

# GRAPH THEORY

GI

- GRAPH:** ordered pair  $G = (V, E)$  of
- o set of **VERTICES** (or **NODES**)  $V$
  - o set of **EDGES**  $E$ , where an edge is a pair of vertices  $u, v \in V$

Sometimes write  $V(G)$  and  $E(G)$  for the set of vertices and edges of a graph.

Most often, we will assume that  $V$  and  $E$  are the vertex and edge sets of whatever graph we are discussing right now

## UNDIRECTED GRAPH

edges are unordered pairs  $\{u, v\}$   
edge has no particular direction

## DIRECTED GRAPH

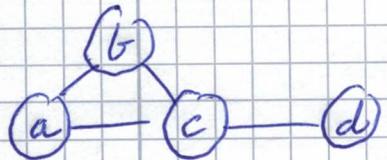
edges are ordered pairs  $(u, v)$   
edge  $u \rightarrow v$  directed from  $u$  to  $v$

But even for undirected graphs, most often we are lazy and write edges as  $(u, v)$  — making clear from context whether graphs are directed or undirected

An undirected graph can be viewed as a directed graph with two directed edges  $(u, v)$  and  $(v, u)$  for every edge  $\{u, v\}$

## EXAMPLE UNDIRECTED GRAPH

| G II



$$V = \{a, b, c, d\}$$

$$E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{c, d\}\}$$

We will tend to assume that graphs are undirected unless specified otherwise

We also assume unless stated otherwise that graphs are SIMPLE, meaning:

- no SELF-LOOPS ( $v, v$ )



- no multiple copies of some edge ( $u, v$ ) (since  $E$  is a set, and sets don't have copies)



Non-simple graphs are sometimes called MULTIGRAPHS

## UNDIRECTED GRAPH NOTATION & TERMINOLOGY

NEIGHBOURS of  $v$        $N(v) = \{w \in V \mid \{v, w\} \in E\}$

$$N(a) = \{b, c\}$$

DEGREE of  $v$        $\deg(v) = |N(v)|$

$$\deg(a) = 2$$

# edges in undirected graph

$$|E| = \frac{1}{2} \sum_{v \in V} \deg(v)$$

## DIRECTED GRAPHS

OUT-NEIGHBOURS

$$N_{out}(v) = \{w \in V \mid (v, w) \in E\}$$

OUT-DEGREE

$$\deg_{out}(v) = |N_{out}(v)|$$

IN-NEIGHBOURS

$$N_{in}(v) = \{u \in V \mid (u, v) \in E\}$$

IN-DEGREE

$$\deg_{in}(v) = |N_{in}(v)|$$

A **PATH**  $v_0 \rightsquigarrow v_n$  (in an undirected or directed graph)

is a sequence  $v_0, v_1, \dots, v_n$  such that  
for all  $i \in [n]$   $(v_{i-1}, v_i) \in E$

Length = # edges (=  $n$  above)

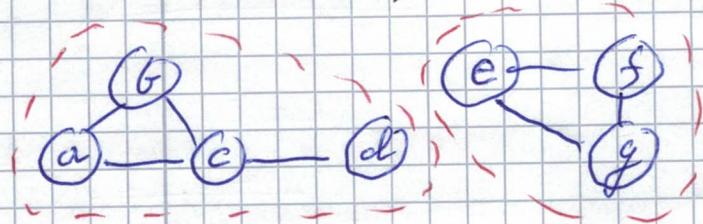
A **SIMPLE PATH** is a path where no vertex repeats

Alternative terminology:

**PATH** = **WALK**

**SIMPLE PATH** = **PATH**

An undirected graph is **CONNECTED** if  
for all  $u, v \in V$  there are paths  $u \rightsquigarrow v$   
and  $v \rightsquigarrow u$ , otherwise **DISCONNECTED**



Disconnected graph with  
2 **CONNECTED COMPONENTS**

For directed graph, more detailed definitions  
of connectedness are used

A **CYCLE** is a walk  $v_0 \rightsquigarrow v_n$  with  $v_0 = v_n$   
that is not allowed to repeat edges

A graph is **ACYCLIC** if it has no cycles

### EXAMPLE

$a \rightarrow c \rightarrow d$  is a (simple) path

$a \rightarrow b \rightarrow c \rightarrow a$  is a cycle

DIRECTED TREES: directed graph  $T = (V, E)$

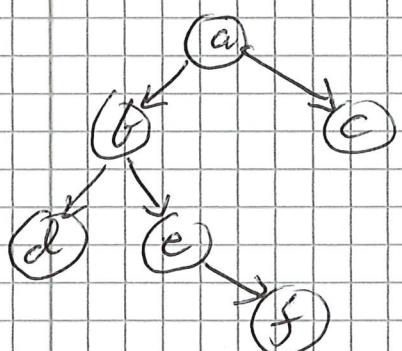
with

- o special vertex  $v_0$  (ROOT)
- o for any  $v \in V \setminus \{v_0\}$  there is a unique walk from  $v_0$  to  $v$
- o there is no walk from  $v_0$  to  $v_0$

Can write  $(T, v_0)$  for clarity to specify clearly which vertex is the root

Ex  $V = \{a, b, c, d, e, f\}$

$$E = \{(a, b), (a, c), (b, d), (b, e), (e, f)\}$$



Can visualize as tree  
(turned upside down,  
so that root is at  
the top)

THEOREM (Properties of directed trees)

Let  $(T, v_0)$  be a directed tree. Then

- 1)  $T$  contains no cycles
- 2)  $v_0$  is the only root
- 3) Indegree of  $v_0$  is 0  
Indegree of all other vertices is 1

Proof By contradiction

1) Suppose  $\exists$  cycle  $C = u_1 \rightarrow u_2 \rightarrow \dots \rightarrow u_k \rightarrow u$ ,

By definition of directed tree,  $\exists$  walk

$W: v_0 \rightarrow \dots \rightarrow u_1$

(in fact, has to be path! Why?)

But then  $W \& C$  concatenated yields  
second path from  $v_0$  to  $u_1$ .  $\leftarrow$

2) Suppose that there exists second  
root  $u$

Since  $u$  root,  $\exists$  path  $P_1: u \rightarrow v_0$

Since  $v_0$  root,  $\exists$  path  $P_2: v_0 \rightarrow u$

But then concatenation of  $P_1$  &  $P_2$   
yields cycle, contradicting 1)  $\leftarrow$

3) Suppose indegree of  $v_0$  is  $\geq 1$  and  
that  $\exists$  edge  $(u, v_0)$

Add edge  $(u, v_0)$  to walk  $v_0$  more  
to get cycle  $\leftarrow$

For  $v \in V \setminus \{v_0\}$  suppose

$\exists$  edges  $(u_1, v)$  and  $(u_2, v)$

for  $u_1 \neq u_2$

By definition  $\exists$  walks  $P_1: v_0 \rightarrow u_1$ ,

$P_2: v_0 \rightarrow u_2$

Concatenate with edges to get two different paths to  $v$   $\leftarrow$

## THEOREM

If  $T = (V, E)$  is a directed tree on  $n = |V|$  vertices, then

$$|E| = n - 1$$

Proof By definition,

$$\# \text{ edges} = \sum_{v \in V} \text{indeg}(v)$$

Use previous theorem:

$$\text{indeg}(v_0) = 0$$

$$\text{indeg}(v) = 1 \quad \text{for } v \in V \setminus \{v_0\}$$

## TREE TERMINOLOGY

LEAVES vertices with no outgoing edges

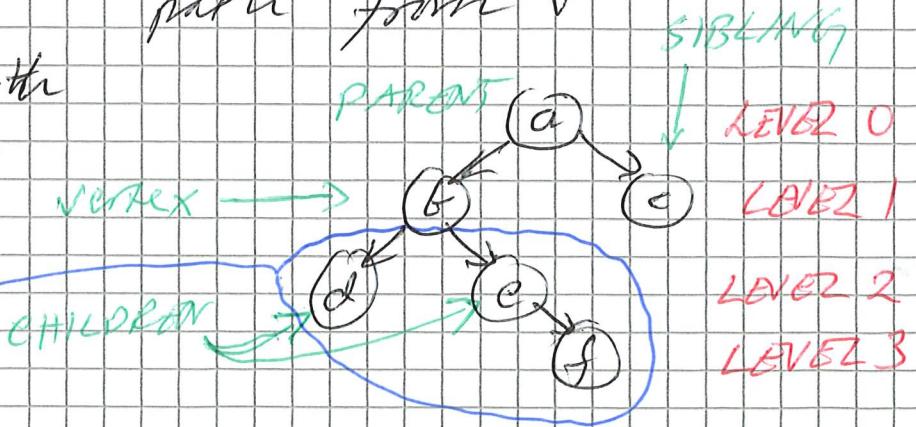
CHILDREN of  $v$ : the out-neighbours of  $v$

PARENT of  $v$ : the in-neighbours of  $v$

SIBLINGS of  $v$ : other children of  
parent of  $v$

DESCENDANTS of  $v$ : all vertices reachable by  
path from  $v$

LEVEL of  $v$ : length  
of path from root  
to  $v$



An undirected **TREE** is a connected graph with no cycles

**THEOREM** A finite graph  $G = (V, E)$

(i.e.,  $|V| < \infty$ ) is a tree if any of the following hold

- ①  $G$  connected and acyclic
- ②  $G$  connected and  $|E| = |V| - 1$
- ③  $G$  acyclic and  $|E| = |V| - 1$
- ④ There exists a unique simple path between any pair of vertices in  $G$

**ROOTED TREE**  $(T, v_0)$

- undirected tree  $T$
- special root vertex  $v_0$

**DIRECTED TREE**: "all edges point away from root"

$G' = (V', E')$  is a **SUBGRAPH** of  $G = (V, E)$   
if  $V' \subseteq V$  and  $E' \subseteq E$

EXAMPLE

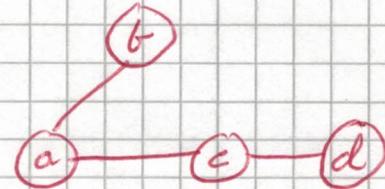
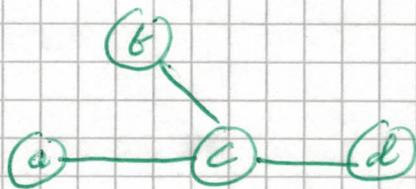
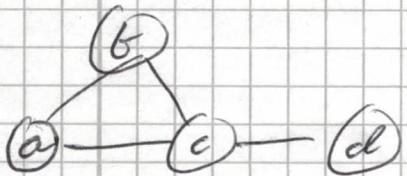


( $G'$  can be obtained from  $G$  by deleting vertices and edges)

A **SPANNING TREE** of a connected graph  $G = (V, E)$  is a tree  $T = (V, E')$  that is a subgraph of  $V$  (but connects all of  $G$ )

A graph can have many spanning trees

G, IV



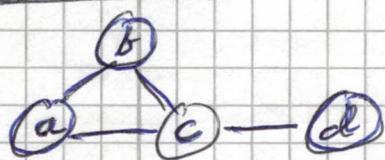
Two graphs  $G_1 = (V_1, E_1)$  and  $G_2 = (V_2, E_2)$  are **ISOMORPHIC** if they are "the same graph"

Formally, there exists a bijective function  
 $f: V_1 \rightarrow V_2$  such that

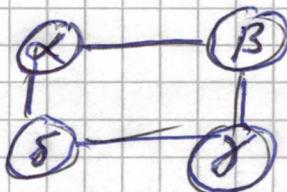
$(v, w) \in E_1$  if and only if  $(f(v), f(w)) \in E_2$

If so, write  $G_1 \cong G_2$

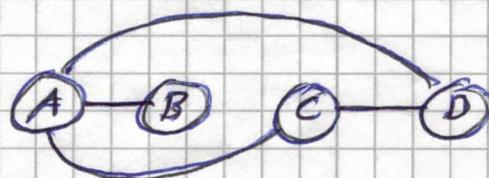
### EXAMPLES



$G_1$



$G_2$



$G_3$

$G_1 \not\cong G_2$

$G_1 \cong G_3$

Graph isomorphism is an equivalence relation —  
"identifies all different graphs"

(Doesn't matter whether vertices are called

- $a, b, c, \dots$
- $\alpha, \beta, \gamma, \dots$
- $A, B, C, \dots$

Want to understand if  
graphs are STRUCTURALLY  
the same)

# GRAPH REPRESENTATION

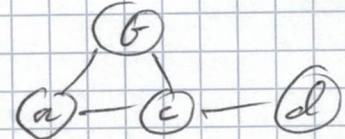
6. V

## ADJACENCY MATRIX

Rows and columns indexed by  $V$ ; 1 is cell  $(u,v)$  if

$(u,v) \in E$ , 0 o/w

	a	b	c	d
a	0	1	1	0
b	1	0	1	0
c	1	1	0	1
d	0	0	1	0



## ADJACENCY LISTS

lists of neighbours for every vertex

(say out-neighbours for directed graph, for concreteness)

$$N(a) = b, c$$

$$N(b) = a, c$$

$$N(c) = a, b, d$$

$$N(d) = c$$

**E T O V R** Given two graphs  $G$  and  $H$  (in adjacency matrix representation, say), how efficiently can we decide whether  $G \cong H$ ?

For  $G = (V, E)$   $H = (V', E')$  we can do this in time  $O(|V|! \cdot |V|^2)$

This is worse than exponential ...

[Babai 2016] Quasipolynomial time algorithm  
 $O(|V|^{(\log(|V|))^k})$  almost polynomial.

Believed to be doable in polynomial time.  
In practice, possible to solve very fast.

## Graph operations to support

- $\text{Adjacent}(u, v)$  - is there an edge  $(uv)$ ?
- $\text{Neighbours}(u)$  - list of (out-) neighbours of  $u$

	Adjacency matrix	Adjacency list
$\text{Adjacent}(u, v)$	$O(1)$	$O(\deg(u))$
$\text{Neighbours}(u)$	$O( V )$	$O(\deg(u))$
Storage requirements:	$O( V ^2)$	$O( V  +  E )$

For sparse graphs ( $|E| \ll |V|^2$ ) preferable to use adjacency lists.

## GRAPH TRAVERSAL

Explore a graph and its structure

Perform operations on vertices

Build spanning tree.

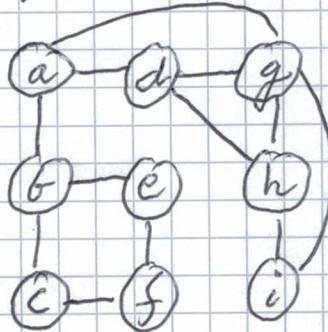
Make sure to visit all vertices exactly once

Simplifying assumption for this lecture:

Graph connected (otherwise deal with undirected and connected components separately)

Two classic algorithms

- depth-first search (DFS)
- breadth-first search (BFS)

Example graph

Neighbour lists

a : b, d, g

b : a, c, e

c : b, f

d : a, g, h

e : b, f

f : c, e

g : a, d, h, i

h : d, g, i

i : g, h

For every vertex  $v$ , maintain $v \cdot \text{marked}$  - has  $v$  been visited $v \cdot d$  - discovery time / distance $v \cdot f$  - finish time (for DFS) $v \cdot \pi$  - edge leading to  $v$  in spanning tree (if viewed as directed)DEPTH-FIRST SEARCH

Recursive algorithm

Mark  $v$  as discoveredVisit all non-visited  $\cancel{\text{out}}^{\text{in}}$  neighbours of  $v$ Add such edges  $(u, v)$  to spanning treeMark  $v$  as finished

DFS ( $s$ )

for all  $v \in V$

$v.\text{marked} := \text{FALSE}$

time := 0

DFS-visit ( $s$ )

DFS-visit ( $v$ )

$v.d := \text{time}$

time := time + 1

$v.\text{marked} := \text{TRUE}$

for all  $u \in N(v)$  deg( $v$ ) times

if NOT( $u.\text{marked}$ )  
 | DFS-visit ( $u$ ).  
 |  $u.\text{ft} := v$

$v.f := \text{time}$   
 time := time + 1

Example: DFS( $a$ )

visit( $a$ )

visit ( $b$ ):  $a$  marked

visit ( $c$ ):  $b$  marked

visit ( $f$ ):  $c$  marked

visit ( $e$ ): everything marked

now  $c$  marked

visit ( $d$ )

a marked

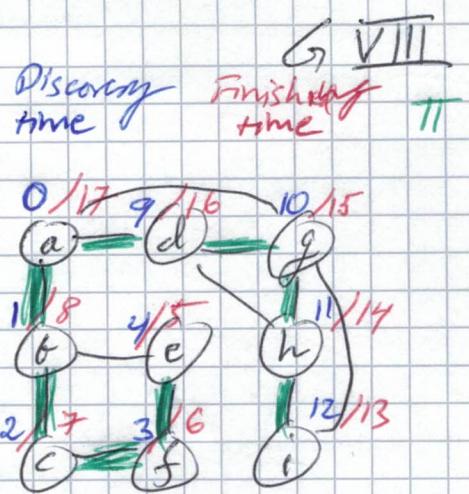
visit ( $g$ ):  $a, d$ , marked

visit ( $h$ ):  $d, g$  marked

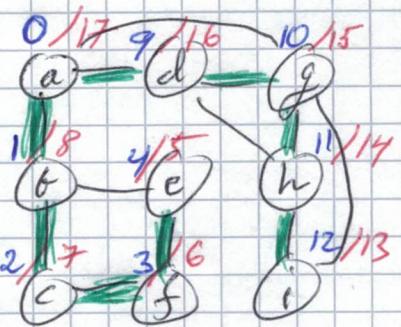
visit ( $i$ ): everything marked

now  $i$  marked

now  $h$  marked



Discovery time Finishing time TT



(out-neighbors for directed graph)

SMALL NOTE: Time above starts from 0

In CLRS, start time from 1

Time complexity (see also previous page)

GIX

Initialization  $O(|V|)$

DFS - visit called  $|V|$  times

Work  $O(|I|) + O(\deg(v))$

TOTAL TIME COMPLEXITY

$$O(|V|) + O(|V|) + \sum_{v \in V} O(\deg(v))$$
$$= \boxed{O(|V| + |E|)}$$

This is assuming adjacency list representation  
(Where is this used?)

# GRAPH TRAVERSAL RECAP

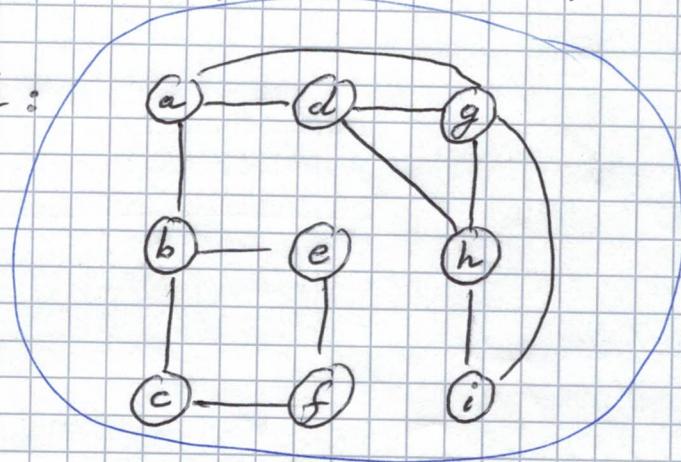
GT I

- Explore structure of graph
- Build spanning tree

Simplifying assumptions during the lecture

- 1) Graph connected (but easy to deal with disconnected graphs)
- 2) Graph undirected (but algorithms work equally well for directed graphs)

Our example graph:

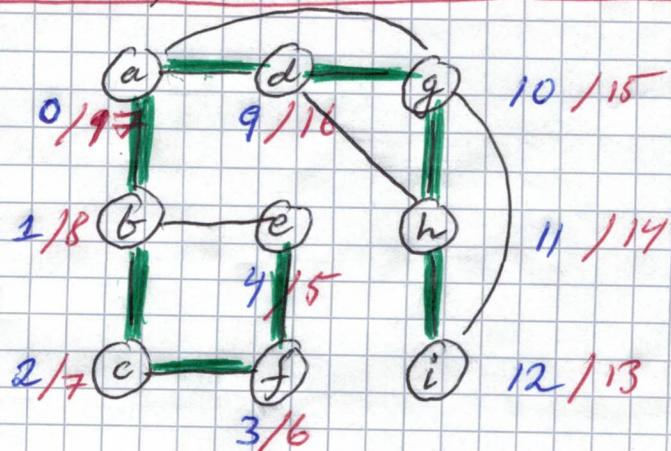


## DEPTH-FIRST SEARCH

Recursive algorithm

- (1) Mark vertex  $v$  as "discovered"
- (2) For all "non-discovered" neighbours  $w$  of  $v$ 
  - add edge  $(v,w)$  to spanning tree
  - make recursive call starting from (1)
- (3) Mark vertex  $v$  as "finished"

DISCOVERY TIME  
FINISHING TIME



## BREADTH-FIRST SEARCH

GT II

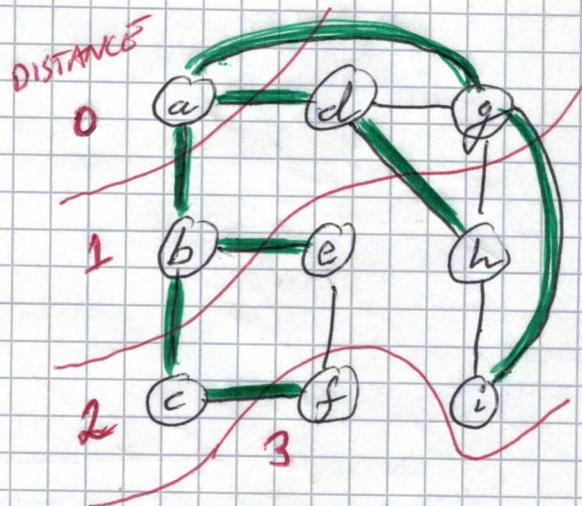
Don't go deep — instead explore vertices in order of increasing distance

Insert first vertex in queue and mark as "discovered".  
Repeat

dequeue first vertex  $v$   
for all "non-discovered" neighbours  $w$  of  $v$   
 - add edge  $(v, w)$  to spanning tree  
 - insert  $w$  at rear of queue and mark as "discovered"  
until queue empty

BFS starting in  $a$

DEQUEUED VERTEX	QUEUE (newly enqueued vertices blue)
-	$a$
$a$	$b, d, g$
$b$	$d, g, c, e$
$d$	$g, c, e, h$
$g$	$c, e, h, i$
$c$	$e, h, i, f$
$e$	$h, i, f$
$h$	$i, f$
$i$	$f$
$f$	—



# Pseudo-code for breadth-first search

GT III

## BFS(G, s)

```

1   for all  $v \in V(G)$ 
2      $v.\text{discovered} := \text{FALSE}$ 
3      $s.\text{discovered} := \text{TRUE}$ 
4      $s.\text{distance} := 0$ 
5     Q. enqueue(s)
6   while NOT (Q. empty())
7      $v := Q.\text{dequeue}()$ 
8     for all  $w \in N(v)$ 
9       if NOT ( $w.\text{discovered}$ )
10       $w.\text{discovered} := \text{TRUE}$ 
11       $w.\text{distance} := v.\text{distance} + 1$ 
12      w.parent := v
13      Q.enqueue(w)
    }
```

loop runs  $|V|$  times

$\sum_v^{} \deg(v)$  for loop steps

$O(1)$

## TIME COMPLEXITY

Assumes  
adjacency list  
representation

Initialization (lines 1 - 5)  $O(|V|)$

while loop on line 6  $|V|$  times

nested for loop total of  $\sum_v^{} \deg(v) = 2 \cdot |E|$   
iterations, constant  $O(1)$  work per iteration

So in total

$$O(|V|) + O(|V|) + O(|E|) = \\ O(|V| + |E|)$$

running time for BFS

## Pseudo-code for depth-first search

| GT IV

For each vertex  $v$ :

V <sub>i</sub> . dtime	discovery time
V <sub>i</sub> . ftime	finish time
V <sub>i</sub> . pred	predecessor in spanning tree
V <sub>i</sub> . colour	
◦ WHITE	non-discovered
◦ GRAY	discovered, under processing
◦ BLACK	finished

$DFS(G, s)$

- 1 for all  $v \in V(G)$
  - 2 v.colour := WHITE
  - 3 time := 0
  - 4 DFS-VISIT ( $G, s$ )

$O(|V|)$

$O(1)$

$\text{DFS-VISIT}(G, v)$

- ```

1   v. dtime := time
2   time := time + 1
3   v. colour := GRAY
4   for all w ∈ N(v)           deg(v) times
5       if (w. colour == WHITE)
6           w. pred := v
7           !DFS-VISIT(G, w)
8   v. colour := BLACK
9   v. ftime := time
10  time := time + 1
    
```

} O(1)

Assumes adjacency list representation

## TIME COMPLEXITY OF DFS

GTV

DFS main procedure  $O(|V|)$

DFS-VISIT called  $|V|$  times

Each call takes time  $O(\deg(v)) + O(1)$

Total time complexity

$$\begin{aligned} & O(|V|) + \sum_v O(\deg(v)) + O(|V|) \\ &= O(|V|) + O(|E|) = \\ & O(|V| + |E|) \quad \text{for DFS} \end{aligned}$$

## SOME PROPERTIES OF DFS

In general, DFS can be run on directed graph (connected or not)

Change main procedure to

### DFS ( $G$ )

- 1 for all  $v \in V(G)$
- 2     $v.\text{colour} := \text{WHITE}$
- 3     $\text{time} := 0$
- 4 for all  $v \in V(G)$ 
  - 5    if ( $v.\text{colour} == \text{WHITE}$ )
    - 6       $\text{DFS-VISIT}(G, v)$

Get a collection of directed trees (a FOREST)  
that together cover all vertices of  $G$

## PARENTHESIS THEOREM (THEM 20.7)

In depth-first search of (directed or undirected)  $G$ , for any two vertices  $u, v$  exactly one of the following 3 conditions hold

- (1) Intervals  $[u.\text{dtime}, u.\text{ftime}]$  and  $[v.\text{dtime}, v.\text{ftime}]$  are entirely disjoint and neither  $u$  nor  $v$  is descendant of the other in depth-first forest
- (2)  $[u.\text{dtime}, u.\text{ftime}] \subseteq [v.\text{dtime}, v.\text{ftime}]$  and  $u$  is descendant of  $v$
- (3)  $[v.\text{dtime}, v.\text{ftime}] \subseteq [u.\text{dtime}, u.\text{ftime}]$  and  $v$  is descendant of  $u$

Proof By case analysis

Suppose  $u.\text{dtime} < v.\text{dtime}$

Subcase (a):  $v.\text{dtime} < u.\text{ftime}$

Then  $v$  descendant of  $u$ .

$\Rightarrow$  Recursive call for  $v$  finishes before recursive call for  $u$

$\Rightarrow v.\text{ftime} < u.\text{ftime}$  so  $[v.\text{dtime}, v.\text{ftime}] \subseteq [u.\text{dtime}, u.\text{ftime}]$

Subcase (b):  $v.\text{dtime} > u.\text{ftime}$  so

$u.\text{dtime} < u.\text{ftime} < v.\text{dtime} < v.\text{ftime}$

Neither vertex was discovered while other was grey, so neither is descendant of other

Other case is  $u.\text{dtime} > v.\text{dtime}$   
Run same proof but with  $u$  &  $v$  exchanged  $\square$

### COROLLARY (COR 20.8)

$v$  is a proper descendant of  
 $u$  in depth first forest for  
(directed or undirected)  $\Leftrightarrow$

IF AND ONLY IF

$u.\text{dtime} < v.\text{dtime} \Leftarrow v.\text{fime} < u.\text{fime}$

Proof Immediate from  
Parenthesis Theorem

## WHITE-PATRE THEOREM (THM 20.7)

In depth-first forest for (directed or undirected)  $G$  it holds that  $w$  is descendant of  $v$

IF AND ONLY IF

at time  $v$ .done there is a path in  $G$  from  $v$  to  $w$  consisting only of white vertices

Intuition The only way  $w$  can become descendant of  $v$  is by chain of DFS-visit calls to white vertices along path to  $w$

### Formal proof

( $\Rightarrow$ ) If  $w = v$ , then vacuously true.

Suppose  $w$  any proper descendant of  $v$

By corollary of Parenthesis Theorem, all such  $w$  have  $w\text{.done} > v\text{.done}$ , i.e. whole subtree of descendants is white

( $\Leftarrow$ ) Suppose  $v \xrightarrow[\text{path}]{\text{white}} w$

Argue by induction over length of path

Base case  $w = v$  trivial

## Induction hypothesis

For white paths of length  $\leq k$  from  $v$  to  $w$   
it holds that  $w$  descendant of  $v$

## Induction step

Consider path of length  $k+1$

$v \xrightarrow{\text{white}} w' \xrightarrow{\text{white}} w$

By IH,  $w'$  descendant of  $v$

By corollary  $w'.\text{frame} \leq v.\text{frame}$

$w$  must be discovered after  $v$ ,  
since  $v.\text{dtime} < w'.\text{dtime}$

$w$  is definitely discovered before  $w'$   
is finished. Hence

$v.\text{dtime} < w'.\text{dtime} \leq w'.\text{frame} \leq v.\text{frame}$

But then by Parenthesis Theorem

$v.\text{dtime} < w'.\text{dtime} < w.\text{frame} \leq v.\text{frame}$   
and  $w$  is also descendant of  $v$

## CATEGORIZATION OF DIRECTED EDGES IN G after DFS on G

- 1) TREE EDGES  $(u, v)$  edges in the depth-first forest
- 2) BACK EDGES  $(u, v)$  connecting vertex u back to ancestor in depth-first tree
- 3) FORWARD EDGES  $(u, v)$  non-tree edge connecting u to descendant v in depth-first tree
- 4) CROSS EDGES  $(u, v)$  all other edges in G

Type of edge  $(u, v)$  determined when out-neighbour  $v$  inspects in call  $\text{DFS-VISIT}(G, u)$ :

- If  $v$  WHITE, then  $(u, v)$  tree edge
- If  $v$  GRAY, then  $(u, v)$  back edge
- If  $v$  BLACK, then  $(u, v)$  forward edge  
or cross edge.

We will care most about tree edges and back edges. See CLRS pages 567 - 570 for more about all of this

## APPLICATION: TOPOLOGICAL SORT

GT VII

We have seen that BFS can be used to compute distances in graph from start vertex s

DFS can be used to compute topological sort

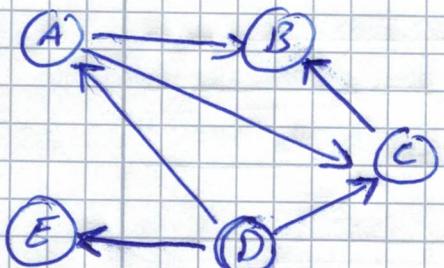
### DIRECTED ACYCLIC GRAPH (DAG):

Directed graph with no cycles

### TOPLOGICAL SORT of DAG $G = (V, E)$

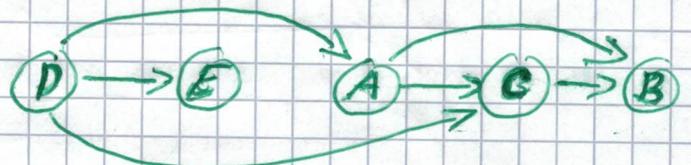
Linear order of vertices  $V$  such that  
if  $(u, v) \in E$ , then  $u$  comes before  $v$   
in the order  
(not necessarily unique)

#### Example



Sorted from left to right

$D, E, A, C, B$



(Are there other topological sorts?)

DAG can encode scheduling constraints  
 $(u, v)$ :  $u$  has to happen before  $v$

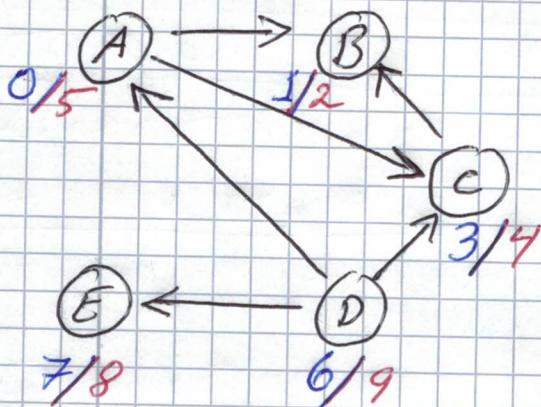
Topological sort schedules events so as to satisfy these constraints

TOPOLOGICAL-SORT ( $G$ )

- 1 Call  $\text{DFS}(G)$  to compute finishing times
- 2 As vertex  $v$  finished, add to front of list
- 3 If back edge  $(v, w)$  for  $w \in \text{GRAY}$  detected, abort
- 4 return list of vertices sorted in decreasing finishing time

Time complexity:

$$\boxed{O(|V| + |E|)}$$

DFSDISCOVERY TIME  
FINISHING TIME

visit A  
visit B; finish B  
visit C; finish C  
finish A  
visit D  
visit E; finish E  
finish D

D, E, A, C, B

Why is this algorithm correct?

## LEMMA

Directed graph  $G$  acyclic iff depth-first search yields no back edges.

Proof. ( $\Rightarrow$ ) Suppose DFS produces back edge  $(u, v)$ . Then  $v$  ancestor of  $u$  in depth-first tree, so  $\exists$  path  $v \rightarrow \dots \rightarrow u$ . Back edge  $(u, v)$  completes a cycle, so  $G$  is not acyclic.

( $\Leftarrow$ ) Suppose  $G$  contains cycle  $c$ . Suppose  $v$  first vertex discovered. Then by white-path theorem, all other vertices in cycle will be descendants. Final edge to  $v$  back edge  $\blacksquare$

## THEOREM

GT IX

$\text{TOPLOGICAL-SORT}(G)$  sorts  $V(G)$  topologically, i.e., if  $(u, v) \in E$ , then  $u$  comes before  $v$ .

Proof. It is sufficient to prove that for edge  $(u, v)$  it holds that  $u.\text{fime} > v.\text{fime}$ .

Consider the call  $\text{DFS-VISIT}(u)$ , when edge  $(u, v)$  is explored.

Case analysis for colour of  $v$ :

- (i)  $v$  GRAY: Impossible — by previous lemma we have back edge and cycle
- (ii)  $v$  WHITE:  $\text{DFS-VISIT}(v)$  is called  
 $v.\text{fime}$  is set before call returns  
 $u.\text{fime} > v.\text{fime}$
- (iii)  $v$  BLACK:  $v$  is already finished, so  $v.\text{fime}$  has been set.  
But  $\text{DFS-VISIT}(u)$  still runs, so  $u.\text{fime}$  will be set later, i.e.,  
 $u.\text{fime} > v.\text{fime}$  

## APPLICATION: STRONGLY CONNECTED COMPONENTS

GT X

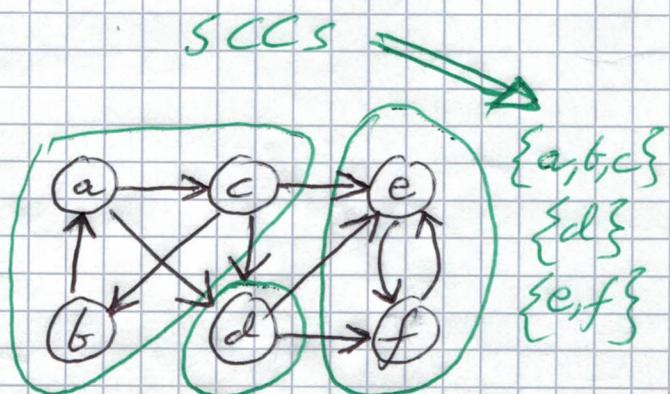
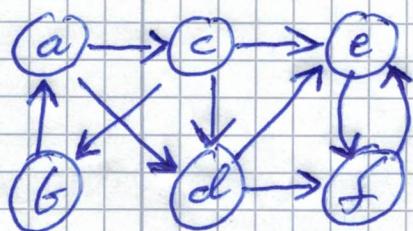
STRONGLY CONNECTED COMPONENT of directed graph  $G = (V, E)$

Maximal vertex set  $C \subseteq V$  such that for  $u, v \in C$   $u \rightarrow v$  and  $v \rightarrow u$

Special case: Always  $v \rightarrow v$

Strongly connected components (SCCs) form a partition of  $V(G)$ , since being in the same strongly connected component is an equivalence relation

### EXAMPLE GRAPH G



Given directed graph  $G = (V, E)$   
compute SCCs in linear time  $O(|V| + |E|)$

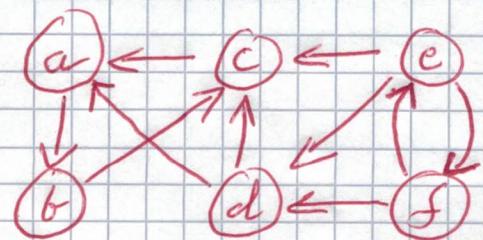
### TRANSPOSE OF $G = (V, E)$

$G^T = (V, E^T)$  for

$$E^T = \{ (v, u) \mid (u, v) \in E \}$$

(Edge directions reversed  
Transpose of adjacency matrix)

$G^T$



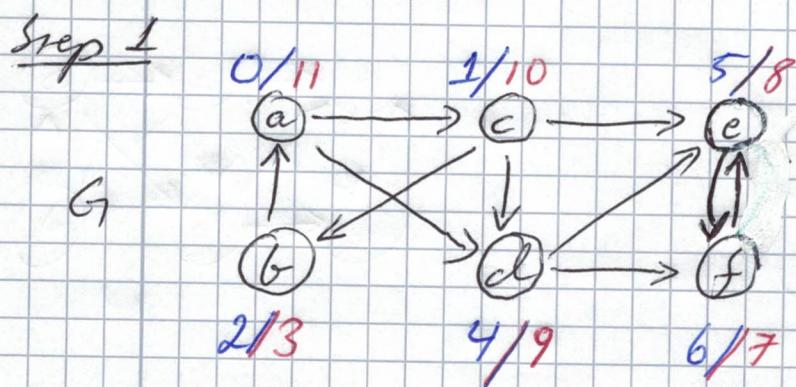
Observe that  $G$  and  $G^T$  have the same strongly connected components. (WHY?)

Here is our algorithm

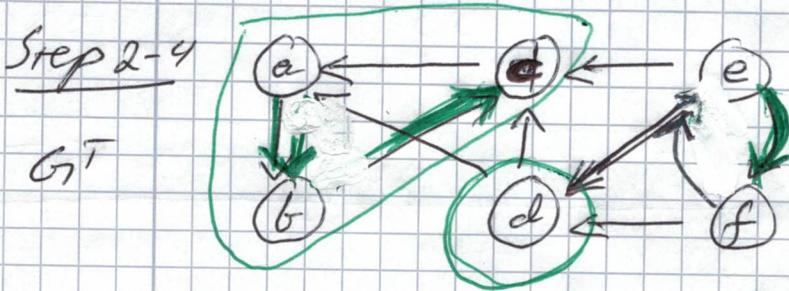
### SSC ( $G$ )

1. Call  $\text{DFS}(G)$ ; store finishing times  $v.\text{fTime}$
2. Compute  $G^T$
3. Call  $\text{DFS}(G^T)$ , but let main loop of DFS consider vertices in decreasing order of  $v.\text{fTime}$
4. Output vertices of each depth-first tree found by  $\text{DFS}(G^T)$  as a separate SCC

All steps clearly doable in time  $O(|V| + |E|)$



DISCOVERY TIME  
FINISH TIME



SCCs

- {a, b, c}
- {d}
- {e, f}

- (i) First a tree  $a \rightarrow b \rightarrow c$  (ii) Then d tree  $d$  (iii) Finally e tree  $e \rightarrow f$

Why is this correct?

GT XII

Detailed argument: CDRS pages 617-620

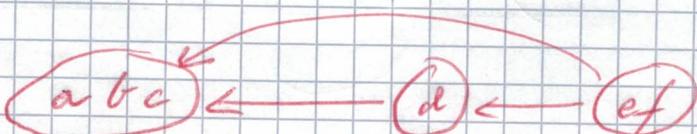
High level intuition

- ① Consider component graph  $G^{SCC}$ , with each SCC contracted to single vertex

Our graph:

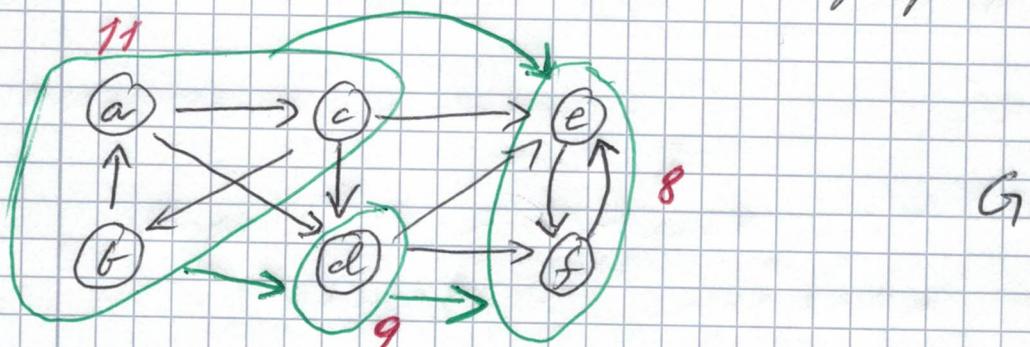


- ②  $G^{SCC}$  must be a DAG — if there were a cycle, then SCC components could be made larger
- ③ In step 3, we essentially run a topological sort of  $G^{SCC}$ , outputting the strongly connected components in topologically sorted order
- ④ Using last finishing time + reversing edges ensures that we list all vertices in every component



- (i) First  $a$  reaches  $a, b, c$   
(ii) Then  $d$  reaches  $d$ ;  $a, b, c$  already discovered  
(iii) Finally  $e$  reaches  $ef$ ;  $a, b, c, d$  already discovered

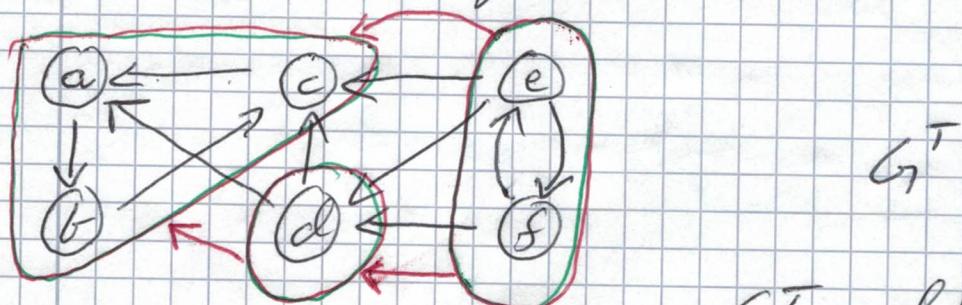
Consider the component graph



Look at the last finishing time in each component — this sorts the SCCs in topological order.

The problem is, we don't yet know what the SCCs are...

If we look at the transposed graph, then the component graph will also be transposed:



If we could somehow start in the first component, then we would visit all vertices in that component and build a tree out of them.

Then starting from any vertex in the second component would visit all of the second component, but not the first (since those vertices have already been visited)

Then starting from any vertex in 3rd component would visit the 3rd component but not components 1 & 2. GT XIV

### Example

- ① Visiting any of a, b, c in  $G^T$  recovers component  $\{a, b, c\}$
- ② Then visiting d yields only  $\{d\}$ , because a, b, c already visited
- ③ Finally, starting with either e or f yields  $\{e, f\}$ , but all other vertices are already visited

But how can we visit components in  $G^T$  in the right order, when we don't know the components?

ANSWER: Visit vertices in order of decreasing finishing times from  $\text{DFS}(G)$

Highest finishing time must be in 1st component.

When that call is finished, the highest finishing time of any non-discovered vertex must be from the 2nd component.

Etcetera... The formal argument on pages 617-620 in CLRS makes this intuition mathematically precise.