

CS 430 Lecture 25 Activities

Opening Questions

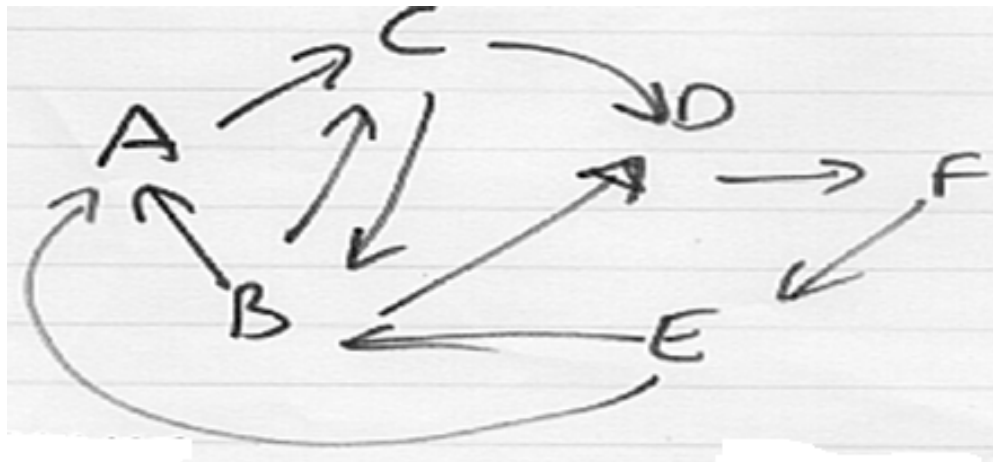
1. What is the runtime of breadth first search (if you restart the search from a new source if everything was not visited from the first source)?
2. Does a breadth-first search always reach all vertices?
3. How can you use a breadth-first search to find the shortest path (minimum number of edges) from a given source vertex to all other vertices?
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?

Depth-First Search

https://www.reddit.com/r/dataisbeautiful/comments/7b7aa0/visualizing_the_depthfirst_search_recursive/?st=J90UDR00&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

```

1: function DFS( $G$ )
2:   for all vertex  $u \in V[G]$  do
3:      $COLOR(u) \leftarrow WHITE$ 
4:      $\pi(u) \leftarrow NIL$ 
5:   end for
6:    $time \leftarrow 0$ 
7:   for all vertex  $u \in V[G]$  do
8:     if  $COLOR(u) == WHITE$ 
9:       then DFS-VISIT( $u$ )
10:    end if
11:  end for
12: end function

```

Algorithm 25.2 Depth First Search Visit

```

1: function DFS-VISIT( $u$ )
2:    $COLOR(u) \leftarrow GRAY$   $\triangleright$  White vertex  $u$  has just
   been discovered.
3:    $time \leftarrow time + 1$ 
4:    $D(u) \leftarrow time$ 
5:   for all  $v \in ADJ(u)$  do  $\triangleright$  Explore edge  $(u, v)$ .
6:     if  $COLOR(v) == WHITE$  then
7:        $\pi(v) \leftarrow u$ 
8:       DFS-VISIT( $v$ )
9:     end if
10:  end for
11:   $COLOR(u) \leftarrow BLACK$   $\triangleright$  Blacken  $u$ ; it is finished.
12:   $F(u) \leftarrow time \leftarrow time + 1$ 
13: 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)?

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?

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

Topological Sort (a DFS application)

- A topological sort of a directed acyclic graph¹, $G = (V, E)$ 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.)

¹sometimes known as a dag

- 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.

<pre> graph TD undershorts --> pants undershorts --> shoes pants --> belt shirt --> belt shirt --> tie tie --> jacket socks --> shoes watch </pre>	<hr/> Algorithm 25.3 Topological Sort <hr/> <pre> 1: function TOPOLOGICAL-SORT(G) 2: DFS(G) to compute finishing times $F(v)$ for each vertex v. 3: As each vertex is finished, insert it onto the front of a linked list. 4: return the linked list of vertices 5: end function </pre> <hr/>
---	---

5. Why does the topological sort work? What is its runtime?

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 form 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.

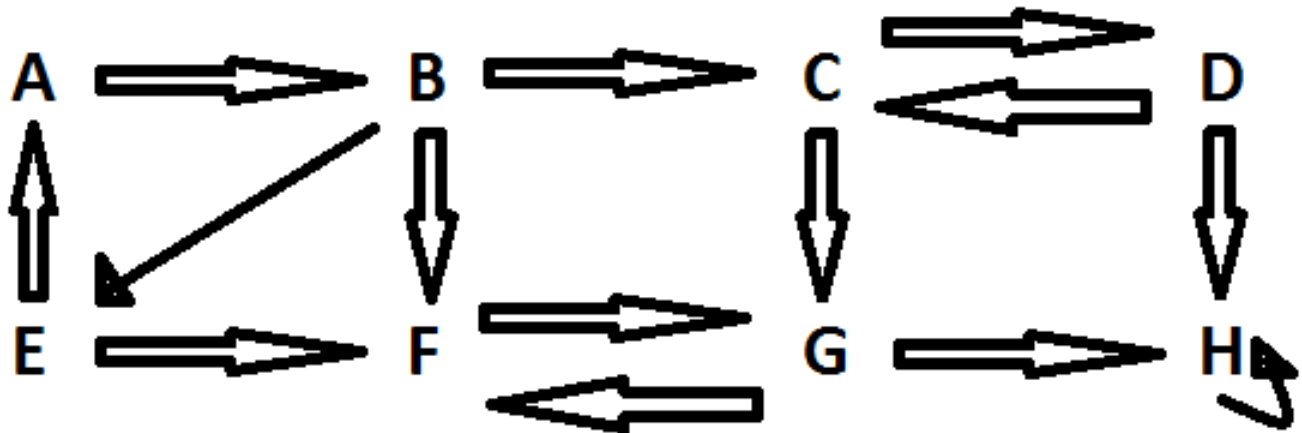
Algorithm 25.4 Strongly Connected Components

- ```

1: function STRONGLY-CONNECTED-COMPONENTS(G)
2: Call DFS(G) to compute finishing times $f[u]$ for each vertex u .
3: Compute G^T (the transpose of the graph).
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).
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

```
-

6. Find the strongly connected components.



7. Discuss:  $G$  and  $G^T$  will have the same strongly connected components.
8. Discuss: The component with the latest finish time vertex will have no edges in the transpose to any other component.