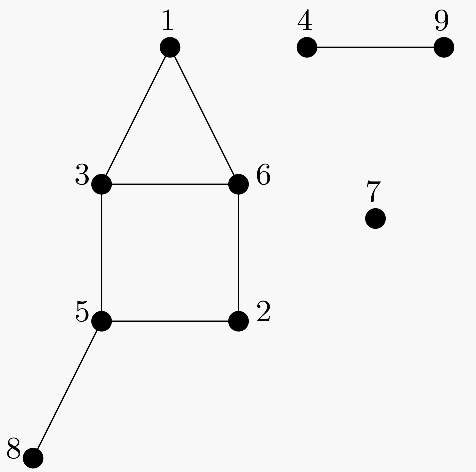
Agenda:

- Graphs — basic definitions
- Graphs Representation
- Graph Traversals: BFS, DFS, Classifying Types of Edges
- DFS applications
  - Topological Sorting
  - Strongly Connected Components (may postpone to the following week)

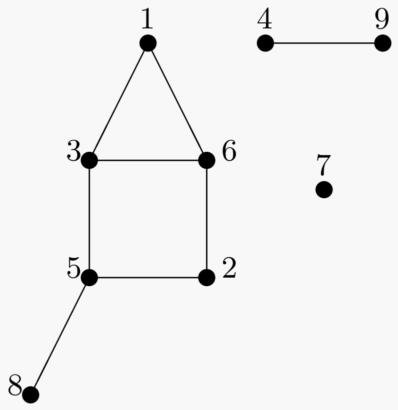
Reading:

- CLRS: 589-623

A Graph:



Basic Graph Definitions $G = (V, E)$



- V Nodes / Vertices: A set of n elements with unique identifiers; usually numbers from $\{1, ..., n\}$.
- E Edges
  - Undirected Graph: each edge is a set of exactly two nodes $e = \{u, v\}$
    - We say e connects u and v
    - We say u and v are adjacent, or neighbors
  - Directed Graph (digraph): each edge/arc – an ordered pair $e = \langle u, v \rangle$.
    - We say e leaves u and enters v; or e is from u to v.
    - We say u and v are adjacent; u is an in-neighbor of v; v is an out-neighbor of u.

Basic Graph Definitions

- adjacent (vertex – vertex, edge – edge)
  e.g., 1 and 3 are adjacent; $(1,3)$ and $(3,5)$ are adjacent
- incident (vertex – edge)
  e.g., 1 is incident with $(1,3)$
- Loops / self-loops: if an edge is allowed to be of the form $e = uu$, i.e. connecting a node to itself.
  - Unless specified otherwise: assume no self loops, and no multiple edges
- Degree of v: # edges that touch v = # neighbors of v
  - In a digraph: separated into in-degree (#in-neighbors) and out-degree (#out-neighbors)
- A path: a sequence of nodes $v_0, v_1, \ldots, v_k$ such there exists k edges $e_1, \ldots, e_k$ where e_i connects v_{i-1} to v_i.
- A simple path: a path where all nodes are unique
  - k is the length of the path
  - We often assume a path is a simple path from v_0 to v_k
- A cycle: a path where $v_0 = v_k$.
- A simple cycle: a cycle where all nodes but v_0 and v_k are unique
  - k is the length of the cycle
  - We often assume a cycle is a simple cycle
- Size of the graph $|G| = |V| = n$
- We often use the notation $V(G), E(G)$.

- "A n-nodes and m-edges graph" — means $|V(G)| = n$ and $|E(G)| = m$.
  - Undirected graph: $m \leq \binom{n}{2}$.
  - Directed graph: $m \leq n(n-1)$.
  - The empty graph: no edge belongs to E
  - The complete graph: all edges belong to E
- Degrees and edges:
  - In an undirected graph $\sum_{v \in V} deg(v) = 2m$
  - In a directed graph $\sum_{v \in V} in\_deg(v) = m = \sum_{v \in V} out\_deg(v)$
- $G' = (V', E')$ is a sub-graph of $G = (V, E)$ if $V' \subset V$ and $E \subset E'$.
  - Removing an edge e from G results in the subgraph $(V, E \setminus \{e\})$
- The induced subgraph on $V' \subset V$ is the graph $G[V'] = G|_{V'} = (V', E')$ where $e \in E'$ iff $e \in E$ and both its vertices are in V'
  - Removing a node v from G results in the induced graph $G[V \setminus \{v\}]$

Connectivity in an Undirected Graph

- u is connected to v $(u \sim v)$ if there exists a path from u to v.
- G is a connected graph if for every $u, v \in V, \quad u \sim v$
- $C \subset V$ is the connected component of u $(CC(u))$ if it is the maximal set C such that $u \in C$ and $G[C]$ is connected.
  - We often identify C with $G[C]$

- Connectivity is an *equivalence relation*
  - Reflexivity: for every u we have $u \sim u$ by a path of length 0
  - Symmetry: for every u and v, $u \sim v$ iff $v \sim u$
  - Transitivity: for every u, v, w, if $u \sim v$ and $v \sim w$ then $u \sim w$
- So $C = CC(u)$ is unique $CC(u) = \{v \in V : \ u \sim v\}$.
  - So $u \sim v$ iff $CC(u) = CC(v)$.
- Thus the different connected components of G form a partition of G where every edge $e \in E$ belongs to a unique CC and no edge connects two components.

Forests and Trees

- A forest F is an acyclic graph.
- A tree T is a connected acyclic graph, and we say T spans the vertices $V(T)$.
  - The connected components of a forest are trees, each spanning all the vertices in its connected component
- All of the following definitions of a tree are equivalent:
  - A maximal acyclic graph
    - Adding any edge to T results in a cycle
  - A minimal connected graph
    - Remove an edge from T and it is no longer connected
  - A connected and acyclic graph
  - An acyclic graph with $n-1$ edges
  - A connected graph with $n-1$ edges
- A graph G is connected iff it has a spanning tree: a subgraph T which is a tree with $V(T) = V(G)$.

Biconnected component:

- Two paths connecting v_1 and v_2 are vertex-disjoint if share no common internal vertex (other than v_1 and v_2).
- Biconnected graph: $|V| \geq 2$, connected, and every pair of vertices are connected via two vertex-disjoint (simple) paths
- Notes:
  - connectivity does NOT implies biconnectivity
  - articulation vertex — cut vertex: its removal disconnects G
  - bridge — cut edge: its removal disconnects G
- Biconnected component — maximal biconnected subgraph
  - a partition of E (not necessarily a partition of V)

Strong Connectivity in a Digraph

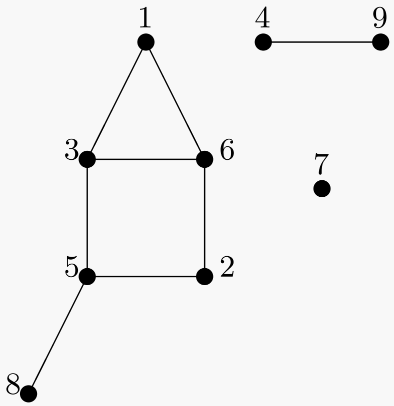
- v is reachable from u ($u \to v$) if there exists a path from u to v.
  - Defines distances: $d(u, v) \stackrel{\text{def}}{=}$ min length of path $u \to v$; or ∞ if no such path exists.
  - Not symmetric: $d(u, v) \neq d(v, u)$ in general (so common to use $d(u \to v)$)
- u and v are strongly connected ($u \sim v$) if there exists a path from u to v and a path from v to u.
  Exists a (directed) cycle containing both u and v.
- G is a strongly connected graph if for every $u, v \in V, \quad u \sim v$
- $C \subset V$ is the strongly-connected component of u ($SCC(u)$) if it is the maximal set C such that $u \in C$ and $G[C]$ is strongly-connected.

- Strong-connectivity is an *equivalence relation*
  - Reflexivity: for every u we have $u \sim u$ by a path of length 0
  - Symmetry: for every u and v, $u \sim v$ iff $v \sim u$
  - Transitivity: for every u, v, w, if $u \sim v$ and $v \sim w$ then $u \sim w$
- Thus the different SCCs of G form a partition of $V(G)$
- There could be an edge between two strongly-connected components, but no edge back

Representing Graphs

- Representing the nodes
  - We will assume that all nodes are stored in an array
  - Nodes will have different attributes / fields as required
    - degree, color, parent, distance, etc...
    - So the code "`if` $(v.color =$`WHITE`$)$" takes $O(1)$-time
    - Not the same as "`if exists some` v `with` $v.color =$`WHITE`" which takes naively $O(n)$-times to check, unless we do something clever...
- Representing the edges
  - Edges are given in one of two representations:
  - Adjacency matrix: an $n \times n$-matrix where the i, j-entry contains e if such an edge exists or 0 o/w.
  - Adjacency lists: each node has an array / a list of all the edges that are adjacent to it
  - Some operations are more efficient in the adjacency-matrix model, some operations are more efficient in the adjacency-list model.
  - NOTE: Edges will have attributes too (color, weight, capacity, label, etc...) Regardless of the representation we use, we assume that once we reach an edge e we can access its attributes in $O(1)$-time

An example:



node $\to$	array/list
1 $\to$	3 6
2 $\to$	6 5
3 $\to$	5 1 6
4 $\to$	9
5 $\to$	2 8 3
6 $\to$	3 2 1
7 $\to$	
8 $\to$	5
9 $\to$	4

	1	2	3	4	5	6	7	8	9
1	0	0	1	0	0	1	0	0	0
2	0	0	0	0	1	1	0	0	0
3	1	0	0	0	1	1	0	0	0
4	0	0	0	0	0	0	0	0	1
5	0	1	1	0	0	0	0	1	0
6	1	1	1	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	1	0	0	0	0
9	0	0	0	1	0	0	0	0	0

Comparison between the two representations

	Adjacency Lists	Adjacency Matrix
Space (so good for)	$O(m)$ sparse graphs	$O(n^2)$ dense graphs
Accessing a node v Traversing all nodes	$O(1)$ $O(n)$	$O(1)$ $O(n)$
Accessing an edge $e = (u, v)$ (finding if e exists)	$O(\|\Gamma(u)\|)$ $\#$ neighbors(u)	$O(1)$
Finding some neighbor of v	$O(1)$	$O(n)$
Traversing all edges/vertices adjacent to a node u	$O(\|\Gamma(u)\|)$	$O(n)$
Traversing all edges	$O(m)$	$O(n^2)$

Comments about Graph Representations

- If G is undirected, then the adjacency matrix is symmetric.
- Sometimes, in runtime analysis it is easier to use a max-degree bound $\Delta = \max_v deg(v)$ (since all lists have length $\leq \Delta$)
- We do not assume the lists are sorted according to the neighbors' identifiers, the neighbors' attirbutes or the edges' attributes. We will be responsible to sort them or keep them in order (using Priority Queues)
- Example: find if u and v are of distance=2
  - This means that exists some w s.t $(u, w) \in E$ and $(w, v) \in E$.
  - $O(n)$ in the matrix model
  - In the adjacency lists model (undirected graphs or if we keep incoming edges for each node):
    - Naïvely: for each neighbor x of u, check if x is in the adjacency list of v. Runtime $O(|\Gamma(u)| \times |\Gamma(v)|)$.
    - Better runtime: Sort first the list for u and for v, then iterate both, $O\left(|\Gamma(u)| \log(|\Gamma(u)|) + |\Gamma(v)| \log(|\Gamma(v)|)\right)$
    - One more way: construct a $\{0, 1\}$ array for all the nodes, and see if they are connected to u and v (i.e., build the respective row from the adjacency matrix) in $O(n)$ time.
    - Finally, you can use a hash-table: build a hash-table with the vertices adjacent to u, and try to `Find()` in it each vertex adjacent to v. This takes $O((|\Gamma(u)| + |\Gamma(v)|) \cdot t)$ where t is the time it takes to hash.

- A bipartite graph is a graph where V can be partitioned into two disjoint sets $V = R \cup L$, such that all edges have one right- and one left-vertex.
- A bipartite graph can be represented also by a $|R| \times |L|$-matrix.

Graph Traversal

- The most elementary graph algorithm:
- Goal: visit all vertices, by following the edge structure of the graph
- Via graph traversals we find all vertices connected/reachable from a given vertex u, find distances, connected components, characterize edges, etc.
  - *E.g.*, maze traversal — is there a path "enter"$\rightarrow$"exit"?
- There are two main principled ways to traverse the graph
  - Breadth First Search (BFS)
    - We start at v, then first visit all of its neighbors, then visit all of its neighbors' neighbors, then neighbors' neighbors' neighbors and so on.
    - Think of a balloon sitting at v and inflating until it shadows the entire graph
  - Depth First Search (DFS)
    - We start at v, take a path for as far as it takes us, then go up the path and take any other branches we can, until we exhaust all paths from v.
    - Think of water being poured on v until the entire graph is flooded.
- Both use the notion of a node color — representing its state

`WHITE`	$\xrightarrow[\text{previsit time}]{\text{discovery}}$	`GRAY`	$\xrightarrow[\text{postvisit time}]{\text{finish}}$	`BLACK`
Never saw v		Working on v		Done with v

  - All vertices start as `WHITE` and end as `BLACK`
  - The order in which we make these $2n$ color changes is of importance! (the time in which a vertex turns gray and when it turns black)

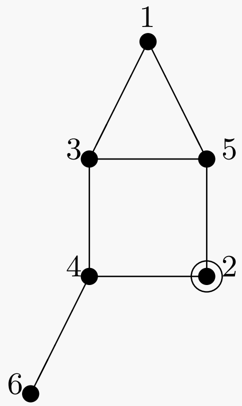
Breadth First Search (BFS):

- Assume for now all nodes are connected to s
- Pseudocode:

```
procedure BFS(G, s)                    ** G = (V, E),  s ∈ V start vertex
foreach v ∈ V do
   v.color ← WHITE                     **unknown yet
   v.dist ← ∞                          **distance from s
   v.predec ← NIL                      **predecessor
Initialize a queue Q                   **waiting vertex queue
s.color ← GRAY                         **in queue Q
s.dist ← 0
enqueue(Q, s)
while (Q ≠ ∅) do
   u ← dequeue(Q)
   foreach neighbor v of u do
      if (v.color = WHITE) then
         v.color ← GRAY                **discovered v
         v.dist ← u.dist + 1
         v.predec ← u
         enqueue(Q, v)
   u.color ← BLACK                     **done with u
```

BFS 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		

BFS example:

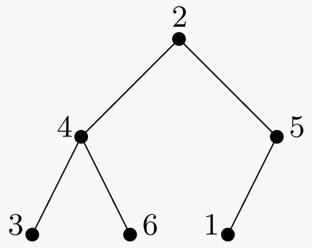
	1	2	3	4	5	6	Q
color	W	G	W	W	W	W	$\{2\}$
distance	∞	0	∞	∞	∞	∞	
parent	NIL	NIL	NIL	NIL	NIL	NIL	
color	W	B	W	G	G	W	$\{4, 5\}$
distance	∞	0	∞	1	1	∞	
parent	NIL	NIL	NIL	2	2	NIL	
color	W	B	G	B	G	G	$\{5, 3, 6\}$
distance	∞	0	2	1	1	2	
parent	NIL	NIL	4	2	2	4	
color	G	B	G	B	B	G	$\{3, 6, 1\}$
distance	2	0	2	1	1	2	
parent	5	NIL	4	2	2	4	
color	G	B	B	B	B	G	$\{6, 1\}$
distance	2	0	2	1	1	2	
parent	5	NIL	4	2	2	4	
color	G	B	B	B	B	B	$\{1\}$
distance	2	0	2	1	1	2	
parent	5	NIL	4	2	2	4	
color	B	B	B	B	B	B	$\emptyset$
distance	2	0	2	1	1	2	
parent	5	NIL	4	2	2	4	

BFS example:

- Adjacency lists:

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

- BFS tree:
  - root is the start vertex s
  - parent of u is predecessor $u.predec$
  - left-to-right child order *depends* on neighbor ordering (in u's list)



Properties of BFS

- Each u that is reachable from s is visited,
  enqueued exactly once (turns `GRAY`)
  and dequeued exactly once (turns `BLACK`)
- For any u denote $d(u, s)$ the true distance between s and u, and $u.dist$ as the *distance* given by BFS.
  Claim: $u.dist = d(u, s)$, and the path from u to s using the predecessors is a shortest path.
  - Prove by induction on d that all nodes u with $d(u, s) = d$ are assigned $u.dist = d$.
  - Base case $d = 0$ and we only have s to consider.
  - Induction step. Let u be a node s.t. $d(u, s) = d + 1$. On *all* shortest-paths from s to u the next-to-last node must be of distance d from s, so by IH it was assigned $dist = d$; and in particular – it had to be turned gray and enqueued by the BFS. So, among all next-to-last-nodes let x be the first node to be enqueued. This means u is discovered by the edge (x, u), which means $u.dist = x.dist + 1 = d + 1$.

Properties of BFS

- Each u that is reachable from s is visited,
  enqueued exactly once (turns `GRAY`)
  and dequeued exactly once (turns `BLACK`)
- For any u denote $d(u, s)$ the true distance between s and u, and $u.dist$ as the *distance* given by BFS.
  Claim: $u.dist = d(u, s)$, and the path from u to s using the predecessors is a shortest path.
- BFS creates layers $L_i = \{u : u.dist = i\}$ such that for any edge (u, v) we have $L(v) - L(u) \leq 1$.
  - For an undirected graph — all edges are between the same or adjacent layers.
- For any u, v, if $L(u) < L(v)$, then u turns `GRAY` before v, enqueued before v and turns `BLACK` before v.
- At any moment, all vertices in the queue belong to the same or adjacent layers. (But never layers at distance ≥ 2)

BFS runtime analysis:

- $n = |V|$, $m = |E|$
- Analysis:
  - each vertex enqueued exactly once: WHITE $\rightarrow$ GRAY
  - each vertex dequeued exactly once: GRAY $\rightarrow$ BLACK
  - running time:
    1. adjacency list representation:
       $\Theta(n + \sum_{v \in V} \texttt{degree}(v)) = n + 2m) = \Theta(n + m)$
    2. adjacency matrix representation:
       $\Theta(n + \sum_{v \in V} n = n + n^2) = \Theta(n^2)$
  - space complexity: (in addition to the list / matrix representation)
    1. Each node has a color attribute $\Omega(n)$
    2. Since each vertex is enqueued exactly once, the queue size never passed $O(n)$
    3. So $\Theta(n)$.
- Warning: vertices in other connected components wouldn't be discovered!!!

Breadth First Search (BFS):

- ```
  procedure BFS(G)                        ** G = (V, E)
  foreach v ∈ V do
      v.color ← WHITE                     **unknown yet
      v.dist ← ∞                          **distance from s
      v.predec ← NIL                      **predecessor
  foreach v ∈ V do
          if (v.color =WHITE) then
              BFS-visit(G, v).
  ```

- ```
  procedure BFS-visit(G, s)               ** G = (V, E), s ∈ V start vertex
  Initialize a queue Q                    **waiting vertex queue
  s.color ← GRAY                          **in queue Q
  s.dist ← 0
  enqueue(Q, s)
  while (Q ≠ ∅) do
      u ← dequeue(Q)
      foreach neighbor v of u do
          if (v.color = WHITE) then
              v.color ← GRAY              **discovered v
              v.dist ← u.dist + 1
              v.predec ← u
              enqueue(Q, v)
      u.color ← BLACK                     **done with u
  ```
- Runtime?
- HW: In an undirected graph — adjust BFS to assign each vertex a label such that the labels indicate the connected components of G.

Depth First Search (DFS):

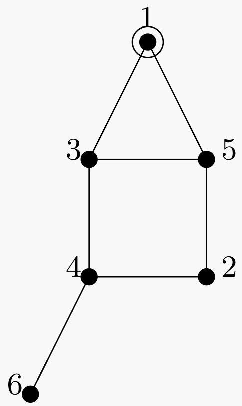
- Input: graph $G = (V, E)$
- Idea: search deeper in the graph whenever possible ...
- Pseudocode (recursive version):

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

procedure DFS-visit(G, s)      **any s ∈ 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(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:

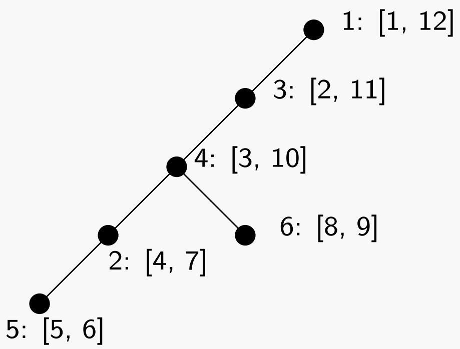
1:	3	5	
2:	4	5	
3:	1	4	5
4:	2	3	6
5:	1	2	3
6:	4		

	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: $[\texttt{dtime}, \texttt{ftime}]$



- 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 $v \Rightarrow [u.\texttt{dtime},\ u.\texttt{ftime}] \subset [v.\texttt{dtime},\ v.\texttt{ftime}]$

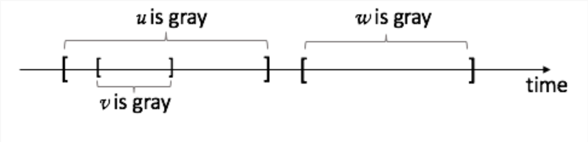
    — u & v on different branches $\Rightarrow [u.\texttt{dtime},\ u.\texttt{ftime}]$ doesn't intersect $[v.\texttt{dtime},\ v.\texttt{ftime}]$

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



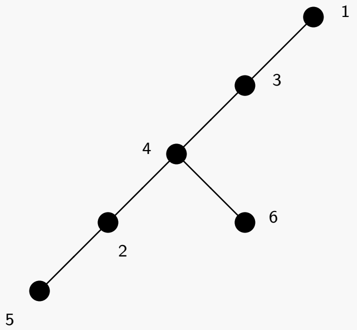
- The White-Path Theorem:
  v is a descendant of u iff at time $u.dtime$ there was a path $u \to v$ along which all vertices are white (except for u).
  - An all gray path at time $v.dtime$
  - and all black path at time $u.ftime$.
- DFS vertex order:
  pre-order of each tree in the DFS forest
- (BFS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

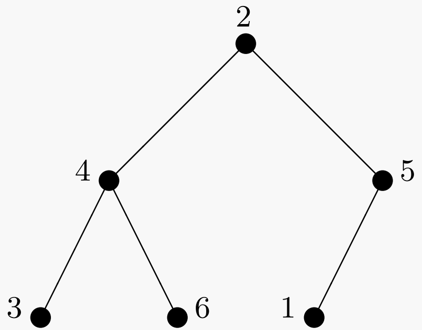
An example:

- 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):



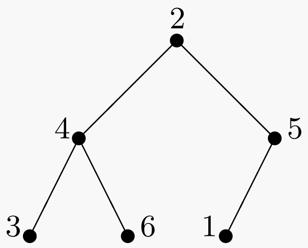
(1,5) is tree edge
no forward edges
(3,5) is a cross edge

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.

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
    $(3, 6, 4, 1, 5, 2)$

Comparing DFS and BFS:

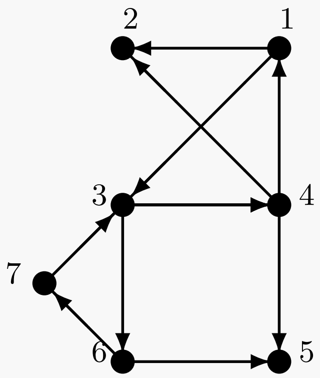
- BFS works well for finding shortest path
- All non-tree edges in
  - BFS are cross edges
  - DFS are back edges

Directed graphs:

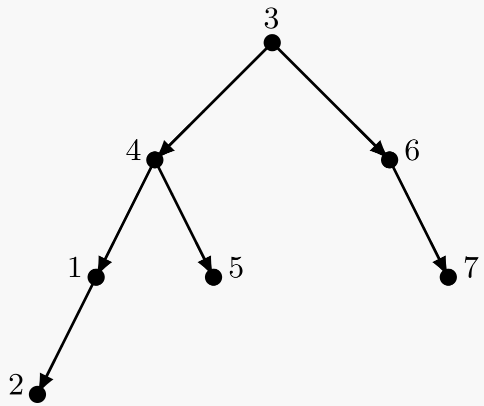
- Recall that in a directed graph every edge is directed (i.e. it is an ordered pair)
- We say u reaches v if there is a directed path from u to v
- Strongly connected digraph: A digraph G is strongly connected if for every pair u, v of vertices, u is reachable from v and v is reachable from u
- The 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?
- We can detect if G is a DAG with just one DFS call.

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 $v_1 \to v_k$ 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, abort and output "found a cycle"; upon DFS conclusion output "DAG".

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 sorting"
- Theorem: A digraph has a topological sorting 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 at least one vertex in G which has no incoming 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.
- The above suggests the following algorithm:

- procedure Topological-Sort(G)

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

- We can maintain (for each node) the value of in-degree(v); all of these can be computed in $\Theta(n+m)$ by going through adjacency list.
- We have n iterations of the while loop; each time we remove a vertex v with out-degree $d_{out}(v)$ we have to update in-degree of all its neighbors (and if any of them becomes zero we insert that node into the queue); this update takes $d_{out}(v)$

- We can also use DFS to find a topological ordering (as in the textbook).
  - G is a DAG $\Rightarrow$ no back-edges, i.e., 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:

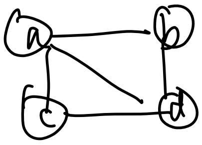
$$v.dtime < v.ftime < u.dtime < u.ftime$$

  - $dtime$ 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 the end of a $TopSort$ array
  - Or `Push()` it into a $TopSort$ stack
- After DFS, print the array / `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 2: Finding Strongly-Connected Components

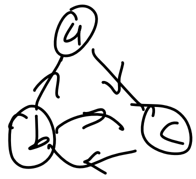
- A directed graph every edge is directed (i.e. it is an ordered pair)
- We say u reaches v if there is a directed path from u to v
- Strongly connected digraph: A digraph G is strongly connected if for every pair u, v of vertices u is reachable from v and v is reachable from u
- Recall: In a digraph 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 $O(n + m)$.

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



undirected

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



directed

$Adj[b] = \{a, c\}$

$Adj[c] = \{b\}$

$Adj[a] = \{c\}$

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

graph representation

- adj. lists: ← $\Theta(V+E)$

array adj. of $|V|$

linked lists

- for each vertex $u \in V$,

$Adj[u]$ stores its neighbors.

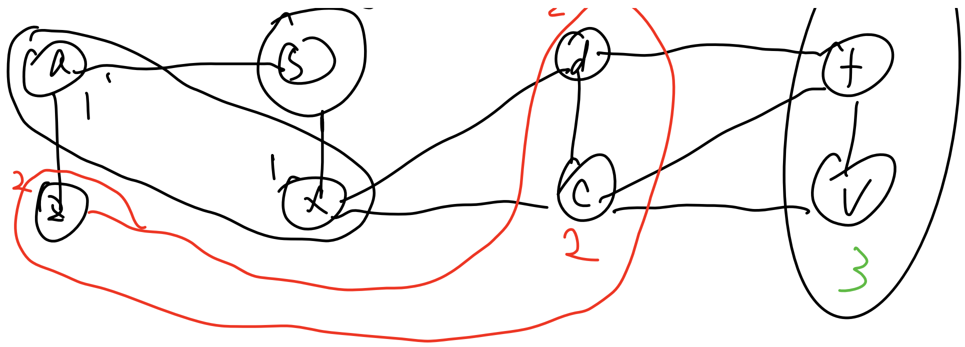
BFS

~ $\Theta(V+E)$ times

- look at nodes reachable.

- visit all nodes reachable from given set

- to avoid duplicates.



shortest path property:

$v \leftarrow parents[v]$

$\leftarrow parent[parent[v]]$

$\leftarrow$

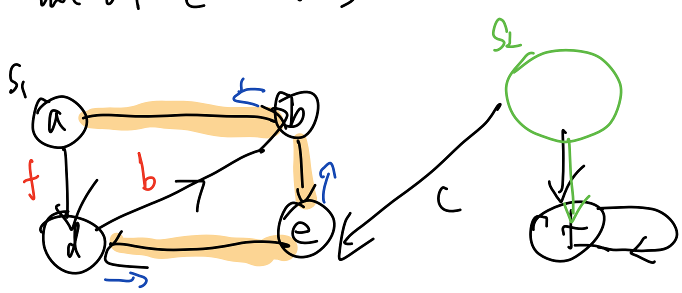
$\leftarrow s$

is a shortest path from s to v

DFS

- recursively explore graph, backtracking as necessary.

- not repeat vertices



time = $\Theta(V+E)$

- visit each vertex once

in DFS alone O(V)

- DFS visit (1, ..., v) called once for vertex v.

<u>edge classification</u>:

→ - tree edge (parent pointer)

visit new vertex via edge

forward edge   node – descendant   tree

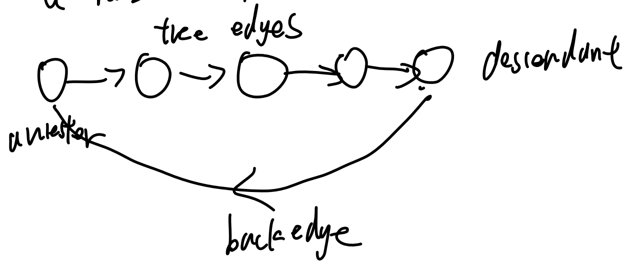
backward edge   node – ancestor   tree

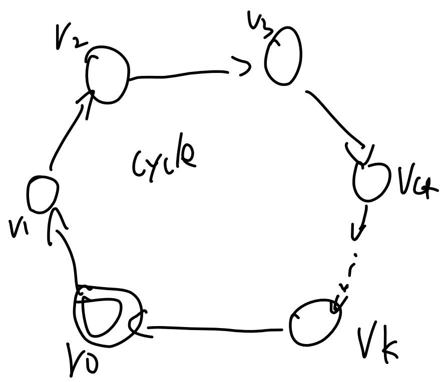
cross edge.   between two   non-ancestor-related   trees

exist in undirected graph

cycle detection

G has a cycle $\longleftrightarrow$ DFS has a back edge.





assume V_0 is first vertex in the cycle visited by DFS.

claim: $(V_k — V_0)$ is back edge.

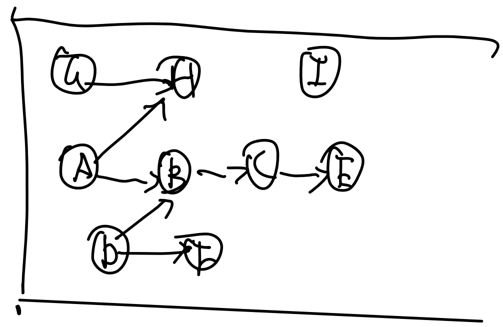
Job scheduling:

given directed $\boxed{\text{acyclic graph}}$ — DAG

order vertices so that all edges point from lower → higher

Topological sort: run DFS

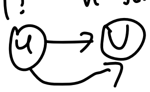
output = reverse of finishing time of vertices



prove: for any edge e=(u,v)

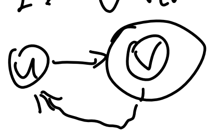
v finishes before u

case 1: u starts before v



visit v before u finishes

case 2: v starts before u



v finishes before visit u

~~back edge~~