GRAPH THEORY

GRAPH: ordered pair $G = (V, E)$ of
- set of **VERTICES** (or NODES) V
- 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

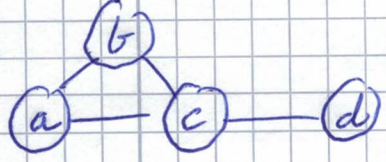
DIRECTED GRAPH

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



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

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

We will ~~reveal~~ 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 same 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 $\quad N(v) = \{w \in V \mid \{v,w\} \in E\}$

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

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

$\deg(a) = 2$

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

DIRECTED GRAPHS

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

OUT-DEGREE $\quad \deg_{out}(v) = |N_{out}(v)|$

IN-NEIGHBOURS $\quad N_{in}(v) = \{u \in V \mid (u,v) \in E\}$

IN-DEGREE $\quad \deg_{in}(v) = |N_{in}(v)|$

A **PATH** $v_0 \rightsquigarrow v_n$ (in an undirected or directed graph) is a sequence $v_0, v_1, \ldots, v_n$ such that for all $i \in [n]$ $(v_{i-1}, v_i) \in E$

Length = # edges (= n above)

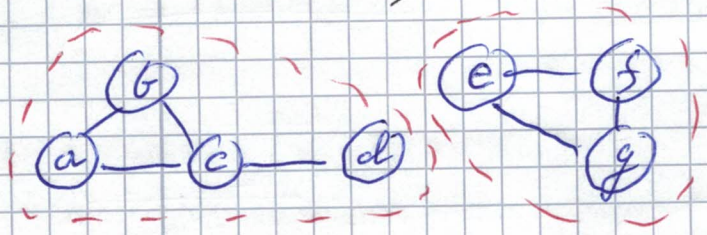
A **SIMPLE PATH** is a path where no vertex repeats

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 graphs, more detailed definitions of connectedness are used

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

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

<u>EXAMPLE</u>

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

$a \to b \to c \to a$ is a cycle

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"

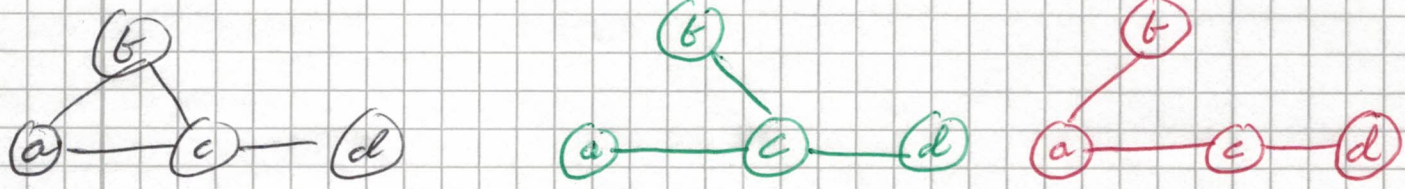
$G' = (V', E')$ is a **SUBGRAPH** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$

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"

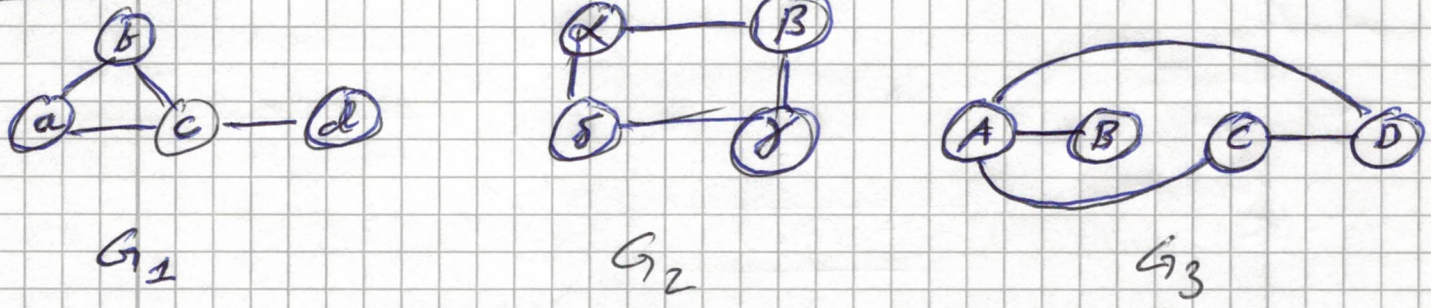
Formally, there exists a bijective function

$f: V_1 \to V_2$ 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

$G_1 \ncong G_2$ $G_1 \cong G_3$

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

(Doesn't matter whether vertices are called
- a, b, c, ...
- $\alpha, \beta, \gamma, \ldots$
- A, B, C, ...

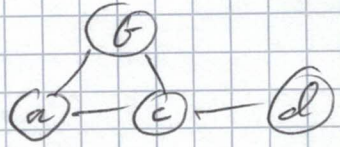
Want to understand if graphs are STRUCTURALLY the same)

GRAPH REPRESENTATION

G 1 V

ADJACENCY MATRIX

Rows and columns indexed by V; 1 in all (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)

$N(a) = b, c$

$N(b) = a, c$

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

$N(d) = c$

DETOUR

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

For $G = (V, E)$ $H = (V', E')$ clearly 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

- Adjacent(u,v) – is there an edge (u,v)?
- Neighbours(u) – list of (out-) neighbours of u

	Adjacency matrix	Adjacency list
Adjacent(u,v)	$O(1)$	$O(\deg(u))$
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.

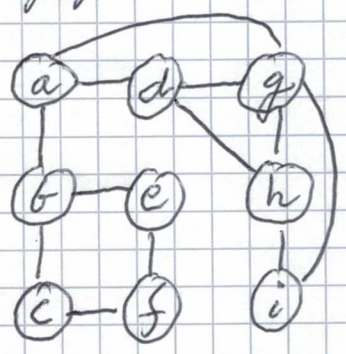
GRAPH TRAVERSAL

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 (undirected case) (otherwise deal with 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

$v.marked$ – has v been visited
$v.d$ – discovery time / distance
$v.f$ – finish time (for DFS)
$v.\pi$ – edge leading to v in spanning tree (if viewed as directed)

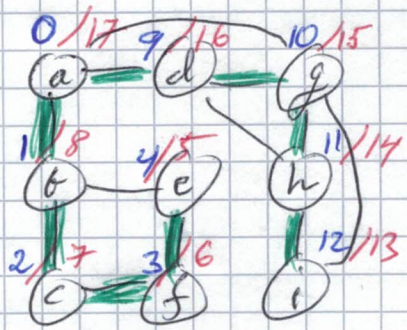
DEPTH-FIRST SEARCH

Recursive algorithm

Mark v as discovered
Visit all non-visited (own) neighbours u of v
    Add such edges (u,v) to spanning tree
Mark v as finished

```
DFS (s)
O(|V|) { for all v ∈ V
             v.marked := FALSE
         time := 0
         DFS-visit (s)
```

Discovery time / Finishing time




```
DFS-visit (v)
    v.d := time
    time := time + 1                          } O(1)
    v.marked := TRUE
    for all u ∈ N(v)                          } deg(v) times
        if NOT(u.marked)                      } O(1)
            DFS-visit (u)
            u.π := v
    v.f := time                               } O(1)
    time := time + 1
```

out-neighbours for directed graph

Example: DFS(a)

```
visit(a)
    visit (b) : a marked
        visit (c) : b marked
            visit (f) : c marked
                visit (e) : everything marked
                            now e 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
```

SMALL NOTE: Time above starts from 0

In CLRS, start time from 1

Time complexity (see also previous page) G IX

Initialization $O(|V|)$

DFS-visit called $|V|$ times

Work $O(1) + O(\deg(v))$

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

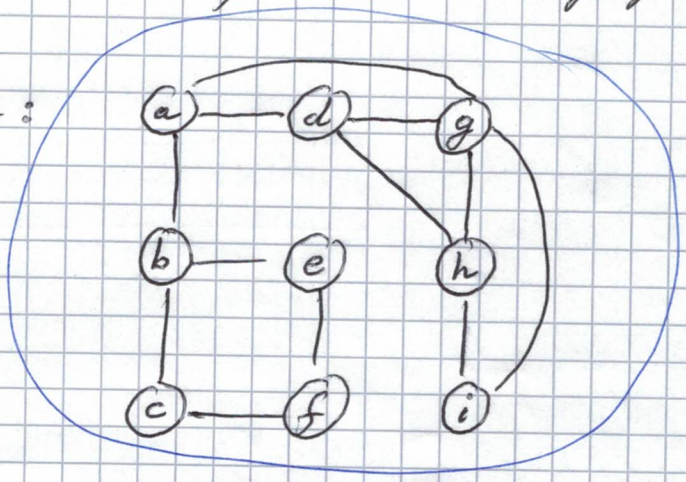
GTI

- 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:

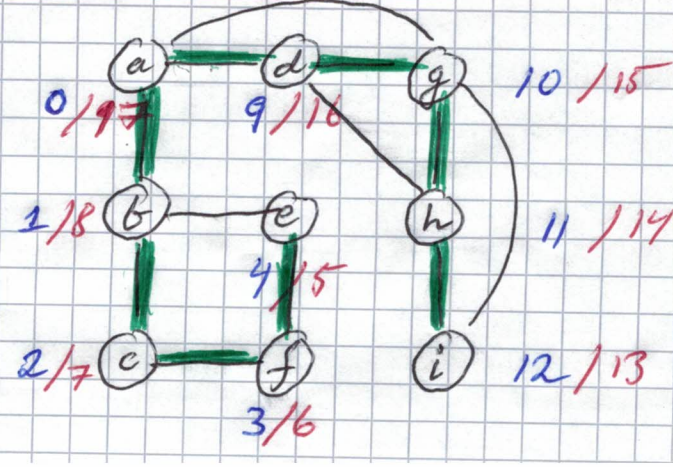


DEPTH-FIRST SEARCH

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

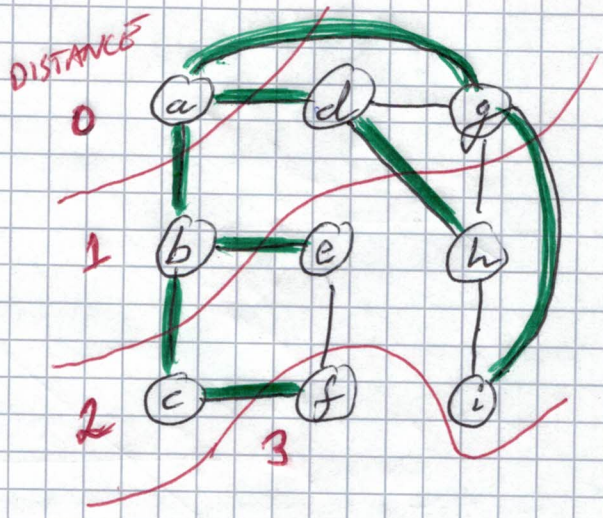
GT II

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 done)
—	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

BFS(G, s)

```
1   for all v ∈ V(G)                          } O(|V|)
2       v.discovered := FALSE                 }
3   s.discovered := TRUE                      } O(1)
4   s.distance := 0                           }
5   Q.enqueue(s)
6   while NOT(Q.empty())                      loop runs |V| times
7       v := Q.dequeue()
8       for all w ∈ N(v)                      TOTAL OF Σ_v deg(v) for loop steps
9           if NOT(w.discovered)
10              w.discovered := TRUE          }
11              w.distance := v.distance + 1  } O(1)
12              w.pred := v                   }
13              Q.enqueue(w)
```

Assumes adjacency list representation

TIME COMPLEXITY

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:

- $v.dtime$ discovery time
- $v.ftime$ finish time
- $v.pred$ predecessor in spanning tree
- $v.colour$
  - WHITE non-discovered
  - GRAY discovered, under processing
  - BLACK finished

DFS (G, s)

```
1    for all v ∈ V(G)                    } O(|V|)
2        v.colour := WHITE               }
3    time := 0                           } O(1)
4    DFS-VISIT ( G, s )                  }
```

DFS-VISIT (G, v)

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

Assumes adjacency list representation

TIME COMPLEXITY OF DFS

DFS main procedure $O(|V|)$

DFS-VISIT called $|V|$ times

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

Total time complexity

$$O(|V|) + \sum_{v} O(\deg(v)) + O(|V|)$$

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

$O(|V| + |E|)$ for DFS

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 ∈ V(G)
2        v.colour := WHITE
3    time := 0
4    for all v ∈ V(G)
5        if (v.colour == WHITE)
6            DFS-VISIT(G, v)
```

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

WHITE-PATH THEOREM

In a depth-first forest for $G=(V,E)$, vertex w is a descendant of vertex v IF AND ONLY IF at time $v.d$ time there is a path from v to w consisting only of white vertices

Proof. See pages 608-609 in CLRS.

(Intuitively, the only way that w can become descendant of v is by a chain of DFS-VISIT calls to white vertices along a ~~path~~ to w

CLASSIFICATION OF (DIRECTED) EDGES after DFS

1) TREE EDGES in 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: all other edges in G

Type of edge (u,v) determined when out-neighbour v inspected in call DFS-VISIT(u):

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

We will care most about tree edges & back edges

See CLRS pages 609-610 for more details

APPLICATION: TOPOLOGICAL SORT

GT VII

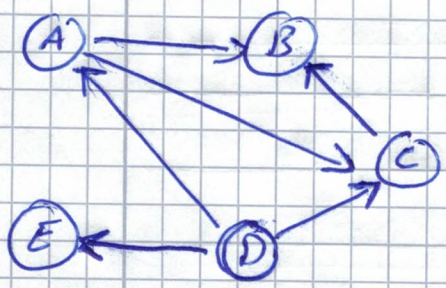
We have seen that BFS can be used to compute distances in graph from start vertex s

DFS can be used to compute topological sort

DIRECTED ACYCLIC GRAPH (DAG):
Directed graph with no cycles

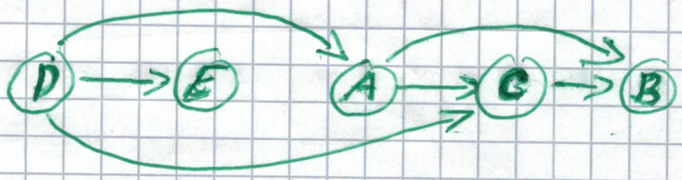
TOPOLOGICAL SORT of DAG $G=(V,E)$
Linear order of vertices V such that if $(u,v) \in E$, then u comes before v in the order (NOT NECESSARILY UNIQUE)

EXAMPLE



Sorted from left to right

D, E, A, C, B



(Are there other topological sorts?)

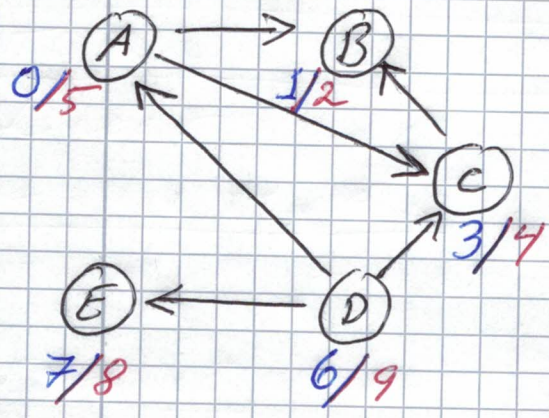
DAG can encode scheduling constraints
(u,v) : u has to happen before v

Topological sort schedules events so as to satisfy these constraints

TOPOLOGICAL-SORT (G)

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

Time complexity: $O(|V|+|E|)$



DFS

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

DISCOVERY TIME
FINISHING TIME

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 $v \rightsquigarrow u$. 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 ☐

THEOREM

TOPOLOGICAL-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 $u.finish > v.finish$

Consider the call 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: DFS-VISIT(v) is called
$v.finish$ is set before call returns
$u.finish > v.finish$

(iii) v BLACK: v is already finished, so
$v.finish$ has been set.
But DFS-VISIT(u) still runs, so
$u.finish$ will be set later, i.e.
$u.finish > v.finish$ ▤

APPLICATION: STRONGLY CONNECTED COMPONENTS

GT X

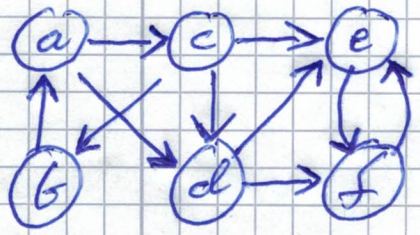
STRONGLY CONNECTED COMPONENT of directed graph $G = (V, E)$

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

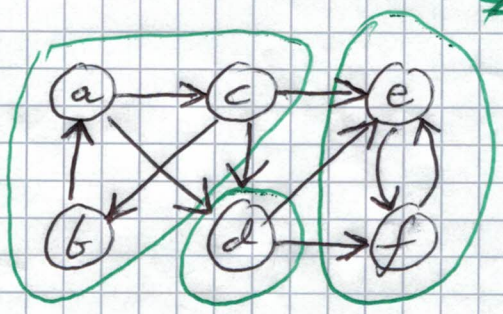
Special case: Always $v \rightsquigarrow v$

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



SCCs



Given directed graph $G = (V, E)$

compute SCCs in linear time $O(|V| + |E|)$

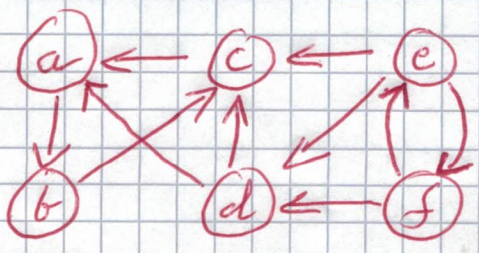
TRANSPOSE OF $G = (V, E)$

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

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

(Edge directions reversed
Transpose of adjacency matrix)

G^T



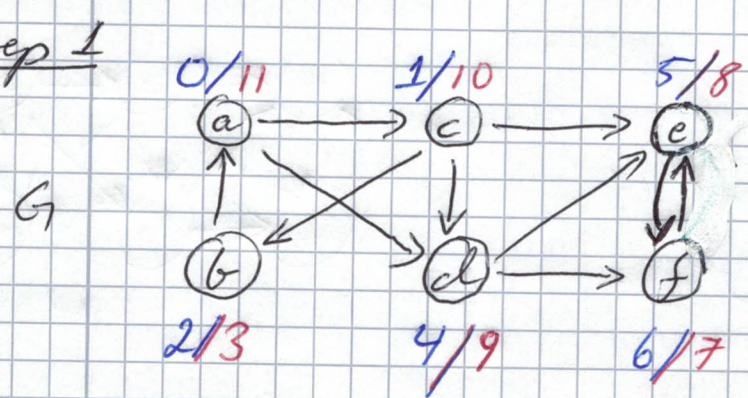
Observe that G and G^T have the same strongly connected components. (WHY?)

Here is our algorithm

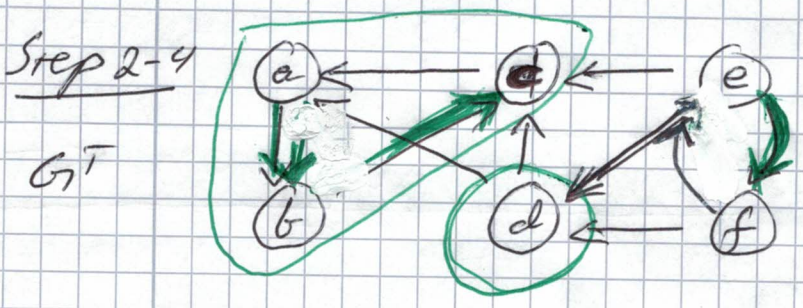
SSC (G)
1. Call DFS(G); store finishing times v. time
2. Compute G^T
3. Call DFS(G^T), but let main loop of DFS consider vertices in decreasing order of v. time
4. Output vertices of each depth-first tree found by DFS(G^T) as a separate SCC

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

Step 1



Step 2-4



SCCs

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

(i) First a tree a → b → c
(ii) Then d tree d
(iii) Finally e tree e → f

Why is this correct?

Detailed argument: CLRS pages 617-620

High level intuition

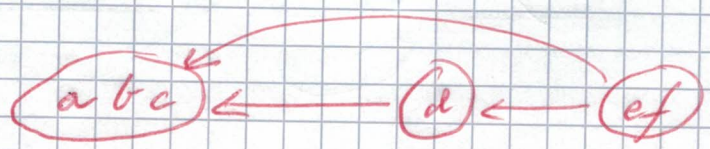
(1) Consider component graph G^{SCC}, with each SCC contracted to single vertex

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

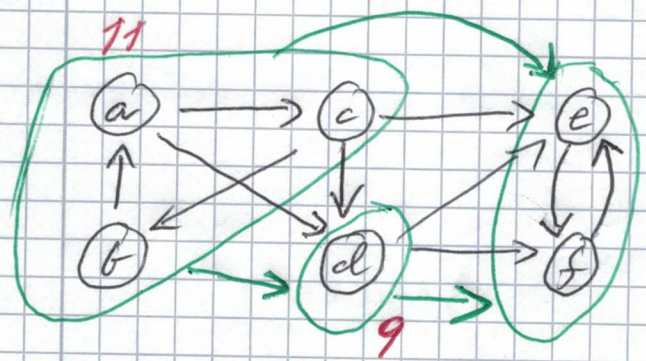
(4) Using last finishing time + reversing edges ensures that we list all vertices in every component



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

EXTRA NOTES ON SCC AFTER THE LECTURE GT XIII

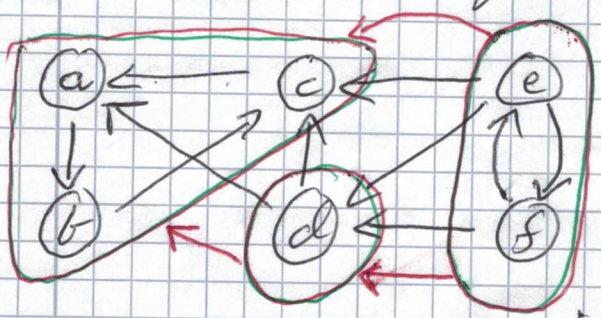
Consider the component graph



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 use G^T and 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 component 1 & 2.

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 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.

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