CS 430 Lecture 25 Activities

Opening Questions

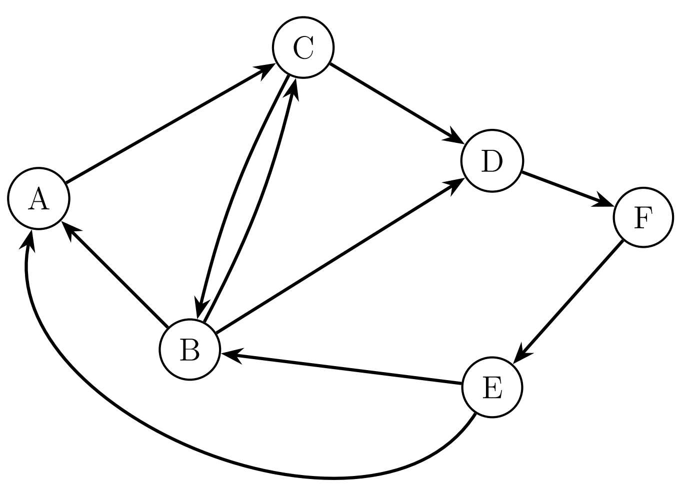
1. What is the runtime of breadth first search (if you restart the search from a new source if everything was not visited from the first source)? $O(|V| + |E|)$. Use a queue for all vertices that haven't been visited. Keep track of d-distance to edges and π-precescessor of which edge got us there.
2. Does a breadth-first search always reach all vertices? No.
3. How can you use a breadth-first search to find the shortest path (minimum number of edges) from a given source vertex to all other vertices? Use the d variable which holds the distance to edges from the source.
4. If you look at the predecessor edges which were used to connect to an unvisited vertex, what do these predecessor edges form? Is it unique for a graph? They yield a breadth-first tree.

Depth-First Search

https://www.reddit.com/r/dataisbeautiful/comments/7b7aa0/visualizing_the_depthfirst_search_recursive/?st=J9OUDR0O&sh=5b671c59

As we visit a vertex, we try to move to a new adjacent vertex that hasn't yet been visited, until there is nowhere else to go, then backtrack. Uses a stack and some way to mark a vertex as visited (white initially, gray when first visited and put in stack, black when out of stack), label a vertex with a counter for first time seen, and another counter for last time seen (we will see why later), and label a vertex with how its predecessor vertex was during the traversal.

1. Perform a depth-first search on this graph.



Algorithm 25.1 Depth-First Search

```
function DFS(G)
    for all vertex u ∈ V[G] do
        Color(u) ← WHITE
        π(u) ← NIL
    end for
    time ← 0
    for all vertex u ∈ V[G] do
        if Color(u) == WHITE then DFS-Visit(u)
        end if
    end for
end function
```

Algorithm 25.2 Depth First Search Visit

```
function DFS-Visit(u)
    Color(u) ← GRAY          ▷ White vertex u has just been discovered.
    time ← time + 1
    d(u) ← time              ▷ Debut
    for all v ∈ Adj(u) do    ▷ Explore edge (u, v).
        if Color(v) == WHITE then
            π(v) ← u
            DFS-Visit(v)
        end if
    end for
    Color(u) ← BLACK         ▷ Blacken u; it is finished.
    f(u) ← time ← time + 1   ▷ Finish
end function
```

2. What is the runtime for depth-first search (if you restart the search from a new source if everything was not visited from the first source)? $O(|V| + |E|)$, each edge and vertex is checked once before it's not allowed to be re-run again.

Another interesting property of depth-first search is that the search can be used to classify the edges of graph G based on how they are traversed.

- **Tree edges** are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
- **Back edges** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops, which may occur in directed graphs, are considered to be back edges.
- **Forward edges** are those non-tree edges (u, v) connecting a vertex u to a descendant v 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.

3. If a graph has no back edges when completing a depth-first search, what does that tell us about the graph? The graph has no cycles, which means it's acyclic.

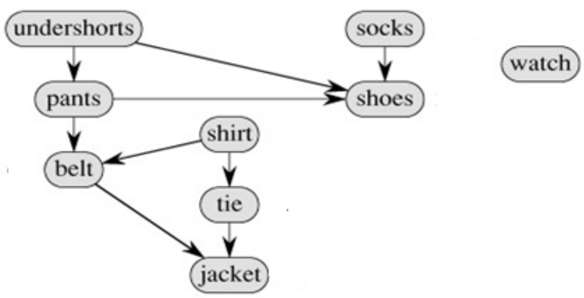
Demo of BFS/DFS: `https://www3.cs.stonybrook.edu/~skiena/combinatorica/animations/search.html`

Topological Sort (a DFS application)

Dependency order

- A topological sort of a directed acyclic graph[1], $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. (If the graph is not acyclic, then no linear ordering is possible.)
- A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges from left to right. Topological sorting is thus different from the usual kind of "sorting" studied earlier.
- Directed acyclic graphs are used in many applications to indicate precedence among events.
- A depth-first search can be used to perform a topological sort of a dag.

4. Perform a topological sort on this graph.



Algorithm 25.3 Topological Sort

function TOPOLOGICAL-SORT[a](G)
    DFS(G) to compute finishing times $\text{F}(v)$ for each vertex v.
    As each vertex is finished, insert it onto the front of a linked list.
    return the linked list of vertices
end function

[a]Start at any vertex, restart if necessary

5. Why does the topological sort work? What is its runtime? We're doing a depth-first traversal, so any time there's a dependency, we access the dependency before the item that depends on it. $O(|V| + |E|)$.

The Parenthesis Theorem

The parenthesis theorem tells us that, for two vertices $u, v \in V$, it cannot be the case the $d[u] < d[v] < f[u] < f[v]$; that is, the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are either disjoint or nested. This is a simple consequence of the depth-first nature of DFS. If the algorithm discovers u and then discovers v, it cannot later back out of u without backing out of v.

Strongly Connected Components (a DFS application)

A graph is said to be strongly connected if every vertex is reachable from every other vertex. The strongly connected components of an arbitrary directed graph from a partition into subgraphs that are themselves strongly connected. It is possible to test the strong connectivity of a graph, or to find its strongly connected components, in linear time.

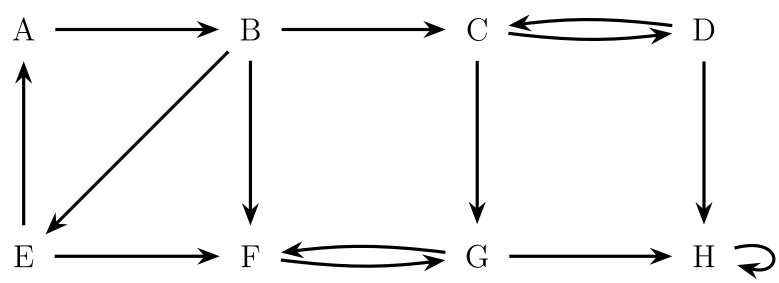
[1]sometimes known as a dag

Algorithm 25.4 Strongly Connected Components

1: **function** STRONGLY-CONNECTED-COMPONENTS(G)
2: Call DFS(G) to compute finishing times $f[u]$ for each vertex u[a] $O(|V| + |E|)$.
3: Compute G^T (the transpose of the graph)[b] $O(|E|)$.
4: Call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $f[u]$ (as computed in line 1) $O(|V| + |E|)$.
5: Output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component.
6: **end function**

[a] Restart if necessary.
[b] Same vertices, edges reversed.

6. Find the strongly connected components.



7. Discuss: G and G^T will have the same strongly connected components. Yes, they must have the same connected components. They will have the same edges and cycles, they're just traveled in the opposite order.

8. Discuss: The component with the latest finish time vertex will have no edges in the transpose to any other component.