Lecture 3: Basic Graph Algorithms

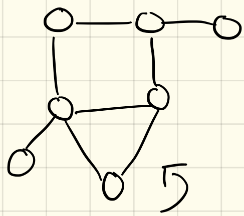
- Graph $G = (V, E)$: V are <u>vertices</u>, and $E \subseteq V \times V$ are edges, written as $\{u, v\}$ $u, v \in V$
- Directed Graph - graph in which each edge (u, v) has a direction from u (the tail) to v (the head) of the edge.

Def: Path P is a sequence of vertices $v_1, v_2, \ldots, v_K$ where each v_i, v_{i+1} is joined by an edge.

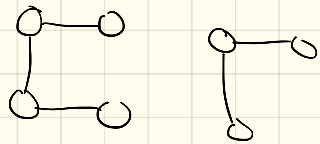
- a path is <u>simple</u> if no vertex is repeated in P
- a path is a cycle if the length of P is > 2 and $v_1 = v_K$

Def: A graph is connected if, $\forall\, u, v \in V$, there exists a path from u to v

E.g. G_1



G_2



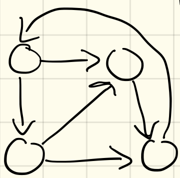
G_1 is connected

G_2 is not connected

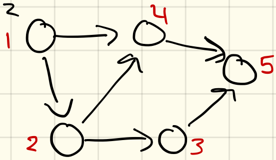
Def: A directed graph is <u>strongly connected</u> iff there is a <u>directed</u> path from u to v $\forall\, u,v \in V$

E.g.

G_1



G_2



G_1 is <u>strongly connected</u>

G_2 is <u>not strongly connected</u>

e.g. no directed path from 4 to 3.

Def: A directed graph is <u>weakly connected</u> if, when viewed as an undirected graph, it is connected.

E.g. G_2 above <u>is</u> weakly connected.

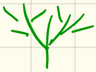
Def: An undirected <u>tree</u> is an undirected graph that is connected and contains no cycles.

Some Facts:

- deletion of any edge will disconnect the tree

- rooted tree – imagine we select a node "r" to be the root, and "conceptually" orient all edges "away" from the root.

  - on the path from the root to some vertex <u>v</u>, we traverse the ancestors of v. The direct ancestor is the <u>parent</u> and v is its <u>child</u>. Vertices with no children are <u>leaves</u>.

Characterizations of trees 

(3.1) Fact: Every n-node tree has exactly n-1 edges.
The following statements are all equivalent and all characterize a tree.

(1) T is a tree
(2) T contains no cycles and n-1 edges
(3) T is connected and has n-1 edges
(4) T is connected and removing any edge disconnects it
(5) Any 2 nodes in T are connected by 1 path
(6) T is acyclic, and adding any edge creates exacty 1 cycle

Note: remembering these different characterizations of trees will be important when we discuss how to create/find trees. That is, one can view many of these as <u>constructive</u> definitions.

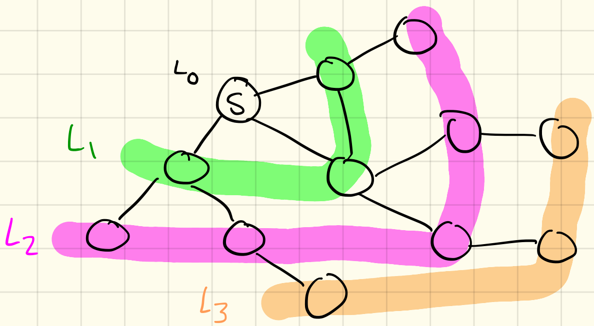
Graph Traversals [Breadth First Search (BFS) and Depth First Search (DFS)]

Problem: s-t connectivity – given a graph $G=(V,E)$ and two nodes $s,t \in V$, does there exist a path P from s to t?

One solution to this problem is to perform a BFS from s and see if we encounter t.

- Begin at s, visit all neighbors of s, visit all neighbors of those neighbors ... etc.

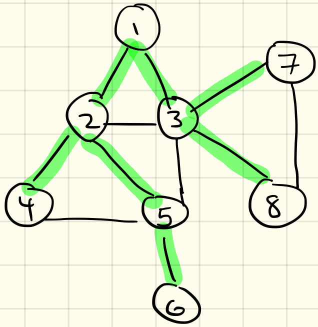
  – vertices are visited in "layers" $s = L_0$, $L_1 = \{u \in V \mid \{s,u\} \in E\}$, ..., $L_{i+1} = \{v \in V \mid \{u,v\} \in E \text{ and } u \in L_i\} - \bigcup_{j=0}^{i} L_j$



Fact: For each $j \geq 1$, L_j consists of all nodes from G at a distance of exactly j hops from s. There is an s-t path iff t appears in some layer.

☆ Note: BFS naturally produces a tree that we call a BFS-tree.

Consider another example: Consider a BFS starting at vertex 1.



- the — edges are in the BFS-tree.
- the — edges are not.

Fact: The nodes of the BFS-tree rooted @ s is <u>precisely</u> the connected component containing s (the set of all t such that an s-t path exists).

⇝ BFS provides an <u>order</u> in which to explore the connected components of G... there are <u>other</u> orders like.

DFS (Depth First Search)

⇒ Basic idea: start at s, follow edges until there are no other visited nodes to which to traverse. <u>Backtrack</u> until the current vertex has unvisited neighbors, repeat.

this approach to traversal is "recursive".

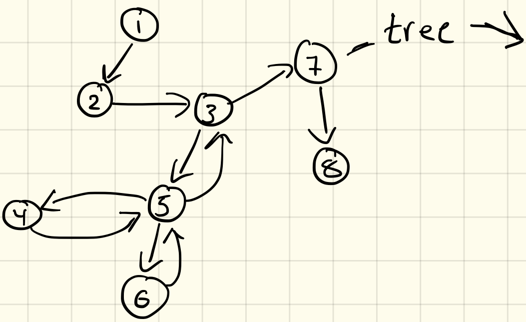
Pseudocode

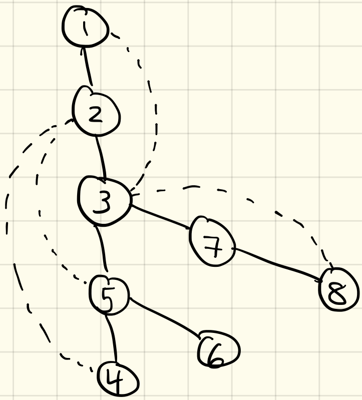
(recursive)

DFS (G, u):

mark u as visited and add u to R
for {u,v} incident to u:
    If v is not visited:
        DFS(G,v)
    EndIf
End For

DFS also results in a tree ... a DFS-tree





Fact: Given a DFS tree T, and two nodes $x, y \in T$ such that $\{x,y\} \in E$ but $\{x,y\} \notin T$. Then either x is an ancestor of y or y is an ancestor of x.

Main difference in implementation between BFS/DFS is the order in which we visit neighbors of a newly-discovered node.

BFS(u,G):

```
Set visited[u] = true and visited[v] = false ∀ v≠u
toVisit.append(u)
T = {}
while toVisit is not empty:
    v = toVisit.front
    toVisit.popFront
    for each {u,v} adjacent to u:
        if visited[v] is false:
            visited[v] = true
            T = T ∪ {u,v}
            toVisit.append(v)
        End if
    End for
End while
```

Note: We push onto the back of the queue, but we remove from the front. This gives us the relevand breadth-first behavior.

(3.11) Claim: The BFS algorithm runs in $O(m+n)$ time, assuming each incident edge to a vertex can be listed in $O(1)$ time (Q: what graph representation(s) can do this?)

(non-recursive)

DFS(u):

```
T = {} ; parent = {} ; parent[u] = u
explored[v] = false   ∀ v ∈ V
S.pushFront(u)
While S is not empty:
    u = S.front
    S.popFront
    if explored[u] is false:
        explored[u] = true
        T = T ∪ {u, parent[u]}
        for each {u,v} incident to u:
            S.pushFront(v)
            parent[v] = u
        End for
    End if
End while
```

$\Rightarrow$ This implementation of DFS is also $O(m+n)$.

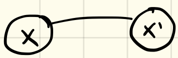
Problem: Testing bipartiteness of a graph

- Given a graph $G = (V, E)$, is G bipartite?
  - $\Rightarrow$ Bonus: return V_1 and V_2, the left + right vertex sets

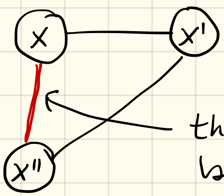
Recall: A graph $G = (V, E)$ is bipartite iff we can decompose V as $V = V_1 \cup V_2$ such that $\forall \{u, v\} \in E$ either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$.

(3.14) **Claim**: A graph is bipartite iff it contains no cycles of an odd length.

Why? Say (wlog) you start at some $x \in V_1$



if G is bipartite, the first edge must take you to some $x' \in V_2$



this edge would prevent G from being bipartite.

the second edge must take you back to some $x'' \in V_1$. If the third edge connects x'' to x, G can't be bipartite. This is true for any odd lengt cycle

Let G be a connected graph, and let $L_0, L_1, L_2, \ldots$ be the layers of BFS(s).

Then, either

(1) There is no edge of G joining two vertices of the same layer $\Rightarrow$ G is bipartite

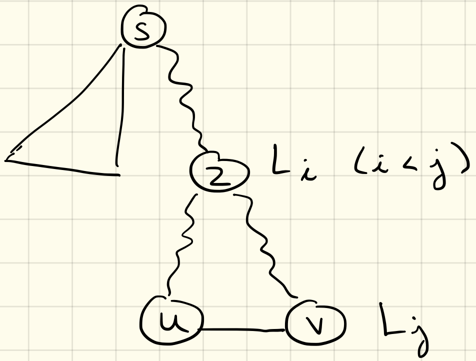
(2) There is an edge of G joining 2 vertices of the same layer $\Rightarrow$ G contains an odd-length cycle $\Rightarrow$ G is <u>not</u> bipartite.

Proof: Consider (1).

Every edge of G can be assigned either to vertices <u>within some layer</u> or vertices <u>between adjacent layers</u>. Since, by (1), no edge joins nodes in the same layer, then <u>every</u> edge is between nodes of adjacent layers. Thus, we can assign every odd layer to V_1 and every even layer to V_2. The resulting labeling shows that the graph is bipartite (i.e. all edges go between V_1 and V_2).

Consider (2). G contains an edge btw verts. of same layer

Let $e = \{u, v\}$ be some such edge with $u, v \in L_j$. Consider the BFS tree of S, and let z be the node in the largest layer that is an ancestor of both u and v. Here, we call z the Lowest Common Ancestor (LCA) of u and v written as $LCA(u,v)$. We have a situation like the following:



Consider the cycle C defined by $z \rightsquigarrow u, e, v \rightsquigarrow z$. What is the length of such a cycle?

$$|C| = \underbrace{(j-i)}_{z \rightsquigarrow u} + \underbrace{1}_{e} + \underbrace{(j-i)}_{v \rightsquigarrow z} = \underbrace{\underbrace{2(j-i)}_{\text{even}} + 1}_{\text{odd}}$$

Thus, any such cycle is odd in length, and implies that G is not bipartite.

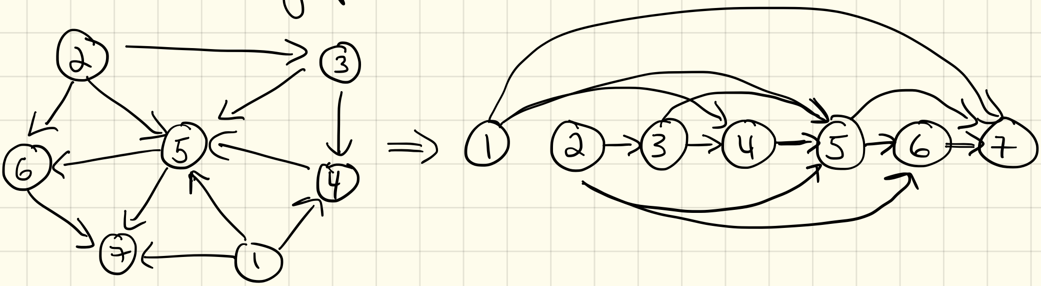
Directed Acyclic Graphs (DAGs) and topological orderings.

DAGs are a special type of directed graph. They will come up again and again in this course (and in algorithms more generally). Being a DAG is equivalent to being a directed graph with no cycles, wich is equivalent to being topologically olderable.

Example: Encode dependencies in a makefile. What targets need to be built before others? DAGs naturally encode precedence or dependency relationships.

Def: A topological ordering of a directed graph G

is an ordering of its nodes $v_1, v_2, \ldots, v_n$ such that for each (v_i, v_j), $i < j$. Intuitively, all edges in the ordering point "forward".



(3.18) Proposition: G has a topo. order $\Rightarrow$ G is a DAG

Proof: Suppose not. Let the topo. ordering be $v_1, v_2, \dots, v_n$

and let there be some cycle C. Let v_i be the node in C with the lowest index and let v_j be the node in C just before v_i. Thus (v_j, v_i) is an edge. But, since v_i was the node in C with the lowest index, we must have $j > i$. This contradicts that $v_1, v_2, \dots, v_n$ is a topological ordering of G. ∎

Does the converse hold? We will show it does via a constructive proof (an algorithm).

(3.19) Claim: In every DAG G, there is a node with no incoming edges.

Proof: Assume not. Then, there must be a cycle ∎

This node (say v) can be safely placed at the beginning of a topological ordering. This is sufficient, with (3.19) and induction, to produce an algorithm.

Inductive Claim: Every DAG has a topological ordering

Base: DAG of size 1, 2 are trivial

Assume: This is true for all DAGs with n nodes.

Then: Given a DAG with $n+1$ nodes, we can find a vertex v with no incoming edges (by 3.19). We can place v first in our topological ordering, since any edges from v point "forward".
Further $G - \{v\}$ is a DAG, since <u>deleting</u> v cannot <u>create</u> cycles. Further, $G - \{v\}$ has n nodes, so we can apply the inductive hypothesis to obtain an order for $G - \{v\}$. The ordering for G then becomes $v, \text{ord}(G - \{v\})$.

(3.20) If G is a DAG then G has a topo. ordering.

<u>Alg</u>: Topo(G):

    Find $v \in G$ with no incoming edges
    return v + Topo($G - \{v\}$)

To make this $O(m+n)$ rather than $O(n^2)$, we keep an "active" array of size n. A node is "active" if it has not yet been deleted. Also, for each node, maintain

(1) # of incoming edges to u from "active" nodes
(2) set S of "active" nodes that have no incoming edges from other "active" nodes.

- Then, algo selects node from S, deletes it, and updates neighbors
- Spends at most constant work per-edge during the algo.

Kahn's algo for topological sorting (wiki)

```
Topo(G):
    L = []
    S = {u | u has no incoming edges}

    while S is not empty:
        remove node x from S
        L.append(x)
        for each outgoing edge (x, y) of x:
            remove (x, y) from E
            if y has no incoming edges:
                S = S U {y}
            End if
        End for
    End while

    if edges remain in G:
        return None    (no valid topo. ord exists)
    else:
        return L
```