6 Connectivity in Digraphs, DFS, Topological Sorting

6.1 Continuation of Proof from Lecture 5

Theorem 6.1
A connected graph with BFS tree T is bipartite if and only if there is no non-tree edge joining vertices in the same layer of T.

Proof. ($\Longleftarrow$) Consider the bipartition $A = L_0 \cup L_2 \cup \cdots$ and $B = L_1 \cup L_3 \cup \cdots$.
This shows the graph is bipartite since all non-tree edges join vertices whose layers differ by 1. (Also note that tree edges always join vertices in adjacent layers)

($\Longrightarrow$) Suppose that G has a non-tree edges $\{u, v\}$ with u and v in the same layer.

Suppose their nearest common ancestor is m layers higher.
Then there is a path from u to v in the BFS tree of length $2m$ going through their nearest common ancestor.

Adding edge $\{u, v\}$ gives a cycle of length $2m + 1$
But a cycle of odd length can not be bipartite: indexing the adjacent vertices by $1, 2, \cdots, 2m + 1$, odd and even vertices must be on opposite sides, but the edge between vertices 1 and $2m + 1$ joins vertices on the same side. □

6.2 Connectivity in directed graphs

One approach to determining whether a directed graph is connected or not is to perform BFS on every vertex, with a time complexity of $O(n(m + n))$, which is a wasteful algorithm.

Lemma 6.2
If u and v are mutually reachable and v and w are mutually reacahble, then u and w are mutually reachable.

Proof. We can go from u to w by going from u to v to w.
Similarly, we can go from w to u. □

So to tell if a graph is strongly connected, we can fix any vertex s, and construct

- BFS starting from s
- BFS starting from s with the direction of all edges being reversed.

The graph is strongly connected if and only if both searches reach every node in the graph.

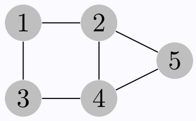
6.3 Depth-first search

Main idea: search recursively, keeping track of where you've been.

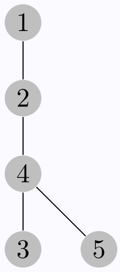
```
DFS(s):
    Let T be the tree with one vertex, s, with pr[s] = Null.
    DFSVisit(s)

DFSVisit(u):
    Mark u as explored
    For each edge {u, v}
        If v is not marked explored then
            Add vertex v and edge {u, v} to T
            Set pr[v] = u
            DFSVisit(v)
        Endif
    Endfor
```

Example 6.3



DFS(1):



6.4 Topological sorting

Recall that a DAG is a digraph with no directed cycles.

Example: A boolean circuit. There maybe multiple ways to compute the values in a circuit, but there are some constraints on the order of operations - you need all of the operations required for the inputs of a gate to be completed before evaluating the result of the gate.

Definition 6.4
A **topological ordering** is an order in which we can perform the operations sequentially, so that all required inputs are available when an operation is performed.

Lemma 6.5
If G has a topological ordering, then G is a DAG.

Proof. By contradiction, suppose G has a topological ordering $v1, v_2, \cdots, v_n$ and a directed cycle C.

Let v_i be the lowest-index vertex on C, and let v_j be the vertex just before v_i on C.
Then (v_j, v_i) is an edge with $i < j$, but we must have $j < i$ in a topological ordering since v_j, v_i is an edge. $\square$

Lemma 6.6
Every DAG has a vertex with indegree 0.

Proof. Contrapositive. If every vertex has positive indegree, then there is a directed cycle.

Start from any vertex. Walk along some edge in the backwards direction (possible since the indegree is positive). After at most $n + 1$ steps of this process, we must have visited some vertex twice, which means we have a directed cycle. $\square$

Lemma 6.7
Every DAG has a topological ordering.

Proof. By induction.

A 1-vertex graph is a DAG.
Suppose the claim holds for any DAG with at most n vertices.
Given an $(n+1)$ vertex DAG, find an indegree-0 vertex and delete it (and all associated edges).
The resulting graph is an n-vertex DAG since deletion can't create a cycle.
So the claim follows by induction. □

```
TopoSort(G):
    let S be an empty set
    for all vertices v
        if v has indegree 0, add v to S
        set count[v] = indeg(v)
    endfor
    while S is nonempty do
        remove a vertex v from S
        output v
        for each vertex u that v points to
            remove edge (v, u) from the graph
            decrement count[u]
            if count[u] = 0 then add u to S
        endfor
    endwhile
    if the graph is nonempty then
        return error "graph is not a DAG"
    endif
```

Running time: $O(n)$ initialization, for loop takes time $O((\operatorname{outdeg} v) + 1)$
So, $O(n + \sum_v (\operatorname{outdeg} v + 1)) = O(n+m)$