Week 10: Graph Algorithms

Agenda:

- Graph traversal — Depth-first search
- DFS application:
  - Finding biconnected components
  - Strongly Connected components
  - Topological sorting

Reading:

- Textbook pages 540 –559

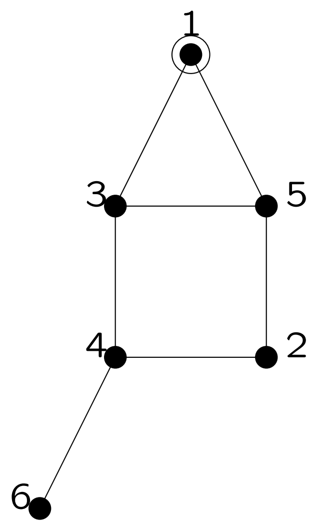
Depth First Search (DFS):

- Input: simple undirected graph $G = (V, E)$
- Output: all vertices discovered (pick one vertex from each component as the start vertex)
- Idea: to search deeper in the graph whenever possible ...
- Pseudocode (recursive version):

```
procedure DFS(G)            **G = (V, E)

for each v ∈ V do
    c[v] ← WHITE            **unknown yet
    p[v] ← NIL              **predecessor
time ← 0
for each v ∈ V do
    if c[v] = WHITE then
        DFS-visit(v)

procedure DFS-visit(v)      **any v ∈ V

c[v] ← GRAY                 **start discovering v
time ← time + 1
dtime[v] ← time
for each u ∈ Adj[v] do
    if c[u] = WHITE then
        p[u] ← v
        DFS-visit(u)
c[v] ← BLACK                **finished discovering
time ← time + 1
ftime[v] ← time
```

DFS example:

- $V = \{1, 2, 3, 4, 5, 6\}$
  $E = \{\{1,3\}, \{1,5\}, \{2,4\}, \{2,5\}, \{3,4\}, \{3,5\}, \{4,6\}\}$
  $s = 2$



Adjacency lists:

1:	3	5	
2:	4	5	
3:	1	4	5
4:	2	3	6
5:	1	2	3
6:	4		

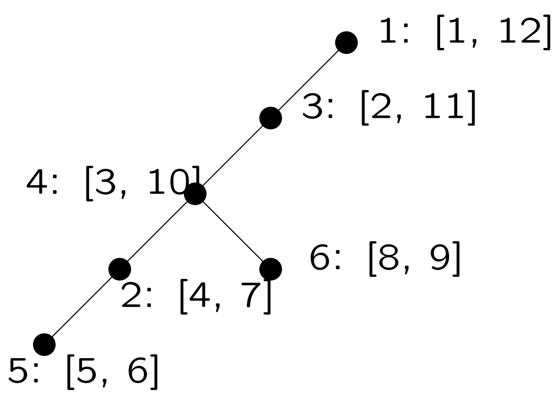
	1	2	3	4	5	6	DFS-visit path
color	W	W	W	W	W	W	
parent	NIL	NIL	NIL	NIL	NIL	NIL	
dtime	∞	∞	∞	∞	∞	∞	`initialization`
ftime	∞	∞	∞	∞	∞	∞	
color	G	W	W	W	W	W	
parent	NIL	NIL	NIL	NIL	NIL	NIL	
dtime	1	∞	∞	∞	∞	∞	`DFS-visit(1)`
ftime	∞	∞	∞	∞	∞	∞	
color	G	W	G	W	W	W	
parent	NIL	NIL	1	NIL	NIL	NIL	
dtime	1	∞	2	∞	∞	∞	`DFS-visit(1-3)`
ftime	∞	∞	∞	∞	∞	∞	
color	G	W	G	G	W	W	
parent	NIL	NIL	1	3	NIL	NIL	
dtime	1	∞	2	3	∞	∞	`DFS-visit(1-3-4)`
ftime	∞	∞	∞	∞	∞	∞	
color	G	G	G	G	W	W	
parent	NIL	4	1	3	NIL	NIL	
dtime	1	4	2	3	∞	∞	`DFS-visit(1-3-4-2)`
ftime	∞	∞	∞	∞	∞	∞	
color	G	G	G	G	G	W	
parent	NIL	4	1	3	2	NIL	
dtime	1	4	2	3	5	∞	`DFS-visit(1-3-4-2-5)`
ftime	∞	∞	∞	∞	∞	∞	
color	G	G	G	G	B	W	
parent	NIL	4	1	3	2	NIL	
dtime	1	4	2	3	5	∞	`DFS-visit(1-3-4-2-5)`
ftime	∞	∞	∞	∞	6	∞	
color	G	B	G	G	B	W	
parent	NIL	4	1	3	2	NIL	
dtime	1	4	2	3	5	∞	`DFS-visit(1-3-4-2)`
ftime	∞	7	∞	∞	6	∞	
color	G	B	G	G	B	G	
parent	NIL	4	1	3	2	4	
dtime	1	4	2	3	5	8	`DFS-visit(1-3-4-6)`
ftime	∞	7	∞	∞	6	∞	
color	G	B	G	G	B	B	
parent	NIL	4	1	3	2	4	
dtime	1	4	2	3	5	8	`DFS-visit(1-3-4-6)`
ftime	∞	7	∞	∞	6	9	
color	G	B	G	B	B	B	
parent	NIL	4	1	3	2	4	
dtime	1	4	2	3	5	8	`DFS-visit(1-3-4)`
ftime	∞	7	∞	10	6	9	
color	G	B	B	B	B	B	
parent	NIL	4	1	3	2	4	
dtime	1	4	2	3	5	8	`DFS-visit(1-3)`
ftime	∞	7	11	10	6	9	
color	B	B	B	B	B	B	
parent	NIL	4	1	3	2	4	
dtime	1	4	2	3	5	8	`DFS-visit(1)`
ftime	12	7	11	10	6	9	

DFS example:

- Adjacency lists:

1:	3	5	
2:	4	5	
3:	1	4	5
4:	2	3	6
5:	1	2	3
6:	4		

- DFS tree: [`dtime`,`ftime`]



Notes:

- the result would be a forest of rooted trees
- the root of each tree is up to the selection (ordering of the vertices)
- parent of x is predecessor $p[x]$
- different orderings of adjacency lists might result in different trees
- nested structure of [`dtime, ftime`] — they don't intersect each other

DFS analysis:

- $n = |V|$, $m = |E|$
- Handshaking Lemma: $\sum_{v \in V} \texttt{degree}(v) = 2m$
- Analysis:
  - each vertex is discovered exactly once (WHITE $\rightarrow$ GRAY $\rightarrow$ BLACK)

    each edge is examined exactly twice
  - running time:
    1. adjacency list representation:
       $\Theta(n + 2m) = \Theta(n + m)$
    2. adjacency matrix representation:
       $\Theta(n + n^2) = \Theta(n^2)$
  - space complexity:
    1. adjacency list representation:
       $\Theta(n + m)$
    2. adjacency matrix representation:
       $\Theta(n^2)$

Classifying graph edges with BFS/DFS:

- During the traversal, all vertices and edges are examined
- Given a BFS/DFS traversal forest:
  - tree root — start vertex for that component
  - tree edge — child discovered while processing the parent
  - each edge in the original graph is examined twice
- Question:

  Where are the other possible edges, besides tree edges ???
- Answer:

  With respect to the traversal forest, categorize graph edges by their first time encounter:
  - tree edges
  - back edges: to ancestor
  - forward edges: to descendant
  - cross edges: to non-ancestor, non-descendant

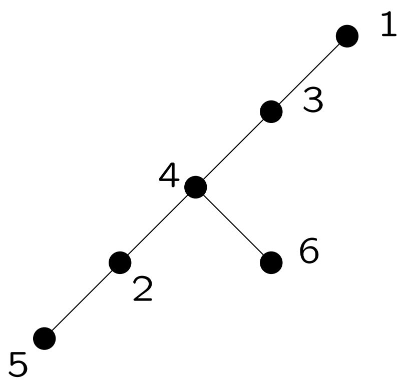
  Note: in undirected graphs, "back" = "forward"
- Examples:

An example:

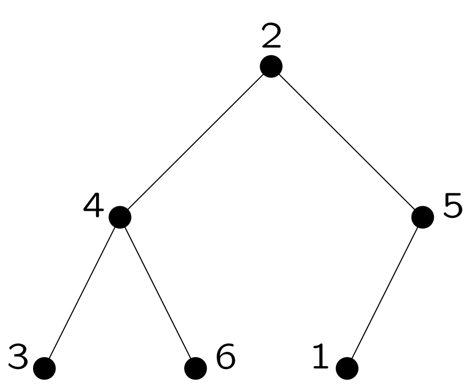
- Adjacency lists:

1:	3	5	
2:	4	5	
3:	1	4	5
4:	2	3	6
5:	1	2	3
6:	4		

- DFS tree (start vertex 1):



- BFS Tree (start vertex 2):

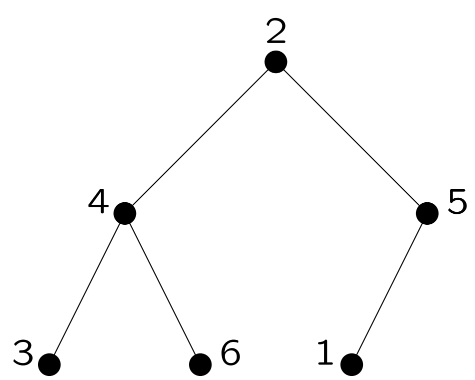


Properties of BFS/DFS:

- BFS:
  - each graph edge connects two vertices with level-difference ≤ 1
    Proof.
  - no back / forward edges
- DFS:
  - each non-tree edge is a back edge
    Proof.
  - no forward edges (if G is undirected)
  - no cross edges
  - vertex processing time intervals $[\texttt{dtime}[v],\ \texttt{ftime}[v]]$ and $[\texttt{dtime}[w],\ \texttt{ftime}[w]]$:
    $[\texttt{dtime}[v],\ \texttt{ftime}[v]] \subset [\texttt{dtime}[w],\ \texttt{ftime}[w]]$ — v is a descendant of w in the DFS forest
    $[\texttt{dtime}[v],\ \texttt{ftime}[v]] \cap [\texttt{dtime}[w],\ \texttt{ftime}[w]] = \emptyset$ — no ancestor-descendant relationship between v and w
- BFS vertex order:
  level-order of each tree in the BFS forest
- DFS vertex order:
  pre-order of each tree in the DFS forest
- Some other vertex order associated with rooted trees:
  - in-order (for binary trees only)
  - post-order

Vertex order with respect to a binary rooted tree:

- Tree:



- Vertex orders:
  - level-order: level by level (each level: left to right)
    $(2, 4, 5, 3, 6, 1)$
  - pre-order: parent - child one - child two - . . . - last child
    $(2, 4, 3, 6, 5, 1)$
  - in-order: left child - parent - right child
    $(3, 4, 6, 2, 1, 5)$
  - post-order: child one - child two - . . . - last child - parent
    $(3, 6, 4, 1, 5, 2)$

Biconnected component:

- Definition — every pair of vertices are connected by two vertex-disjoint paths
- Cut vertex — its removal increases the number of connected components
- Fact: biconnected $\Longleftrightarrow$ no cut vertices
- Biconnected component $\Longleftrightarrow$ maximal connected subgraph containing no cut vertex
- In a DFS tree:
  - root is a cut vertex **iff** it has ≥ 2 child vertices (Why ???)

    $\longrightarrow$ One simplest implementation (assuming connected):

    1. try every vertex v as the start vertex and do the DFS
    2. in the DFS tree, if $\texttt{degree}_{DFS}(v) > 1$, decompose the graph accordingly into $\texttt{degree}_{DFS}(v)$ subgraphs sharing one common vertex v
    3. repeat on subgraphs until for every subgraph the DFS tree with every possible start vertex has root degree 1

    Problem: too time consuming $\Theta(n(n+m))$ ...
  - any other vertex is a cut vertex **iff** vertices in the child subtrees have no back edges to its proper ancestors

    $\longrightarrow$ Idea in the improved implementation — $(\Theta(n+m))$: for each vertex v, and each of its child w, keep track of furthest back edge from the w-subtree

DFS application: finding biconnected components

- Idea in the improved implementation — ($\Theta(n+m)$):

  for each vertex v, and each of its child w, keep track of furthest back edge from the w-subtree

- Details:
  - for every vertex v, 1st encounter child w, recur from w
  - last encounter w (just before backing up to v), check whether v cuts off the w-subtree (rooted at w)
  - maintain `dtime`$[v]$, $b[v]$, $p[v]$ for v:
    1. `dtime`$[v]$ — discovery time
    2. $b[v]$ — `dtime` of the furthest ancestor of v to which there is back edge from a descendant w of v
       (a) updated when the first back edge is encountered
       (b) updated when last time encounter of v (backing up)
    3. $p[v]$ — parent of v in the DFS tree

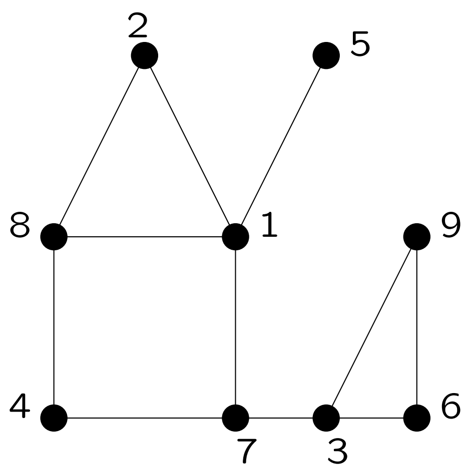
- Reporting biconnected components:
  - recall that biconnected components form a partition of edge set E
  - when edge e first encountered, push into edge stack
  - when a cut vertex discovered, pop necessary edges

Finding biconnected components — pseudocode:

```
procedure bicomponents(G)           **G = (V, E)

S = ∅                               **S is the edge stack
time ← 0
for each v ∈ V do
   p[v] ← 0                         **unknown yet:  NIL
   dtime[v] ← time
   b[v] ← n + 1
for each v ∈ V do
   if dtime[v] = 0 then
      biDFS(v)

procedure biDFS(v)                  **discover v

time ← time +1
dtime[v] ← time
b[v] ← dtime[v]                     **no back edge from descendant yet
for each neighbor w of v do         **first time encounter w
   if dtime[w] = 0 then             **unknown yet
      push(v, w)
      p[w] ← v
      biDFS(w)                      **recursive call
      if b[w] ≥ dtime[v] then
         print ‘‘new biconnected component’’
         repeat
            pop & print
         until (popped edge is (v, w))
      else
         b[v] ← min{b[v], b[w]}
   else if (dtime[w] < dtime[v] and w ≠ p[v]) then
                                    **(v, w) is a back edge from v
      push(v, w)
      b[v] ← min{b[v], dtime[w]}
```

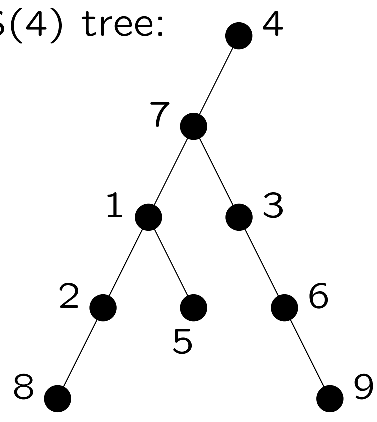
Finding biconnected components — example:

Execute `biDFS(4)` on the following graph, assuming no previous `biDFS()` calls:



1: 2 5 7 8
2: 1 8
3: 6 7 9
4: 7 8
5: 1
6: 3 9
7: 1 3 4
8: 1 2 4
9: 3 6

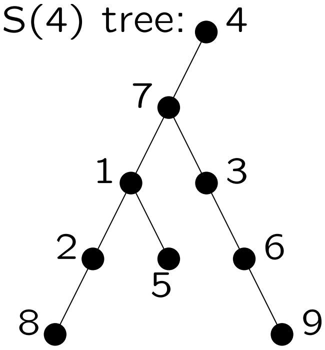
DFS(4) tree:



Finding biconnected components — answer:

1: 2 5 7 8
2: 1 8
3: 6 7 9
4: 7 8
5: 1
6: 3 9
7: 1 3 4
8: 1 2 4
9: 3 6

DFS(4) tree:



dtime	3	4	7	1	6	8	2	5	9
	$b[1]$	$b[2]$	$b[3]$	$b[4]$	$b[5]$	$b[6]$	$b[7]$	$b[8]$	$b[9]$
biDFS(4)	10	10	10	1	10	10	10	10	10
4} biDFS(7)	10	10	10	1	10	10	2	10	10
4, 7} biDFS(1)	3	10	10	1	10	10	2	10	10
4, 7, 1} biDFS(2)	3	4	10	1	10	10	2	10	10
4, 7, 1, 2} (2,1)									
4, 7, 1, 2} biDFS(8)	3	4	10	1	10	10	2	5	10
4, 7, 1, 2, 8} (8,1)	3	4	10	1	10	10	2	*3*	10
4, 7, 1, 2, 8} (8,2)									
4, 7, 1, 2, 8} (8,4)	3	4	10	1	10	10	2	*1*	10
4, 7, 1, 2} biDFS(8) done	3	*1*	10	1	10	10	2	1	10
4, 7, 1} biDFS(2) done	*1*	1	10	1	10	10	2	1	10
4, 7, 1} biDFS(5)	1	1	10	1	6	10	2	1	10
4, 7, 1, 5} (5,1)									
4, 7, 1} biDFS(5) done	new biconnected component: (1, 5)								
4, 7, 1} (1,7)									
4, 7, 1} (1,8)									
4, 7} biDFS(1) done	1	1	10	1	6	10	*1*	1	10
4, 7} biDFS(3)	1	1	7	1	6	10	1	1	10
4, 7, 3} biDFS(6)	1	1	7	1	6	8	1	1	10
4, 7, 3, 6} (6,3)									
4, 7, 3, 6} biDFS(9)	1	1	7	1	6	8	1	1	9
4, 7, 3, 6, 9} (9,3)	1	1	7	1	6	8	1	1	*7*
4, 7, 3, 6, 9} (9,6)									
4, 7, 3, 6} biDFS(9) done	1	1	7	1	6	*7*	1	1	7
4, 7, 3} biDFS(6) done	new biconnected component: (9, 3), (6, 9), (3, 6)								
4, 7, 3} (3,7)									
4, 7, 3} (3,9)									
4, 7} biDFS(3) done	new biconnected component: (7, 3)								
4, 7} (7,4)									
4} biDFS(7) done	new biconnected component: (8, 4), (8, 1), (2, 8), (1, 2), (7, 1), (4, 7)								
biDFS(4) done	1	1	7	1	6	7	1	1	7

Finding biconnected components — analysis:

- Correctness ???

- Complexity — running time and space requirement:
  - running time:
    constant for each vertex encounter and each edge encounter $\longrightarrow$
    $\Theta(c_1 n + c_2 \sum_{v \in V} \texttt{degree}(v)) = \Theta(n+m)$
  - space:
    assume adjacency list representation: space for graph, arrays of size n, edge stack, and runtime stack
    1. space for graph and arrays $\Theta(m+n)$
    2. edge stack requires $O(m)$ — since every edge pushed
    3. runtime stack $O(n)$ — since at most n `biDFS` activations each is of constant size
    4. therefore, $\Theta(n+m)$ in total

Comparing DFS and BFS:

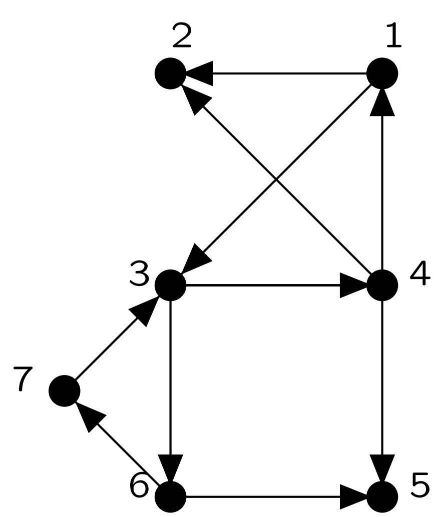
- BFS works well for finding shortest path
- All non-tree edges in
  - BFS are cross edges
  - DFS are back edges

Directed graphs:

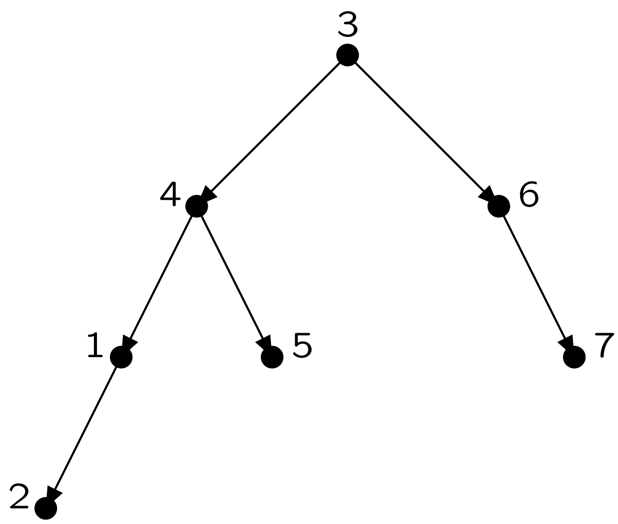
- Recall that in a directed graph every edge is directed (i.e. it is an ordered pair)
- We say u reaches v if there is a directed path from u to v
- Strongly connected digraph: A digraph G is strongly connected if for every pair u, v of vertices u is reachable from v and v is reachable from u
- The notion of a directed cycle is defined similarly.
- Directed Acyclic Graph (DAG): A digraph with no (di)cycles.

Traversing Directed graphs:

- DFS and BFS can be adapted to work on directed graphs.
- The only difference is that we travel edges according to their direction.
- Every edge that is discovered is a "tree-edge"
- In a DFS, back-edges may exist (from a node to one of its ancestors)
- We may also have a "forward-edge": a non-tree edge from a node to one of its descendant:
- Example: $V = \{1, 2, 3, 4, 5, 6, 7\}$
  $E = \{(1, 2), (1, 3), (3, 4), (3, 6), (4, 1), (4, 2), (4, 5), (6, 5), (6, 7), (7, 3)\}$



- Then calling DFS(3) gives:



- edges $(1,3)$ and $(7,3)$ are back edges and $(4,2)$ is a forward edge.

- If we call DFS(v) in a digraph, we visit all vertices that are reachable from v in G. The DFS tree contains directed paths from v to every such vertex.

- How to check if G is strongly connected?

- Run DFS from every v. If every tree visit all the vertices then it is strongly connected.

- Time: $\Theta(n \times (n+m))$.

- Do we really need that many calls to DFS? or can we do better?

Strongly connected

- First run DFS from some vertex v
- If it does not reach some vertex then return “No”
- Else (all vertices are reachable from v) reverse the directions of all edges.
- Run DFS again from v. If all vertices (in this new graph) are reachable then G is strongly connected because every vertex has a directed path “to” and “from” v in G.

  So every vertex is reachable from every other one via v.
- Time: $\Theta(n + m)$.

Topological ordering in DAG’s

- Suppose we have a set of tasks to be performed
- For each task we have a requirement that some of the other tasks must be done before we can perform this.
- This requirement is given as a directed graph G which is DAG (directed acyclic).
- If $(u, v) \in E$ it means we must perform u before we can perform v.
- Goal: find an ordering of the tasks (vertices of G) such that for each task all its requirements appear earlier in that ordering,

- i.e. find an ordering $v_1, \ldots, v_n$ of vertices of G such that for every edge (v_i, v_j), $i < j$. This is called a "topological soring"

- Theorem: A digraph has a topological soring if and only if it is acyclic.

- Clearly if we have a cycle we cannot have a topological ordering (why?)

- Now suppose that G is a DAG.

- We prove the theorem by induction on n. Base case $n = 1$ is trivial (any ordering will do).

- So assume that $n \geq 2$. There is a vertex in G which has no ingoing edges or else G has a cycle (why?)

- Say $in-degree(u) = 0$. Remove v from G, call the new graph G' (which has $n - 1$ vertices).

- G' is acyclic so by I.H. has a topological ordering $v_2, \ldots, v_n$.

- Since u has only outgoing edges, $u, v_2, \ldots, v_n$ is a topological ordering of G.

```
procedure Topological-Sort(G)
    S ← ∅
    for each v ∈ V do
    if in − degree(v) = 0 then
        S.push(v)
    i ← 1
    While S ≠ ∅ do
        v ← S.pop()
        output v
        i ← i + 1
        for each vu do
            Remove uv (so decrease in − degree(u))
            if in − degree(u) = 0 then
                S.push(u)
    if i < n then
        return ``G has a cycle''
```