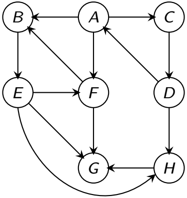
CMP_SC 3050: Directed Graphs

Rohit Chadha

October 1, 2014

Directed Graphs

Directed graphs capture asymmetric pairwise relationships amongst objects



A directed graph (also called a digraph) is $G = (V, E)$, where

- V is a set of vertices or nodes
- E is set of ordered pairs of vertices called edges

Examples of Digraphs

Informational Networks

The vertices of the world-wide web, viewed as a graph, are web pages and there is an edge from x to y, if x has a hyperlink to y

Dependency Networks

Vertices are tasks, and an edge from task x to task y denotes that y depends on task x

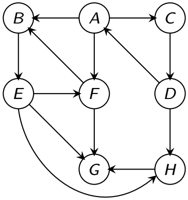
Program Analysis

Functions/procedures form the vertices and an edge from vertex x to vertex y denotes that procedure x can call y

Some terminology

- The end points of an edge (u, v) are the vertices u and v
- A self-loop is an edge both of whose endpoints are the same
- An edge (u, v) is said to be incident from vertex u and incident to v
- The out-degree of a vertex is the number of edges incident from it
- The in-degree of a vertex is the number of edges incident to it
- A vertex v is said to be adjacent to vertex u if there is an edge (u, v) in the graph

Example



$$\begin{aligned}
\text{in-degree}(A) &= 1\\
\text{out-degree}(A) &= 3\\
\text{in-degree}(B) &= 2\\
\text{out-degree}(B) &= 1
\end{aligned}$$

Paths and Cycles

- A path P in a graph is a sequence of vertices $v_1, v_2, \ldots v_k$ such that for every i $(1 \leq i < k)$, (v_i, v_{i+1}) is a directed edge in the graph, and all vertices are distinct. In such a case, P is a path from v_1 to v_k and is of length $k - 1$
- Distance of a vertex v from a vertex u is the length of the shortest path from u to v
- A cycle is a closed path of length ≥ 1 and can be defined analogously to the undirected case

Connectivity

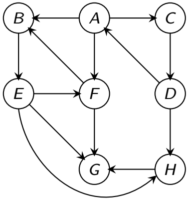
We will say a vertex v is reachable from u, if there is a directed path from u to v

$\mathrm{rch}(u) = \{v \mid v \text{ is reachable from } u\}$

Unlike the undirected case, it is possible that

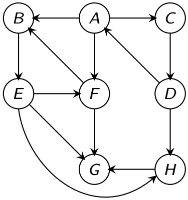
- u is reachable from v but v is not reachable from u

Example



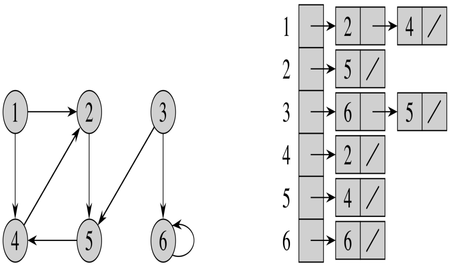
Example

$$
\begin{aligned}
\mathrm{rch(A)} &= \{A, B, C, D, E, F, G, H\} \\
\mathrm{rch(B)} &= \{B, E, F, G, H\} \\
\mathrm{rch(C)} &= \{A, B, C, D, E, F, G, H\} \\
\mathrm{rch(D)} &= \{A, B, C, D, E, F, G, H\} \\
\mathrm{rch(E)} &= \{B, E, F, G, H\} \\
\mathrm{rch(F)} &= \{B, E, F, G, H\} \\
\mathrm{rch(G)} &= \{G\} \\
\mathrm{rch(H)} &= \{H, G\}
\end{aligned}
$$



Digraph Representation

Can use Adjacency matrix or Adjacency List representation



(a) (b)

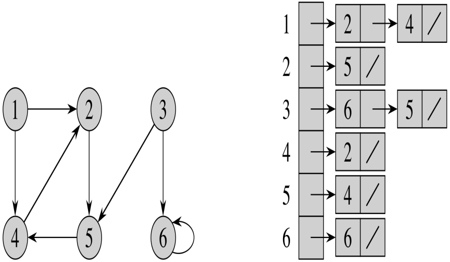
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

- We will allow self-loops in directed graphs

Digraph Representation

Can use Adjacency matrix or Adjacency List representation



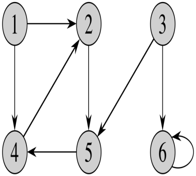
$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
$$

(a) (b) (c)

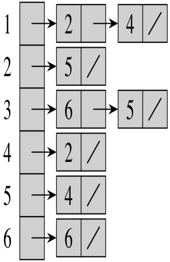
- We will allow self-loops in directed graphs
- Unlike the undirected case, the adjacency matrix is not symmetric

Digraph Representation

Can use Adjacency matrix or Adjacency List representation



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

- We will allow self-loops in directed graphs
- Unlike the undirected case, the adjacency matrix is not symmetric
- As in the undirected case, we will assume that the graph is usually presented in the adjacency list representation.

Connectivity Problems

Important Basic Algorithmic Questions

- Given graph G and vertices s and t, can you reach t from s?
- Given graph G and vertex u, compute $\mathrm{rch}(u)$

Connectivity Problems

Important Basic Algorithmic Questions

- Given graph G and vertices s and t, can you reach t from s?
- Given graph G and vertex u, compute $\mathrm{rch}(u)$

BFS and DFS

- Both the questions can be solved by performing either a BFS or a DFS traversal on the directed graphs
- These algorithms are identical to the case of undirected graphs
- BFS and DFS Trees are also defined in the same way except now the edges are directed

DFS of a graph

Till now, DFS used to search vertices reachable from a source vertex. We extend DFS to the whole graph!

1. Pick a vertex
2. Perform a DFS on the picked vertex
3. Pick a new vertex which has not been reached as yet and repeat Step 2

Pseudocode for DFS of a graph with visit times

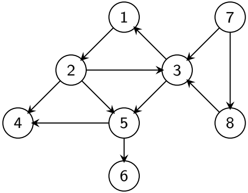
$\text{DFS}(V, E)$

```
for each u ∈ V
    u.color = WHITE
time = 0
for each u ∈ V
    if u.color == WHITE
        DFS_VISIT(V, E, u)
```

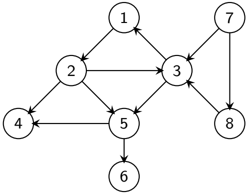
$\text{DFS\_VISIT}(V, E, u)$

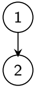
```
time = time + 1
u.d = time
u.color = GRAY
for each v ∈ G.Adj[u]
    if v.color == WHITE
        DFS_VISIT(V, E,v)
u.color = BLACK
time = time + 1
u.f = time
```

Example

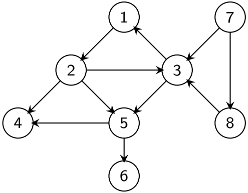


Example

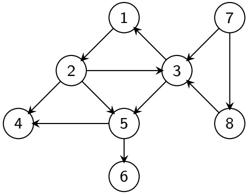


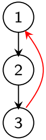


Example

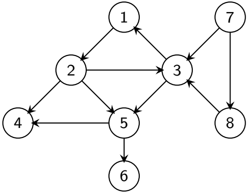


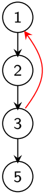
Example



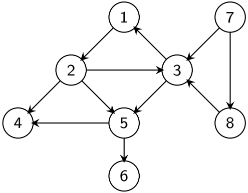


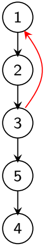
Example



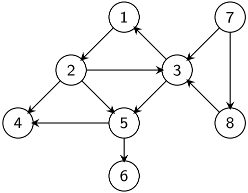


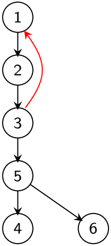
Example



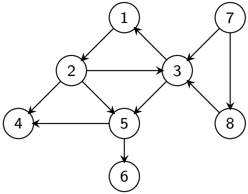


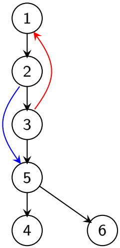
Example



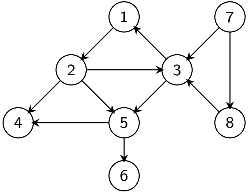


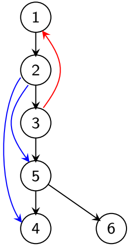
Example



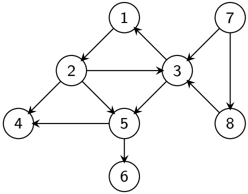


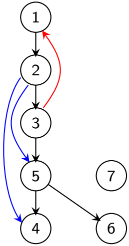
Example



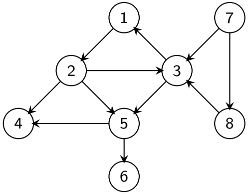


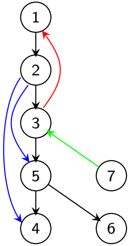
Example



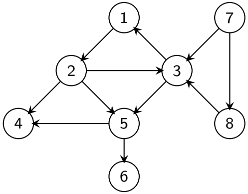


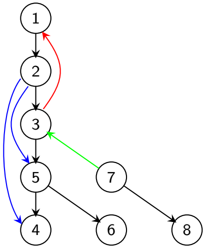
Example



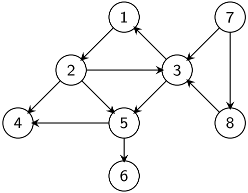


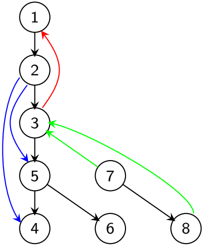
Example



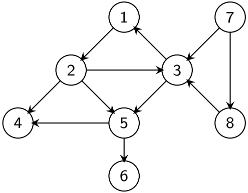


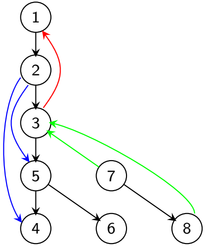
Example





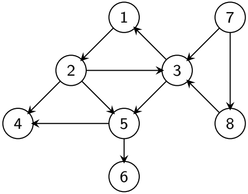
Example

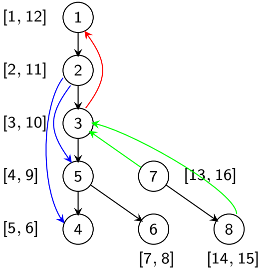




- Black edges form the DFS forest (and not a tree)
- Note all the edges drawn in the forest also belong to the original graph
- The color coding is there for a purpose

Example with Visit Times





Types of Edges

With respect to a DFS forest T, the edges of graph G can be classified as follows.

- Tree Edges are edges of G that appear in T

Types of Edges

With respect to a DFS forest T, the edges of graph G can be classified as follows.

- Tree Edges are edges of G that appear in T
- A forward edge is a non-tree edge (u, v) such that $u.d < v.d < v.f < u.f$

Types of Edges

With respect to a DFS forest T, the edges of graph G can be classified as follows.

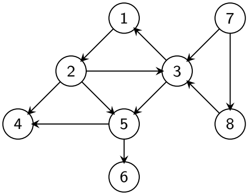
- Tree Edges are edges of G that appear in T
- A forward edge is a non-tree edge (u, v) such that $u.d < v.d < v.f < u.f$
- A back edge is either a self-loop or a non-tree edge (u, v) such that $v.d < u.d < u.f < v.f$

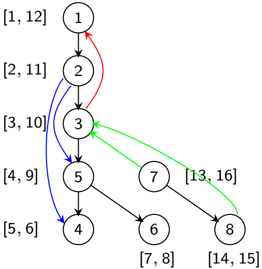
Types of Edges

With respect to a DFS forest T, the edges of graph G can be classified as follows.

- Tree Edges are edges of G that appear in T
- A forward edge is a non-tree edge (u, v) such that $u.d < v.d < v.f < u.f$
- A back edge is either a self-loop or a non-tree edge (u, v) such that $v.d < u.d < u.f < v.f$
- A cross edge is a non-tree edge (u, v) such that the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint.

Example





- Black edges are Tree edges
- Red edges are back edges
- Blue edges are forward edges
- Green edges are cross edges

Directed acyclic graph (dag)

A dag is a directed graph with no cycles

Directed acyclic graph (dag)

A dag is a directed graph with no cycles

Good for modeling processes and structures that have a partial order:

- $a > b$ and $b > c$ implies that $a > c$
- $a > b$ and $b > a$ cannot happen
- But may have a and b such that neither $a > b$ nor $b > c$

Directed acyclic graph (dag)

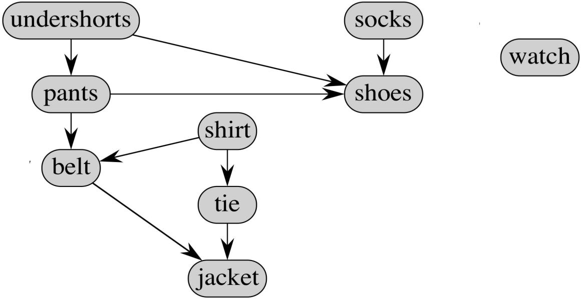
A dag is a directed graph with no cycles

Good for modeling processes and structures that have a partial order:

- $a > b$ and $b > c$ implies that $a > c$
- $a > b$ and $b > a$ cannot happen
- But may have a and b such that neither $a > b$ nor $b > c$

For a dag, it is useful to think of an edge from u to v as saying that $u < v$

Example



How to check if a directed graph is a dag or not?

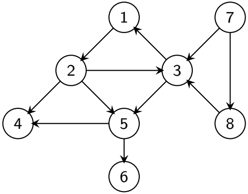


Figure : Graph A

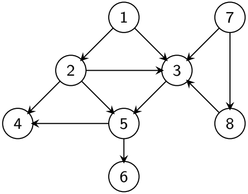


Figure : Graph B

DFS of graph A

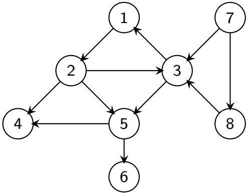
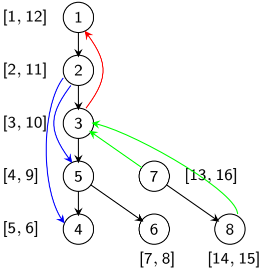


Figure : Graph A



DFS of graph B

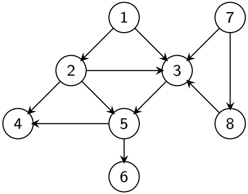
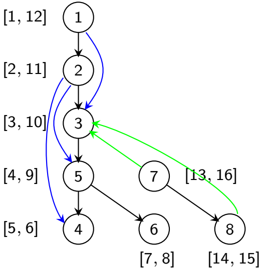


Figure : Graph B



DFS of graph B

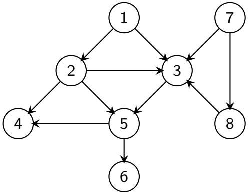
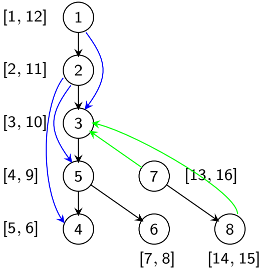


Figure : Graph B



A digraph G is a dag iff the DFS of G yields no back edges

How to check if a directed graph is a dag or not?

A digraph G is a dag iff G the DFS of G yields no back edges

How to check if a directed graph is a dag or not?

A digraph G is a dag iff G the DFS of G yields no back edges

Can be checked in $\Theta(|V| + |E|)$ time

How to check if a directed graph is a dag or not?

A digraph G is a dag iff G the DFS of G yields no back edges

Can be checked in $\Theta(|V| + |E|)$ time

Why does this algorithm work?

- A dag cannot have self-loops

How to check if a directed graph is a dag or not?

A digraph G is a dag iff G the DFS of G yields no back edges

Can be checked in $\Theta(|V| + |E|)$ time

Why does this algorithm work?

- A dag cannot have self-loops
- If there was a back edge (u, v) in the DFS of G then v is an ancestor of u and therefore there is a path from v to u in G. Extending this path by the edge (u, v) yields a cycle in G

How to check if a directed graph is a dag or not?

A digraph G is a dag iff G the DFS of G yields no back edges

Can be checked in $\Theta(|V| + |E|)$ time

Why does this algorithm work?

- A dag cannot have self-loops
- If there was a back edge (u, v) in the DFS of G then v is an ancestor of u and therefore there is a path from v to u in G. Extending this path by the edge (u, v) yields a cycle in G
- Now, if a graph G has a cycle c
  - If v is the first vertex of c discovered in the DFS then every other vertex in c shall become a descendant of v
  - If u is the last vertex of c discovered in the DFS then the edge (v, u) becomes a back edge

Topological sort of a dag

Recall: A dag is a directed graph with no cycles

Good for modeling processes and structures that have a partial order:

- $a > b$ and $b > c$ implies that $a > c$.
- But may have a and b such that neither $a > b$ nor $b > c$

For a dag, it is useful to think of an edge from u to v as saying that $u < v$

Topological sort of a dag

Recall: A dag is a directed graph with no cycles

Good for modeling processes and structures that have a partial order:

- $a > b$ and $b > c$ implies that $a > c$.
- But may have a and b such that neither $a > b$ nor $b > c$

For a dag, it is useful to think of an edge from u to v as saying that $u < v$

Can always make a total order from a partial order

- either $a > b$ or $b > a$ for all a, b

Example: Total order of vertices of a dag

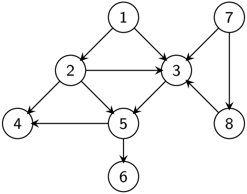


Figure : Graph B

Many possible total orders can be made:

- $7 < 8 < 1 < 2 < 3 < 5 < 4 < 6$
- $7 < 1 < 8 << 2 < 3 < 5 < 4 < 6 < 8$
- $7 < 1 < 2 < 8 < 3 < 5 < 4 < 6$
- $\cdots$

Topological sort of a dag

A total ordering of vertices such that if $(u, v) \in E$ then u appears somewhere before v

Topological sort of a dag

A total ordering of vertices such that if $(u, v) \in E$ then u appears somewhere before v

Algorithm

- Run DFS on G and compute finishing times
- Output vertices in order of decreasing finishing times

A topological sort of graph B

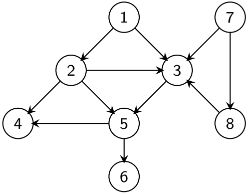
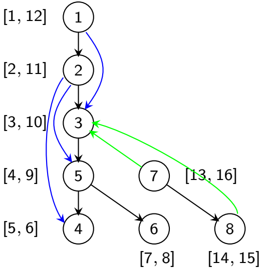


Figure : Graph B



A topological sort of graph B

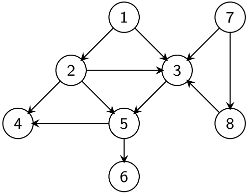
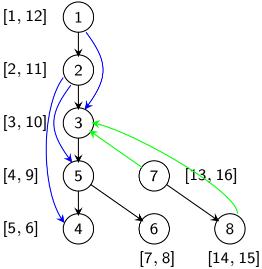


Figure : Graph B



The algorithm outputs:

$$7 < 8 < 1 < 2 < 3 < 5 < 6 < 4$$

Topological sort of a dag

A total ordering of vertices such that if $(u, v) \in E$ then u appears somewhere before v

Algorithm

- Run DFS on G and compute finishing times
- Output vertices in order of decreasing finishing times

Topological sort of a dag

A total ordering of vertices such that if $(u, v) \in E$ then u appears somewhere before v

Algorithm

- Run DFS on G and compute finishing times
- Output vertices in order of decreasing finishing times

Dont need to sort by finishing times

Topological sort of a dag

A total ordering of vertices such that if $(u, v) \in E$ then u appears somewhere before v

Algorithm

- Run DFS on G and compute finishing times
- Output vertices in order of decreasing finishing times

Dont need to sort by finishing times

- When DFS finishes processing a vertex, put the vertex onto the front of a linked list.

Topological sort of a dag

A total ordering of vertices such that if $(u, v) \in E$ then u appears somewhere before v

Algorithm

- Run DFS on G and compute finishing times
- Output vertices in order of decreasing finishing times

Dont need to sort by finishing times

- When DFS finishes processing a vertex, put the vertex onto the front of a linked list.
- The list contains vertices in topologically sorted order

Topological sort of a dag

A total ordering of vertices such that if $(u, v) \in E$ then u appears somewhere before v

Algorithm

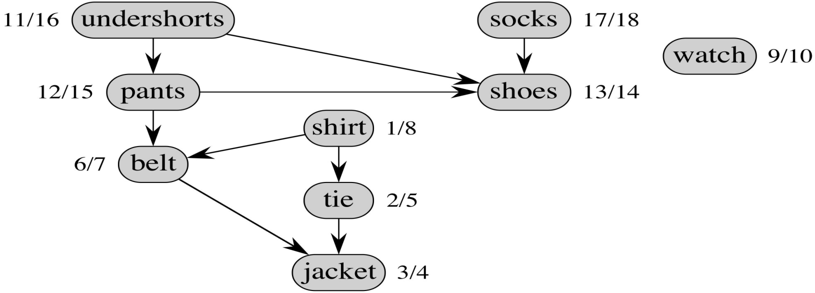
- Run DFS on G and compute finishing times
- Output vertices in order of decreasing finishing times

Dont need to sort by finishing times

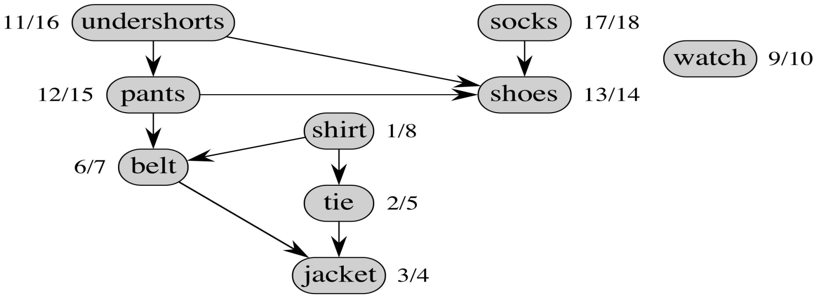
- When DFS finishes processing a vertex, put the vertex onto the front of a linked list.
- The list contains vertices in topologically sorted order

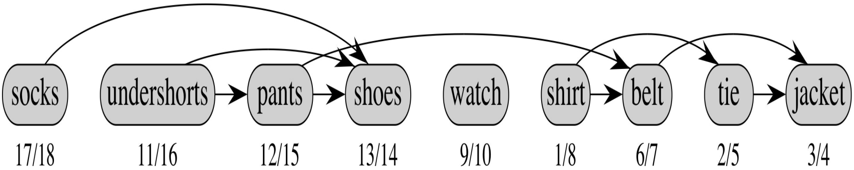
Running time: $\Theta(|V| + |E|)$

Example


11/16
undershorts
socks
17/18
watch
9/10
12/15
pants
shoes
13/14
shirt
1/8
6/7
belt
tie
2/5
jacket
3/4

Example


11/16 undershorts
socks 17/18
watch 9/10
12/15 pants
shoes 13/14
shirt 1/8
6/7 belt
tie 2/5
jacket 3/4


socks
undershorts
pants
shoes
watch
shirt
belt
tie
jacket
17/18
11/16
12/15
13/14
9/10
1/8
6/7
2/5
3/4

Why does this algorithm work?

We want that $(u, v) \in E$ then v finishes before u

Why does this algorithm work?

We want that $(u, v) \in E$ then v finishes before u

A DFS of a graph must eventually explore the edge (u, v)

Why does this algorithm work?

We want that $(u, v) \in E$ then v finishes before u

A DFS of a graph must eventually explore the edge (u, v)

When we explore (u, v) what are the colors of u and v

- u is gray

Why does this algorithm work?

We want that $(u, v) \in E$ then v finishes before u

A DFS of a graph must eventually explore the edge (u, v)

When we explore (u, v) what are the colors of u and v

- u is gray
  - Is v gray? No, because then v would be an ancestor of u in the DFS-forest and we are working with a dag

Why does this algorithm work?

We want that $(u, v) \in E$ then v finishes before u

A DFS of a graph must eventually explore the edge (u, v)

When we explore (u, v) what are the colors of u and v

- u is gray
  - Is v gray? No, because then v would be an ancestor of u in the DFS-forest and we are working with a dag
  - Is v white? Then v becomes child of u in the DFS-forest and $u.d < v.d < v.f < u.f$

Why does this algorithm work?

We want that $(u, v) \in E$ then v finishes before u

A DFS of a graph must eventually explore the edge (u, v)

When we explore (u, v) what are the colors of u and v

- u is gray
  - Is v gray? No, because then v would be an ancestor of u in the DFS-forest and we are working with a dag
  - Is v white? Then v becomes child of u in the DFS-forest and $u.d < v.d < v.f < u.f$
  - Is v black? Then v is already finished (and u is not)