DS 320 Algorithms for Data Science **Lecture #5 Worksheet**
Fall 2024 Profs. Krzysztof Onak and Aldo Pacchiano

Depth–First Search

Graph search algorithms: good for exploring a graph, determining whether two nodes are connected, determining some properties regarding the ordered structure of a directed graph, . . .

Depth–First Search: shoots as far away from a node as possible to see if it results in a successful path, and only turns around if it dead ends.

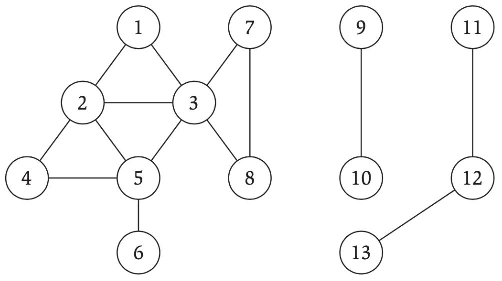
For now, ignore the calls previsit(v) and postvisit(v) while reading the code below.

Algorithm 1 DFS(G)

```
Input: graph $G = (V, E)$
for each $v \in V$ do
    $v$ is unexplored
    set $v$'s parent to Null
end for
for each $v \in V$ do
    if $v$ is unexplored then
        search($v, G$)
    end if
end for
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($v$)
```



Exercise:

1. Consider the pseudocode when called on the above example, that is, what happens when we run search(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 sample graph.

Running Time

What is the running time of DFS(G), assuming that previsit(...) and postvisit(...) do nothing or run in $O(1)$ time?

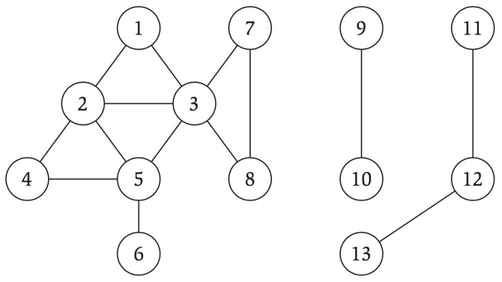
Preorder, Postorder, and Tree vs. Non-Tree Edges

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 previsit(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 postvisit(v) assigns a vertex v's *postorder* time, that is, i if it is popped off of the stack at time i.

Exercise: Assign preorder and postorder times to vertices in the graph below.



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 non-tree edges (u, v) connecting a vertex u to an ancestor in a given depth–first tree. We consider self-loops, which may occur in directed graphs, to be back edges.

  Note: If the graph is undirected and (u, v) is a tree edge—in which u is the parent and v is the child—then it is not considered a back edge when it is encountered processing v.
- *Forward edges* are those nontree edges (u, v) connecting a vertex u to a descendant in a depth–first tree.

  Note: In undirected graphs, if an edge is not classified as a tree edge, it will be both a back and forward edge, depending on the endpoint from which we look at it. (Exercise: Why? See also Claim 1 below.)
- *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.

Exercise: Look back at our sample DFS tree/forest and assign edges to the above categories.

Claim 1. The DFS tree (or forest) of an undirected graph does not have cross edges.

Claim 2. If $(u, v) \in E$ and is not a tree edge, then postorder$(u) <$ postorder$(v) \iff (u, v)$ is a back edge.

Claim 3. $G = (V, E)$ has a cycle $\iff$ the DFS tree of G yields a back edge.

Application: Topological Sort

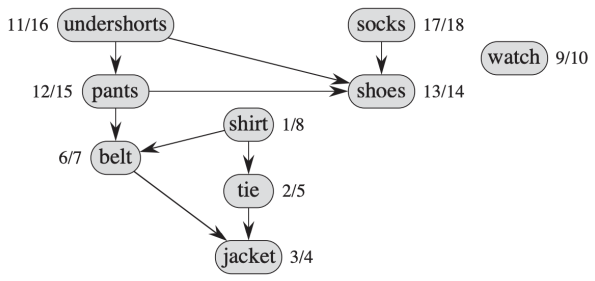


Figure 1: A sample graph that has a topological ordering (from CLRS).

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

Topological Sorting Algorithm:

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

Many Cool Applications of DFS

We are just listing some examples here, but among many other things, DFS can be used to design efficient algorithms for:

- Finding "bridges"/"cut edges": edges that when removed one of connected components breaks into two
- Finding "articulation points"/"cut vertices": vertices that when removed one of connected components breaks into two
- For connected graphs with no bridges: turning an undirected graph into a directed one by assigning a direction to every edge such that any vertex still reachable from any other one

---

Based on material developed by Prof. Kira Goldner