DS 320 Algorithms for Data Science **Lecture #4**
Spring 2023 Prof. Kira Goldner

Depth-First Search

If we want to explore a graph, determine whether two nodes are connected, or determine some properties regarding the ordered structure of a directed graph, we use *graph search* algorithms. The two most basic graph search algorithms are *depth-first* and *breadth-first*. Depth-First Search is the algorithm that shoots as far away from a node possible to see if it results in a successful path, and only turns around if it dead ends.

For now, we ignore the calls previsit(v) and postvisit(v).

Algorithm 1 DFS(s, G)

Input: Graph $G = (V, E)$ and vertex v.
for each $v \in V$ **do**
    v is unexplored
    set v's parent to Null
end for
search(s, G)
return forest F formed by parents

Algorithm 2 search(v, G)

Input: Graph $G = (V, E)$ and vertex v.
mark v as explored
previsit(v)
for $(v, w) \in E$ **do**
    if w is unexplored **then**
        set w's parent to v
        search(w, G)
    end if
end for
postvisit(v)

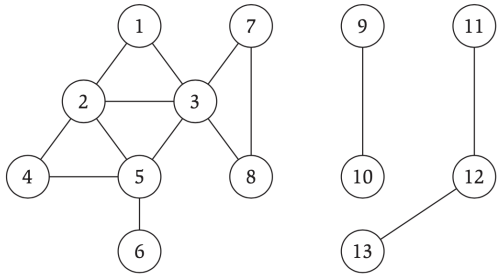

Figure 1: Example graph G. From Kleinberg Tardos.

Exercise.

1. Consider the pseudocode when called on the above example, that is, what happens when we run DFS(1, G) where G is the graph above? Draw the DFS forest as the graph is explored.

2. DFS implicitly uses a stack. Draw the stages of the stack as it runs on the example graph.

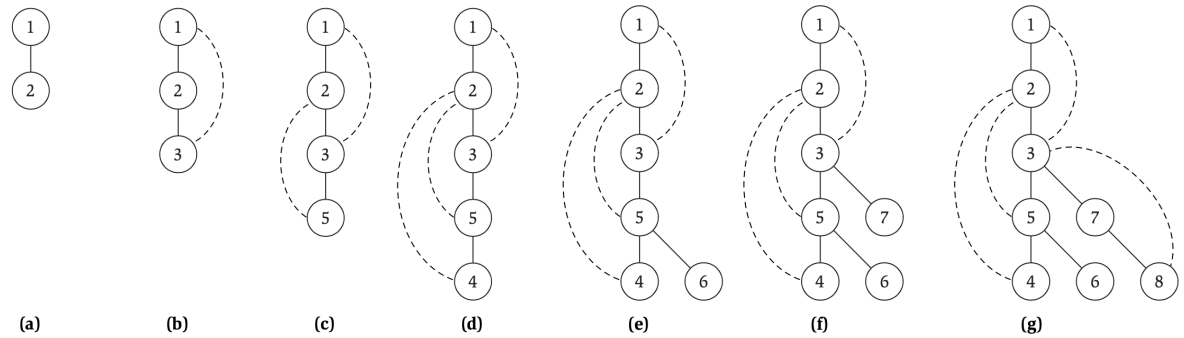

Figure 2: The DFS tree is shown in solid edges as constructed in stages by the above algorithm. The dashed edges are the edges that do not belong to the DFS tree but do belong to G. From Kleinberg Tardos.

Preorder, Postorder, and Tree vs. Non-Tree Edges

Preorder and Postorder: Now, we'll define what happens in *previsit*() and *postvisit*(). Our idea is to track when nodes are pushed onto the stack and when they are popped off, as this order winds up giving us important properties. Formally, we have the following definitions.

Definition 1. The method previst(v) assigns a vertex v's *preorder* time, that is, i if it is pushed onto the stack at time i.

Definition 2. The method postvist(v) assigns a vertex v's *postorder* time, that is, i if it is popped off of the stack at time i.

That is, we set "clock" to 0 when vertices are marked unexplored, and then $previsit(v)$ increments the clock by 1 and sets $v.preorder = clock$ and $postvisit(v)$ increments the clock by 1 and sets $v.postorder = clock$.

It should be clear then that $[preorder(u), postorder(u)]$ and $[preorder(v), postorder(v)]$ are either disjoint or nester. Moreover, the former contains the latter if and only if DFS(v) is called during the execution of DFS(u), or equivalently, if and only if u is an ancestor of v in the DFS tree.

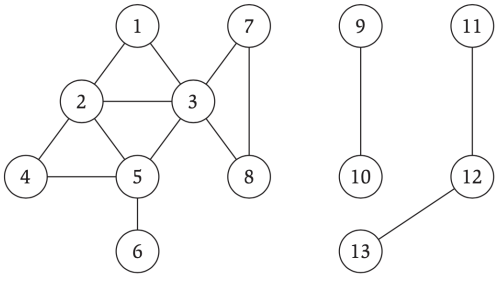

Figure 3: Example graph G. From Kleinberg Tardos.

We can partition the edges of G into four types:

- *Tree edges* are edges in the depth-first forest F. Edge (u, v) is a tree edge if was first discovered by exploring edge (u, v).
- *Back edges* are those edges (u, v) connecting a vertex u to an ancestor in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges. In undirected graphs, there is no distinction between back edges and forward edges.
- *Forward edges* are those nontree edges (u, v) connecting a vertex u to a descendant in a depth-first tree.
- *Cross edges* are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees. There are no cross edges in undirected graphs. (Can you explain why?)

Claim 1. If $(u, v) \in E$ then $postorder(u) < postorder(v) \iff (u, v)$ is a back edge.

Proof. If $(u, v) \in E$, then before u is popped off of the stack, we *could* have pushed v onto the stack via (u, v). ($\Leftarrow$) But if (u, v) is a backedge and not a tree edge, it must already be on the stack underneath u, and thus will pop after u. ($\Rightarrow$) Or, if $postorder(u) < postorder(v)$, (u, v) cannot be a tree edge or we'd push and pop v after pushing u, and thus we'd have $postorder(v) < postorder(u)$, a contradiction. Then (u, v) must be a backedge. □

Claim 2. $G = (V, E)$ has a cycle $\iff$ the DFS tree of G yields a back edge.

Proof. If (u, v) is a back edge, then (u, v) together with the path from v to u in the DFS forest form a cycle.
Conversely, for any cycle in $G = (V, E)$, consider the vertex assigned the smallest postorder number. Then the edge leaving this vertex in the cycle must be a back edge by Claim 1, since it goes from a lower postorder number to a higher postorder number. □

Application: Topological Sort

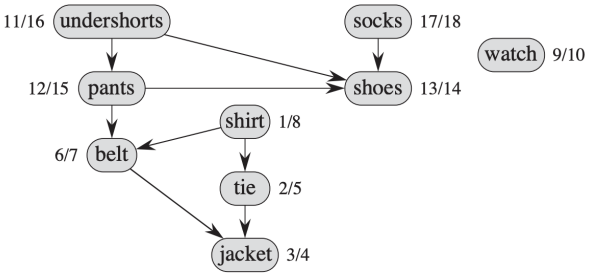

Figure 4: Top sort example graph from CLRS.

Theorem 1. *G has a topological order $\iff$ G is a DAG.*

Proof. ($\Rightarrow$) Topological ordering means only edges $(i, j) \in E$ where $i < j$. Consider the smallest i in the cycle. There exists an edge (j, i) in the cycle for $j > i$. Contradiction.

($\Leftarrow$) If there's a DAG, this implies that there exists a node with no incoming edges. Otherwise, one could backtrack, and after n steps, would find a cycle. □

Topological Sort Algorithm:

- Naive algorithm: Recursively remove a node with no incoming edges. $T(n) = O(n^2)$.
- Or, run DFS to assign postorder times, and then sort the DFS forest by decreasing postorder. $T(n) = O(n + m)$.

Theorem 2. *If the tasks are scheduled by decreasing postorder number, then all precedence constraints are satisfied.*

Proof. If G is acyclic then the DFS tree of G produces no back edges by Claim 2. Therefore by Claim 1, $(u, v) \in G$ implies $postorder(u) > postorder(v)$. So, if we process the tasks in decreasing order by postorder number, when task v is processed, all tasks with precedence constraints into v (and therefore higher postorder numbers) must already have been processed. □