2IL50 Data Structures

2023-24 Q3

Lecture 11: Elementary Graph Algorithms

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Honors track:
Competitive Programming and Problem Solving

Introduction Event: March 21, 13:30 – 17:00, Atlas -1.825

Sign up: send email to k.a.b.verbeek@tue.nl

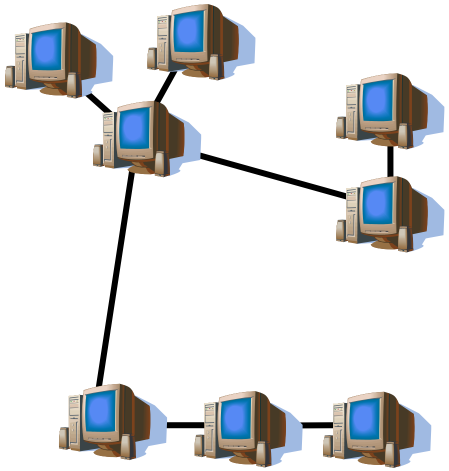
For more info: [Honors Academy TUe]

© Heleen Otten

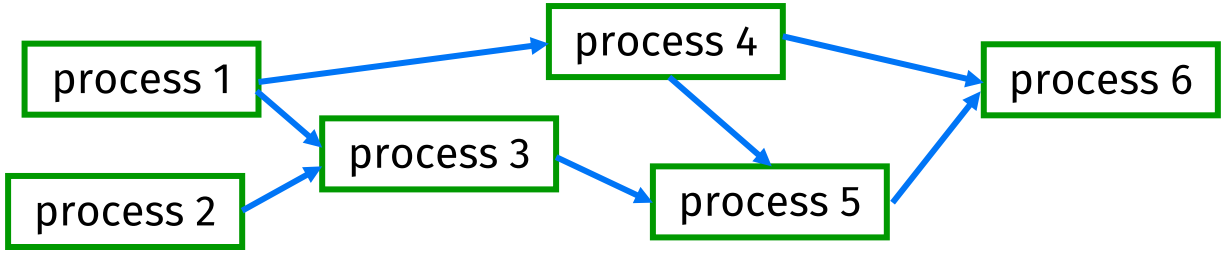
Networks and other graphs

road network

computer network



execution order for processes



Graphs: Basic definitions and terminology

A graph G is a pair $G = (V, E)$

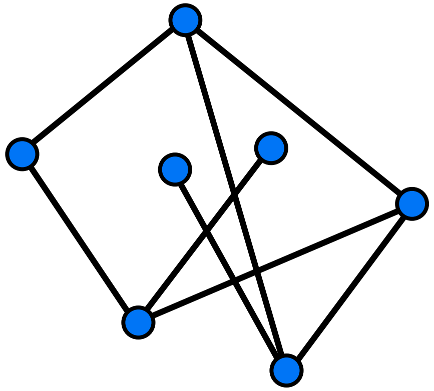
- V is the set of nodes or vertices of G
- $E \subset V \times V$ is the set of edges or arcs of G

If $(u, v) \in E$ then vertex v is adjacent to vertex u

undirected graph

(u, v) is an unordered pair: $(u, v) = (v, u)$

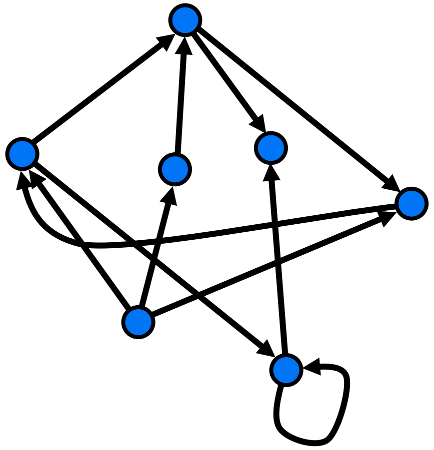
self-loops forbidden



directed graph

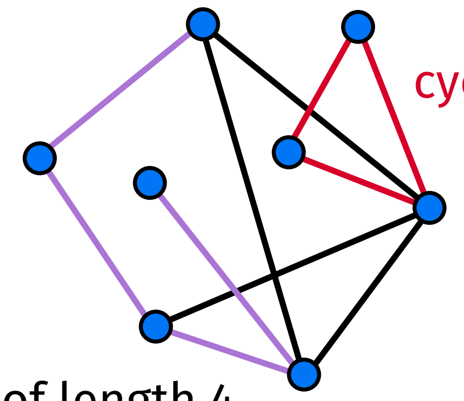
(u, v) is an ordered pair: $(u, v) \neq (v, u)$

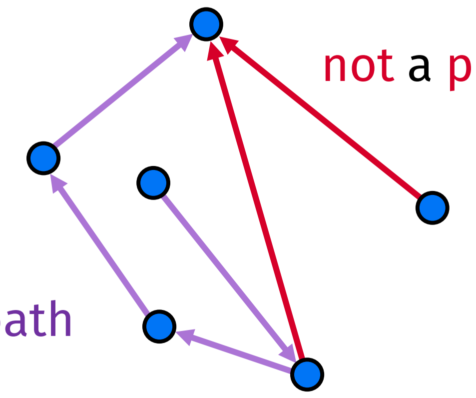
self-loops possible



Graphs: Basic definitions and terminology

Degree of a vertex	number of edges attached to vertex
Path in a graph	sequence $\langle v_0, v_1, \dots, v_k \rangle$ of vertices, such that $(v_{i-1}, v_i) \in E$ for $1 \le i \le k$
Cycle	path with $v_0 = v_k$
Length of a **path**	number of edges in the path
Distance between vertex u and v	length of a **shortest path** between u and v (∞ if v is not **reachable** from u)

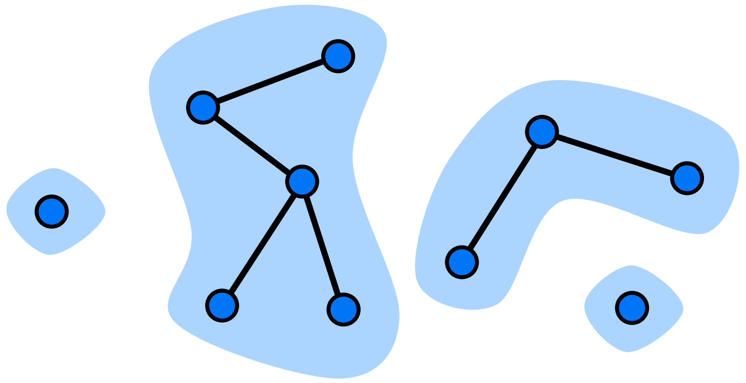




Graphs: Basic definitions and terminology

An undirected graph is connected
if every pair of vertices is connected by a path.

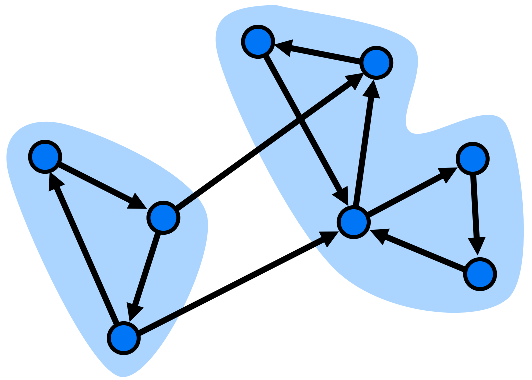
connected components



A directed graph is strongly connected
if every two vertices are reachable from each other.

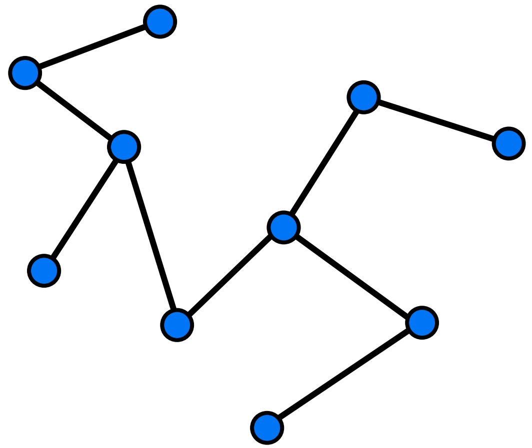
For every pair of vertices u and v we have
a directed path from u to v
and a directed path from v to u.

strongly connected components



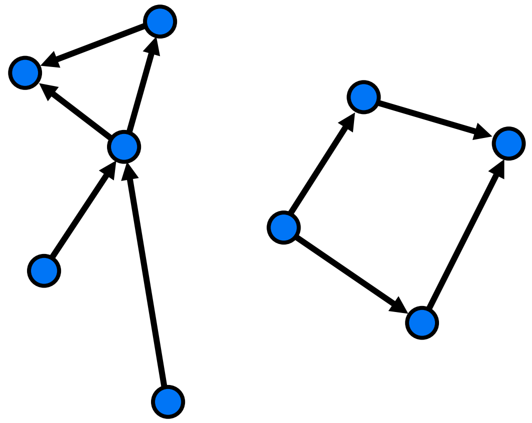
Some special graphs

Tree   connected, undirected, acyclic graph



Every tree with n vertices has exactly $n - 1$ edges

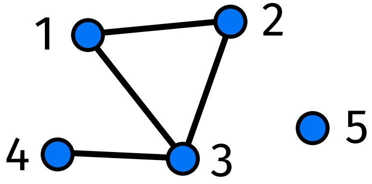
DAG   directed, acyclic graph



Check Appendix B.4 for more basic definitions

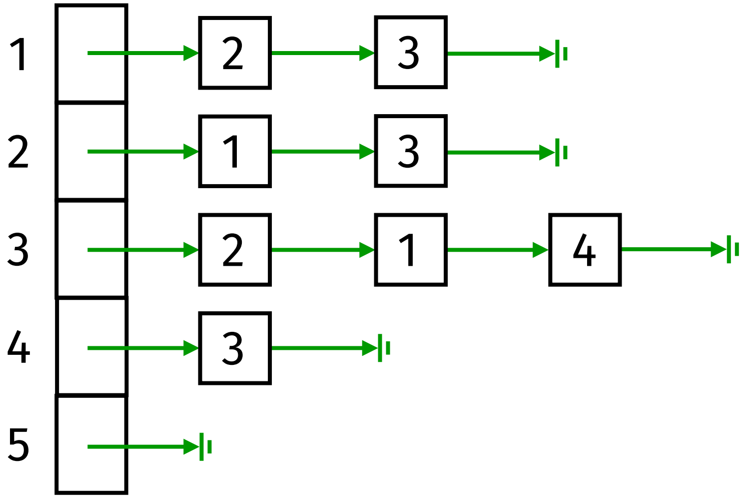
Graph representation

Graph $G = (V, E)$



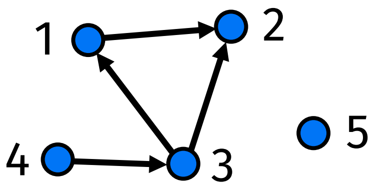
1. Adjacency lists array Adj of $|V|$ lists, one per vertex
   $\mathrm{Adj}[u]$ = linked list of all vertices v with $(u, v) \in E$

   works for both directed and undirected graphs



Graph representation

Graph $G = (V, E)$

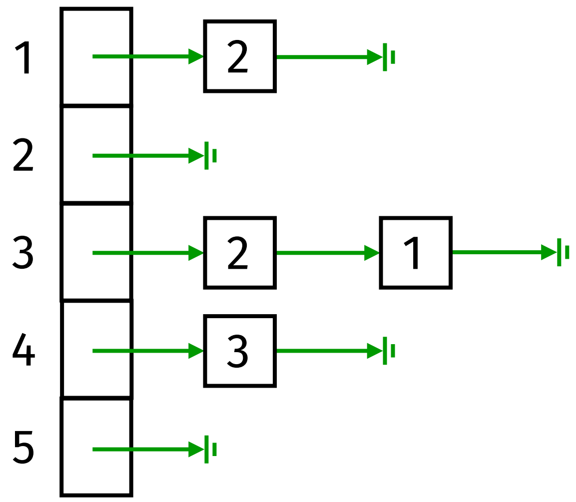


1. Adjacency lists

array Adj of $|V|$ lists, one per vertex

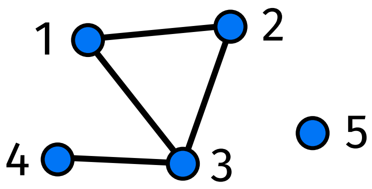
$\mathrm{Adj}[u] =$ linked list of all vertices v with $(u, v) \in E$

works for both directed and undirected graphs



Graph representation

Graph $G = (V, E)$



1. Adjacency lists array Adj of $|V|$ lists, one per vertex
   $\mathrm{Adj}[u]$ = linked list of all vertices v with $(u, v) \in E$

2. Adjacency matrix $|V| \times |V|$ matrix $A = (a_{ij})$

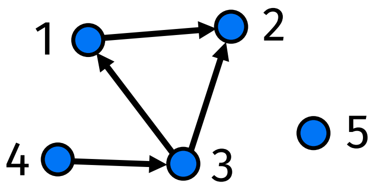
$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

	1	2	3	4	5
1		1	1		
2	1		1		
3	1	1		1	
4			1		
5					

also works for both directed and undirected graphs

Graph representation

Graph $G = (V, E)$



1. Adjacency lists — array Adj of $|V|$ lists, one per vertex
   $\mathrm{Adj}[u]$ = linked list of all vertices v with $(u, v) \in E$

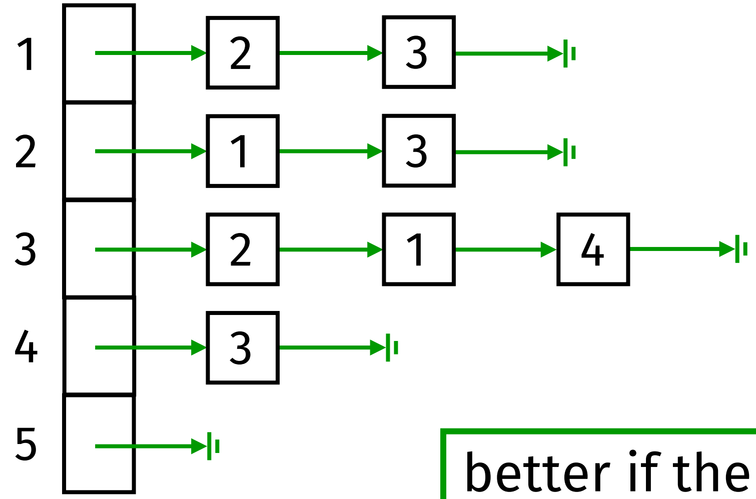
2. Adjacency matrix — $|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$

also works for both directed and undirected graphs

	1	2	3	4	5
1		1			
2					
3	1	1			
4			1		
5					

Adjacency lists vs. adjacency matrix



	1	2	3	4	5
1		1	1		
2	1		1		
3	1	1		1	
4			1		
5					

better if the graph is sparse ... use V for $|V|$ and E for $|E|$

	Adjacency lists	Adjacency matrix
Space	$\Theta(V+E)$	$\Theta(V^2)$
Time to list all vertices adjacent to u	$\Theta(\text{degree}(u))$	$\Theta(V)$
Time to check if $(u,v) \in E$	$\Theta(\text{degree}(u))$	$\Theta(1)$

Searching a graph: BFS and DFS

Basic principle:

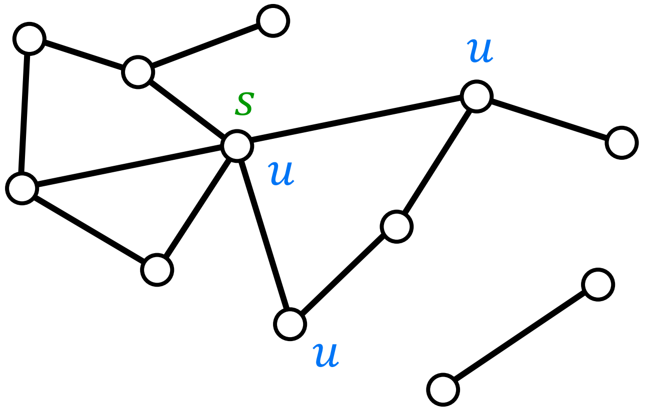
- start at source s
- each vertex has a color: white = not yet visited (initial state)
  gray = visited, but not finished
  black = visited and finished

Searching a graph: BFS and DFS

Basic principle:

- start at source s
- each vertex has a color: white = not yet visited (initial state)
  gray = visited, but not finished
  black = visited and finished

$s.\mathrm{color} =$ gray; $S = \{s\}$
while $S \neq \emptyset$
  remove a vertex u from S
  for each $v \in \mathrm{Adj}[u]$
    if $v.\mathrm{color} ==$ white
      $v.\mathrm{color} =$ gray; $S = S \cup \{v\}$
  $u.\mathrm{color} =$ black

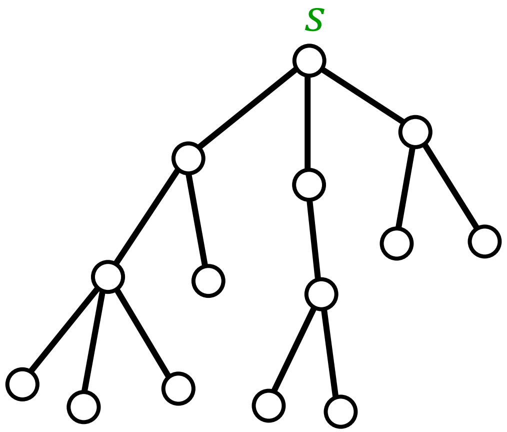


and so on ...

BFS and DFS choose u in different ways;
BFS visits only the connected component that contains s.

BFS and DFS

Breadth-first search

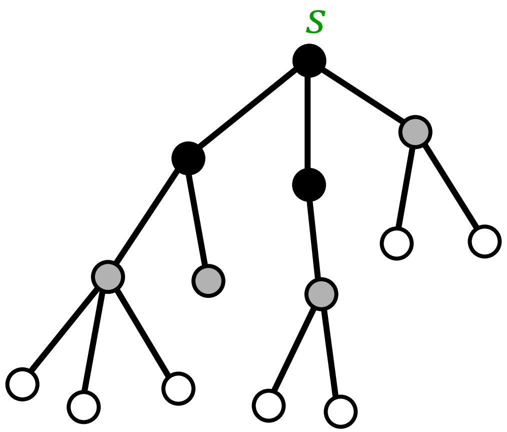


BFS uses a queue

➡ it first visits all vertices at distance 1 from s,
then all vertices at distance 2, ...

BFS and DFS

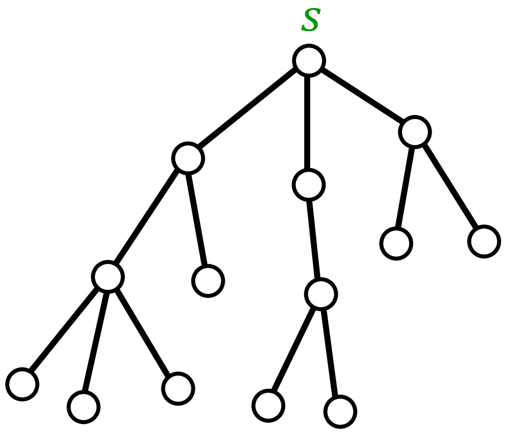
Breadth-first search



BFS uses a queue

➡ it first visits all vertices at distance 1 from s, then all vertices at distance 2, ...

Depth-first search



DFS uses a stack

Breadth-first search (BFS)

BFS(G, s)

for each $u \neq s$
    $u.\text{color} = \text{white};\ u.d = \infty;\ u.\pi = NIL$
$s.\text{color} = \text{gray};\ s.d = 0;\ s.\pi = NIL$
$Q = \emptyset$
Enqueue(Q, s)
while $Q \neq \emptyset$
    $u = \text{Dequeue}(Q)$
    for each $v \in \text{Adj}[u]$
        if $v.\text{color} == \text{white}$
            $v.\text{color} = \text{gray};\ v.d = u.d + 1;\ v.\pi = u$
            Enqueue(Q, v)
    $u.\text{color} = \text{black}$

$u.d$ becomes distance from s to u

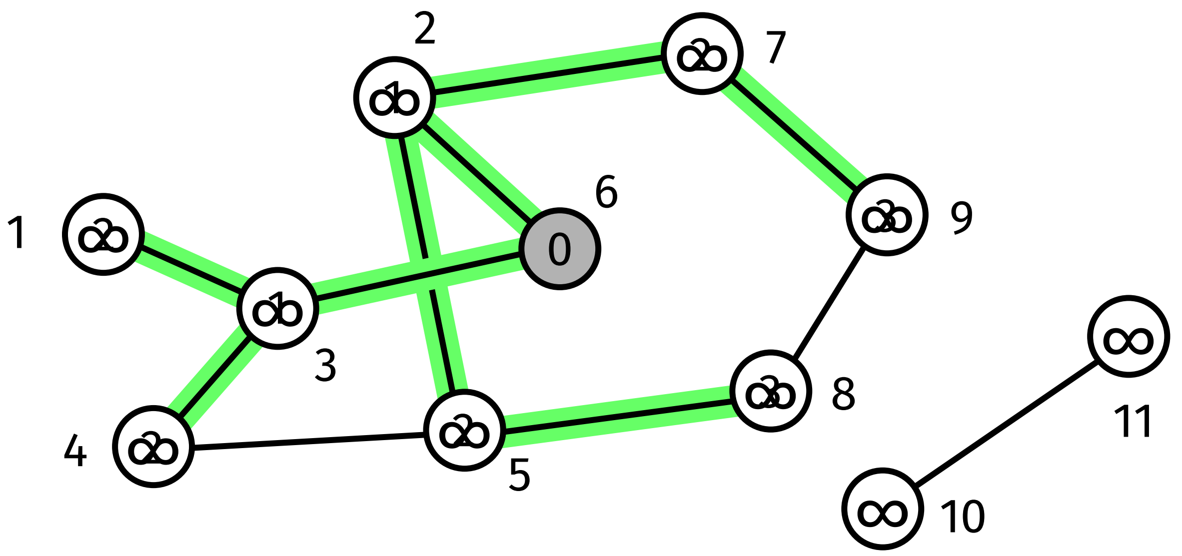
$u.\pi$ becomes predecessor of u

BFS on an undirected graph

Adjacency lists

1	3
2	6, 7, 5
3	1, 6, 4
4	5, 3
5	2, 4, 8
6	3, 2
7	2, 9
8	9, 5
9	7, 8
10	11
11	10

Source $s = 6$



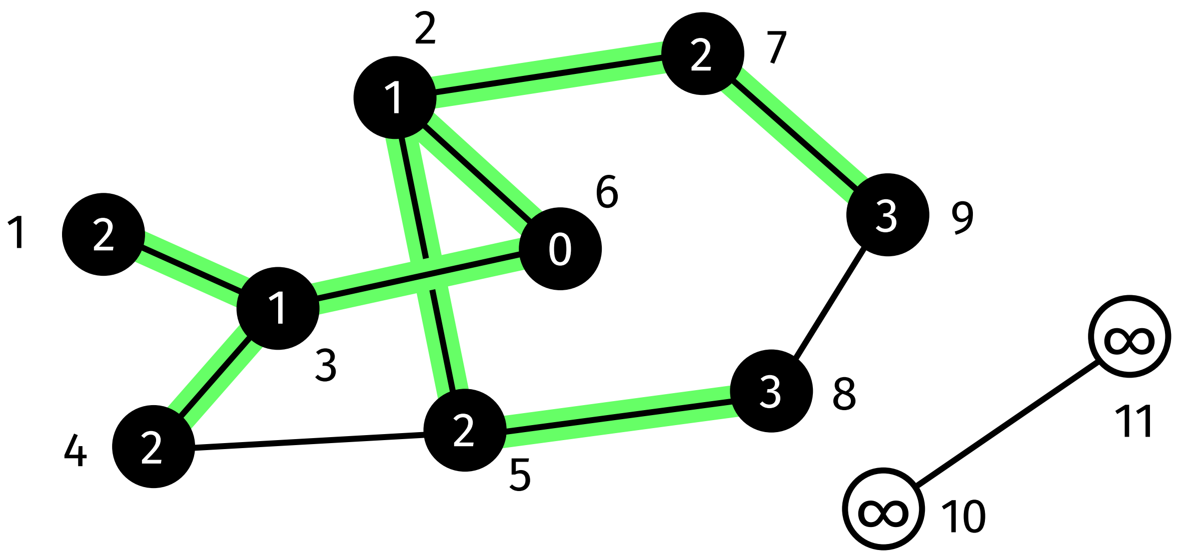
Queue Q ~~6~~ ~~3~~ ~~2~~ ~~1~~ ~~4~~ ~~7~~ ~~5~~ ~~8~~ ~~9~~

BFS on an undirected graph

Adjacency lists

1	3
2	6, 7, 5
3	1, 6, 4
4	5, 3
5	2, 4, 8
6	3, 2
7	2, 9
8	9, 5
9	7, 8
10	11
11	10

Source $s = 6$



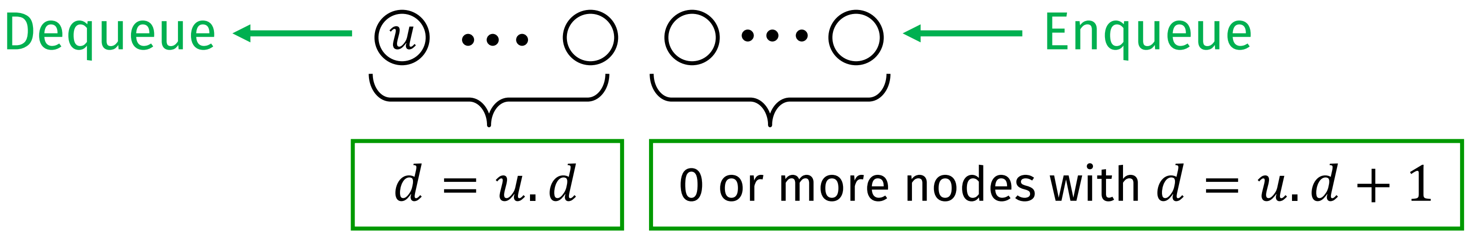
Queue Q ~~6~~ ~~3~~ ~~2~~ ~~1~~ ~~4~~ ~~7~~ ~~5~~ ~~9~~ ~~8~~

Note: BFS only visits the nodes that are reachable from s

BFS: Properties

Invariants

- Q contains only gray vertices
- gray and black vertices never become white again
- the queue has the following form:



- the d fields of all gray vertices are correct
- for all white vertices we have: (distance to s) $> u.d$

BFS: Analysis

Invariants

- Q contains only gray vertices
- gray and black vertices never become white again

This implies

- every vertex is enqueued at most once

  ➡ every vertex is dequeued at most once

- processing a vertex u takes $\Theta(1 + |\mathrm{Adj}[u]|)$ time

  ➡ running time at most $\sum_u \Theta(1 + |\mathrm{Adj}[u]|) = O(V + E)$

BFS: Properties

After BFS has been run from a source s on a graph G we have

- each vertex u that is reachable from s has been visited
- for each vertex u we have $u.d =$ distance to s
- if $u.d < \infty$, then there is a shortest path from s to u that is a shortest path from s to $u.\pi$ followed by the edge $(u.\pi, u)$

Proof: follows from the invariants (*details see book*)

Depth-first search (DFS)

DFS(G)

```
for each u ∈ V
        u.color = white;  u.π = NIL
time = 0
for each u ∈ V
        if u.color == white:  DFS-Visit(u)
```

time = global timestamp for discovering and finishing vertices

DFS-Visit(u)

```
u.color = gray; u.d = time;  time = time + 1
for each v ∈ Adj[u]
        if v.color == white
                v.π = u;  DFS-Visit(v)
u.color = black; u.f = time;  time = time + 1
```

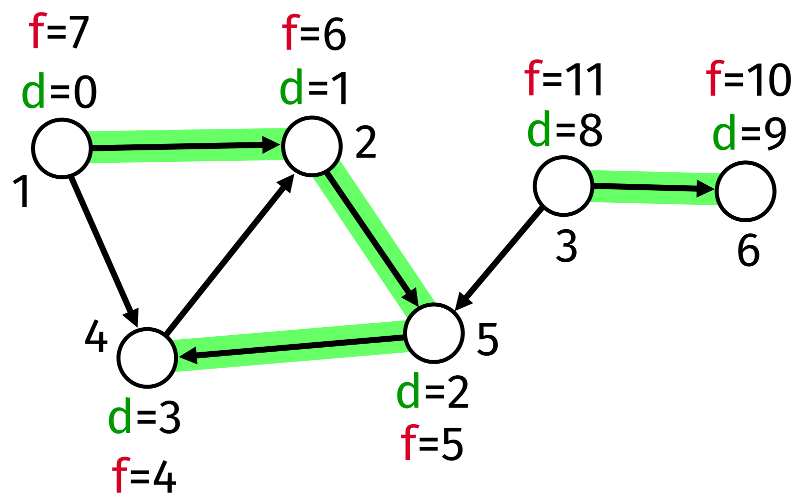
$u.d$ = discovery time

$u.f$ = finishing time

DFS on a directed graph

Adjacency lists

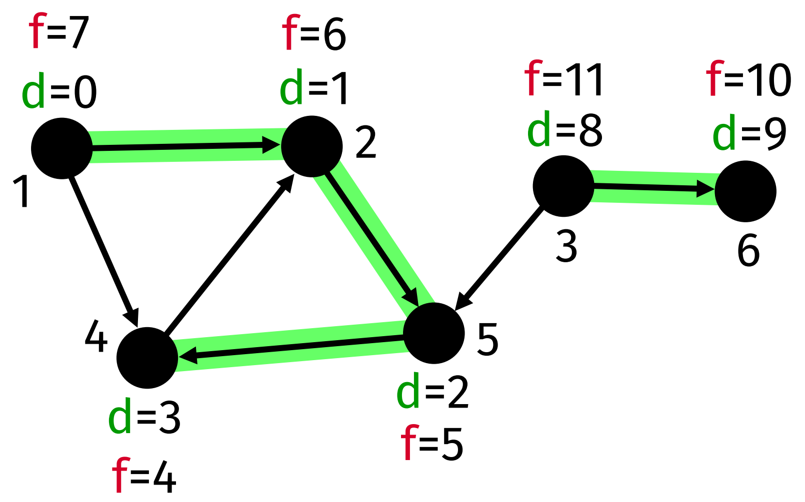
1	2, 4
2	5
3	5, 6
4	2
5	4
6	--



DFS on a directed graph

Adjacency lists

1	2, 4
2	5
3	5, 6
4	2
5	4
6	--



Note: DFS always visits all vertices

DFS: Properties

DFS visits all vertices and edges of G

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

DFS forms a depth-first forest comprised of ≥ 1 depth-first trees.

Each tree is made of edges (u, v) such that u is gray and v is white when (u, v) is explored.

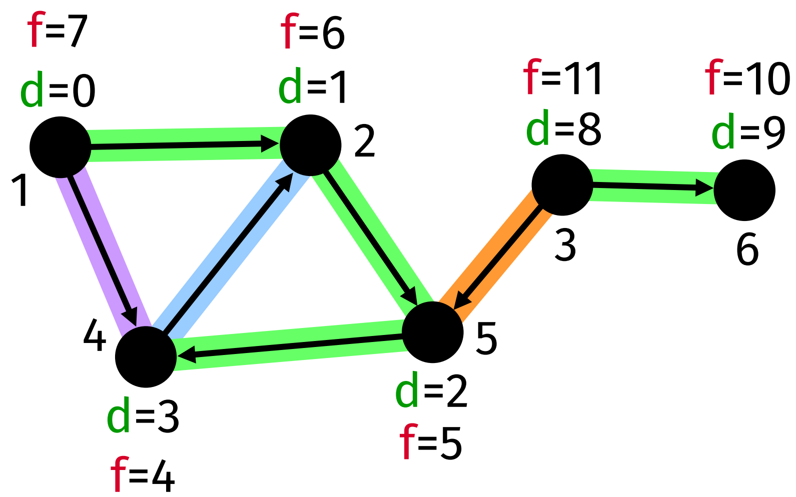
DFS: Edge classification

Tree edges
: edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v); the tree edges form a forest, the DF-forest

Back edges
: edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree

Forward edges
: non-tree edges (u, v) connecting a vertex u to a descendant v

Cross edges
: all other edges



DFS: Edge classification

Tree edges

edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v);
the tree edges form a forest, the DF-forest

Back edges

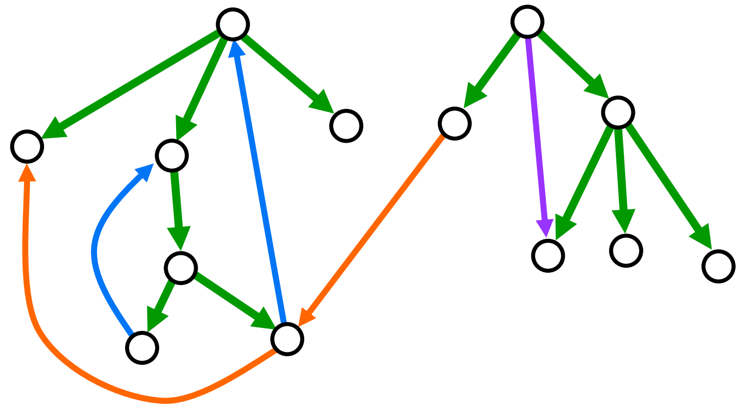
edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree

Forward edges

non-tree edges (u, v) connecting a vertex u to a descendant v

Cross edges

all other edges



Undirected graph

(u, v) and (v, u) are the same edge;
classify by first type that matches.

DFS: Properties

DFS visits all vertices and edges of G

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

DFS forms a depth-first forest comprised of ≥ 1 depth-first trees.

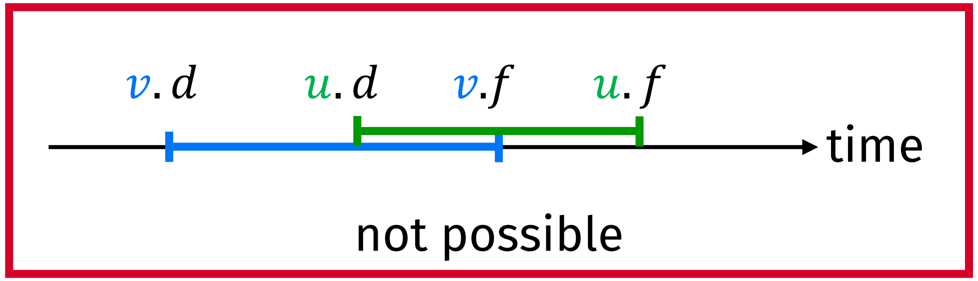
Each tree is made of edges (u, v) such that u is gray and v is white when (u, v) is explored.

DFS of an undirected graph yields only tree and back edges.
No forward or cross edges.

Discovery and finishing times have parenthesis structure.

[ ] { }   [{ }]   { [ ] }   { [}]   [{] }

Discovery and finishing times

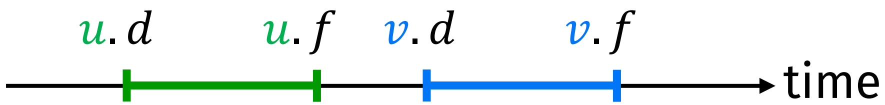


Theorem

In any depth-first search of a (directed or undirected) graph $G = (V, E)$,
for any two vertices u and v, exactly one of the following three conditions holds:

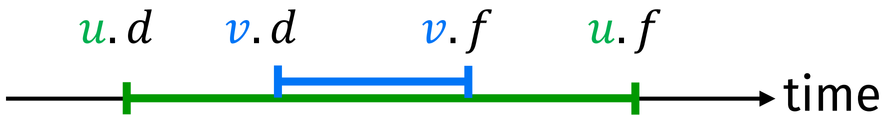
1. the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint

   neither of u or v is a descendant of the other



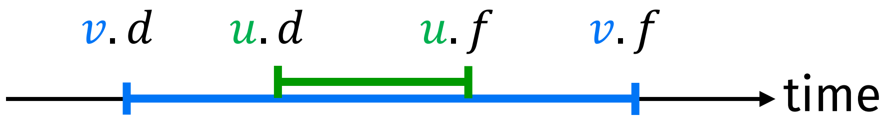
2. the interval $[u.d, u.f]$ entirely contains the interval $[v.d, v.f]$

   v is a descendant of u



3. the interval $[v.d, v.f]$ entirely contains the interval $[u.d, u.f]$

   u is a descendant of v



DFS: Discovery and finishing times

Proof (sketch)

- assume $u.d < v.d$

case 1: v is discovered in a recursive call from u

➡ v becomes a descendant of u

recursive calls are finished before u itself is finished

➡ $v.f < u.f$

case 2: v is not discovered in a recursive call from u

➡ v is not reachable from u and not one of u's descendants

➡ v is discovered only after u is finished

➡ $u.f < v.d$

➡ u cannot become a descendant of v since it is already discovered

∎

DFS: Discovery and finishing times

Corollary v is a proper descendant of u if and only if $u.d < v.d < v.f < u.f$.

Theorem (White-path theorem)

v is a descendant of u if and only if at time $u.d$, there is a path $u \rightsquigarrow v$ consisting of only white vertices.

(Except for u which was *just* colored gray.)

(*See the book for details and proof.*)

Topological sort

Using depth-first search …

Topological sort

Input directed, acyclic graph (DAG) $G = (V, E)$

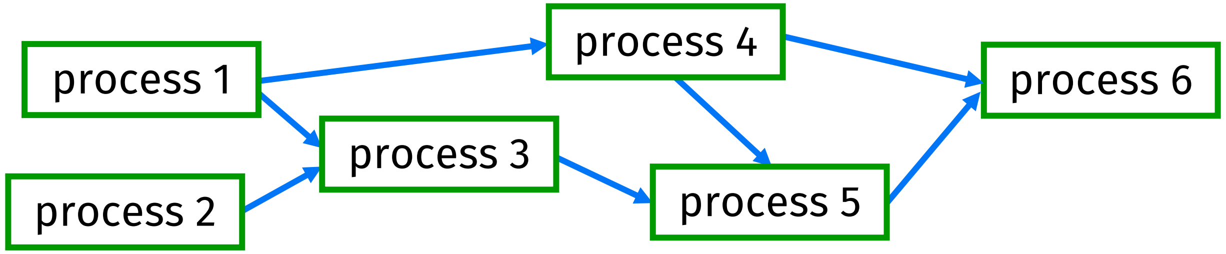
Output a linear ordering $v_1, v_2, \dots, v_n$ of the vertices such that if $(v_i, v_j) \in E$ then $i < j$

DAGs are useful for modeling processes and structures that have a partial order

Partial order

- $a > b$ and $b > c$ ➡ $a > c$
- but may have a and b such that neither $a > b$ nor $b > a$

execution order for processes



Topological sort

Input directed, acyclic graph (DAG) $G = (V, E)$

Output a linear ordering $v_1, v_2, \dots, v_n$ of the vertices such that if $(v_i, v_j) \in E$ then $i < j$

DAGs are useful for modeling processes and structures that have a partial order

Partial order

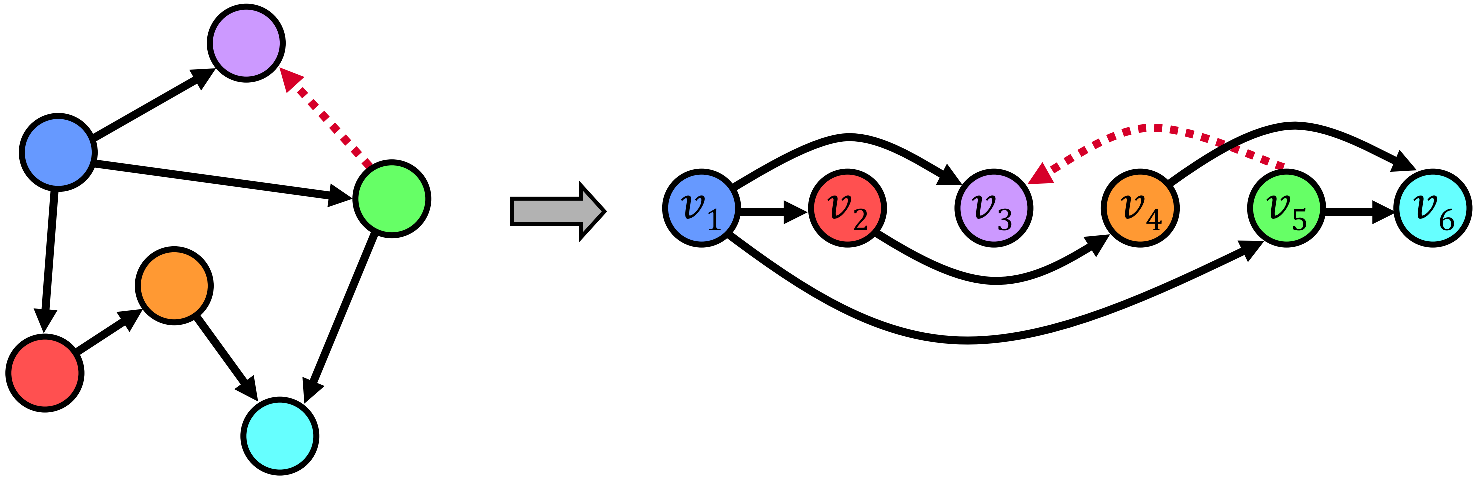
- $a > b$ and $b > c$ ➡ $a > c$
- but may have a and b such that neither $a > b$ nor $b > a$
- a partial order can always be turned into a total order
(either $a > b$ or $b > a$ for all $a \neq b$)

  that's what a topological sort does ...

Topological sort

Input directed, acyclic graph (DAG) $G = (V, E)$

Output a linear ordering $v_1, v_2, \dots, v_n$ of the vertices such that if $(v_i, v_j) \in E$ then $i < j$



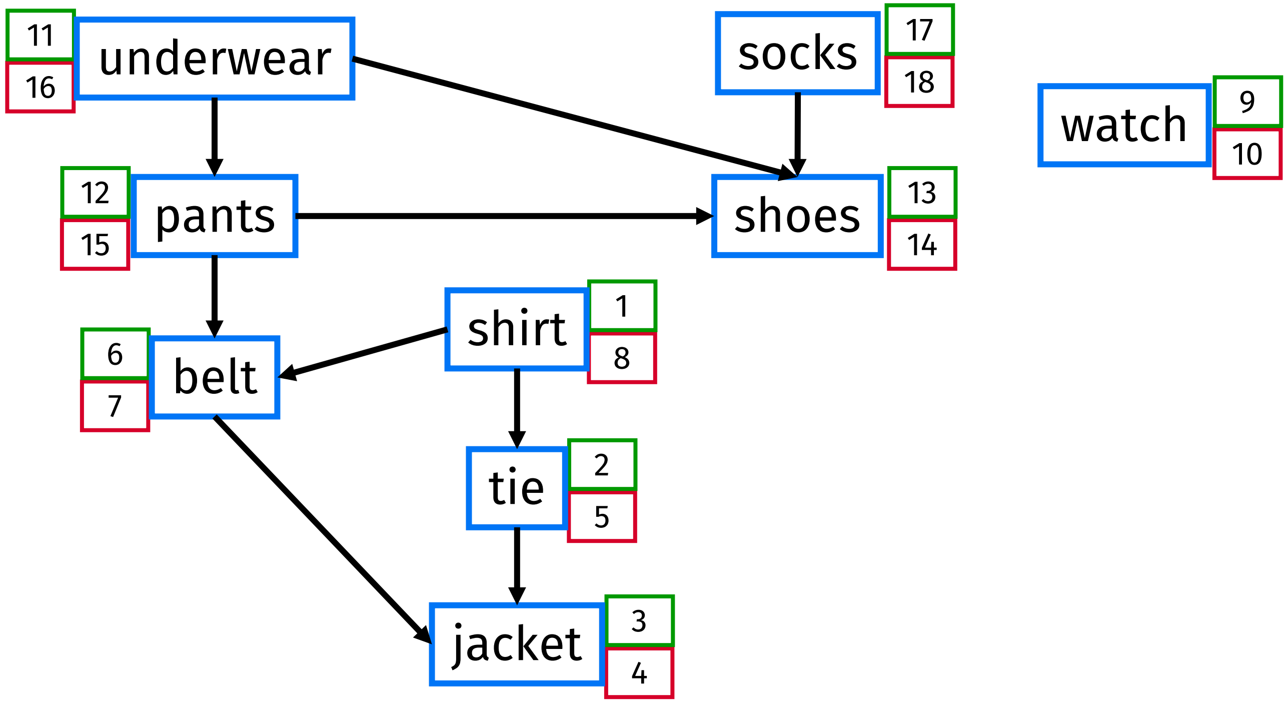
Every directed, acyclic graph has a topological order

Lemma A directed graph G is acyclic if and only if DFS of G yields no back edges.

Topological sort

TopologicalSort(V, E)

call DFS(V, E) to compute finishing time $v.f$ for all $v \in E$
output vertices in order of decreasing finishing time

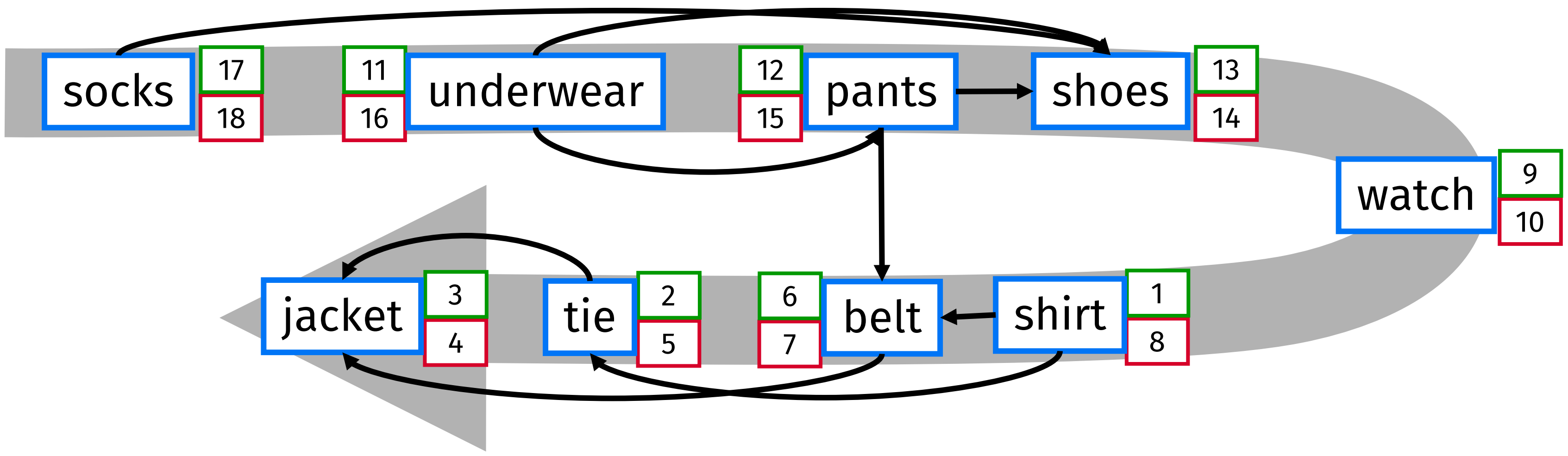


Topological sort

TopologicalSort(V, E)

1 call DFS(V, E) to compute finishing time $v.f$ for all $v \in E$

2 output vertices in order of decreasing finishing time



Topological sort

TopologicalSort(V, E)

call DFS(V, E) to compute finishing time $v.f$ for all $v \in E$
output vertices in order of decreasing finishing time

Lemma

TopologicalSort(V, E) produces a topological sort of a directed acyclic graph $G = (V, E)$.

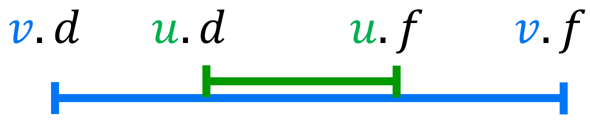
Topological sort

Lemma

TopologicalSort(V, E) produces a topological sort of a directed acyclic graph $G = (V, E)$.

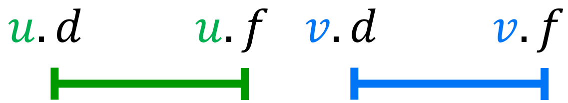
Proof Let $(u, v) \in E$ be an arbitrary edge.
To show: $u.f > v.f$
Consider the intervals $[d, f]$ and assume $u.f < v.f$

case 1:



➡ u is a descendant of v
G has a cycle

case 2:



When DFS-Visit(u) is called, v has not been discovered yet.
DFS-Visit(u) examines all outgoing edges from u, also (u, v).
➡ v is discovered before u is finished.

Topological sort

Lemma

TopologicalSort(V, E) produces a topological sort of a directed acyclic graph $G = (V, E)$.

Running time?

- we do not need to sort by finishing times
- just output vertices as they are finished

➡ $\Theta(V + E)$ for DFS and $\Theta(V)$ for output

➡ $\Theta(V + E)$

Honors track:
Competitive Programming and Problem Solving

Introduction Event: March 21, 13:30 – 17:00, Atlas -1.825

Sign up: send email to k.a.b.verbeek@tue.nl

For more info: Honors Academy TUe