Algorithms & Data Structures I
CSC 225

Ali Mashreghi

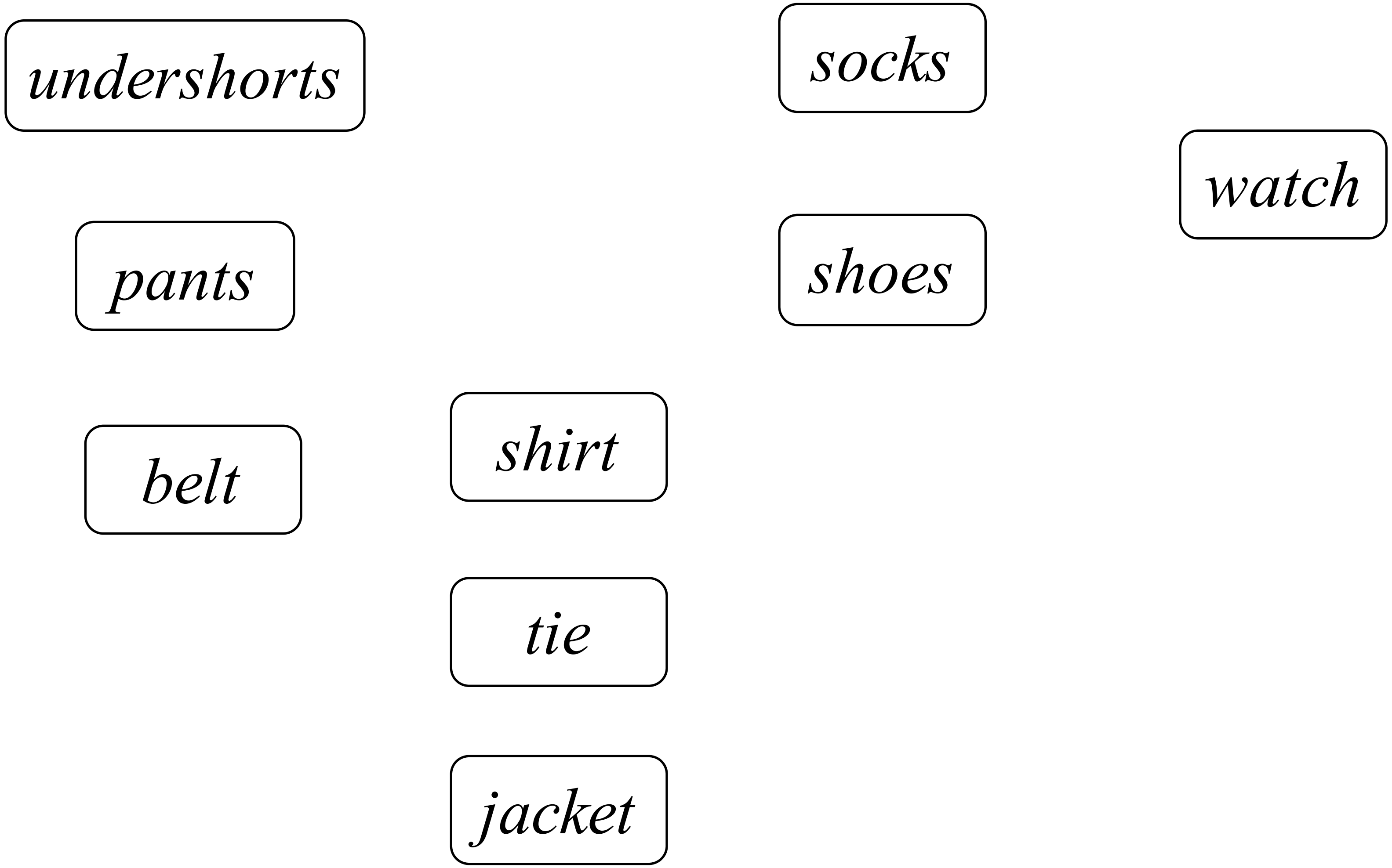
Fall 2018

Department of Computer Science, University of Victoria


Welcome back from reading break!

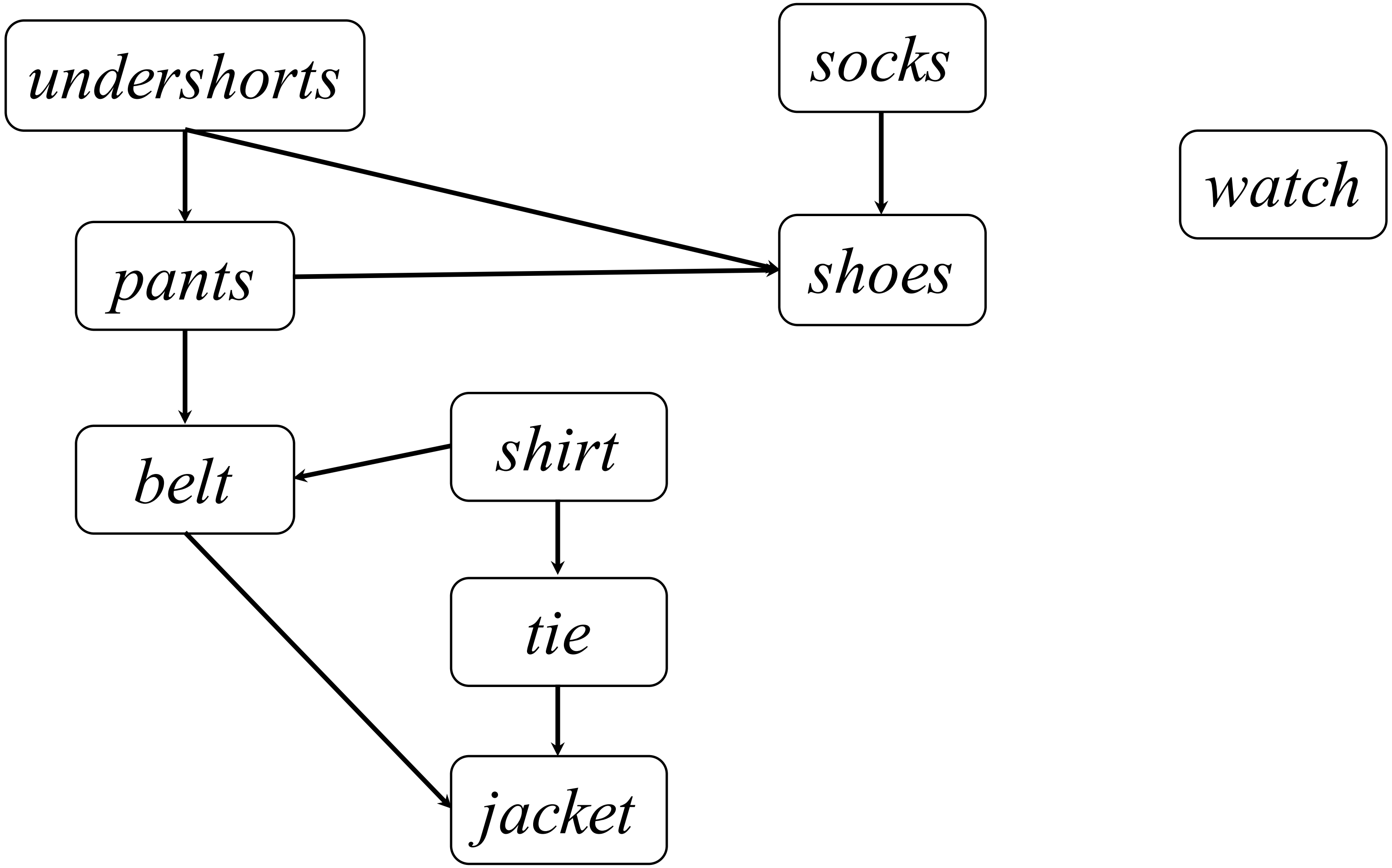
Topological sort

- Let’s say you are getting ready for work!



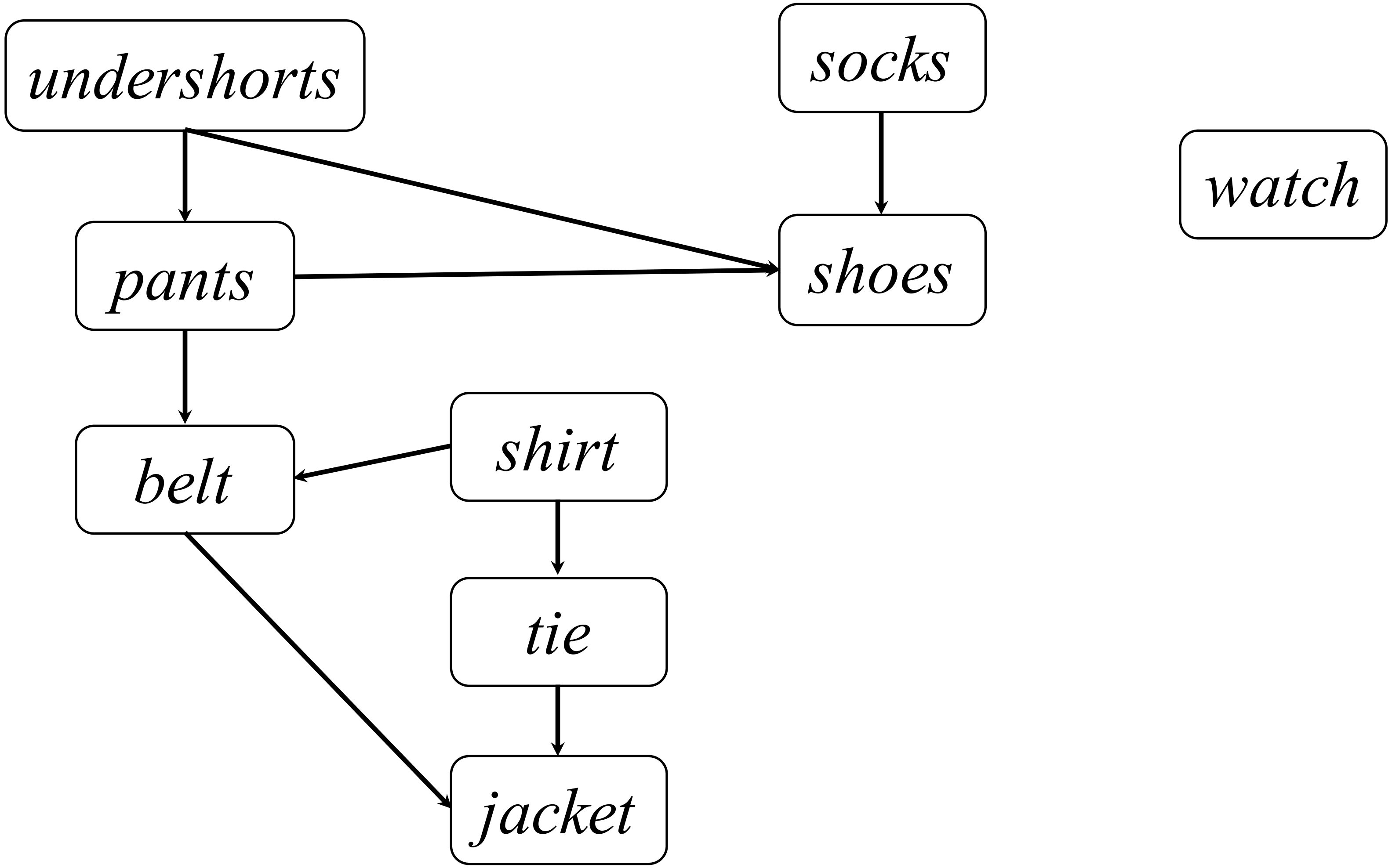
Topological sort

- Some pieces you have to wear before others.



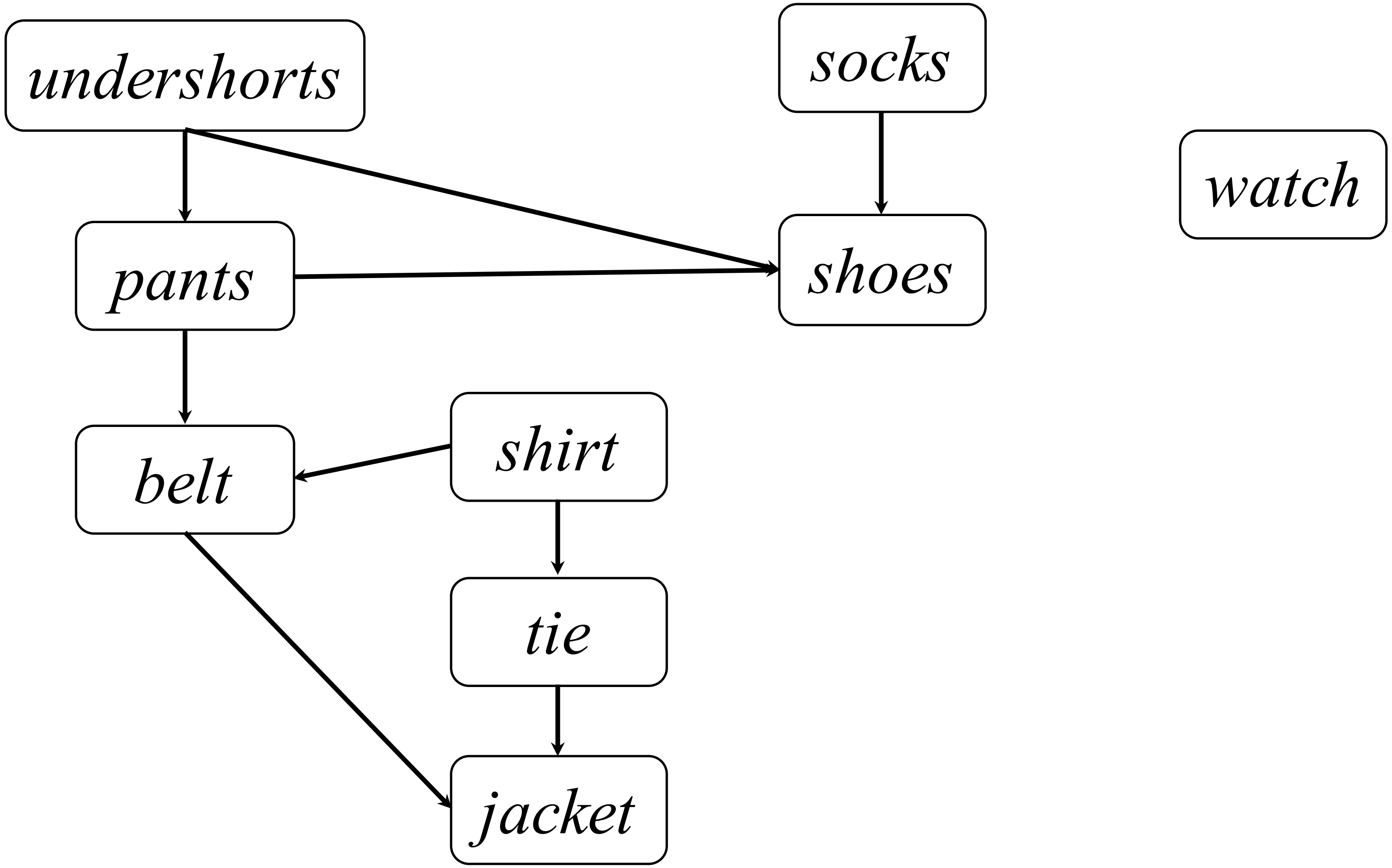
Topological sort

- In what order can you start wearing your clothes?



Topological sort

- This is an example of a **Directed Acyclic Graphs** (DAGs)



Topological sort

- We are looking for a topological order, i.e. if u has an edge to v, u has to appear before v in the ordering.

- We say **topological sort** because we have to sort based on a **topology or arrangement**.

- The main application of such an ordering is to **schedule jobs** in an operating system.

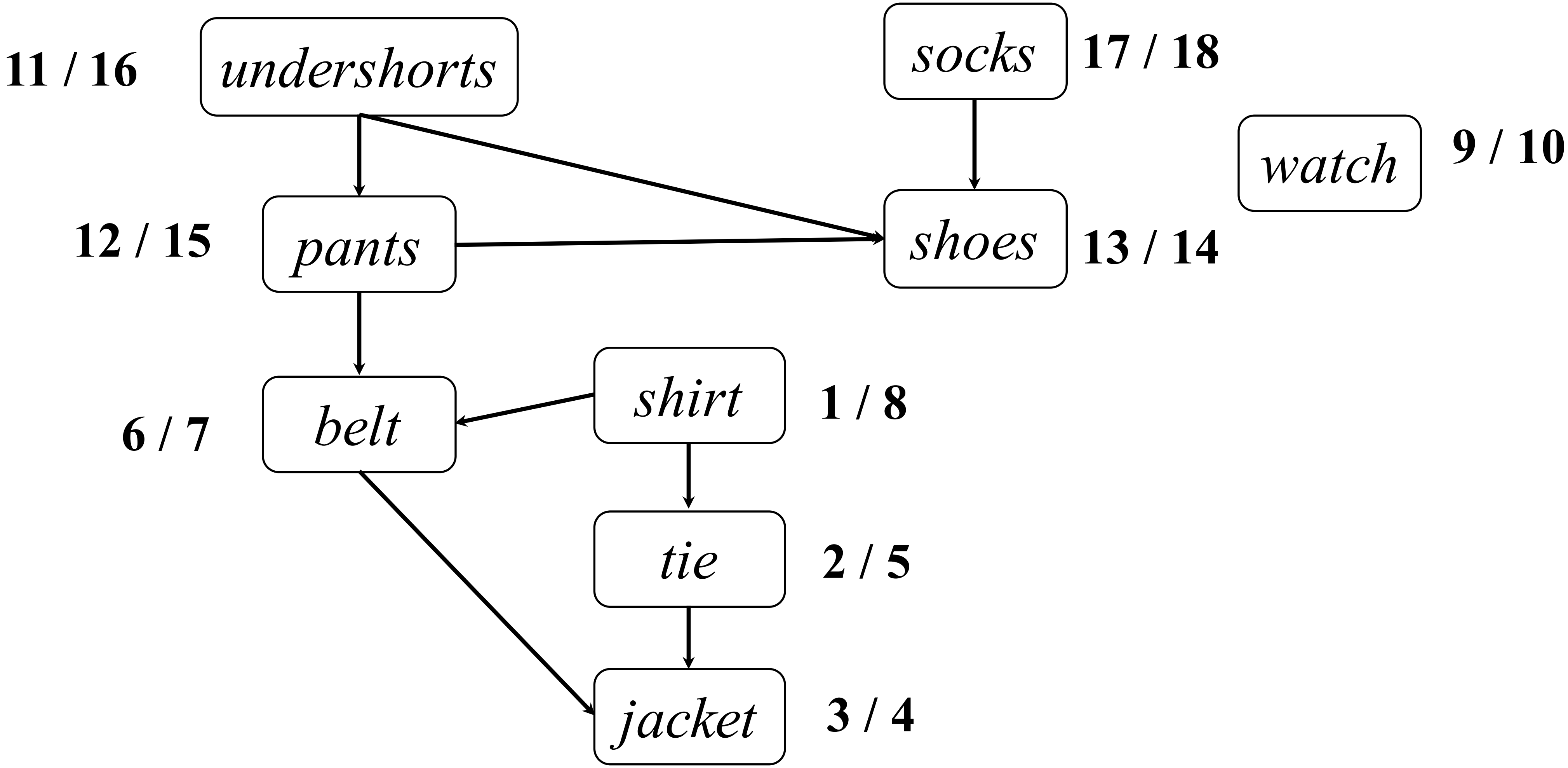
Topological sort

- The idea is to run the DFS on the graph and order the nodes according to their **finish times** in **reverse order**.

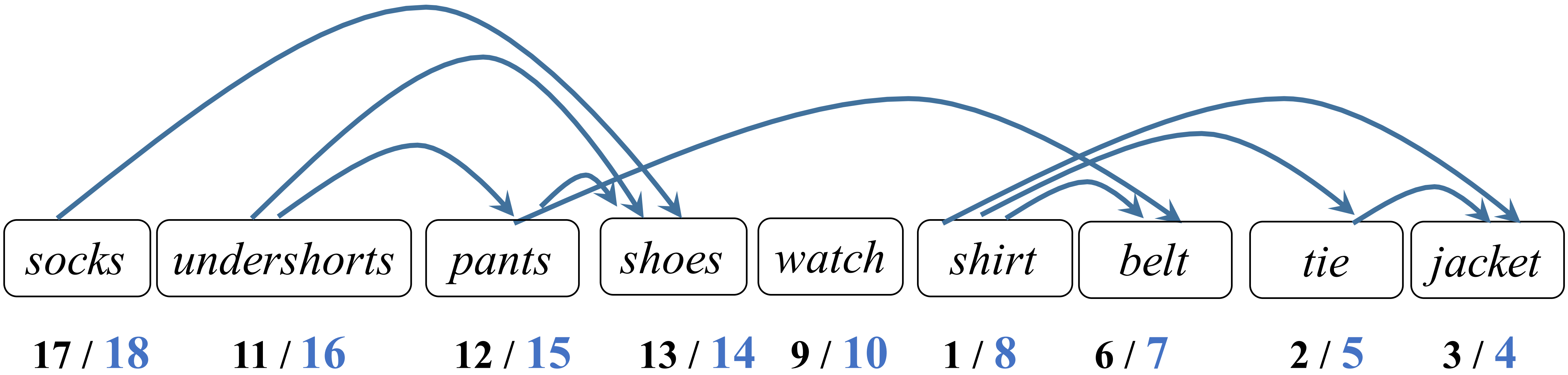
TOPOLOGICAL-SORT(G)

call DFS(G) to compute finishing times $v.f$ for each vertex v
as each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices

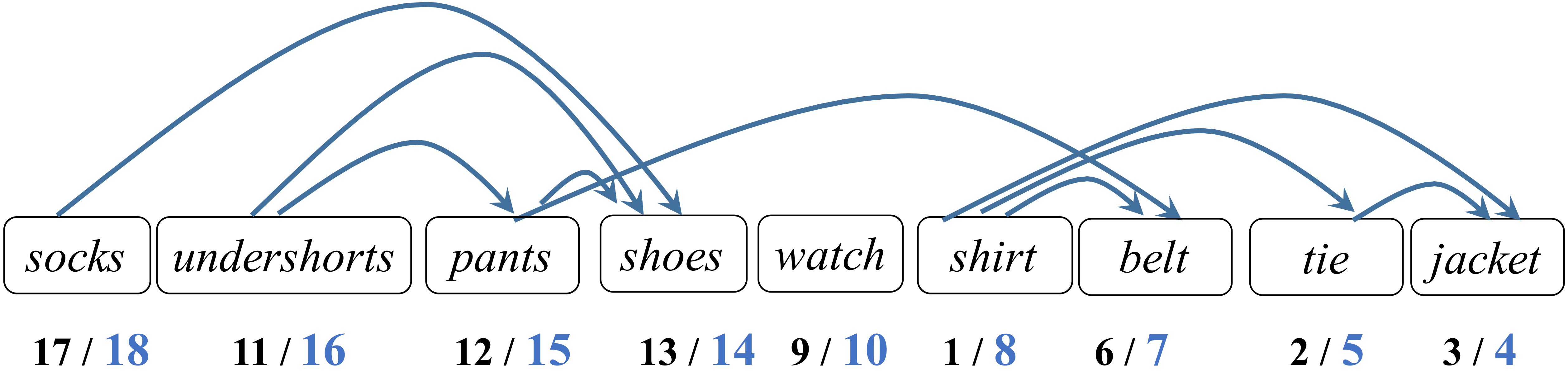
Topological sort



Topological sort



Topological sort



For any item x, all the items that x depends on come before it.

Topological sort

- **Theorem:** Regardless of the order in which the DFS algorithm visits the nodes, if u has an edge to v, u appears before v, in the list returned by the $\textsc{Topological-sort}$ algorithm.

Topological sort

- **Proof:** Assuming the edge (u, v) is in the graph.
- Case 1: If u starts before v, then in order for u to finish, v has to finish first; therefore, u will appear before v in the list.
- Case 2: If v starts before u, then, the only situation to get a wrong order is when there is a path from v which causes u to finish first and then u. However, such a path is impossible as it will create a cycle by adding the edge (u, v). (A DAG is acyclic.)

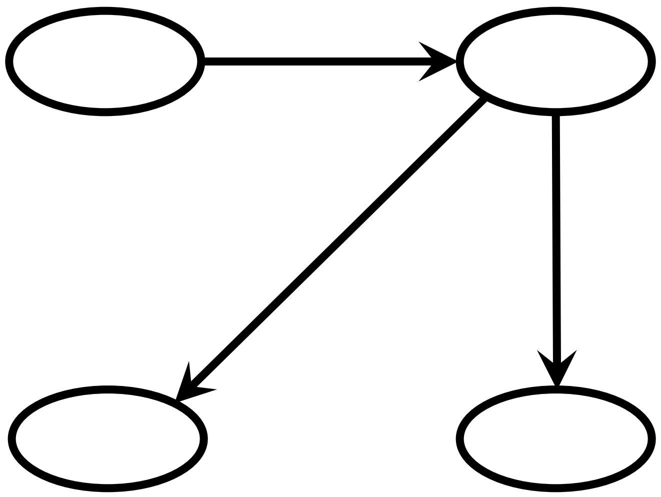
Digraphs and connectedness

- **Digraph** is another term for a directed graph.

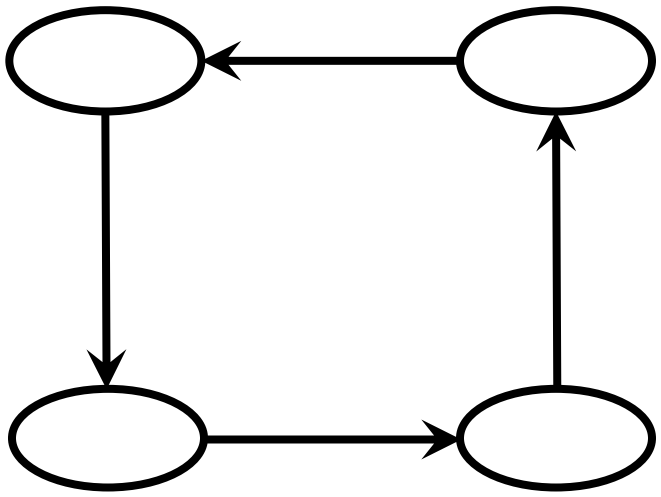
- We define connectedness for a digraph as follows:
- **Strongly connected:** If for all pairs of vertices u and v, u can reach v and also v can reach u.

- **Weakly connected:** If a digraph is not strongly connected but when we make the graph undirected all nodes can reach each other, the digraph is weakly connected.

Digraphs and connectedness



Weakly connected



Strongly connected

Strongly connected components

- In a directed graph we define a **strongly connected component** as follows:

✓ A **maximal set** of nodes such that for any pair of vertices u and v in the set, $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

Strongly connected components

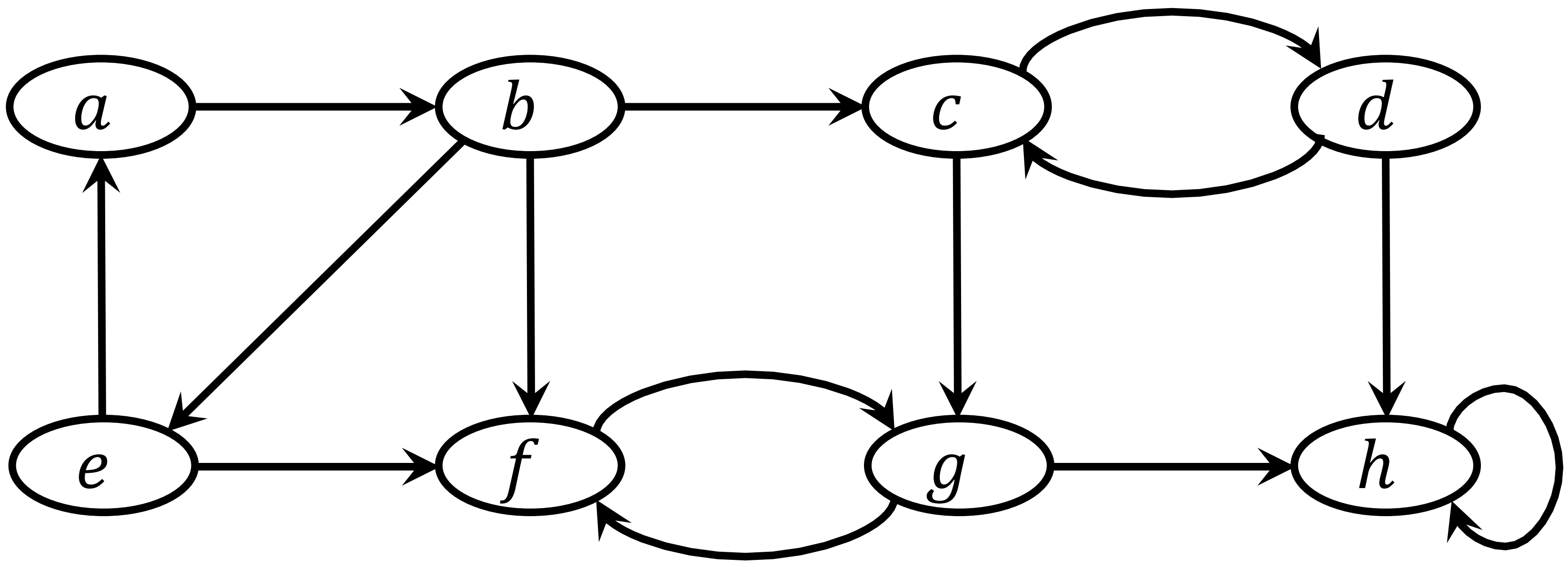
- In a directed graph we define a **strongly connected component** as follows:

✓ A **maximal set** of nodes such that for any pair of vertices u and v in the set, $u \rightsquigarrow v$ and $v \rightsquigarrow u$.

- A set with some property is **maximal** if we cannot add any more nodes to the set such that the property still holds.

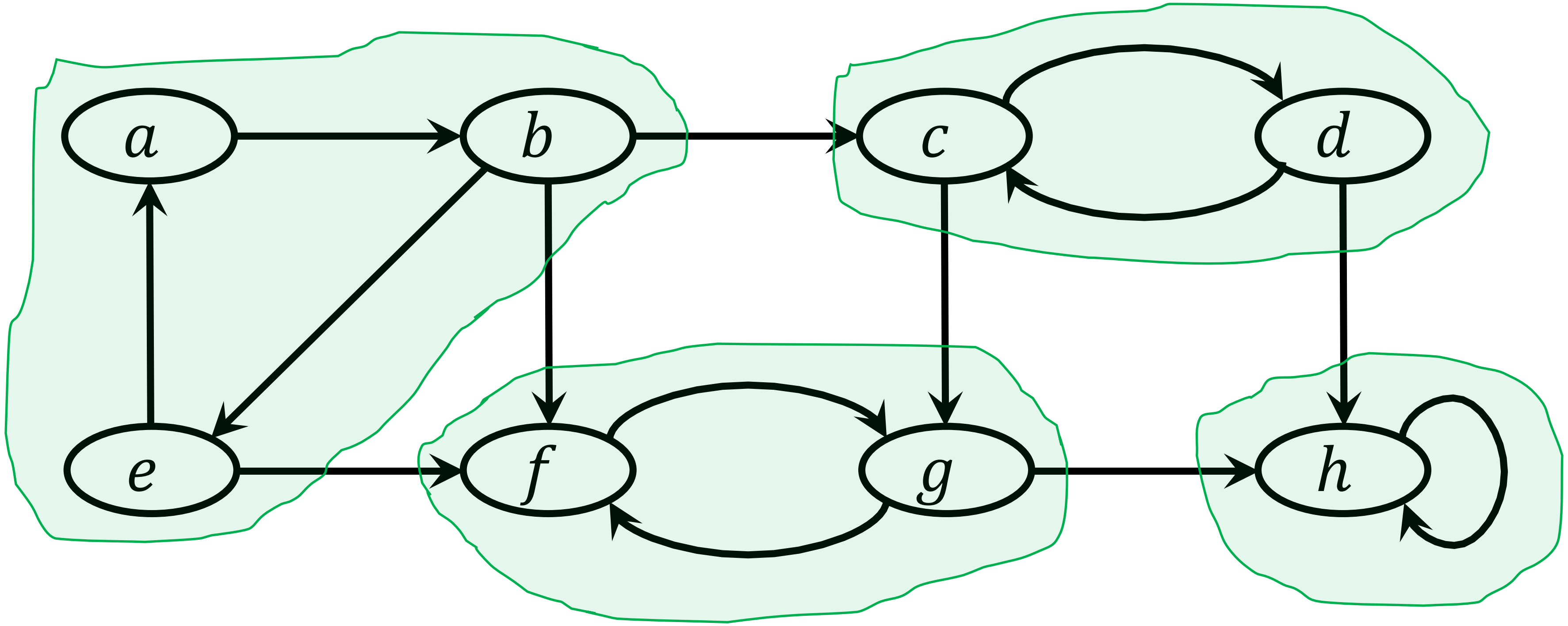
- $x \rightsquigarrow y$ means that there is a path from x to y, or x can reach y.

Strongly connected components



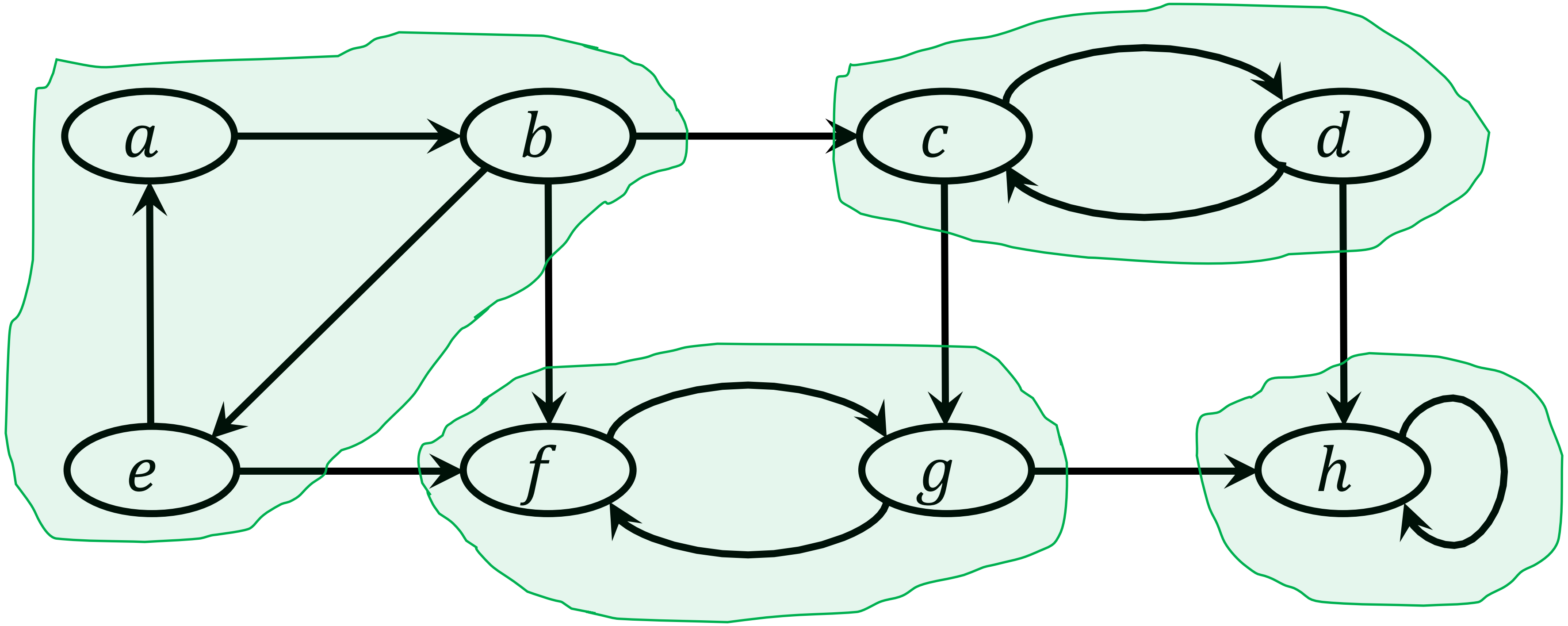
Strongly connected components

- There are four strongly connected components.



Strongly connected components

- There are four strongly connected components.



- We can find the components in **linear time**, i.e. $\boldsymbol{O(n+m)}$

Algorithm

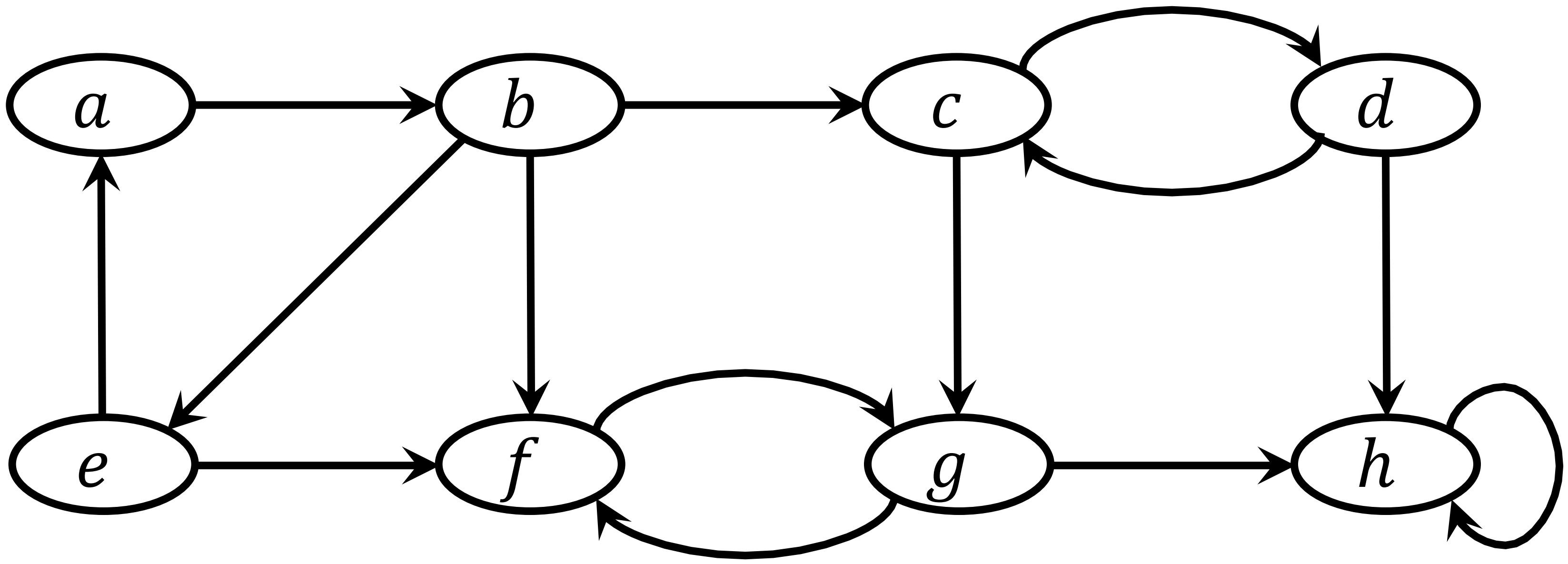
STRONGLY-CONNECTED-COMPONENTS(G)

1 call DFS(G) to compute finishing times $u.f$ for each vertex u
2 compute G^{T}
3 call DFS(G^{T}), but in the main loop of DFS, consider the vertices
    in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a
    separate strongly connected component

- G^T is the **transpose** of the input graph G.
- Basically, in G^T the direction of all edges is **reversed**.

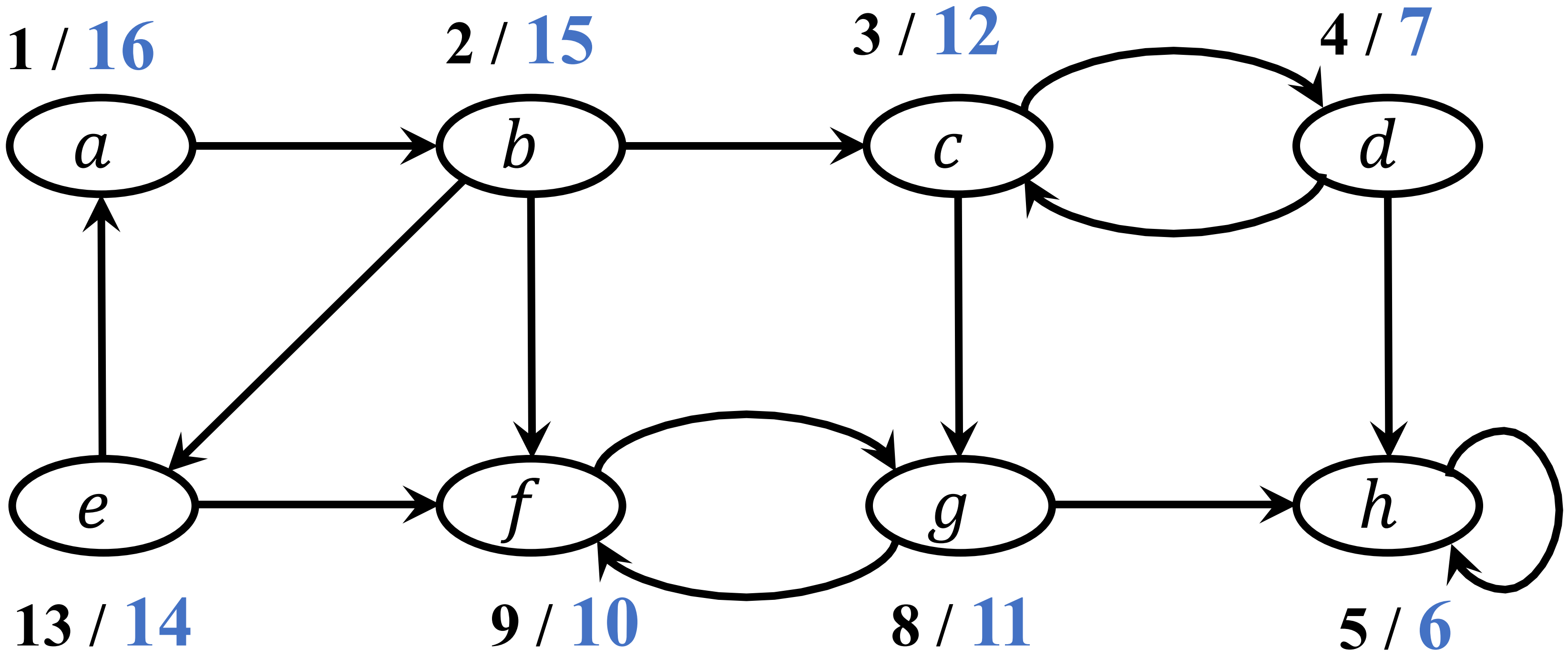
Algorithm

- Let this be G



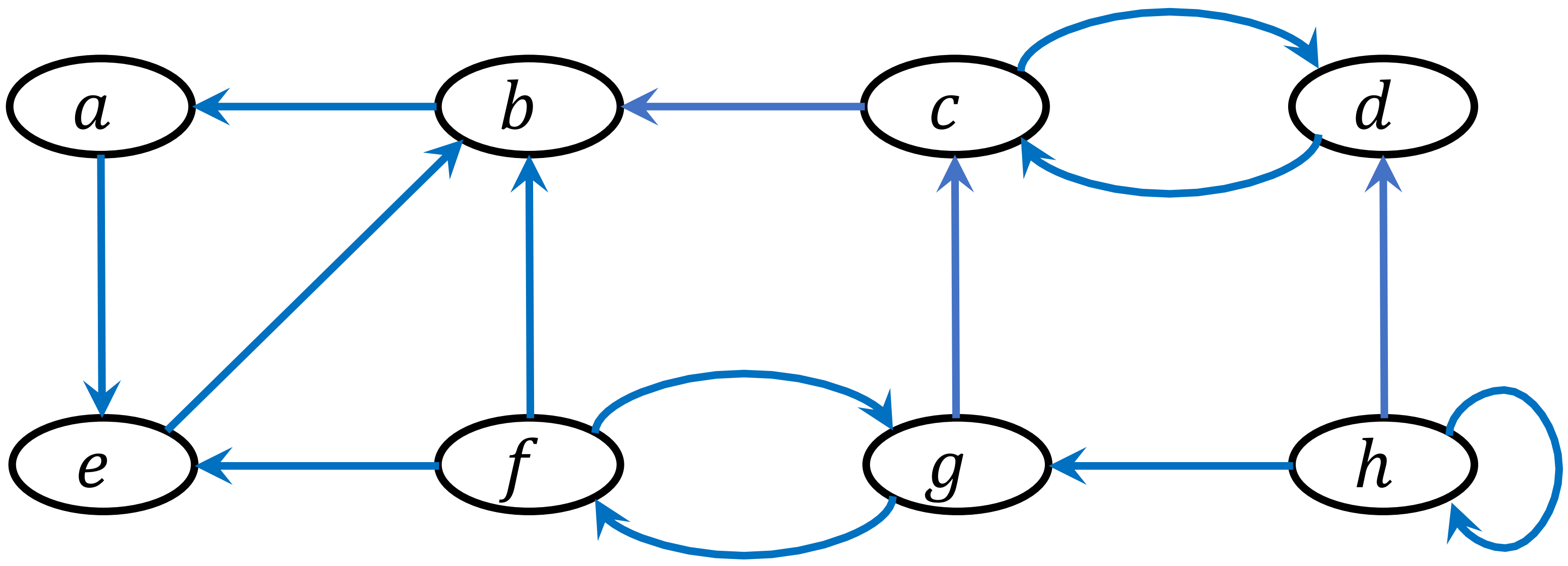
Algorithm

1. We run a DFS on G. The order of picking the nodes **doesn't matter** at this point.



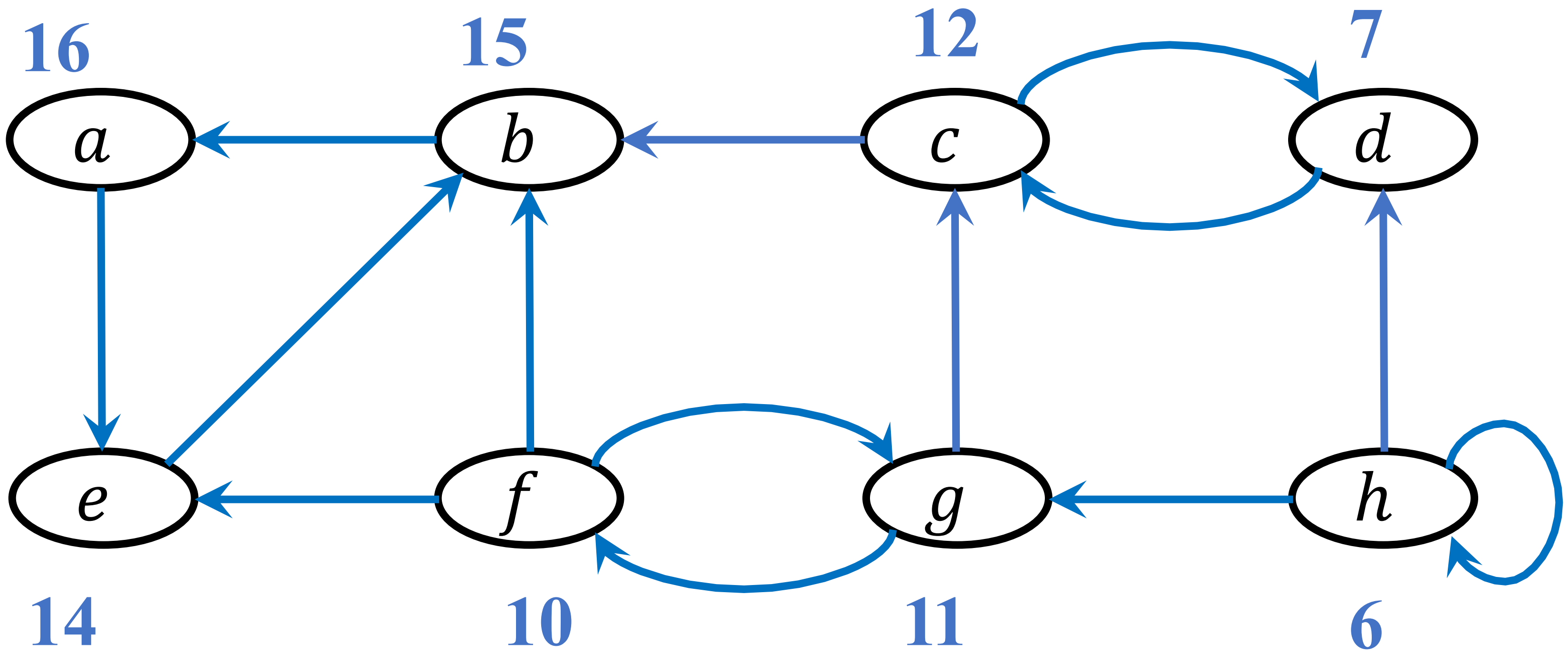
Algorithm

2. Then, we compute G^T by making a new graph where the edges have been reversed.



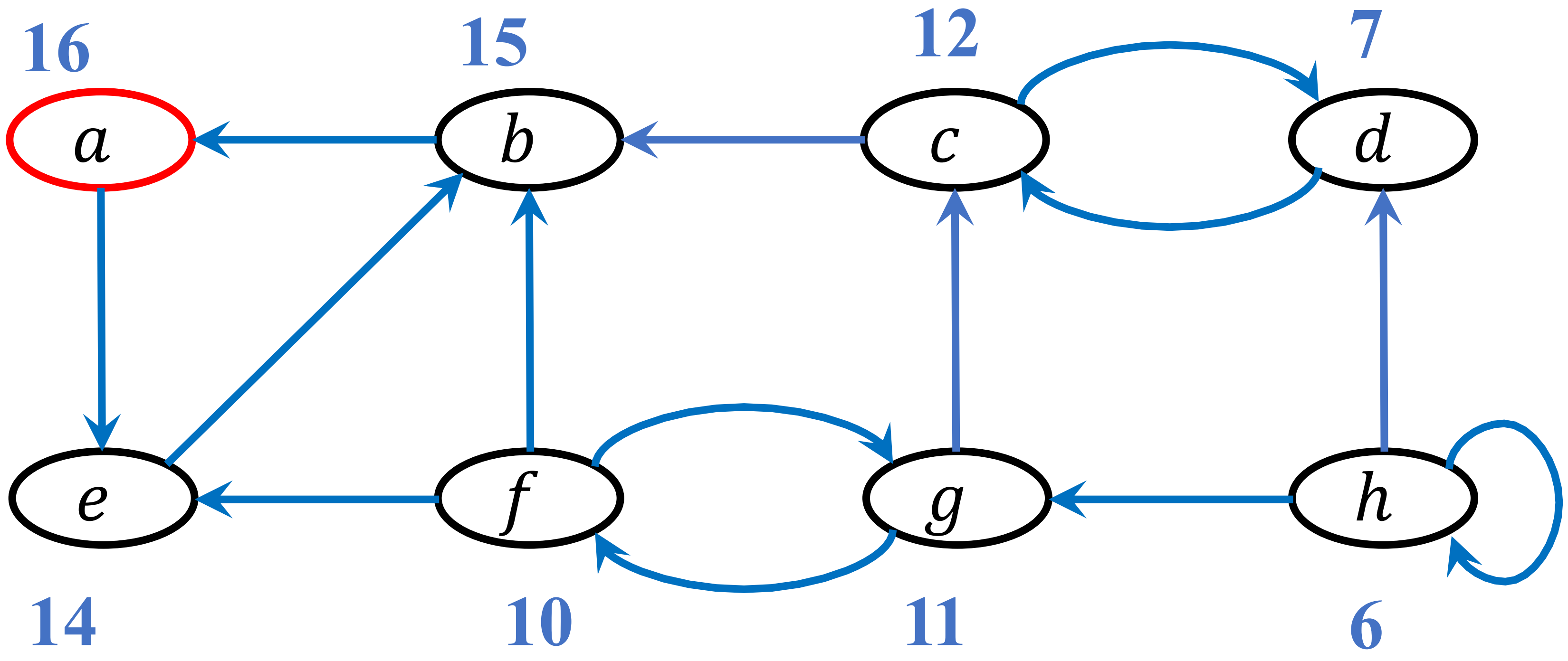
Algorithm

3. We run a DFS on G^T but we pick the nodes with higher finishing times first (computed at step 1).



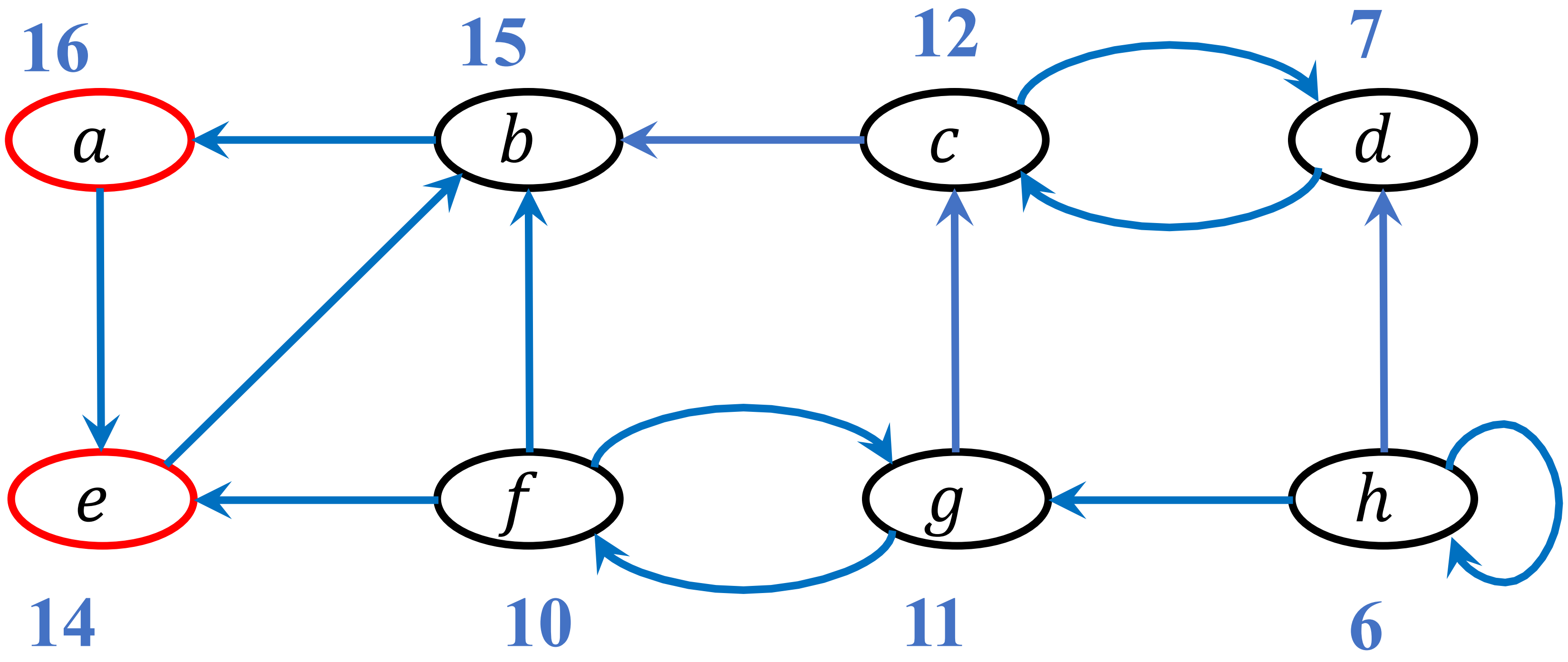
Algorithm

4. Each time we start with a node we put all reachable nodes from it in the same component.



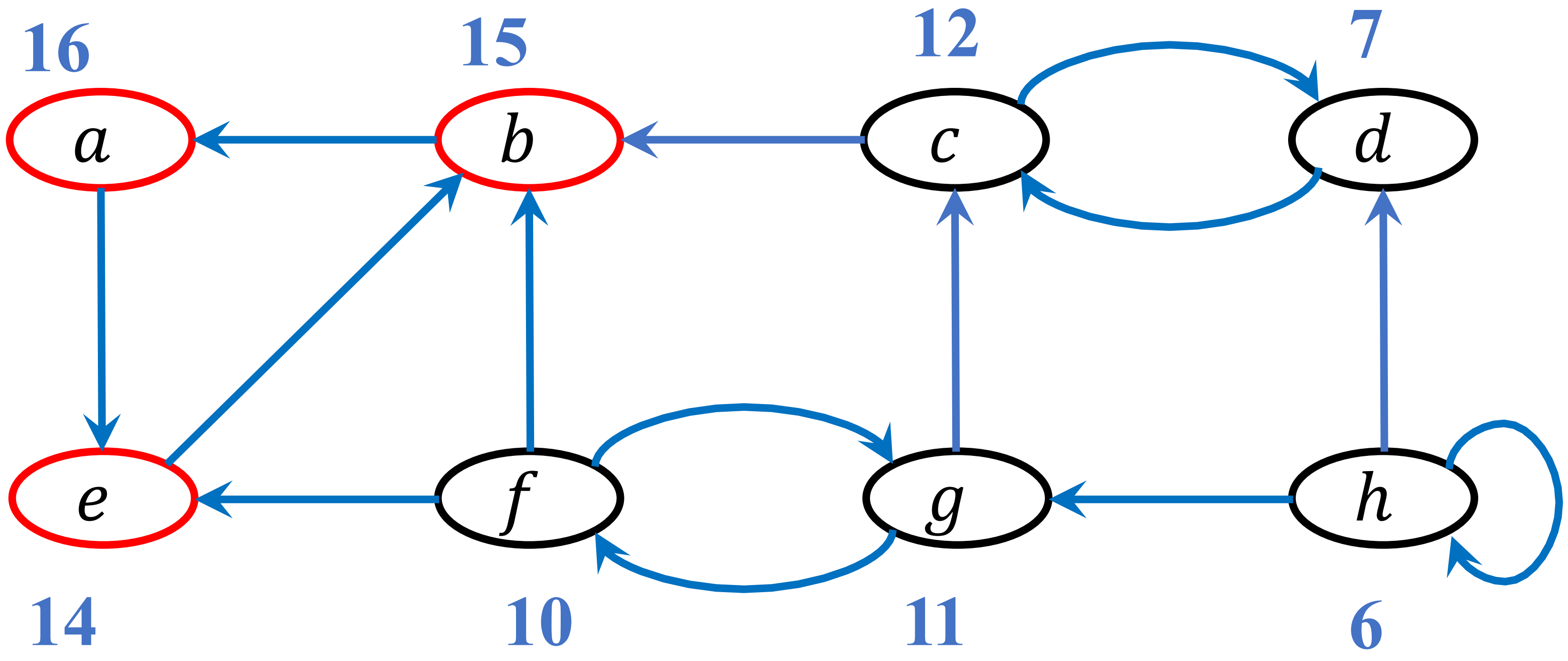
Algorithm

4. Each time we start with a node we put all reachable nodes from it in the same component.



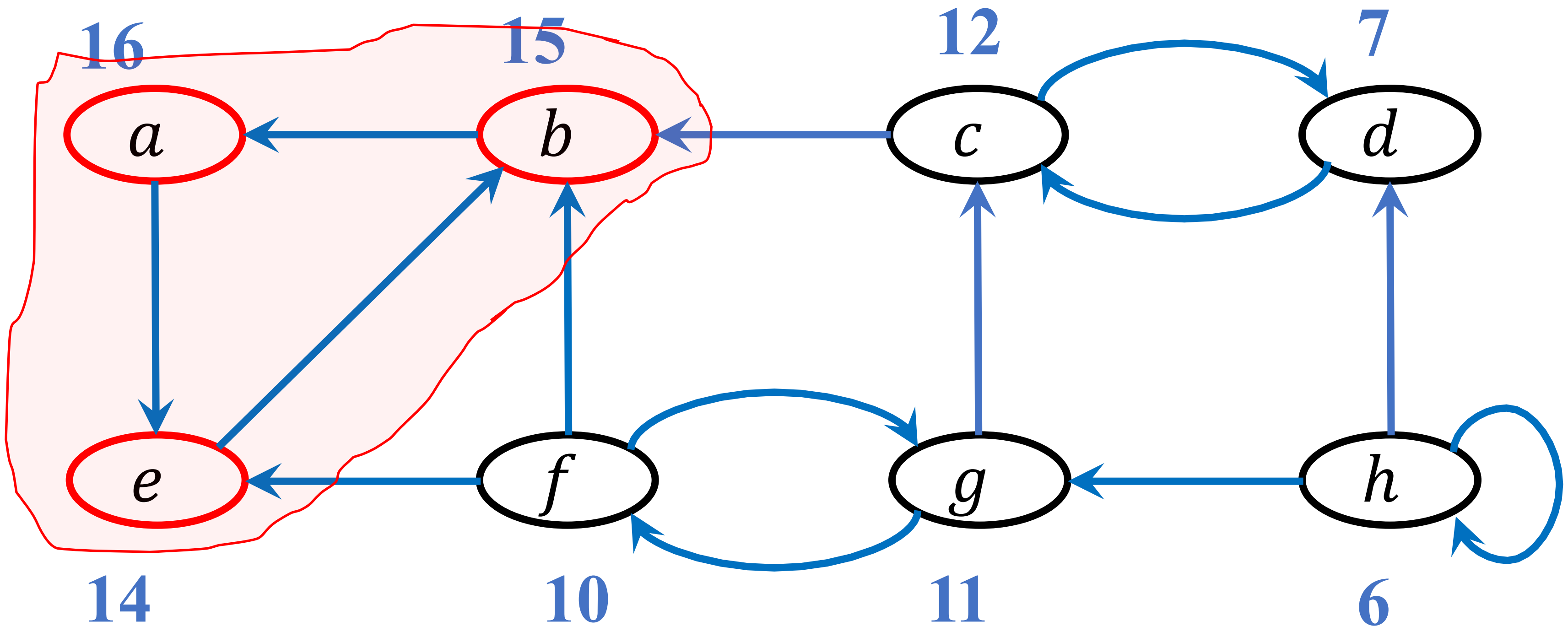
Algorithm

4. Each time we start with a node we put all reachable nodes from it in the same component.



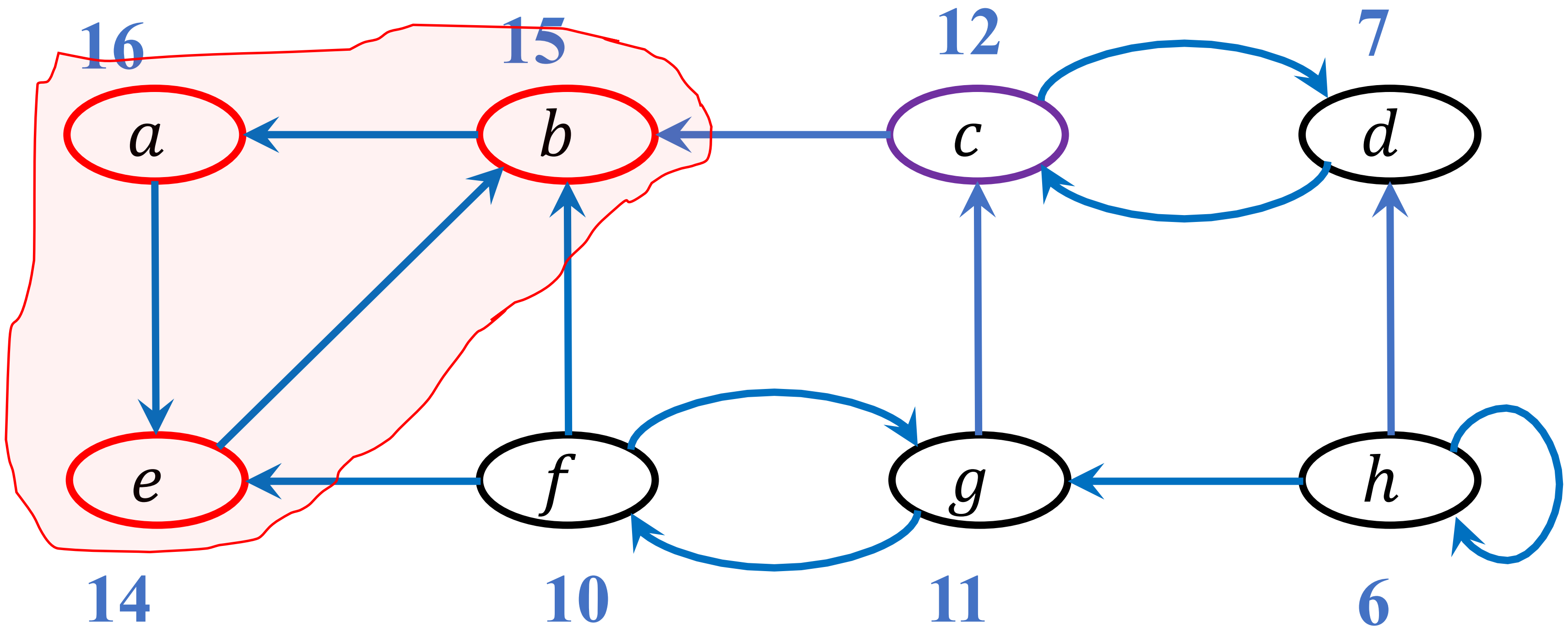
Algorithm

4. Each time we start with a node we put all reachable nodes from it in the same component.



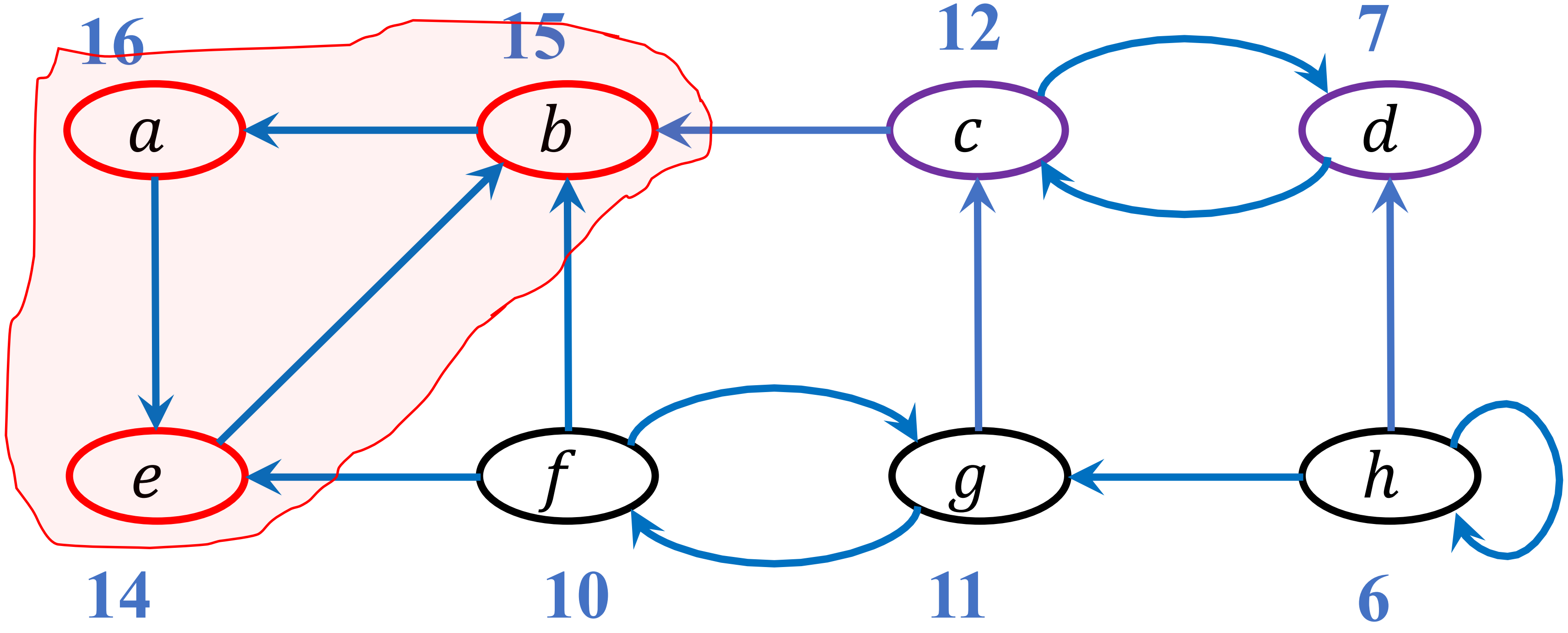
Algorithm

4. Each time we start with a node we put all reachable nodes from it in the same component.



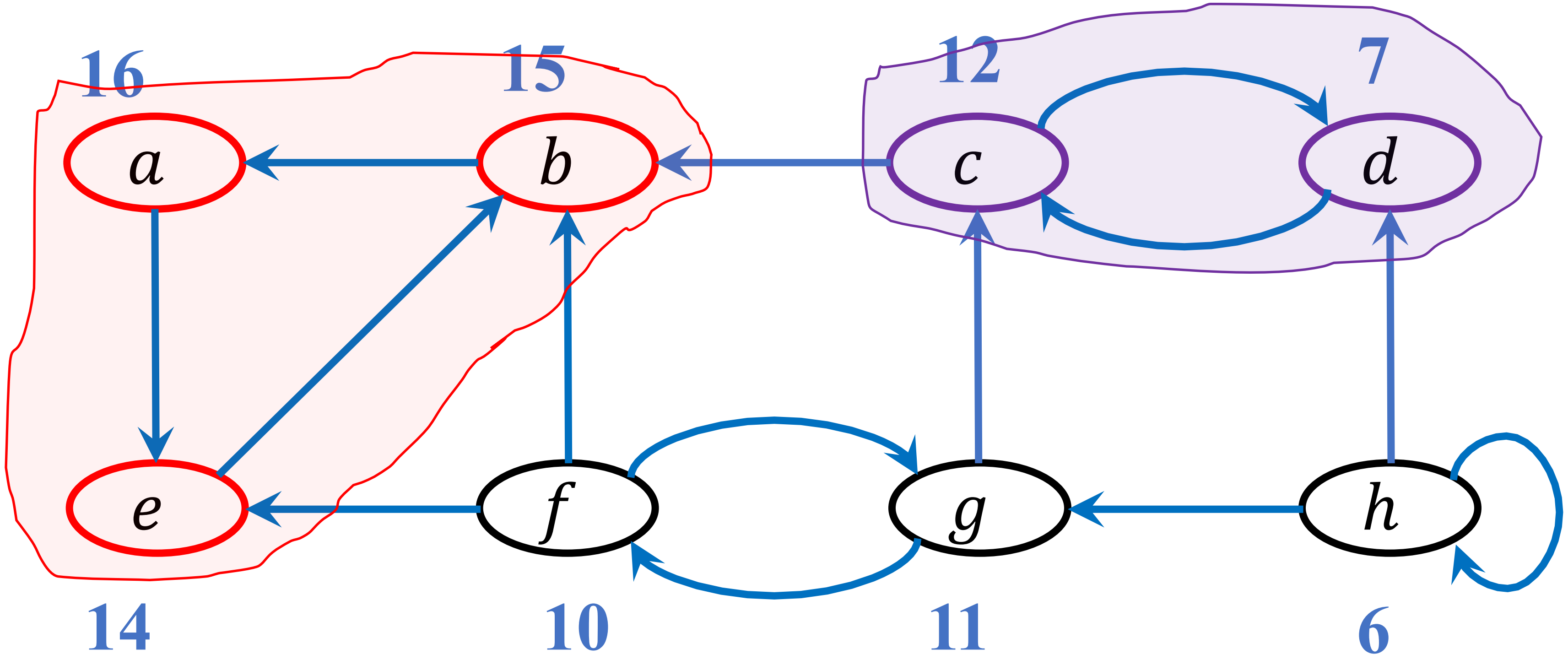
Algorithm

4. Each time we start with a node we put all reachable nodes from it in the same component.



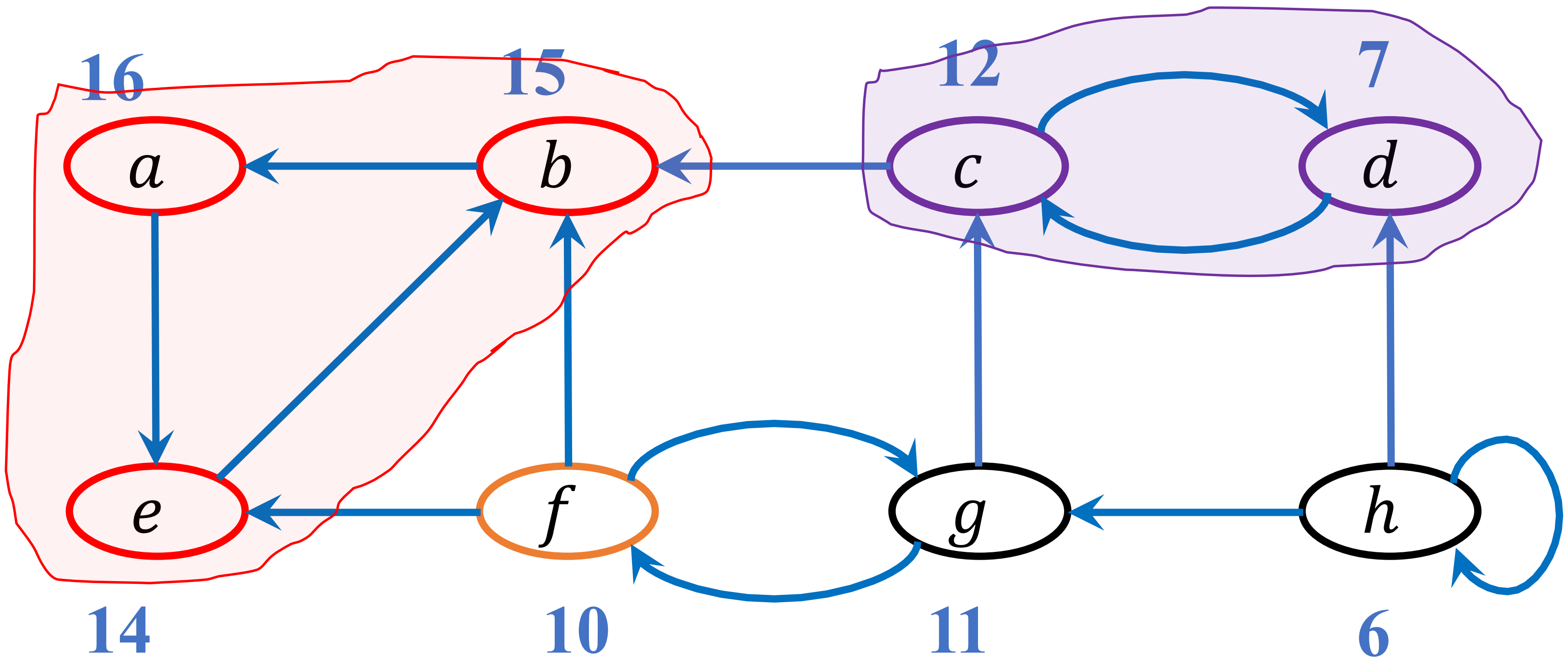
Algorithm

4. Each time we start with a node we put all reachable nodes from it in the same component.



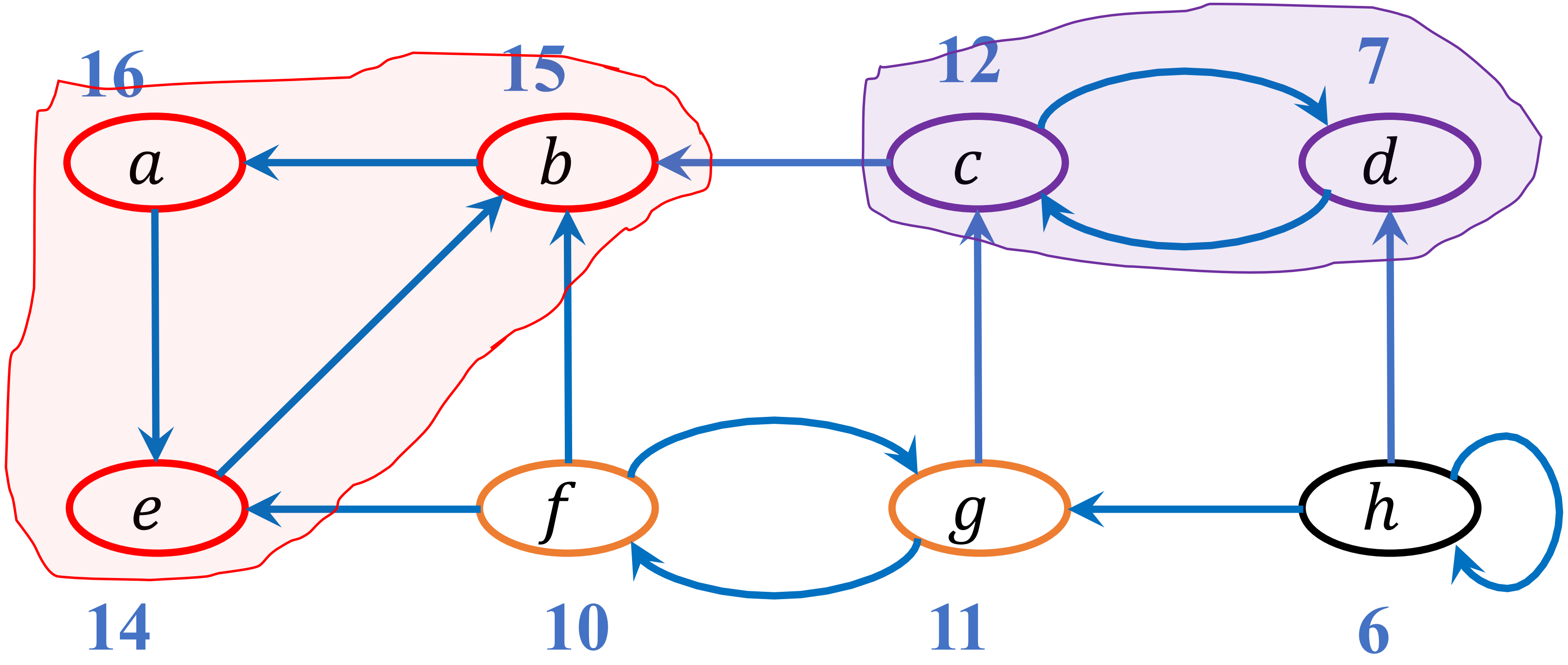
Algorithm

4. Each time we start with a node we put all reachable nodes from it in the same component.



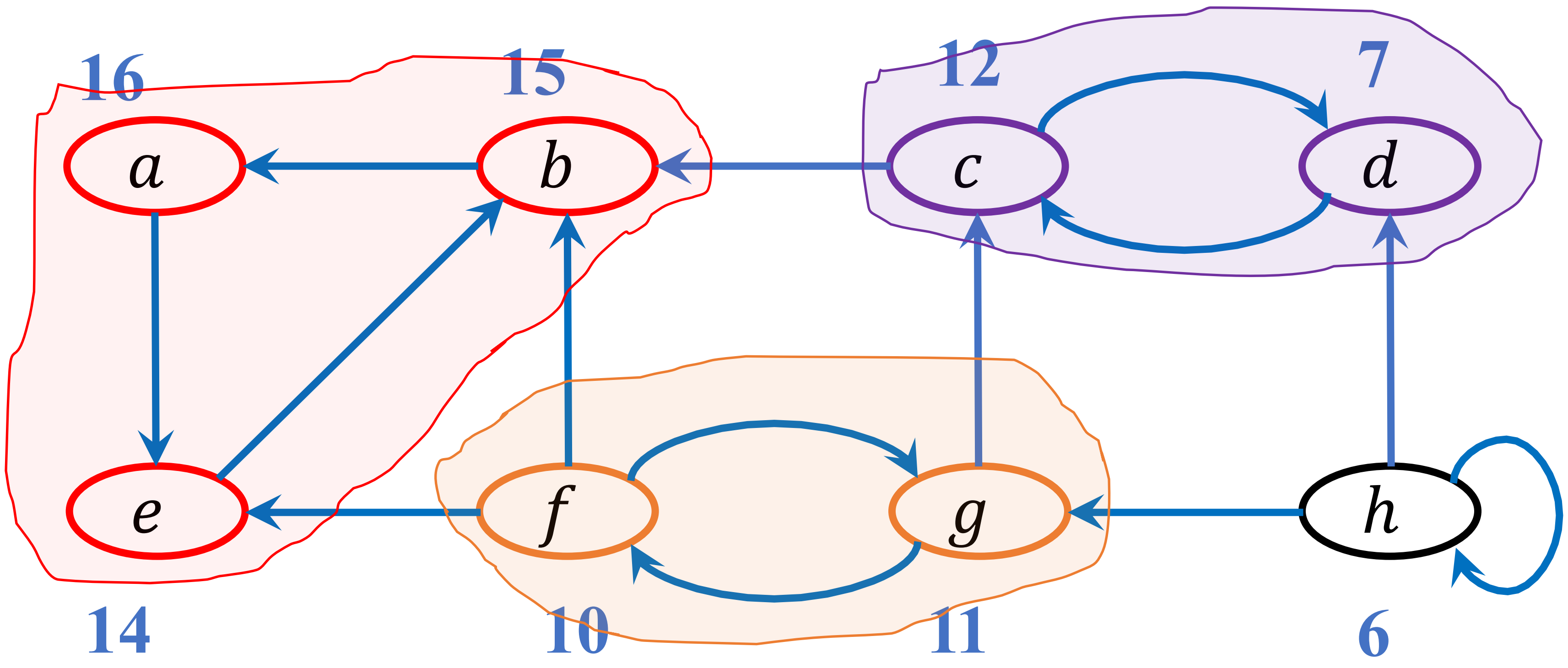
Algorithm

4. Each time we start with a node we put all reachable nodes from it in the same component.



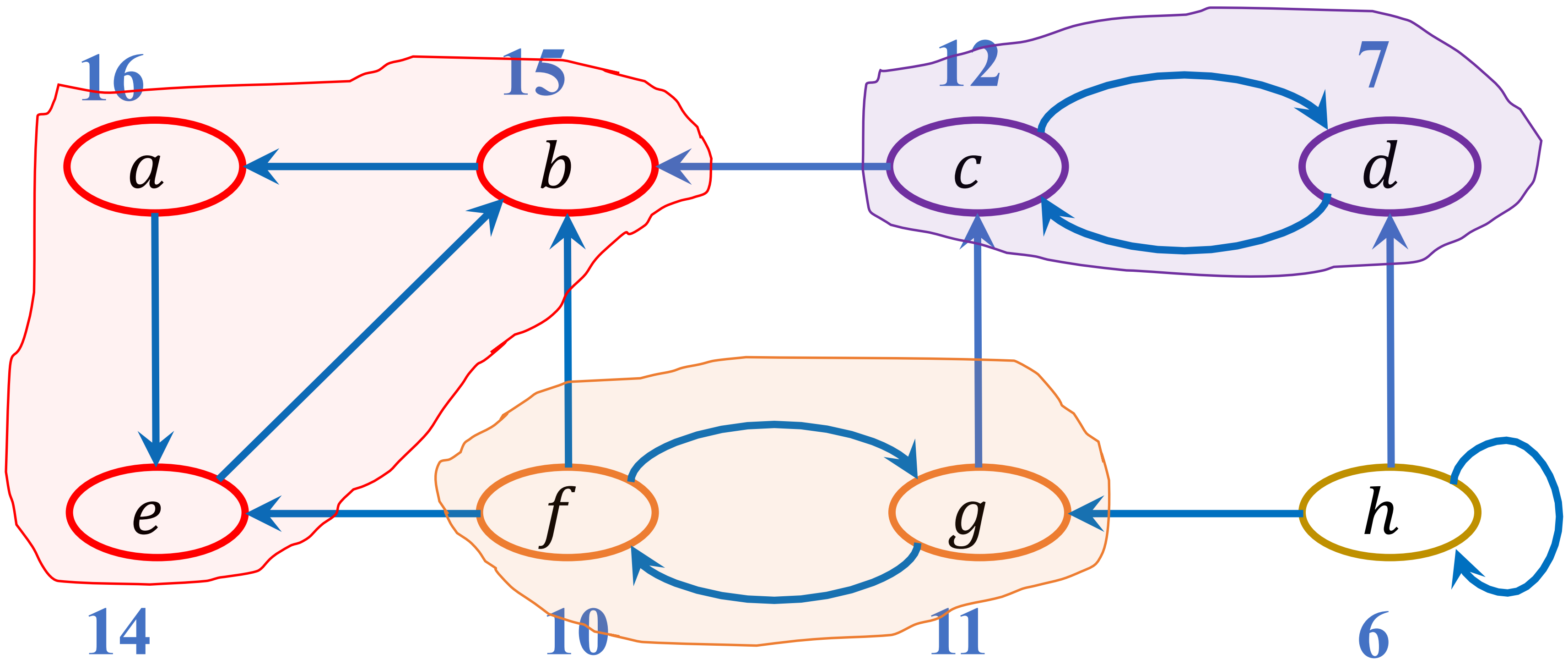
Algorithm

4. Each time we start with a node we put all reachable nodes from it in the same component.



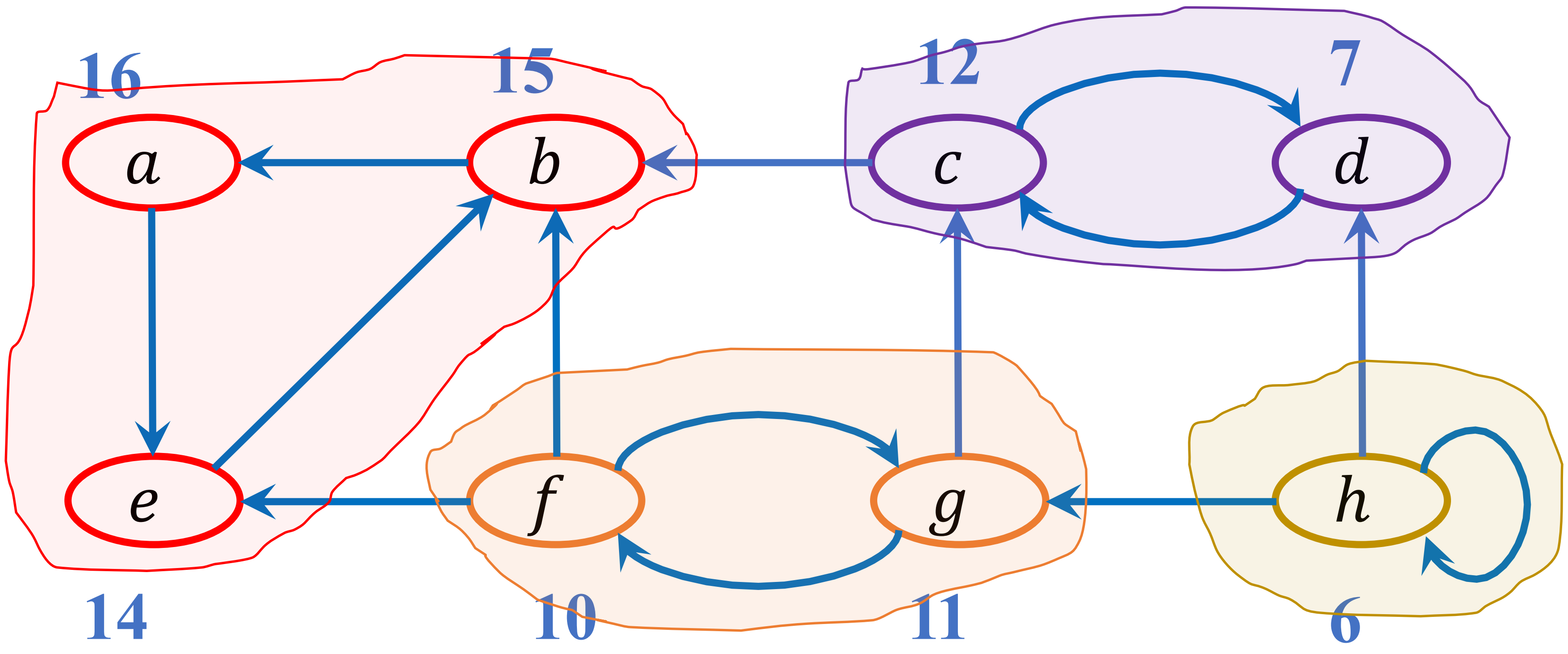
Algorithm

4. Each time we start with a node we put all reachable nodes from it in the same component.



Algorithm

4. Each time we start with a node we put all reachable nodes from it in the same component.



Analysis

1. The first DFS takes $O(n+m)$, and as we finish visiting the nodes we can put them in a linked list just like the topological sort so that we don't have to sort the nodes again.
2. Making the transpose graph also takes $O(n+m)$. We read the adjacency list of the original graph G, and each time we find an edge (u, v), we add the edge (v, u) to G^T by inserting u to v's adjacency list.
3. The second DFS also takes $O(n+m)$

- So, overall the algorithm works in linear time (with respect to the input size.)

Proof of correctness

Lemma 1: A pair of nodes u and v are put in the same component by the algorithm **if and only if** $u \leadsto v$ and $v \leadsto u$ in G.

Lemma 2: The components computed by the algorithm are maximal and hence strongly connected components.

Proof of correctness

Lemma 1: A pair of nodes u and v are put in the same component by the algorithm **if and only if** $u \rightsquigarrow v$ and $v \rightsquigarrow u$ in G.

Lemma 2: The components computed by the algorithm are maximal and hence strongly connected components.

Proof of Lemma 2: Assume that these components are not maximal. Then, for some component C there must be some node x outside C that $x \rightsquigarrow u$ and $u \rightsquigarrow x$. However, this is a contradiction since by Lemma 1 u and x are in the same component.

Proof of correctness

Proof of Lemma 1:

1. First we prove that if $u \rightsquigarrow v$ and $v \rightsquigarrow u$ in G, then u, v are in the same component:

Without loss of generality let's assume that u is visited first by the DFS on G. Since $u \rightsquigarrow v$, the finishing time of v will be smaller (similar to the argument for topological sort). So, $u.f > v.f$. As a result, when we run the DFS on G^T, u is picked first. Also, because there is path from v to u in G, there is path from u to v in G^T. Therefore, u will reach v in the second DFS and they are put in the same component.

Proof of correctness

Proof of Lemma 1:

2. Now, we prove that if u, v are in the same component, then we have $u \rightsquigarrow v$ and $v \rightsquigarrow u$ in G.

Proof of correctness

We use **proof by contrapositive** for this.

✓ **Proof by contrapositive:** It means that instead of prove the direct statement that **if A then B**, we prove the equivalent statement of **if not B then not A**.

- **Example**: The statement **if someone is tall, they play basketball**, is equivalent to **if someone doesn't play basketball then they're not tall.**

Proof of correctness

Proof of Lemma 1:

2. Now, we prove that if u, v are in the same component, then we have $u \rightsquigarrow v$ and $v \rightsquigarrow u$ in G.

Assume that in G either u can't reach v or v can't reach u (or both). We prove that u and v are in different components.

There are 2 cases:

Case 1: if none of u or v can reach the other then, in the second DFS (line 3), they will not be placed in the same component.

Proof of correctness

Proof of Lemma 1:

Case 2: without loss of generality we can just assume that u can reach v but v can't reach u. (The other case is symmetric)

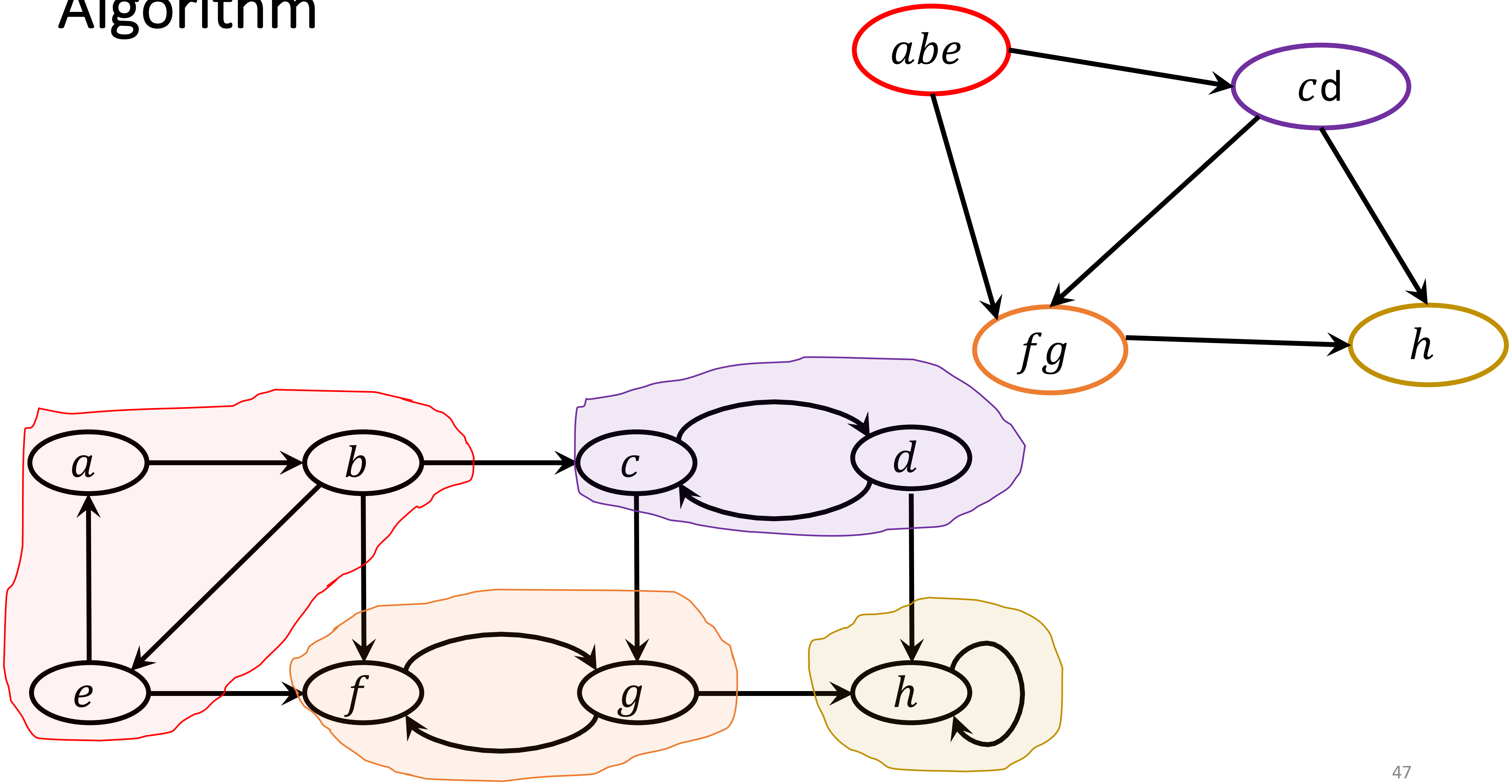
Similar to the argument for topological sort, regardless of whether the first DFS (line 1) picks u first or v first, the finishing time for v is smaller than u. ($u.f > v.f$)

So, in the second DFS (line 3), u is picked first. But since there is no path from v to u in G, there will be no path from u to v in G^T, and u and v will not be put in the same component.

Component graph

- After computing finding the components we sometimes simplify the graph by constructing the **component graph**.
- Each component corresponds to a node.
- Component A has an edge to another component B if in the **original graph**, there is a node in A that has an edge to a node in B.

Algorithm



Algorithm

Question: Can a component graph contain any cycles?