Depth-first search

- DFS on undirected graphs
- DFS on directed graphs
- Edge classification
- Topological sort

Last lecture: BFS

Input: $G = (V, E)$, directed or undirected, in adj list format
$s \in V$, source vertex

Output: $d[v]$: distance of v from s, for all $v \in V$
$\pi[v]$: parent/predecessor of v

Time: $O(V + E)$

Note: distance between two vertices u and w: length (# of edges) of a shortest simple (no repeated vertices) path between them

Note: Solves single-source shortest-paths problem

BFS facts:

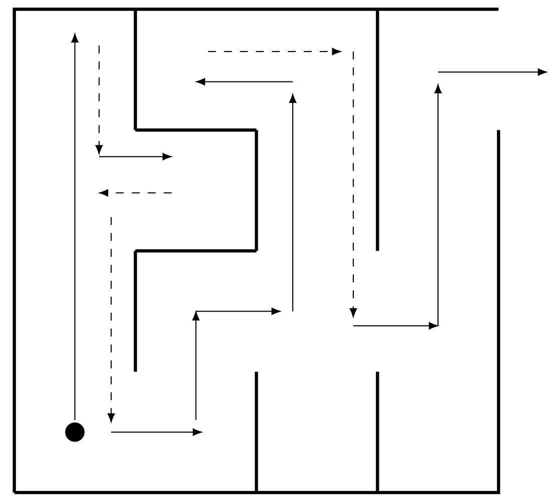
- Finds distances “level-by-level”: fully explore each level before moving to next level 1 edge-length away

- Uses FIFO queue: as process vertices at distance i, add distance $i+1$ vertices to queue to process when done with all distance i vertices (need $O(V)$ space)

- May not reach every vertex

Depth-first search (DFS): like exploring a maze

↑ explore
⇣ backtrack



- follow path as deeply as possible until reach dead end
- backtrack to last unexplored edge; explore it (recursively)
- avoid repeating a vertex previously visited

Code for DFS

DFS(G)// G in Adj list format

```
for each vertex u ∈ V
    color[u] = WHITE
    π[u] = NIL
time = 0 // track time start & finish exploring from vertex
for each vertex u ∈ V
    if color[u] == WHITE
    DFS-VISIT(G, u)
```

DFS-Visit(G, u)

```
time = time + 1 // white vertex u just been discovered
d[u] = time
color[u] = GRAY
for each v ∈ Adj[u] // explore edge (u, v)
    if color[v] == WHITE
        π[v] = u
        DFS-Visit(G, v)
color[u] = BLACK // vertex u finished
time = time + 1
f[u] = time
```

DFS: Explore entire graph

Input: $G = (V, E)$, directed or undirected, in adj list format

Output: $d[v]$: discovery time of vertex v
$f[v]$: finish time of v
$\pi[v]$: parent/predecessor of v

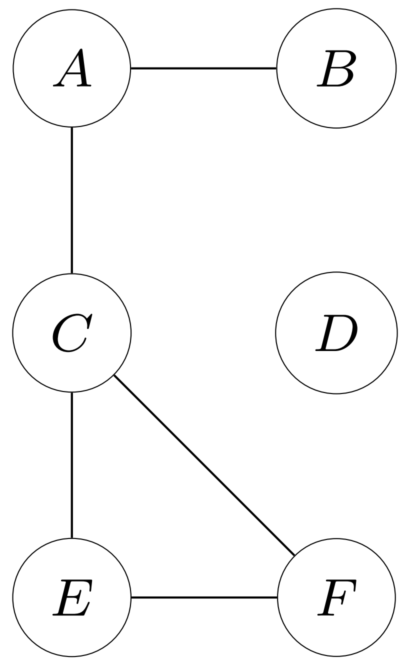
Note: maintain variable $color[v]$ indicating whether already visited v

Time: $\Theta(V + E)$

Runtime Analysis:

- DFS-Visit is called once per vertex v (because afterward $color[v]$ is no longer White)
- Adjacency list of v scanned only once (in that call)
- $\Rightarrow$ Time in DFS-Visit $= \sum_{v \in V} |Adj[v]| = O(E)$
- DFS outer loop adds just $O(V)$
- $\Rightarrow O(V + E)$: linear time

Example (undirected graph):



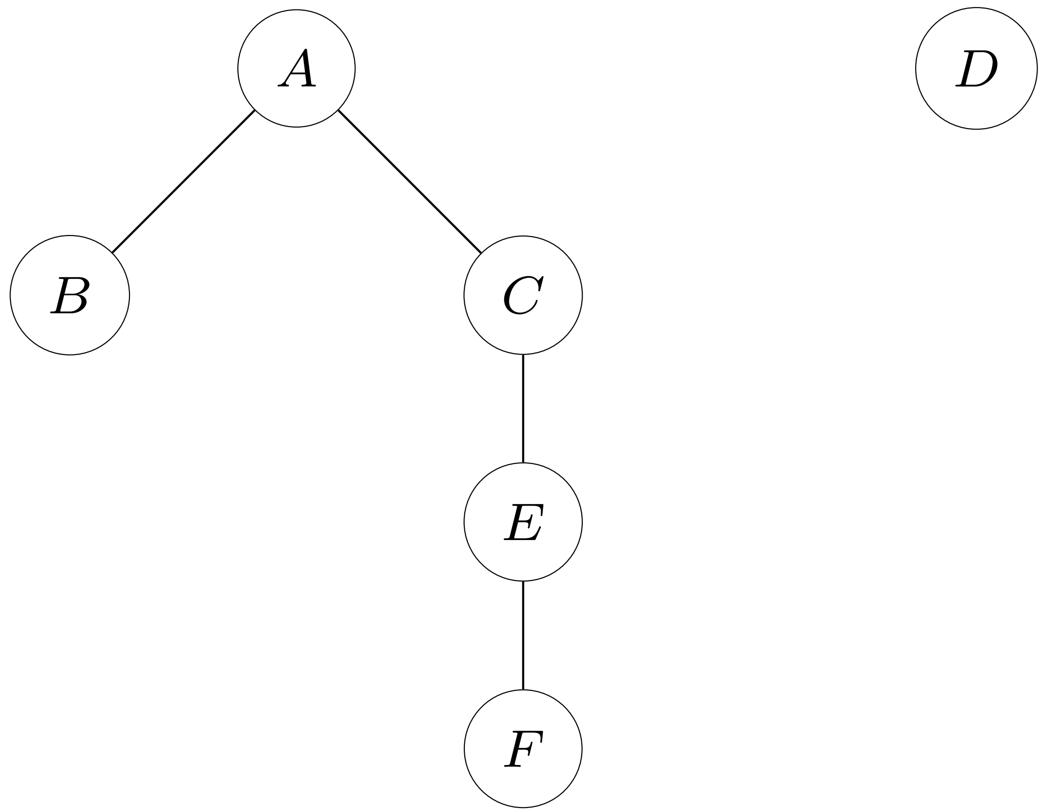
Output:

	A	B	C	D	E	F
d	1	2	4	11	5	6
f	10	3	9	12	8	7
π	N	A	A	N	C	E

Outer loop of DFS calls DFS-Visit twice, on A and D: As a result, there are two trees, each rooted at one of these starting points; together they constitute a *forest*

DFS forest:

Two trees generated: one rooted at A, one at D



Not all edges in input graph G are in DFS forest:

Tree edges	Non-tree edges
AB, AC, CE, EF	CF

Non-tree edges are called *back edges*: they lead back to vertices already visited

Tree edge: vertex v WHITE when edge (u, v) explored 1st time

Back edge: vertex v GRAY when edge (u, v) explored 1st time

Cycles in undirected graphs:

A *cycle* is a circular path $v_0, v_1, \ldots, v_k, v_0$

Is there a cycle in example graph? $C - E - F - C$

Presence of back edge indicates existence of cycle

HW: modify DFS to detect a cycle in an undirected graph

Connected components in undirected graphs:

Def: An undirected graph is connected if there is a path between every pair of vertices in the graph

Our example graph is NOT connected: no path from D to any other vertex

It has two disjoint connected regions, corresponding to the sets of vertices:

$$\{A, B, C, E, F\} \quad \{D\}$$

These regions are called *connected components*, subgraphs that are internally connected but have no edges to remaining vertices

When DFS-Visit is started at a particular vertex, it identifies precisely the connected component containing that vertex

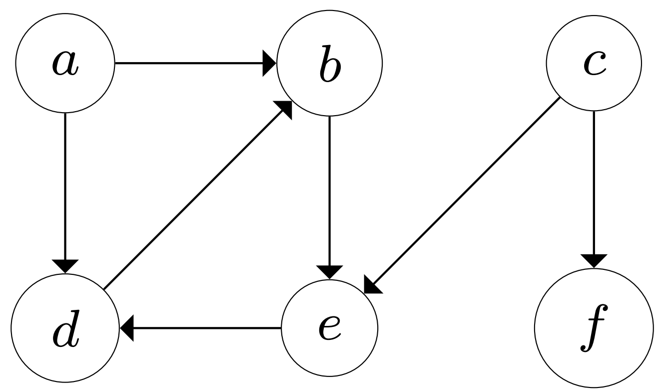
Each time DFS (line 5) calls DFS-Visit, a new connected component is identified

"Do" exercise: modify DFS to identify the connected components of an undirected graph

DFS in directed graphs

DFS can be run verbatim on directed graphs

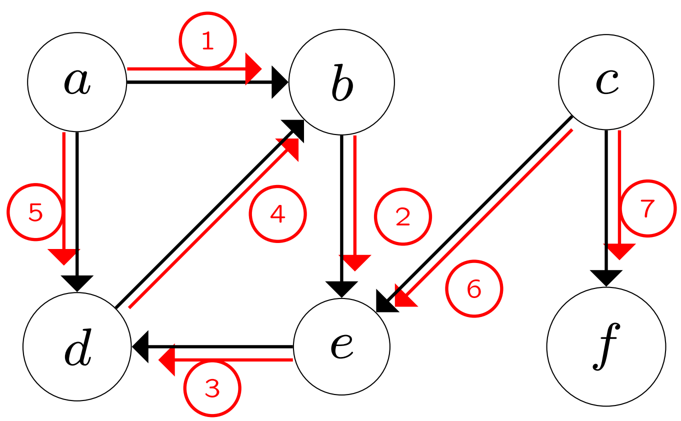
Example:



DFS output:

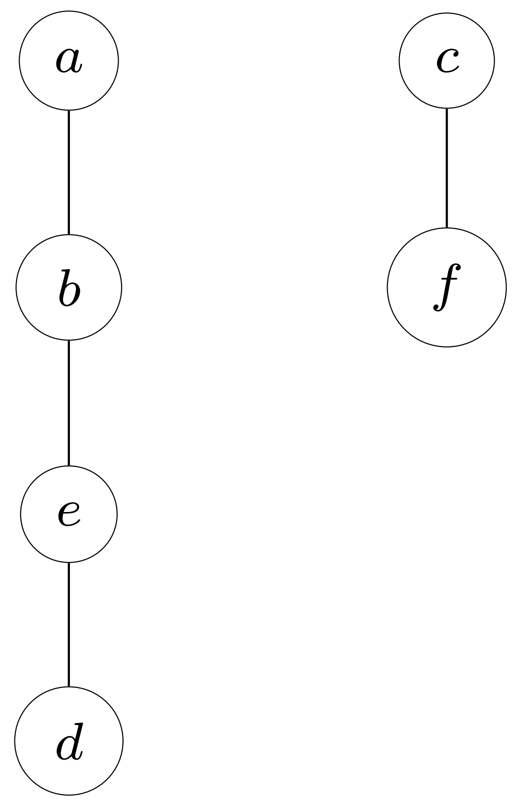
	a	b	c	d	e	f
d	1	2	9	4	3	10
f	8	7	12	5	6	11
π	N	a	N	e	b	c

Circled numbers (not part of output) indicate order in which edges are processed:



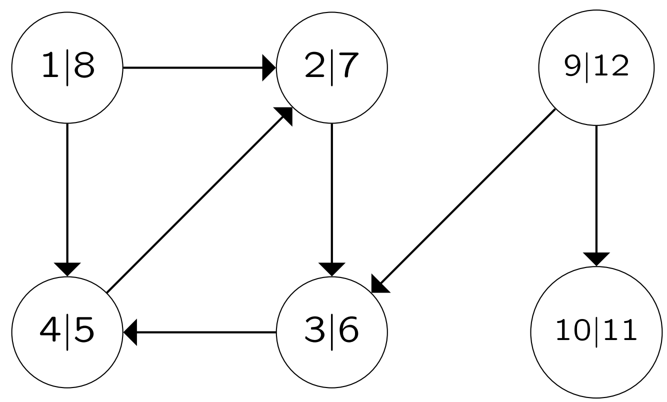
DFS forest:

Two trees generated: one rooted at a, one at c



Edge classification:

- idea: for each vertex keep track of
  - time vertex 1st visited | time vertex completed



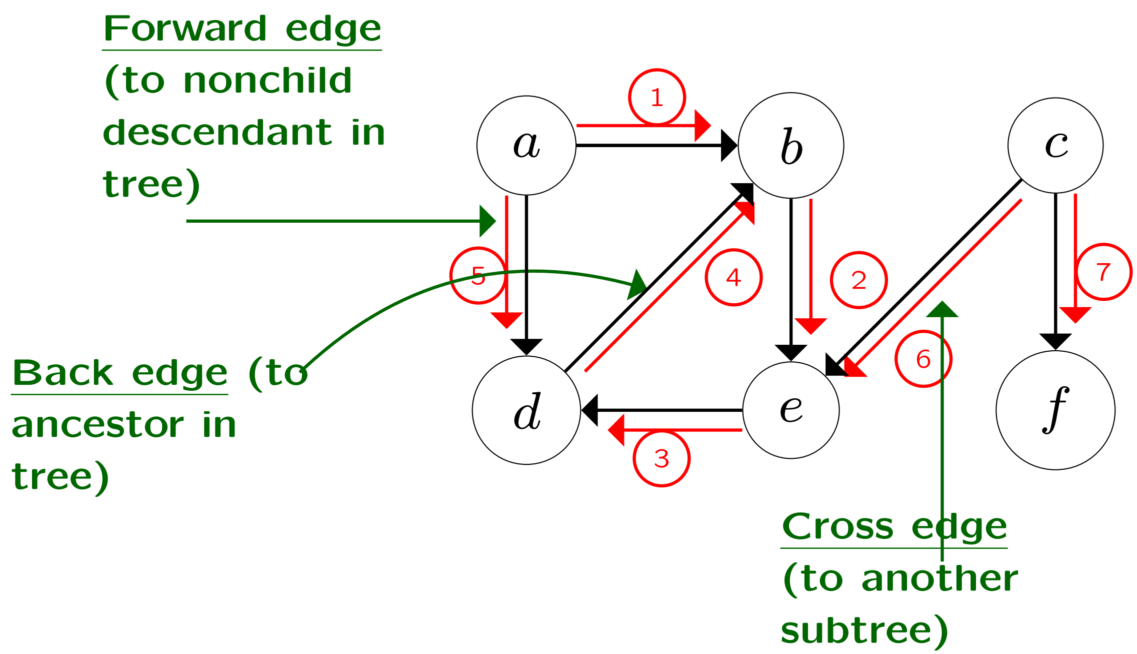
- parenthesis structure: open paren at start, close paren at finish:

1	2	3	4	5	6	7	8	9	10	11	12
(	(	(	(	)	)	)	)	(	(	)	)

- vertex w inside vertex u means w is descendant of u on DFS tree
- u disjoint w means neither ancestor nor descendant: different subtrees

Edge types:

- Tree edges: (leads to new child in DFS tree)
- Back edges: (leads to ancestor in DFS tree)
- Forward edges: (leads to nonchild descendant in DFS tree)
- Cross edges: (leads to neither ancestor nor descendant)

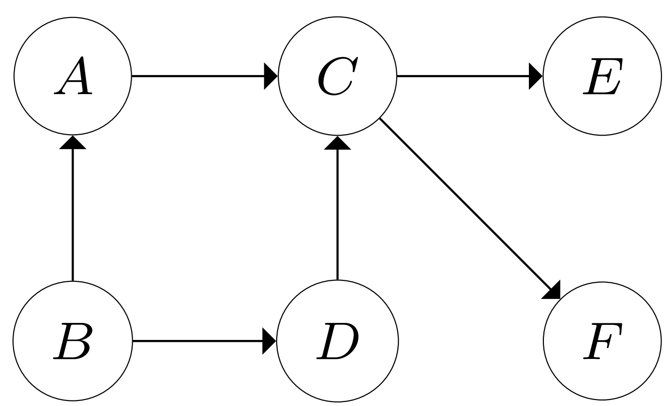


- Tree edges: 1, 2, 3, 7
- Back edge: 4
- Forward edge: 5
- Cross edge: 6

Directed acyclic graphs:

Def: Let $G = (V, E)$ be a directed graph. G is acyclic if it contains no cycles. (DAG, for "directed acyclic graph").

Example:



Theorem: G is acyclic $\Leftrightarrow$ DFS of G produces no back edges.

Proof: ($\Rightarrow$): if there is a back edge: $u \to$ ancestor of u, a cycle is created

($\Leftarrow$): say a vertex "finishes" when its DFS-Visit call terminates

Lemma: in DFS, all edges (u, v) that are not back edges have property: v finishes before u

- tree edge (DFS-Visit(u) calls DFS-Visit(v))
- forward edge (DFS-Visit(v) already done)
- cross edge (DFS-Visit(v) already done)

$\therefore$ there are no cycles, since following a path must yield earlier and earlier finishing times q.e.d

Claim: Can determine in $O(V + E)$ time whether a directed graph is acyclic

Proof: Run DFS, see if any back edges are produced

Topological Sort

If events require that some occur before others, we can represent these dependencies with a directed graph (earlier $\rightarrow$ later)

If the directed graph is acyclic, then it can be "topologically sorted" to produce an ordering of events consistent with dependencies (constraints)

Given: a directed acyclic graph $G = (V, E)$

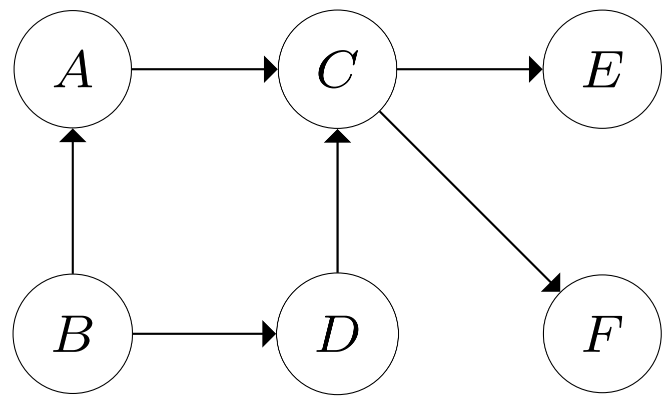
Output: A list of its vertices $v_1, v_2, \ldots, v_n$ in some order such that if G contains edge (u, v), then u appears before v in the ordering

How do it?

No cycles in G, so all edges (u, v) have decreasing (high to low) finish times $f[u] > f[v]$ by Lemma just proved

<u>Idea</u>: Order vertices by decreasing finish time

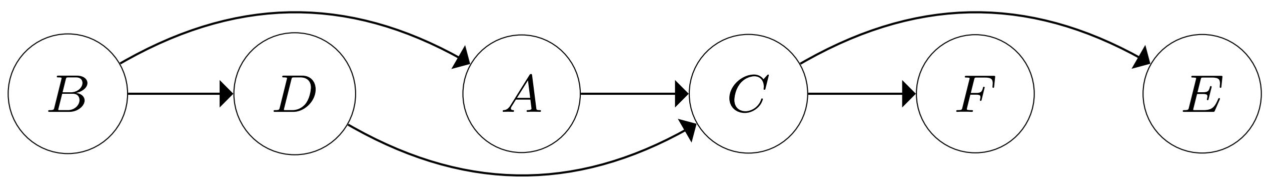
Run DFS on Example graph:



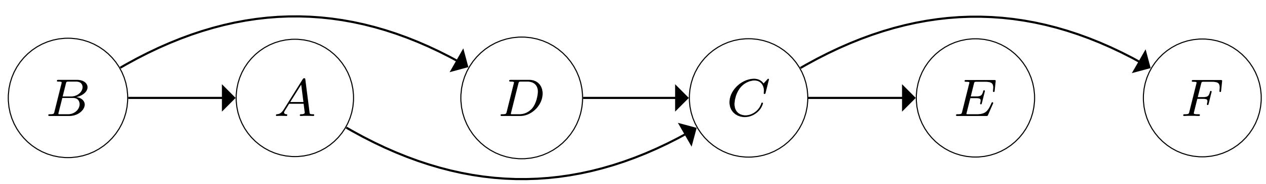
Output:

	A	B	C	D	E	F
d	1	9	2	10	3	5
f	8	12	7	11	4	6
π	N	N	A	B	C	C

Topological ordering:



There are other topological orderings of G, e.g.:



Topological-Sort(G)// G in Adj list format

$L = \emptyset$

DFS(G) to compute $f[v]$ for all $v \in V$

when vertex v is finished, append it to L

$L' =$ reverse(L)

return L'

Runtime Analysis:

$O(V + E)$ time to topological sort a DAG

Correctness:

NTS: If $(u, v) \in E$, then $f[v] < f[u]$

Proof: When explore (u, v) what are colors of u and v?

- u is Gray
  - Is v Gray? No, b/c then (u, v) would be a back edge
    By previous theorem, not possible
  - Is v White? If so, $d[u] < d[v] < \underline{f[v] < f[u]}$
  - Is v Black? If so, v is already finished
    Since exploring (u, v), u is not yet finished
    $\therefore \underline{f[v] < f[u]}$ q.e.d