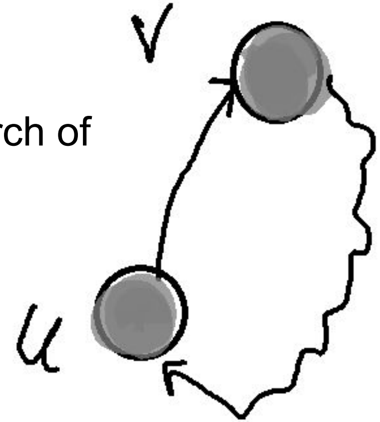
Graph Algorithms

Applications of DFS

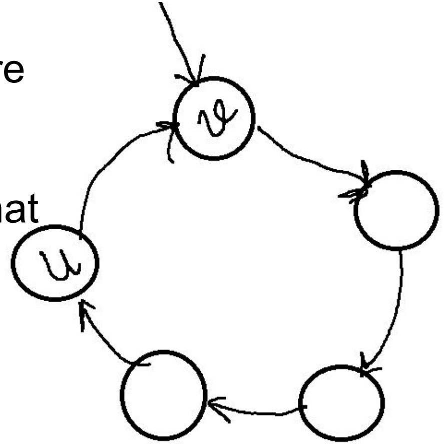
Detecting cycles in directed graphs

- **Lemma 1:** A graph G is acyclic if and only if a depth-first search of G yields no back edges
- **Equivalent to Lemma.** A directed graph has a (directed) cycle iff DFS has a back edge.

Detecting Cycles in graphs



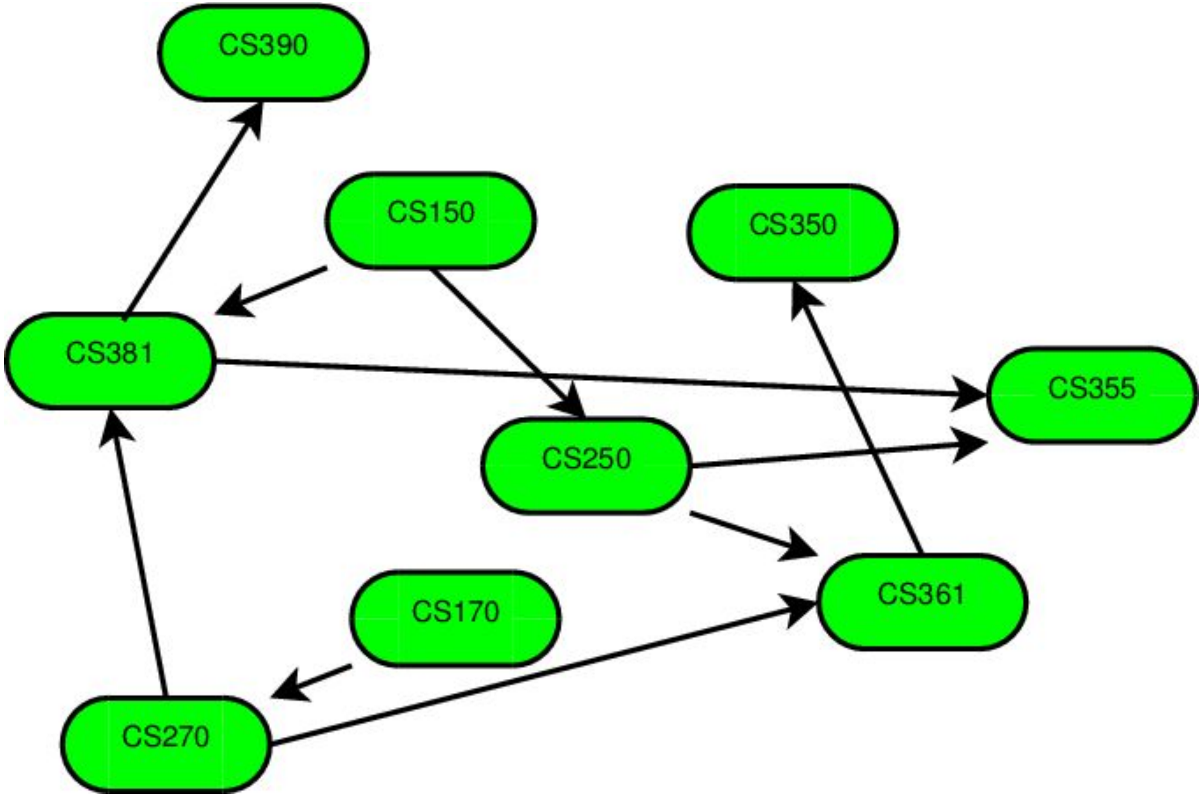
- **Lemma 1:** A graph G is acyclic if and only if a depth-first search of G yields no back edges
- **Proof** $\Rightarrow$ By contradiction.
  - Suppose DFS generates a back edge (u, v).
    - Therefore, v is an ancestor of u in the DFS tree
    - There is a path from v to u and the edge (u,v) completes the cycle→contradiction: We assumed G is acyclic
  - $\Leftarrow$ By contradiction. Suppose G contains a cycle
    - **v**: the first vertex to be discovered in the cycle
    - All the vertices on the cycle must be discovered before we finish v. When we test edge (u,v), it will be a back edge
    - **(u,v)** is a back edge → contradicts the assumption that DFS of G yields no back edge



Topological Sorting

- **Input:**
  - A Directed Acyclic Graph (DAG), G
- **Output:**
  - a linear ordering of all vertices such that if (u,v) is an edge in G, then u appears before v in the ordering
    - Linear ordering is called topological ordering
- Application: prerequisite among courses

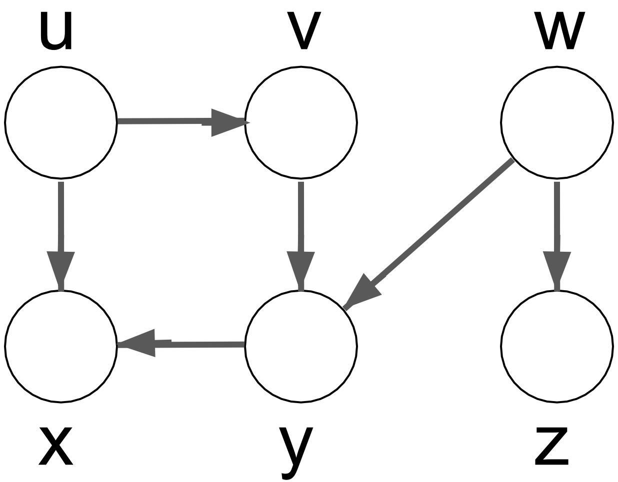
Topological Sorting: Application



Topological Sorting: First Algorithm

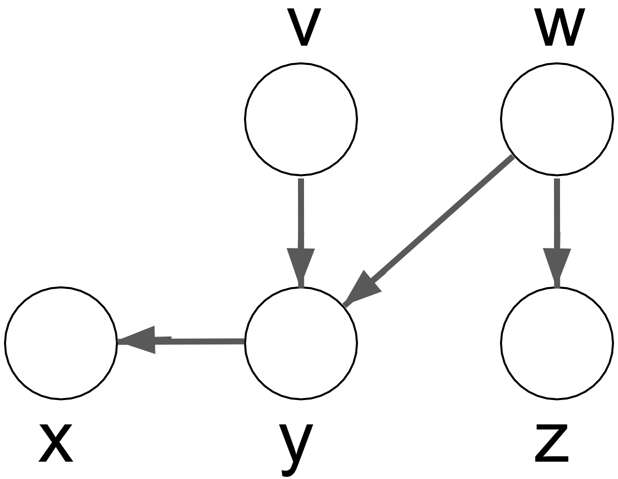
- repeatedly find a node with no incoming edges, remove it, and add it to the result
  - This algorithm is discussed in Exercise 22.4-5 of CLR (3rd edition)

Topological Sorting



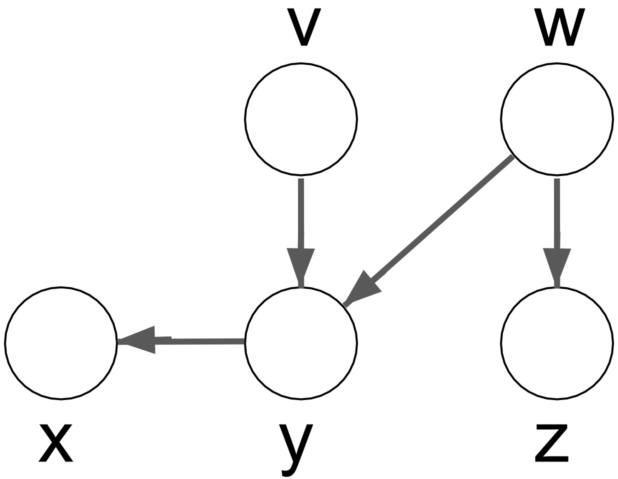
Ordered list:

Topological Sorting



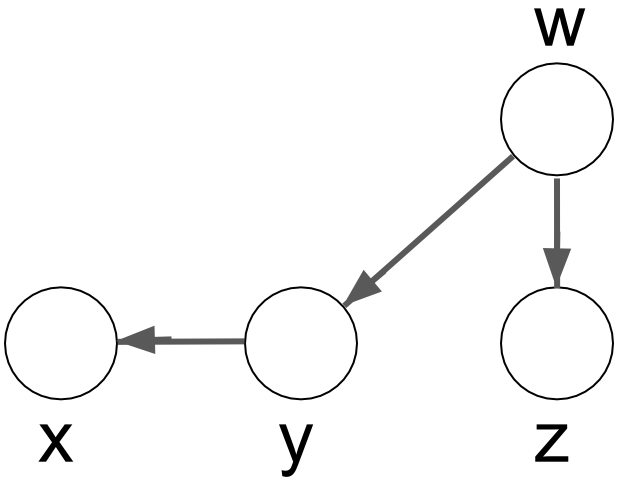
Ordered list: u

Topological Sorting



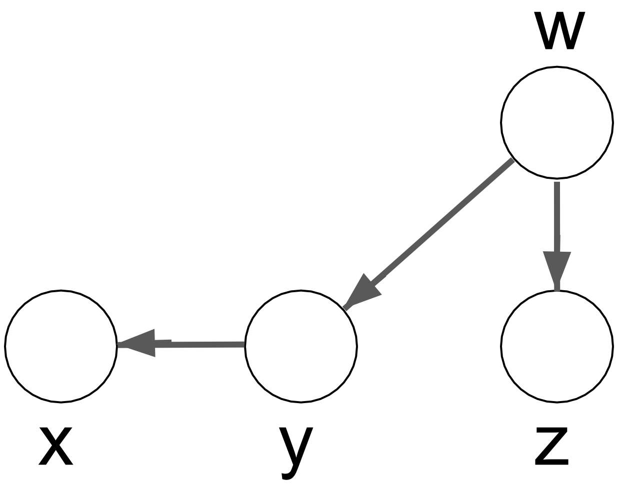
Ordered list: u v

Topological Sorting



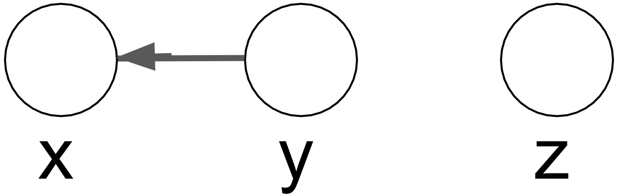
Ordered list: u v

Topological Sorting



Ordered list: u v w

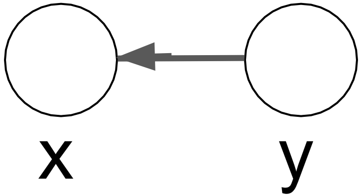
Topological Sorting



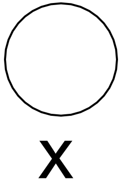
Ordered list: **u v w**

Topological Sorting

Ordered list: **u v w z**



Topological Sorting

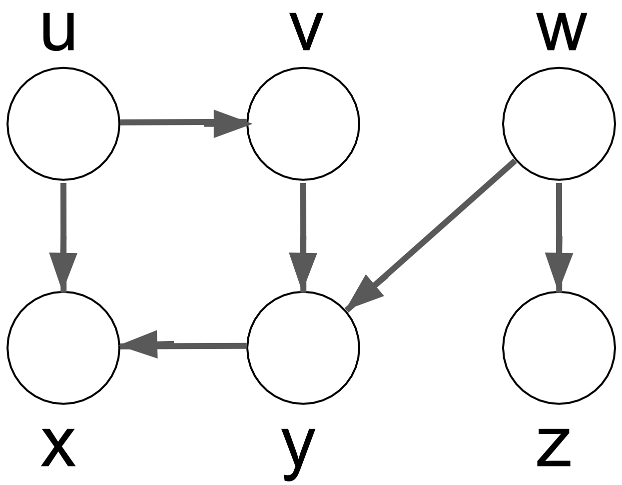


Ordered list: u v w z y

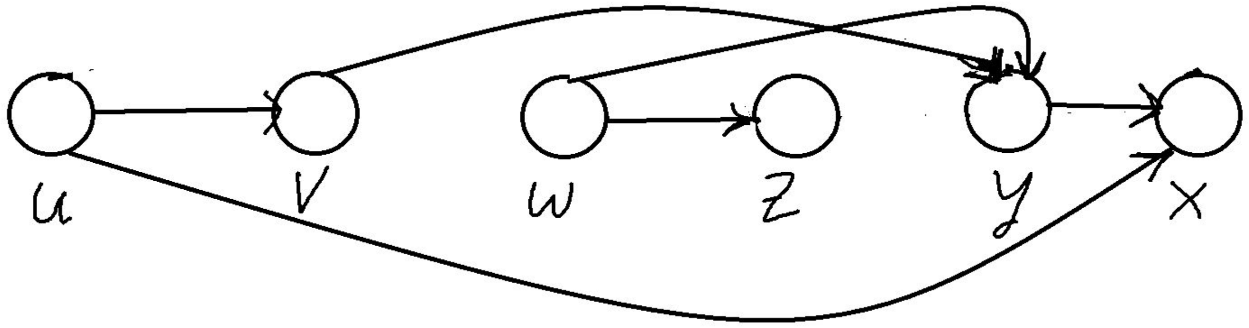
Topological Sorting

Ordered list: **u v w z y x**

Topological Sorting



Ordered list: u v w z y x



Topological Sorting: First Algorithm: Runtime

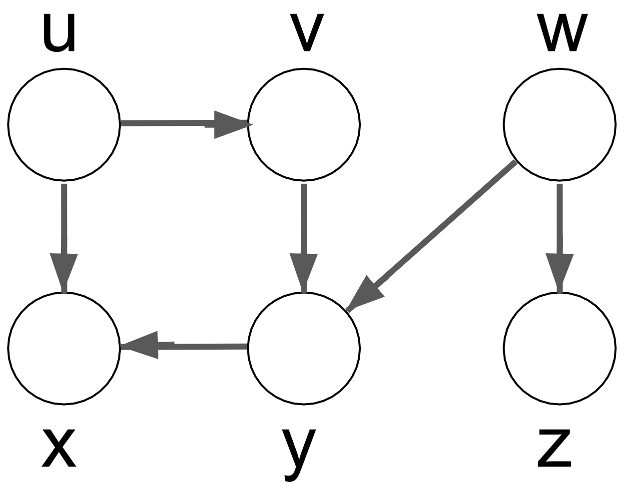
```
def topologicalSort-FirstAlg(G):
    result=[]
    while G is not empty:
        v=a vertex in G with indegree 0
        add v to result
        remove v and its edges from G
    return result
```

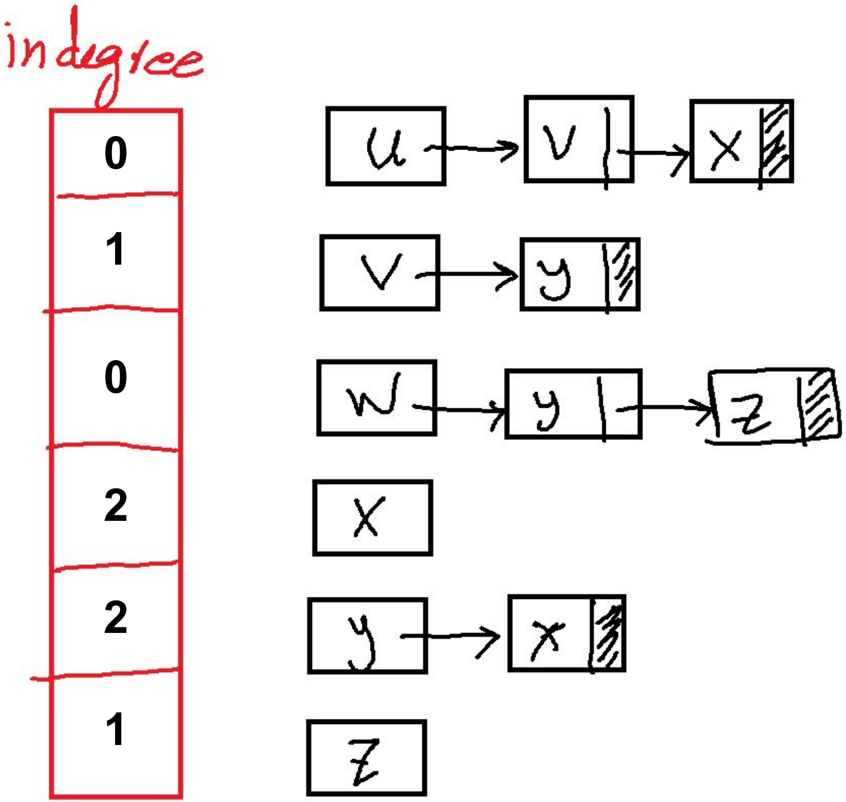
Runtime: $O(V^2+E)$

Can we do better?

- By changing the way finding for each vertex with indegree 0 is done
- Use a queue/stack to keep track of vertices with indegree 0
- Runtime: $\Theta(V + E)$

Topological Sorting: First Algorithm: Runtime





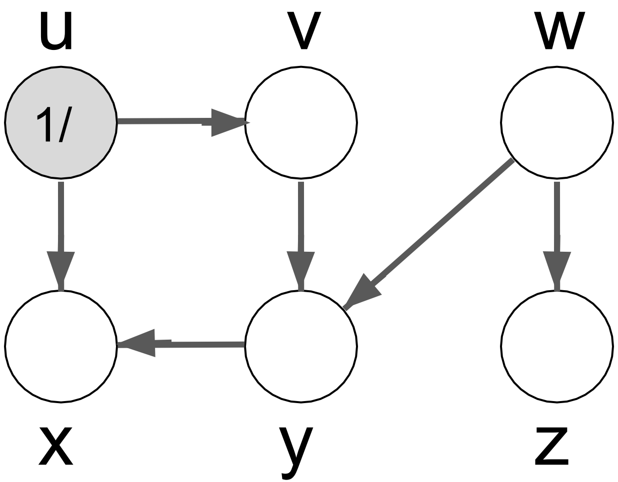
Topological Sorting: First Algorithm: proof of correctness

- Whenever a node v is added to the result, it has no incoming edges:
  - v never had any incoming edges, in which case adding v to result cannot place v out of order
  - All of the predecessors of v have already been placed into result, and v comes after all of them
- The algorithm does not get stuck, since every nonempty DAG has at least one source (a node with no incoming edges)
  - Why?

Topological Sorting: An algorithm Based on DFS

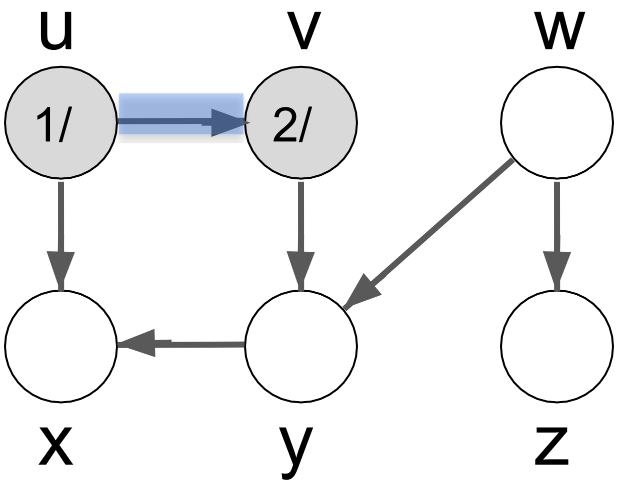
```
def TopologicalSort(G):   #G must be a DAG
    Run DFS(G) to compute finish[v] for all v ∈ V
    Output the vertices in decreasing order of their finish time
    return the linked list
```

Topological Sorting: An algorithm Based on DFS



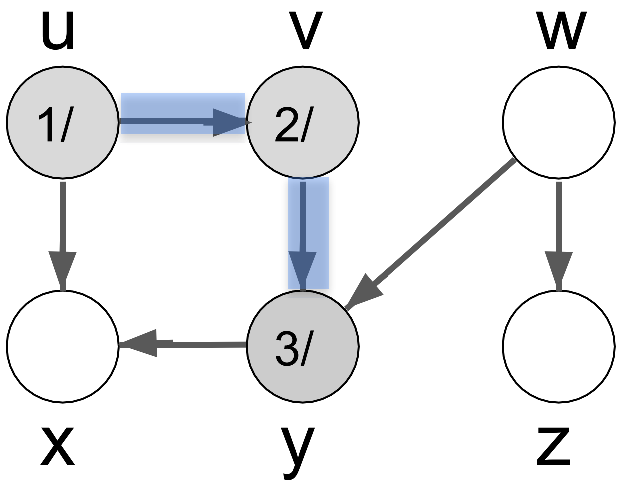
Ordered list:

Topological Sorting: An algorithm Based on DFS



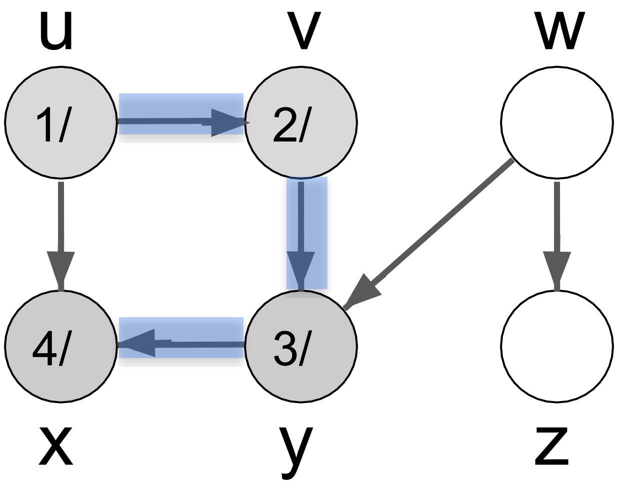
Ordered list:

Topological Sorting: An algorithm Based on DFS



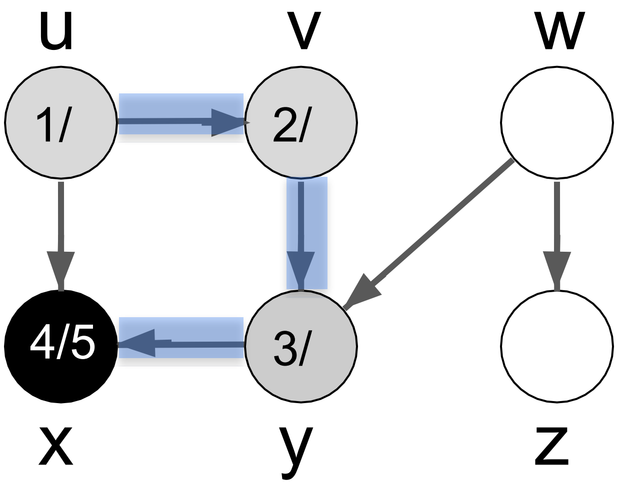
Ordered list:

Topological Sorting: An algorithm Based on DFS



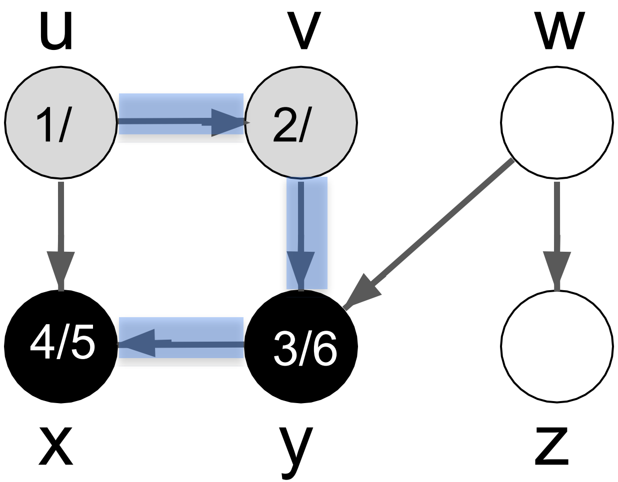
Ordered list:

Topological Sorting: An algorithm Based on DFS



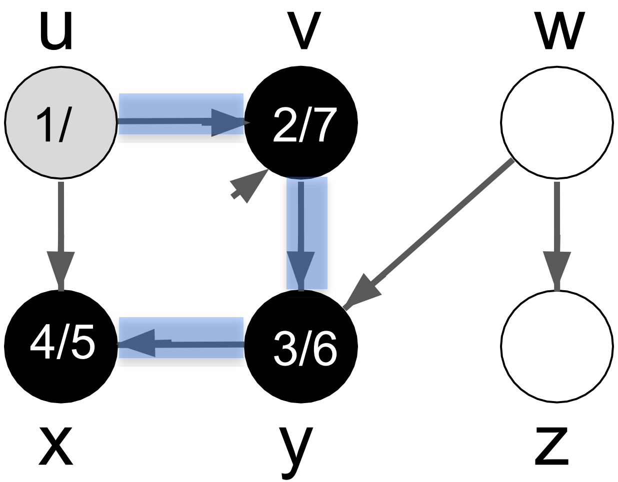
Ordered list: **x**

Topological Sorting: An algorithm Based on DFS



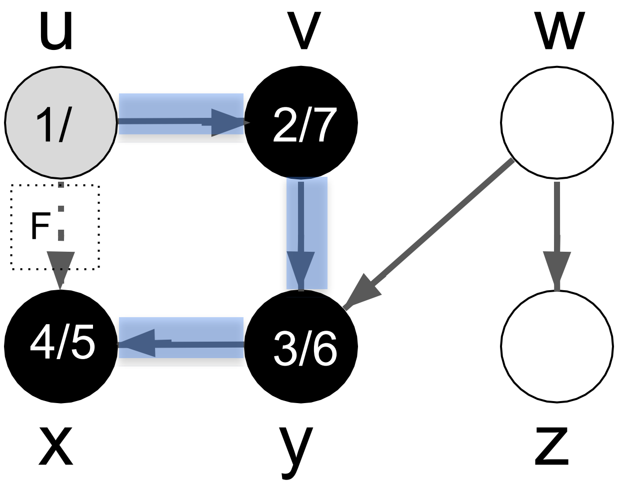
Ordered list: **y x**

Topological Sorting: An algorithm Based on DFS



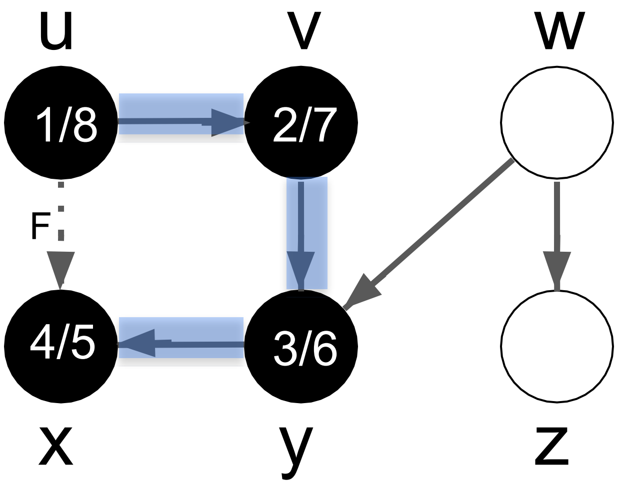
Ordered list: **v y x**

Topological Sorting: An algorithm Based on DFS



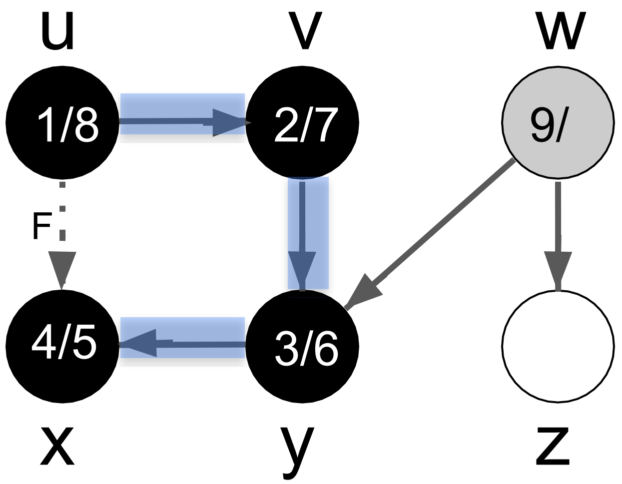
Ordered list: **v y x**

Topological Sorting: An algorithm Based on DFS



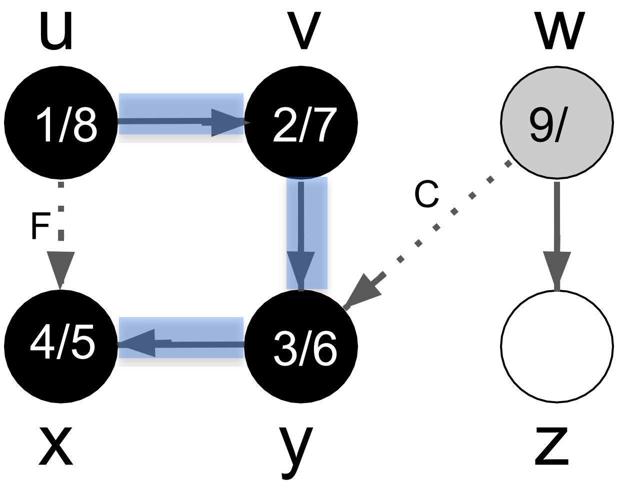
Ordered list: u v y x

Topological Sorting: An algorithm Based on DFS



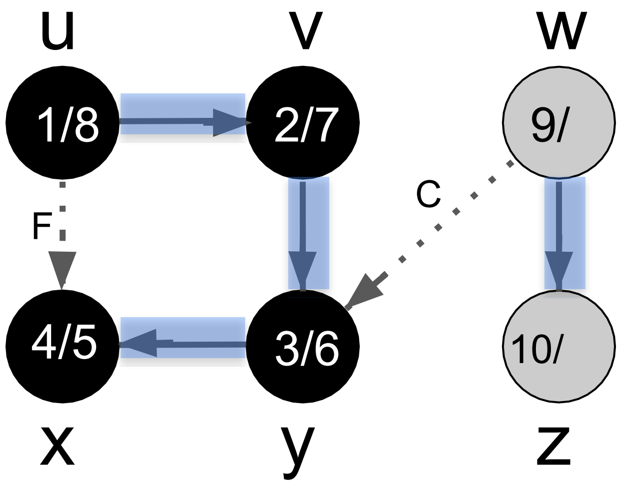
Ordered list: u v y x

Topological Sorting: An algorithm Based on DFS



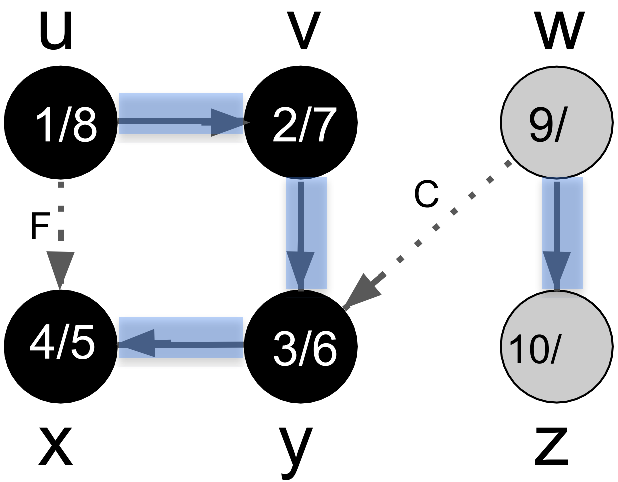
Ordered list: u v y x

Topological Sorting: An algorithm Based on DFS



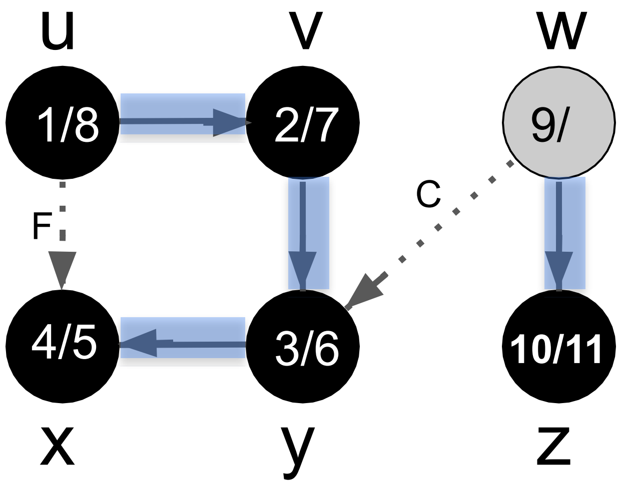
Ordered list: u v y x

Topological Sorting: An algorithm Based on DFS



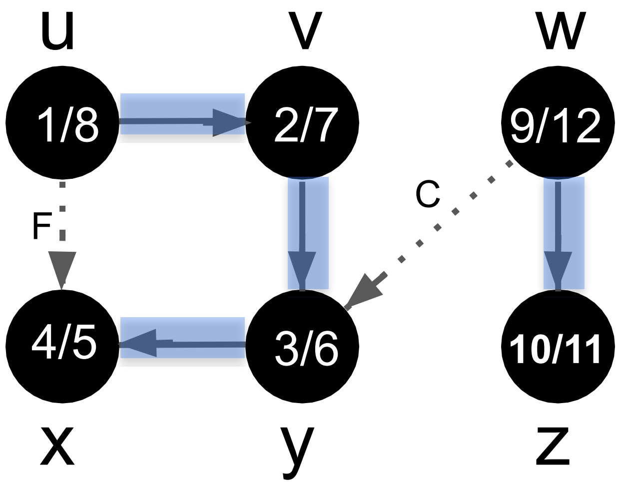
Ordered list: **u v y x**

Topological Sorting: An algorithm Based on DFS



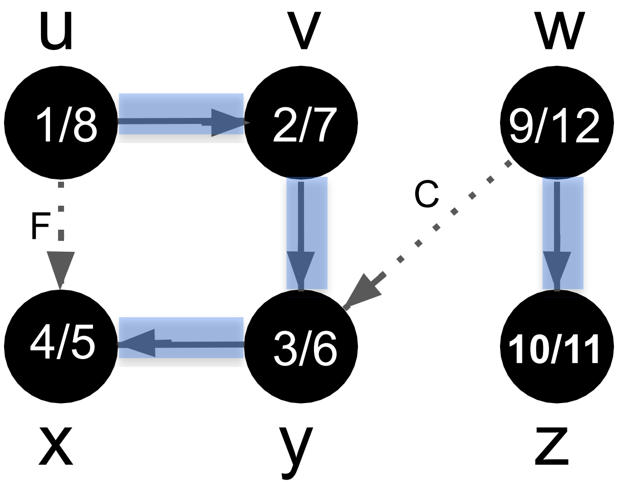
Ordered list: **z u v y x**

Topological Sorting: An algorithm Based on DFS



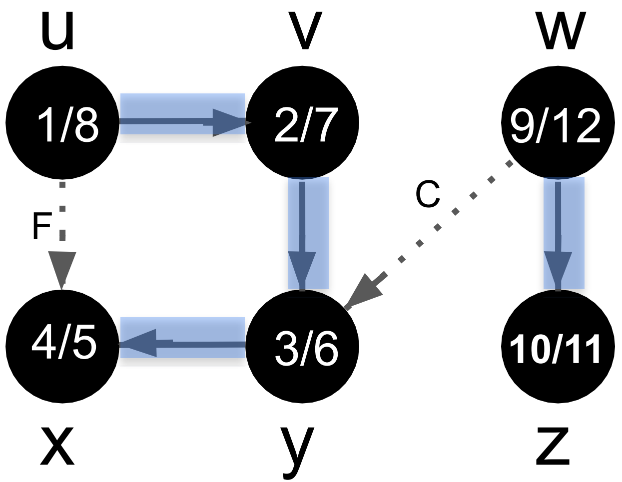
Ordered list: **w z u v y x**

Topological Sorting: An algorithm Based on DFS

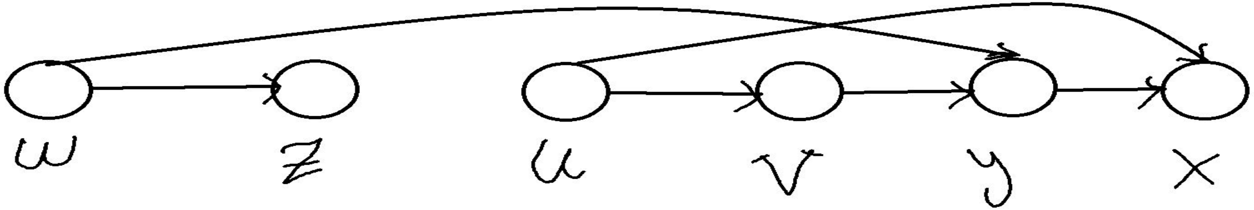


Ordered list: **w z u v y x**

Topological Sorting: An algorithm Based on DFS

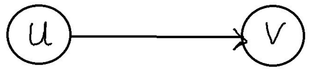


Ordered list:



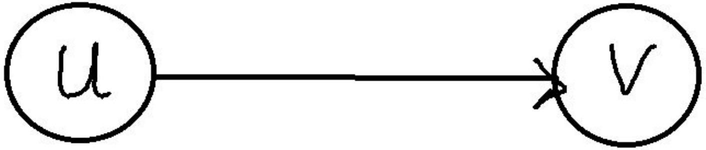
DFS-based topological Sorting: proof of correctness

- For any pair of distinct vertices u and v
  - if **(u,v) ∊ G** then **finish[v]<finish[u]** → **u** appears before **v** in the ordering
- **Proof.** Consider any edge (u, v). There are two cases:
  - **d[u] < d[v]**   DFS discovers u before v
  - When exploring v, v cannot be gray, since then v would be an ancestor of u and it means there is a cycle in the graph while we have an acyclic graph.
  - Therefore v is either
    - WHITE
      - Vertex v becomes a descendant of u → f[v] < f[u]
    - BLACK
      - It has been finished, but u is yet to be finished → f[v] < f[u]
  - **d[v] < d[u]**
  - Since the graph is acyclic, u is not reachable from v
  - →u cannot be a descendant of v
  - By the parenthesis property, the intervals [d [v], f[v]] and [d[u], f[u]] must be disjoint.
  - The only possibility left is d[v] <f[v] < d[u] < f[u]



finish[u] > finish[v]

Since the graph has no cycle u is an ancestor of v



finish[u] > finish[v]

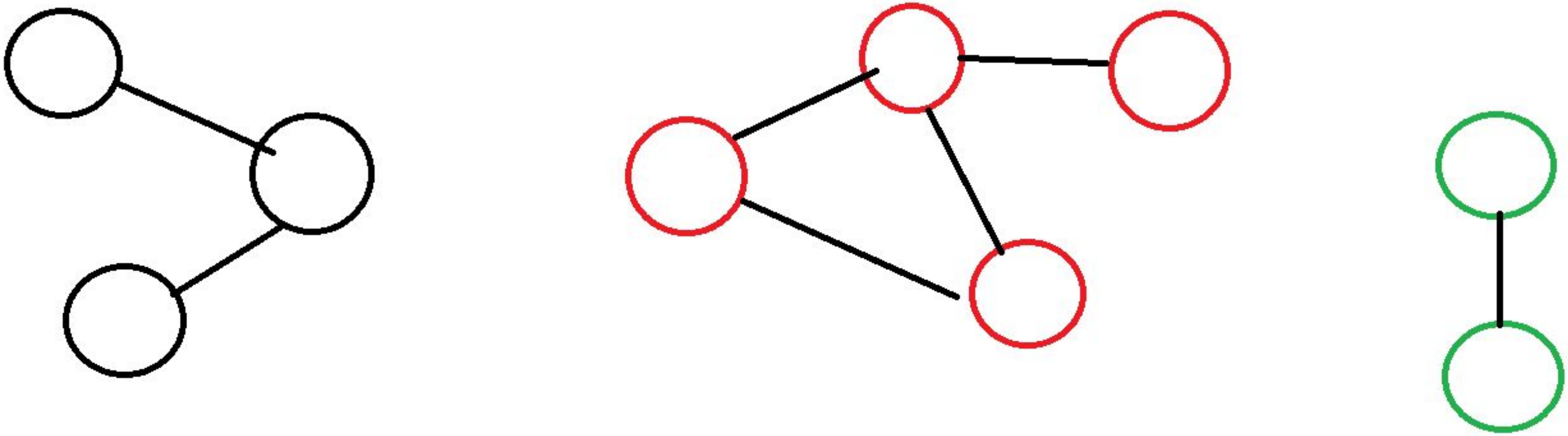
$d[u] < d[v] < f[v] < f[u]$

Strongly Connected Graphs
Strongly Connected Components

Connected Components: Undirected Graph

- In an undirected graph $G = (V, E)$ two vertices $u, v \in V$ are **connected** iff there is a path from u to v
- An undirected graph is **connected** if every vertex is reachable from all other vertices
- A **connected component** of G is a set $C \subseteq V$ which has the following properties:
  - C is nonempty
  - For any $u, v \in C$: u and v are connected
  - For any $u \in C$, $v \in V - C$: u and v are not connected

Connected Components: Undirected Graph



Connected Components: Undirected Graph

- How to find connected components in an undirected graph?
    - Using DFS
        - DFS(G, u) finds all nodes reachable from u in the graph

Strongly Connected Components

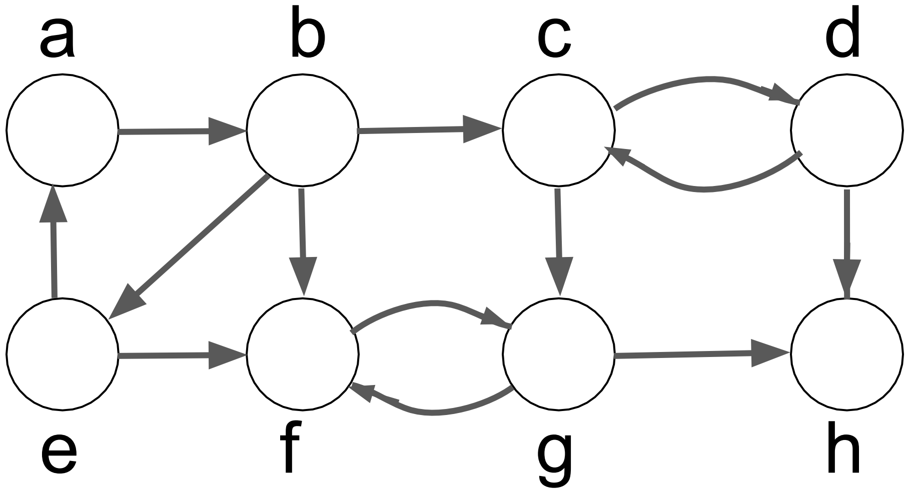
Strongly Connected Components

- In a **directed graph** G
  - v is reachable from u iff there is a path from u to v.
- In an undirected graph, if there is a path from u to v, there is also a path from v to u.
- In a directed graph, it is possible for v to be reachable from u, but for u not to be reachable from v.
- How would we generalize the idea of a connected component to a directed graph?

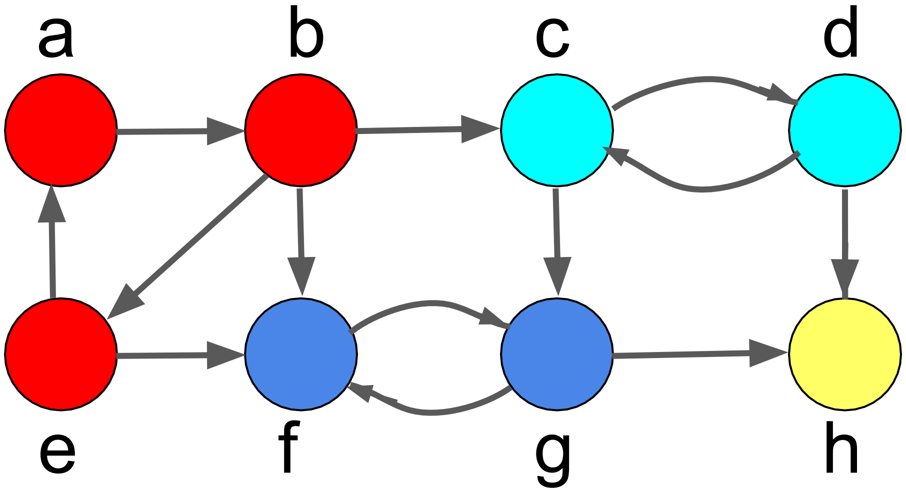
Strongly Connected Components

- Let $G = (V, E)$ be a directed graph
- Two vertices $u \in V$ and $v \in V$ are **strongly connected** iff v is reachable from u and u is reachable from v
- A **directed graph is strongly connected** if and only if every pair of vertices is strongly connected.
- A **strong connected component (**or **SCC)** of G is a maximal strongly connected subgraph of G.
- A **SCC** of G is a set $C \subseteq V$ such that:
  - C is not empty
  - For any $u, v \in C$: u and v are strongly connected
  - For any $u \in C$ and $v \in V - C$: u and v are not strongly connected.

Strongly Connected Components



Strongly Connected Components



Strongly Connected Graphs

Input: A directed graph $G = (V, E)$

Output: Yes if G is strongly connected; no otherwise

Brute-force solutions:

- For each pair u,v check whether there is a path from u to v, v to u
  - Runtime: $O(n^2(n+m))$
- For each vertex v, whether all vertices can be reached from v
  - Runtime: $O(n(n+m))$

- What if the graph was undirected?

Strongly Connected Graphs: Observation

Lemma. G is strongly connected if and only if every vertex v is reachable from s and s is reachable from every vertex v, where s is an arbitrary vertex

- ⇒ by the definition of a strongly connected component
- ⇐ For any u,v ∈ V, we obtain a path from u to v by combining a path from u to s and a path from s to v →G is strongly connected

- How do we check whether s is reachable from every vertex $v \in V$?
  - Idea: Reverse the graph
  - Claim: Given $G = (V, E)$, we reverse the direction of all the edges to obtain $G^{\mathsf{T}} = (V, \overleftarrow{E})$. Then, there is a path from v to s in G if and only if there is a path from s to v in G^{T}. So, s is reachable from every $v \in V$ in G if and only if every $v \in V$ is reachable from s in G^{T}
- Example

Strongly Connected Graphs: Algorithm

- Check whether all vertices in G are reachable from s by one DFS
- Reverse the direction of all the edges in G to obtain G^T
- Check whether all vertices in G^T are reachable from s by one DFS
- If both yes, return “SC” graph, otherwise, return “not SC”

Runtime: O(m+n)

Strongly Connected Components

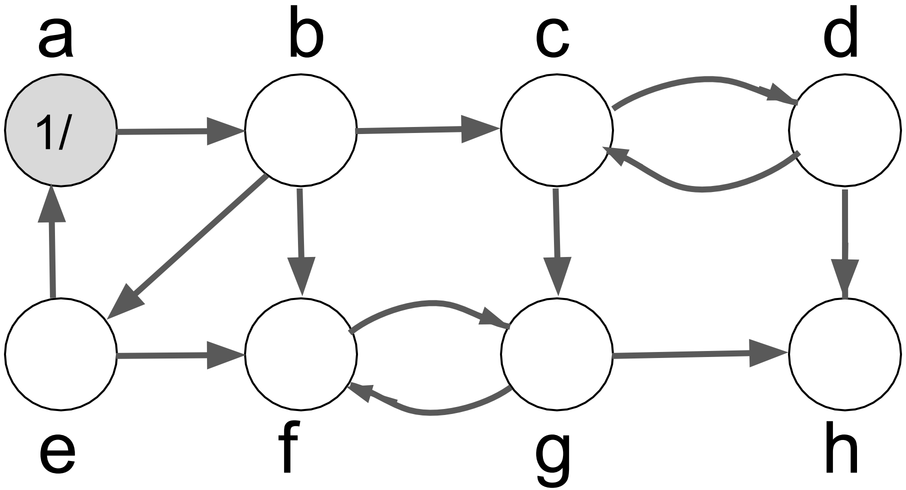
Strongly Connected Components

- Brute-force:
    - Consider all possible subset of vertices and check using the previous algorithm
    - Exponential: at least compute all subsets
- Solution 2:
    - For each pair (u,v)
        - C_1 = DFS(u) to find if there are path between u and v
        - C_2 = DFS(v) to find if there are path between u and v
        - Build the SCCs accordingly
    - Runtime: $O(n^2(n + m))$
        - We need to run DFS on each pair of nodes

Strongly Connected Components: pseudocode

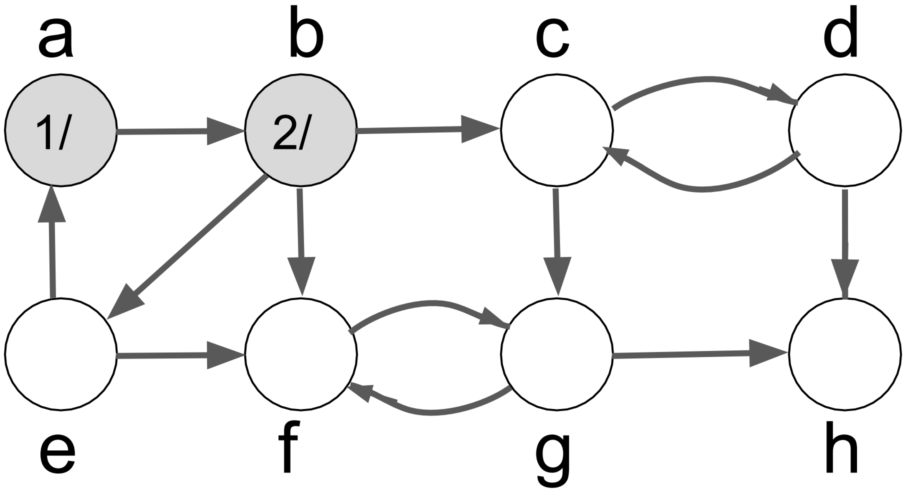
```
def StronglyConnectedComponents(G):
 1.    DFS(G)        # to compute finish time f[u] for each vertex u
 2.    Compute G^T
 3.    Call DFS(G^T) # traverse vertices in decreasing order of finish time
 4.    The SCCs are the different DFS tree in G^T
```

Strongly Connected Components: Example



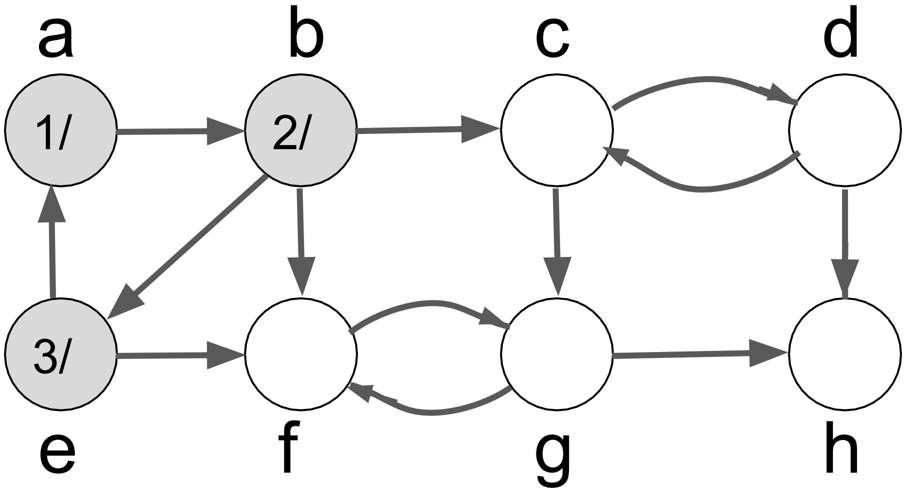
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



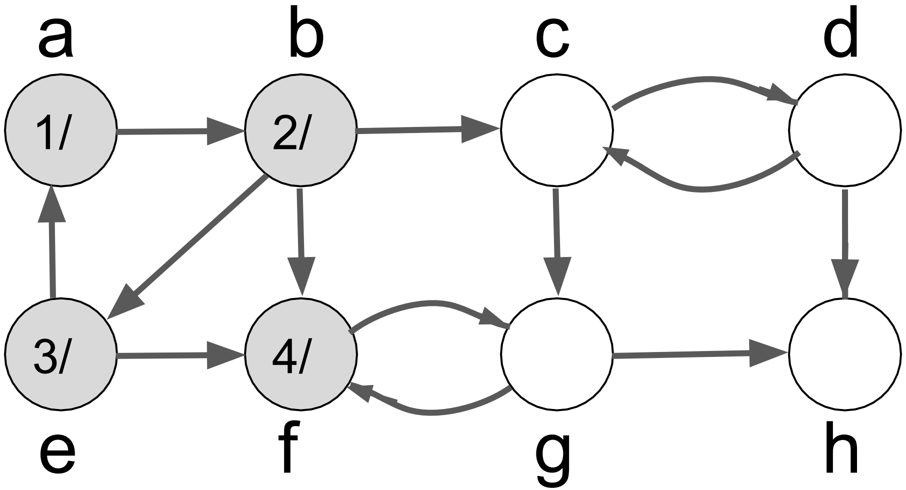
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



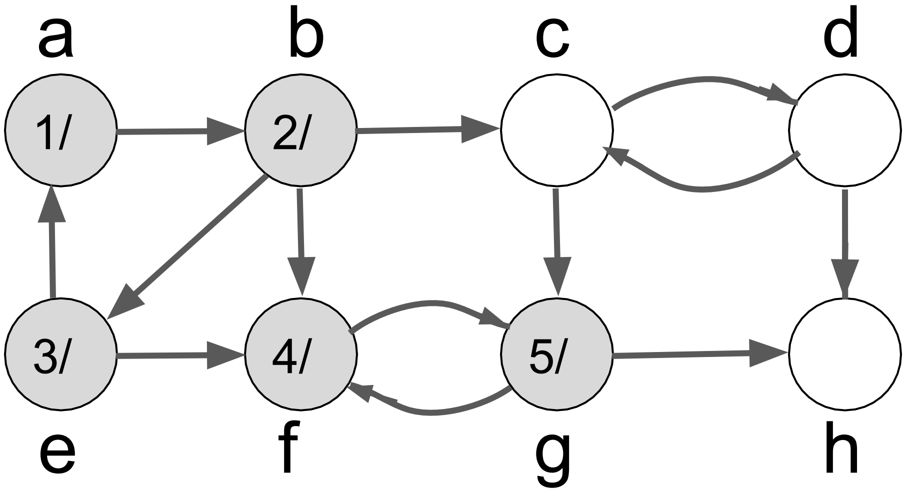
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



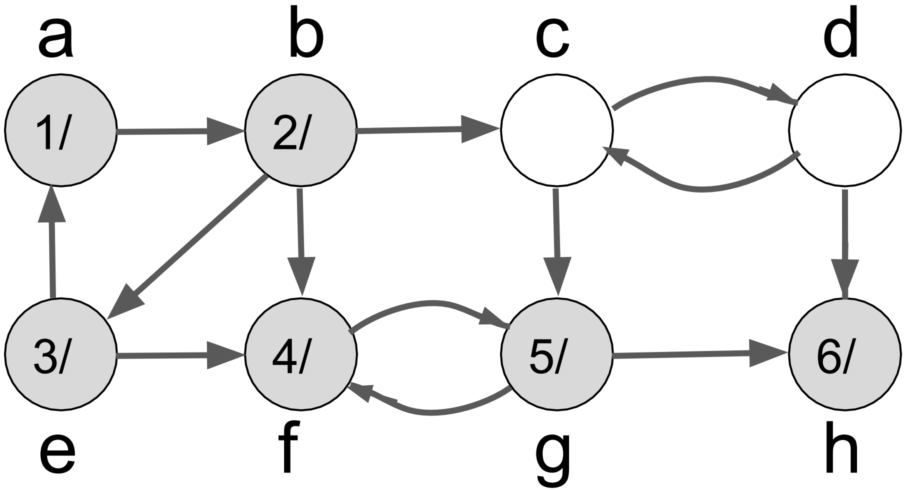
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



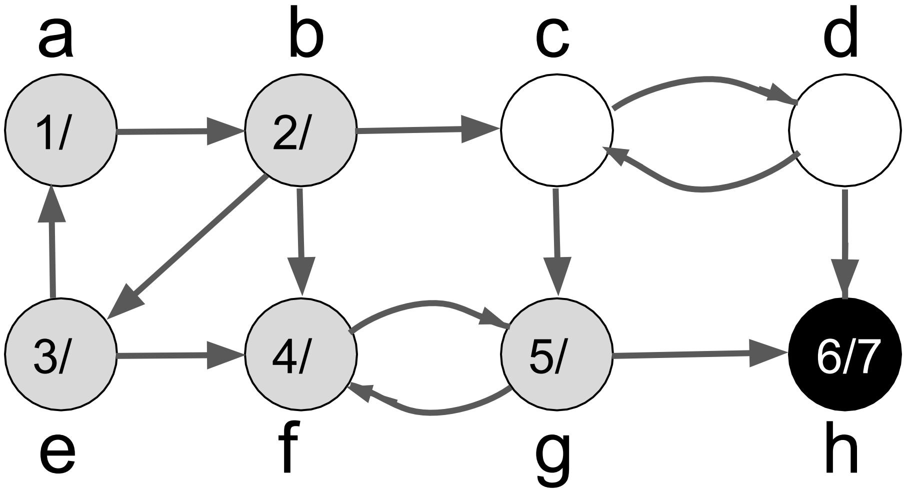
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



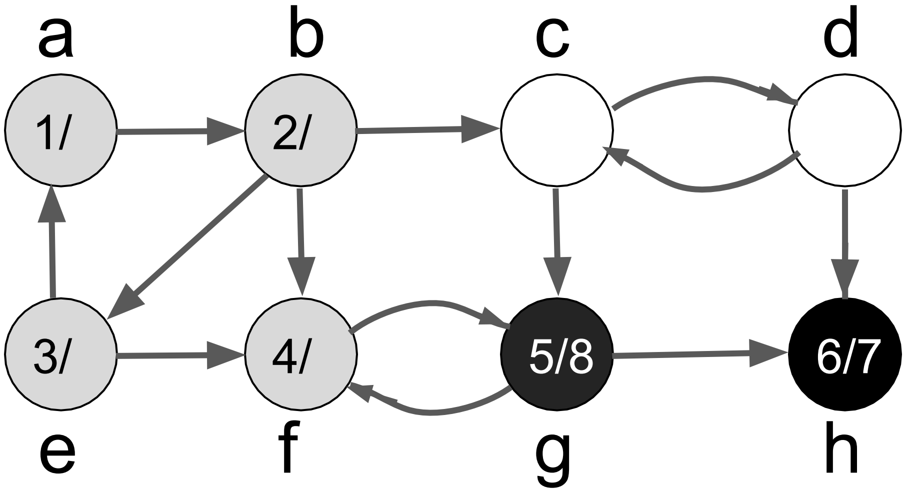
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



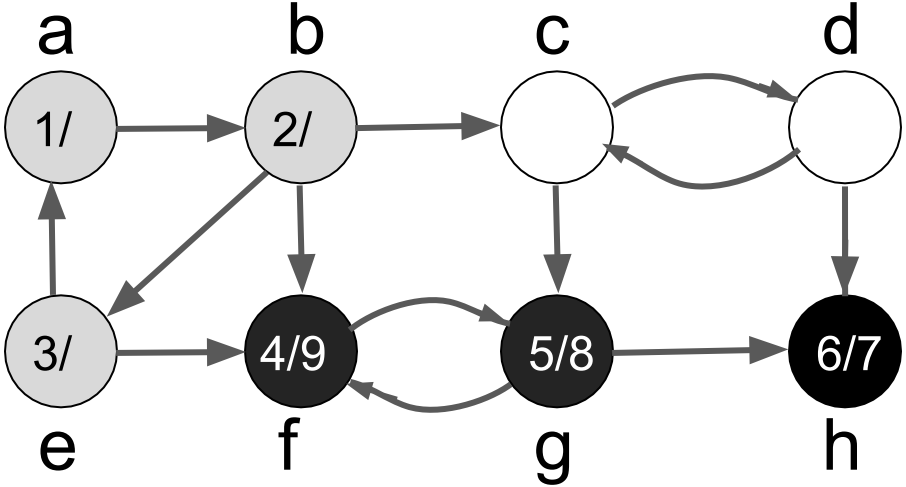
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



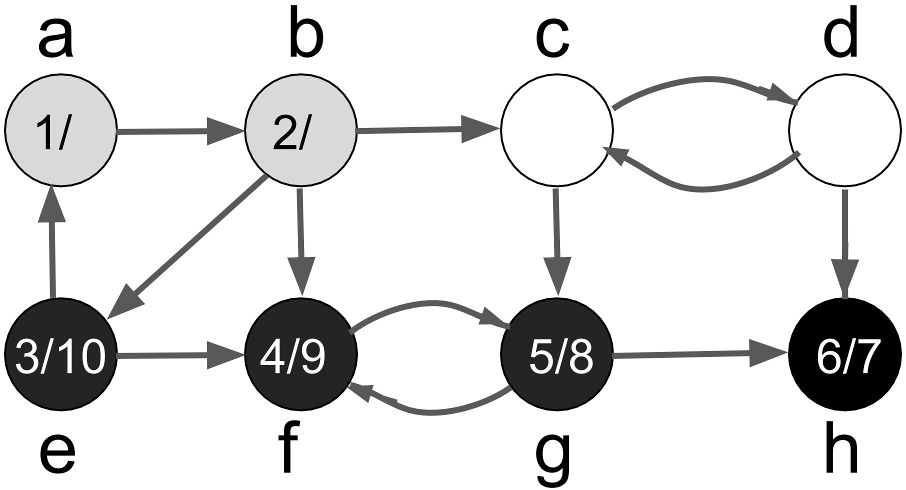
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



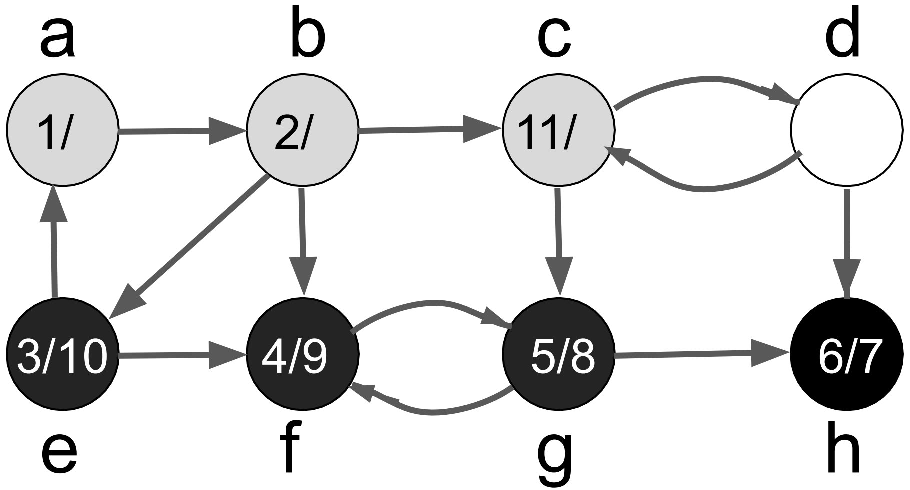
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



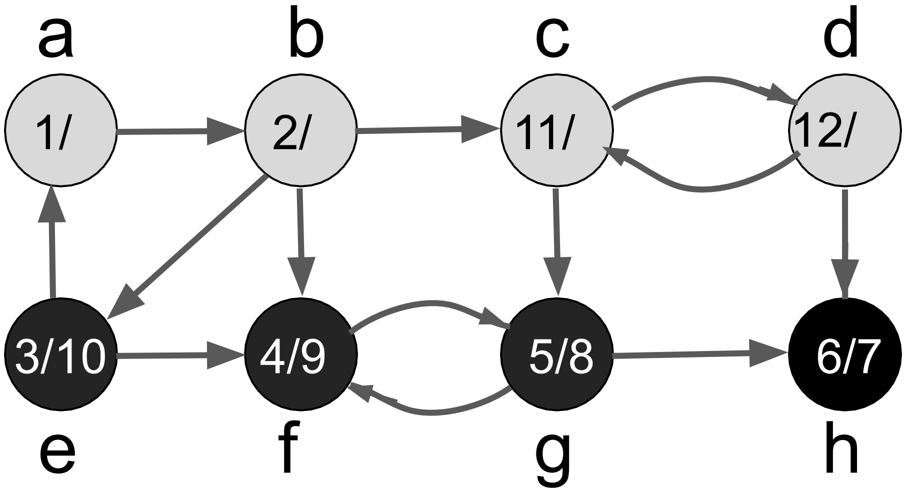
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



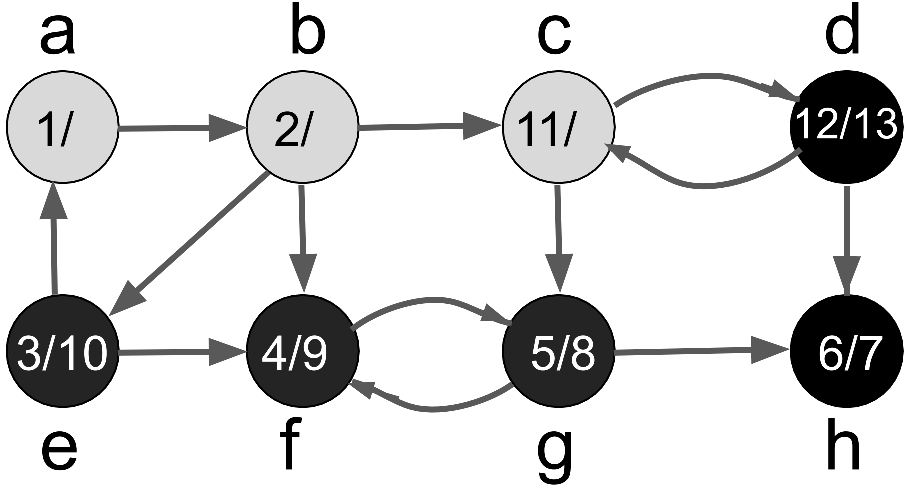
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



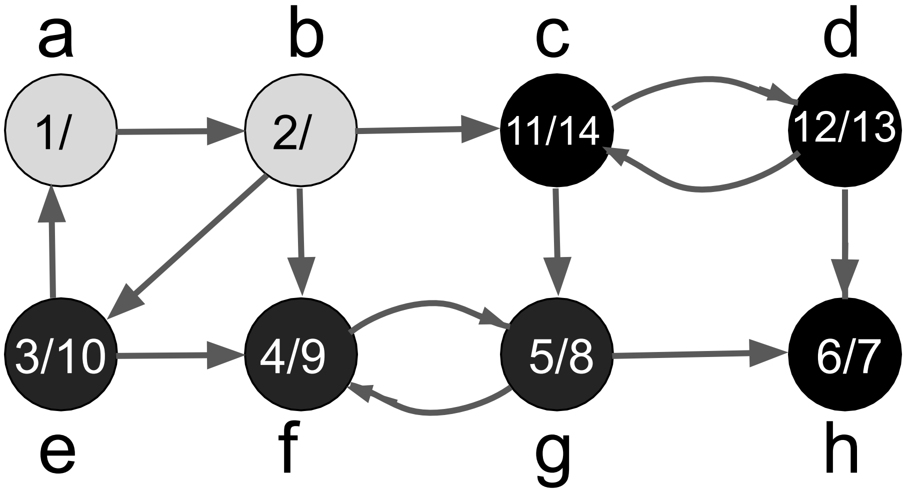
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



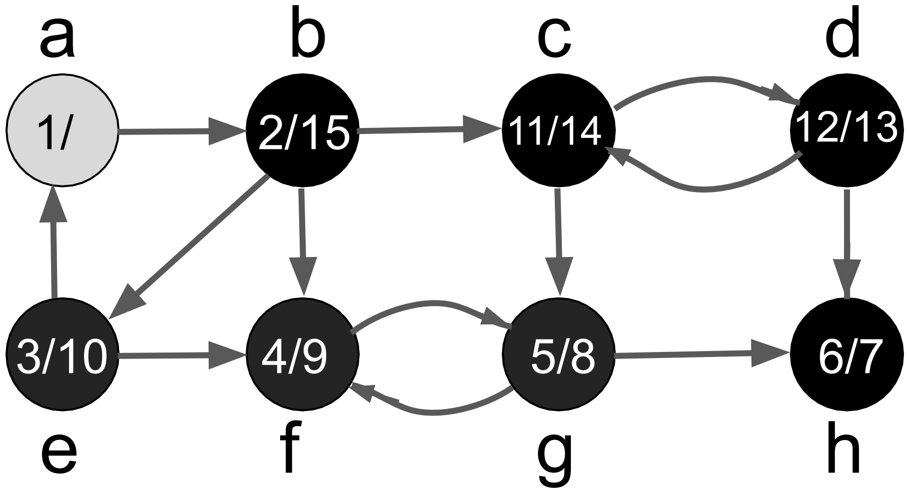
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



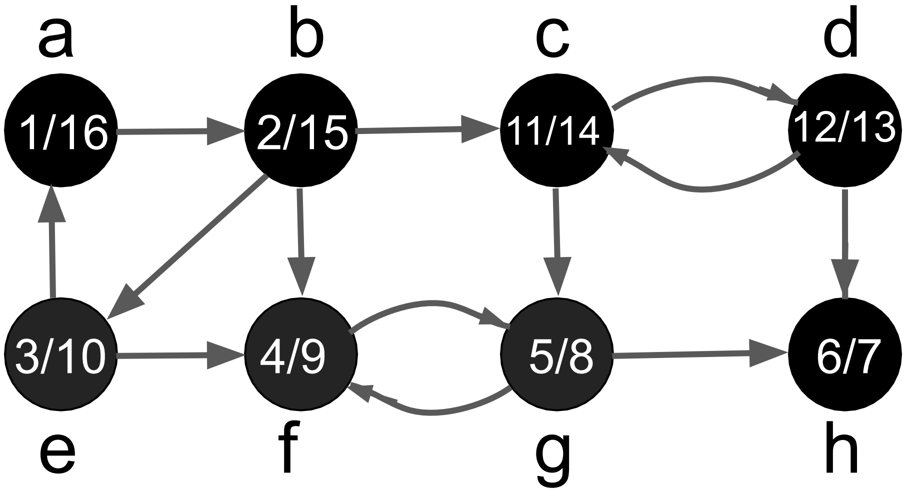
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



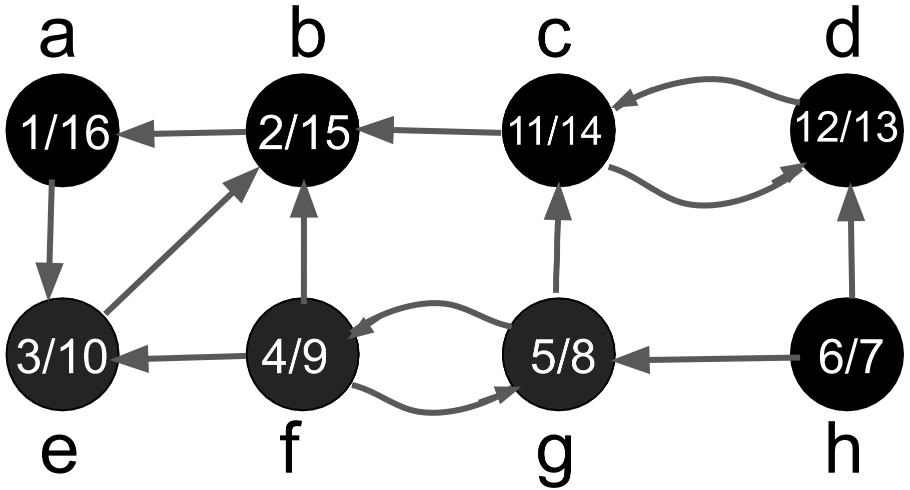
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



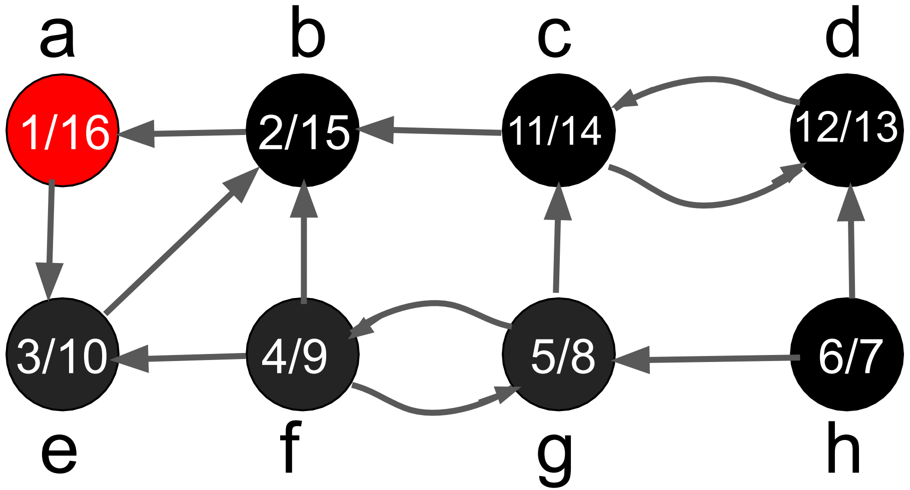
1. DFS(G) to compute finish time f[u] for each vertex u

Strongly Connected Components: Example



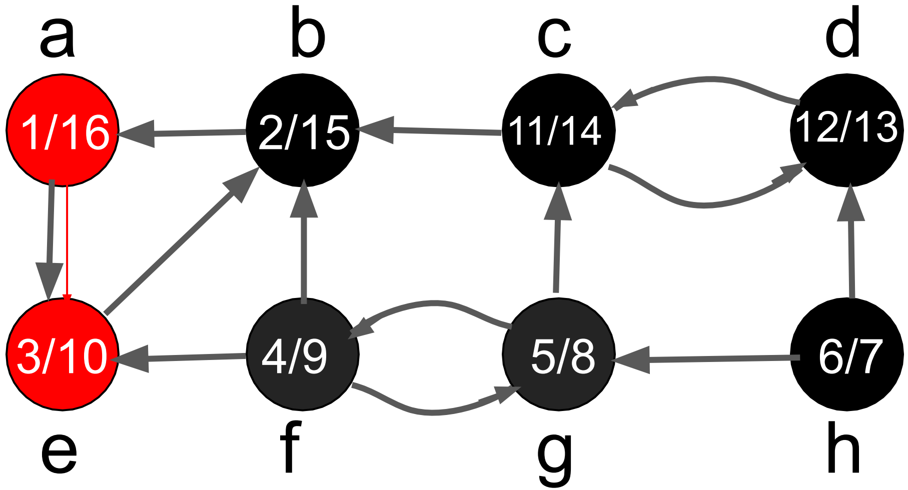
1. DFS(G) to compute finish time f[u] for each vertex u
2. Compute G^T

Strongly Connected Components: Example



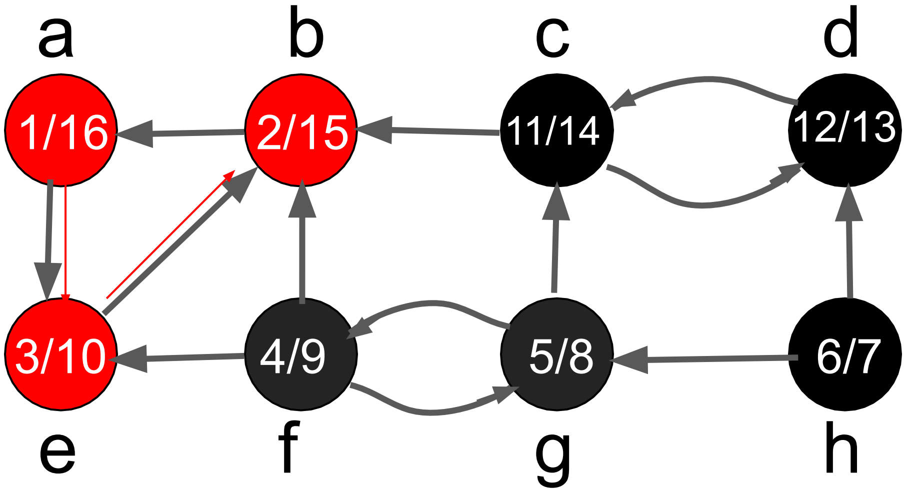
1. DFS(G) to compute finish time f[u] for each vertex u
2. Compute G^T
3. DFS(G^T) traverse vertices in decreasing order of finish time

Strongly Connected Components: Example



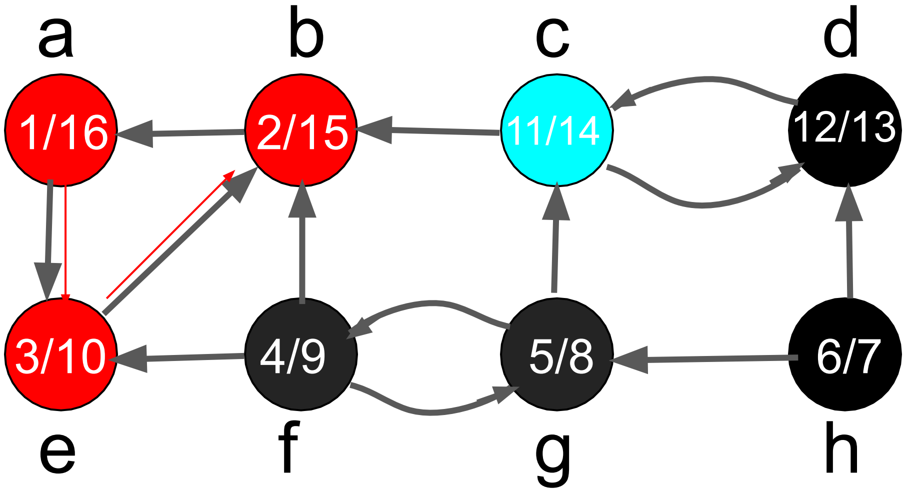
1. DFS(G) to compute finish time f[u] for each vertex u
2. Compute G^T
3. DFS(G^T) traverse vertices in decreasing order of finish time

Strongly Connected Components: Example



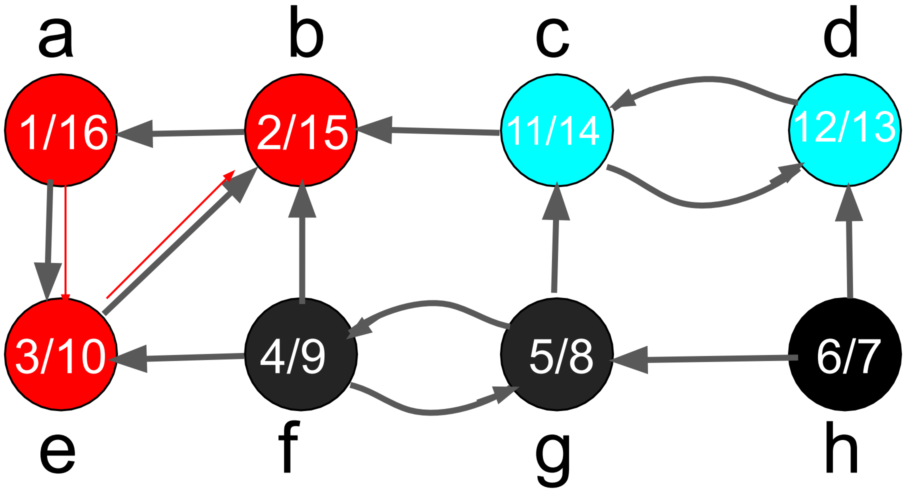
1. DFS(G) to compute finish time f[u] for each vertex u
2. Compute G^T
3. DFS(G^T) traverse vertices in decreasing order of finish time

Strongly Connected Components: Example



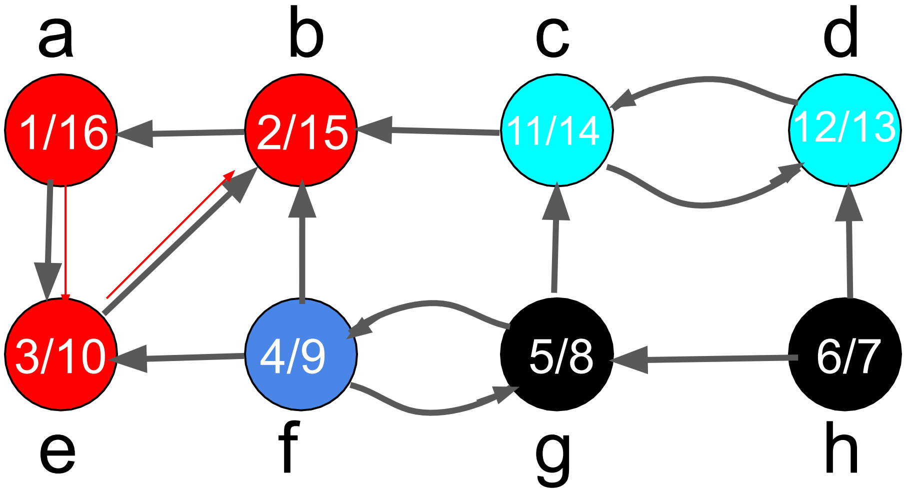
1. DFS(G) to compute finish time f[u] for each vertex u
2. Compute G^T
3. DFS(G^T) traverse vertices in decreasing order of finish time

Strongly Connected Components: Example



1. DFS(G) to compute finish time f[u] for each vertex u
2. Compute G^T
3. DFS(G^T) traverse vertices in decreasing order of finish time

Strongly Connected Components: Example

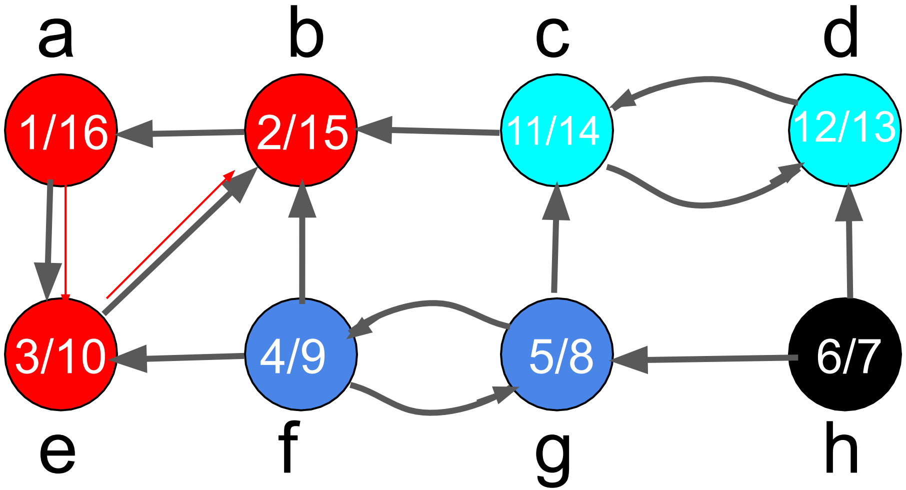


1. DFS(G) to compute finish time f[u] for each vertex u

2. Compute G^T

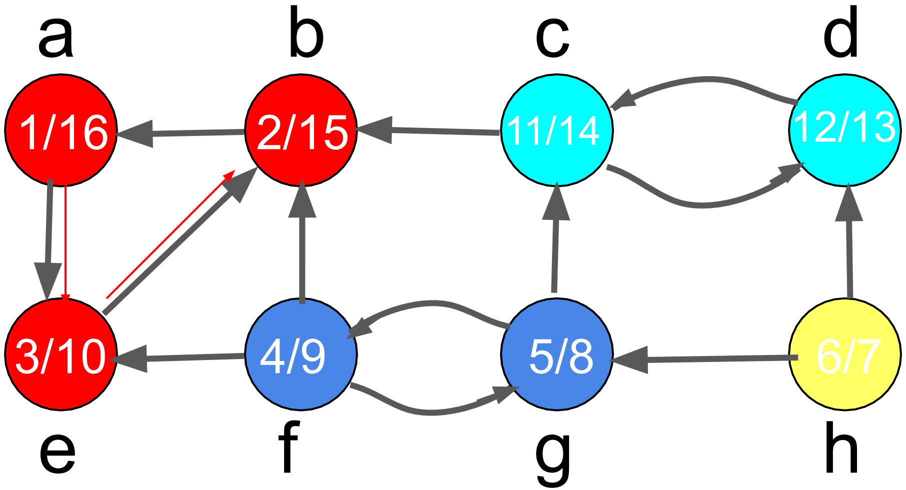
3. DFS(G^T) traverse vertices in decreasing order of finish time

Strongly Connected Components: Example



1. DFS(G) to compute finish time f[u] for each vertex u
2. Compute G^T
3. DFS(G^T) traverse vertices in decreasing order of finish time

Strongly Connected Components: Example

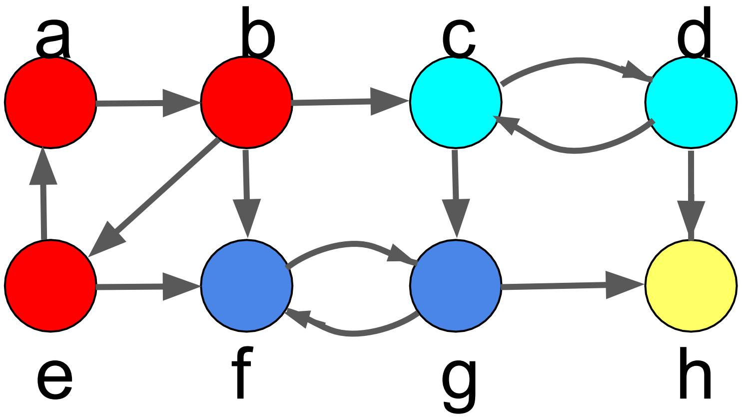


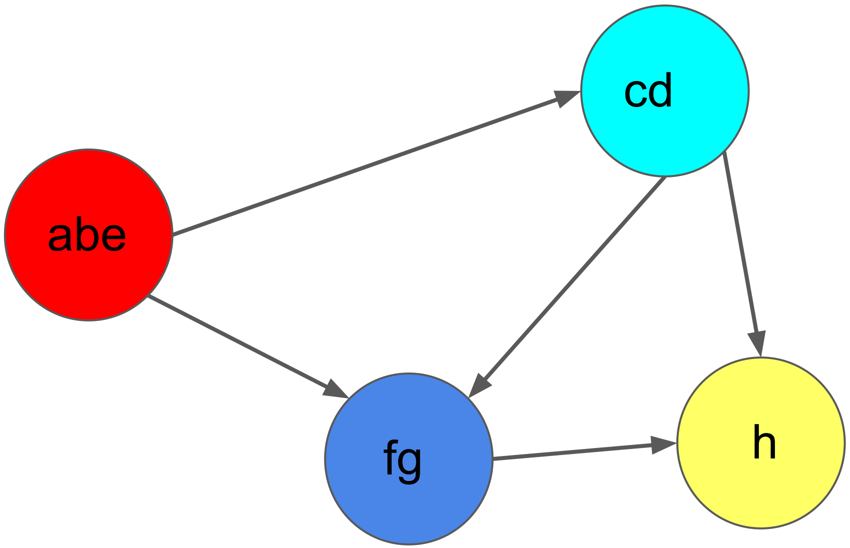
1. DFS(G) to compute finish time f[u] for each vertex u
2. Compute G^T
3. DFS(G^T) traverse vertices in decreasing order of finish time

Strongly Connected Components: Correctness

- The **component graph** $G^{SCC}=(V^{SCC}, E^{SCC})$:
  - Is Obtained by contracting every strongly connected component into a single vertex
  -
  - The vertices of G^{SCC} are are the SCCs of G
  -
  - (C_1, C_2) is an edge in G^{SCC} if and only if $(u,v) \in E$ and $u \in C_1$ and $v \in C_2$

Strongly Connected Components: component graph





Strongly Connected Components: Correctness

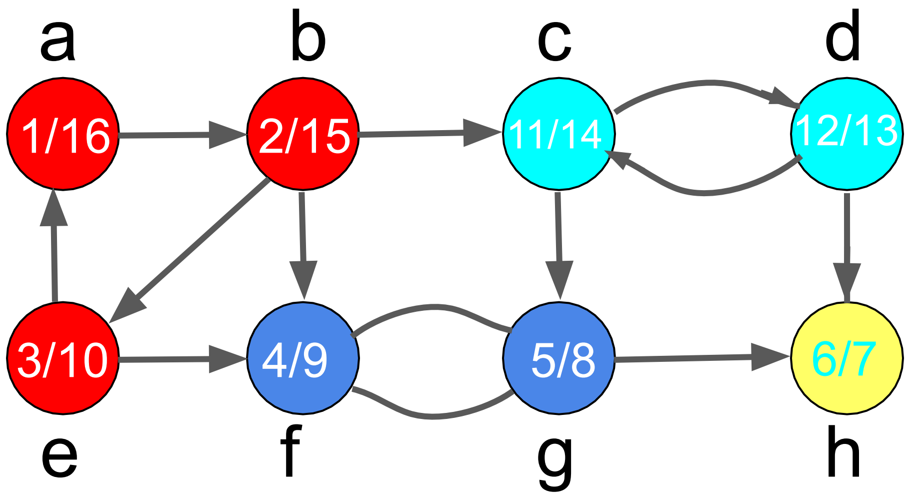
- **Lemma**. The component graph is a Directed Acyclic Graph
- Proof idea. If not, then two SCCs would collapse into one

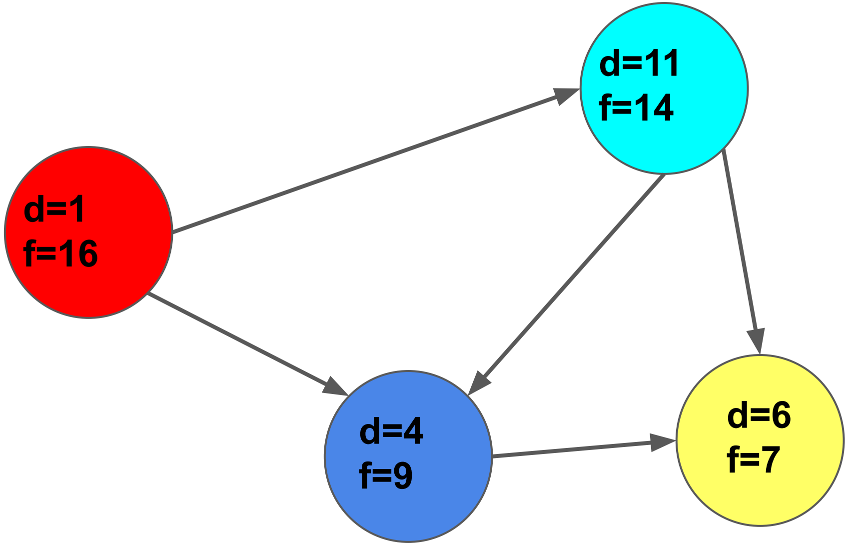
Strongly Connected Components: Correctness: notations

- The discovery and finish time times for a set $U \subseteq V$:
  - $f(U)$: The finish time of a set $U \subseteq V$ is the largest finish time of any vertex $v \in U$
  - $d(U)$: The discovery time of a set $U \subseteq V$ is the smallest discovery time of any vertex $v \in U$

Strongly Connected Components: Correctness

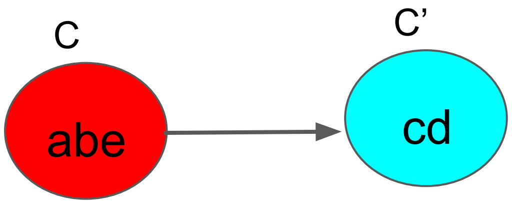
The **component graph** with discovery and finish times for each component





Strongly Connected Components: Correctness

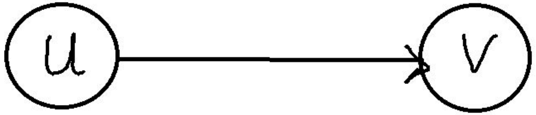
- **Lemma.** Let C and C' be distinct SCC in directed graph G=(V, E). Suppose that there is an edge (u, v) ∈ E, where u ∈ C and v ∈ C'. Then f(C) > f(C')
- **Proof.** There are two cases:
  - **Case 1:** We reached C' before C in the first DFS. There are no paths from C' to C.
  - Since there is a path from C to C', there cannot be a path from C' to C otherwise there would be cycle in the component graph which is a DAG. So we finish exploring C' and never reach C and C is explored later there
  - finish(C) > finish(C')
  - **Case2:** Suppose the first vertex v discovered is in C. Since vertices in C ∪ C' are reachable from v, all vertices in C ∪ C' will be finished before v is finished and so v ∈ C has the largest finish time



finish(C) > finish(C')

Strongly Connected Components: Correctness

- Remember that in a DAG, if $(u,v) \in E$
  - finish[u] > finish[v]

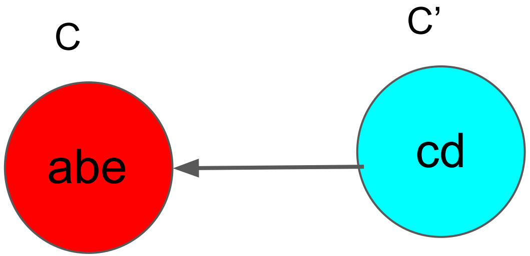


finish[u] > finish[v]

$$d[u] < d[v] < f[v] < f[u]$$

Strongly Connected Components: Correctness

- **Corollary.** Let C and C' be distinct SCC in directed graph G=(V, E). Suppose that there is an edge $(v, u) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$ where finish times are generated by running DFS on G=(V, E)

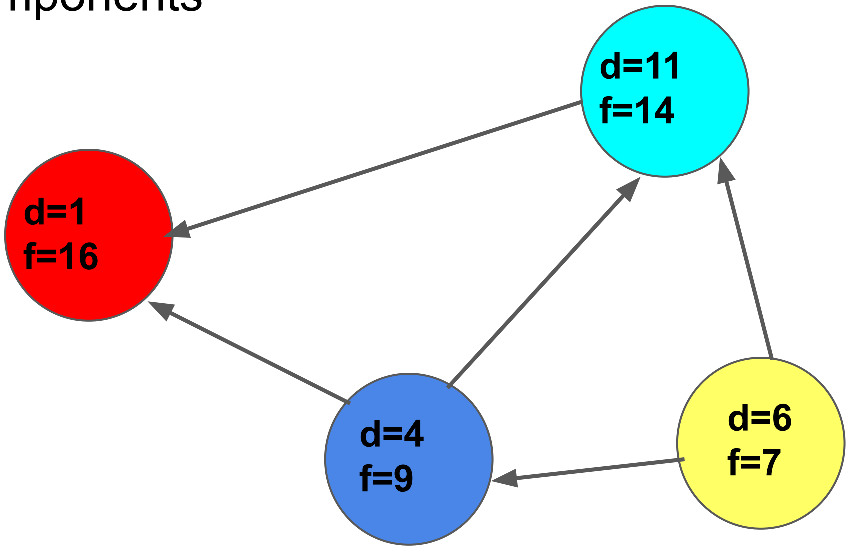


finish(C) > finish(C')

- **This means if we choose a SCC component C with largest finish time, there would be no edge from C to any other SCC**

Strongly Connected Components: Correctness

- Consider the component graph where all edges are reversed
- The SCC with the largest finish time has no edges going out
- So by running DFS there, we'll get exactly that component
- Then we repeat the process on other components



Strongly Connected Components: Algorithm

- Reverse the edges of component graph
- Repeat
  - The SCC with the largest finish time has no edges going out
    - Only that connected component is reachable by second DFS
    - So by running DFS there, we'll get exactly that component
  - Then we delete that component and repeat the process on other components