# *Design and analysis of algorithms Lecture 12 & 13*

Edyta Szymańska

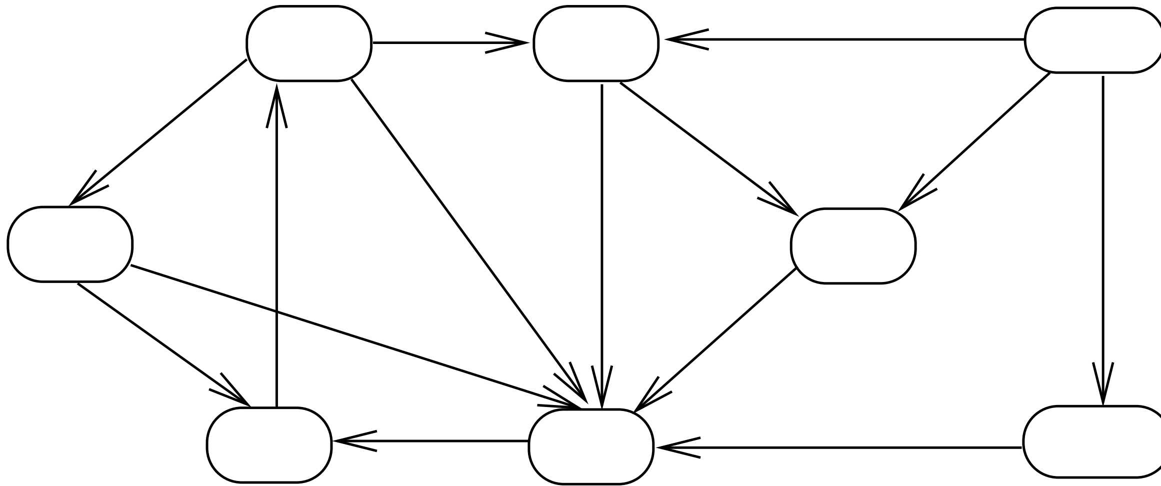
edyta@cc.gatech.edu.pl

# *DFS in directed graphs*

The same algorithm applies:

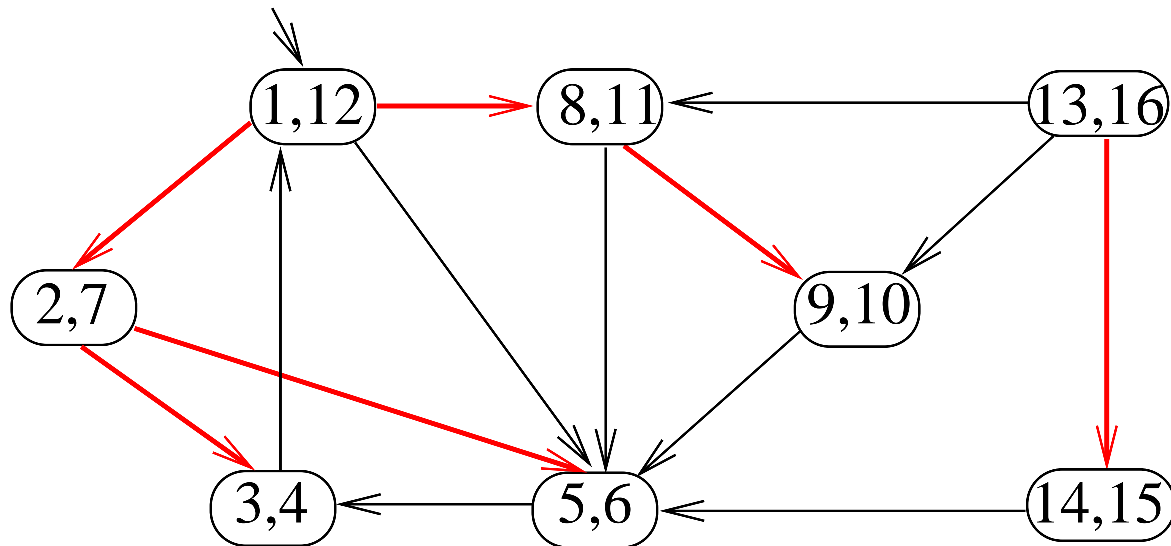
# DFS in directed graphs

The same algorithm applies:

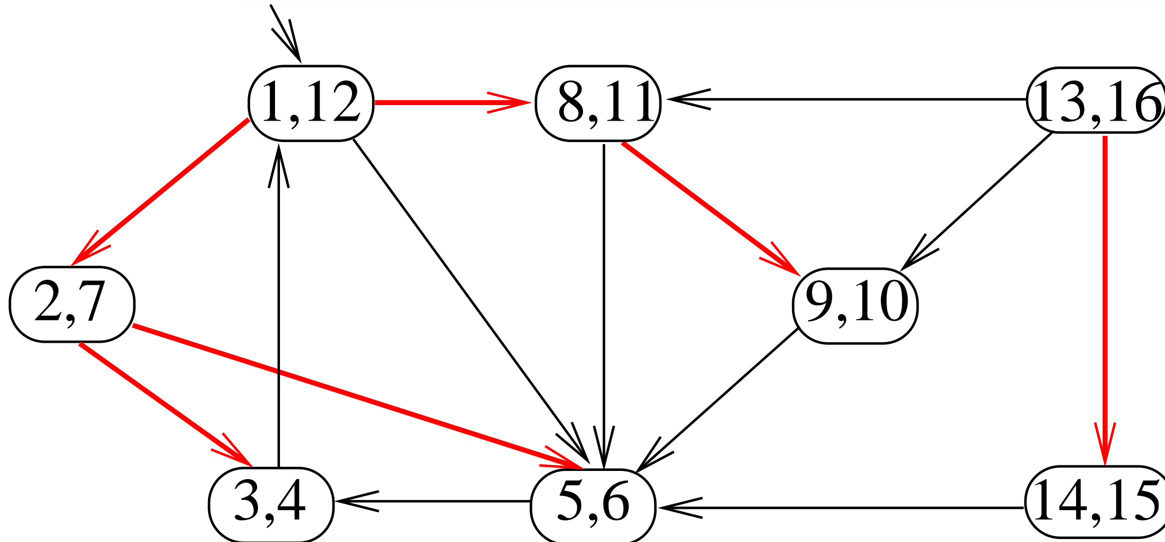


# DFS in directed graphs

The same algorithm applies:



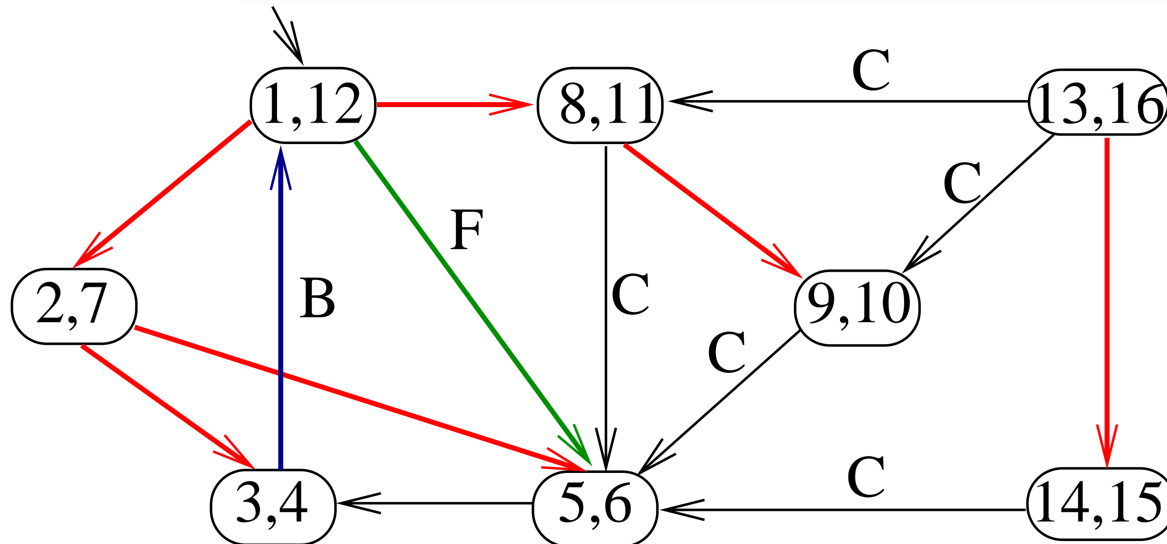
# *DFS in directed graphs*




Edge classification:

- TREE edge (T), part of DFS forest
- FORWARD edge (F), from a node to a non-child descendant in DFS tree
- BACK edge (B),leads to an ancestor in DFS tree
- CROSS edge (C), remainder, between trees and subtrees

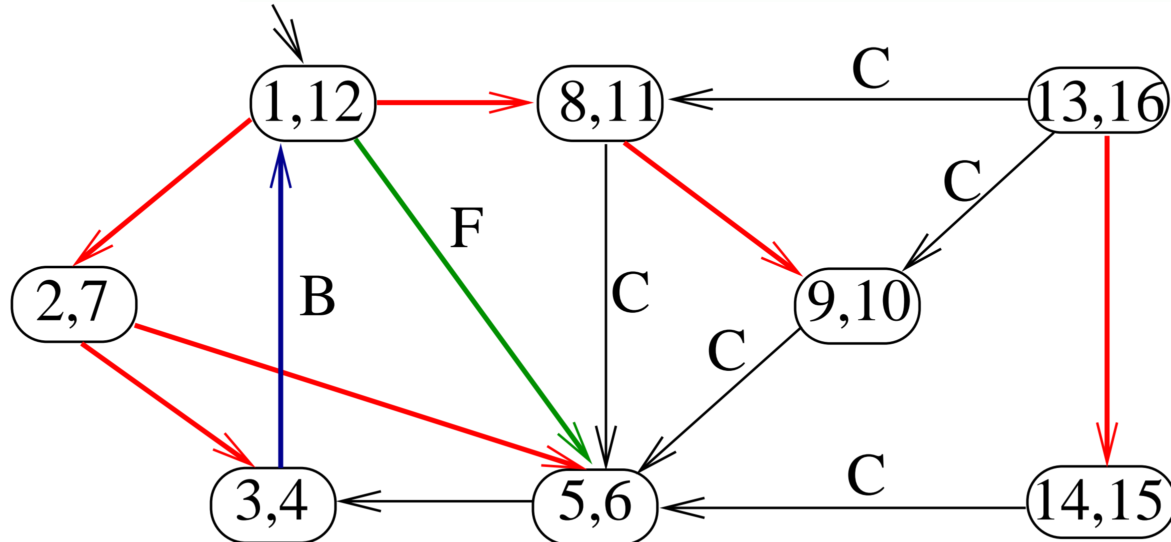
# *DFS in directed graphs*




Edge classification:

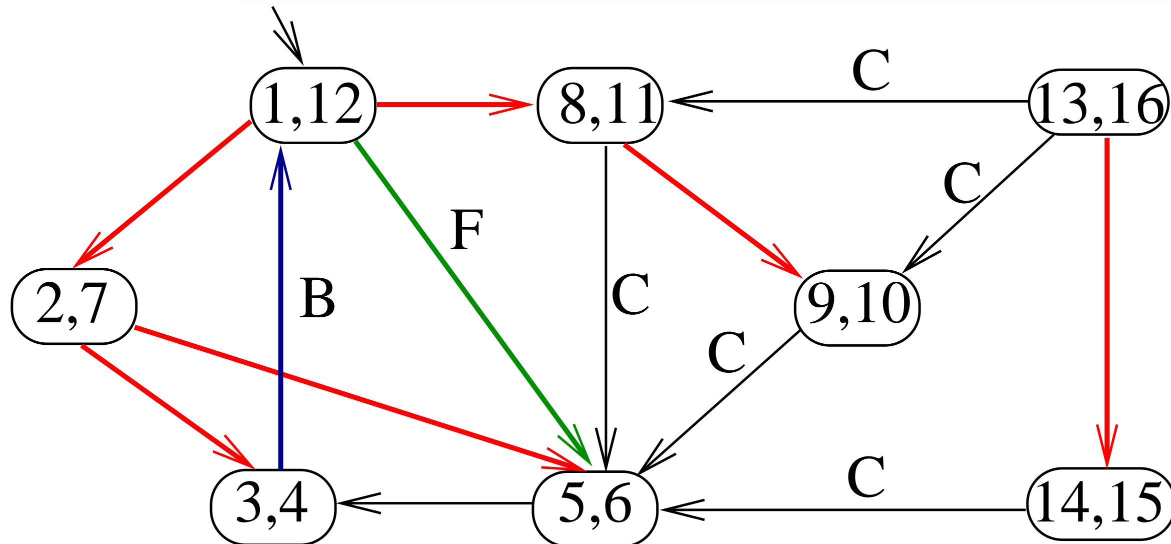
- TREE edge (T), part of DFS forest
- FORWARD edge (F), from a node to a non-child descendant in DFS tree
- BACK edge (B),leads to an ancestor in DFS tree
- CROSS edge (C), remainder, between trees and subtrees

# *DFS in directed graphs*




Note: in the undirected case we only have tree (T) and nontree edges(B).

## *DFS in directed graphs*




Note: in the undirected case we only have tree (T) and nontree edges(B).
We can classify edges based on the timestamps ($f[]$, and $d[]$).

# DFS in directed graphs - edge categories

| Type of edge | $d, f$ for $(u, v)$ | Part of DFS-tree? |
|---|---|---|
| Tree | $d[u] < d[v] < f[v] < f[u]$ | yes |
| Forward | $d[u] < d[v] < f[v] < f[u]$ | no |
| Back | $d[v] < d[u] < f[u] < f[v]$ | no |
| Cross | $d[v] < f[v] < d[u] < f[u]$ | no |

# DFS in directed graphs - applications

**Detecting cycles:**

# DFS in directed graphs - applications

**Detecting cycles:**

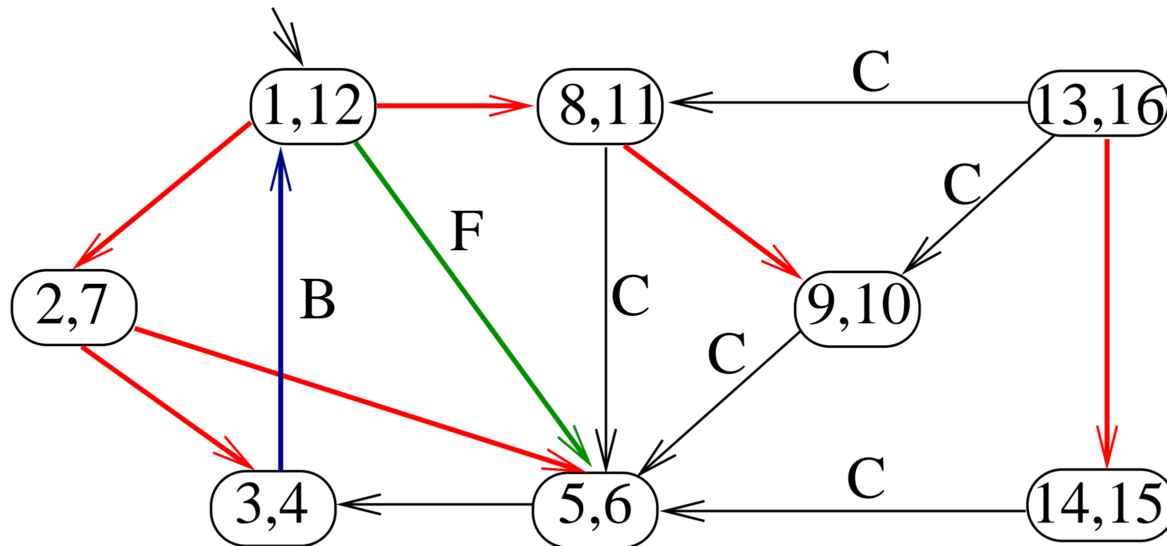
A cycle in a directed graph is a circular path
$v_0 \longrightarrow v_1 \longrightarrow \ldots \longrightarrow v_k \longrightarrow v_0$.

# DFS in directed graphs - applications

**Detecting cycles:**

A cycle in a directed graph is a circular path

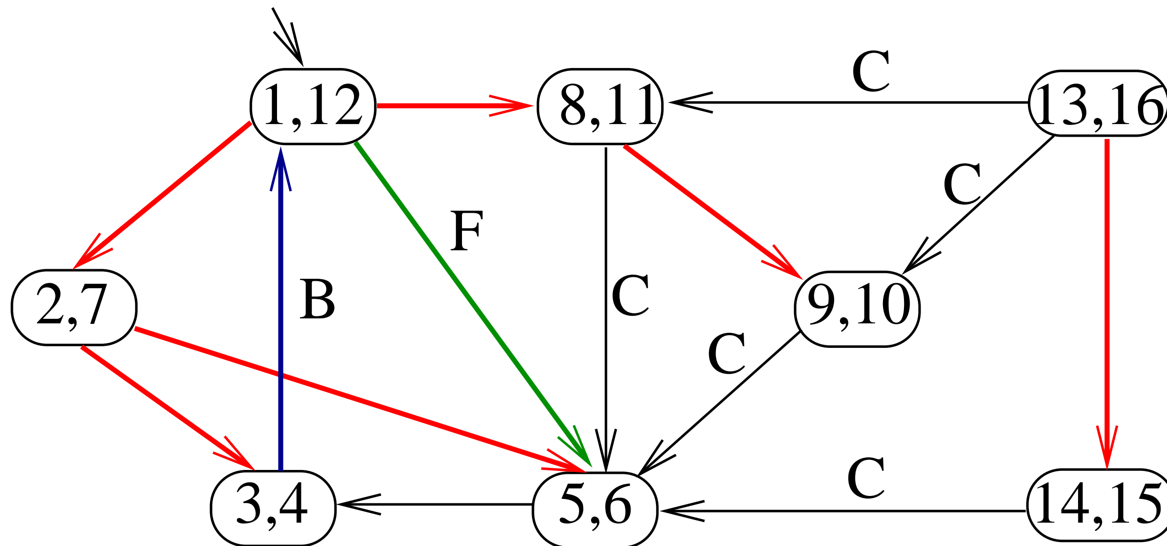
$v_0 \longrightarrow v_1 \longrightarrow \ldots \longrightarrow v_k \longrightarrow v_0.$



# DFS in directed graphs - applications

**Detecting cycles:**

A cycle in a directed graph is a circular path $v_0 \longrightarrow v_1 \longrightarrow \ldots \longrightarrow v_k \longrightarrow v_0$.




A directed graph has a cycle if and only if its depth-first search reveals a BACK edge.

# *DFS in directed graphs - applications*

**Detecting cycles:**

A cycle in a directed graph is a circular path
$v_0 \longrightarrow v_1 \longrightarrow \ldots \longrightarrow v_k \longrightarrow v_0$.

A directed graph has a cycle if and only if its depth-first search reveals a BACK edge.

A graph without a cycle is acyclic.

DAG- directed acyclic graph

# DFS in directed graphs - applications

**Detecting cycles:**

A cycle in a directed graph is a circular path $v_0 \longrightarrow v_1 \longrightarrow \ldots \longrightarrow v_k \longrightarrow v_0$.

A directed graph has a cycle if and only if its depth-first search reveals a BACK edge.

A graph without a cycle is acyclic.

DAG- directed acyclic graph

We can check if $G$ is a DAG using DFS, if BACK edge detected then CYCLE, time $O(n + m)$.

# DFS in directed graphs - applications

**Detecting cycles:**

A cycle in a directed graph is a circular path
$v_0 \longrightarrow v_1 \longrightarrow \ldots \longrightarrow v_k \longrightarrow v_0$.

A directed graph has a cycle if and only if its depth-first search reveals a BACK edge.

A graph without a cycle is acyclic.

DAG- directed acyclic graph

We can check if $G$ is a DAG using DFS, if BACK edge detected then CYCLE, time $O(n+m)$.

*Note: the same method works for cycles in undirected graphs, but faster* $O(n)$ *- why ?*

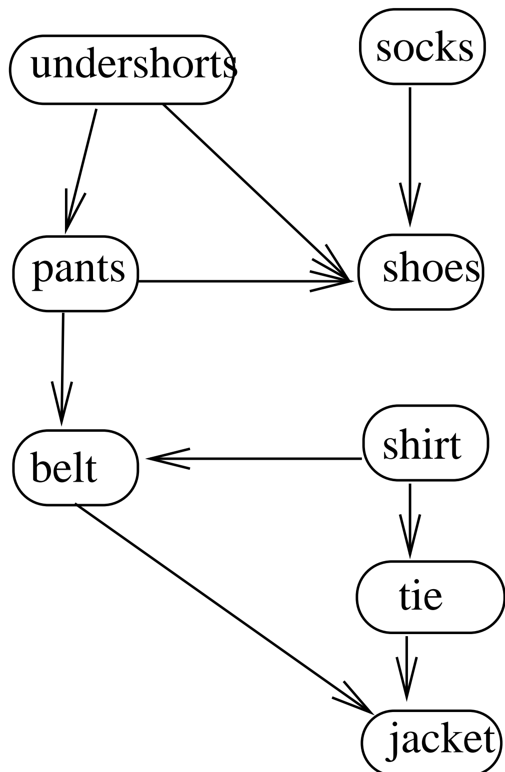
# *Topological Sort of a DAG*

By a topological sorting of a directed acyclic graph we understand a linear ordering of vertices such that if $(u, v) \in E$ then $u \leq v$ in the ordering.

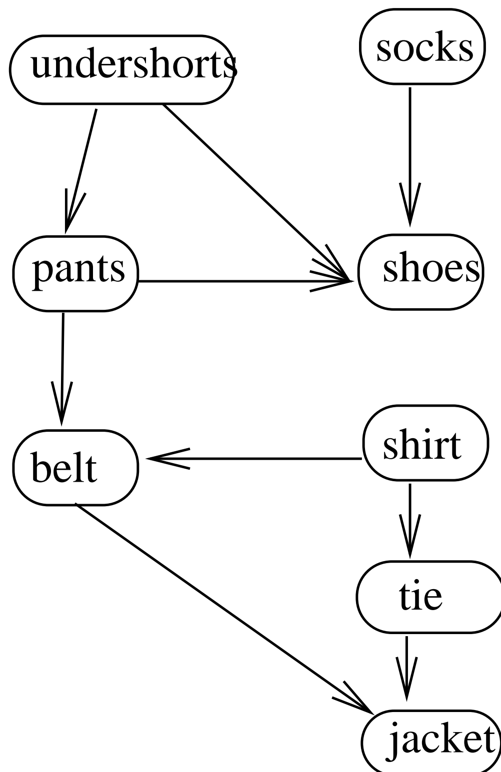
# ***Topological Sort of a DAG***

Example: *Getting dressed*

# Topological Sort of a DAG



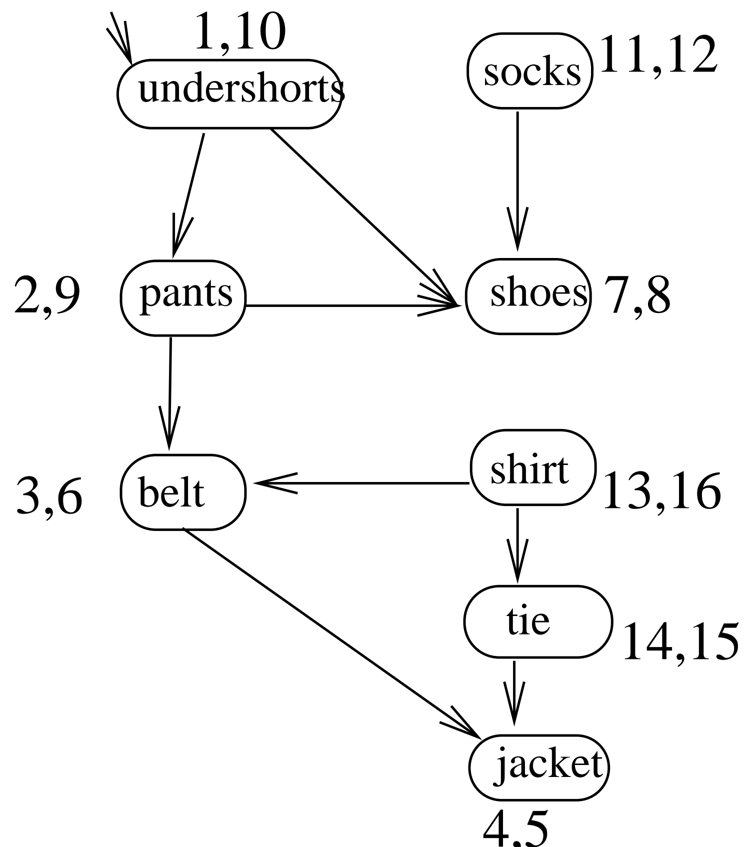
# *Topological Sort of a DAG*




TOPOLOGICAL SORT$(G)$

1. Run DFS$(G)$
2. as a vertex $u$ is finished ($f[u]$ computed), output it in front
3. return the list

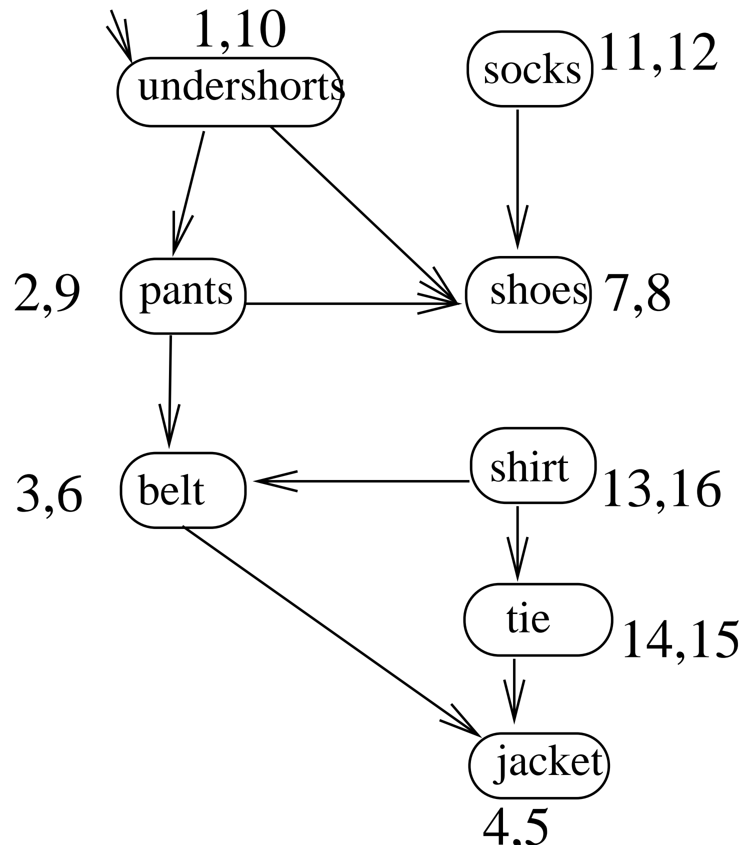
# *Topological Sort of a DAG*

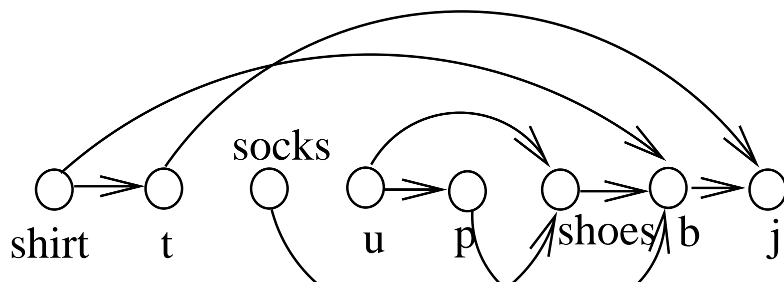



TOPOLOGICAL SORT$(G)$

1. Run DFS$(G)$
2. as a vertex $u$ is finished ($f[u]$ computed), output it in front
3. return the list

# *Topological Sort of a DAG*

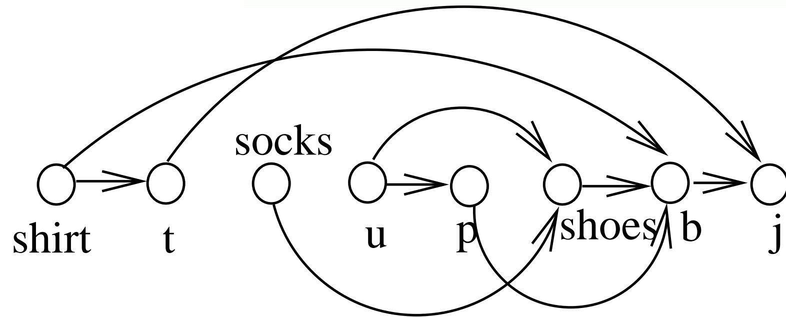


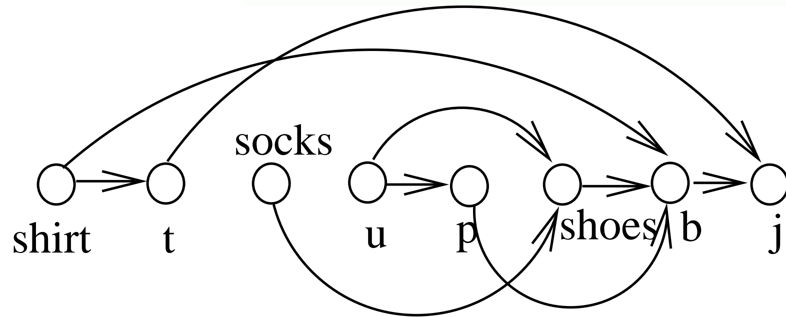

TOPOLOGICAL SORT$(G)$

1. Run DFS$(G)$
2. as a vertex $u$ is finished ($f[u]$ computed), output it in front
3. return the list

# Topological Sort of a DAG - analysis

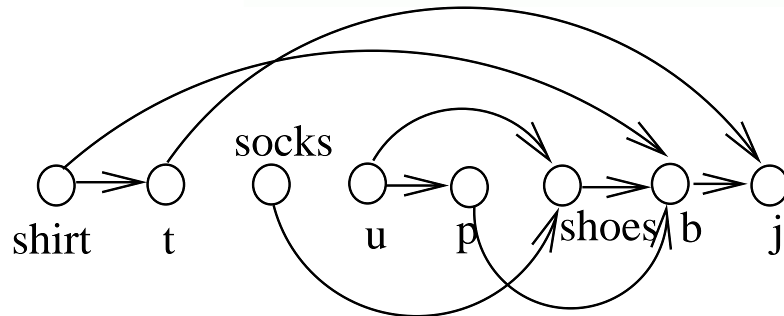


# *Topological Sort of a DAG - analysis*




**Correctness:**

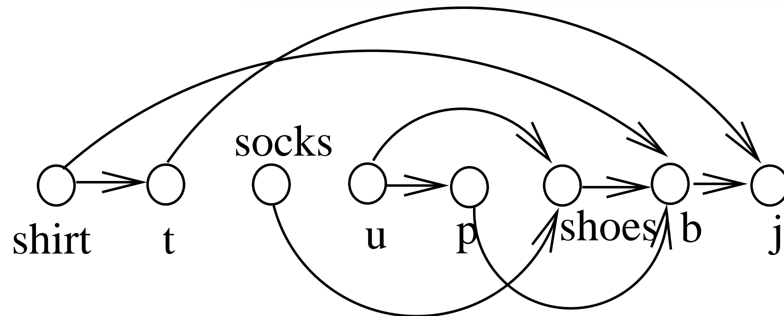
# Topological Sort of a DAG - analysis




**Correctness:**

it is enough to show that *for every pair* $u, v$, *if there is an edge* $(u, v) \in E$ *then* $f[u] > f[v]$.

# *Topological Sort of a DAG - analysis*



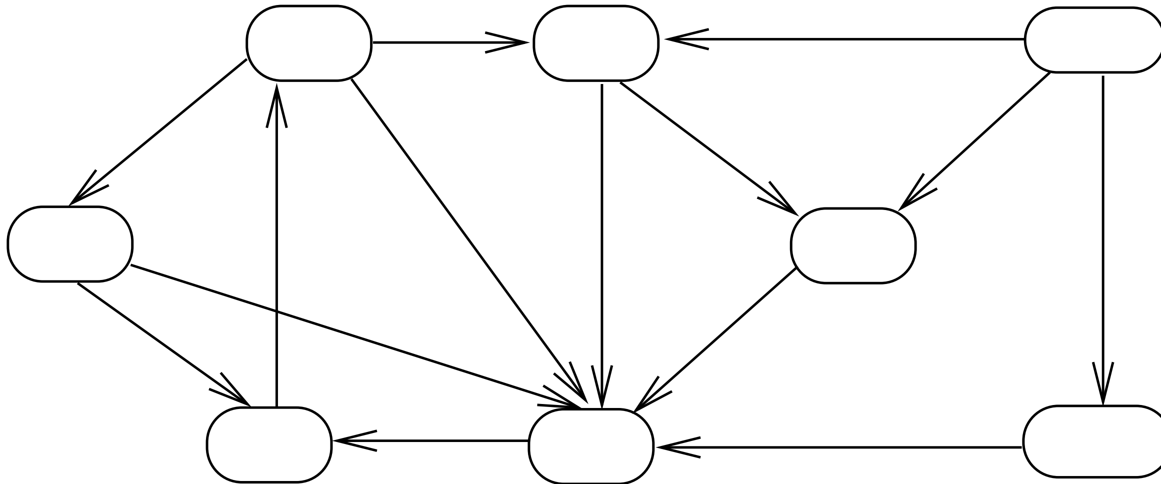

**Correctness:**
it is enough to show that *for every pair* $u, v$, *if there is an edge* $(u, v) \in E$ *then* $f[u] > f[v]$.
Proof: the only $(u, v)$ edge such that $f[u] < f[v]$ is a BACK edge but $G$ is a DAG and has no B edges.

# *DFS in directed graphs - applications*

**Strong connectivity:**

# *DFS in directed graphs - applications*

**Strong connectivity:**



# *DFS in directed graphs - applications*

**Strong connectivity:**

A directed graph is strongly connected if for every pair of nodes $u, v$ there is a path from $u$ to $v$ and a path from $v$ to $u$.

# DFS in directed graphs - applications

**Strong connectivity:**

A directed graph is strongly connected if for every pair of nodes $u, v$ there is a path from $u$ to $v$ and a path from $v$ to $u$. The relation between nodes is an equivalence relation and partitions $V$ into disjoint sets called strongly connected components (SCC).

# *DFS in directed graphs - applications*

**Strong connectivity:**

A directed graph is strongly connected if for every pair of nodes $u, v$ there is a path from $u$ to $v$ and a path from $v$ to $u$. The relation between nodes is an equivalence relation and partitions $V$ into disjoint sets called strongly connected components (SCC).

How can we use DFS to check if a directed $G$ is strongly connected?

# DFS in directed graphs - applications

**Strong connectivity:**

A directed graph is strongly connected if for every pair of nodes $u, v$ there is a path from $u$ to $v$ and a path from $v$ to $u$.
The relation between nodes is an equivalence relation and partitions $V$ into disjoint sets called strongly connected components (SCC).
How can we use DFS to check if a directed $G$ is strongly connected?
Run DFS on $G$ and on $G^R$($G$ with all edges reversed. Both have the same SCC's (why?)).

# DFS in directed graphs - applications

**Strong connectivity:**

A directed graph is strongly connected if for every pair of nodes $u, v$ there is a path from $u$ to $v$ and a path from $v$ to $u$.

The relation between nodes is an equivalence relation and partitions $V$ into disjoint sets called strongly connected components (SCC).

How can we use DFS to check if a directed $G$ is strongly connected?

Run DFS on $G$ and on $G^R$($G$ with all edges reversed. Both have the same SCC's (why?)).

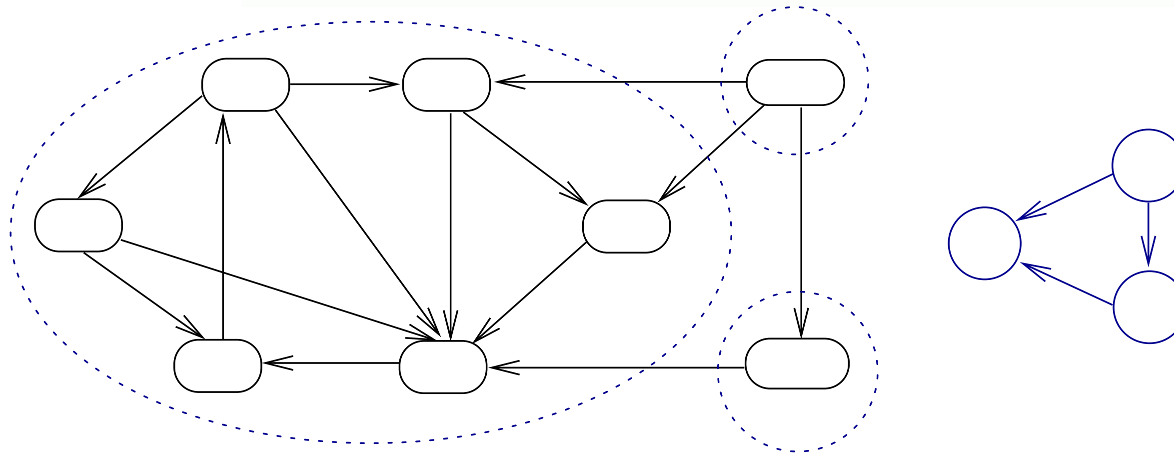
How can we identify its strongly connected components? - more advanced!

# DFS in directed graphs - applications

**Strong connectivity:**
A directed graph is strongly connected if for every pair of nodes $u, v$ there is a path from $u$ to $v$ and a path from $v$ to $u$.
The relation between nodes is an equivalence relation and partitions $V$ into disjoint sets called strongly connected components (SCC).
How can we use DFS to check if a directed $G$ is strongly connected?
Run DFS on $G$ and on $G^R$($G$ with all edges reversed. Both have the same SCC's (why?)).
How can we identify its strongly connected components? - more advanced!
**Motivation:** searching for *mutual recursions* in program debugging.

# Strong connectivity - useful properties

# Strong connectivity - useful properties



- If we shrink each strongly connected component (SCC) down to a single node and draw an edge if there is any edge between the corresponding components then the resulting graph is a DAG.

# Strong connectivity - useful properties

- If we shrink each strongly connected component (SCC) down to a single node and draw an edge if there is any edge between the corresponding components then the resulting graph is a DAG.
- $v$ is a sink if $d_{out}(v) = 0$ and $v$ is a source if $d_{in}(v) = 0$.

# Strong connectivity - useful properties

- If we shrink each strongly connected component (SCC) down to a single node and draw an edge if there is any edge between the corresponding components then the resulting graph is a DAG.
- $v$ is a sink if $d_{out}(v) = 0$ and $v$ is a source if $d_{in}(v) = 0$.
- Every DAG contains a sink and a source

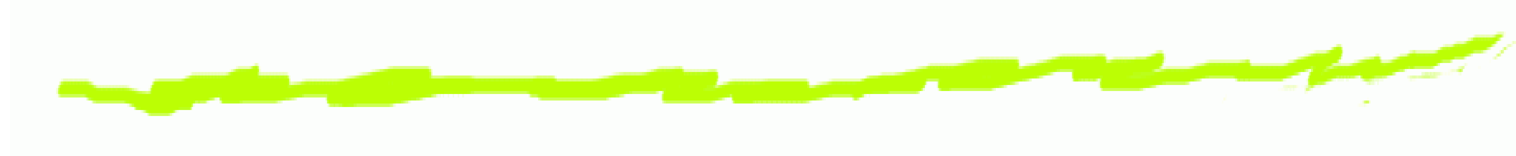
# Strong connectivity - useful properties

- If we shrink each strongly connected component (SCC) down to a single node and draw an edge if there is any edge between the corresponding components then the resulting graph is a DAG.
- $v$ is a sink if $d_{out}(v) = 0$ and $v$ is a source if $d_{in}(v) = 0$.
- Every DAG contains a sink and a source
- Identifying a vertex in a sink of a DAG(SCC) would lead us to a component corresponding to it - not easy, but source is easier!

# Strong connectivity - useful properties

- If we shrink each strongly connected component (SCC) down to a single node and draw an edge if there is any edge between the corresponding components then the resulting graph is a DAG.
- $v$ is a sink if $d_{out}(v) = 0$ and $v$ is a source if $d_{in}(v) = 0$.
- Every DAG contains a sink and a source
- Identifying a vertex in a sink of a DAG(SCC) would lead us to a component corresponding to it - not easy, but source is easier!

# Algorithm for strongly connected components



# *Algorithm for strongly connected components*

Fact 1: the node which receives the highest finishing time in DFS must lie in a source connected component.

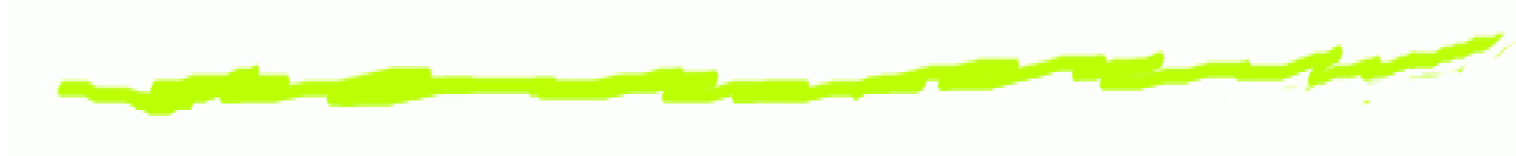
# Algorithm for strongly connected components

Fact 1: the node which receives the highest finishing time in DFS must lie in a source connected component.
Follows from the following:
Fact 2: *If $C$ and $C'$ are strongly connected components, and there is an edge from a node in $C$ to a node in $C'$, then the highest finishing time in $C$ is bigger than the highest finishing time in $C'$*

# *Algorithm for strongly connected components*

Fact 1: the node which receives the highest finishing time in DFS must lie in a source connected component.
Follows from the following:
Fact 2: *If $C$ and $C'$ are strongly connected components, and there is an edge from a node in $C$ to a node in $C'$, then the highest finishing time in $C$ is bigger than the highest finishing time in $C'$*
Fact 2 (restated): *the strongly connected components can be topologically sorted by arranging them in decreasing order of their highest finishing time.*

# An efficient algorithm for strong connectivity:



# *An efficient algorithm for strong connectivity:*

STRONG_CONNECTIVITY$(G)$

1. Run DFS on $G^R$ ($G$ with all edges reversed)
2. Run DFS on $G$ considering the vertices in decreasing order of their finishing time computed in step 1.
3. Output the vertices of each tree formed in step 2 as a separate strongly connected component.

# An efficient algorithm for strong connectivity:

STRONG_CONNECTIVITY$(G)$

1. Run DFS on $G^R$ ($G$ with all edges reversed)
2. Run DFS on $G$ considering the vertices in decreasing order of their finishing time computed in step 1.
3. Output the vertices of each tree formed in step 2 as a separate strongly connected component.

**Running time:**
computing $G^R$: $O(m+n)$
DFS($G^R$): $O(m+n)$
DFS($G$): $O(m+n)$
**total: :** $O(m+n)$