Graphs (cont.)

Reading: Sections 3.3, 3.5 and 3.6
Graph Traversal

- **Breadth-first Search**
  - Traverse graph in layers
  - Find shortest distance from node $s$ to all other nodes in the connected component

- **Depth-first Search**
  - Traverse graph by going as deeply as possible
  - Find a path from node $s$ to all other nodes in the connected component
Implementation

Maintain set of *explored* and *discovered* nodes

- Explored = have seen this node and explored its outgoing edges

- Discovered = have seen the node, but not explored its outgoing edges.
Graph Traversal

L = data structure of discovered nodes

Traverse(s)
Put s in L

while L is not empty do
Take a node v from L
if v is not marked "explored" then
Mark v as "explored"
for each edge (v, w) incident to v do
Put w in L
end for
end if
end while

● BFS?
● DFS?
● Neither?
BFS implementation

L = queue of discovered nodes (FIFO)

Traverse(s)
Put s in L

while L is not empty do
    Take a node v from L
    if v is not marked "explored" then
        Mark v as "explored"
        for each edge (v, w) incident to v do
            Put w in L
        end for
    end if
end while

● BFS?
● How can we get tree?
● Running time
BFS properties

Claim. Let $T$ be a breadth-first search tree, let $u$ and $v$ be nodes in $T$ belonging to layers $L_i$ and $L_j$, respectively. If $(u,v)$ is an edge in $G$, then $i$ and $j$ differ by at most 1.

Claim. For each $j \geq 1$, layer $L_j$ produced by BFS consists of all nodes at distance exactly $j$ from $s$. There is a path from $s$ to $t$ if and only if $t$ appears in some layer.
DFS implementation

L = stack of discovered nodes (LIFO)

Traverse(s)
Put s in L

while L is not empty do
  Take a node v from L
  if v is not marked “explored” then
    Mark v as “explored”
    for each edge (v, w) incident to v do
      Put w in L
    end for
  end if
end while

● DFS?
● How can we get tree?
● Running time
DFS properties

Claim. Let $T$ be a depth-first search tree, let $u$ and $v$ be nodes in $T$, and let $(u,v)$ be an edge of $G$ that is not an edge of $T$. Then one of $u$ or $v$ is an ancestor of the other.
Connected Components

**Def.** The connected component $C(v)$ of node $v$ is the set of all nodes with a path to $v$.

**Claim.** For any two nodes $u$ and $v$ either $C(u) = C(v)$, or $C(u)$ and $C(v)$ are disjoint.

- Can we use traversals to find connected components?
Direction Matters
Directed Graphs

- Set of nodes $V$

- Set of ordered pair of nodes (edges) $E$

- *Important.* the existence of a path from $s$ to $t$ does not imply there is a path from $t$ to $s$
Connectivity in Directed Graphs

- G is *strongly connected* if for any nodes $u, v$ in G, there is a path from $u$ to $v$ and path from $v$ to $u$.

- *Strongly connected component* containing vertex $s$ is the set of all nodes with paths to and from $s$.

- Graph traversals?
**Directed Acyclic Graphs (DAG)**

**Def.:** A DAG is a directed graph that contains no directed cycles.

**Def.:** A topological order of a DAG is an ordering of its nodes $v_1, v_2, \ldots, v_n$ so that for every edge $(v_i, v_j)$ we have $i < j$. 

![Directed Acyclic Graphs Diagram](image)
Dependency Graph on DAG

- Usually reflect dependencies or requirements
  - I.e., Assembly lines, Supply lines, Organizational charts, …

- Understanding dependencies requires “topological sorting”
Topological Sorting Algorithm

TPsort(G)

while there are nodes remaining do
    Find a node v with no incoming edges
    Place v next in the order
    Delete v and all of its outgoing edges from G
end while