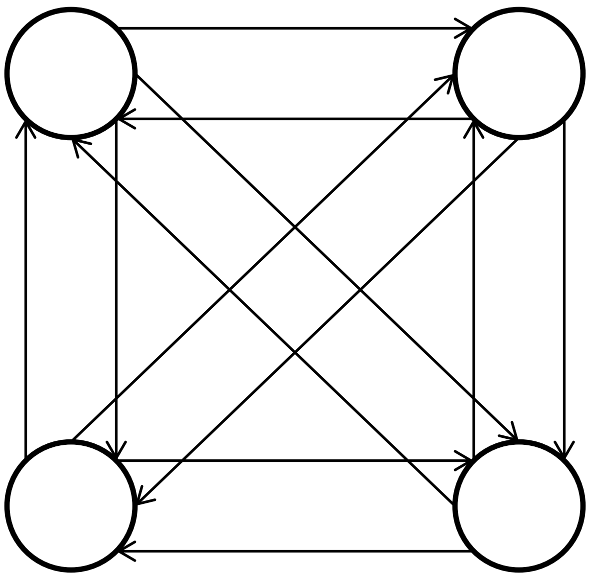
CS 161: Design and Analysis of Algorithms

Graphs 2:
Directed Connectivity

- Types of directed graphs
- Directed acyclic graphs
- Strongly connected components

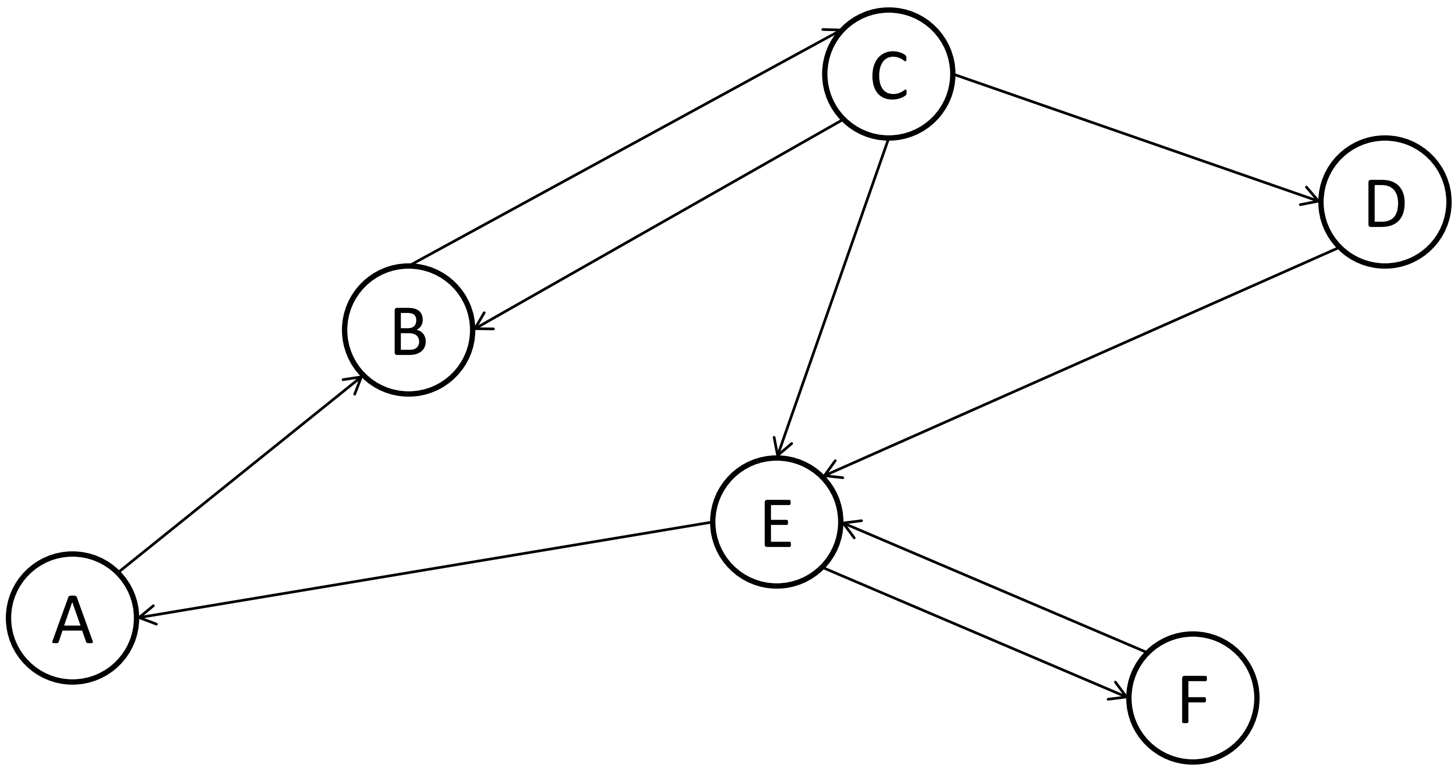
Types of Directed Graphs

- Complete graph



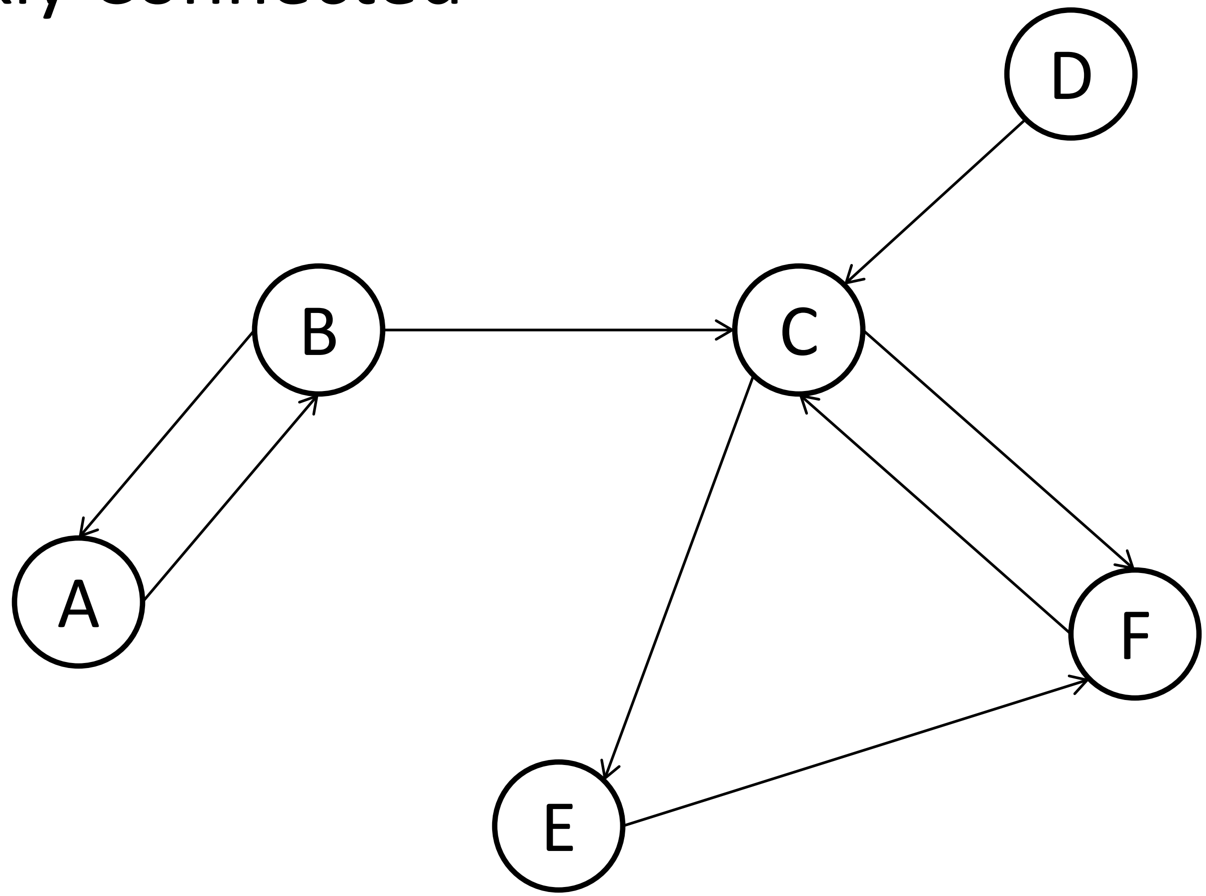
Type of Directed Graphs

- Strongly Connected



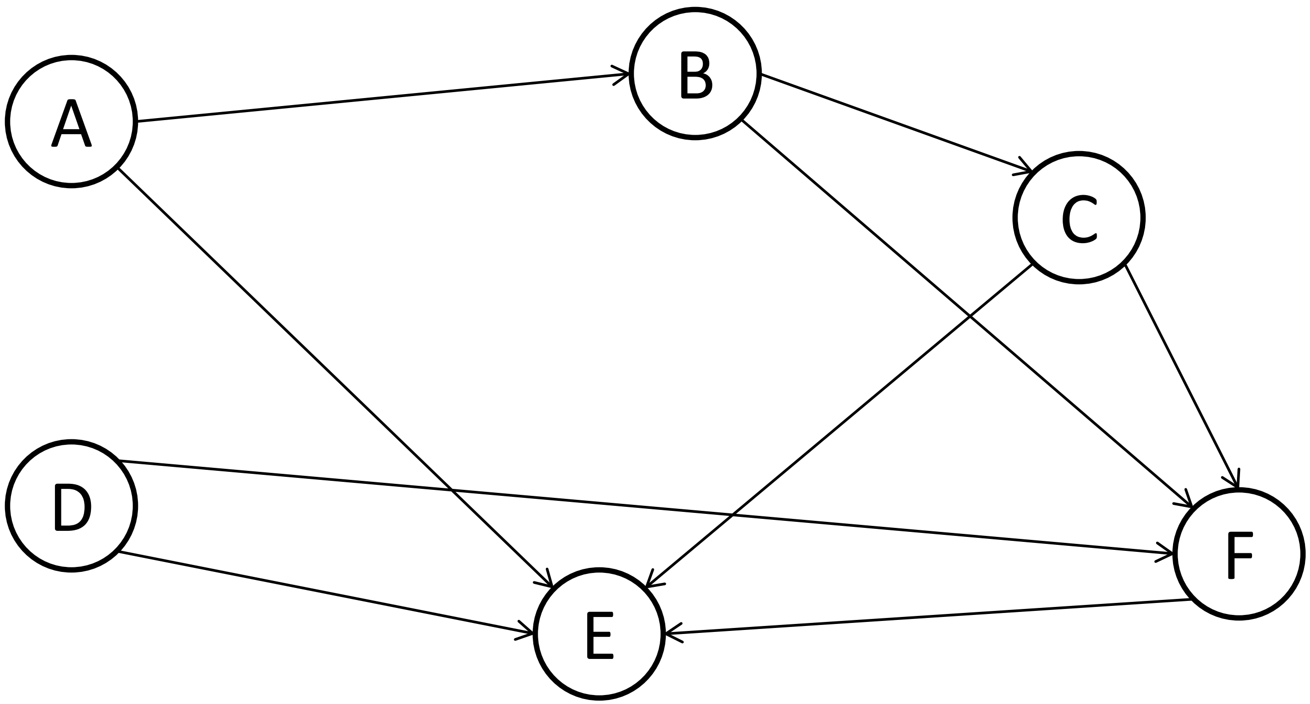
Types of Directed Graphs

- Weakly Connected



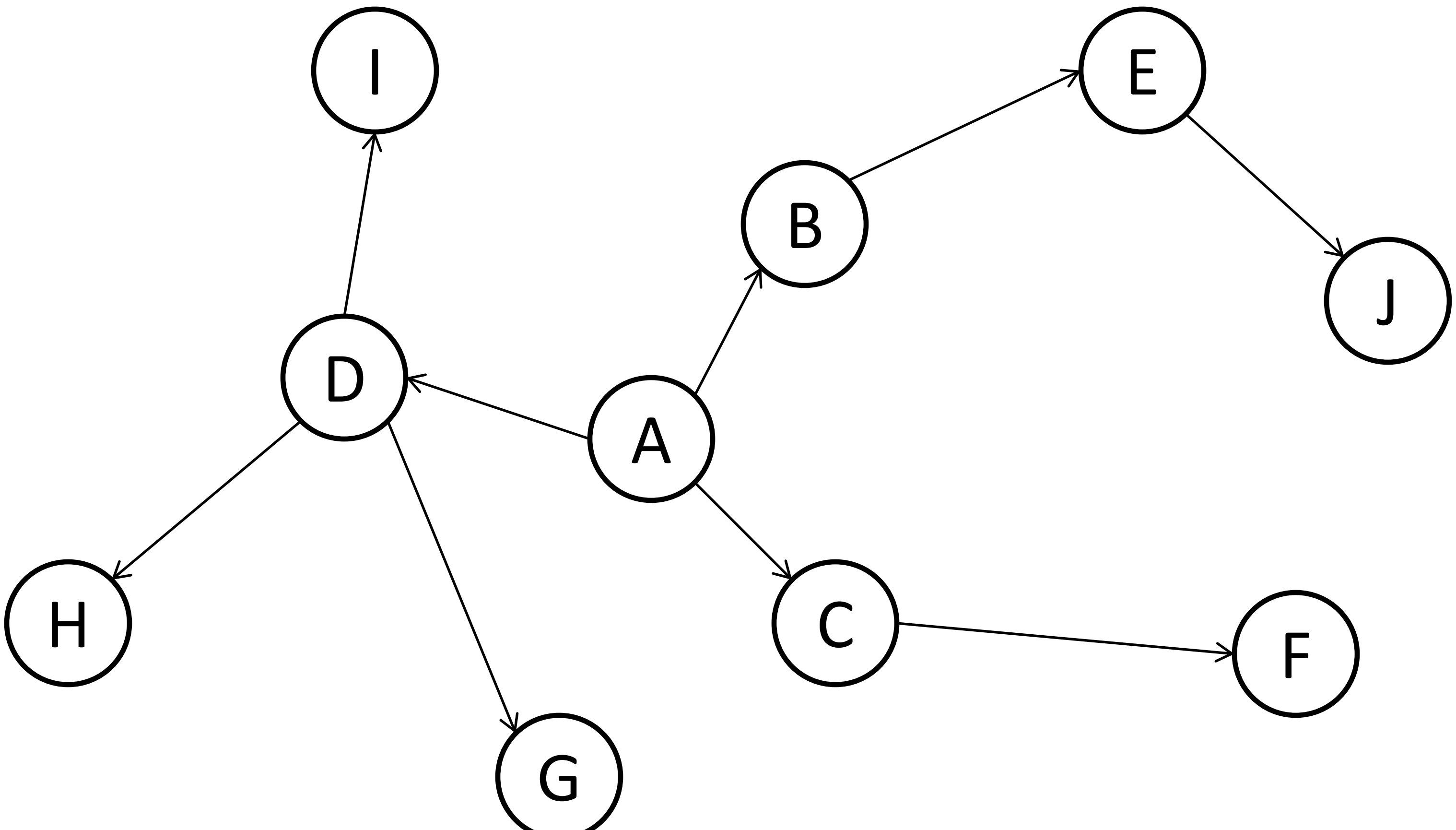
Types of Directed Graphs

- Directed Acyclic Graph (DAG)



Types of Graphs

- Rooted Tree



Directed Acyclic Graphs

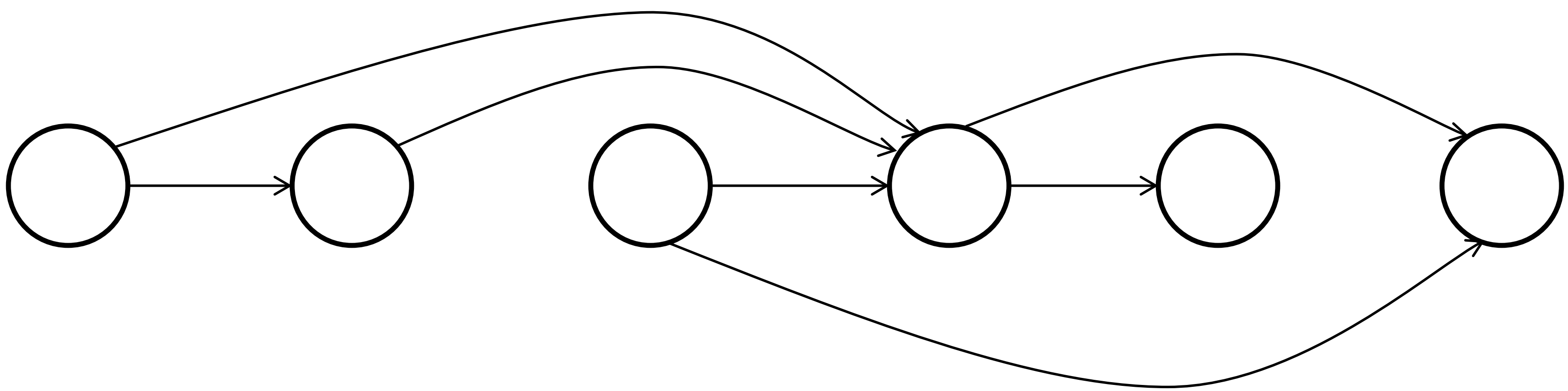
- Examples:
  - Computer science curriculum
  - Family “trees”
  - Any cause and effect relationships

Directed Acyclic Graphs

- Source: node with no incoming edges
- Sink: node with no outgoing edges
- Property: In every dag, there is at least one source and one sink
  - Proof: Pick any node. Repeatedly follow out edges.
  - If process never stops, must have cycle
  - Therefore, eventually find sink

Topological Ordering

- Assign labels 1 ,2 , … to nodes such that all edges (i,j) have i < j



- Any graph with a topological ordering is a dag
- What dags admit topological orderings?

Ordering a Dag

- Algorithm to compute topological ordering:
  - Find a source node v, give it label 1
  - Remove v from graph (along with its outgoing edges)
  - Repeat with label 2, 3, … until no nodes left.

Ordering a Dag

- Algorithm to compute topological ordering:
  - Find a source node v, give it label 1
  - Remove v from graph (along with its outgoing edges)
  - Repeat with label 2, 3, … until no nodes left.
- In a dag, there is always a v, so this algorithm always runs successfully
- Whenever a node is removed, the only edges removed go to unlabeled nodes
  - Those nodes will end up having a higher label
- All edges (i,j) have i < j

Ordering a Dag

- How to find source node?
  - Pick any node, repeatedly follow incoming edges until we hit a sink

- Can easily modify algorithm to check if graph is a dag:
  - Pick any node, mark it as visited, and repeatedly follow incoming edges until we hit a sink or a visited node

Running Time?

- Finding each source could take $O(|V|)$ time.
- $|V|$ nodes removed, so total time $O(|V|^2)$
- Can we improve to $O(|V|+|E|)$?
  - Depth first search!

DFS Revisited

- DFS(G) =

  initialize()

  visited(v) = false for all v

  For all v,

    If not visited(v):

      update()

      explore(G,v)

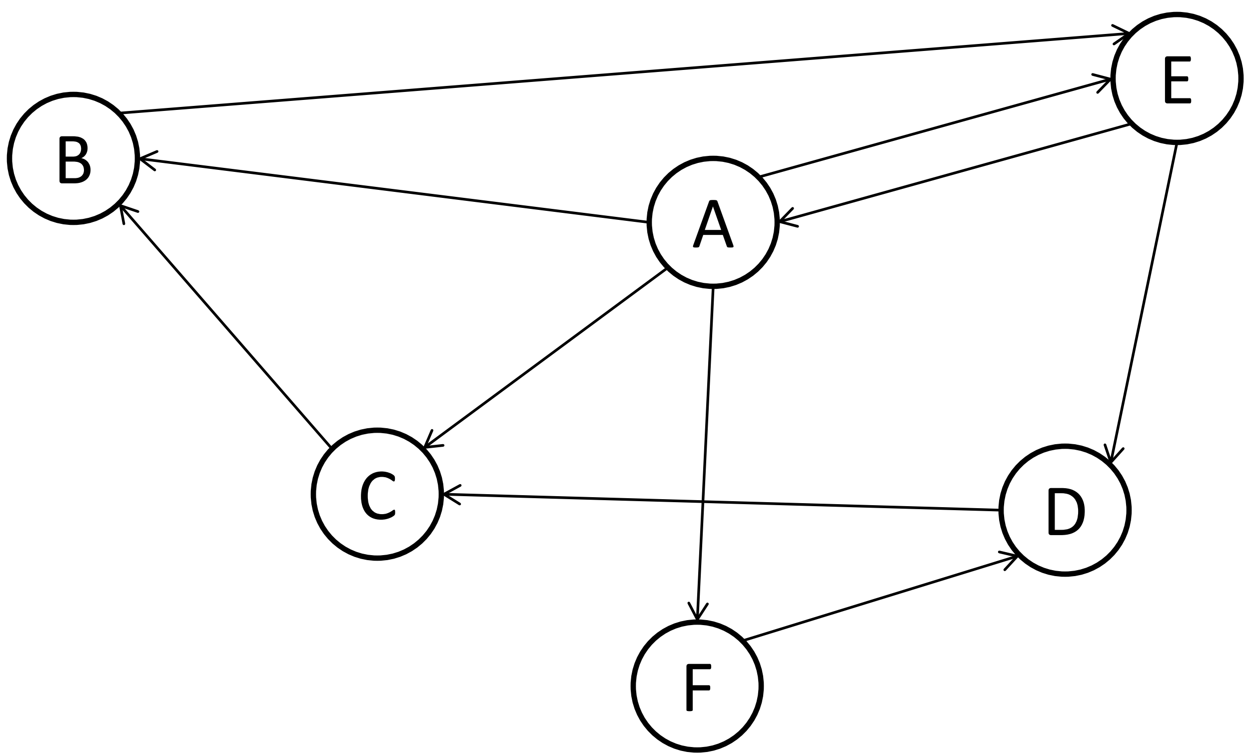
DFS Revisited

- Explore(G,u) =

```
visited(u) = true
previsit(u)
For each edge (u,v) where not visited(v):
   explore(v)
postvisit(u)
```

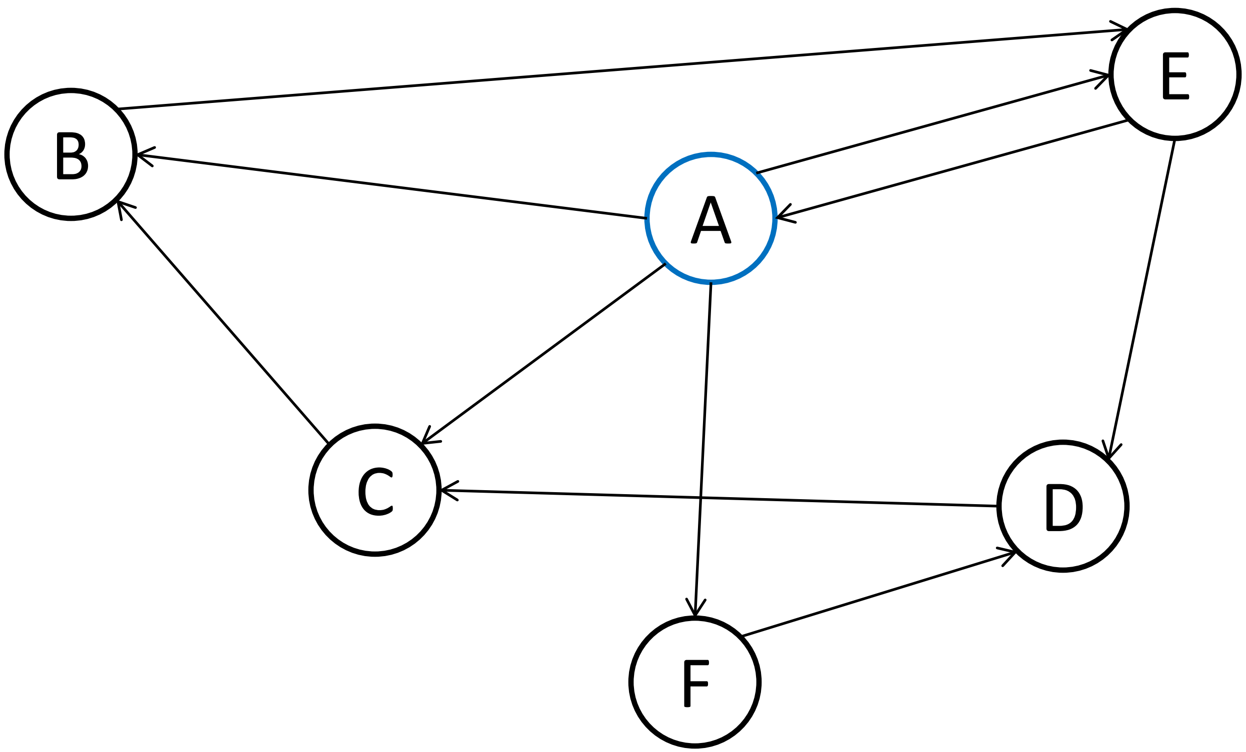
DFS Tree

- Rooted tree formed by only looking at the edges followed



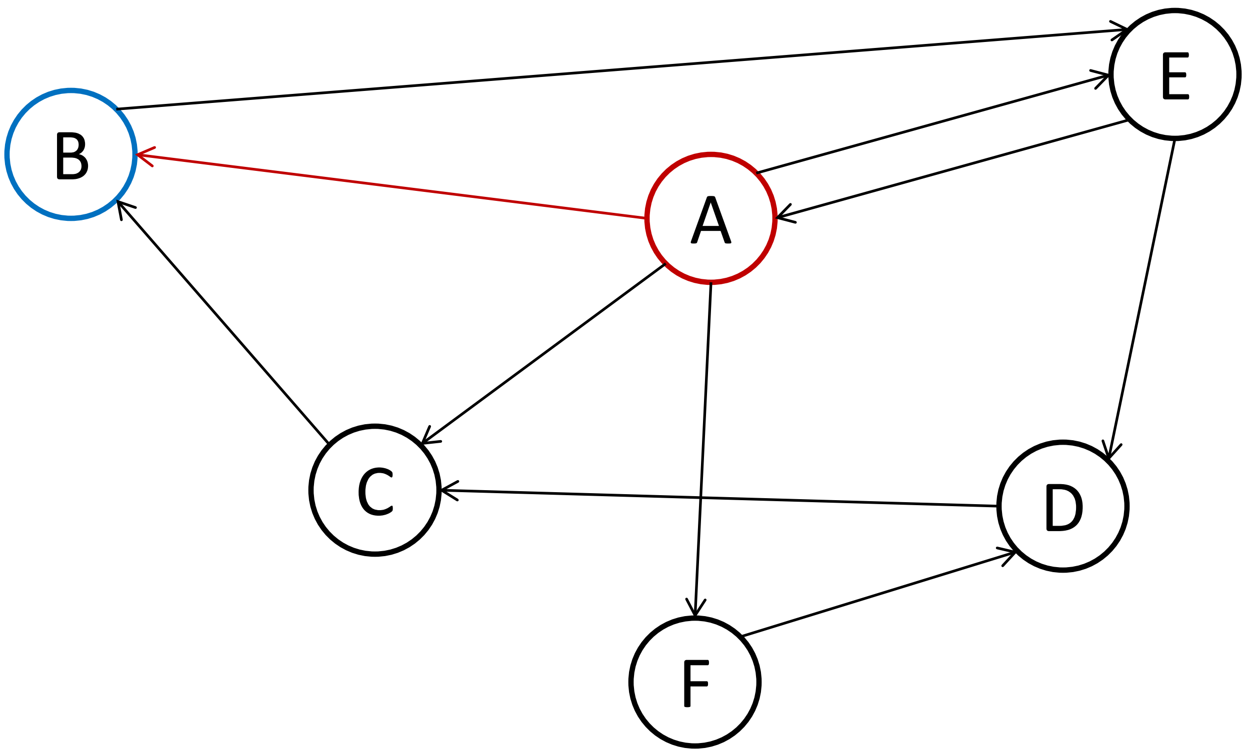
DFS Tree

- Rooted tree formed by only looking at the edges followed



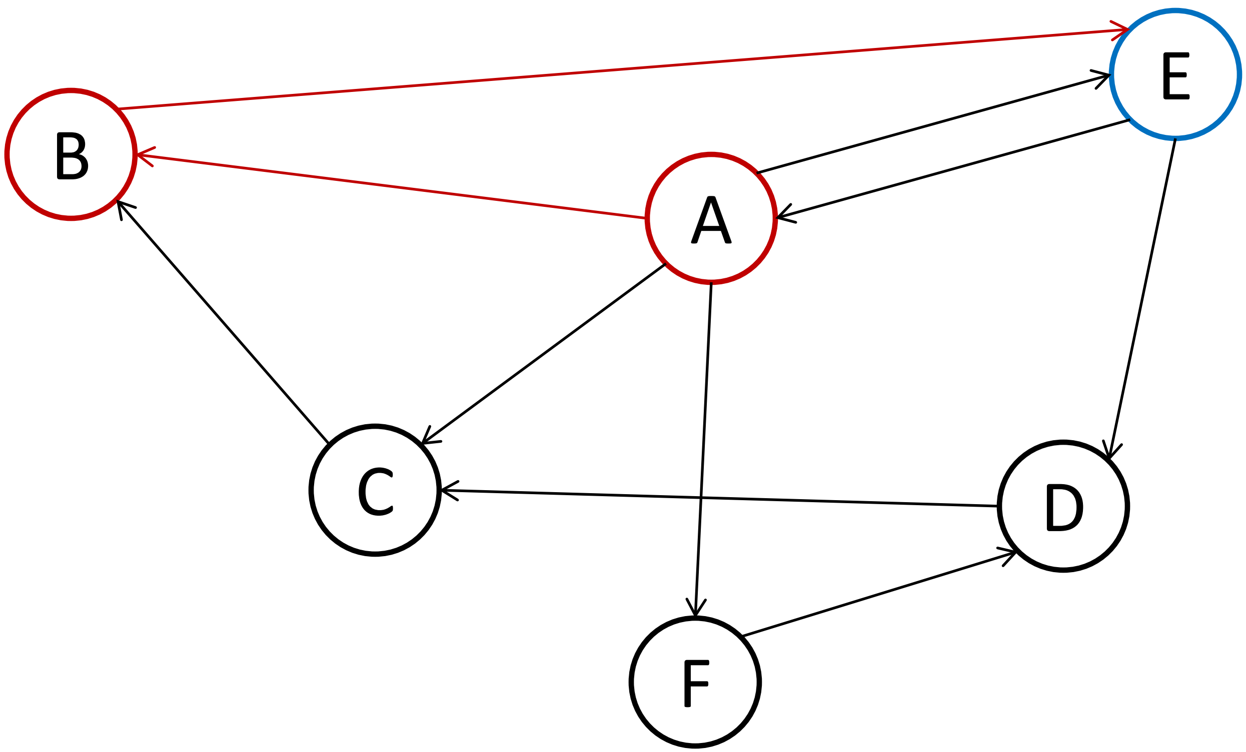
DFS Tree

- Rooted tree formed by only looking at the edges followed



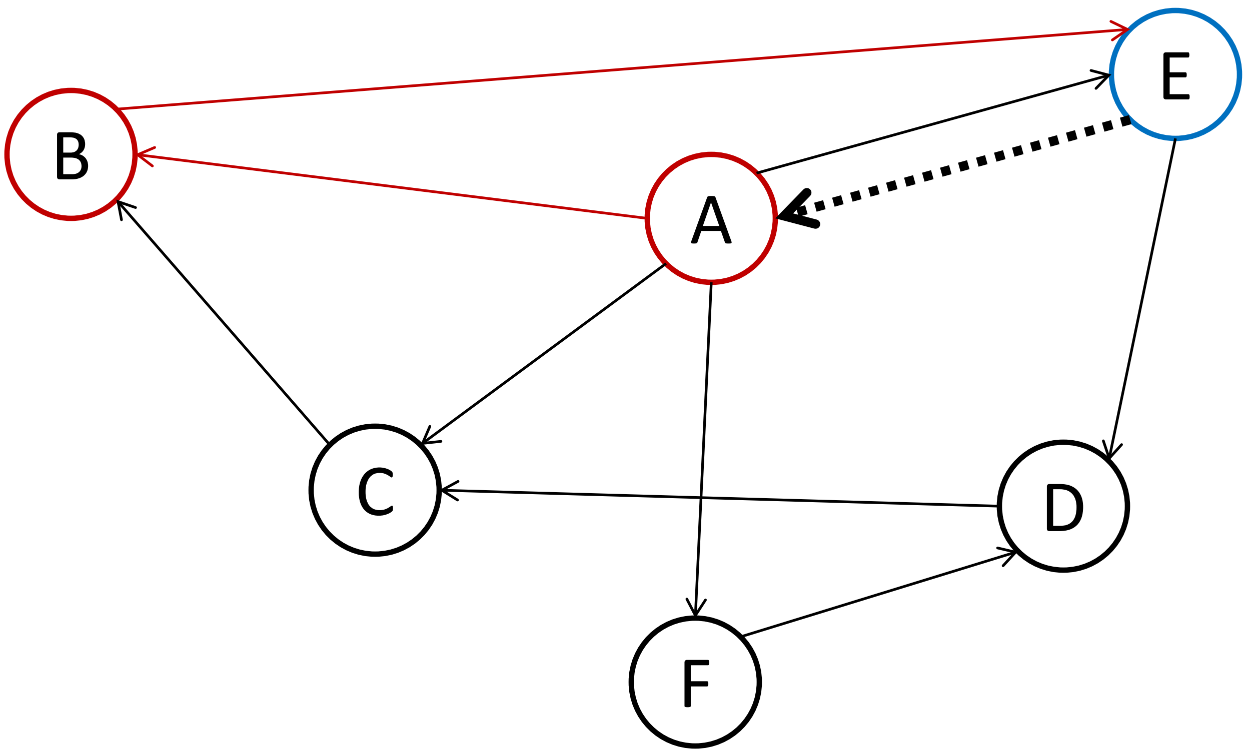
DFS Tree

- Rooted tree formed by only looking at the edges followed



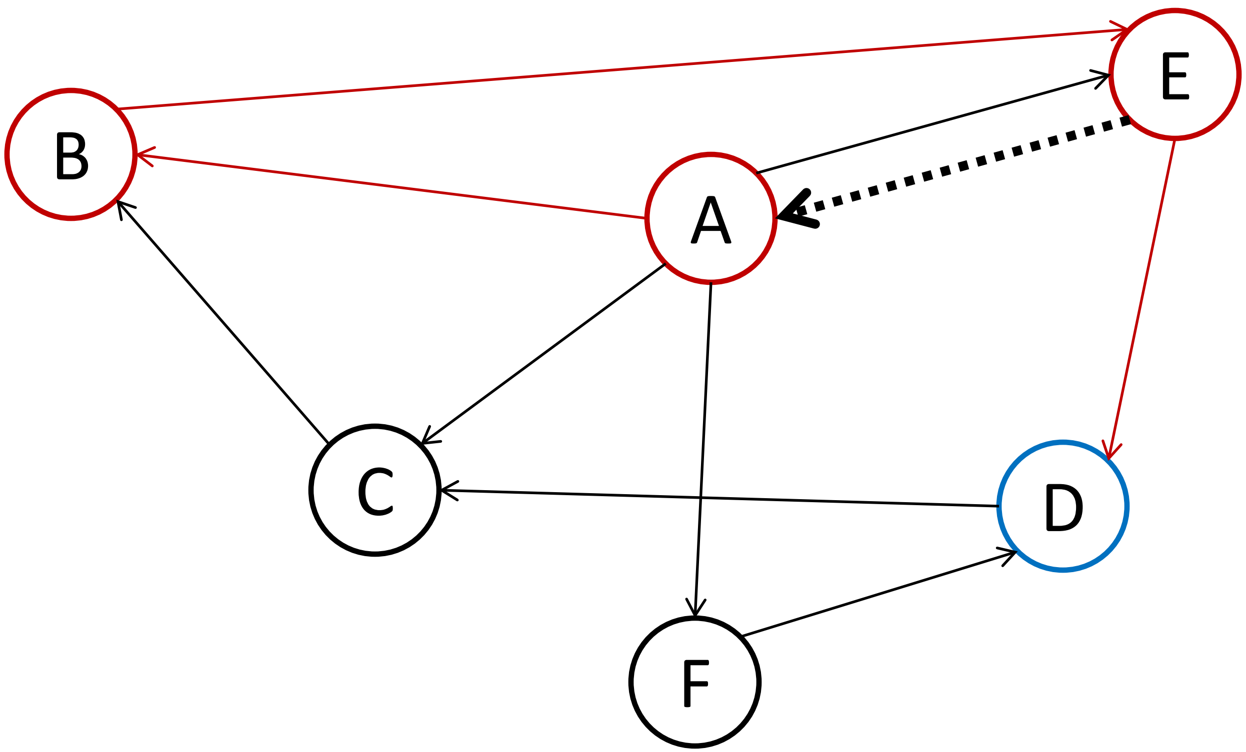
DFS Tree

- Rooted tree formed by only looking at the edges followed



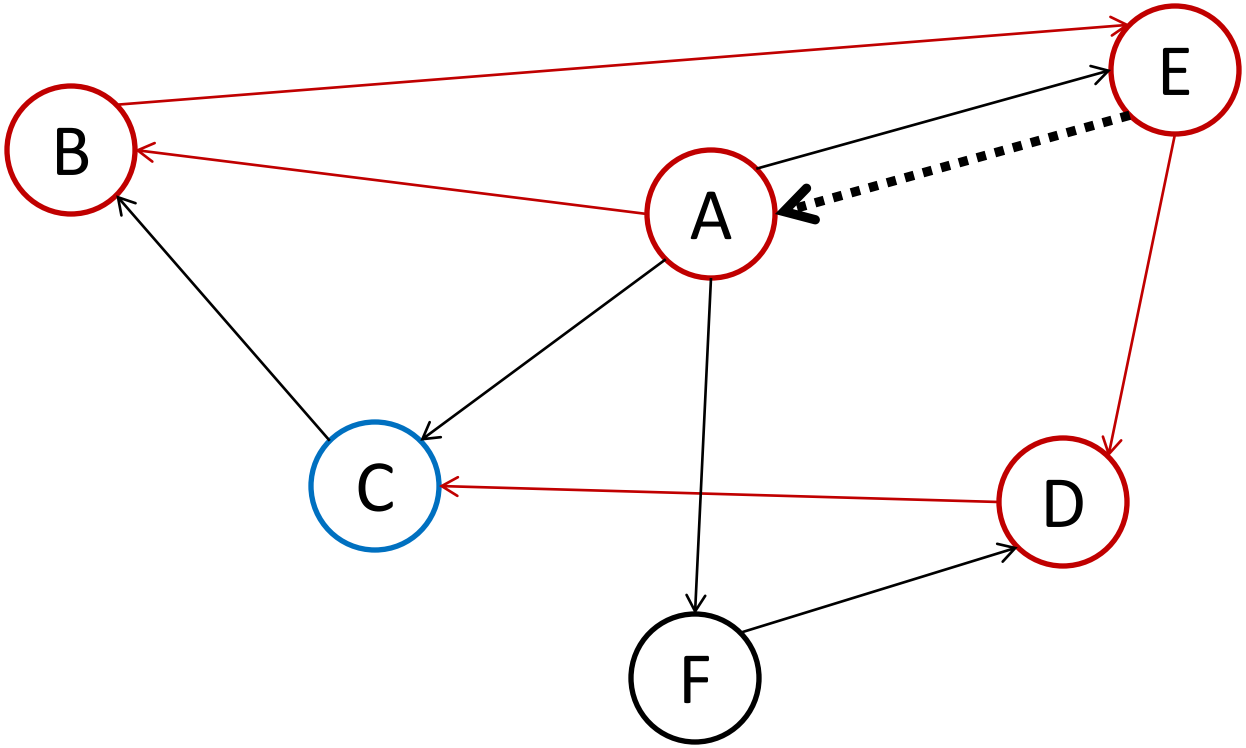
DFS Tree

- Rooted tree formed by only looking at the edges followed



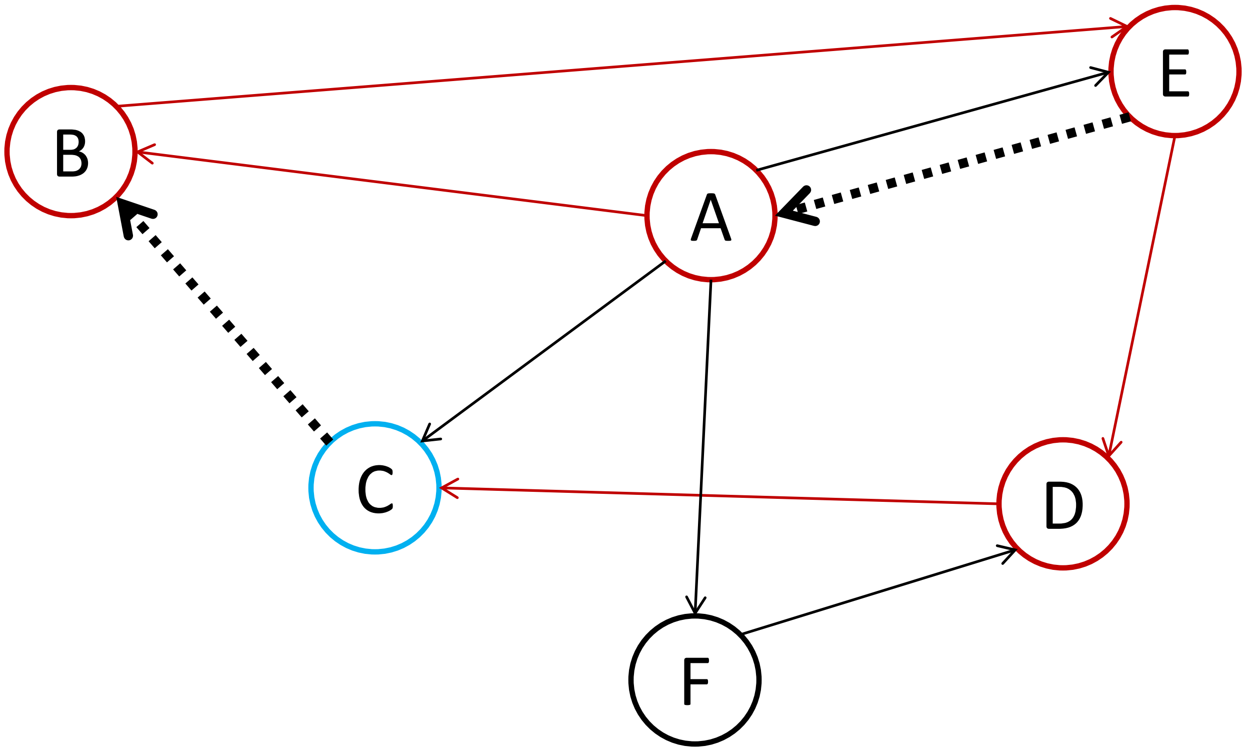
DFS Tree

- Rooted tree formed by only looking at the edges followed



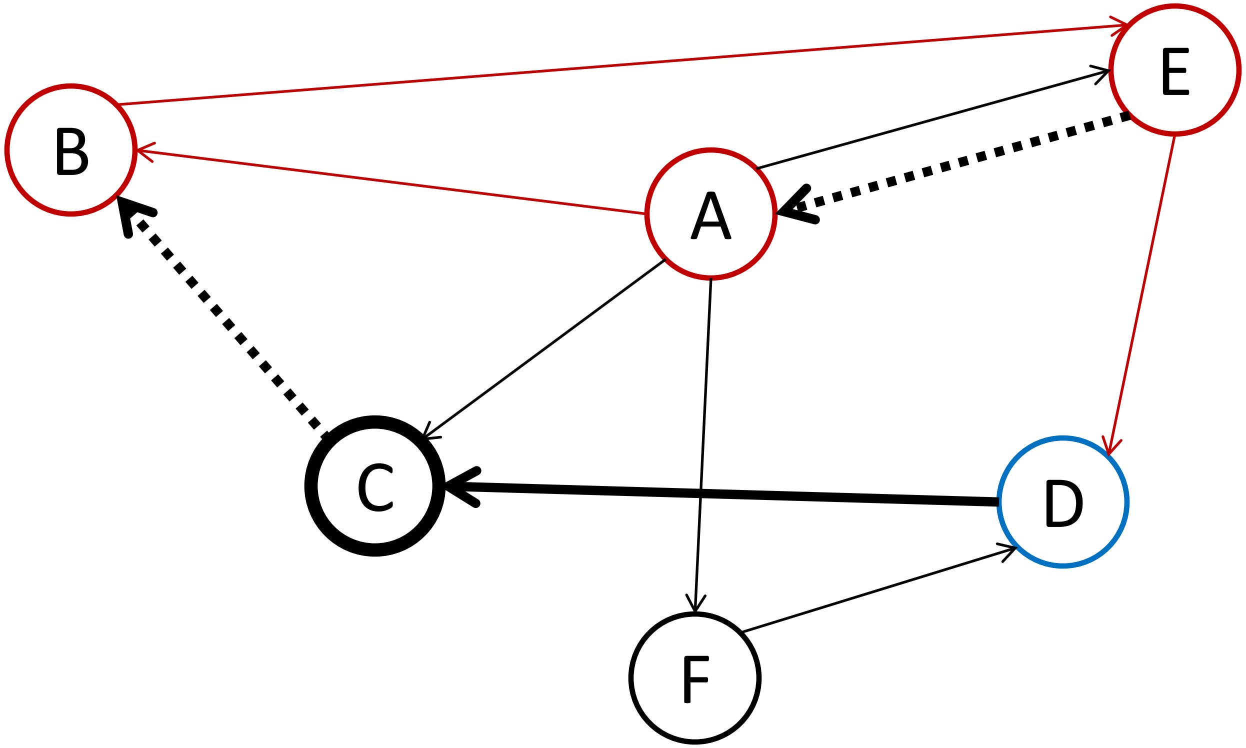
DFS Tree

- Rooted tree formed by only looking at the edges followed



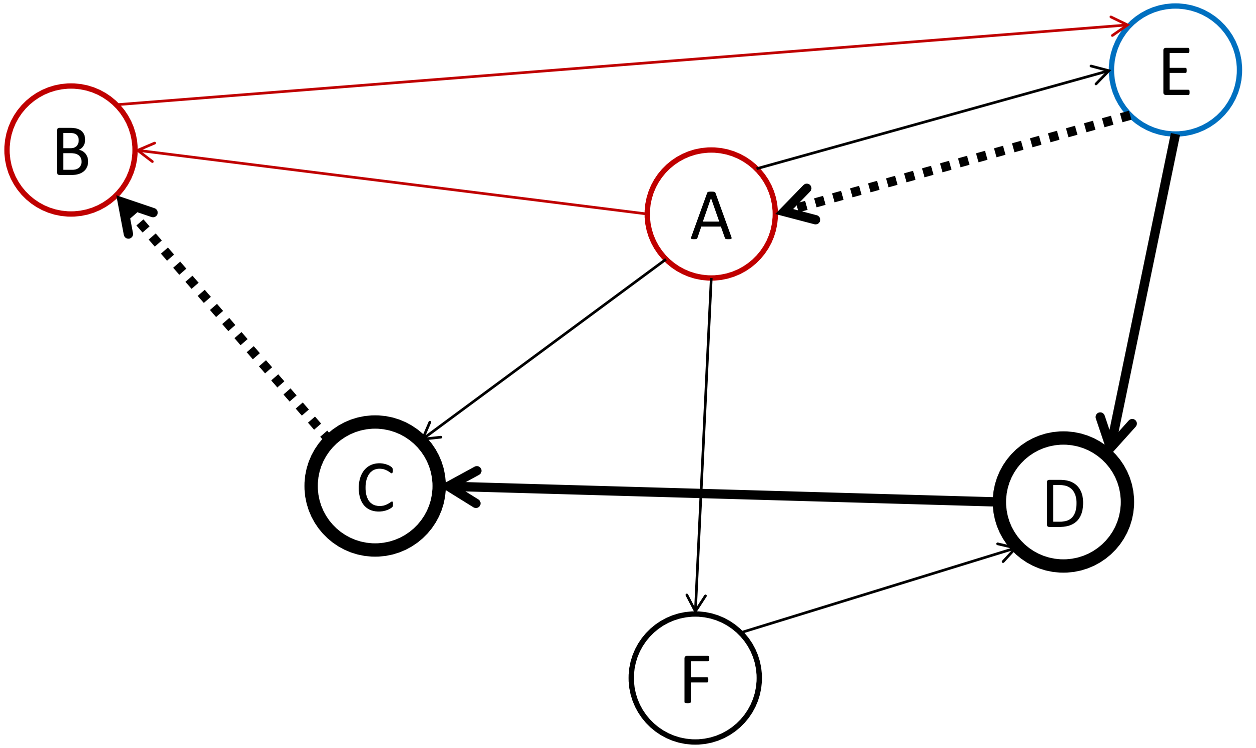
DFS Tree

- Rooted tree formed by only looking at the edges followed



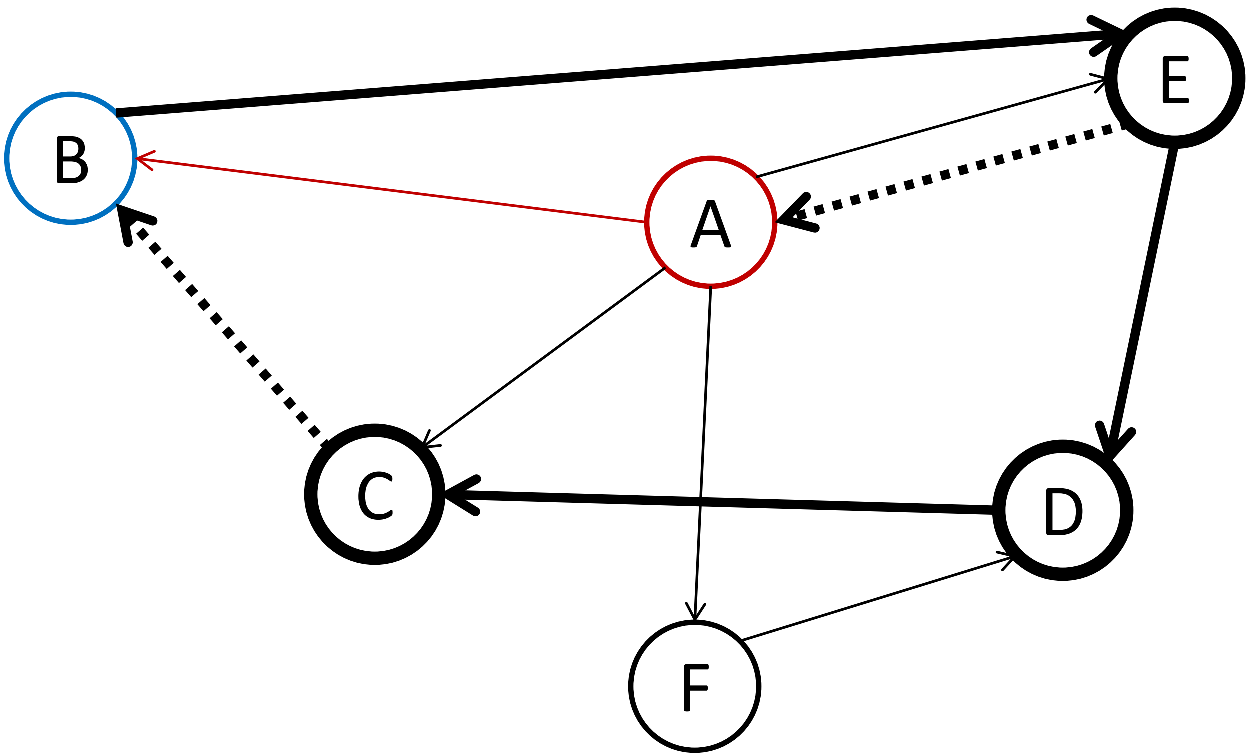
DFS Tree

- Rooted tree formed by only looking at the edges followed



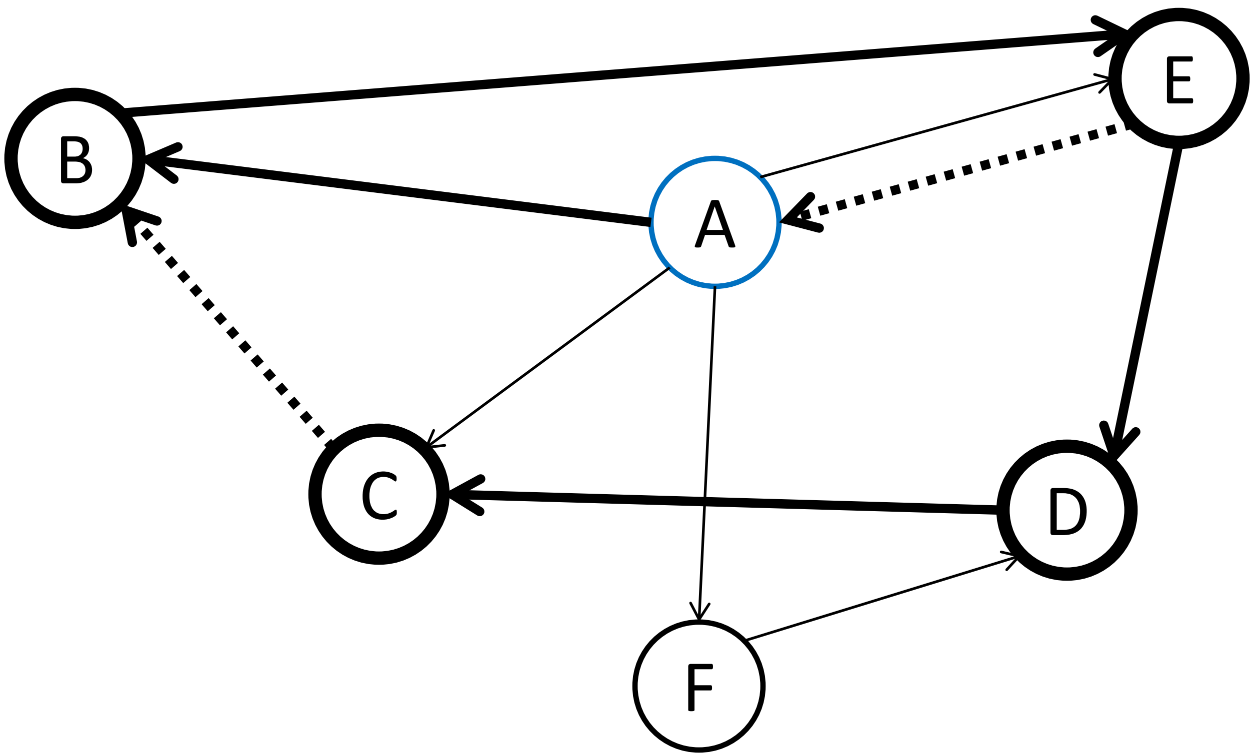
DFS Tree

- Rooted tree formed by only looking at the edges followed



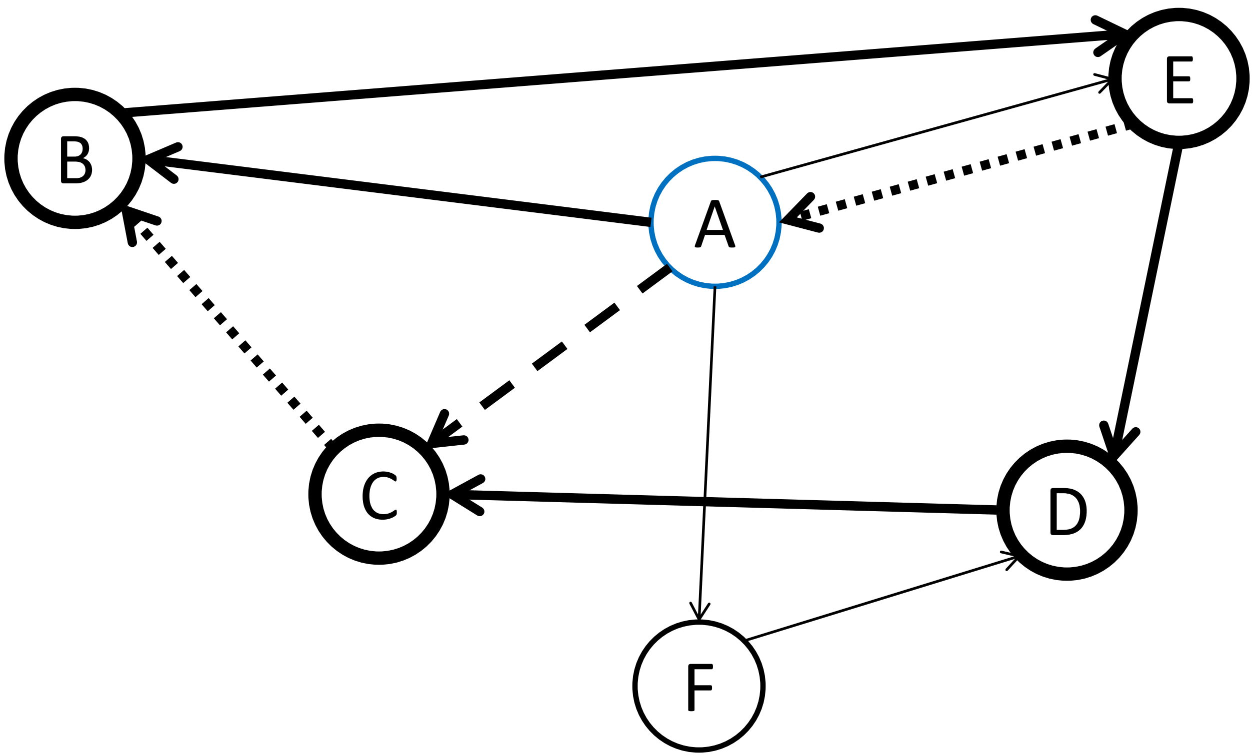
DFS Tree

- Rooted tree formed by only looking at the edges followed



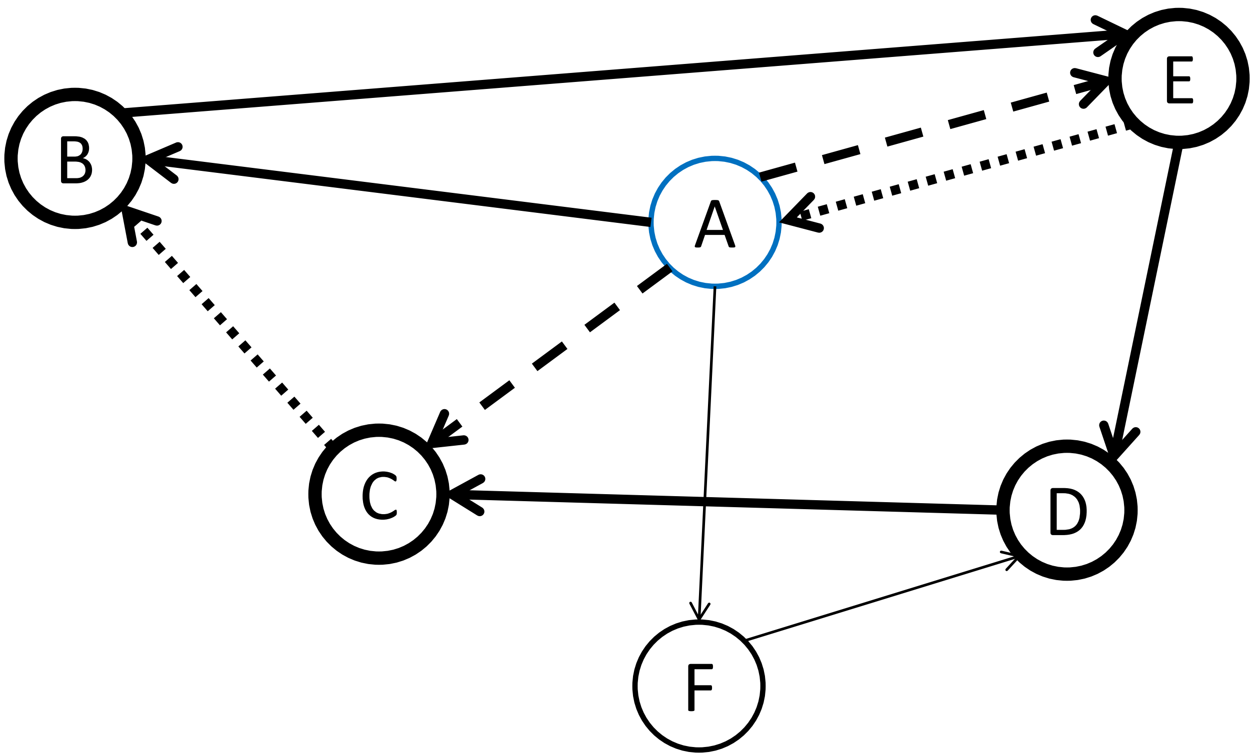
DFS Tree

- Rooted tree formed by only looking at the edges followed



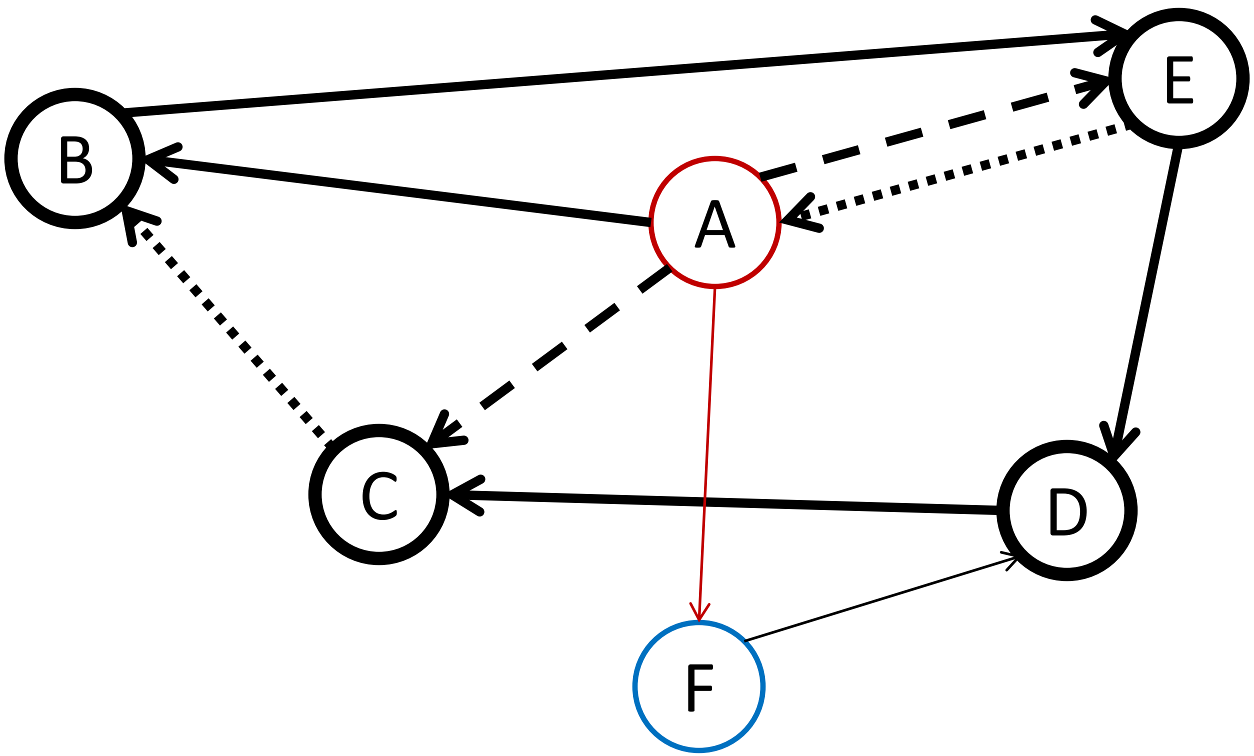
DFS Tree

- Rooted tree formed by only looking at the edges followed



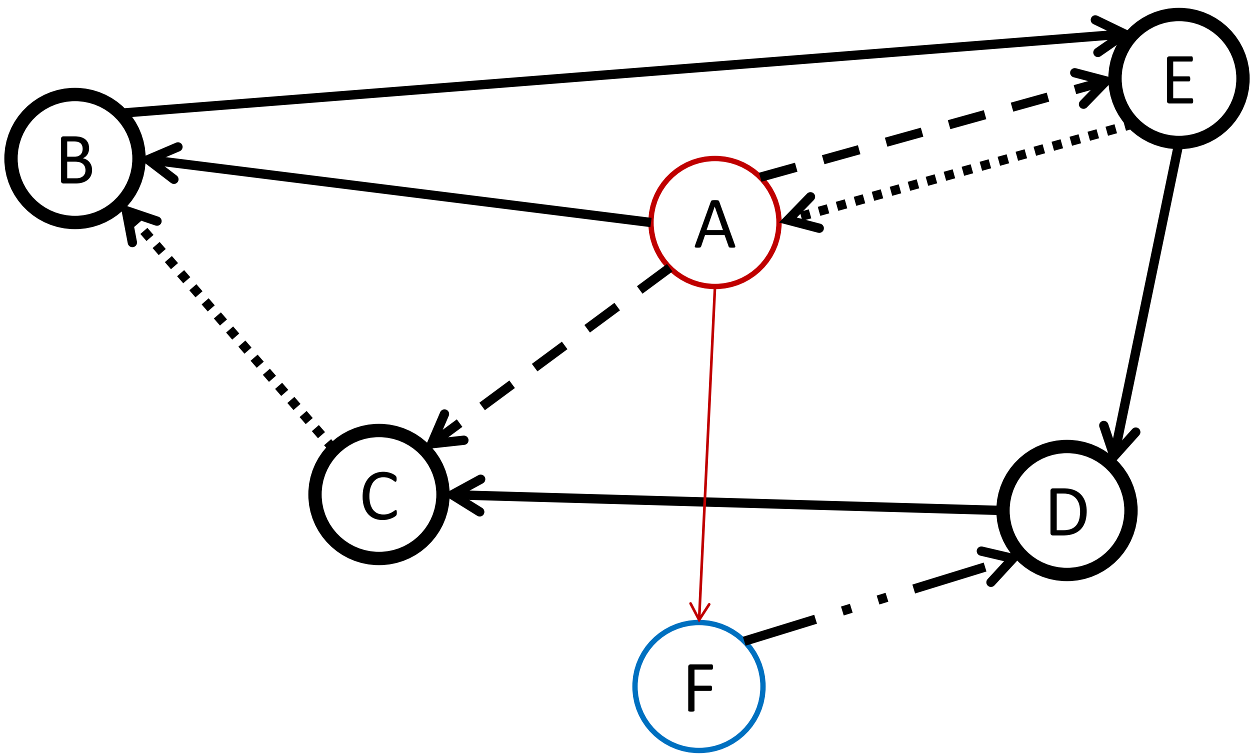
DFS Tree

- Rooted tree formed by only looking at the edges followed



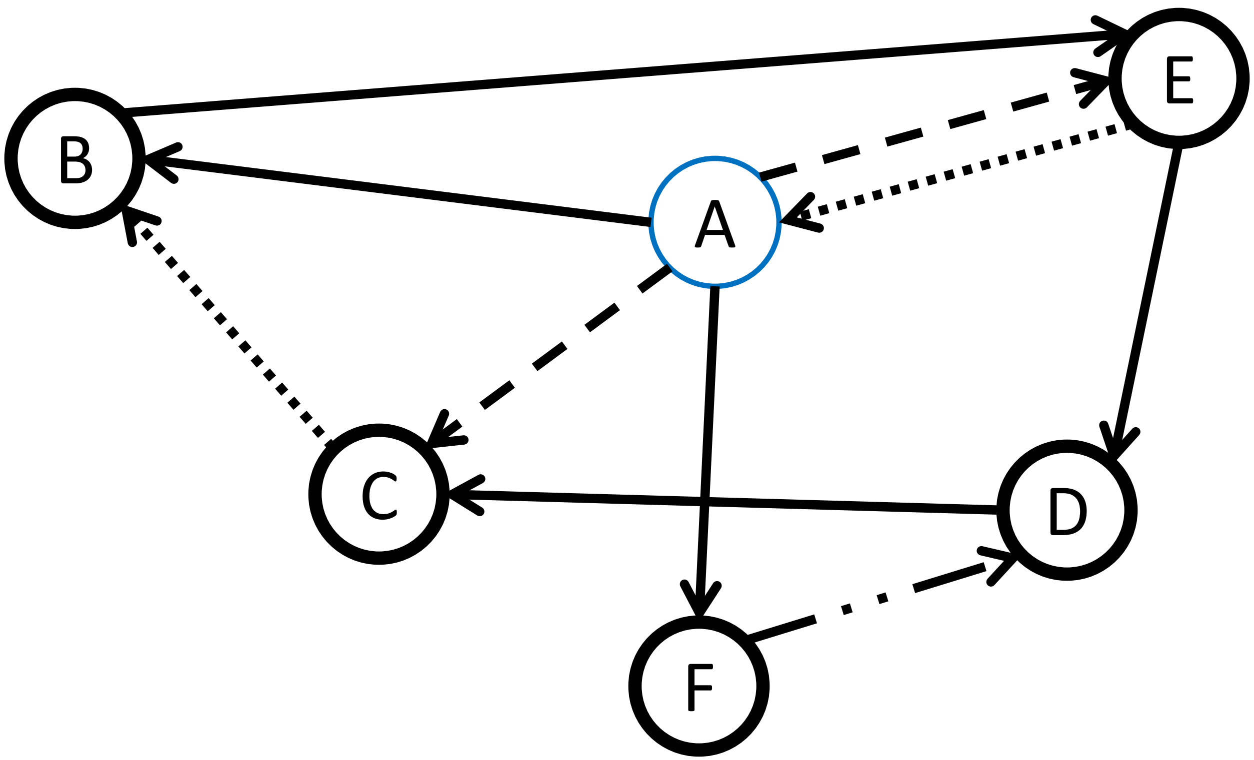
DFS Tree

- Rooted tree formed by only looking at the edges followed



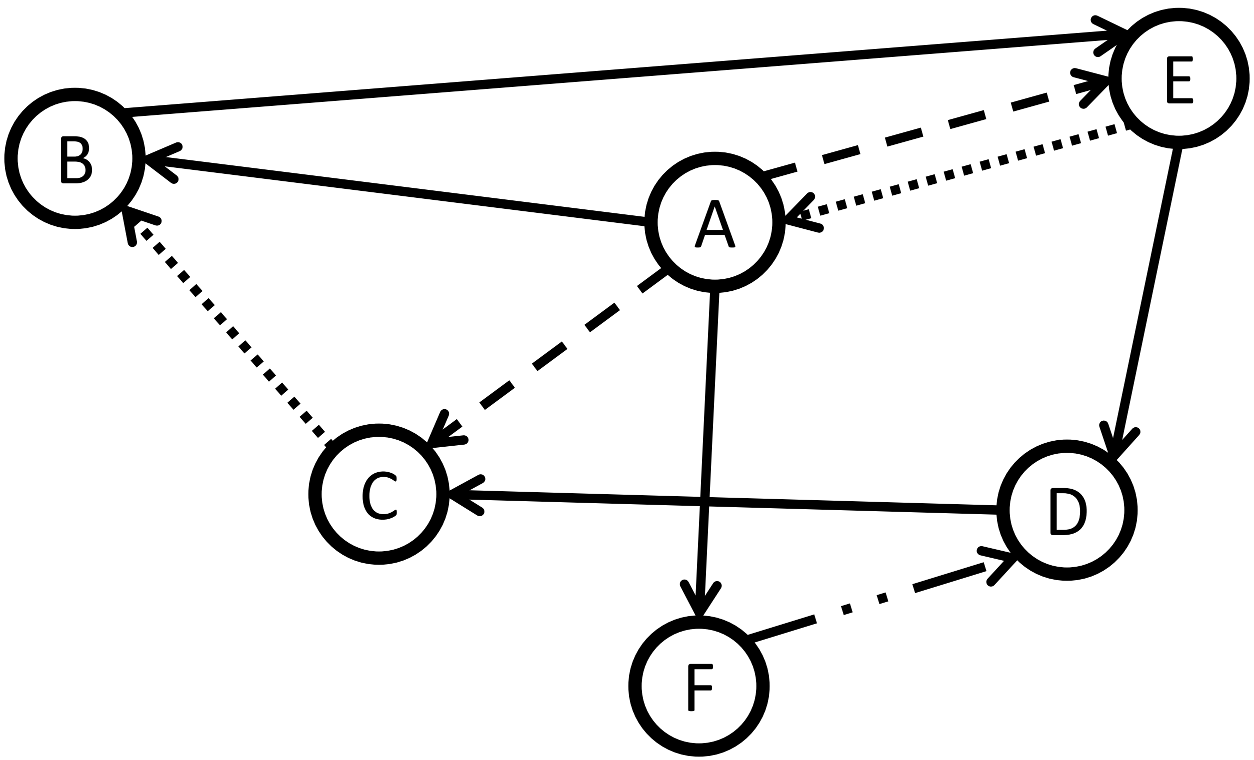
DFS Tree

- Rooted tree formed by only looking at the edges followed



DFS Tree

- Rooted tree formed by only looking at the edges followed

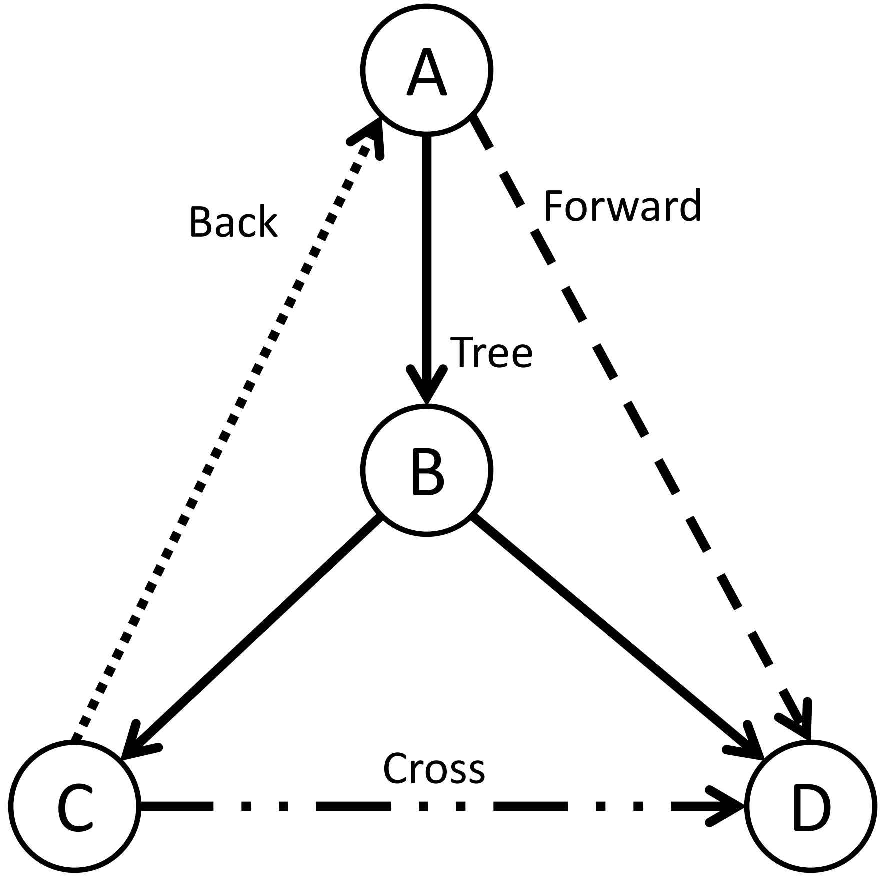


Types of Edges

- Tree edges: in DFS tree
- Forward edges: point to descendent in tree
- Back edges: point to ancestor in tree
- Cross edges: point to cousin in tree

- A dag is a graph with no back edges!
- How do we determine edge types?

Types of Edges



Previsit and Postvisit orderings

- initialize(): count = 0
- previsit(v): pre(v) = count; count++
- postvisit(v): post(v) = count; count++

Previsit and Postvisit orderings

- For each node v, there is an interval

$$I(v) = [pre(v),post(v)]$$

  - Represents time when v is on the stack
- Given an edge (u,v), what possible relationships are there for the intervals I(u), I(v)?

Previsit and Postvisit orderings

- What type of edge is (u,v)?
  - Tree edge/Forward ege: $I(v) \subset I(u)$
  - Back edge: $I(u) \subset I(v)$
  - Cross edge: $I(u)$ and $I(v)$ are disjoint, $I(v)$ before $I(u)$
- Last in, First out behavior guarantees that these are the only possibilities
- Can tell if graph is dag by looking at intervals

In a dag

- No back edges
- For edge (u,v):
  - Tree edge/Forward ege: $I(v) \subset I(u)$
  - Cross edge: $I(u)$ and $I(v)$ are disjoint, $I(v)$ before $I(u)$
- In either case, $post(v) < post(u)$
- Topological ordering: order by decreasing post values
- How to sort by post values?

Topological Ordering

- Run DFS with pre/post orderings
- Create an array of length 2|V|
- For each node v, put that node at index post(v)
- Starting from end of array, read off nodes stored in array
  - Ignore empty indicies

Run Time

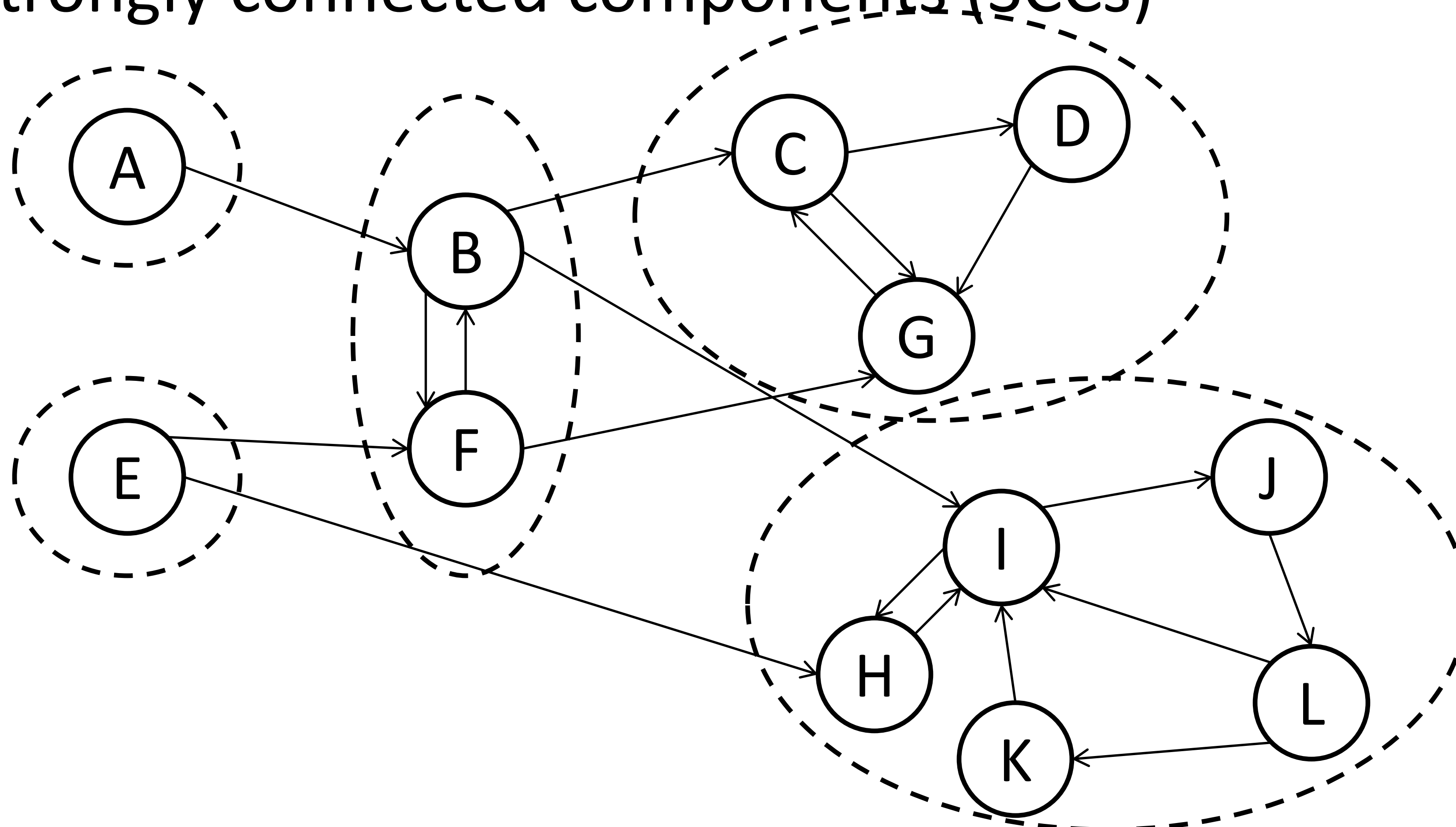
- DFS: $O(|V|+|E|)$
- Sorting: $O(|V|)$
- Total run time: $O(|V|+|E|)$
  - Linear!

Decomposition of Directed Graphs

- In undirected, used equivalence relation "connected" = a path from u to v.
- In directed, "connected" not an equivalence relation
  - u connected to v ≠ v connected to u
- Strongly connected: a path from u to v and a path from v to u

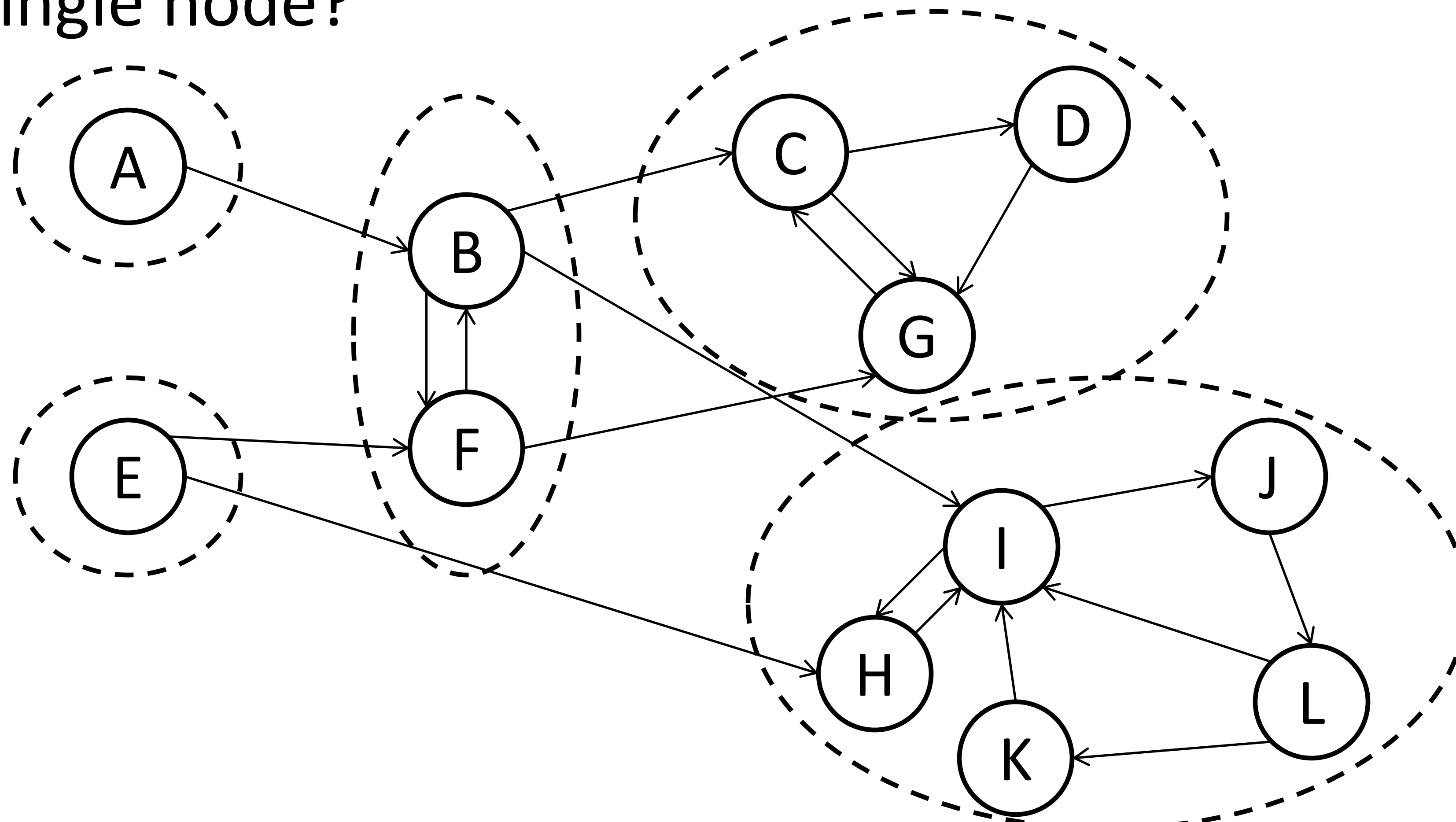
Strongly Connected Components

- Strongly connected induces equivalence class: strongly connected components (SCCs)



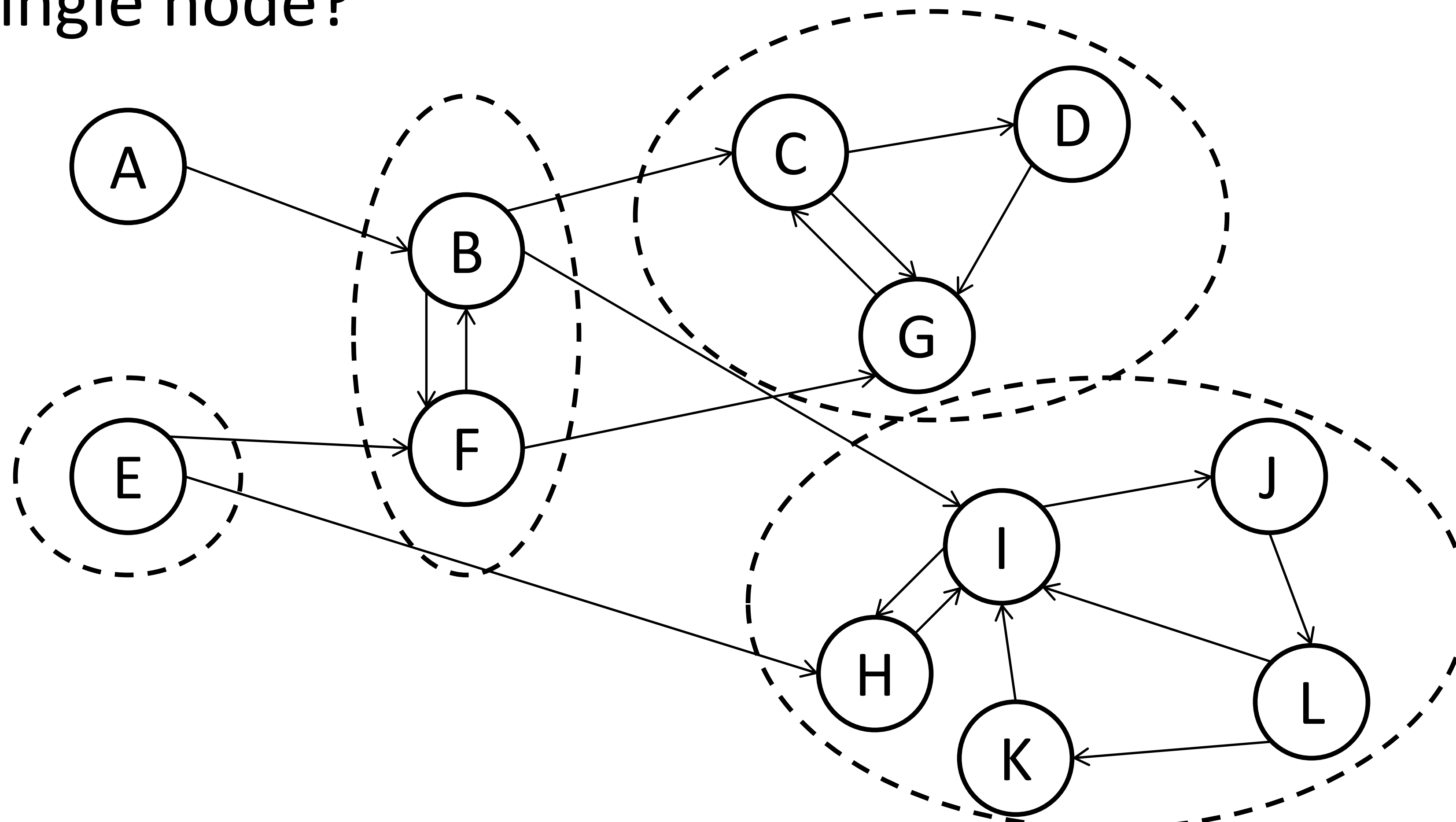
Strongly Connected Components

- What happens if we shrink each SCC into a single node?



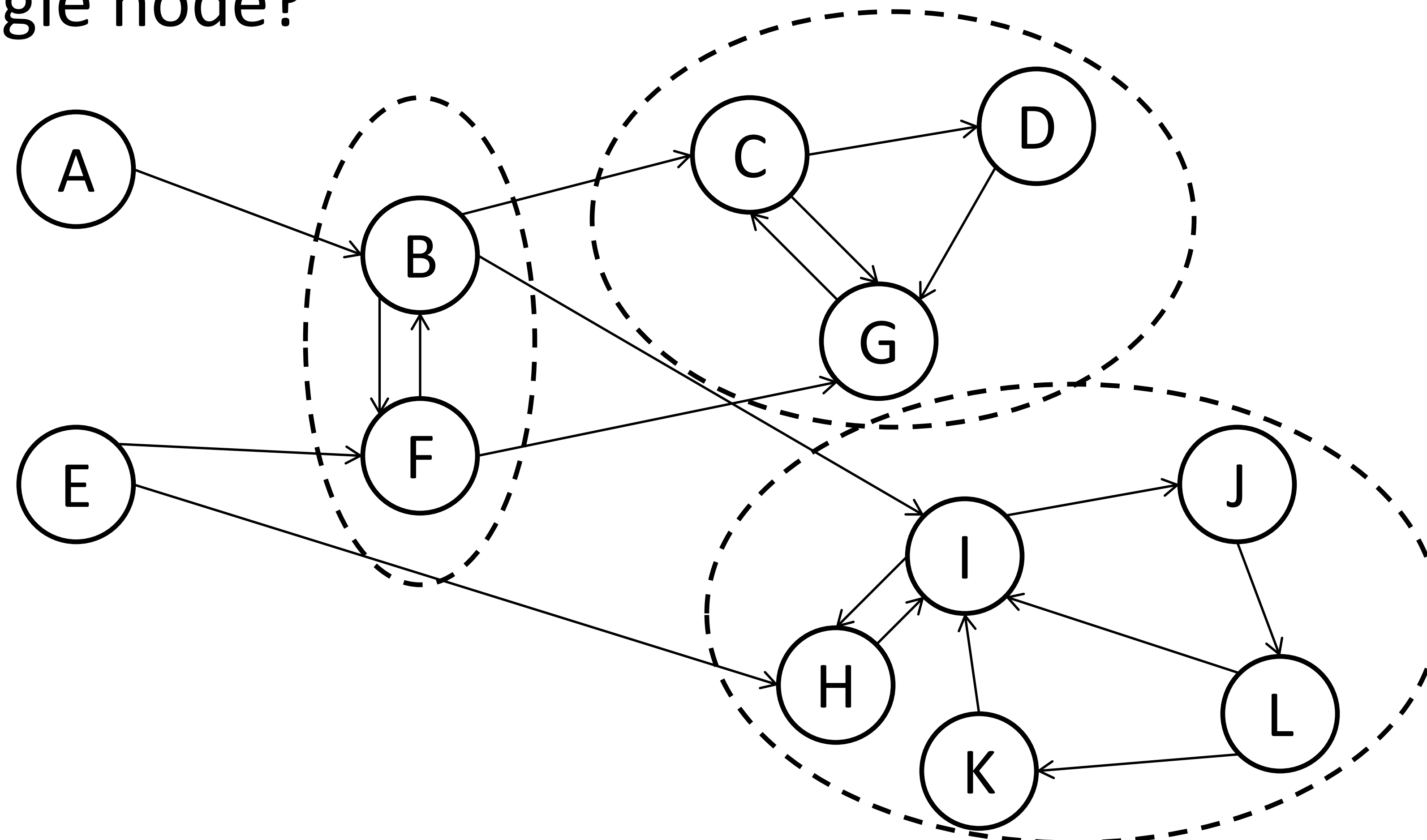
Strongly Connected Components

- What happens if we shrink each SCC into a single node?



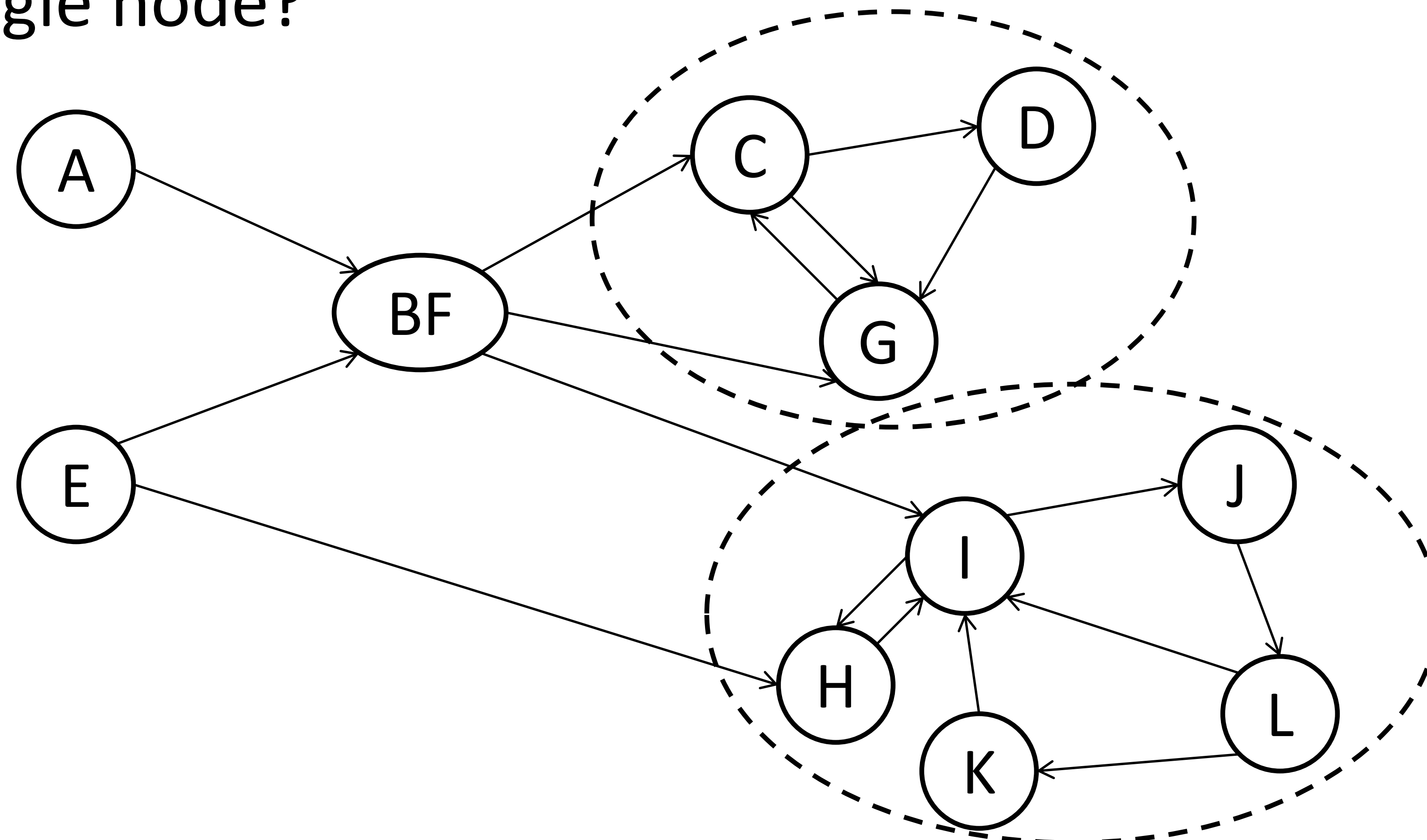
Strongly Connected Components

- What happens if we shrink each SCC into a single node?



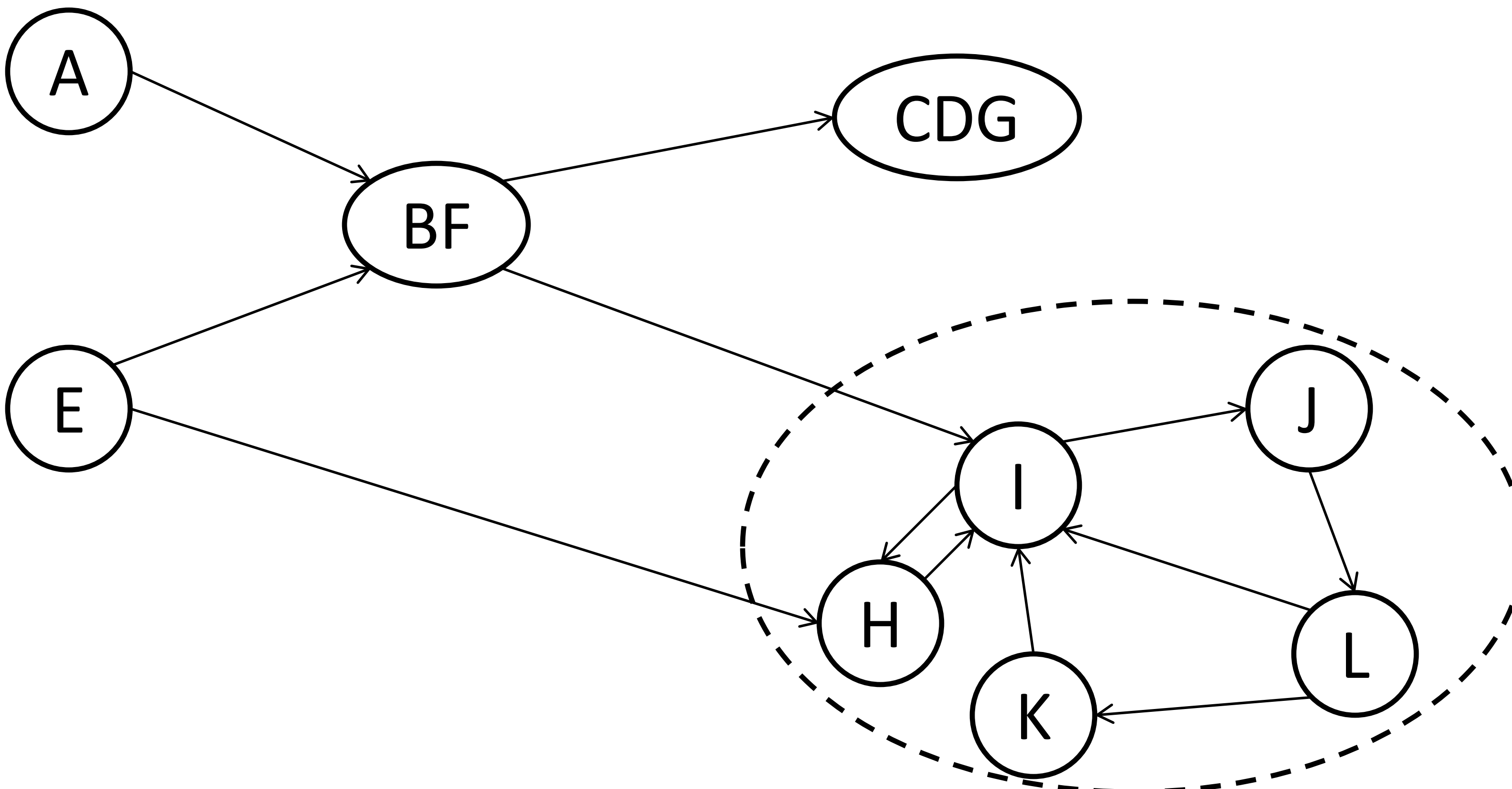
Strongly Connected Components

- What happens if we shrink each SCC into a single node?



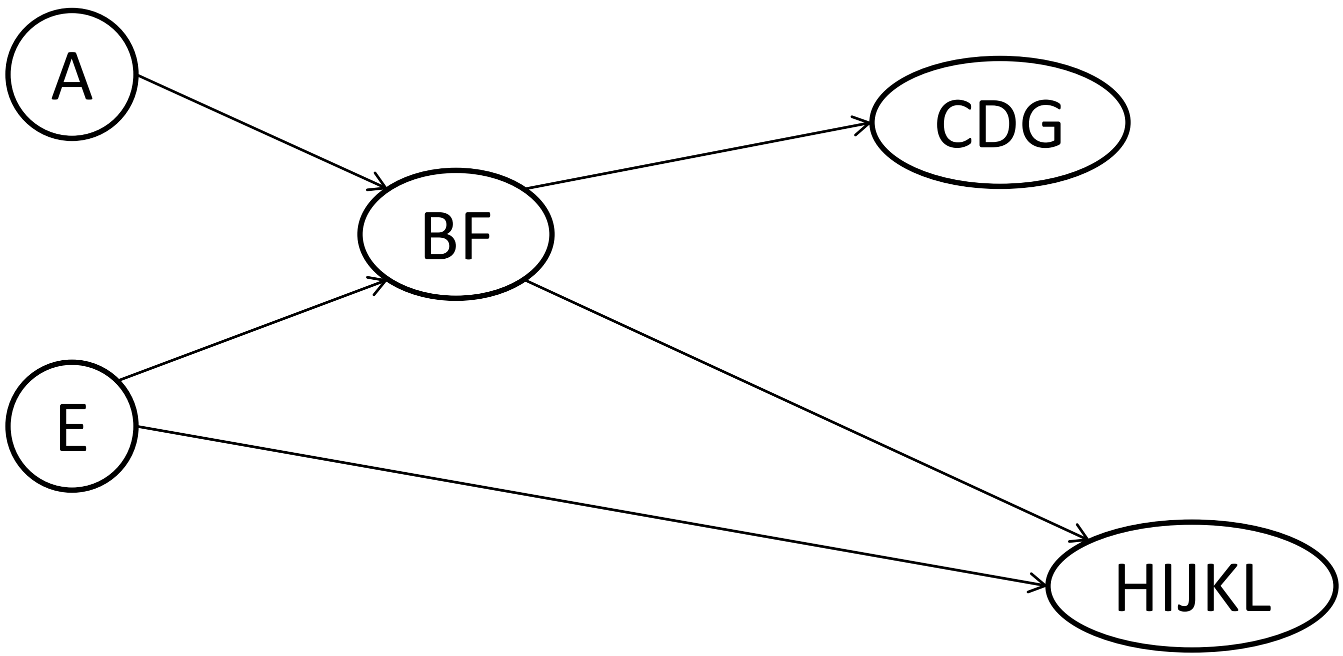
Strongly Connected Components

- What happens if we shrink each SCC into a single node?



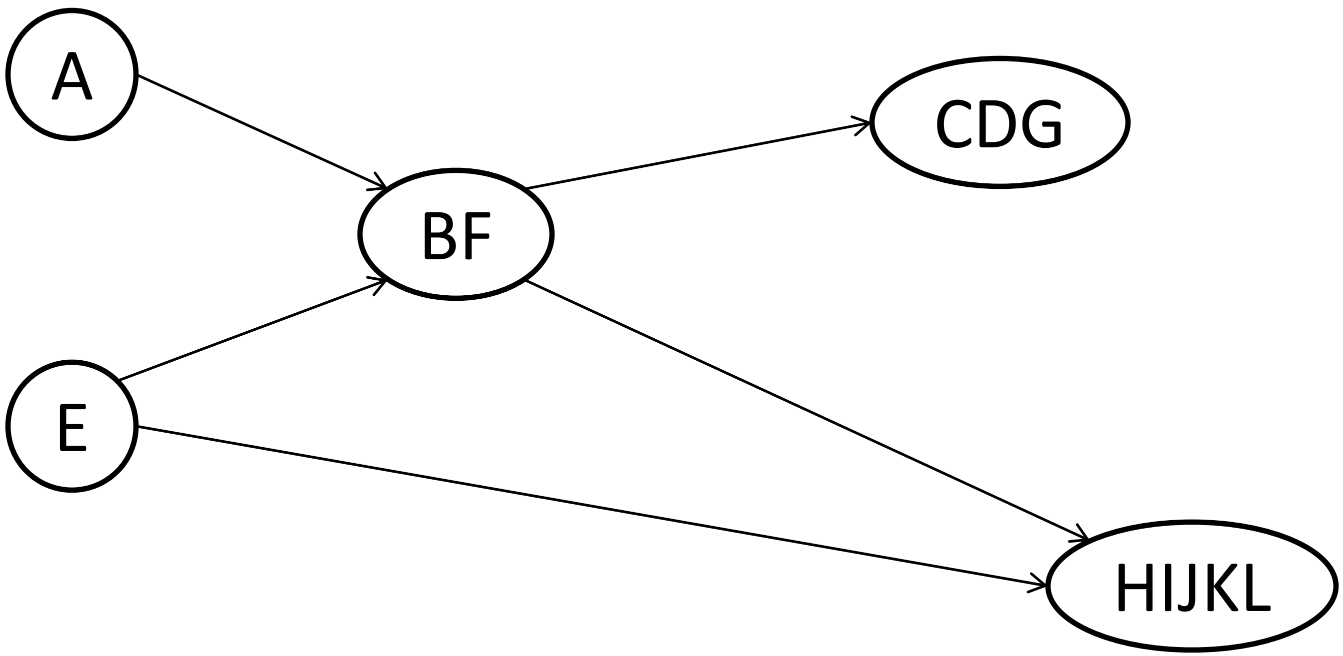
Strongly Connected Components

- What happens if we shrink each SCC into a single node?



Strongly Connected Components

- What happens if we shrink each SCC into a single node?



- We get a dag!

Strongly Connected Components

- Fact: Every graph is a dag of its strongly connected components
- Types of SCCs:
  - Source SCC: SCC is a source in the resulting dag
  - Sink SCC
- How do we find SCCs and construct this dag?
- Answer: DFS

Strongly Connected Components

- Recall: in dag, for edge (u,v), u must have higher post order
- Theorem: If C and C’ are two SCCs and there is an edge from C to C’, then the highest post number in C is larger than highest post number in C’

Proof

- If DFS visits C’ before C, it will get stuck visiting C’, and C will get visited on a later call to explore, resulting in a higher post number
- If DFS visits C before C’, all of C and C’ will be visited before explore gets stuck, so the first node in C will have higher post order than any in C’

Corollary

- Node with highest post number must lie in source SCC

Strongly Connected Components

- If we know the SCCs, we can order them by decreasing highest post number
- How do we find SCCs?
  - If we have a node u in a sink SCC, run explore on u.
  - Nodes visited will be exactly the sink SCC
  - Remove this SCC from graph, and repeat

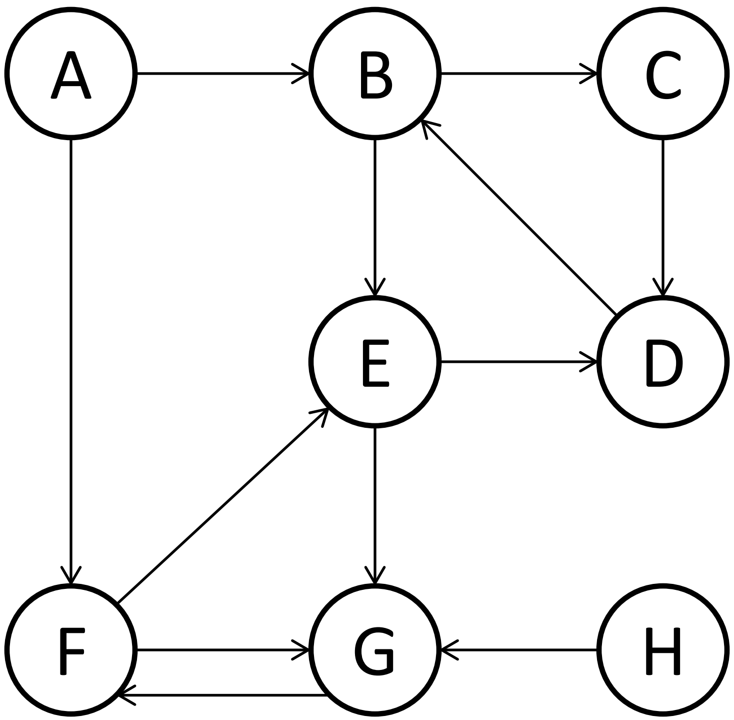
Strongly Connected Components

- We know how to find source SCCs, need sink SCCs.
- Reverse graph: G^R is the graph obtained by reversing every edge in G.
- If we run DFS on G^R, highest post number will be in a sink SCC

Strongly Connected Components

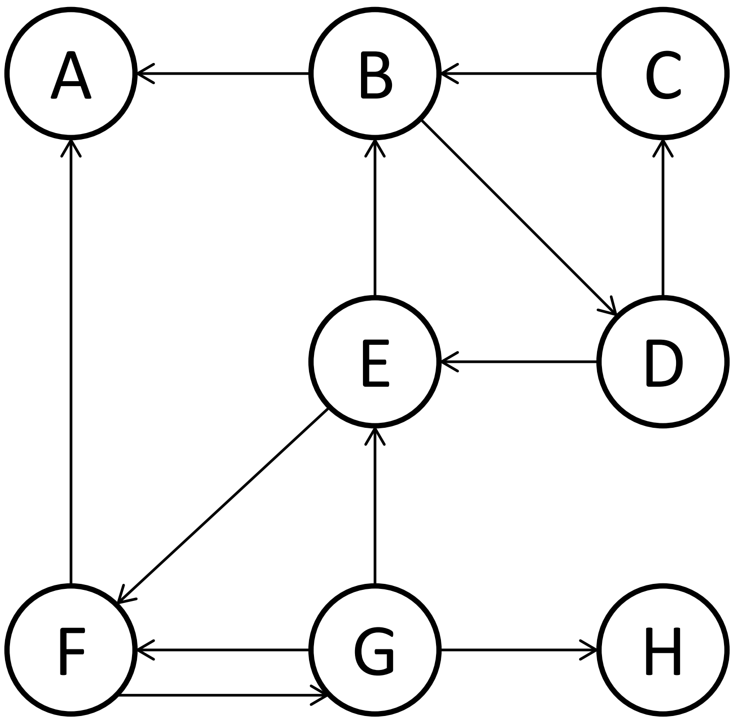
- Algorithm:
  - Run DFS on G^R, keeping track of post numbers
  - Run DFS on G in order of decreasing post numbers
    - Keep track of ccnum (will be label for SCC)
    - Whenever we look at an edge to a visited node, see what SCC it goes to, and add edge in the dag

Strongly Connected Components



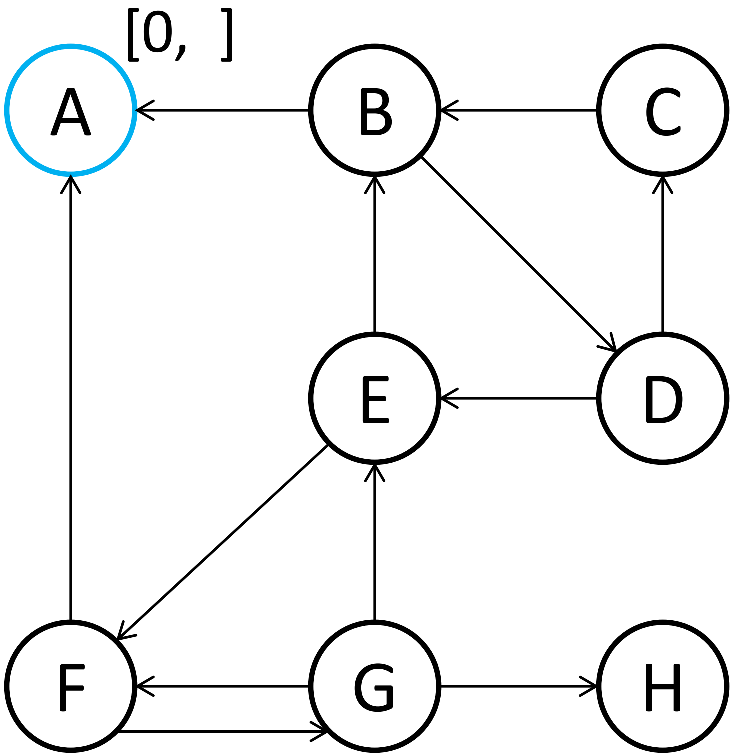
Strongly Connected Components

- DFS on G^R



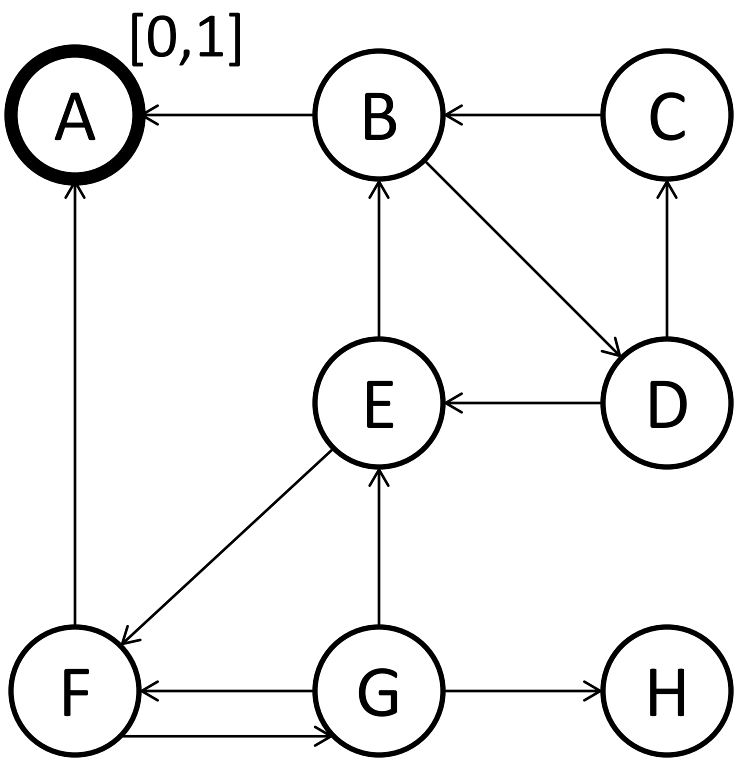
Strongly Connected Components

- DFS on G^R



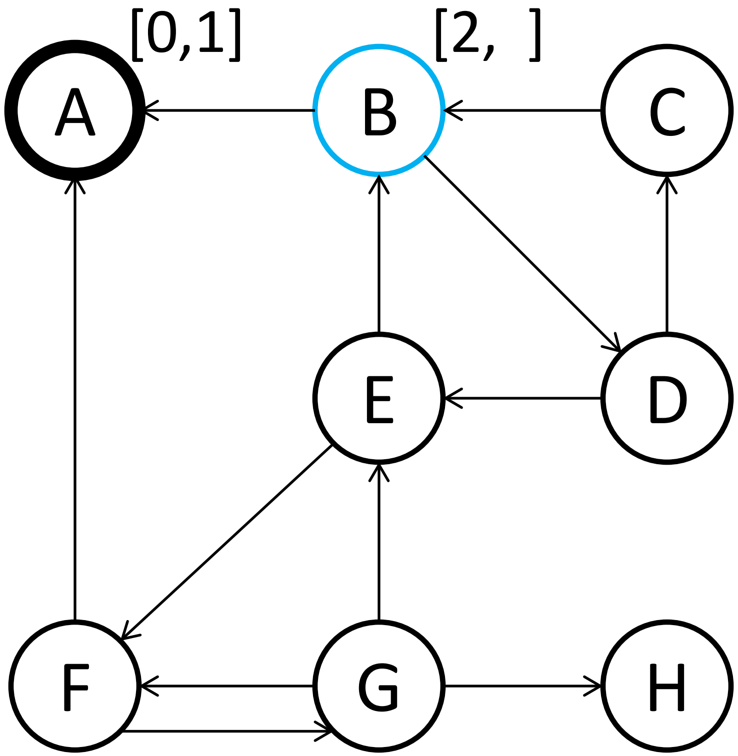
Strongly Connected Components

- DFS on G^R



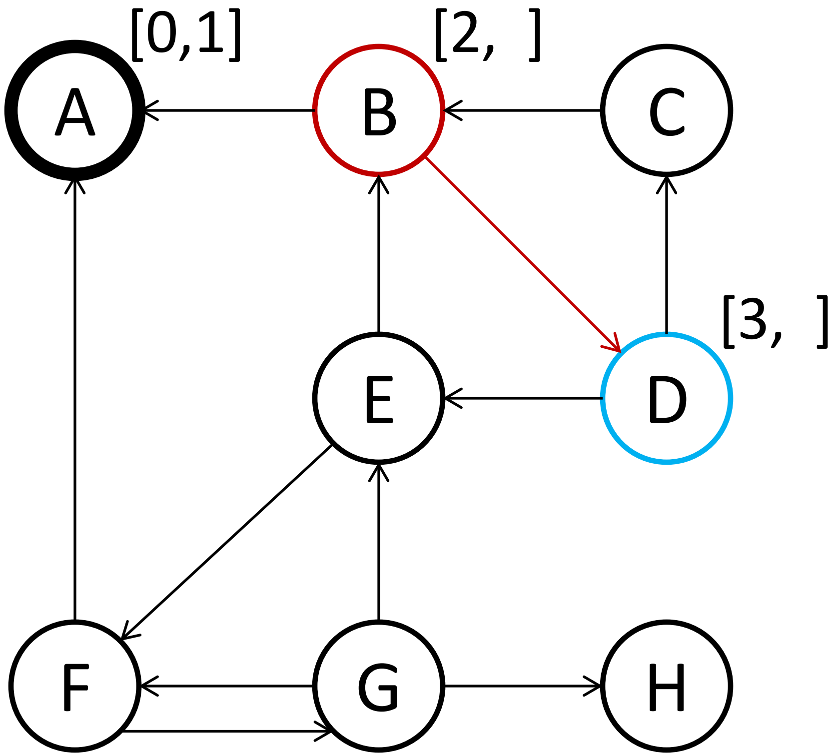
Strongly Connected Components

- DFS on G^R



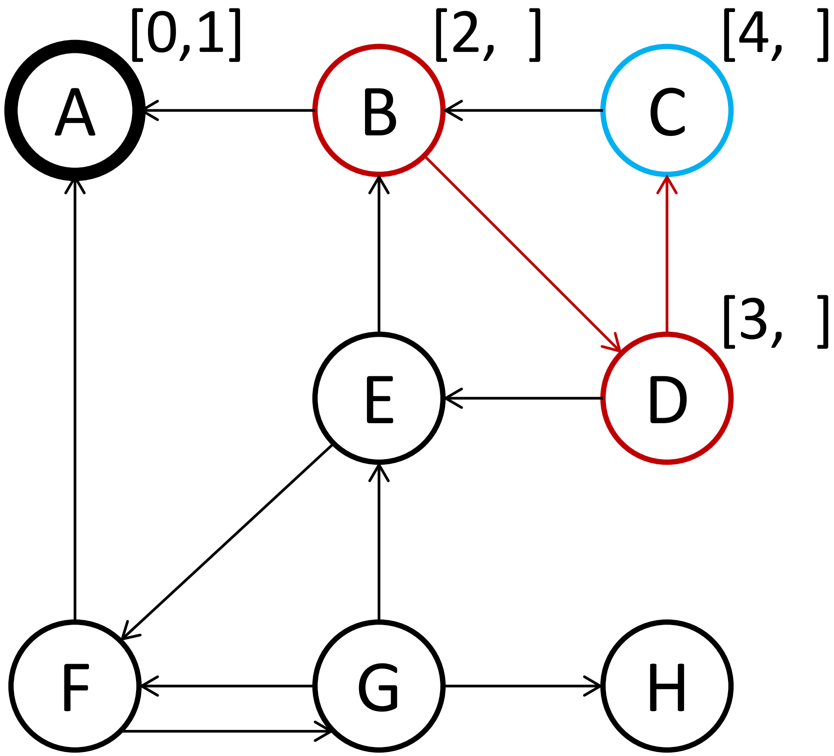
Strongly Connected Components

- DFS on G^R



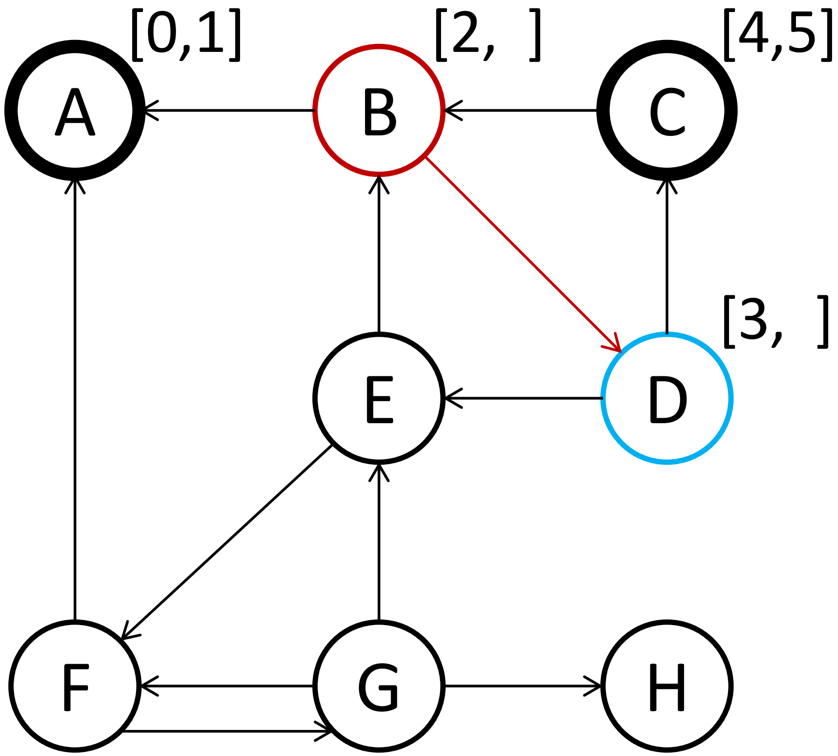
Strongly Connected Components

- DFS on G^R



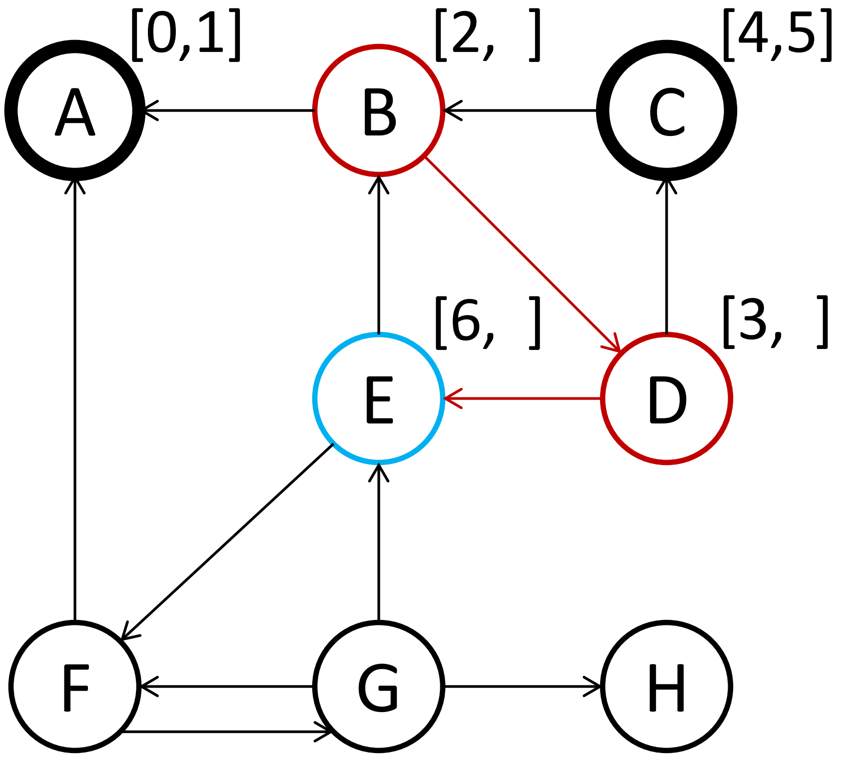
Strongly Connected Components

- DFS on G^R



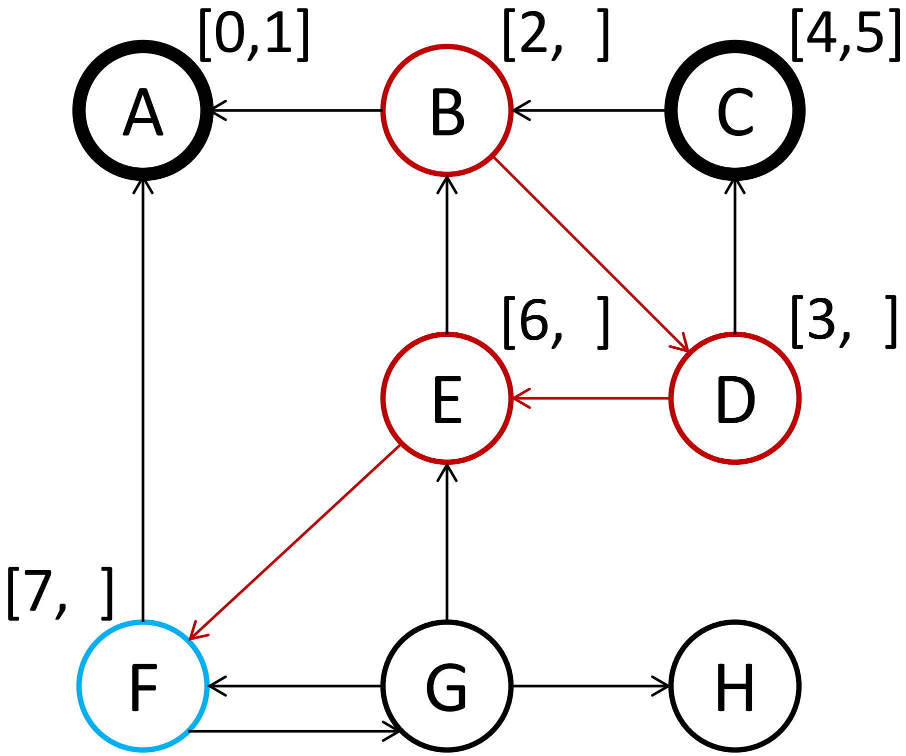
Strongly Connected Components

- DFS on G^R



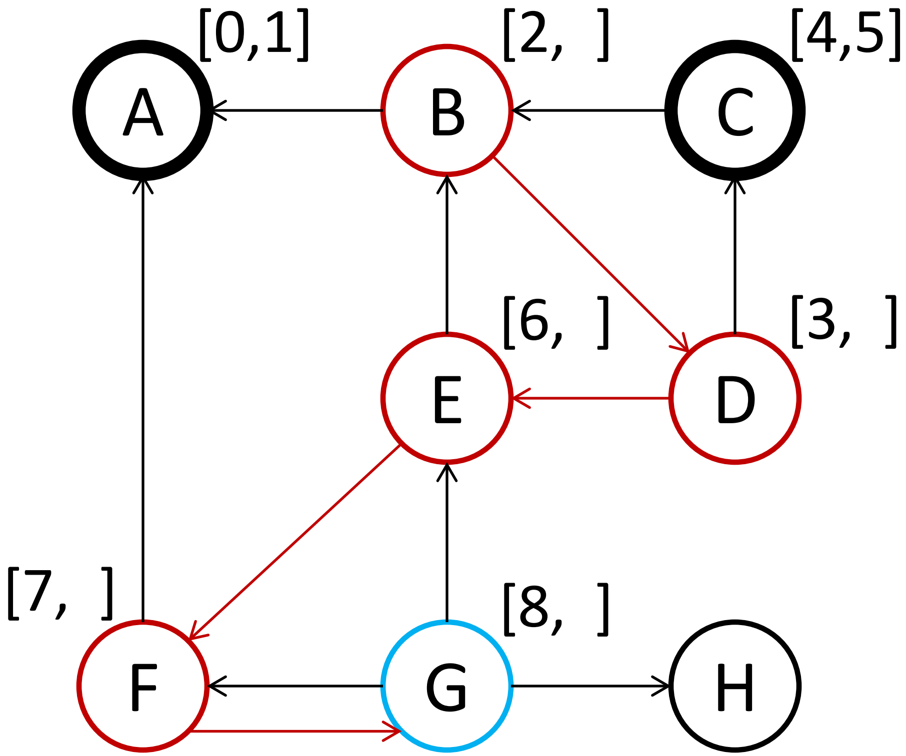
Strongly Connected Components

- DFS on G^R



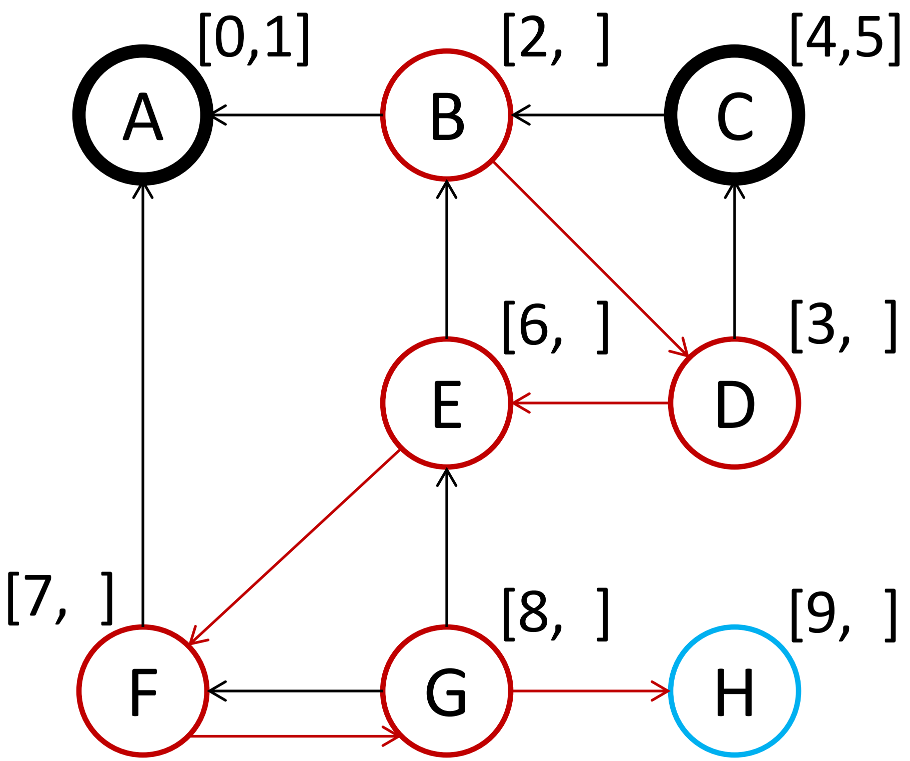
Strongly Connected Components

- DFS on G^R



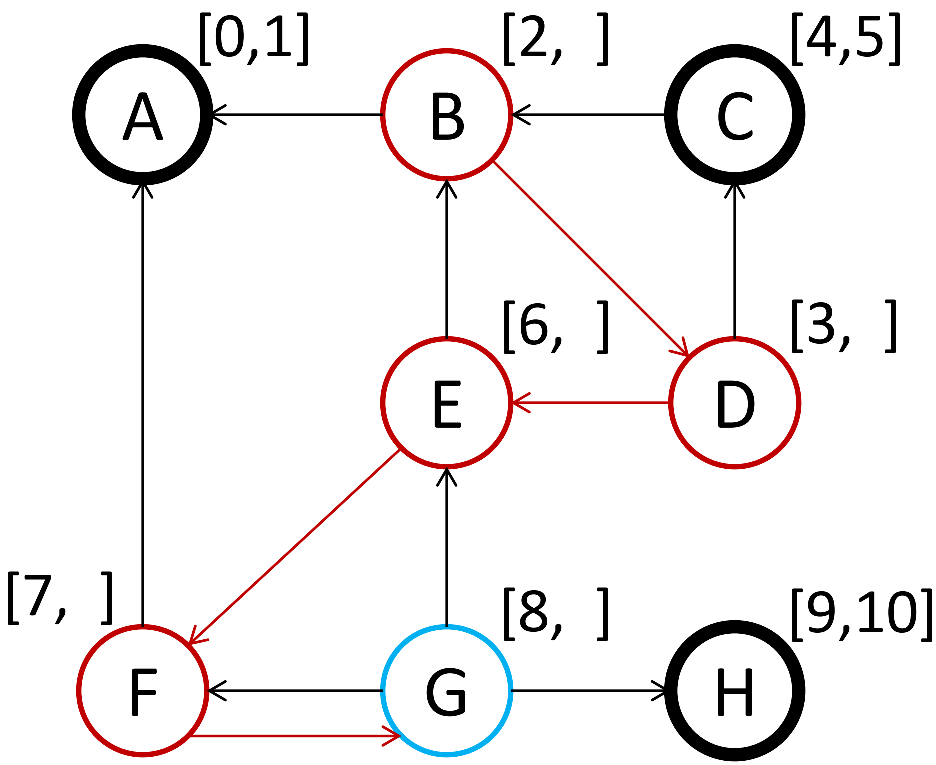
Strongly Connected Components

- DFS on G^R



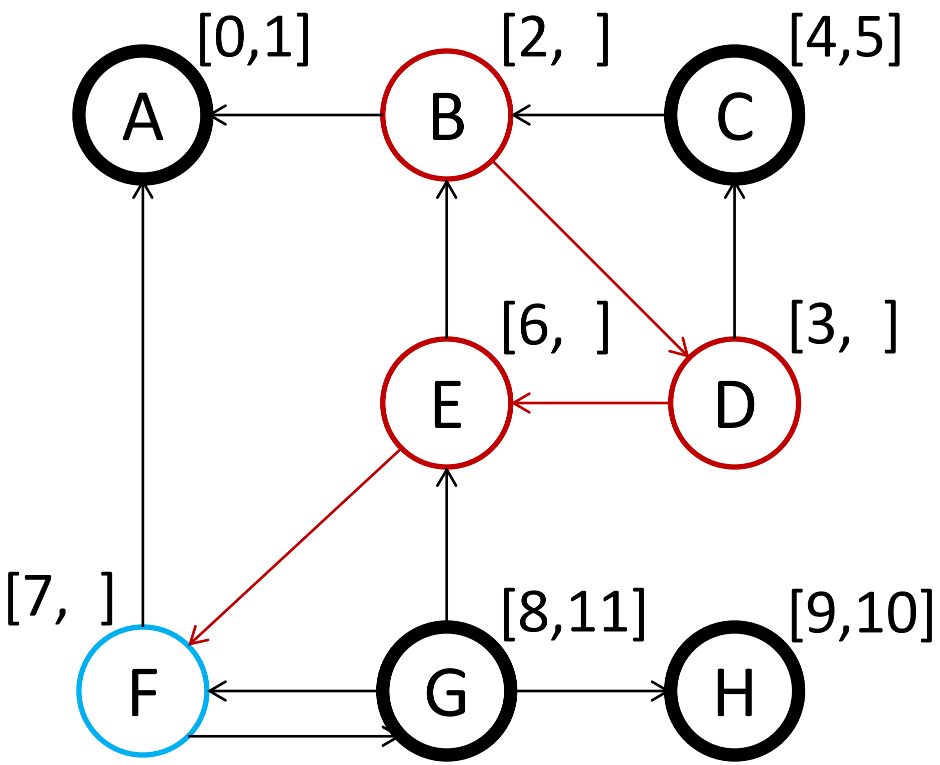
Strongly Connected Components

- DFS on G^R



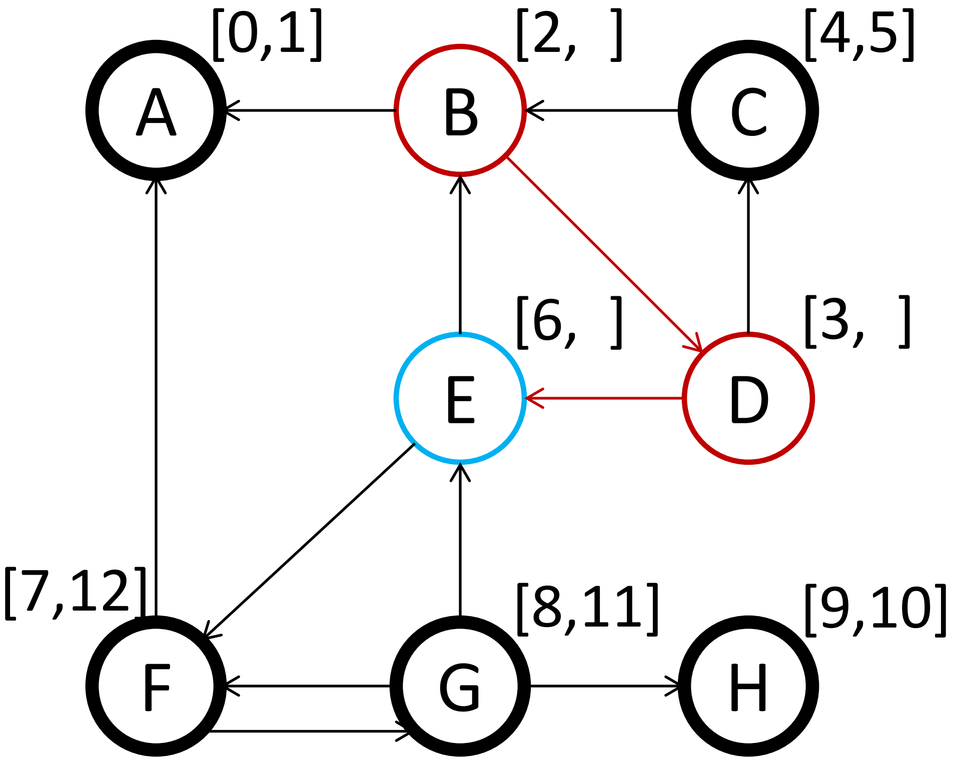
Strongly Connected Components

- DFS on G^R



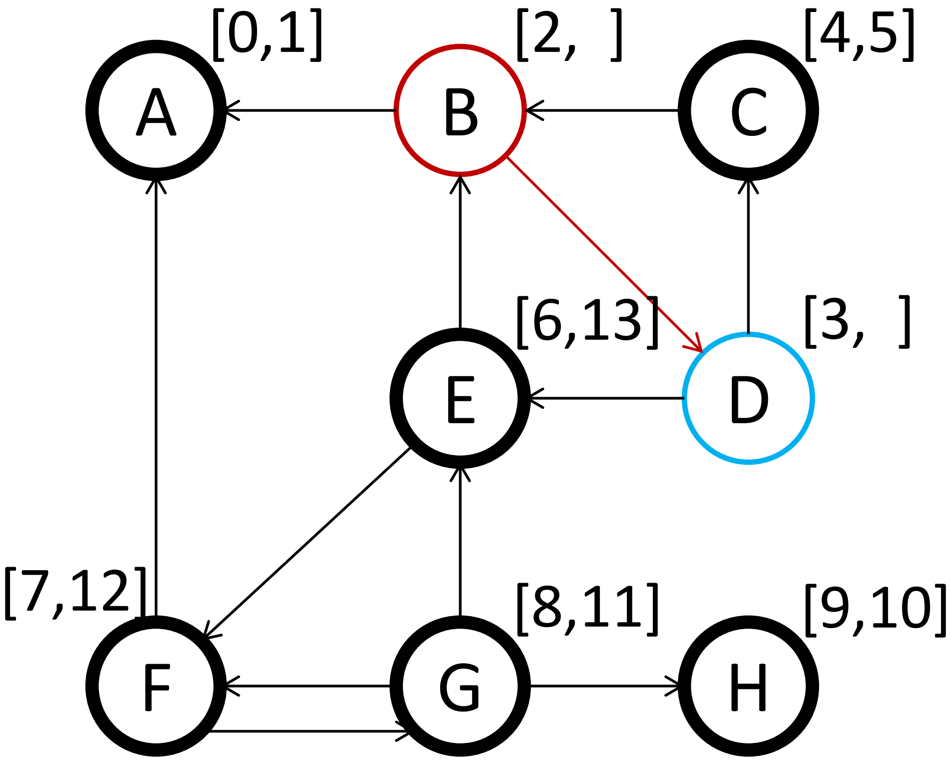
Strongly Connected Components

- DFS on G^R



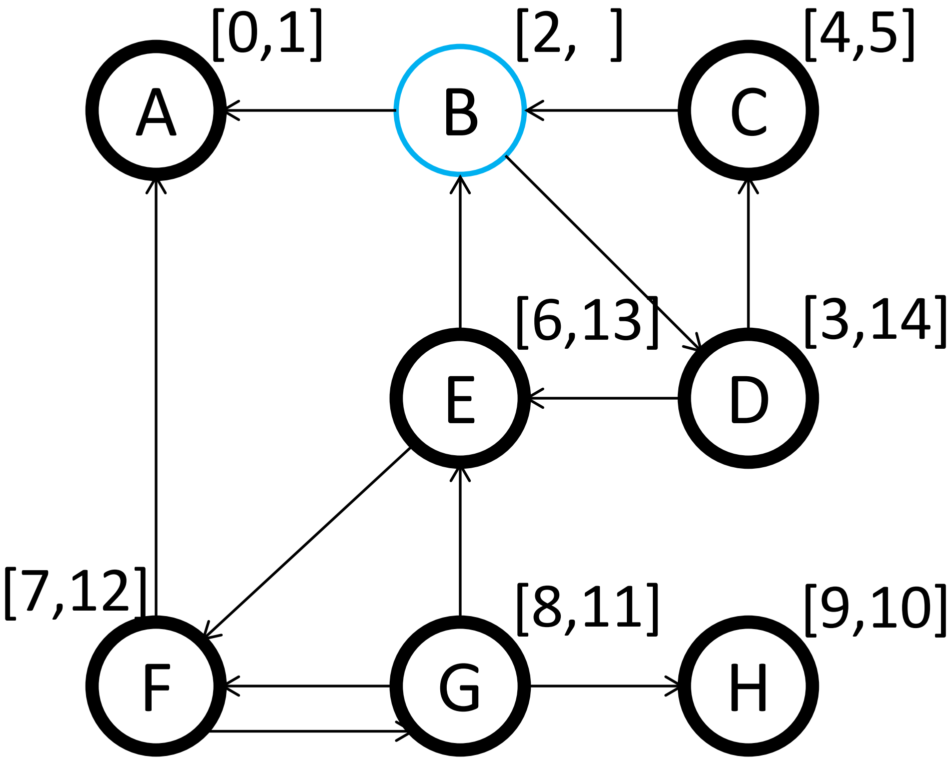
Strongly Connected Components

- DFS on G^R



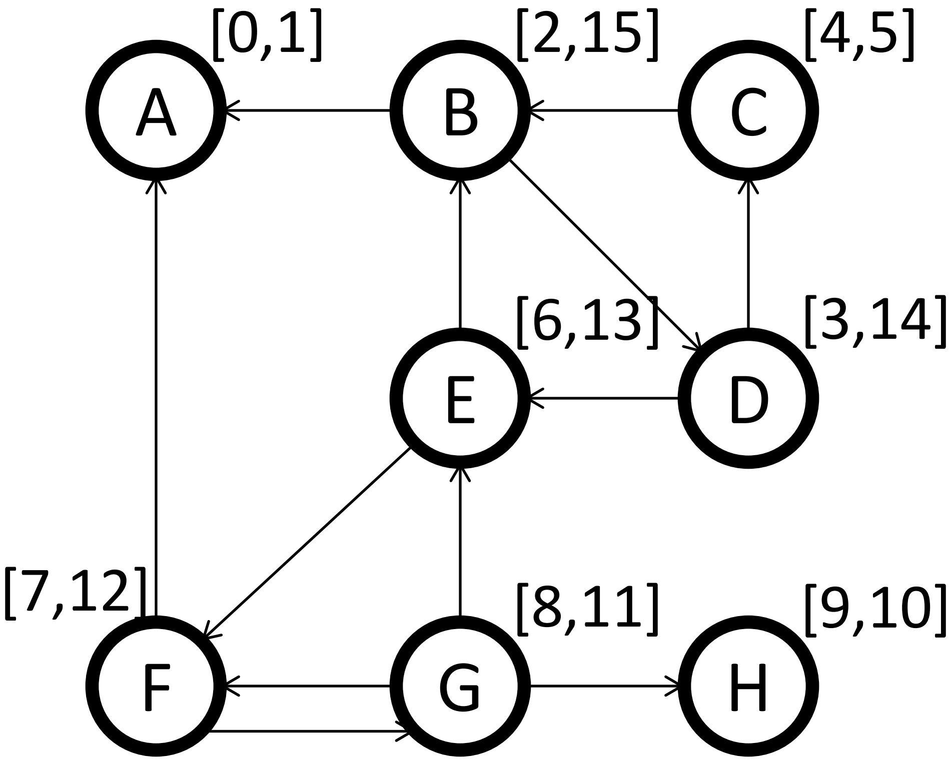
Strongly Connected Components

- DFS on G^R



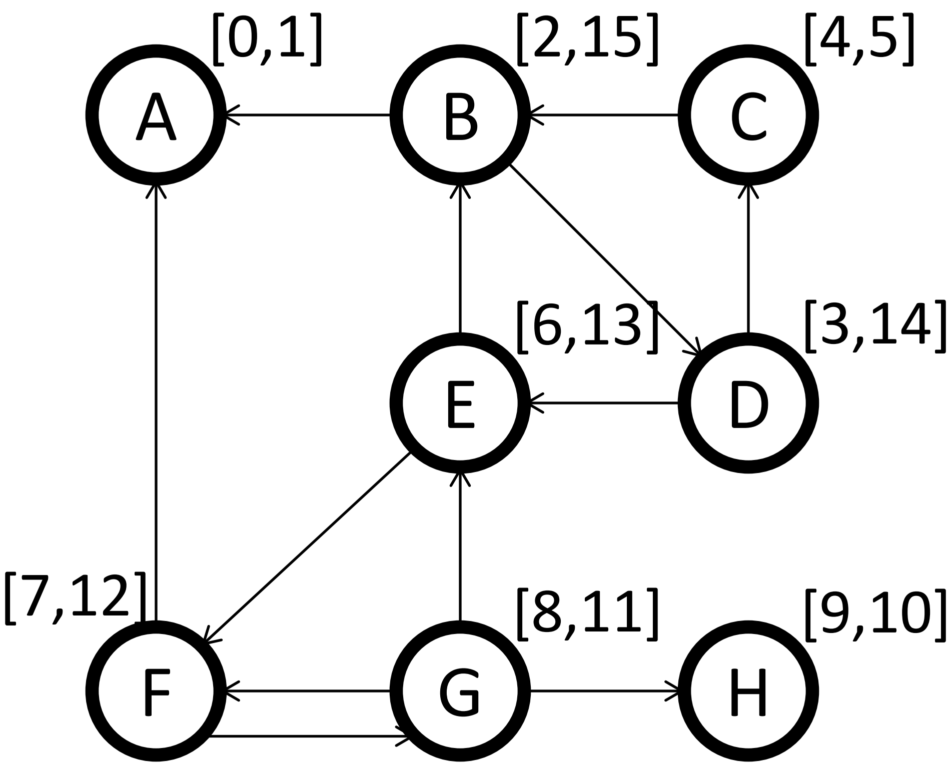
Strongly Connected Components

- DFS on G^R



Strongly Connected Components

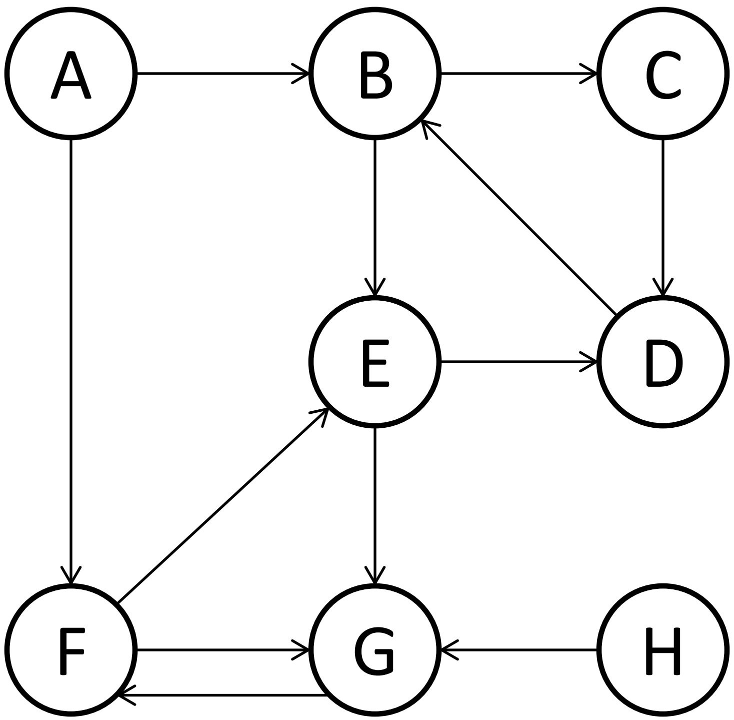
- DFS on G^R



- Order: BDEFGHCA

Strongly Connected Components

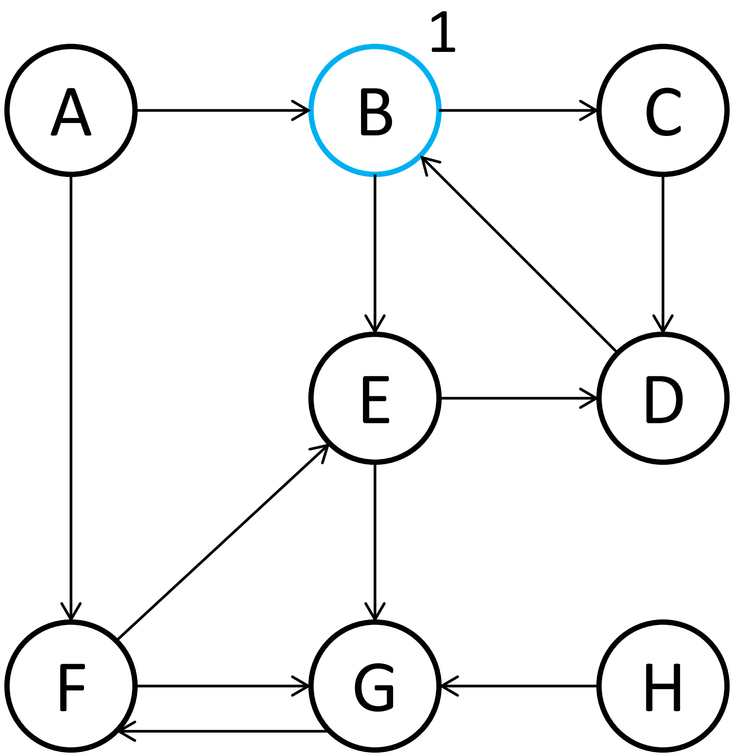
- DFS on G



- Order: BDEFGHCA

Strongly Connected Components

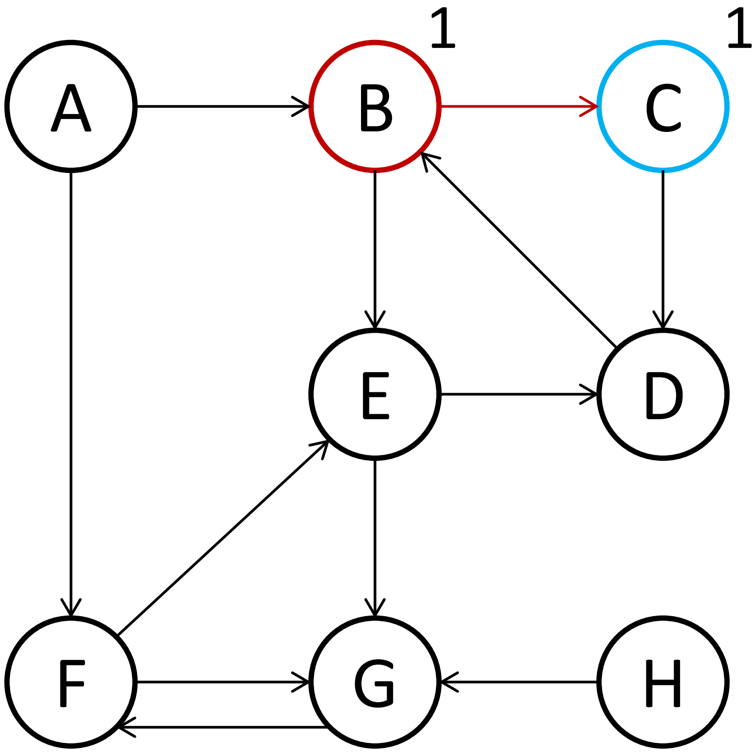
- DFS on G



- Order: BDEFGHCA

Strongly Connected Components

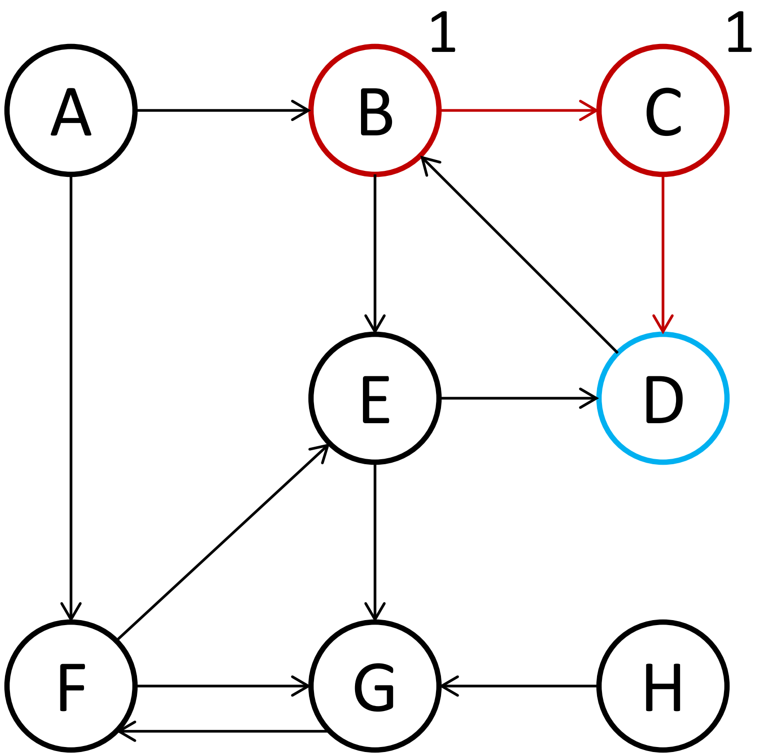
- DFS on G



- Order: BDEFGHCA

Strongly Connected Components

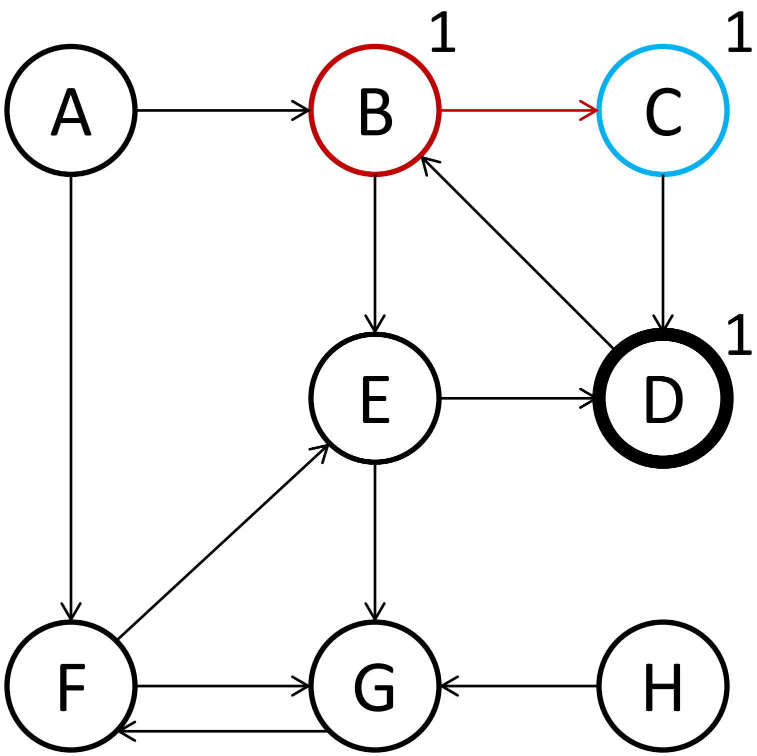
- DFS on G



- Order: BDEFGHCA

Strongly Connected Components

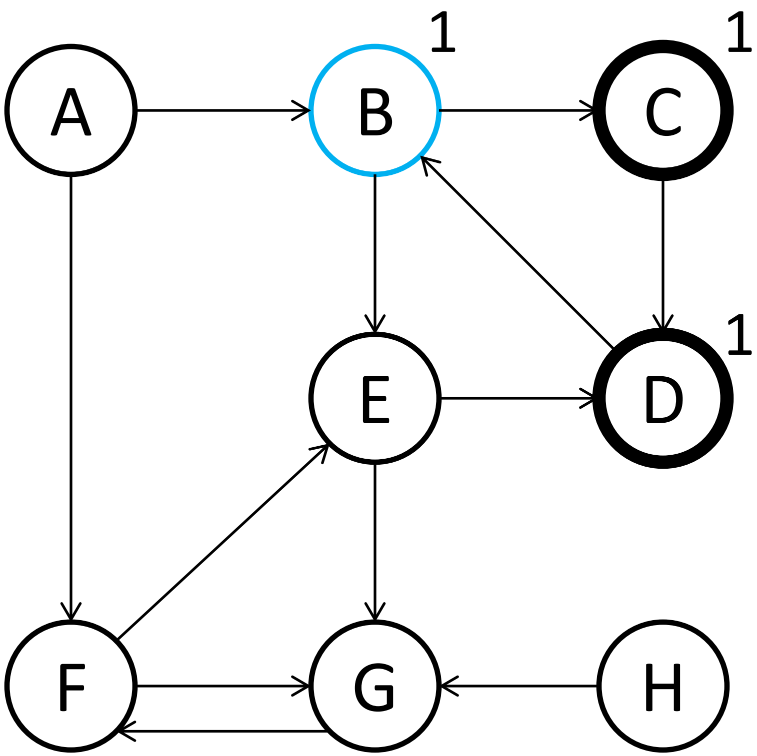
- DFS on G



- Order: BDEFGHCA

Strongly Connected Components

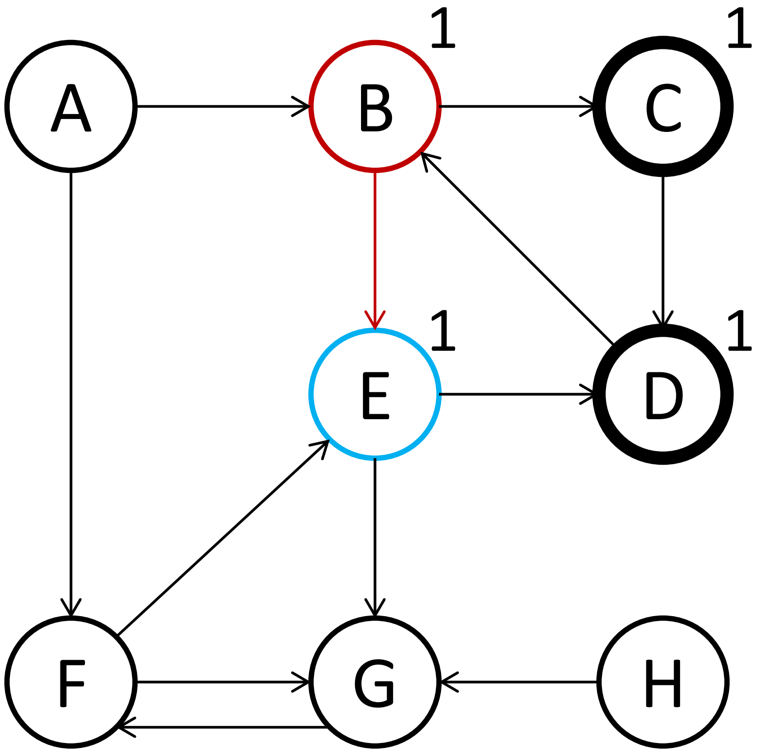
- DFS on G



- Order: BDEFGHCA

Strongly Connected Components

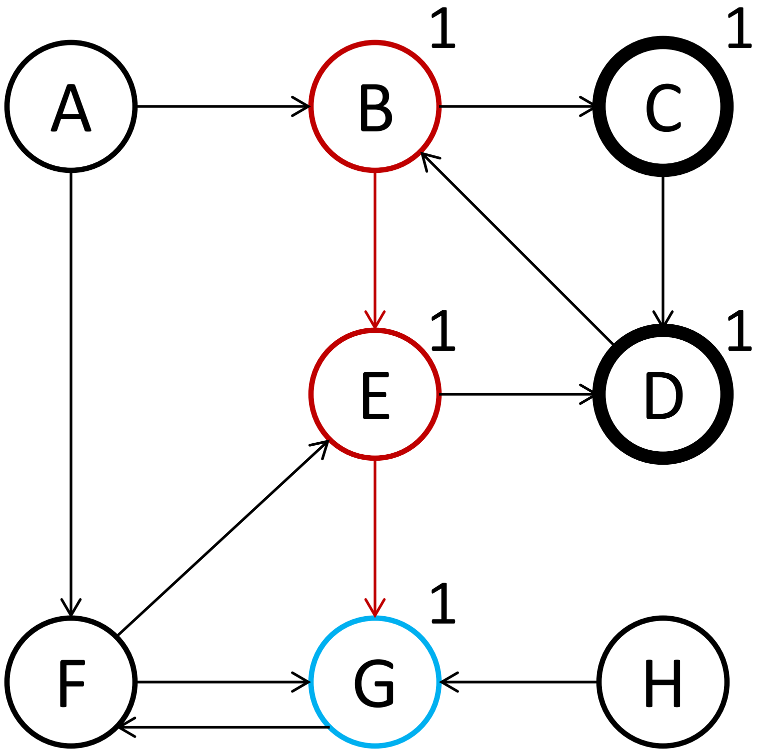
- DFS on G



- Order: BDEFGHCA

Strongly Connected Components

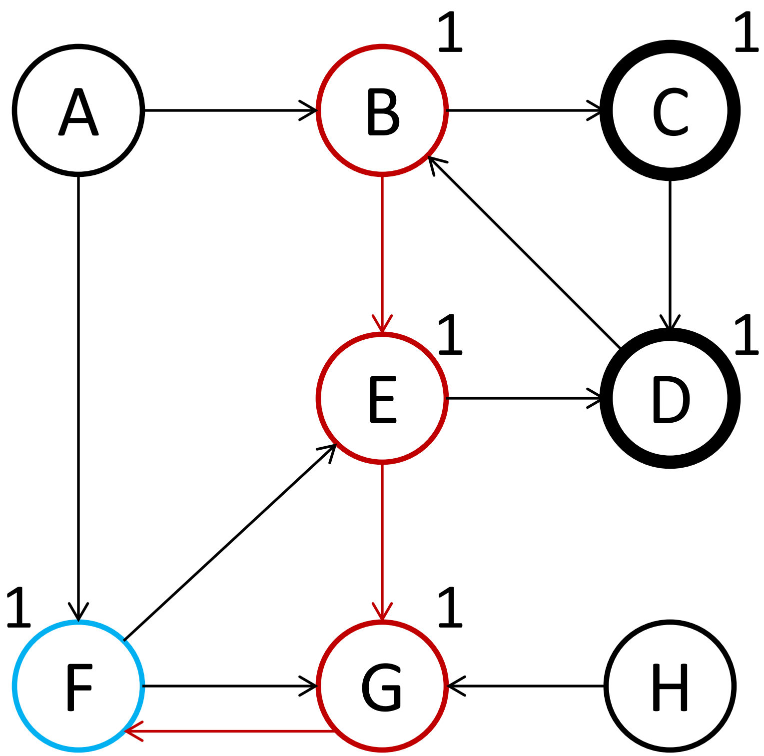
- DFS on G



- Order: BDEFGHCA

Strongly Connected Components

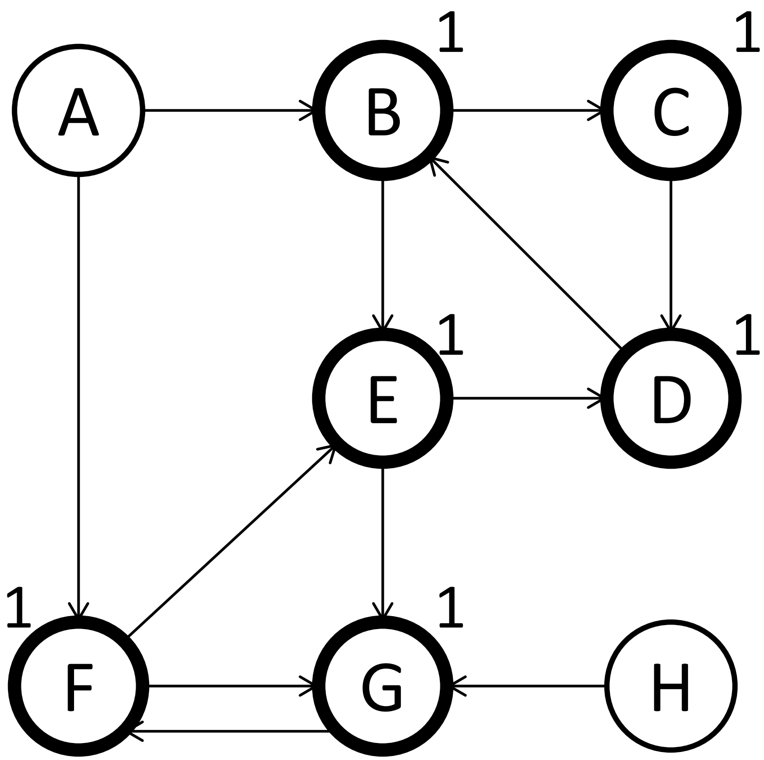
- DFS on G



- Order: BDEFGHCA

Strongly Connected Components

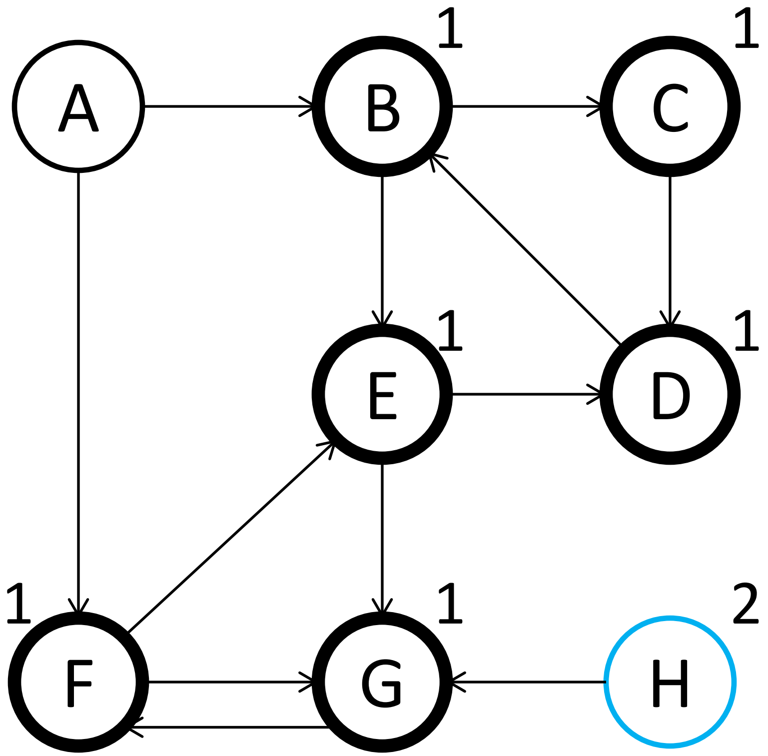
- DFS on G



- Order: BDEFGHCA

Strongly Connected Components

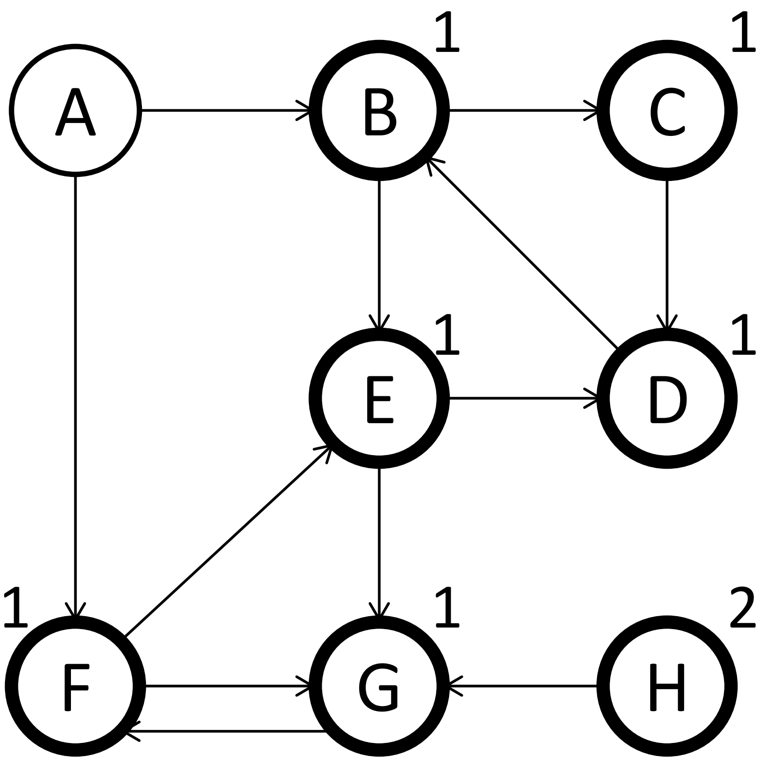
- DFS on G

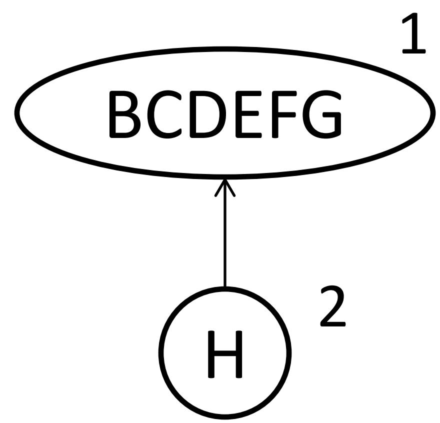


- Order: BDEFGHCA

Strongly Connected Components

- DFS on G

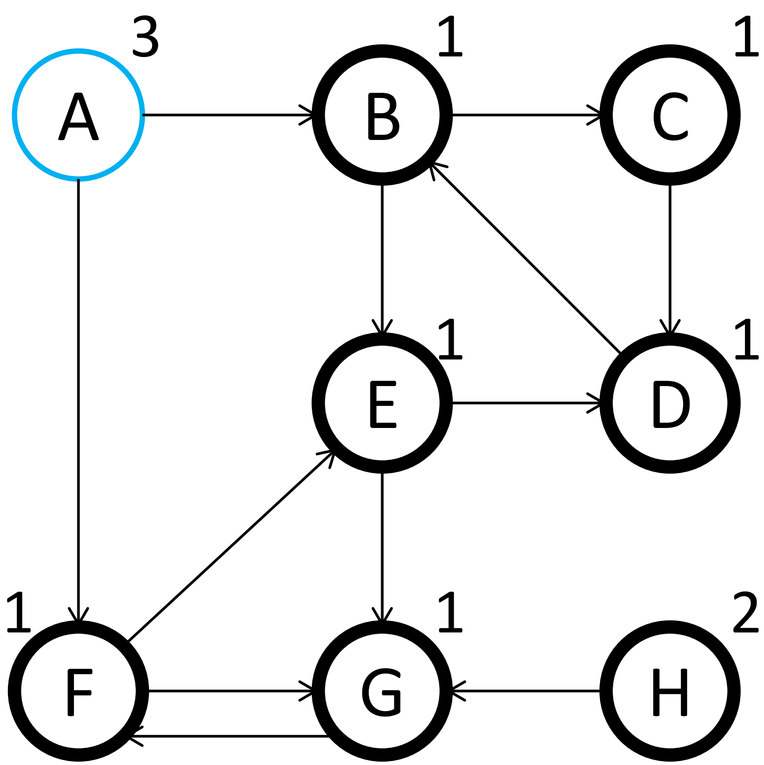


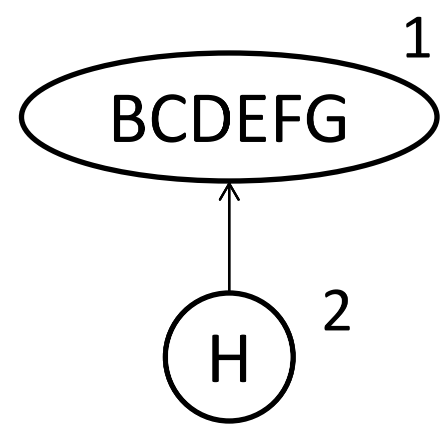


- Order: BDEFGHCA

Strongly Connected Components

- DFS on G

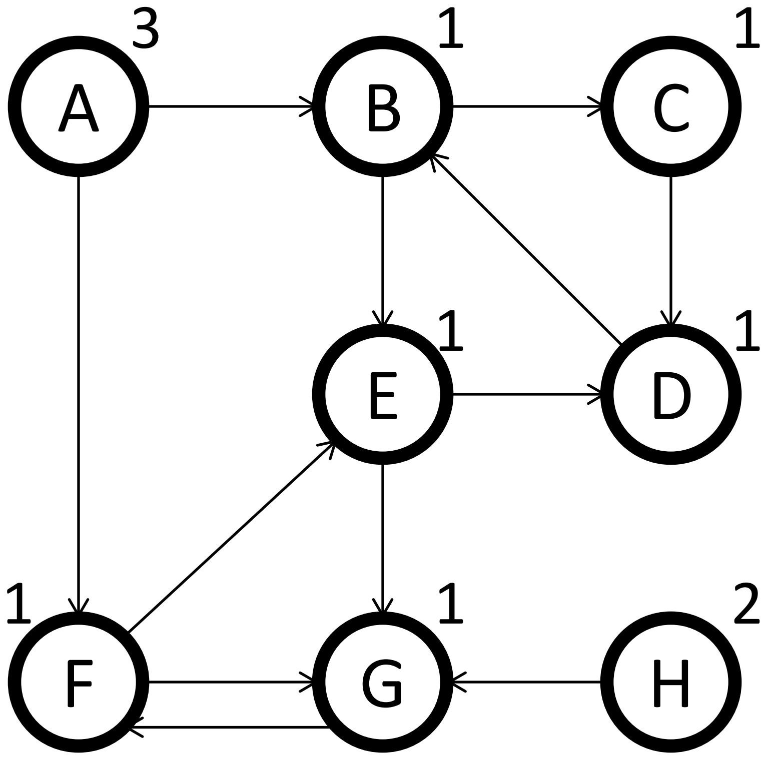


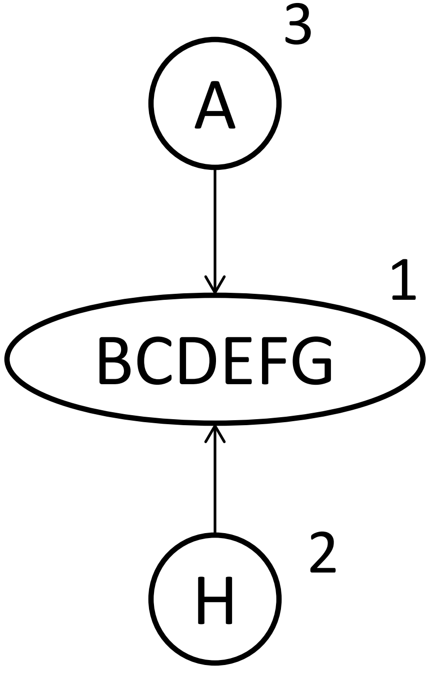


- Order: BDEFGHCA

Strongly Connected Components

- DFS on G





- Order: BDEFGHCA

Running Time

- $O(|V|+|E|)$ each for two runs of DFS
- $O(|V|+|E|)$ overall