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 Traversal

(a) Recall that given a DFS tree, we can classify edges into one of four types:

- Tree edges are edges in the DFS tree,
- Back edges are edges \((u,v)\) not in the DFS tree where \(v\) is the ancestor of \(u\) in the DFS tree
- Forward edges are edges \((u,v)\) not in the DFS tree where \(u\) is the ancestor of \(v\) in the DFS tree
- Cross edges are edges \((u,v)\) not in the DFS tree where \(u\) is not the ancestor of \(v\), nor is \(v\) the ancestor of \(u\).

For the directed graph above, perform DFS starting from vertex \(A\), breaking ties alphabetically. As you go, label each node with its pre- and post-number, and mark each edge as Tree, Back, Forward or Cross.

(b) What are the strongly connected components of the above graph?

(c) Draw the DAG of the strongly connected components of the graph.

**Solution:**
(a) 

(b) 

\{A\}, \{B\}, \{E\}, \{G, H, I\}, \{C, J, F, D\}

(c)
2 BFS Intro

In this problem we will consider the shortest path problem: Given a graph $G(V, E)$, find the length of the shortest path from $s$ to every vertex $v$ in $V$. For an unweighted graph, the length of a path is the number of edges in the path. We can do this using the breadth-first search (BFS) algorithm, which we will see again in lecture this week.

BFS can be implemented just like the depth-first search (DFS) algorithm, but using a queue instead of a stack. Below is pseudo-code for another implementation of BFS, which computes for each vertex the shortest path from $s$ to every vertex.

```
Input: A graph $G(V, E)$, starting vertex $s$
2: for all $v \in V$ do
3: \hspace{1em} \textit{visited}(v) = \text{False}
4: $L_0 = \{s\}$
5: for $i$ from 0 to $n - 1$ do
6: \hspace{1em} $L_{i+1} = \{\}$
7: \hspace{2em} for $u \in L_i$ do
8: \hspace{3em} for $(u, v) \in E$ do
9: \hspace{4em} if \textit{visited}(v) = \text{False} then
10: \hspace{5em} $L_{i+1}$.add($v$)
11: \hspace{1em} \textit{visited}(v) = \text{True}
```

In other words, we start with $L_0 = \{s\}$, and then for each $i$, we set $L_{i+1}$ to be all neighbors of vertices in $L_i$ that we haven’t already added to a previous $L_i$.

(a) Prove that BFS computes the correct value of $L_i$ for all $i$ (Hint: Use induction to show that for all $i$, $L_i$ contains all vertices distance $i$ from $s$, and only contains these vertices).

\textbf{Solution:} We claim that before we start iteration $i$ of the for loop: (1) all vertices at exactly distance $i$ from $s$ are in $L_i$, and (2) All vertices at distance more than $i$ from $s$ have not been added to any $L_j$, $j \leq i$. We will prove this inductively holds for all $i$, which implies the algorithm is correct.

This holds for $i = 0$. Assume it holds for $i = k$. We will show it holds for $i = k + 1$. (1) holds for $i = k + 1$ because every vertex at distance $k + 1$ is adjacent to some vertex at distance $k$, and thus by inductive hypothesis (1) gets added to $L_{k+1}$ in iteration $k$. (2) holds because no vertex at distance $k + 2$ or more can be adjacent to a vertex at distance $k$ or less, and so the only vertices added to any $L_{k+1}$ in iteration $k$ are those at distance exactly $k + 1$.

(b) Show that just like DFS, the above algorithm runs in $O(m + n)$ time, where $n$ is the number of nodes and $m$ is the number of edges.

\textbf{Solution:} Initializing \textit{visited} takes $O(n)$ time. Iteration $i$ of the for loop takes time $O(\sum_{v \in L_i} \delta(v))$, where $\delta(v)$ is the degree of $v$. Since no $v$ appears in more than one $L_i$, the overall time is $O(n + \sum_{v \in V} \delta(v)) = O(n + m)$.

(c) We might instead want to find the shortest weighted path from $s$ to each vertex. That is, each edge has weight $w_e$, and the length of a path is now the sum of weights of edges in the path. The above algorithm works when all $w_e = 1$, but can easily fail if some $w_e \neq 1$.

Fill in the blank to get an algorithm computing the shortest paths when $w_e$ are integers: We replace each edge $e$ in $G$ with \underline{___} to get a new graph $G'$, then run BFS on $G'$ starting from $s$. Justify your answer.

\textbf{Solution:} A path of $w_e$ unweighted edges. See the below figure for e.g. a directed graph:
(d) What is the runtime of this algorithm as a function of the weights $w_e$? How many bits does it take to write down all $w_e$? Is this algorithm’s runtime a polynomial in the input size?

**Solution:** The runtime is $O(\sum_{e \in E} w_e)$, since $G'$. The number of bits needed is $\Theta(\sum_{e \in E} \log w_e)$. So even though this algorithm’s runtime looks like a polynomial, it takes time exponential in the input size when some $w_e$ are large.
3 Counting Shortest Paths

Given an undirected unweighted graph $G$ and a vertex $s$, let $p(v)$ be the number of distinct shortest paths from $s$ to $v$. We will use the convention that $p(s) = 1$ in this problem. Give an $O(|V| + |E|)$-time algorithm to compute $p(v) \mod 1337$ for all vertices. Only the main idea and runtime analysis are needed.

(Hint: For any $v$, how can we express $p(v)$ as a function of other $p(u)$?)

Note: As a secondary question, you should ask yourself whether the runtime would remain the same if we were computing $p(v)$ rather than $p(v) \mod 1337$.

**Solution:** Main idea If $v$ is distance $d > 0$ from $s$, the first $d - 1$ edges in any shortest path to $v$ will be a shortest path to a neighbor of $v$ distance $d - 1$ from $s$. Furthermore, this mapping from shortest paths to $v$ and shortest paths to neighbors of $v$ is bijective. So letting $N(v)$ be the set of neighbors of $v$ at distance $d - 1$, we get that $p(v) = \sum_{u \in N(v)} p(u)$.

Our algorithm is now: use BFS to compute all distances from $s$. Next, we set $p(s) = 1$, then for the remaining vertices in increasing distance order, we can compute $p(v) \mod 1337 = \sum_{u \in N(v)} p(u) \mod 1337$. Since we look at vertices in increasing distance order, all $u$ in $N(v)$ have $p(u)$ computed already.

**Runtime analysis** BFS takes $O(|V| + |E|)$ time. Computing $p(v) \mod 1337$ from its neighbors takes time $O(\deg(v))$, so the total time to compute all $p(v) \mod 1337$ is $O(|E|)$.

If we did not have the $\mod 1337$, and we instead wanted $p(v)$ exactly, we would have a higher runtime, as the number of length-$d$ paths to $v$ can be exponential in $d$ and so arithmetic on these numbers would not be constant time.

4 More Graph Proofs

Only prove the following statements for simple graphs (i.e. graphs that do not have any parallel edges or self-loops).

(a) An undirected graph $G$ is called **bipartite** if we can separate its vertices into two subsets $A$ and $B$, such that every edge in $G$ must cross between $A$ and $B$. Show that a graph is bipartite if and only if it has no odd cycles.

Hint: Consider a spanning tree of the graph, which is a subset of the graph’s edges which forms a tree on all of its vertices.

(b) 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:**

(a) Without loss of generality, assume that the graph is connected. Otherwise, apply the following proof to each connected component.

First, assume that the graph has no odd cycles. Do a BFS from some root vertex $r$. Color a vertex red if it has an odd distance from $r$ and black if it has an even distance from $r$. We now show that the red and black vertices form a bipartition of the graph. Suppose for the sake of contradiction that there are two adjacent vertices $u$ and $v$ that are both red. Let $x$ be the least common ancestor of $u$ and $v$ in the tree. Since $u$ and $v$ are both red, the $ux$ and $xv$ paths both have even lengths or both have odd lengths. In particular, concatenating the paths shows that there is an even-length path that connects $u$ to $v$. Adding the $uv$ edge to this path creates an odd cycle, which is a contradiction.

To complete the proof, we now show that a bipartite graph can never have an odd cycle. Suppose there is an odd cycle; we can try to color it. WLOG assume that the first vertex of the coloring
is blue. This uniquely determines the color of the next vertex, and so on. So the odd vertices on
this path are blue and the even vertices are red. The last vertex cannot be colored either red or
blue, since it is a neighbor of the first vertex which is odd, and the previous vertex which is even.
Hence, no odd cycle can exist.

(b) 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 \( v_1, v_2, \ldots, v_n \) 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 \( G \).

5 Preorder, Postorder

Suppose we just ran DFS on a directed (not necessarily strongly connected) graph \( G \) starting from
vertex \( r \), and have the pre-visit and post-visit numbers \( \text{pre}(v), \text{post}(v) \) for every vertex. We now delete vertex \( r \) and all edges adjacent to it to get a new graph \( G' \). Given just the arrays \( \text{pre}(v), \text{post}(v) \),
describe how to modify them to arrive at new arrays \( \text{pre}'(v), \text{post}'(v) \) such that \( \text{pre}'(v), \text{post}'(v) \) are a
valid pre-visit and post-visit ordering for some DFS of \( G' \).

**Solution:** For all \( v \) such that \( \text{pre}(r) < \text{pre}(v) < \text{post}(v) < \text{post}(r) \), set \( \text{pre}'(v) = \text{pre}(v) - 1, \text{post}'(v) = \text{post}(v) - 1 \). For all other \( v \) in \( G' \), \( \text{pre}'(v) = \text{pre}(v) - 2, \text{post}'(v) = \text{post}(v) - 2 \).

One valid DFS on \( G' \) is: Run DFS, whenever we need to pick a new vertex to explore from, or
whenever we choose a neighbor of the “current vertex” to explore, choose the unvisited vertex with
the smallest value of \( \text{pre}(v) \). This will visit all vertices in \( G' \) in the same order as the DFS on \( G \).
For example, notice that vertices adjacent to \( r \) have lower pre-visit numbers than vertices that can’t
be reached from \( r \). So this DFS on \( G' \) will explore the vertices reachable from \( r \) in \( G \) first, and then
vertices not reachable from \( r \), just like the DFS on \( G \).

For vertices with \( \text{pre}(r) < \text{pre}(v) < \text{post}(v) < \text{post}(r) \), their pre/post-visit number decreases by
1 in this DFS since we no long pre-visit \( r \) before them. For all other vertices, their pre/post-visit
number decreases by 2 since we no long pre-visit or post-visit \( r \) before them.