Design and Analysis of Algorithms

Part 5

Graph Decomposition

Elias Koutsoupias
with thanks to Giulio Chiribella

Hilary Term 2021

Why graphs? [DPV 3.1]

Graphs are one of the most fundamental notions in CS:
Many CS problems have an underlying graph structure.

Example: Colouring a map

Problem: what is the minimum number of colours needed so that neighbouring countries have different colours?

Graph formulation:

□ One vertex for each country.

□ Two vertices are linked by an edge if they represent neighbouring countries.

The original problem can be reduced to a graph problem, known as the **Graph Colouring Problem:**
find the minimum number of colours needed to colour the vertices of the graphs so that no edge has endpoints of the same colour.

Basic definitions: directed graphs, paths, cycles

A ***directed graph*** (V, E) consists of a set V of *nodes* (or *vertices*) and a set $E \subseteq V \times V$ of *edges*, each edge e being an ordered pair (u, v) of nodes; u is the *source* of e, and v is the *target* of e; we say that e is *incident on* u and v. In this case, we also say that u and v are *adjacent*.

A ***path*** of length k from a vertex u to a vertex u' is a sequence $\langle v_0, v_1, \cdots, v_k \rangle$ of vertices such that $u = v_0$, $u' = v_k$, and each $(v_i, v_{i+1}) \in E$. If there is a path from u to u', then u' is *reachable* from u.

A path $\langle v_0, v_1, \cdots, v_k \rangle$ forms a ***cycle*** if $v_0 = v_k$ and the path contains at least one edge. A *self-loop* is a cycle of length 1. The cycle is *simple* if, in addition, $v_1, \cdots, v_k$ are distinct.

An ***acyclic*** directed graph (dag) is a directed graph with no cycles.

A graph (V, E) is ***undirected*** if E is symmetric i.e. $(u, v) \in E$ iff $(v, u) \in E$.

Two representations of graphs [DPV 3.1.1, CLRS 22.1]

A graph $G = (V, E)$ with $V = \{ v_0, \cdots, v_{n-1} \}$ can be represented by the following data structures, ***adjacency matrix*** and ***adjacency lists***.

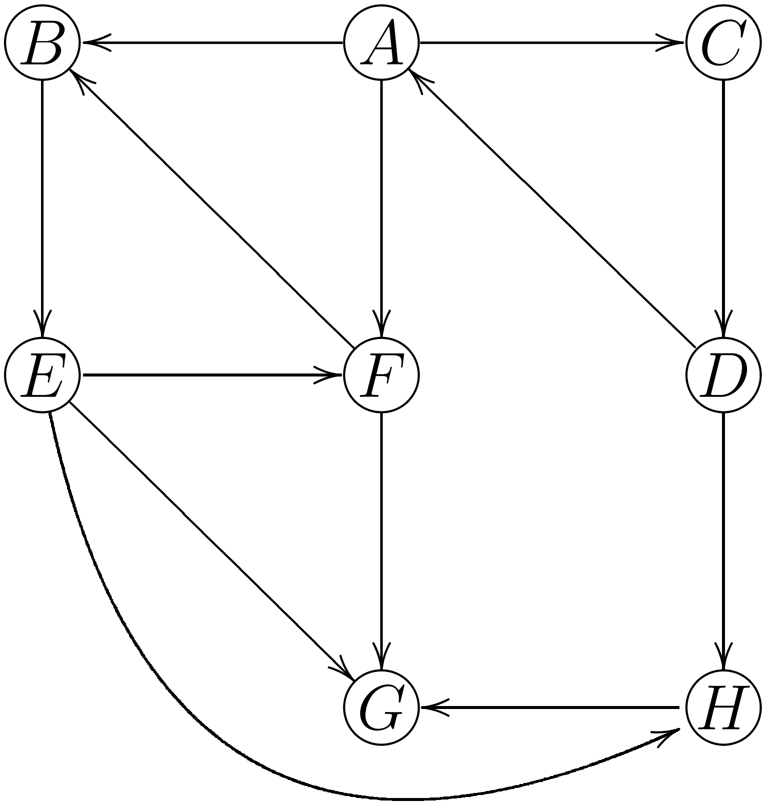
Adjacency matrix

$n \times n$ array whose (i, j)-th entry is

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

- □ Presence of an edge can be checked in constant time.
- □ Data structure has size $O(|V|^2)$.
- □ For an undirected graph, the matrix is symmetric.

Example: adjacency matrix



	A	B	C	D	E	F	G	H
A	0	1	1	0	0	1	0	0
B	0	0	0	0	1	0	0	0
C	0	0	0	1	0	0	0	0
D	1	0	0	0	0	0	0	1
E	0	0	0	0	0	1	1	1
F	0	1	0	0	0	0	1	0
G	0	0	0	0	0	0	0	0
H	0	0	0	0	0	0	1	0

Two representations of graphs, cont'd

Adjacency lists

It consists of $|V|$ linked lists, one per vertex. The list for vertex u holds the names of vertices to which u has an outgoing edge.

□ Presence of an edge is not checkable in constant time.

□ Data structure has size $O(|V| + |E|)$.

□ For an undirected graph, u is in v's adjacency list iff v is in u's.

Two representations of graphs, cont'd

Adjacency lists

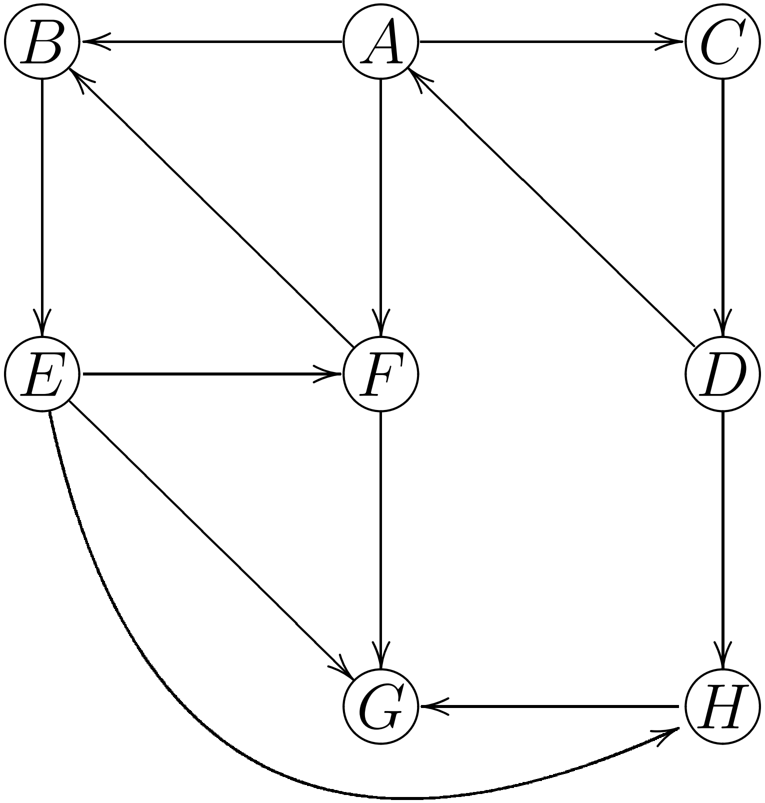
It consists of $|V|$ linked lists, one per vertex. The list for vertex u holds the names of vertices to which u has an outgoing edge.

- □ Presence of an edge is not checkable in constant time.
- □ Data structure has size $O(|V| + |E|)$.
- □ For an undirected graph, u is in v's adjacency list iff v is in u's.

Question

Which would you use to represent the World Wide Web (there is an edge between two sites if they there is a link from one to the other)?

Example: adjacency lists



$Vertices$	$Adj.\ Lists$
A	$[B, C, F]$
B	$[E]$
C	$[D]$
D	$[A, H]$
E	$[F, G, H]$
F	$[B, G]$
G	$[\ ]$
H	$[G]$

Depth-first search (DFS) [DPV 3.2.1]

DFS is a versatile linear-time algorithm that answers the basic question:

> *What parts of the graph are reachable from a given vertex?*

It works for both directed and undirected graphs.

Motivation: Exploring a maze

All one needs to explore a maze are:

1. a piece of chalk (to prevent looping), and
2. a ball of string (to enable return to passages encountered before but not yet explored).

We use the same basic idea in depth-first search of a graph.

Overview

Idea

As soon as a new vertex is discovered, explore from it.

As DFS progresses, every vertex is assigned a ***colour***:

- □ WHITE = not discovered yet
- □ GREY = discovered, but its adjacency list has not been fully explored yet
- □ BLACK = finished (i.e. all the vertices in its adjacency list have been explored).

The Algorithm [CLRS 22.3]

DFS takes a graph $G = (V, E)$, directed or undirected, and, for each vertex $v \in V$, returns a backpointer $\pi[v]$ (the “predecessor of v”) and two timestamps,

- □ ***discovery time*** $d[v]$
- □ ***finishing time*** $f[v]$

```
DFS$(V, E)$

    **Input**: Graph $G = (V, E)$, directed or undirected
    **Output**: Timestamps $d[v]$ and $f[v]$, and predecessor $\pi[v]$ for each vertex $v$
    **for** $u \in V$
        $colour[u] = \text{WHITE}$        $\pi[u] = \text{NIL}$
    $time = 0$
    **for** $u \in V$
        **if** $colour[u] = \text{WHITE}$
            DFS-VISIT$(u)$
```

Algorithm: DFS-Visit

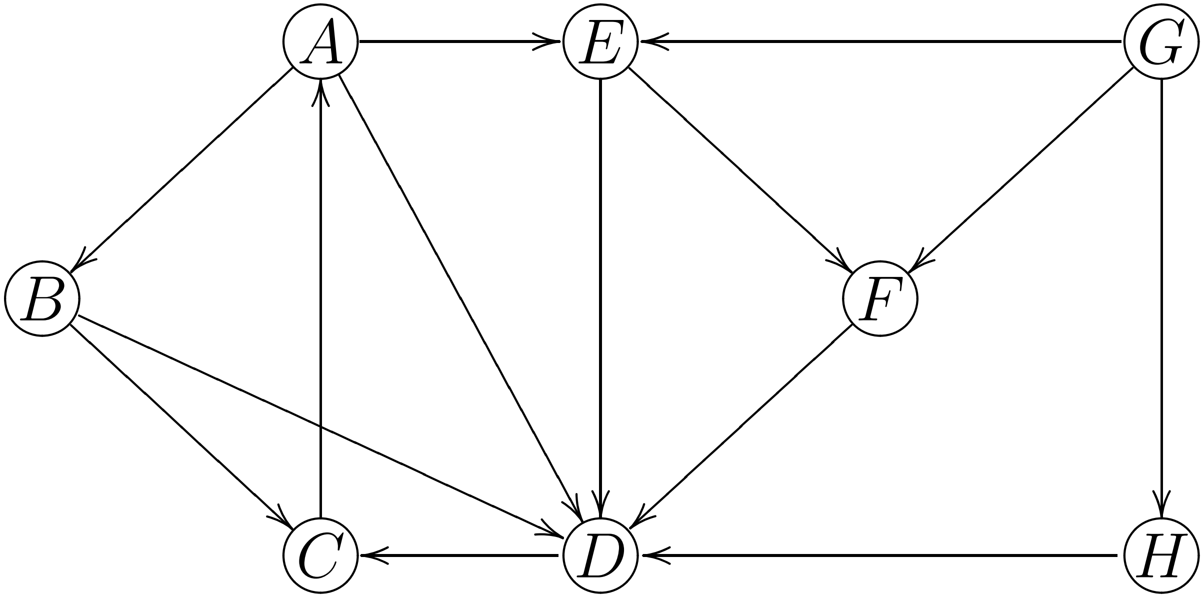
DFS-VISIT(u) assigns timestamps $d[v]$ and $f[v]$ to all vertices reachable from u (including u itself), and a predecessor $\pi[v]$ to all vertices $v \neq u$.

```
DFS-VISIT$(u)$

$time = time + 1$            // vertex $u$ has been discovered
$d[u] = time$                // record discovery time
$colour[u] = $ GREY          // mark vertex $u$ visited
for $v \in Adj[u]$           // explore from $v$ and come back once finished
    if $colour[v] = $ WHITE
        $\pi[v] = u$
        DFS-VISIT$(v)$
$time = time + 1$            // vertex $u$ has been finished
$f[u] = time$                // record finishing time
$colour[u] = $ BLACK         // mark vertex $u$ finished
```

Remark: For all $u \in V$, one has $1 \leq d[u] < f[u] \leq 2\,|V|$.

Example



	Adj. Lists
A	$[B, D, E]$
B	$[C, D]$
C	$[A]$
D	$[C]$
E	$[D, F]$
F	$[D]$
G	$[E, F, H]$
H	$[D]$

	A	B	C	D	E	F	G	H
$d[\cdot]$	1	2	3	5	8	9	13	14
$f[\cdot]$	12	7	4	6	11	10	16	15
$\pi[\cdot]$	NIL	A	B	B	A	E	NIL	G

Remark: note that d, f, and π generally depend on the order in which the vertices of the graph are visited (alphabetic order in the above example), and on the order of the vertices in the adjacency lists.

Analysis of DFS running time

The loops on lines 1-2 and lines 4-6 of DFS take time $\Theta(|V|)$, excluding the time to execute the calls to DFS-VISIT.

Fact 1: DFS-VISIT is called once and only once for each $v \in V$, since it is invoked only on white vertices, and, when it runs on a white vertex it immediately paints it grey.

Fact 2: when DFS-VISIT runs on a vertex $v \in V$, it takes time $\Theta(|Adj(v)|)$.

Since $\sum_{v\in V} |Adj(v)| = |E|$, this yields $T = \Theta(|V| + |E|)$.

The DFS forest

Define the set of edges $E_\pi := \{(\pi[u], u) \; : u \in V \, , \pi[u] \neq \text{NIL}\}$ and the graph $G_\pi := (V, E_\pi)$.

The graph G_π is called the ***depth-first search (DFS) forest****.
The DFS is consists of one or more ***DFS trees****.
Each tree is composed of edges (u, v) such that, *when* (u, v) *is explored*, u is grey and v is white.

We say that u is a ***descendant*** of v just if it is so ***in the DFS forest*** G_π (not just in the original graph G).

Remark: the DFS forest depends on the order in which the vertices are listed.

* with a small abuse of notation:
normally, “forests” and “trees” are *undirected* graphs.

The Parenthesis Theorem [CLRS Theorem 22.7, p. 606]

Discovery and finishing times have a bracketing property:

Theorem 1 (Parenthesis Theorem)**.** *For all u, v, exactly one of the following holds:*

1. $d[u] < f[u] < d[v] < f[v]$ *or* $d[v] < f[v] < d[u] < f[u]$*, and neither of* u *and* v *is a descendant of the other.*
2. $d[u] < d[v] < f[v] < f[u]$ *and* v *is a descendant of* u*.*
3. $d[v] < d[u] < f[u] < f[v]$ *and* u *is a descendant of* v*.*

Using shorthand: $\boxed{\begin{array}{cccc} d[u] & f[u] & d[v] & f[v] \\ (_u & _u) & (_v & _v) \end{array}}$ the Theorem says:

- □ $(_u \quad _u) \quad (_v \quad _v)$ and $(_u \quad (_v \quad _v) \quad _u)$ are possible
- □ but $(_u \quad (_v \quad _u) \quad _v)$ cannot happen

Characterizations of descendancy in the DFS forest

Corollary 1. *Vertex v is a descendant of vertex u iff* $d[u] < d[v] < f[v] < f[u]$.

Proof. Immediate from Parenthesis Theorem. □

Theorem 2 (White Path). *Vertex v is a descendant of vertex u iff, at the time $d[u]$ that the search discovers u, there exists a path from u to v consisting entirely of white vertices.*

Proof. The "only if" part is immediate. The "if" part is by induction on the length of the path. If the path has length 1, v is in $Adj[u]$ and will become a descendant of u. Now, suppose the White Path Theorem holds for paths of length l, and let $(u, u_1, \ldots, u_l, v)$ be a white path of length $l + 1$. By induction, u_l is a descendant of u and therefore $d[u_l] < f[u]$. When u_l is discovered, either v is white (in which case v will be a descendant of u_l, and therefore of u), or it is not (in which case $d[u] < d[v] < d[u_l] < f[u]$, and the Parenthesis Theorem implies that v is a descendant of u). □

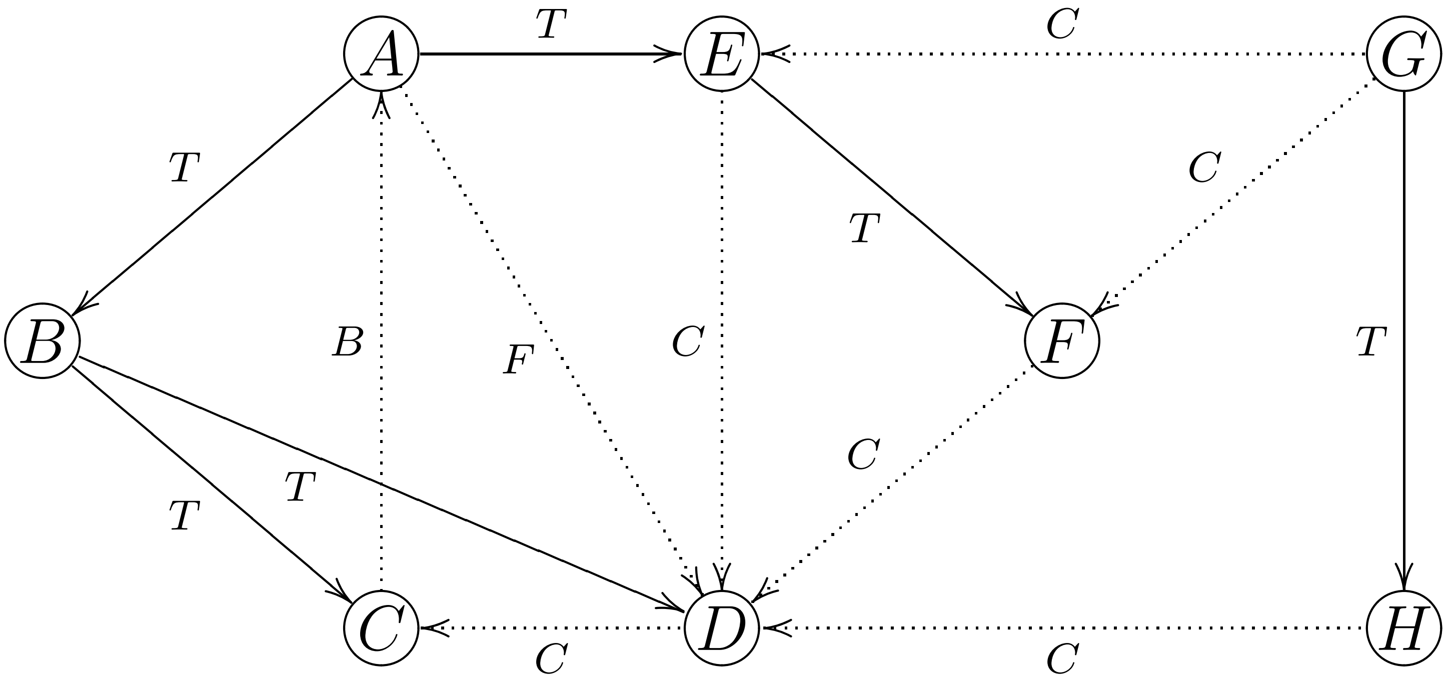
Classification of edges

- □ ***Tree*** edges (u, v) are edges of the DFS forest.
  If (u, v) is a tree edge, then v is **white** when (u, v) is explored.
- □ ***Back*** edges (u, v) lead from a node to an ancestor in the DFS tree.
  If (u, v) is a back edge, then v is **grey** when (u, v) is explored.
- □ ***Forward*** edges lead from a node u to a *non-child* descendant in the DFS tree.
  If (u, v) is a forward edge, then v is **black** when (u, v) is explored.
- □ ***Cross*** edges lead neither to a descendant nor an ancestor.
  Cross edges can link vertices in the same tree, or in different trees.
  If (u, v) is a cross edge, then v is **black** when (u, v) is explored.

Summary:

Discovery and finishing times	**Type of edge** (u, v)
$d[u] < d[v] < f[v] < f[u]$	tree or forward
$d[v] < d[u] < f[u] < f[v]$	back
$d[v] < f[v] < d[u] < f[u]$	cross
$d[u] < f[u] < d[v] < f[v]$	cannot happen

Example revisited



Edge labels: T = tree edge, B = back, C = cross, F = forward.

	A	B	C	D	E	F	G	H
$d[\cdot]$	1	2	3	5	8	9	13	14
$f[\cdot]$	12	7	4	6	11	10	16	15

Detecting cycles [CLRS 22.4, Lemma 22.11]

We can detect the presence of cycles in linear time using DFS.

Lemma 1 (Characterization)**.** *A directed graph has a cycle iff its DFS reveals a back edge.*

Proof

"$\Leftarrow$": If (u, v) is a back edge, then there is a cycle consisting of this edge with the path from v to u in the DFS tree.

"$\Rightarrow$": Suppose $\langle\, v_0, \cdots, v_k, v_0 \,\rangle$ is a cycle. Let v_i be the first node to be discovered (the node with the lowest d-number).

Since all other nodes in the cycle are reachable from v_i, they are descendants of it in the DFS tree.

Hence (v_{i-1}, v_i) $(v_{i-1}$ is v_k if $i = 0)$ is by definition a back edge. □

DAGs and schedules

A directed acyclic graph (DAG) can be used to represent the dependences among a set of events.

Examples:

□ ***completing jobs:*** represent jobs as vertices, and draw a edge from A to B if job A must be completed before job B can start.

□ ***solving subproblems:*** represent subproblems as vertices, and draw an arrow from subproblem A to subproblem B if the solution of subproblem B requires the solution of subproblem A (cf. dynamic programming).

In these examples, it is important to have a ***"schedule"*** that tells us in which order we should perform the jobs (or solve the subproblems). Such a "schedule" is called a "topological sort".

Topological sort [CLRS 22.4]

A ***topological sort*** of a DAG $G = (V, E)$ is a total ordering of vertices, $< \subseteq V \times V$, such that if $(u, v) \in E$ then $u < v$.

```
TOPOLOGICAL-SORT$(V, E)$

     Input: A DAG $(V, E)$
     Output: Elements of $V$ sorted in topological order.
1    Call DFS$(V, E)$ to compute finishing times $f[v]$ for all $v \in V$.
2    Output vertices in order of decreasing finishing times.
```

Remark 1: the algorithm defines $u < v$ iff $f[u] > f[v]$
Remark 2.: We can just output vertices as they finish, with the understanding that we want the reverse of the list; or put them in front of a linked list as they are finished. When done the list contains vertices in topologically sorted order.

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

Correctness

Correctness of the algorithm TOPOLOGICAL-SORT amounts to:

Proposition 1 (Correctness)**.** *In a DAG, if* $(u, v) \in E$ *then* $f[u] > f[v]$.

Proof. When (u, v) is explored, u is grey. Consider the colours of v.

□ v is white. Then v is a descendant of u.
By the Paranthesis Theorem, $d[u] < d[v] < f[v] < f[u]$.

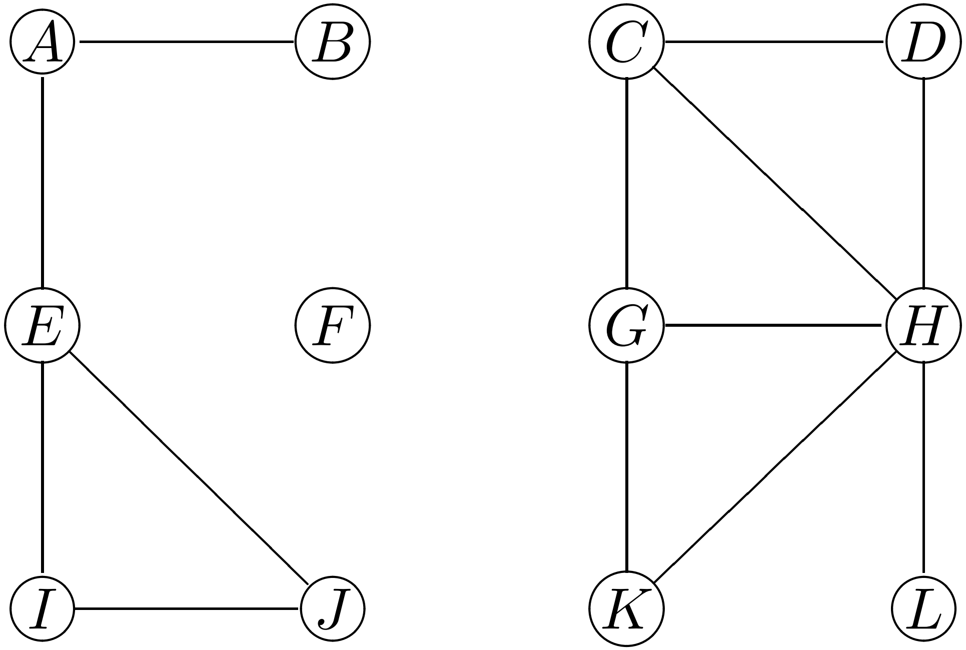
□ v is black. Then the visit of v is already finished.
Since the visit of u has not finished yet, $f[v] < f[u]$.

□ v is grey. This case cannot occur. If v were grey, then u would be a descendant of v.
Hence, (u, v) would be a back edge, in contradiction with the fact that the graph is acyclic, and therefore it does not have back edges (cf. Lemma 1).

□

Connected components of an undirected graph

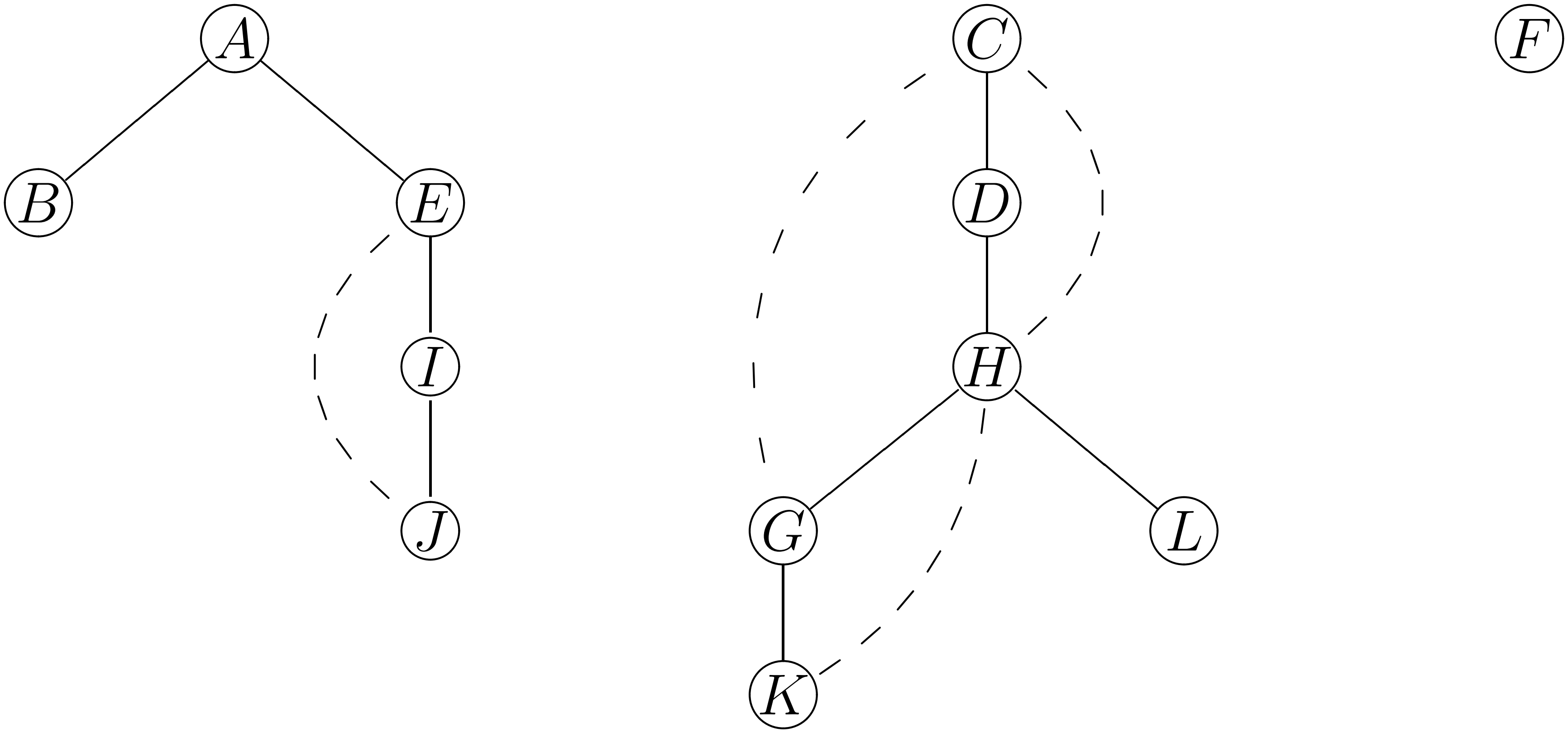
When DFS is run on an undirected graph, the DFS trees identify the connected component of the graph.



	A	B	C	D	E	F	G	H	I	J	K	L
$d[\cdot]$	1	2	11	12	4	23	14	13	5	6	15	18
$f[\cdot]$	10	3	22	21	9	24	17	20	8	7	16	19

DFS forest for an undirected graph

Connected components:



	A	B	C	D	E	F	G	H	I	J	K	L
$d[\cdot]$	1	2	11	12	4	23	14	13	5	6	15	18
$f[\cdot]$	10	3	22	21	9	24	17	20	8	7	16	19

Strongly connected components [CLRS 22.5]

- □ Connectivity in undirected graphs is straightforward: the connected components can be enumerated by DFS (see Exercise 22.3-12 at p. 612 of [CLRS]).

- □ Connectivity for *directed graphs* is more subtle. The right way to define connectivity for directed graphs is:
  Two vertices u and v are ***strongly connected*** if there is a path from u to v and a path from v to u.

Take a directed graph $G = (V, E)$. A ***strongly connected component*** (SCC) of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, there is a path from u to v and there is a path from v to u.

Problem: how to identify the SCCs?

Ingredient 1: structure of the SCCs

Lemma 2. *Let C and C' be distinct SCCs in G, let $u, v \in C$ and $u', v' \in C'$, and suppose there is a path from u to u' in G. Then there cannot also be a path from v' to v in G.*

Proof. For every $x \in C$ and $y \in C'$, there exists a path from x to y, passing through u and u'.
If there existed a path from v' to v, then there would be a path from y to x, passing through v' to v'. Then, C and C' would be the same SCC, in contradiction with the hypothesis that C and C' are distinct SCCs. □

Parenthesis: the SCC graph

The ***SCC graph*** of $G = (V, E)$ is the graph $G^{SCC} = (V^{SCC}, E^{SCC})$ where

□ V^{SCC} has one vertex v_C for each SCC C in G

□ $(v_C, v_{C'}) \in E^{SCC}$ if there is an edge between the corresponding SCCs in G (i.e. there exists $(u, u') \in E$ such that $u \in C$ and $u' \in C'$).

Fact: G^{SCC} is a DAG (by Lemma 2).

Ingredient 2: the finishing times

We extend the notation for $d[\cdot]$ and $f[\cdot]$ to sets of vertices $U \subseteq V$.

□ $d[U] := \min\{\, d[u] : u \in U \,\}$ i.e. *earliest* discovery time among U

□ $f[U] := \max\{\, f[u] : u \in U \,\}$ i.e. *latest* finishing time among U.

We say that ***there is an edge from C to C'*** if there exists an edge (u, u') with $u \in C$ and $u' \in C'$.

Lemma 3. *Let C and C' be distinct SCCs in $G = (V, E)$.*
If there is an edge from C to C', then $f[C] > f[C']$.

Proof: same argument used in the proof of Proposition 1.

Equivalent formulation of Lemma 3: if $C \neq C'$ and $f[C] < f[C']$, then there cannot be an edge from C to C'.

Ingredient 3: the transpose of a graph

Let G be a directed graph. The ***transpose*** of G is $G^T = (V, E^T)$ where

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

We can create G^T in $\Theta(|V| + |E|)$ time using adjacency lists.

Fact. *G and G^T have the same SCCs.*

Obvious (but important) observation:
Let C and C' be distinct SCCs in G.
If $f[C] > f[C']$, then

□ ***in*** G there cannot be an edge from C' to C.

□ ***in*** G^T there cannot be an edge from C to C'.

This means that, if we run DFS on G^T starting from SCC with the *largest* finishing time, we will not find edges to any other SCC.

The algorithm $\mathrm{SCC}(G)$

```
SCC($G$)

    Input: A directed graph $G$.
    Output: Elements of each SCCs of $G$ output in turn.
1   Call DFS($G$) to compute finishing times $f[u]$ for all $u$.
2   Compute $G^T$.
3   Call DFS($G^T$), visiting the vertices
    in order of decreasing $f[u]$ (as computed by the call to DFS in line 1).
4   Output the vertices in each tree of the DFS forest
    formed in second DFS as a separate SCC.
```

Time: $\Theta(|V| + |E|)$

Correctness

In the following, $f[C]$ denotes the finishing time of C *relative to the first DFS*.

Let u be the vertex with maximum finishing time.
By definition, u belongs to the SCC C with maximum $f[C]$.

□ Starting from u, the 2nd DFS visits all vertices in C.

□ Since $f[C] > f[C']$ for all $C' \neq C$,
there are no edges from C to C' in G^T.

Hence, the DFS tree rooted at u contains exactly the vertices of C.

Correctness

In the following, $f[C]$ denotes the finishing time of C *relative to the first DFS*.

Let u be the vertex with maximum finishing time.
By definition, u belongs to the SCC C with maximum $f[C]$.

- □ Starting from u, the 2nd DFS visits all vertices in C.
- □ Since $f[C] > f[C']$ for all $C' \neq C$,
  there are no edges from C to C' in G^T.

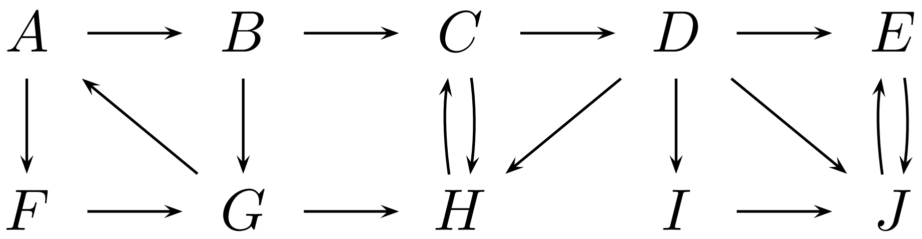
Hence, the DFS tree rooted at u contains exactly the vertices of C.

The next root chosen in the 2nd DFS call is in the SCC C' such that $f[C']$ is maximum in all SCCs *less* C. The 2nd DFS visits all vertices in C'.
C' cannot have edges to other SCCs, except for C.
But C but C has been already visited.
Hence, the only *tree* edges (in the 2nd DFS call) will be to vertices in C'.

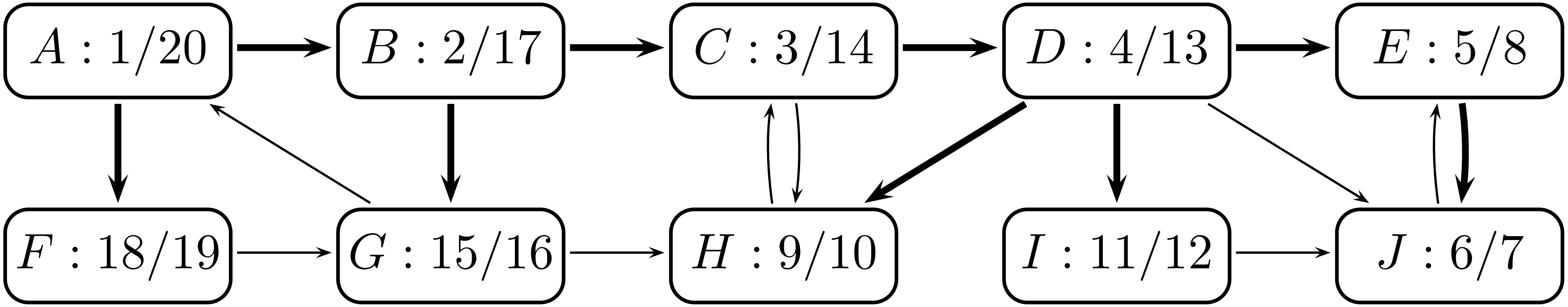
This process continues until all SCCs have been identified.

Example: SCC

Let G be the following graph



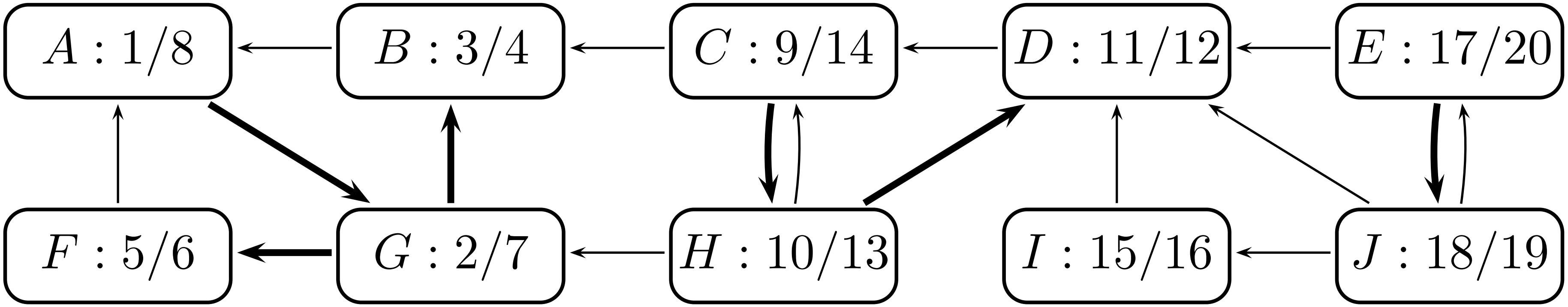
The first DFS produces the following discovery/finishing times:



Therefore, $A, F, B, G, C, D, I, H, E, J$ is the order for the outer loop in the second DFS.

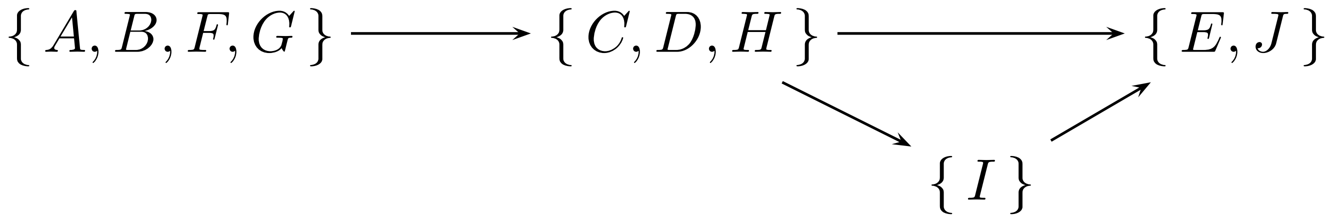
Example: SCC (cont'd)

Using the order $A, F, B, G, C, D, I, H, E, J$ for the outer loop, the second DFS produces:



SCC produces the SCCs as $\{ A, G, B, F \}$, $\{ C, H, D \}$, $\{ I \}$ and $\{ E, J \}$.

Finally the SCC graph G^{SCC} looks like:


$\{ A, B, F, G \}$
$\{ C, D, H \}$
$\{ E, J \}$
$\{ I \}$