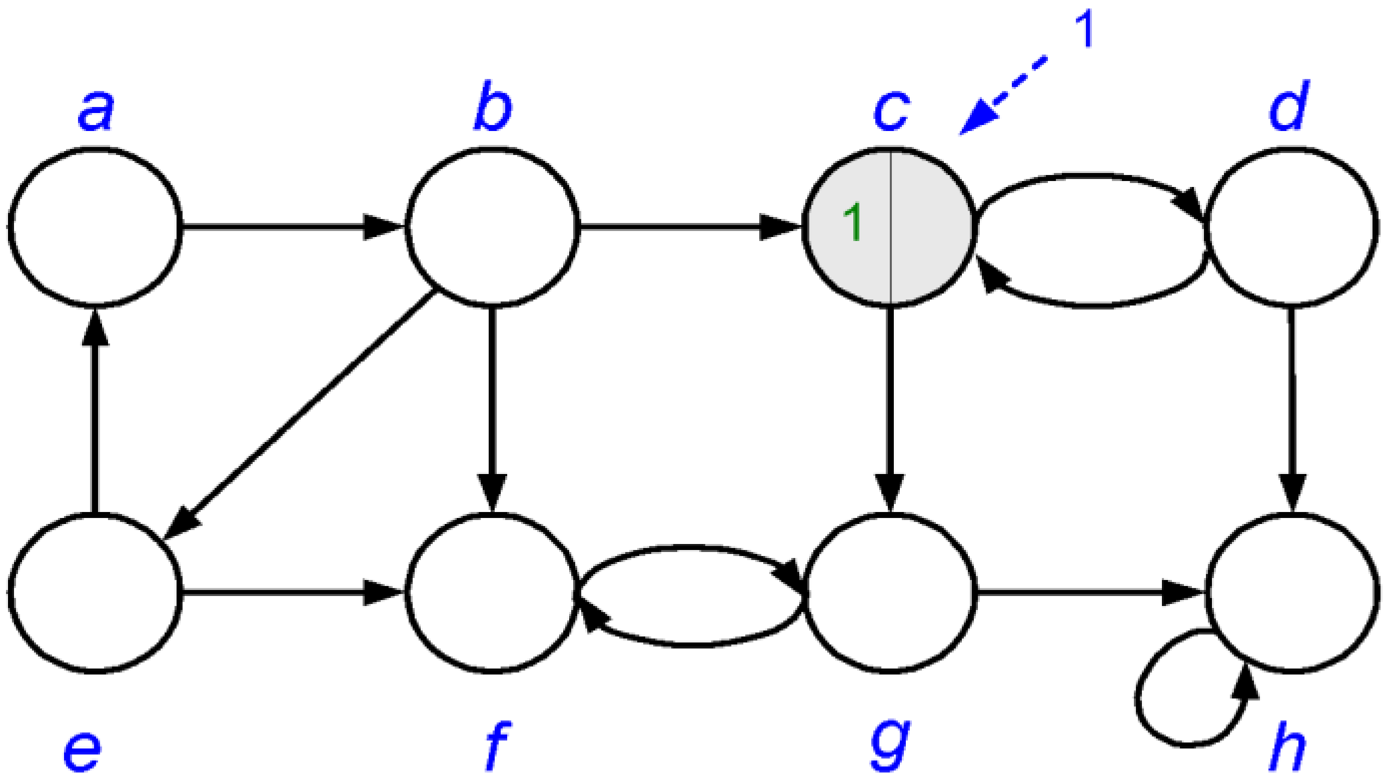
Time: $\Theta(V + E)$.

SCC: Example



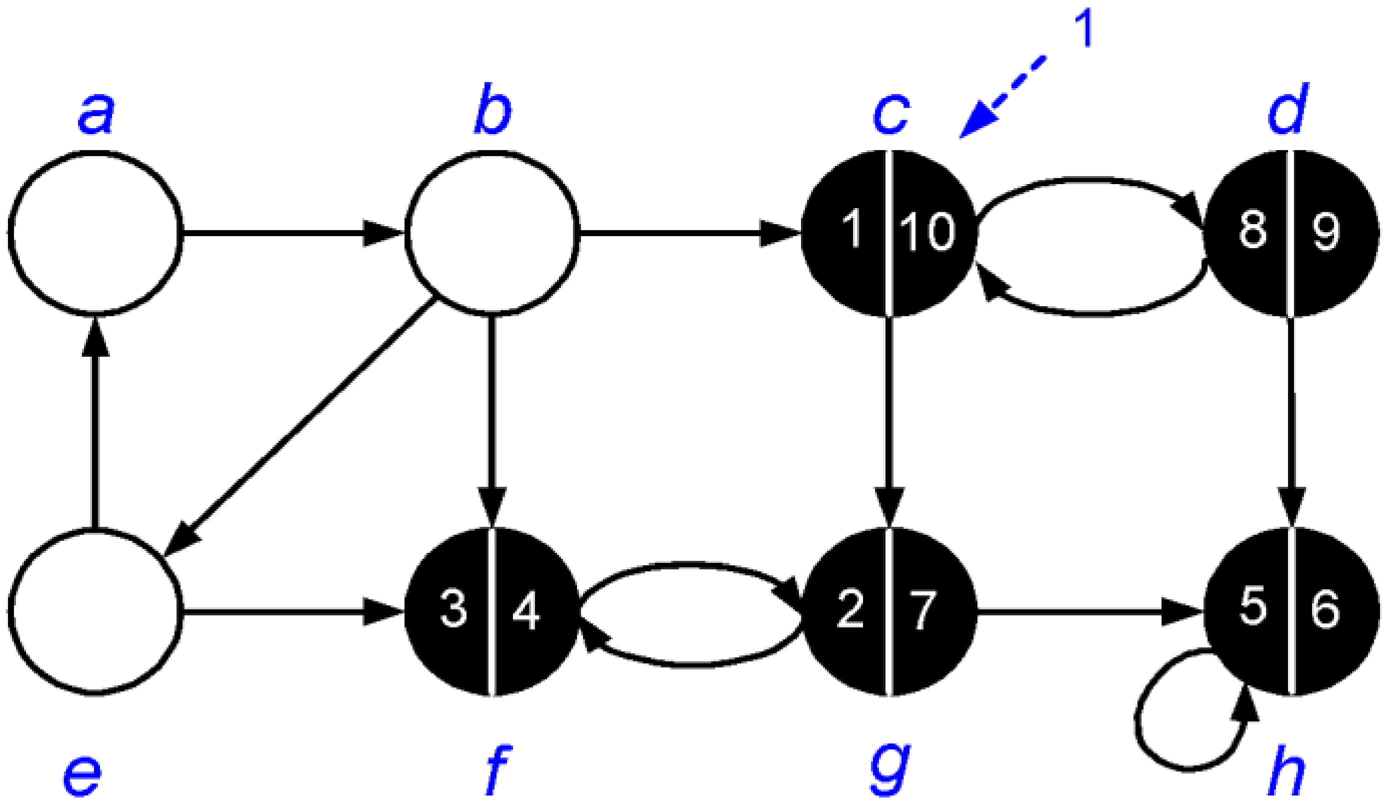
SCC: Example

(I) Run DFS(G) to compute finishing times for all u∈V



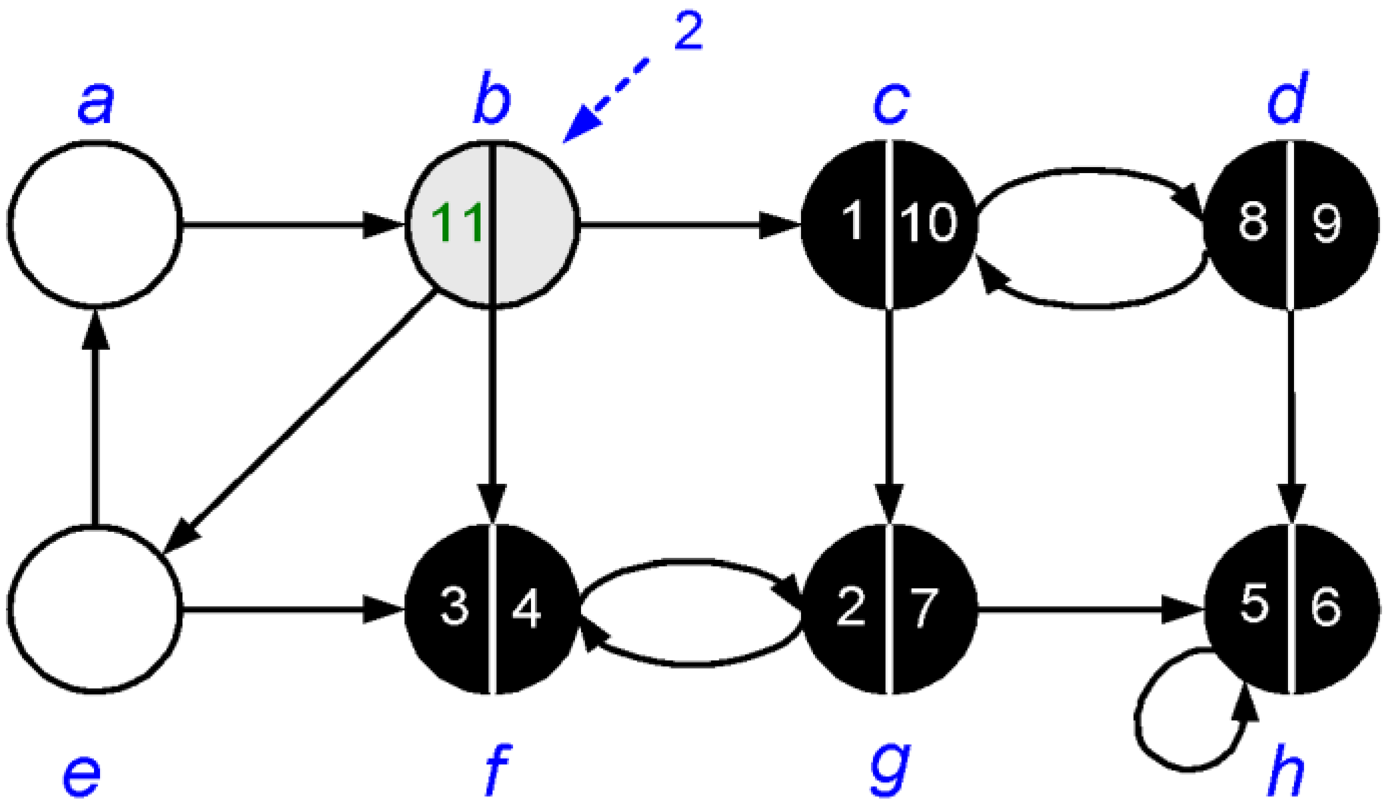
SCC: Example

(1) Run DFS(G) to compute finishing times for all u∈V

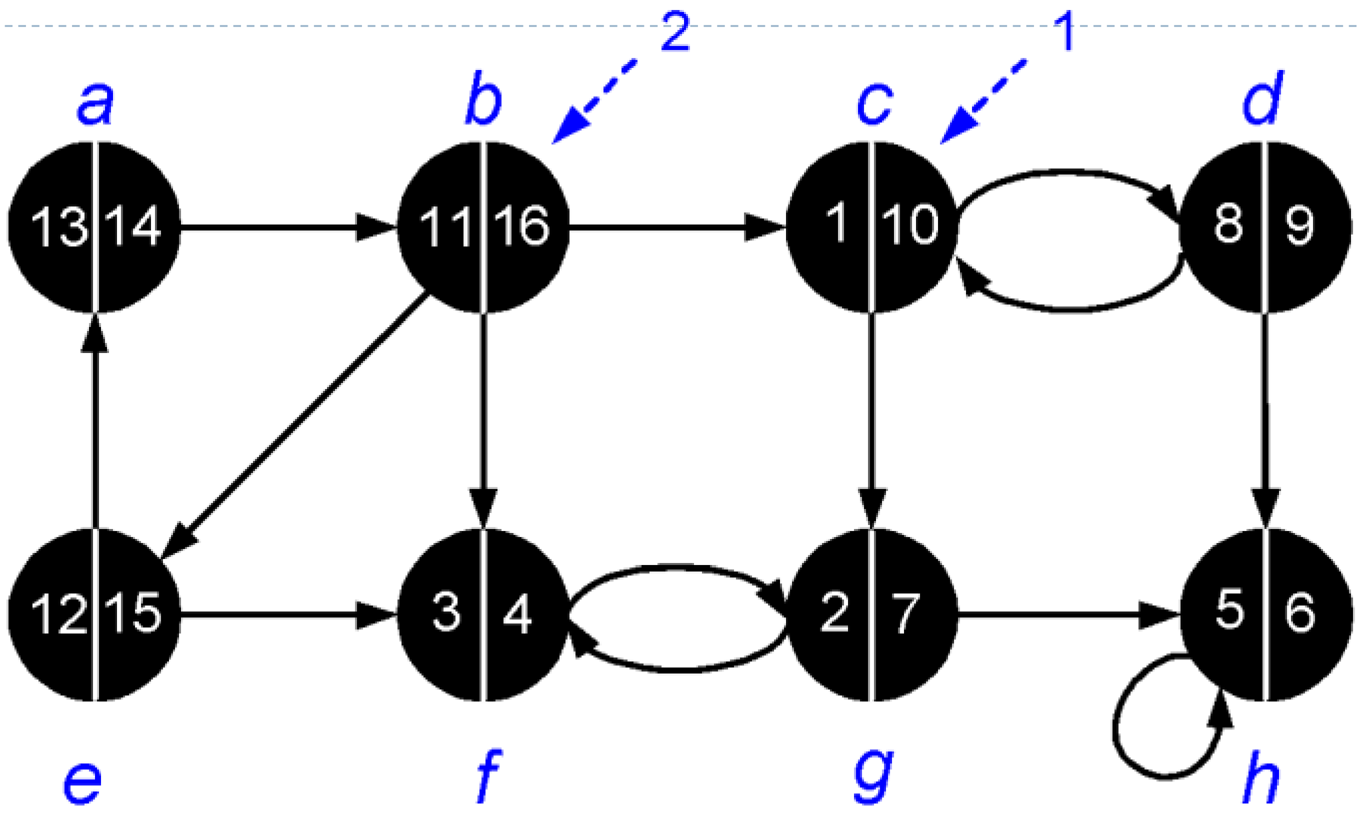


SCC: Example

(I) Run DFS(G) to compute finishing times for all u∈V



SCC: Example

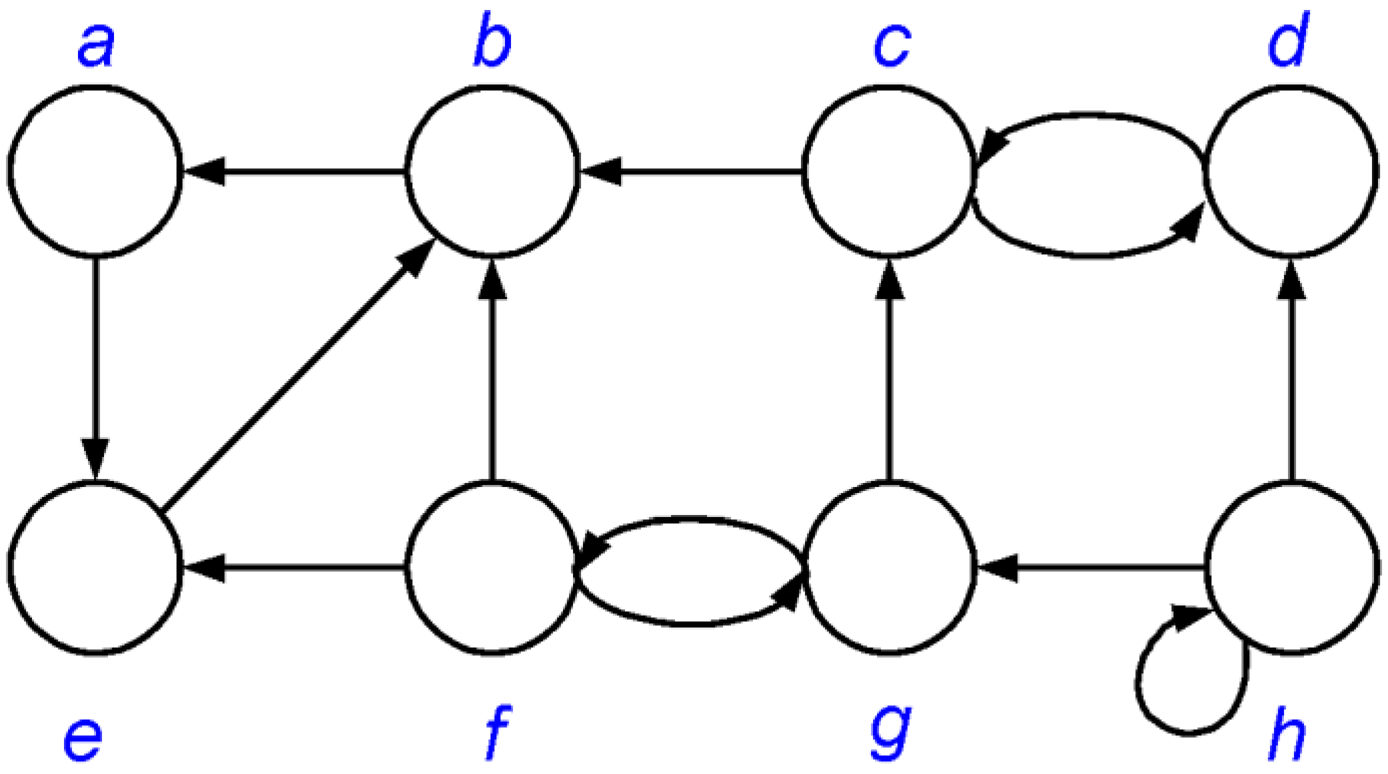


Vertices sorted according to the finishing times:

⟨b, e, a, c, d, g, h, f ⟩

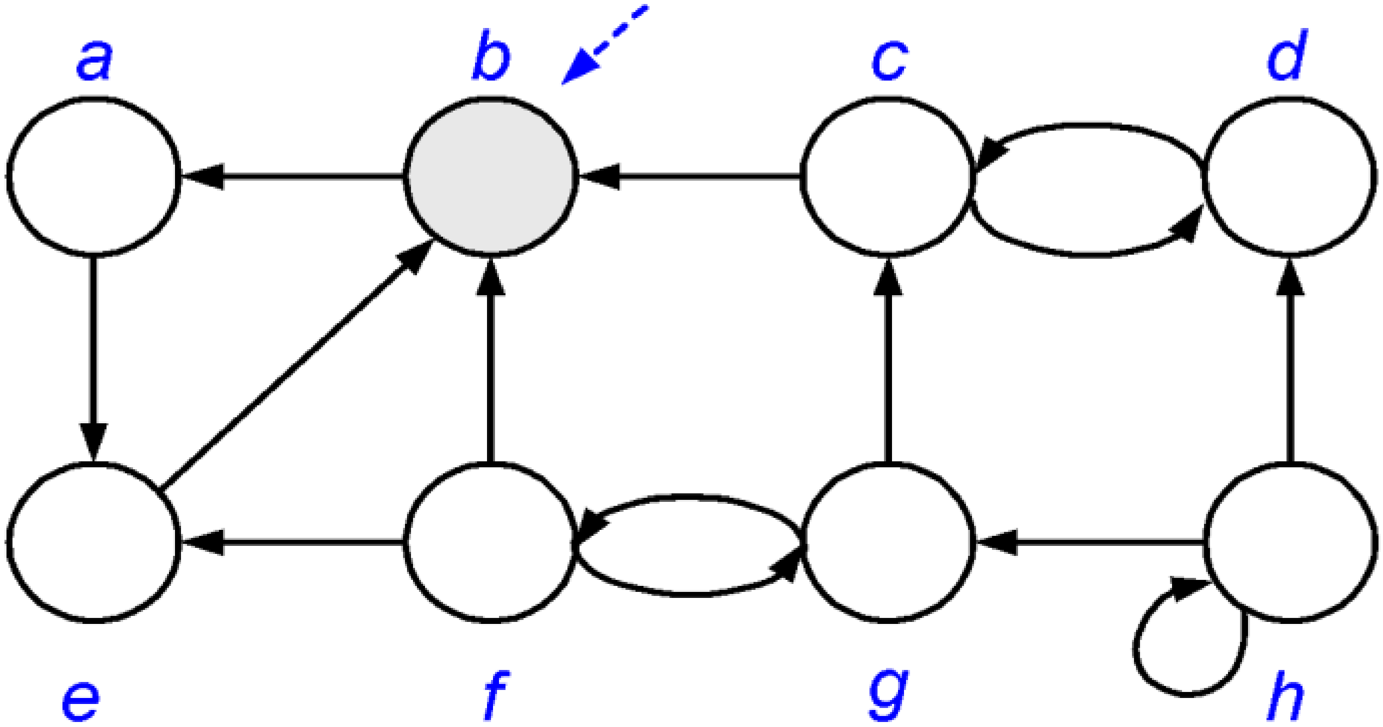
SCC: Example

(2) Compute G^T



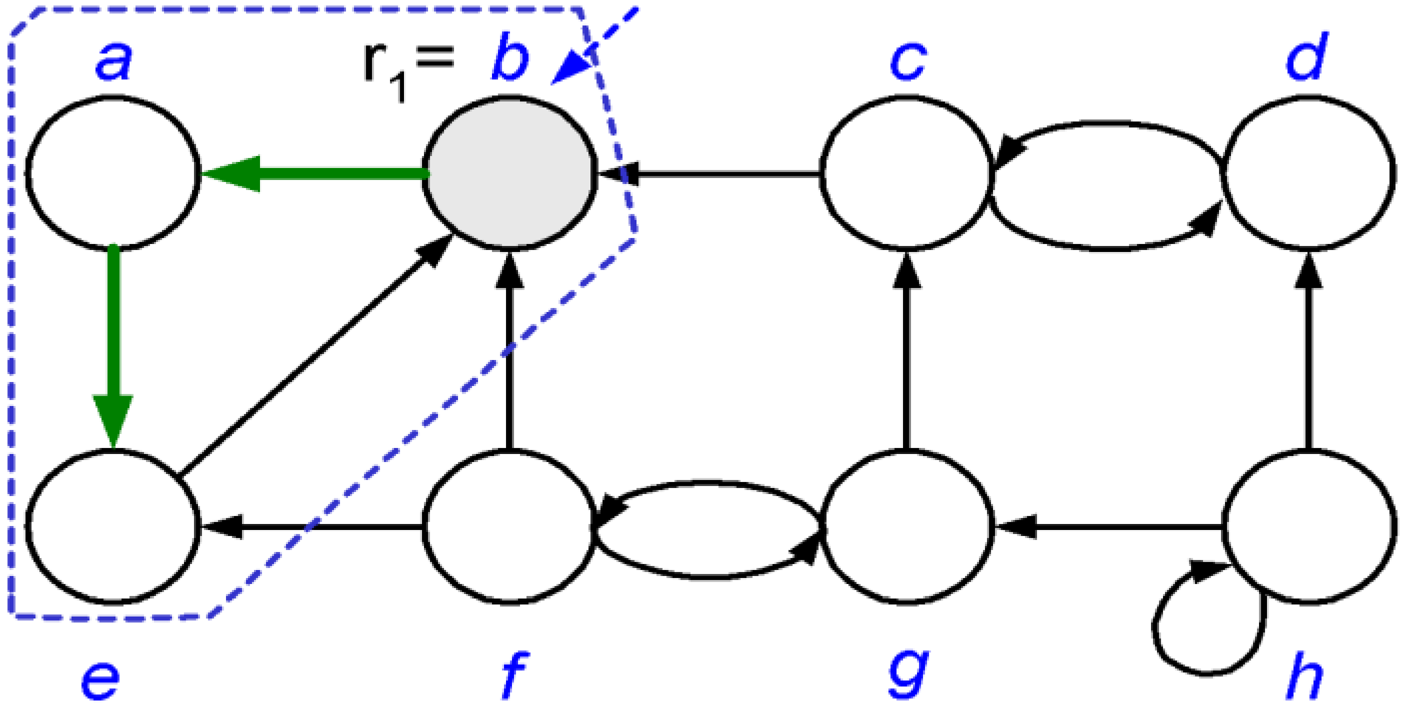
SCC: Example

(3) Call DFS(G^T) processing vertices in main loop in decreasing f[u] order: ⟨**b**, e, a, c, d, g, h, f⟩



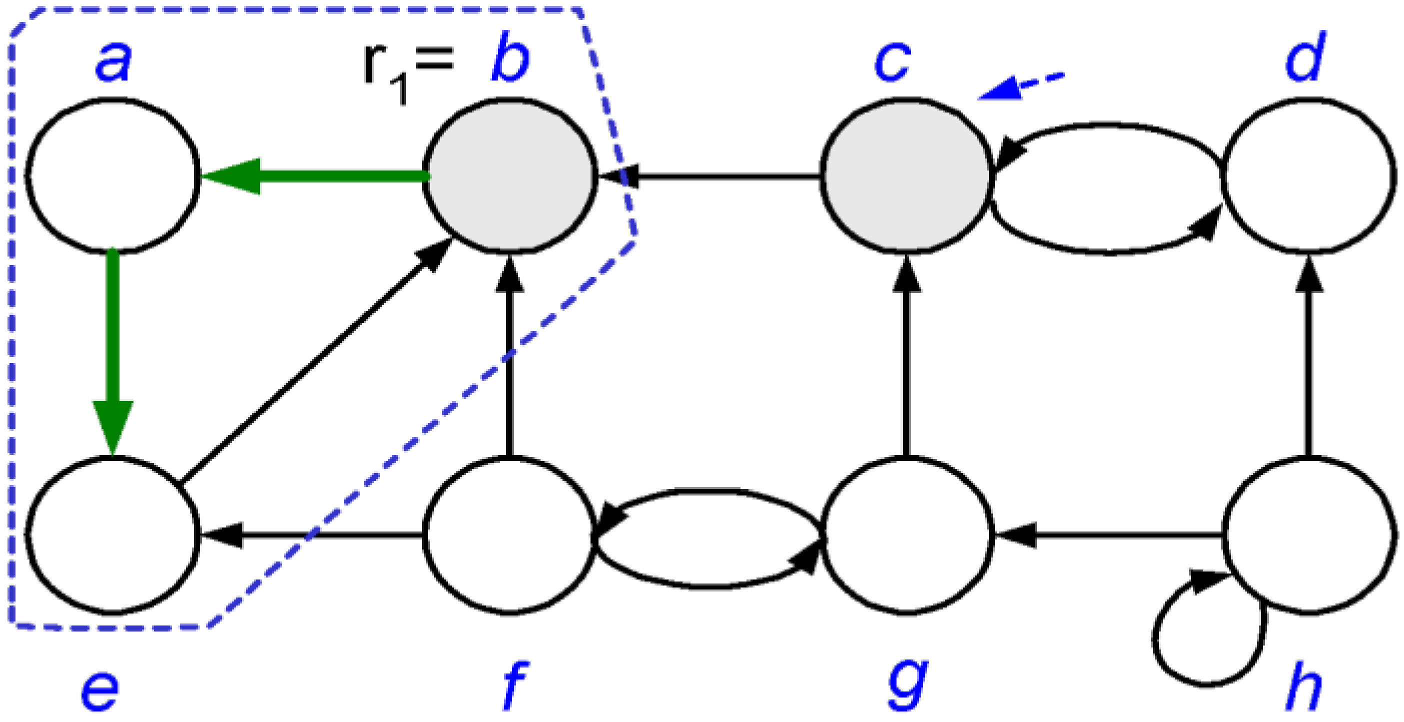
SCC: Example

(3) Call DFS(G^T) processing vertices in main loop in decreasing f[u] order: ⟨**b**, e, a, c, d, g, h, f⟩



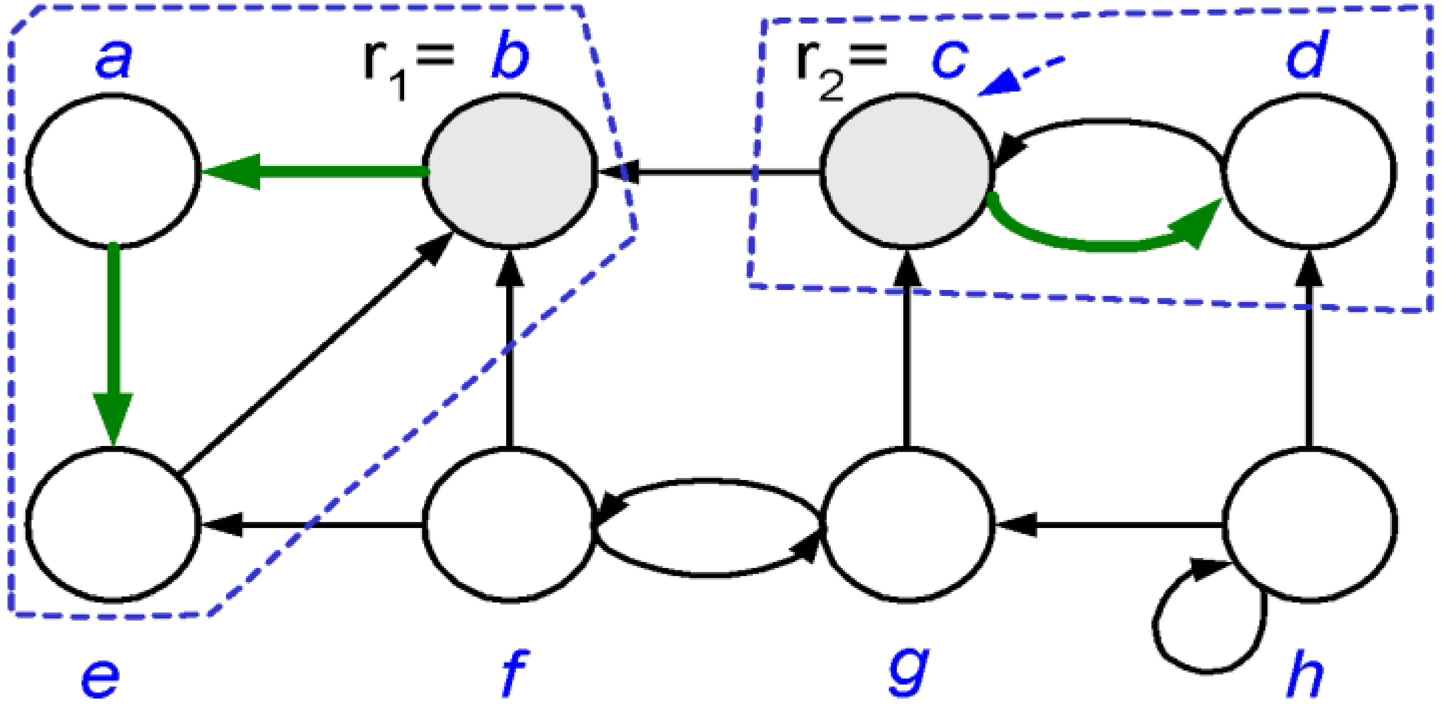
SCC: Example

(3) Call DFS(G^T) processing vertices in main loop in decreasing f[u] order: ⟨b, e, a, c, d, g, h, f⟩



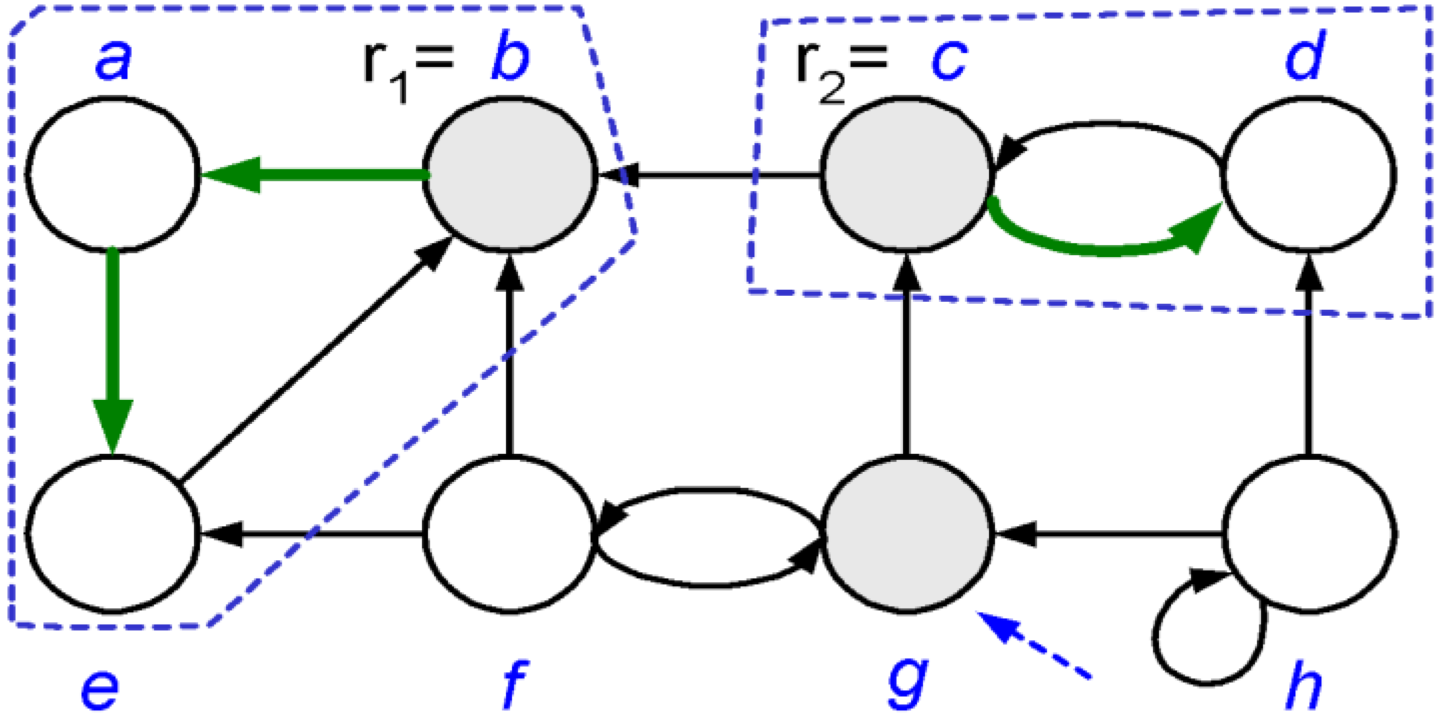
SCC: Example

(3) Call DFS(G^T) processing vertices in main loop in decreasing f[u] order: ⟨b, e, a, c, d, g, h, f⟩



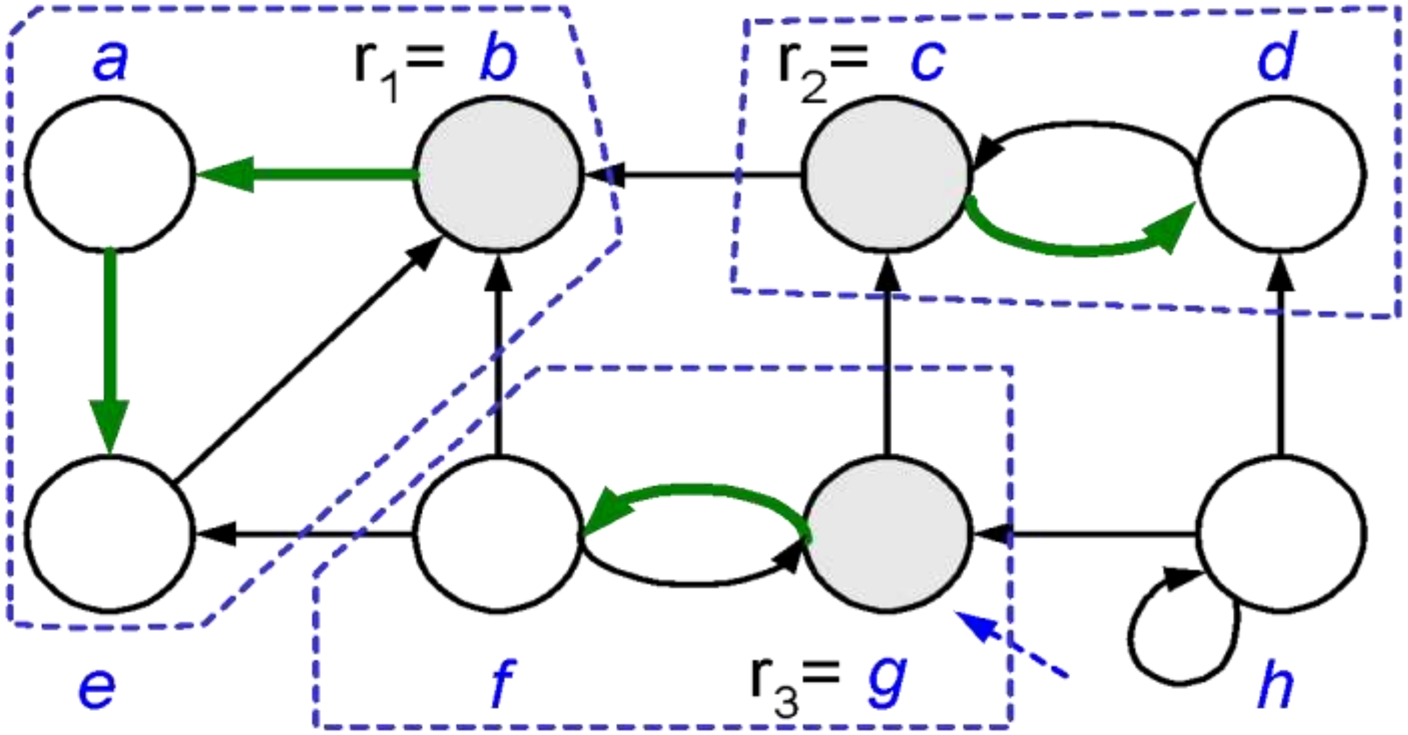
SCC: Example

(3) Call DFS(G^T) processing vertices in main loop in decreasing f[u] order: ⟨b, e, a, c, d, g, h, f ⟩



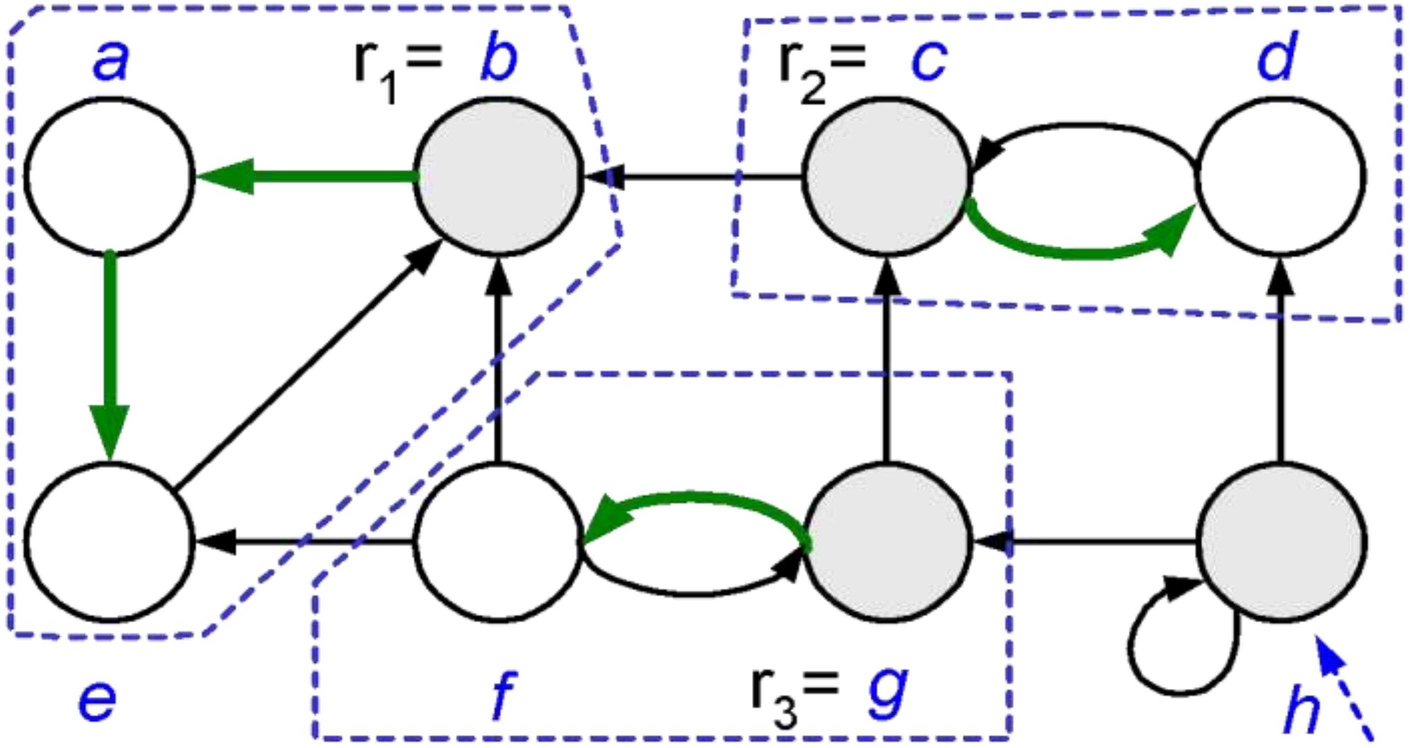
SCC: Example

(3) Call DFS(G^T) processing vertices in main loop in decreasing f[u] order: ⟨b, e, a, c, d, g, h, f⟩



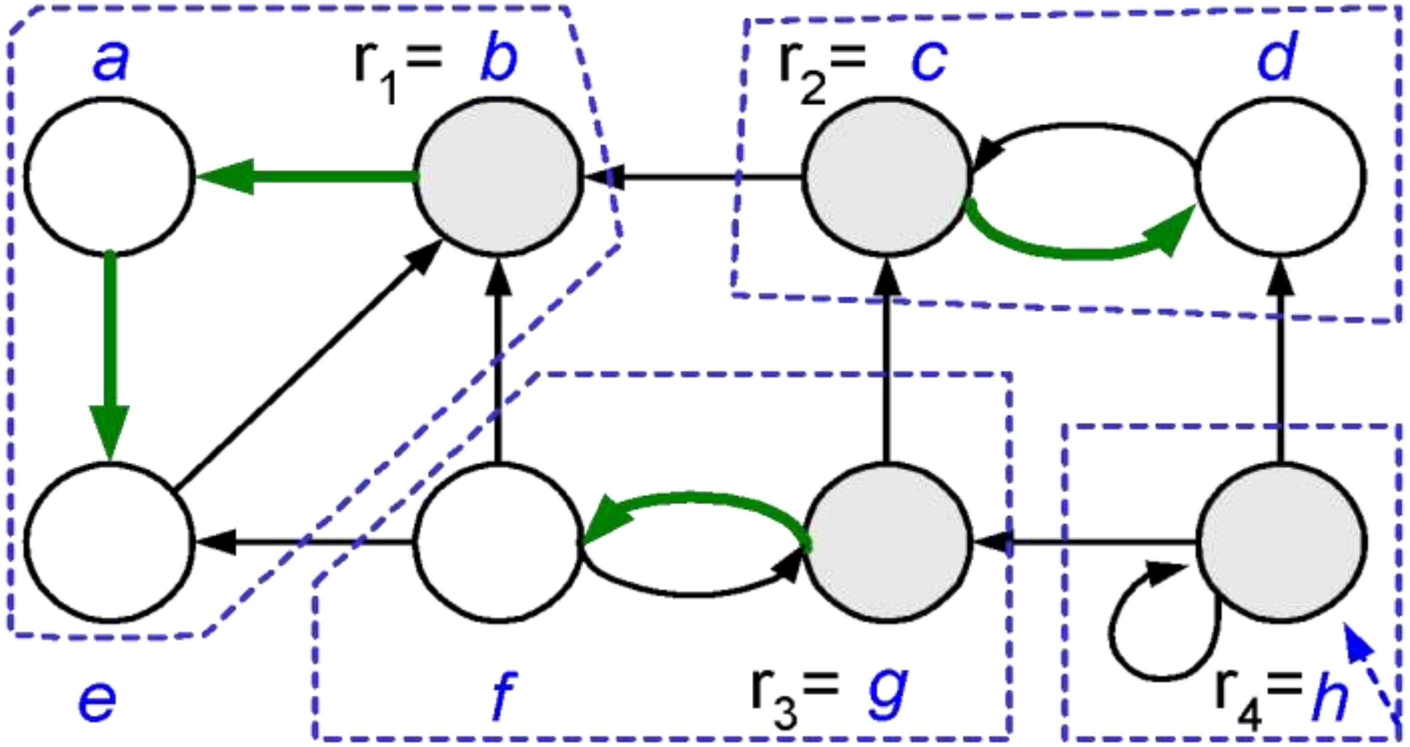
SCC: Example

(3) Call DFS(G^T) processing vertices in main loop in decreasing f[u] order: ⟨b, e, a, c, d, g, h, f⟩



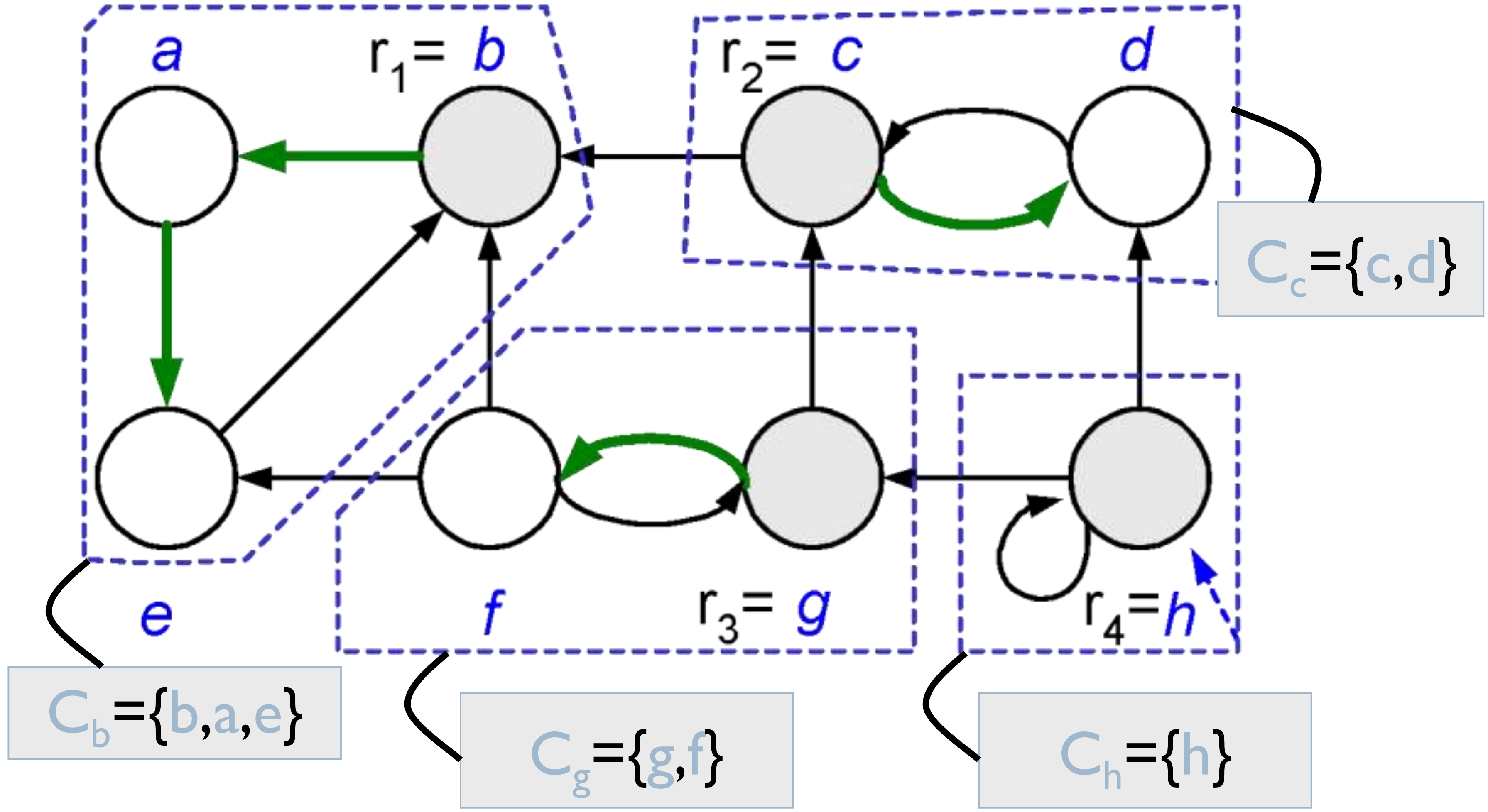
SCC: Example

(3) Call DFS(G^T) processing vertices in main loop in decreasing f[u] order: 〈b, e, a, c, d, g, **h**, f 〉

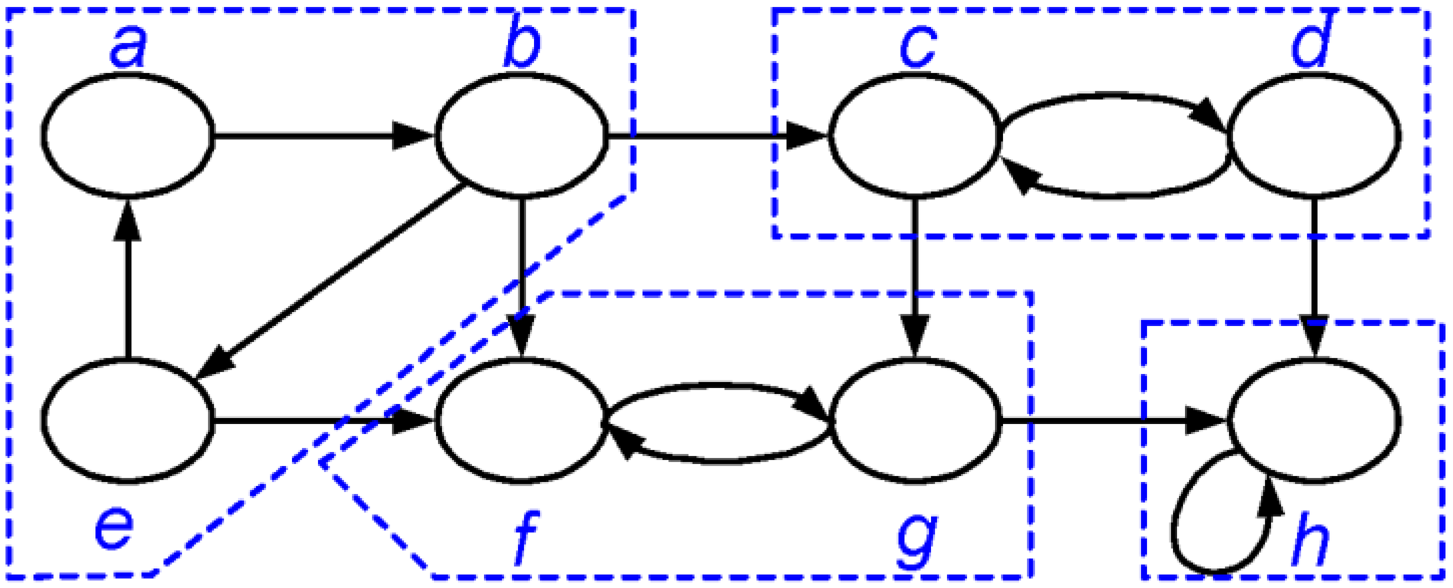


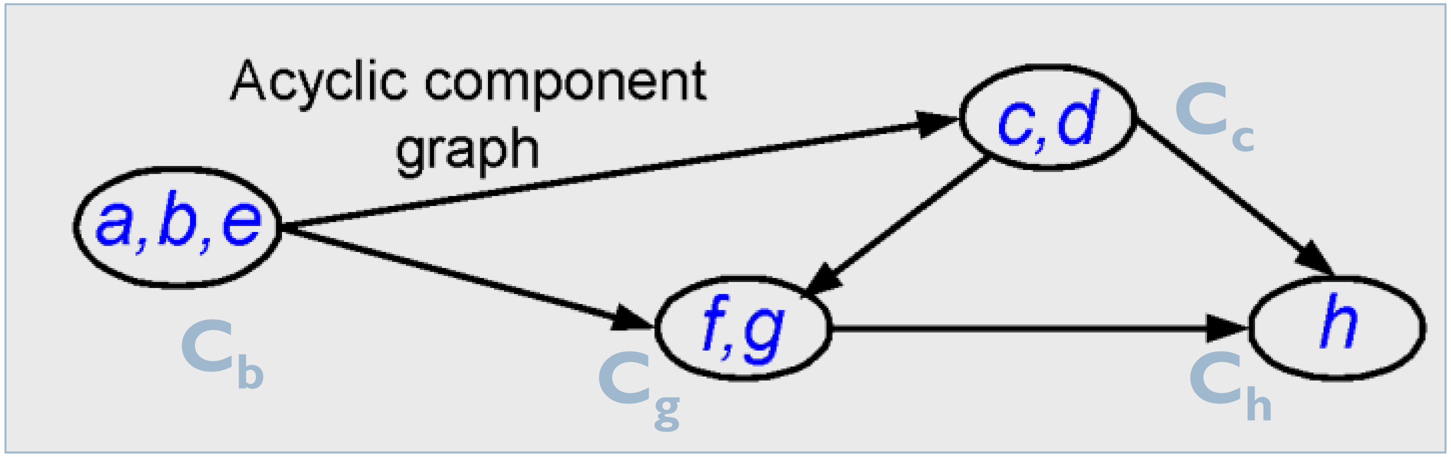
SCC: Example

(4) Output vertices of each DFT in DFF as a separate SCC



SCC: Example





How does it work?

- **Idea:**
  - By considering vertices in second DFS in decreasing order of finishing times from first DFS, we are visiting vertices of the component graph in topologically sorted order.
  - Because we are running DFS on G^T, we will not be visiting any *v* from a *u*, where *v* and *u* are in different components.