# Graphs 1: Strongly Connected Components

## Graph Algorithms

1. Outline
    - Review
    - Directed Graphs: SCC's
    - Application: 2-SAT
    - MST
    - PageRank

## Outline

1. Connected components via DFS-based algorithms
    - Depth-first search
    - Undirected graphs
    - Directed graphs
        - DAG: Directed acyclic graph
            * Topological sorting
        - SCC: Strongly connected components
            * Find with 2 DFS's

## Undirected Graphs

1. How do we get connected components in undirected G?
    - Run DFS and keep track of component number

```
DFS(G):
input: G(V,E) in adjacency list representation
output: Vertices labeled by connected components
cc = 0
for all v in V:
    visited(v) = false
for all v in V:
    if not visited(v):
        cc++
        Explore(v)
```

## Exploring Undirected Graphs

1. How to explore a graph:
2. Running time: O(n+m)
    - n = |V|, m = |E|

```
Explore(z):
ccnum(z) = cc
visited(z) = true
for all (z,w) in E:
    if not visited(w):
        Explore(e)
```

## DFS: Paths

1. DFS: connected components
    - How to find a path between connected vertices?

```
DFS(G):
input: G(V,E) in adjacency list representation
output: Vertices labeled by connected components
cc = 0
for all v in V:
    visited(v) = false
    prev(v) = null
for all v in V:
    if not visited(v):
        cc++
        Explore(v)

Explore(z):
ccnum(z) = cc
visited(z) = true
for all (z,w) in E:
    if not visited(w):
        Explore(e)
        prev(w) = z
```

## DFS on Directed Graphs

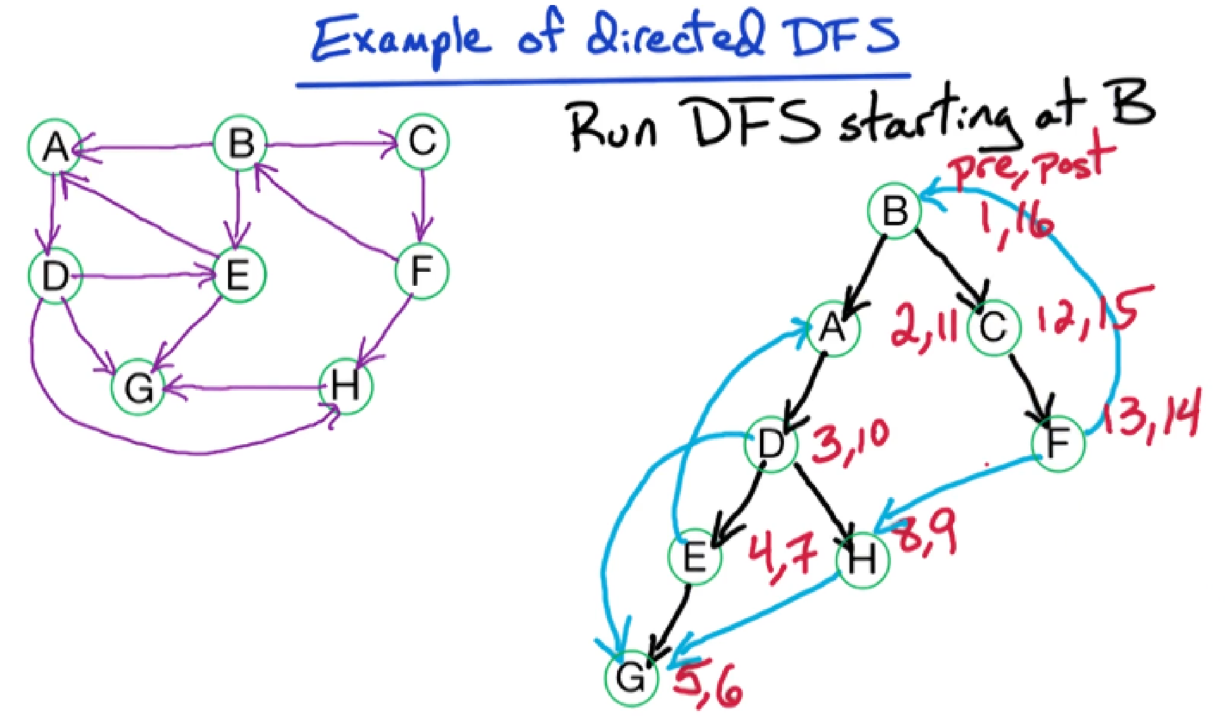
1. How do we get connectivity info for directed G?
    - Use DFS: add pre/postorder numbers

```
DFS(G):
input: G(V,E) in adjacency list representation
output: Vertices labeled by connected components
clock = 1
for all v in V:
    visited(v) = false
    prev(v) = null
for all v in V:
    if not visited(v):
        Explore(v)

Explore(z):
pre(z) = clock; clock++;
visited(z) = true
for all (z,w) in E:
    if not visited(w):
        Explore(e)
        prev(w) = z
post(z) = clock; clock++;
```

## Directed DFS: Example

1. Run DFS starting at B
    - Assume linked lists are stored in alphabetical order




DFS Example

## Types of Edges

1. Edge z -> w:
   - Tree edge:
     - B -> A, A -> D, . . .
     - post(z) > post(w)
   - Back:
     - E -> A, F -> B
     - post(z) < post(w)
   - Forward:
     - D -> G, B -> E
     - post(z) > post(w)
   - Cross:
     - F -> H, H -> G
     - post(z) > post(w)

## Cycles

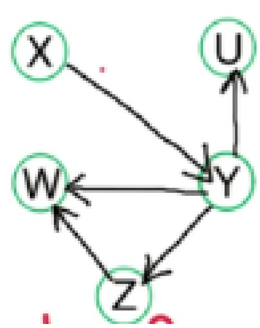
1. G has a cycle iff its DFS tree has a back edge
2. Proof:
   - cycle a -> b -> c -> . . . -> j -> a
   - Suppose our graph has a cycle, prove that the DFS tree has a back edge
     - One of these is first explored, say i
       * Back edge: i-1 -> i (from descendant to ancestor)
   - Suppose our DFS tree has a back edge, prove the tree contains a cycle
     - Back edge a -> b

## Topological Sorting

1. DAG: Directed acyclic graph
   - No cycles = No back edges
2. Topologically sorting a DAG
   - Order vertices so that all edges go lower -> higher
3. Run DFS on DAG G
   - for all z -> w, we know post(z) > post(w)
   - Order vertices by decreasing post-order number
     - Post-order numbers range from 1 to 2n
     - Linear time sorting because we can just place them at their index in an array
   - Overall runtime: O(n+m)

## Topological Ordering Quiz

1. Give all topological orderings:
   - XYUZW
   - XYZWU
   - XYZUW
   - Z must come before W
2. How many topological orderings are there?




Topological Ordering Quiz

## DAG Structure

1. Source vertex = no incoming edges = highest post-order number
2. Sink vertex = no outgoing edges = lowest post-order number
3. DAG guarantees at least one source and one sink
4. Alternative topological sorting algorithm:
   - Use for general directed graphs
   - 1) Find a sink, output it and delete it
   - 2) Repeat (1) until the graph is empty

## Outline Review

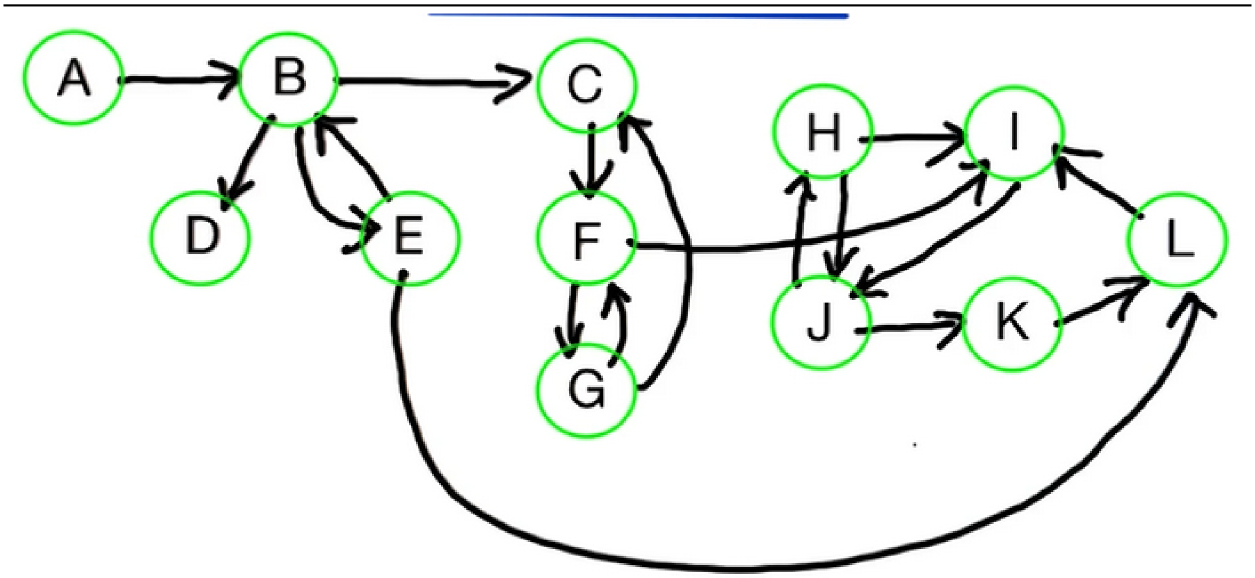
1. Connected components via DFS-based algorithms
   - Undirected graphs
   - Directed graphs
     - DAG = directed acyclic graph
       - Topological sorting
     - SCC = strongly connected components
       - Find with 2 DFS's

## Connectivity in Directed Graphs

1. Vertices v and w are strongly connected if:
   - There is a path v -> w and w -> v
2. SCC = Strongly connected component
   - Maximal set of strongly connected vertices

## SCC Quiz

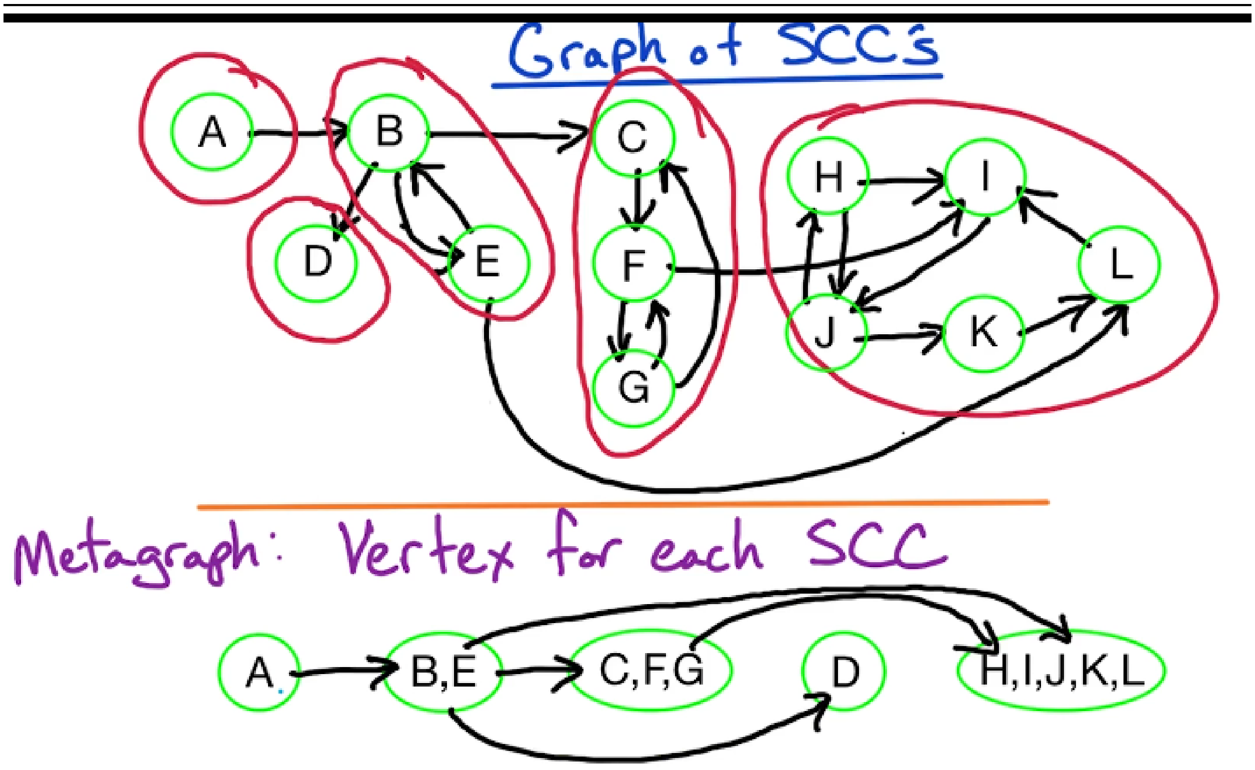
1. How many SCCs in this graph?
   - 5
2. What are they?
   - {A}
   - {B,E}
   - {C,F,G}
   - {D}
   - {H,I,J,K,L}




SCC Quiz

## Graph of SCC

1. Metagraph: Vertex for each SCC
   - Metagraph of SCC is always a DAG
     - Two SCCs that have a cycle are, by definition, a single SCC
2. Every directed graph is a DAG of its SCC's
   - Find SCCs and topological ordering with two runs of DFS




SCC Metagraph

## SCC Algorithm Idea

1. Steps to the algorithm
   - Find sink SCC S
   - Output it
   - Remove it
   - Repeat
2. Why sink SCC?
   - Take any v in S where S is sink SCC
   - Run Explore(v): visit all of S and nothing else
   - If we start with a source, like we could for a topological sort, we'll find that we can reach many vertices
3. How do we find a sink SCC?

## Vertex in sink SCC

1. In a DAG:
   - Vertex with lowest postorder number is a sink
2. In a general directed G:
   - Does v with lowest postorder number always lie in a sink SCC?
     - No
     - B <-> A -> C
       * Start at A
       * A = (1,6)
       * B = (2,3)
       * C = (4,5)

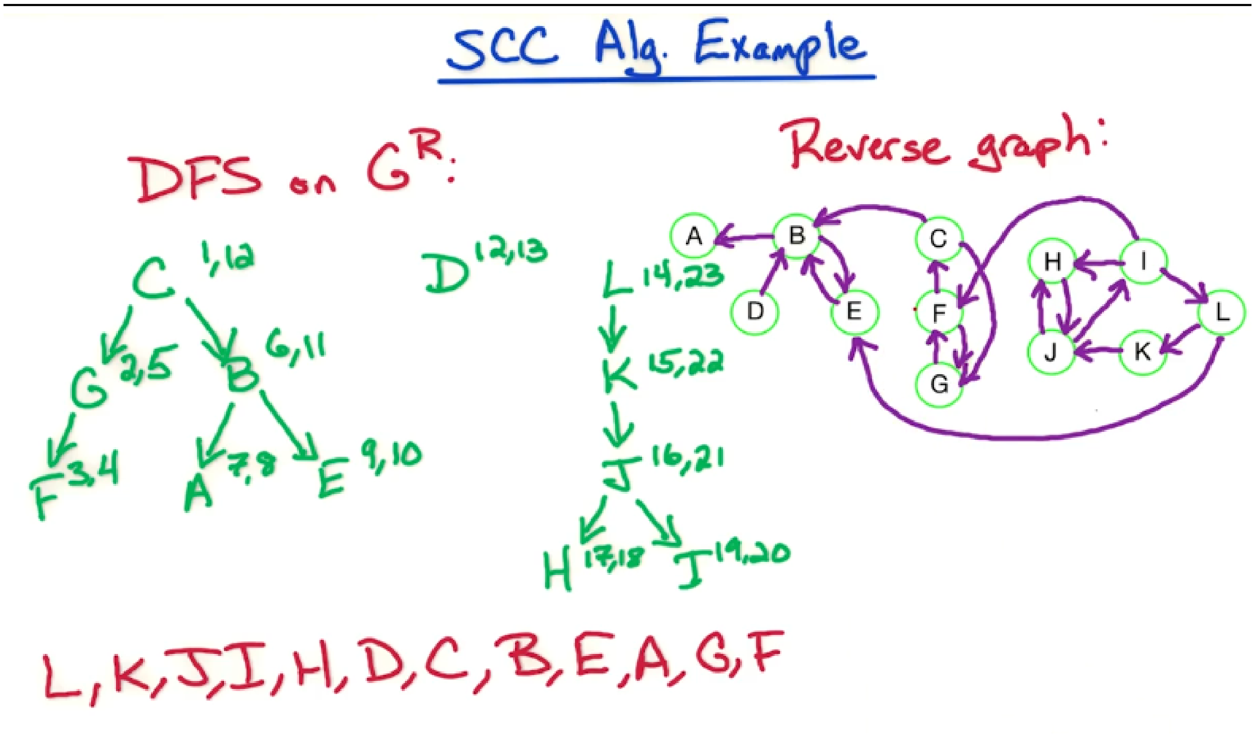
    * B has the lowest postorder number, but is part of a source SCC
- Does v with highest postorder number always lie in a source SCC?
  - Yes

## Finding sink SCC

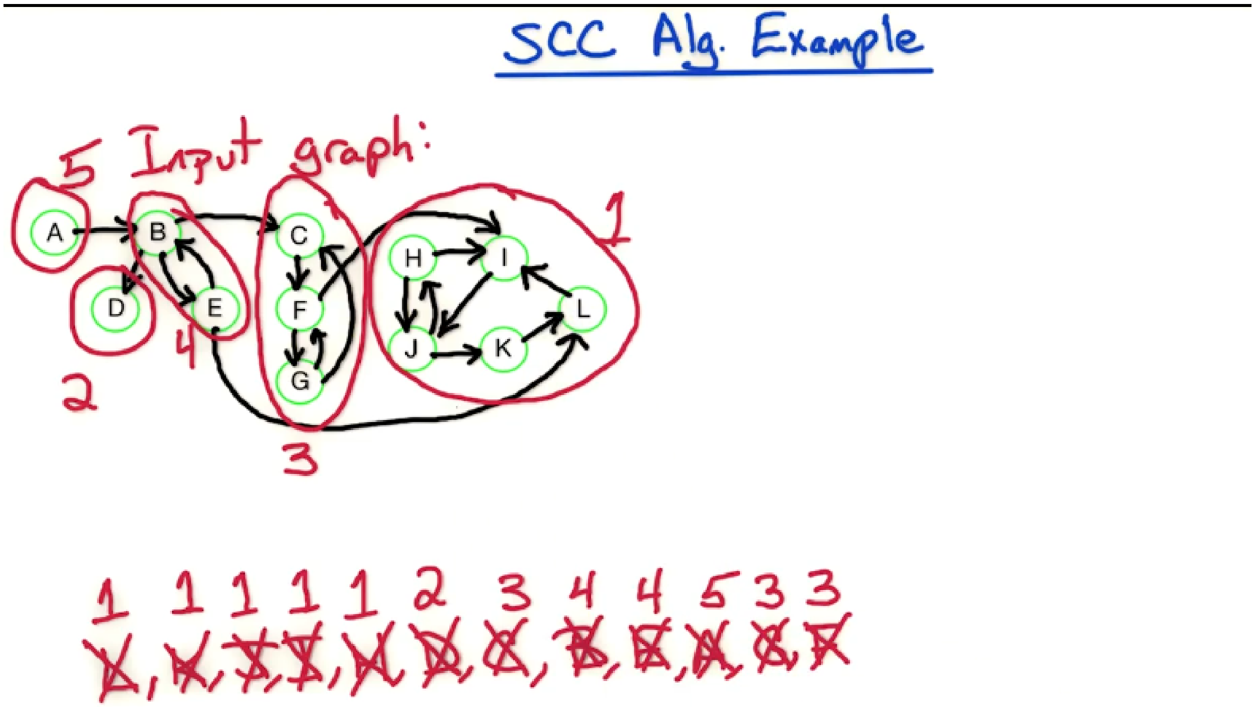
1. Vertex v with highest postorder number lies in a source SCC
   - How to get w in sink SCC?
     - Reverse the graph and find a source SCC
       * Sources become sinks, sinks become sources
   - For directed G = (V,E), look at $G^R = (V,E^R)$ = reverse of G
     - Er = {wv : vw in E} = reverse of every edge in E
     - Source SCC in G = sink SCC in $G^R$
     - Sink SCC in G = source SCC in $G^R$

## SCC Example

1. Sort the vertices by postorder number on the reverse graph in decreasing order
   - Two runs of DFS provide the strongly connected components and structure them in topological order




SCC Algorithm




SCC Algorithm

## SCC Algorithm

1. Runtime: O(n+m)

```
SCC(G):
    input: directed G = (V,E) in adjacency list
    Construct Gr
    Run DFS on Gr
    Order V by decreasing postorder number
    Run undirected connected components algorithm on G
```

## Proof of Key SCC Fact

1. Vertex with highest postorder number lies in a source SCC
   - Simpler claim:
     - For SCC's S and S'
       * if v in S -> w in S'
       * then max postorder number in S > max postorder number in S'
   - Topologically sort by max postorder number

## Simpler Claim

1. For SCC's S and S'
   - if v in S -> w in S'
   - then max postorder number in S > max postorder number in S'
   - No path from S' -> S because S and S' are both SCCs
2. Run DFS on G
   - Some vertex z in S U S' visited first

    - if z in S':
        * when we Explore(z) we see all of S' and none of S
        * All postorder numbers in S' < all min(postorder numbers is S)
    - if z in S:
        * when we Explore(z), z has max postorder number in S U S'
        * We will traverse the whole tree and finish at z

## BFS/Dijkstra's

1. DFS: Connectivity
2. BFS:
    - Input: G(V,E) and start vertex s in V
    - Output: for all v in V, dist(v) = min number of edges from s to v
        - Also, prev(v)
    - Runtime: O(n+m)
3. Dijkstra's:
    - Input: G(V,E) and start vertex s in V, l(e) > 0 for every e in E
    - Output: for all v in V, dist(v) = length of shortest s -> v path
        - Also, prev(v)
    - Runtime: O((n+m)logn)
    - Like BFS, but considers weights for each edge
    - Uses a min-heap