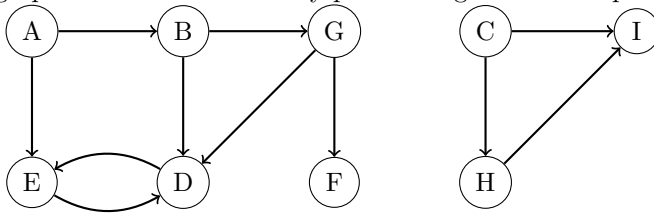


Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Graph Basics

For part (a), refer to the figure below. For parts (b) and (c), please prove only for simple graphs; that is, graphs that do not have any parallel edges or self-loops.



- (a) Run DFS at node A, trying to visit nodes alphabetically (e.g. given a choice between nodes D and F, visit D first).
- List the nodes in the order you visit them (so each node should appear in the ordering exactly once).
 - List each node with its pre- and post-number. The numbering starts from 1 and ends at 18.
 - Label each edge as **T**ree, **B**ack, **F**orward or **C**ross.
- (b) Let $|E|$ be the number of edges in a simple graph and $|V|$ be the number of vertices. Show that $|E|$ is in $O(|V|^2)$.
- (c) For each vertex v_i , let d_i be the *degree*- the number of edges incident to it. Show that $\sum d_i$ must be even.

Solution:

- (a) Ordering: ABDEGFCHI

nodes	pre-visit	post-visit
A	1	12
B	2	11
D	3	6
E	4	5
G	7	10
F	8	9
C	13	18
H	14	17
I	15	16

Tree: AB, BD, DE, BG, GF, CH, HI

Back: ED

Forward: AE, CI

Cross: GD

- (b) The maximum number of edges a graph can have is when every vertex is connected to every other vertex. This describes the ‘complete’ graph and has $\binom{V}{2} = \frac{V(V-1)}{2} \in O(V^2)$ edges.

- (c) Each edge in the graph belongs to precisely 2 vertices. Thus, the total number of edges is $\frac{\sum d_i}{2}$. Since there are an integer number of edges, $\sum d_i$ must be even.

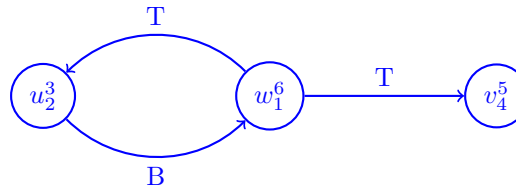
2 Short Answer

For each of the following, either prove the statement is true or give a counterexample to show it is false.

- (a) If (u, v) is an edge in an undirected graph and during DFS, $\text{post}(v) < \text{post}(u)$, then u is an ancestor of v in the DFS tree.
- (b) In a directed graph, if there is a path from u to v and $\text{pre}(u) < \text{pre}(v)$ then u is an ancestor of v in the DFS tree.
- (c) In any connected undirected graph G there is a vertex whose removal leaves G connected.

Solution:

- (a) True. There are two possible cases: $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$ or $\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$. In the first case, u is an ancestor of v . In the second case, v was popped off the stack without looking at u . However, since there is an edge between them and we look at all neighbors of v , this cannot happen.



- (b) False. Consider the following case:
- (c) True. Remove a leaf of a DFS tree of the graph.

3 Semiconnected DAG

A directed acyclic graph G is *semiconnected* if for any two vertices A and B , there is either a path from A to B or a path from B to A . Show that G is semiconnected if and only if there is a directed path that visits all of the vertices of G .

Solution: First, we show that the existence of a directed path p that visits all vertices implies that G is semiconnected. For any two vertices A and B , consider the subpath of p between A and B . If A appears before B in p , then this subpath will go from A to B . Otherwise, it will go from B to A . In either case, A and B are semiconnected for all pairs of vertices (A, B) in G .

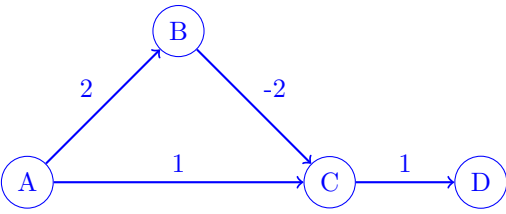
Now we show that if G is semiconnected, then there is a directed path that visits all of the vertices. Consider a topological ordering v_1, v_2, \dots, v_n of the vertices in G . For any pair of consecutive vertices v_i, v_{i+1} , we know that there is a path from v_i to v_{i+1} or from v_{i+1} to v_i by semiconnectedness. But topological orderings do not have any edges from later vertices to earlier vertices. Therefore, there is a path from v_i to v_{i+1} in G . This path cannot visit any other vertices in G because the path cannot travel from later vertices to earlier vertices in the topological ordering. Therefore, the path from v_i to v_{i+1} must be a single edge from v_i to v_{i+1} . This edge exists for any consecutive pair of vertices in the topological ordering, so there is a path from v_1 to v_n that visits all vertices of G .

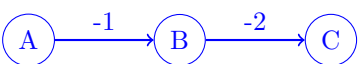
4 Dijkstra's Algorithm Fails on Negative Edges

Draw a graph with five vertices or fewer, and indicate the source where Dijkstra's algorithm will be started from.

1. Draw a graph with no negative cycles for which Dijkstra's algorithm produces the wrong answer.
2. Draw a graph with at least two negative weight edge for which Dijkstra's algorithm produces the correct answer.

Solution:

1. Here's one example:
 

Dijkstra's algorithm from source A will give the distance to D as 2 rather than 1, because it visits C before B .
2. Dijkstra's algorithm always works on directed paths. For example:
 

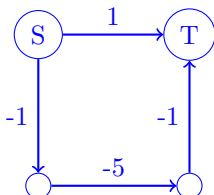
5 Fixing Dijkstra's Algorithm with Negative Weights

Dijkstra's algorithm doesn't work on graphs with negative edge weights. Here is one attempt to fix it:

1. Add a large number M to every edge so that there are no negative weights left.
2. Run Dijkstra's to find the shortest path in the new graph.
3. Return the path found by Dijkstra's, but with the old edge weights (i.e. subtract M from the weight of each edge).

Show that this algorithm doesn't work by finding a graph for which it must give the wrong answer.

Solution: The above algorithm doesn't work when the actual shortest path has more edges than other potential shortest paths. In this case, the paths with more edges have their weights increased more than the paths with fewer edges since M is added per edge and not per path. We can see this in the following counterexample:



The shortest path is “down-right-up” (weight -7). After adding $M = 5$ to each edge, we increase the actual shortest path by 15. The path “right” only increases by 5 and so the algorithm returns this path as the shortest path.

6 Updating Labels

You are given a tree $T = (V, E)$ with a designated root node r , and a non-negative integer label $l(v)$. If $l(v) = k$, we wish to relabel v , such that $l_{\text{new}}(v)$ is equal to $l(w)$, where w is the k th ancestor of v in the tree. We follow the convention that the root node, r , is its own parent. Give a linear time algorithm to compute the new label, $l_{\text{new}}(v)$ for each v in V .

Slightly more formally, the *parent* of any $v \neq r$, is defined to be the node adjacent to v in the path from r to v . By convention, $p(r) = r$. For $k > 1$, define $p^k(v) = p^{k-1}(p(v))$ and $p^1(v) = p(v)$ (so p^k is the k th ancestor of v). Each vertex v of the tree has an associated non-negative integer label $l(v)$. We want to find a linear-time algorithm to update the labels of all vertices in T according to the following rule: $l_{\text{new}}(v) = l(p^{l(v)}(v))$.

Solution:

Main Idea When we implement DFS with a stack, the stack at any given moment will always contain all the ancestors of the current node we're visiting. We want to maintain the labels of the relevant vertices currently on the stack, in a separate array. To ensure that our array only contains vertices on our current path down the DFS tree, we'll only add a vertex to our array (at index equal to the current depth) when we've actually visited it once (not when we first dd it to the stack). Since a path can have at most n vertices, the length of this array is at most n . Once we've processed all the children of a node, we can index into the array and set its label equal to the index of its k th ancestor. Notice that if we relabel the vertex before processing its children, we overwrite a label that the children of the vertex could depend on.

Runtime Analysis Since we add only a constant number of operations at each step of DFS, the algorithm is still linear time.