Unit 10: Depth First Search

Agenda:

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

Reading:

- CLRS: 603-621

Depth First Search (DFS):

- Input: graph $G = (V, E)$
- Idea: search deeper in the graph whenever possible ...
- <u>procedure DFS(G)</u> **$G = (V, E)$

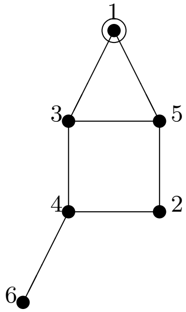
```
foreach v ∈ V do
    v.color ← WHITE          **unknown yet
    v.predec ← NIL           **predecessor
time ← 0                     **global variable
foreach v ∈ V do
    if (v.color = WHITE) then
        DFS-visit(G, v)
```

<u>procedure DFS-visit(G, s)</u> **any $s \in V$

```
s.color ← GRAY               **start discovering s
time ← time + 1
s.dtime ← time
foreach u neighbor of s do
    if (u.color = WHITE) then
        u.predec ← s
        DFS-visit(G, u)
s.color ← BLACK              **finished discovering
time ← time + 1
s.ftime ← time
```

DFS example:

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



Adjacency lists:

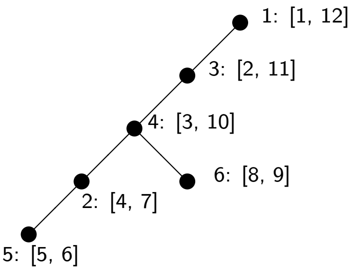
1:	3	5	
2:	4	5	
3:	1	4	5
4:	2	3	6
5:	1	2	3
6:	4		

	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	∞	

	1	2	3	4	5	6	DFS-visit path
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:

- DFS tree: [dtime, ftime]



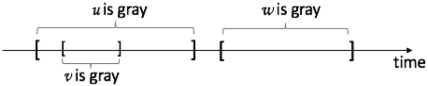
- Notes:
  - the result would be a forest of rooted trees
  - the root of each tree is up to the selection (ordering of the vertices)
  - parent of x is predecessor $x.predec$
  - different orderings of adjacency lists might result in different trees
  - **Nested structure of [dtime, ftime]**
    — u is a descendant of $v \Rightarrow [u.\texttt{dtime},\ u.\texttt{ftime}] \subset [v.\texttt{dtime},\ v.\texttt{ftime}]$
    — u & v on different branches $\Rightarrow [u.\texttt{dtime},\ u.\texttt{ftime}]$ doesn't intersect $[v.\texttt{dtime},\ v.\texttt{ftime}]$

DFS analysis:

- $n = |V|$, $m = |E|$
- Handshaking Lemma: $\sum_{v \in V} \deg(v) = 2m$
- Analysis:
  - each vertex is discovered exactly once (WHITE $\rightarrow$ GRAY $\rightarrow$ BLACK)
    in an undirected graph: each edge is examined exactly twice
    in a directed graph: each edge is examined once
  - 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:
       $\Theta(n^2)$

Properties of DFS:

- The Parentheses Theorem:
  two vertex processing time intervals $[\mathtt{dtime}[v],\ \mathtt{ftime}[v]]$ and $[\mathtt{dtime}[w],\ \mathtt{ftime}[w]]$ can only have one of the following two applied to them: contained or disjoint.
  I.e. we either have (i) $[\mathtt{dtime}[v],\ \mathtt{ftime}[v]] \subset [\mathtt{dtime}[w],\ \mathtt{ftime}[w]]$ — v is a descendant of w in the DFS forest (or vice-versa)
  or we have (ii) $[\mathtt{dtime}[v],\ \mathtt{ftime}[v]] \cap [\mathtt{dtime}[w],\ \mathtt{ftime}[w]] = \emptyset$ — no ancestor-descendant relationship between v and w



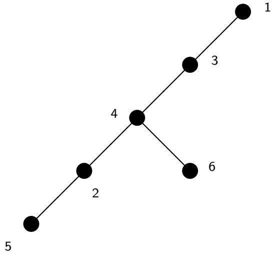
- The White-Path Theorem:
  v is a descendant of u iff at time $u.dtime$ there was a path $u \to v$ along which all vertices are white (except for u).
  - An all gray path at time $v.dtime$
  - and all black path at time $u.ftime$.
- DFS vertex order:
  dtime: pre-order of each tree in the DFS forest
  ftime: post-order of each tree in the DFS forest
- (BFS vertex order:
  level-order of each tree in the BFS forest)

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
  - (undirected) each edge in the original graph is examined twice
    (digraph) each edge in the original digraph is examined once
- With respect to the traversal forest, categorize edges into 4 types.
  An edge $e = (u, v)$ is a
  1. Tree edge: the edge (u, v) is in the forest
  2. Forward edge: v is a descendant of u
  3. Back edge: v is an ancestor of u
     Note: in undirected graphs, "back" $=$ "forward"
  4. Cross edge: v is a non-ancestor and non-descendant of u

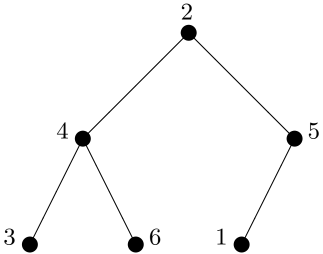
An example:

- DFS tree (start vertex 1):
-



(4,2) is a tree edge
(1,5) is a forward edge
no cross edges

BFS Tree (start vertex 2):



(1,5) is tree edge
no forward edges
(3,5) is a cross edge

Classifying graph edges with BFS/DFS:

- With respect to the traversal forest, categorize edges into 4 disjoint sets. An edge $e = (u, v)$ is a
  1. Tree edge: the edge (u, v) is in the forest
  2. Forward edge: v is a descendant of u
  3. Back edge: v is an ancestor of u
     Note: in undirected graphs, "back" $=$ "forward"
  4. Cross edge: v is a non-ancestor and non-descendant of u
- Whenever we traverse an edge (u, v), u has to be gray (it was discovered and we are not done with u yet)
- **In DFS** the color of v classifies the edge:
  - v is white $\Rightarrow (u, v)$ is a tree edge
  - v is gray $\Rightarrow (u, v)$ is a back edge
  - v is black $\Rightarrow (u, v)$ is a cross edge / forward edge
- **In DFS on an undirected graph** there are only tree- and back-edges.
  - ASOC that (u, v) is a cross-edge.
  - A cross-edge means $[v.dtime, v.ftime]$ comes before $[u.dtime, u.ftime]$.
  - Therefore, at time $v.ftime$, u is white.
  - So we are done traversing all neighbors of v and ignored u. Contradiction.
- **In BFS on an undirected graph** there are only tree- and cross-edges.
  - For any edge (u, v) we have $|L(u) - L(v)| \leq 1$ so a back-edge must be a tree edge.

DFS Application 1: Directed Acyclic Graph (DAG)

- Thm 1: DFS has a back edge iff G contains a cycle.
  - Proof: $\Rightarrow$ the back-edge (u, v) along with the tree edges connecting v to u is a cycle in G.
    $\Leftarrow$ If there's a cycle let v_1 be the first node on the cycle that turns gray. So the cycle is $(v_1, v_2, .., v_k, v_1)$.
    At time $v_1.dtime$ the $v_1 \rightarrow v_k$ path is all white, so v_k is a descendant of v_1. Thus when the edge (v_k, v_1) is traversed, both vertices are gray, so it is a back-edge.
- Corollary: G is a DAG iff the DFS has no back-edges.
- An algorithm to determine if G is a DAG:
  Run DFS.
  - If DFS encounters a gray-gray edge (u, v), abort and output "found a cycle" (traverse $predec$ from u until you reach v to output the cycle itself)
  - If DFS concludes without a gray-gray edge, output "DAG".
- Thm 2: G is a DAG iff there exists a **topological sorting** of its vertices

  Topological Sort: An ordering of V such that for every edge (u, v) in the DAG, u appears before v.
  - $\Leftarrow$ If there's a cycle $(v_1, ..., v_k, v_1)$, then any ordering of V must place either v_1 after v_k or v_k after v_1 and cannot be a topological sort.
  - $\Rightarrow$ Why if G is a DAG there must be a topological sort???

DFS Application 1: Directed Acyclic Graph (DAG)

- If G is a DAG, we construct the topological sort, using DFS.
  - G is a DAG $\Rightarrow$ no back-edges
  - No gray-gray edges.
  - (u, v) is a gray-white edge:

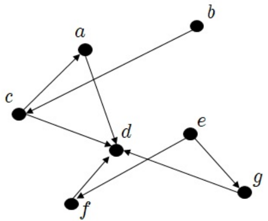
$$u.dtime < v.dtime < v.ftime < u.ftime$$

  - (u, v) is a gray-black edge:

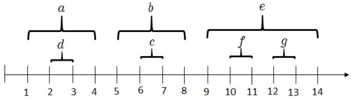
$$v.dtime < v.ftime < u.dtime < u.ftime$$

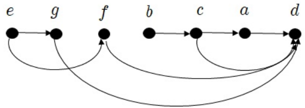
  - $dtime$ isn't consistent, but $ftime$ is:
    we must have $v.ftime < u.ftime$ for any edge (u, v)
- Sort the vertices by descending order of $ftime$ and you got a topological sort.
- Doesn't have to take extra $O(n \log(n))$. Can be done as part of the DFS algorithm
  - When a node turns black, insert it to a $TopoSort$ array
  - Or `Push()` it into a $TopoSort$ stack
- After DFS, print the array in reverse order / `Pop()` and print elements in the stack.
- Conclusion: A $O(n + m)$-time algorithm for topologically-sort a DAG or output a cycle.

DFS Application 1: Directed Acyclic Graph (DAG)



- An example:
  Assume nodes are stored in alphabetic order. $V = [a, b, c, d, e, f, g]$
- Running DFS results in the following timeline
  (Brackets indicate the subinterval when the node was gray)





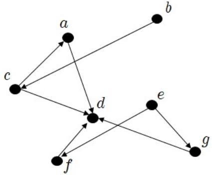
- The resulting topological sort is
- Note: if V held the nodes in a different order, the resulting topological sort would have been different. (Try it!)

DFS Application's Application: Finding Longest Path in Digraph

- A $O(n+m)$-time algorithm for topologically-sort a DAG or output a cycle.
- If the digraph has a cycle, the longest path in G has length ∞.
  - So don't confuse this with the LONGEST SIMPLE PATH IN DIGRAPH problem — very hard for digraphs with cycles
- If G is a DAG, how can we find the longest path?
- Note: longest-path in $G = \max\{LP(v,G)\}$ where $LP(v,G) \stackrel{\text{def}}{=}$ the longest path in G starting with v.
- Moreover, denote $V_{\geq v}$ as the set of vertices that appear after v in the topological sorting. Any path starting at v can only traverse nodes in $V_{\geq v}$.
- Therefore $LP(v,G) = LP(v, G[V_{\geq v}])$.
- We set an array A such that $A[v] =$length of $LP(v, G[V_{\geq v}])$.
  - For vertices with no out-neighbor (such as the last vertex in the topological order) $A[v] = 0$
  - For vertices with out-neighbors: $A[v] = \max\limits_{\{u \text{ out-neighbor of } v\}} 1 + A[u]$

    Hence, runtime per node $= O(|\Gamma(v)|)$.
- We now use A to print the longest simple path on the DAG:
  - To find the starting node of the longest simple path — `FindMax(`A`)`.
  - Given a node v on this path, its following node u is an out-neighbor u of v for which $A[v] = 1 + A[u]$. (Finding u takes $O(|\Gamma(v)|)$-time.)
- All in all: $O(|V| + |E|)$-time.
- This is called a "Dynamic Programming" type of an algorithm.

DFS Application's Application: Finding Longest Path in Digraph



- An example:
- We have already sorted it: $Sort = [e, g, f, b, c, a, d]$
- We fill A in the reverse order of $Sort$:
  - First, $A[d] \leftarrow 0$.
  - Then $A[a] \leftarrow 1$.
  - Then $A[c] \leftarrow 1 + \max\{1, 0\} = 2$
  - Then $A[b] \leftarrow 3$.
  - (skipping ahead), result: $A = [2, 1, 1, 3, 2, 1, 0]$.
- Now printing the longest path:
  - Find max entry of A: b
  - Find u — neighbor of b such that $A[b] = 1 + A[u]$ — $u \leftarrow c$.
  - Find u — neighbor of c such that $A[c] = 1 + A[u]$ — $u \leftarrow a$
  - Find u — neighbor of a such that $A[a] = 1 + A[u]$ — $u \leftarrow d$
  - Now $A[d] = 0$ (or d has no out-degree neighbors), we halt.

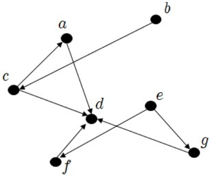
DFS Application 2: Finding Strongly-Connected Components

- Recall: In a digraph G, $SCC(u)$ is the set of all nodes v that are reachable from u and that u is reachable from them.
- Recall: $v \in SCC(u)$ iff $u \in SCC(v)$
- Recall: the SCCs of G form a partition of V into $\{C_1, C_2, ..., C_k\}$.
- Moreover, draw graph G_{SCC} on k nodes: $v_1, ..., v_k$ (so that v_i represents C_i). Put en edge (v_i, v_j) iff for some $x \in C_i, y \in C_j$ such that (x, y) is an edge in G. Then G_{SCC} is a DAG.
- Moreover, C is a SCC in G iff it is a SCC in the flipped graph G^T. ((u, v) is an edge in G iff (v, u) is an edge in G^T)

- To find the SCCs of G
  1. Run DFS on G.
  2. Flip G's edges to create G^T
  3. Run DFS on G^T **but** the main DFS loop traverses nodes in a decreasing $ftime$ order
  4. SCCs of G are the trees of the DFS-forest of G^T
- Runtime $O(n + m)$.

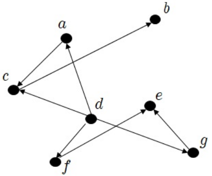
DFS Application 2: Finding Strongly-Connected Components

- An example:

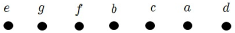


  (Note: this graph is a DAG, so the answer should be 7 *singleton* components)
  $ftime$ order (from smallest to largest): $[d, a, c, b, f, g, e]$

- Flipping the graph:



- When you do DFS on the *flipped* graph, using the order $[e, g, f, b, c, a, d]$ it doesn't traverse even a single edge.

  DFS forest is 7 singleton components: 

- (Test yourself:) Run the algorithm on the same graph but with (c, d)-edge flipped. What forest do you get at the end?

DFS Application 2: Finding Strongly-Connected Components

- To find the SCCs of G
  1. Run DFS on G.
  2. Flip G's edges to create G^T
  3. Run DFS on G^T **but** the main DFS loop traverses nodes in a decreasing $ftime$ order
  4. SCCs of G are the trees of the DFS-forest of G^T
- **OPTIONAL:** Intuition for correctness
  - First observe that $(G^T)_{SCC} = (G_{SCC})^T$
  - For any SCC C_i of G, let $x_i \in C_i$ be the node with largest $ftime$.
  - Because G_{SCC} is a DAG, sorting $x_1, x_2, ..., x_k$ in descending order of $ftime$ is a topological sort of G_{SCC}.
    Denote this ordering $x_{i_1}, ..., x_{i_k}$.
  - And so, $x_{i_1}, ..., x_{i_k}$ is *the inverse* of a topological sort of G^T_{SCC}. This means that not a single edge leaves x_{i_1} in G^T_{SCC}.
  - x_{i_1} is the first vertex in the second DFS pass (DFS of G^T). Therefore, all node reachable from x_{i_1} are nodes in its SCC.
  - Continue inductively to argue that by the time we call DFS on x_{i_j} then all nodes in the SCCs $C_{i_1}, ..., C_{i_{j-1}}$ are already black, so DFS from x_{i_j} only reaches nodes in the SCC of x_{i_j}.