Lecture 10: Depth-First Search

Previously

- Graph definitions (directed/undirected, simple, neighbors, degree)
- Graph representations (Set mapping vertices to adjacency lists)
- Paths and simple paths, path length, distance, shortest path
- Graph Path Problems
  - Single Pair Reachability \( (G, s, t) \)
  - Single Source Reachability \( (G, s) \)
  - Single Pair Shortest Path \( (G, s, t) \)
  - Single Source Shortest Paths \( (G, s) \) (SSSP)
- Breadth-First Search (BFS)
  - algorithm that solves Single Source Shortest Paths
  - with appropriate data structures, runs in \( O(|V| + |E|) \) time (linear in input size)

Examples

\[ G_1 \]
\[ G_2 \]
Depth-First Search (DFS)

- Searches a graph from a vertex \( s \), similar to BFS
- Solves Single Source Reachability, not SSSP. Useful for solving other problems (later!)
- Return (not necessarily shortest) parent tree of parent pointers back to \( s \)

**Idea!** Visit outgoing adjacencies recursively, but never revisit a vertex
- i.e., follow any path until you get stuck, backtrack until finding an unexplored path to explore
- \( P(s) = \text{None} \), then run \( \text{visit}(s) \), where
- \( \text{visit}(u) : \)
  - for every \( v \in \text{Adj}(u) \) that does not appear in \( P \):
    * set \( P(v) = u \) and recursively call \( \text{visit}(v) \)
  - (DFS finishes visiting vertex \( u \), for use later!)

**Example:** Run DFS on \( G_1 \) and/or \( G_2 \) from \( a \)

**Correctness**

- **Claim:** DFS visits \( v \) and correctly sets \( P(v) \) for every vertex \( v \) reachable from \( s \)
- **Proof:** induct on \( k \), for claim on only vertices within distance \( k \) from \( s \)
  - Base case (\( k = 0 \)): \( P(s) \) is set correctly for \( s \) and \( s \) is visited
  - Inductive step: Consider vertex \( v \) with \( \delta(s,v) = k' + 1 \)
  - Consider vertex \( u \), the second to last vertex on some shortest path from \( s \) to \( v \)
  - By induction, since \( \delta(s,u) = k' \), DFS visits \( u \) and sets \( P(u) \) correctly
  - While visiting \( u \), DFS considers \( v \in \text{Adj}(u) \)
  - Either \( v \) is in \( P \), so has already been visited, or \( v \) will be visited while visiting \( u \)
  - In either case, \( v \) will be visited by DFS and will be added correctly to \( P \)

**Running Time**

- Algorithm visits each vertex \( u \) at most once and spends \( O(1) \) time for each \( v \in \text{Adj}(u) \)
- Work upper bounded by \( O(1) \times \sum_{u \in V} \deg(u) = O(|E|) \)
- Unlike BFS, not returning a distance for each vertex, so DFS runs in \( O(|E|) \) time
**Full-BFS and Full-DFS**

- Suppose want to explore entire graph, not just vertices reachable from one vertex
- **Idea!** Repeat a graph search algorithm $A$ on any unvisited vertex

  - Repeat the following until all vertices have been visited:
    - Choose an arbitrary unvisited vertex $s$, use $A$ to explore all vertices reachable from $s$

- We call this algorithm **Full-$A$**, specifically Full-BFS or Full-DFS if $A$ is BFS or DFS
- Visits every vertex once, so both Full-BFS and Full-DFS run in $O(|V| + |E|)$ time
- **Example:** Run Full-DFS/Full-BFS on $G_1$ and/or $G_2$

![Graphs $G_1$ and $G_2$]

**Graph Connectivity**

- An undirected graph is **connected** if there is a path connecting every pair of vertices
- In a directed graph, vertex $u$ may be reachable from $v$, but $v$ may not be reachable from $u$
- Connectivity is more complicated for directed graphs (we won’t discuss in this class)
- **Connectivity**($G$): is undirected graph $G$ connected?
- **ConnectedComponents**($G$): given undirected graph $G = (V, E)$, return partition of $V$ into subsets $V_i \subseteq V$ (connected components) where each $V_i$ is connected in $G$ and there are no edges between vertices from different connected components
- Consider a graph algorithm $A$ that solves Single Source Reachability
- **Claim:** $A$ can be used to solve Connected Components
- **Proof:** Run Full-$A$. For each run of $A$, put visited vertices in a connected component
Topological Sort

- A **Directed Acyclic Graph (DAG)** is a directed graph that contains no directed cycle.
- A **Topological Order** of a graph \( G = (V, E) \) is an ordering \( f \) on the vertices such that: every edge \( (u, v) \in E \) satisfies \( f(u) < f(v) \).
- **Exercise:** Prove that a directed graph admits a topological ordering if and only if it is a DAG.
- How to find a topological order?
- A **Finishing Order** is the order in which a Full-DFS finishes visiting each vertex in \( G \)
- **Claim:** If \( G = (V, E) \) is a DAG, the reverse of a finishing order is a topological order
- **Proof:** Need to prove, for every edge \( (u, v) \in E \) that \( u \) is ordered before \( v \), i.e., the visit to \( v \) finishes before visiting \( u \). Two cases:
  - If \( u \) visited before \( v \):
    * Before visit to \( u \) finishes, will visit \( v \) (via \( (u, v) \) or otherwise)
    * Thus the visit to \( v \) finishes before visiting \( u \)
  - If \( v \) visited before \( u \):
    * \( u \) can’t be reached from \( v \) since graph is acyclic
    * Thus the visit to \( v \) finishes before visiting \( u \)  

Cycle Detection

- Full-DFS will find a topological order if a graph \( G = (V, E) \) is acyclic
- If reverse finishing order for Full-DFS is not a topological order, then \( G \) must contain a cycle
- Check if \( G \) is acyclic: for each edge \( (u, v) \), check if \( v \) is before \( u \) in reverse finishing order
- Can be done in \( O(|E|) \) time via a hash table or direct access array
- To return such a cycle, maintain the set of **ancestors** along the path back to \( s \) in Full-DFS
- **Claim:** If \( G \) contains a cycle, Full-DFS will traverse an edge from \( v \) to an ancestor of \( v \).
- **Proof:** Consider a cycle \( (v_0, v_1, \ldots, v_k, v_0) \) in \( G \)
  - Without loss of generality, let \( v_0 \) be the first vertex visited by Full-DFS on the cycle
  - For each \( v_i \), before visit to \( v_i \) finishes, will visit \( v_{i+1} \) and finish
  - Will consider edge \( (v_i, v_{i+1}) \), and if \( v_{i+1} \) has not been visited, it will be visited now
  - Thus, before visit to \( v_0 \) finishes, will visit \( v_k \) (for the first time, by \( v_0 \) assumption)
  - So, before visit to \( v_k \) finishes, will consider \( (v_k, v_0) \), where \( v_0 \) is an ancestor of \( v_k \)  

