CS 361
Data Structures & Algs
Lecture 14

Prof. Tom Hayes
University of New Mexico
10-7-2010

Last Time

- Inside BFS
  - Running Time
  - Implementation
  - Degrees & Degree sums
  - Properties of BFS and DFS trees
- Identifying trees in the wild

Today

More tree forensics

READING: Finish reading Chapter 3

identifying BFS & DFS trees

Where is the root?

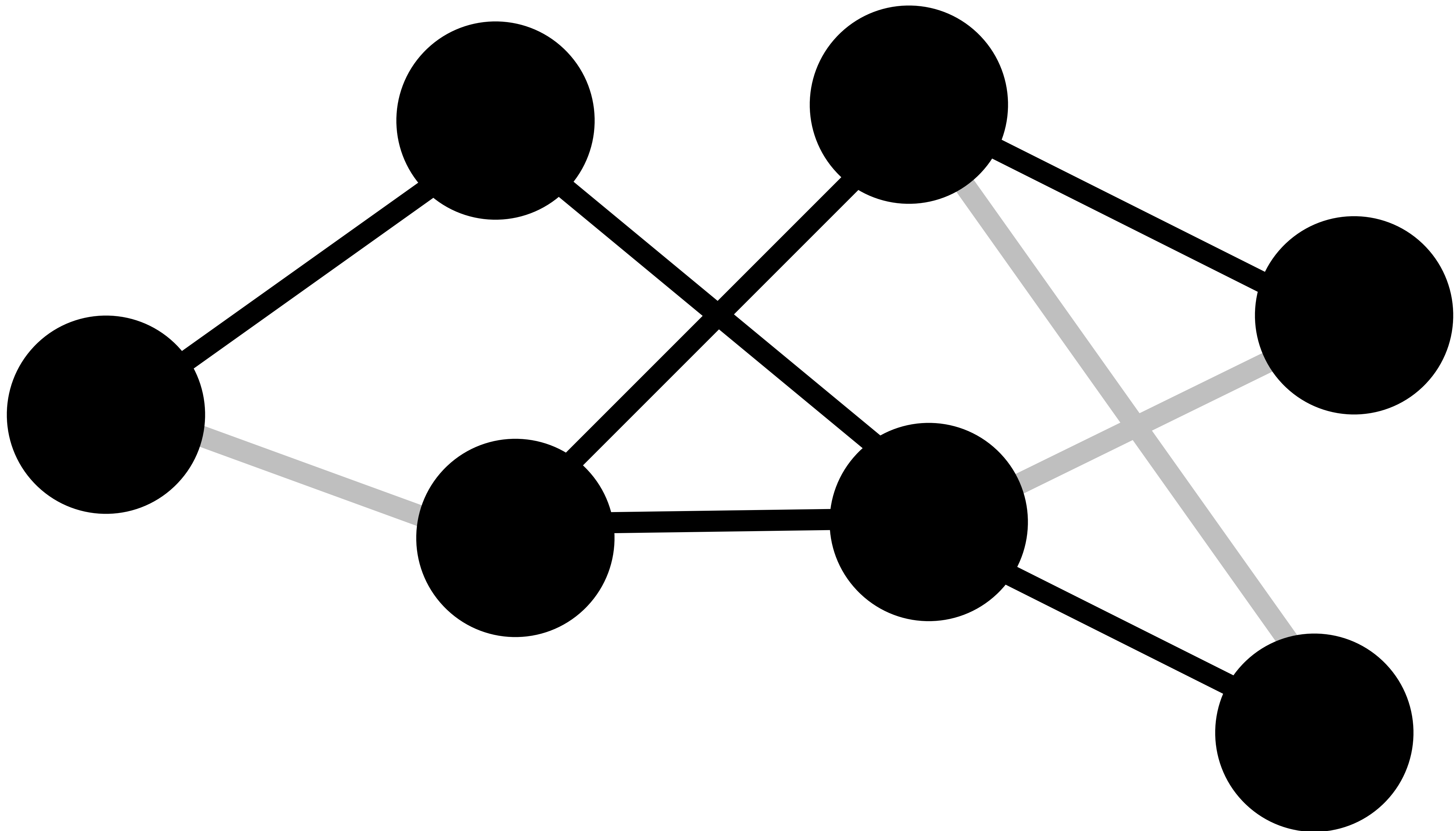
Key: Edges left out of the tree:

- in BFS, level difference is {-1, 0, 1}
  - not incident with the root.
- in DFS, join a node to its ancestor

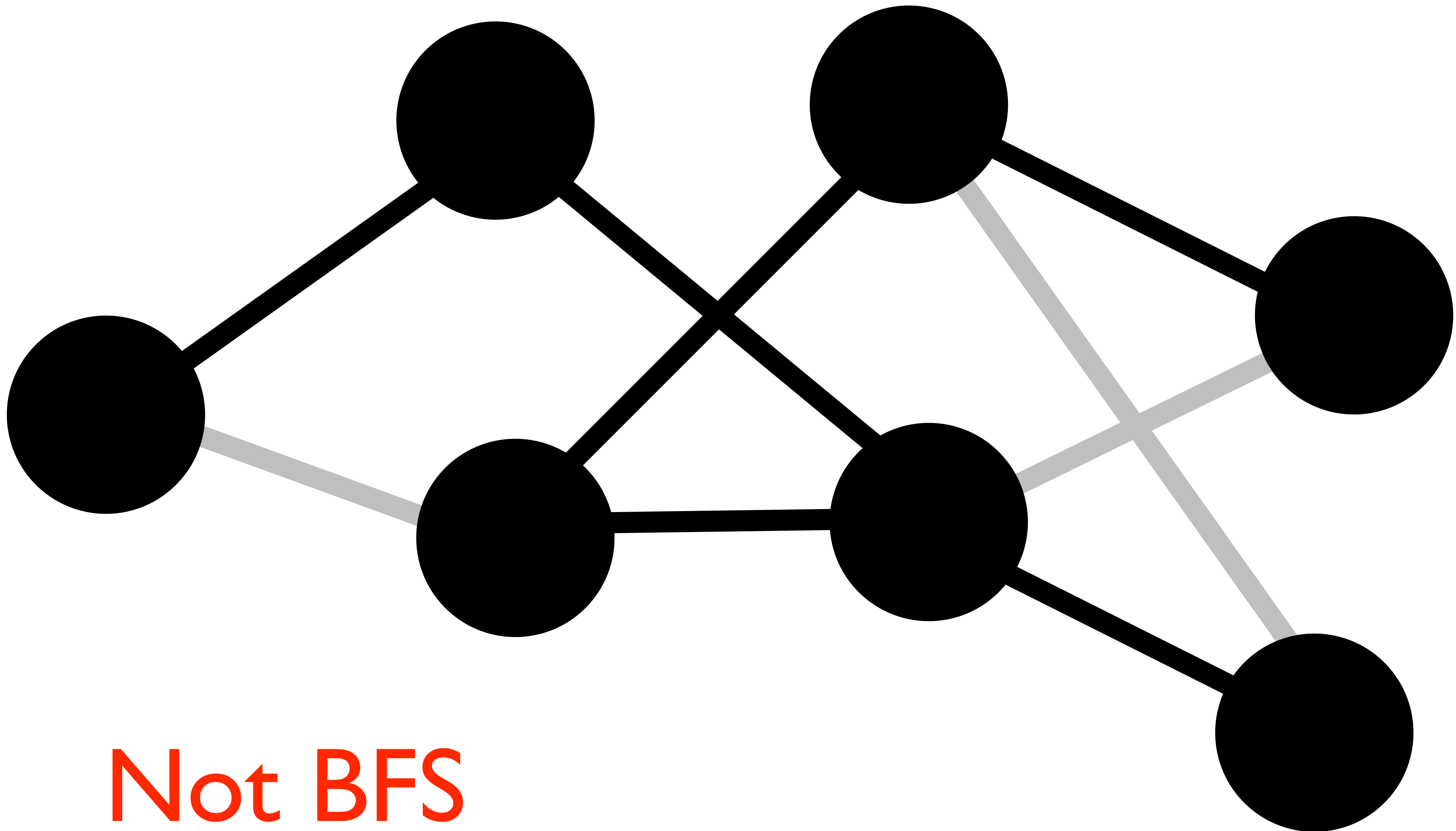
Welcome to the morgue!

Somebody constructed this tree.
Please help me identify it.

BFS or DFS tree?

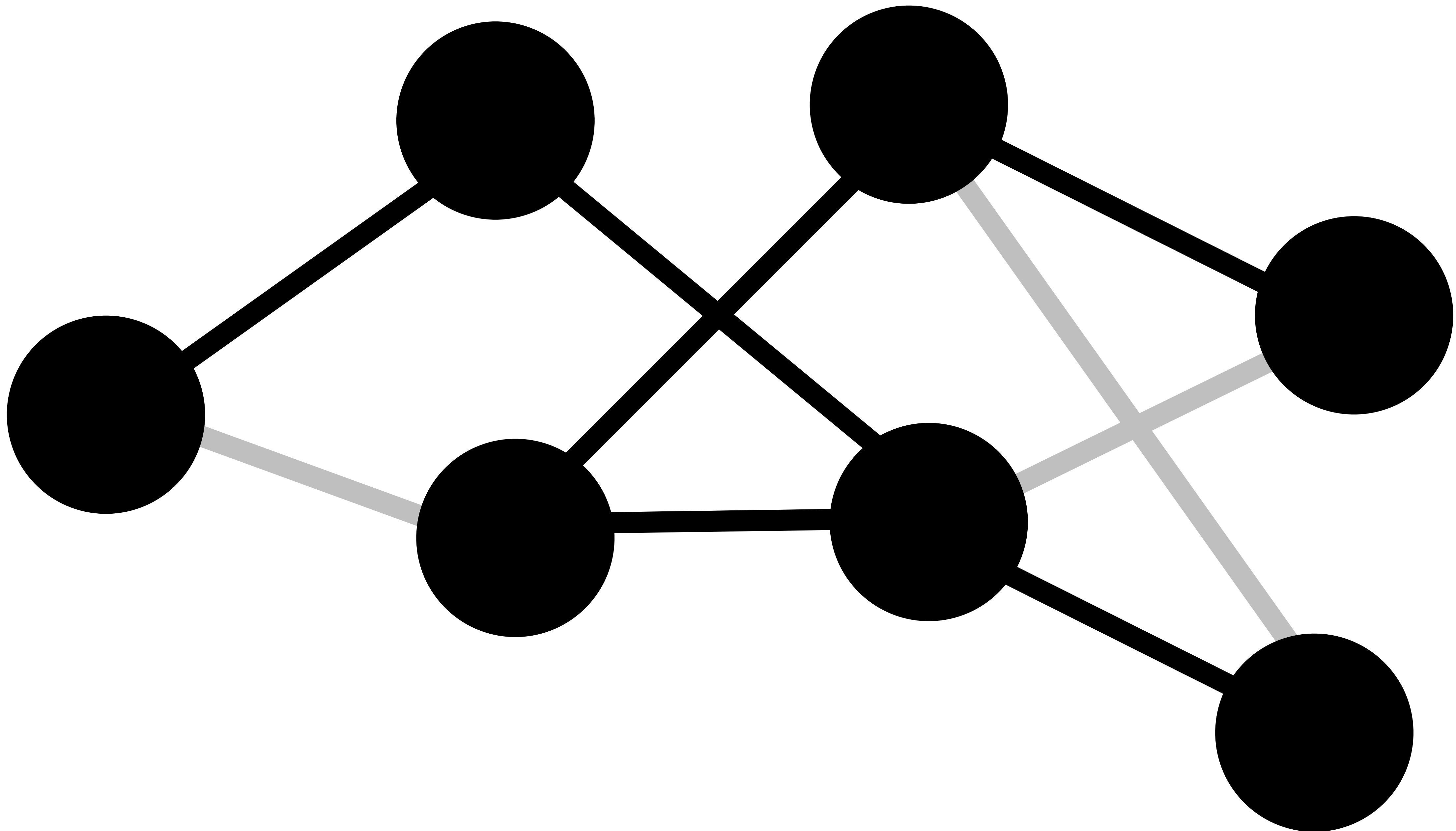


BFS or DFS tree?

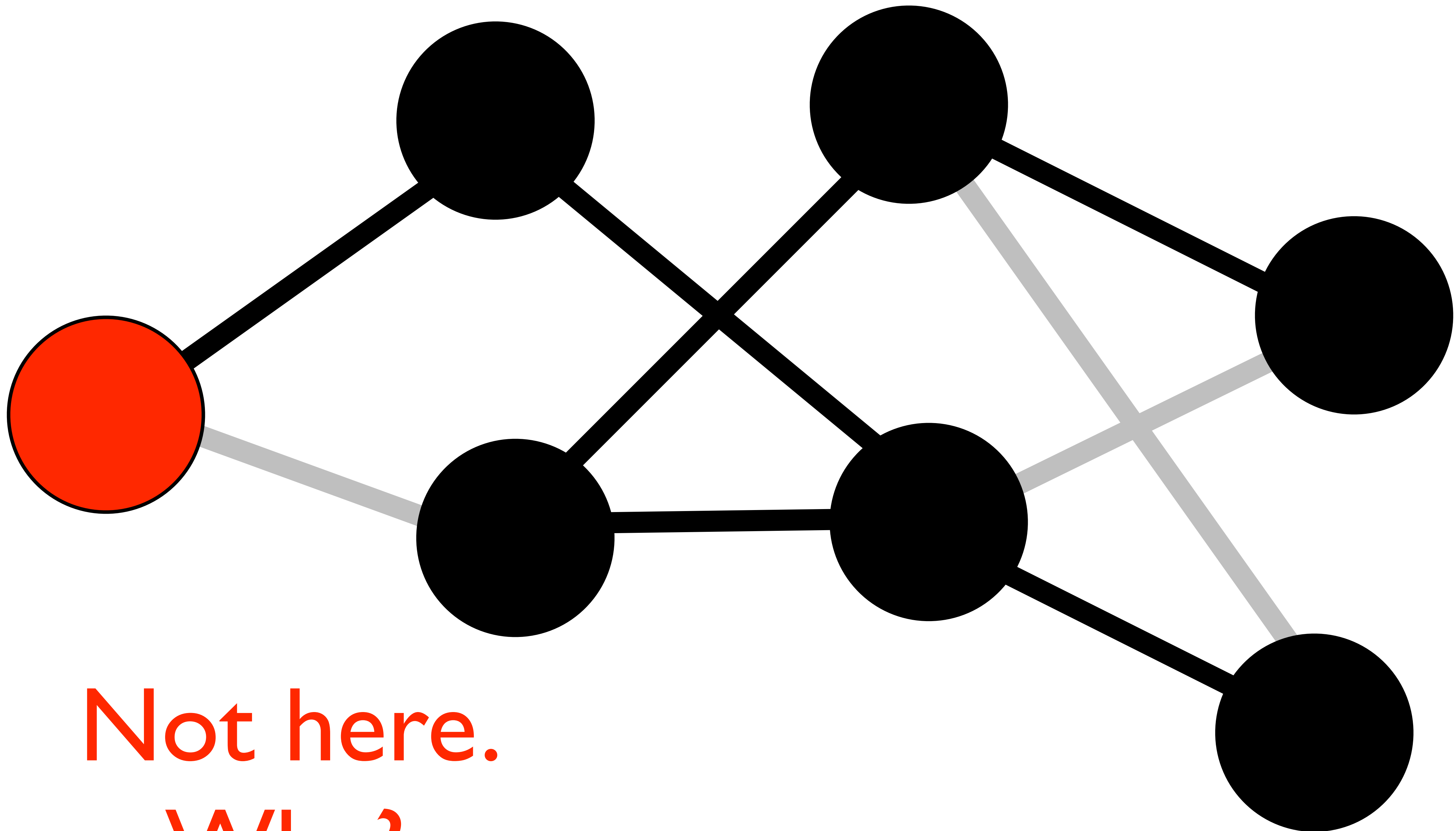


Not BFS

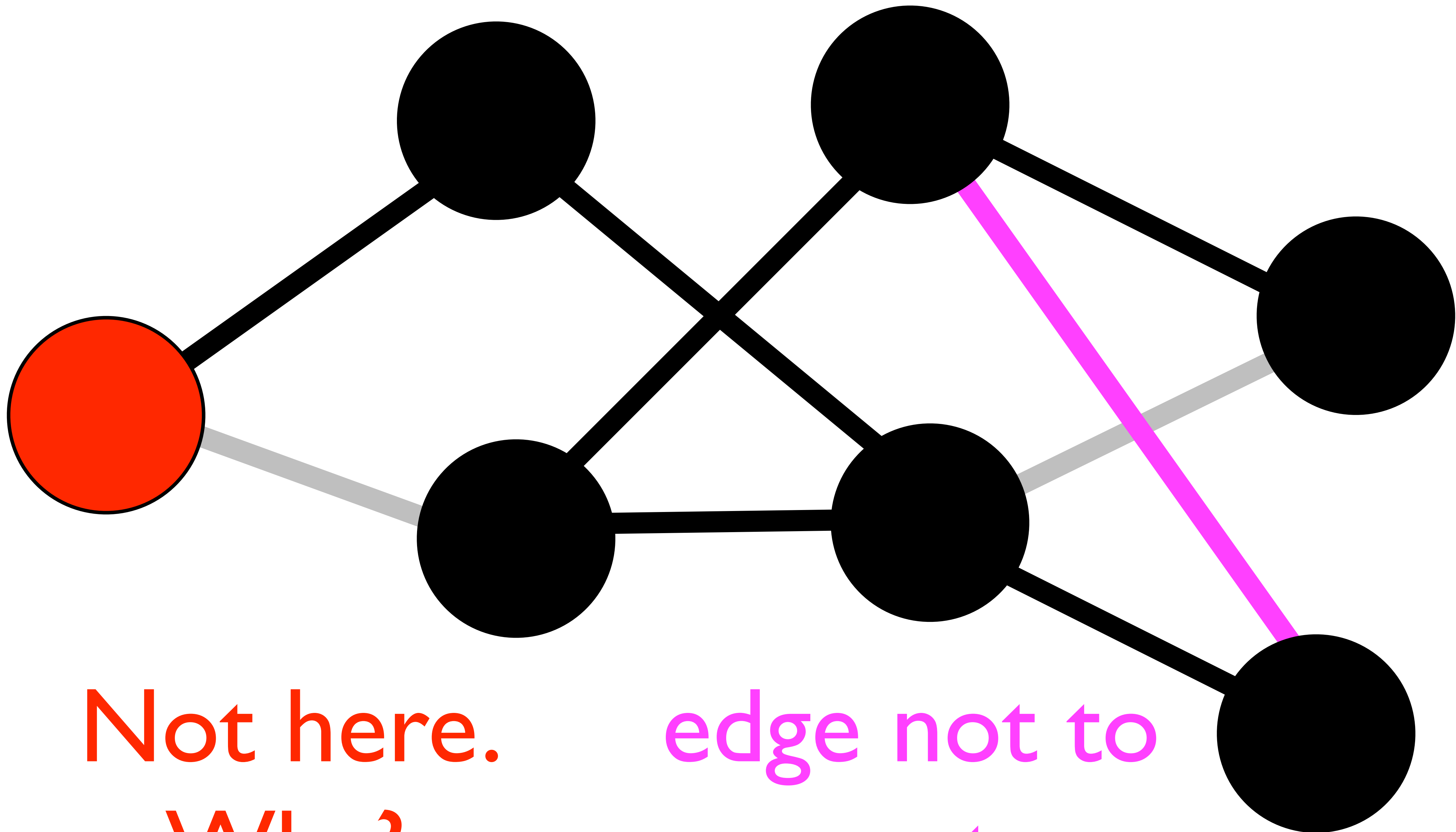
Where is the root (DFS)?



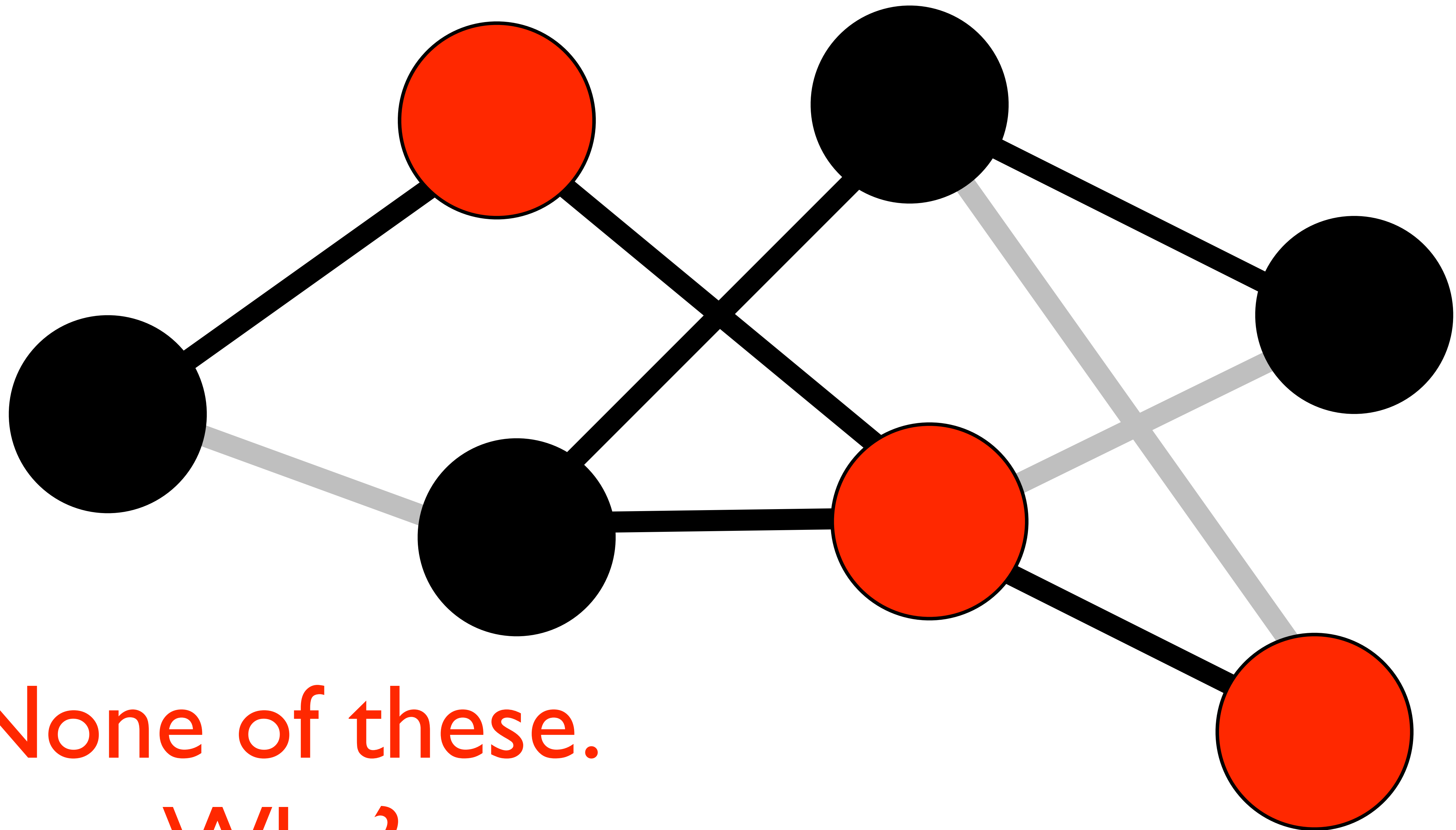
Where is the root (DFS)?



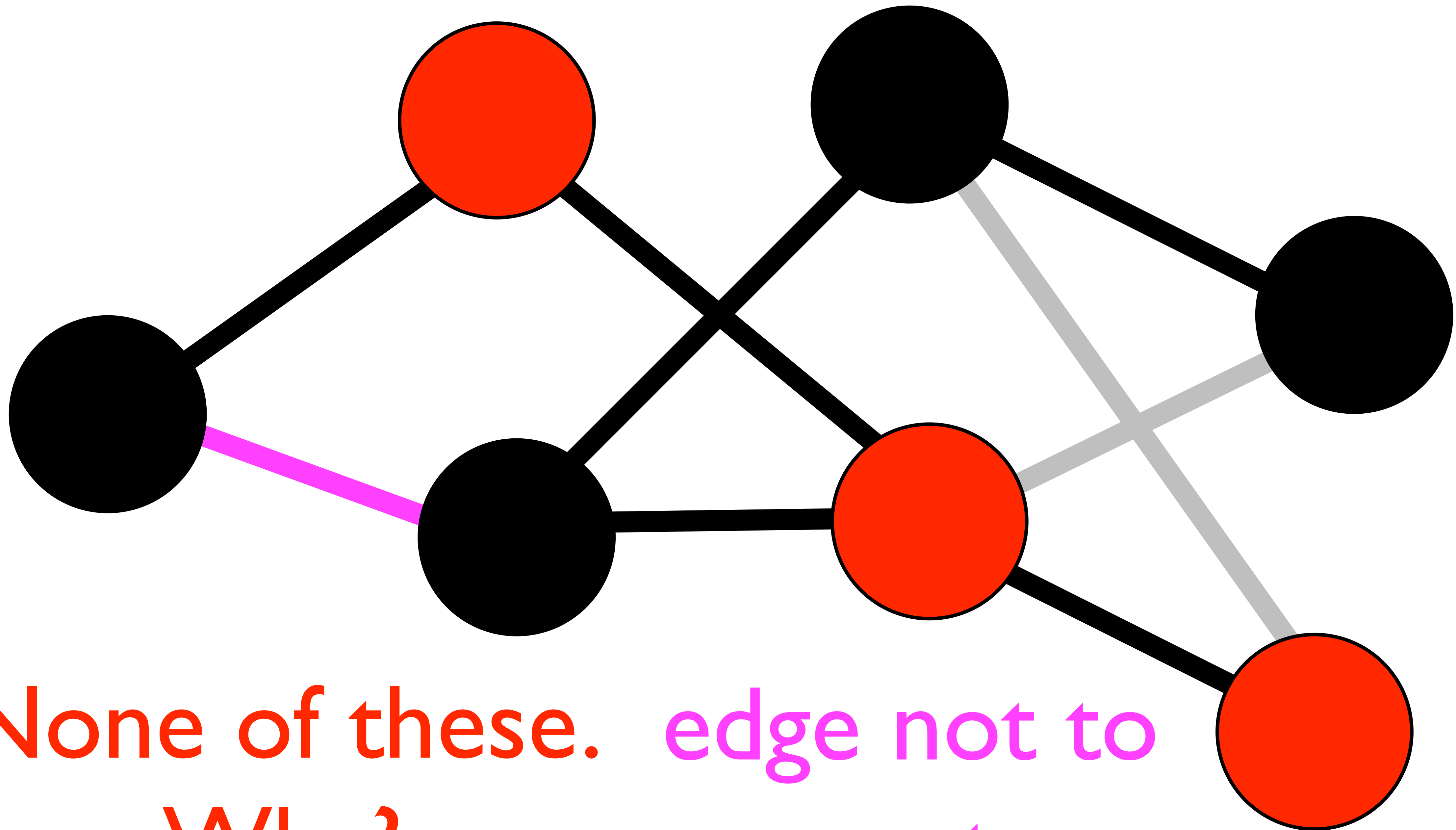
Where is the root (DFS)?



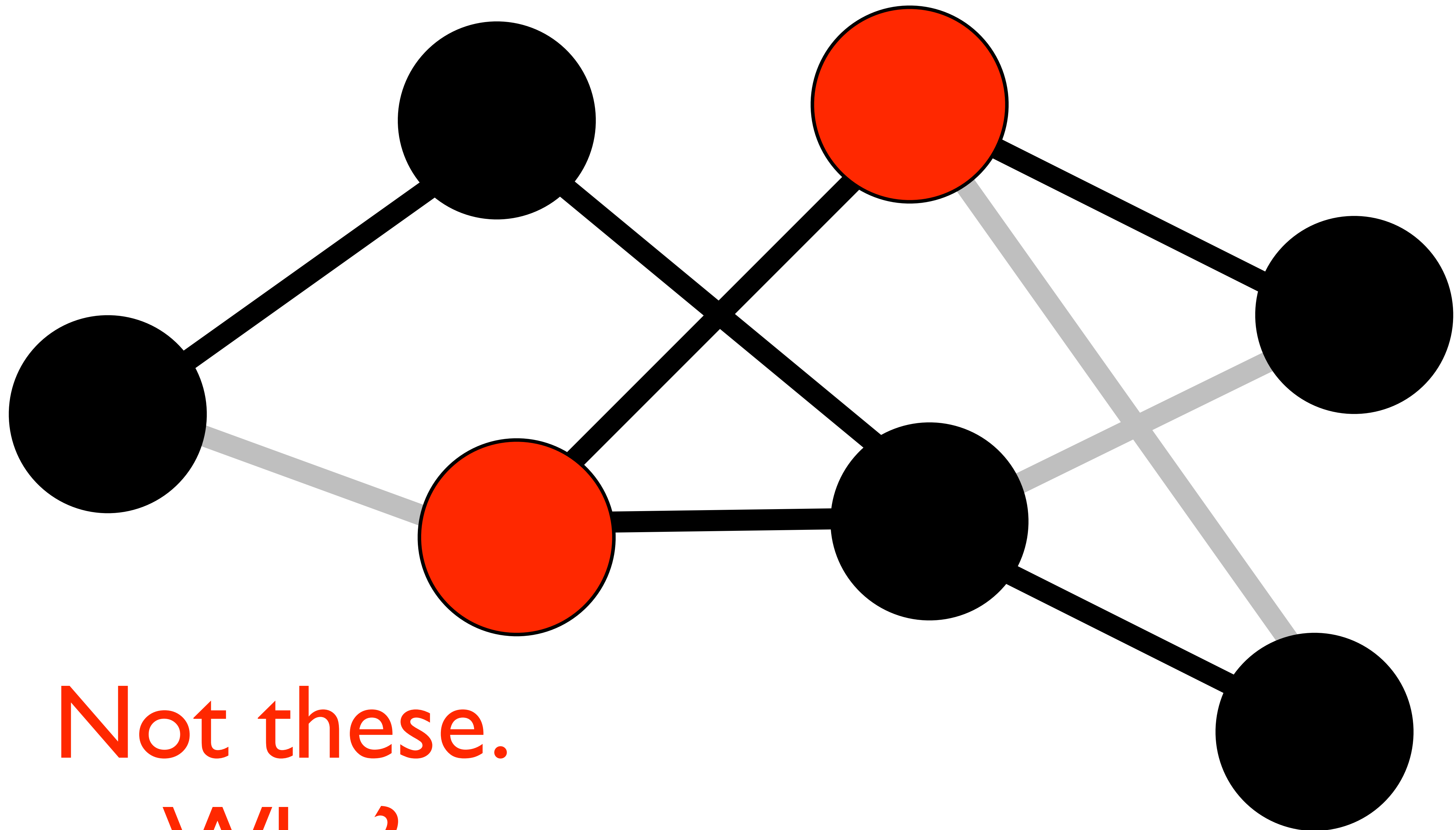
Where is the root (DFS)?



Where is the root (DFS)?



Where is the root (DFS)?

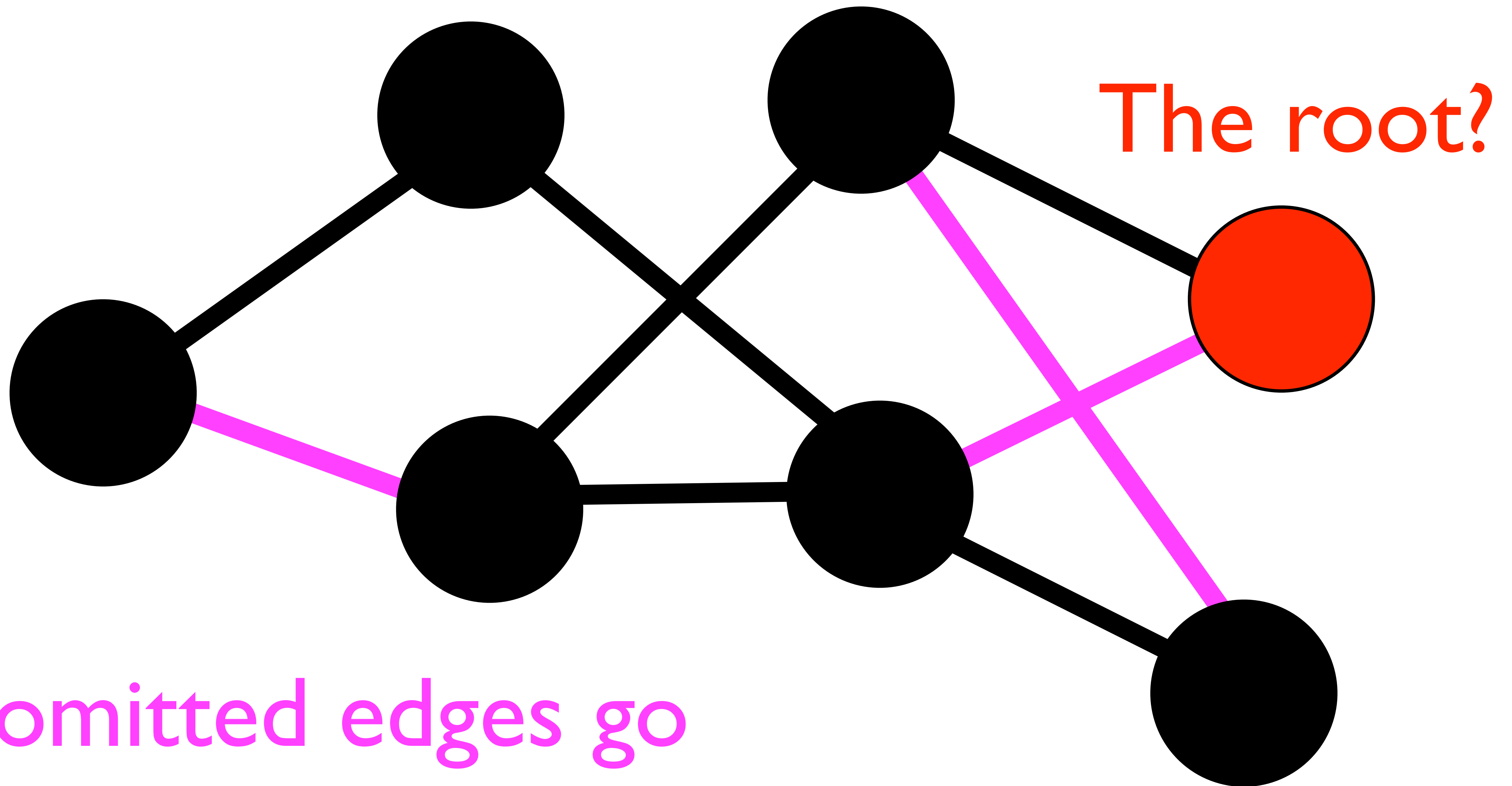


Where is the root (DFS)?

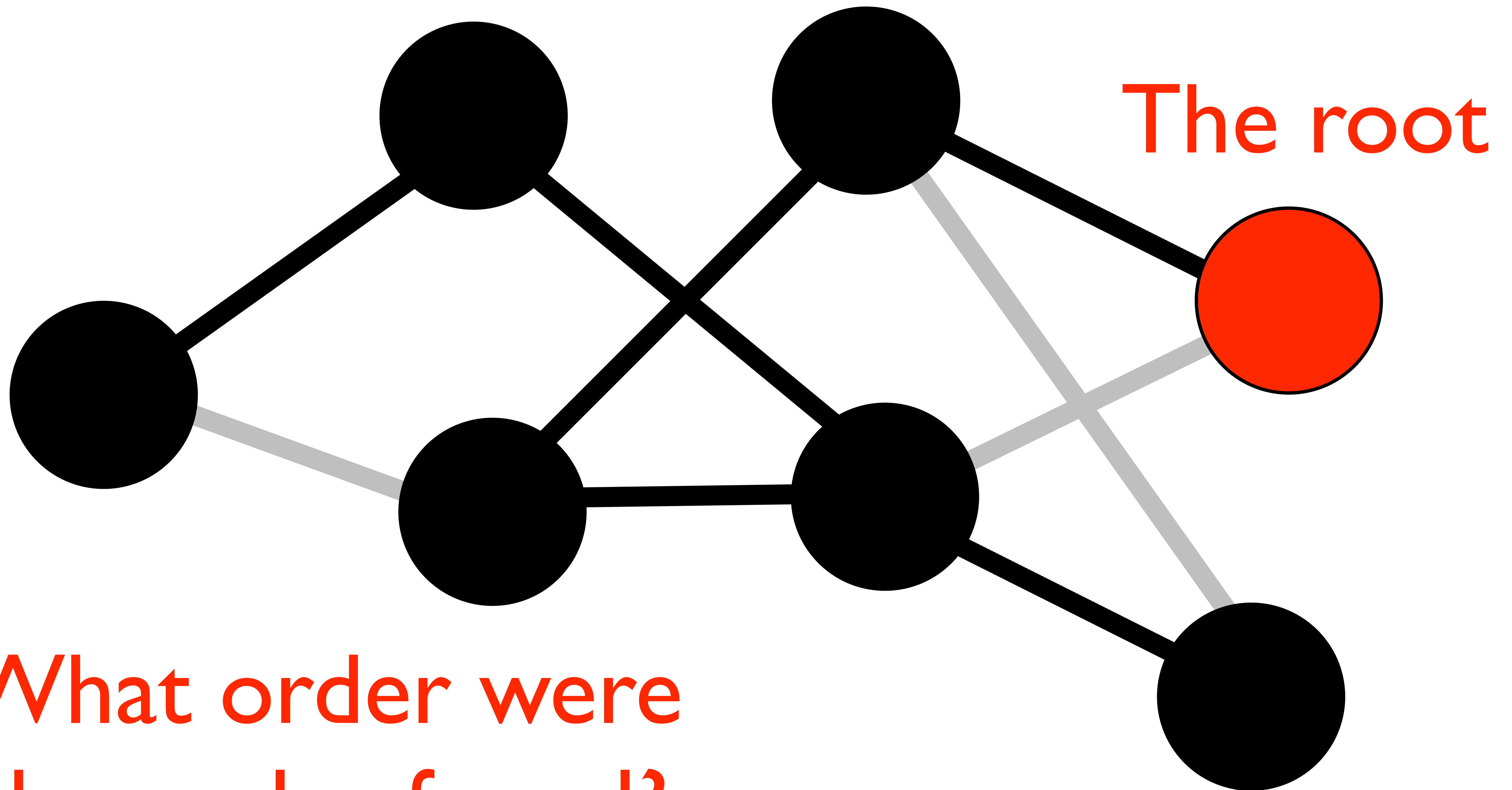
Not these.
Why?

edge not to
ancestor

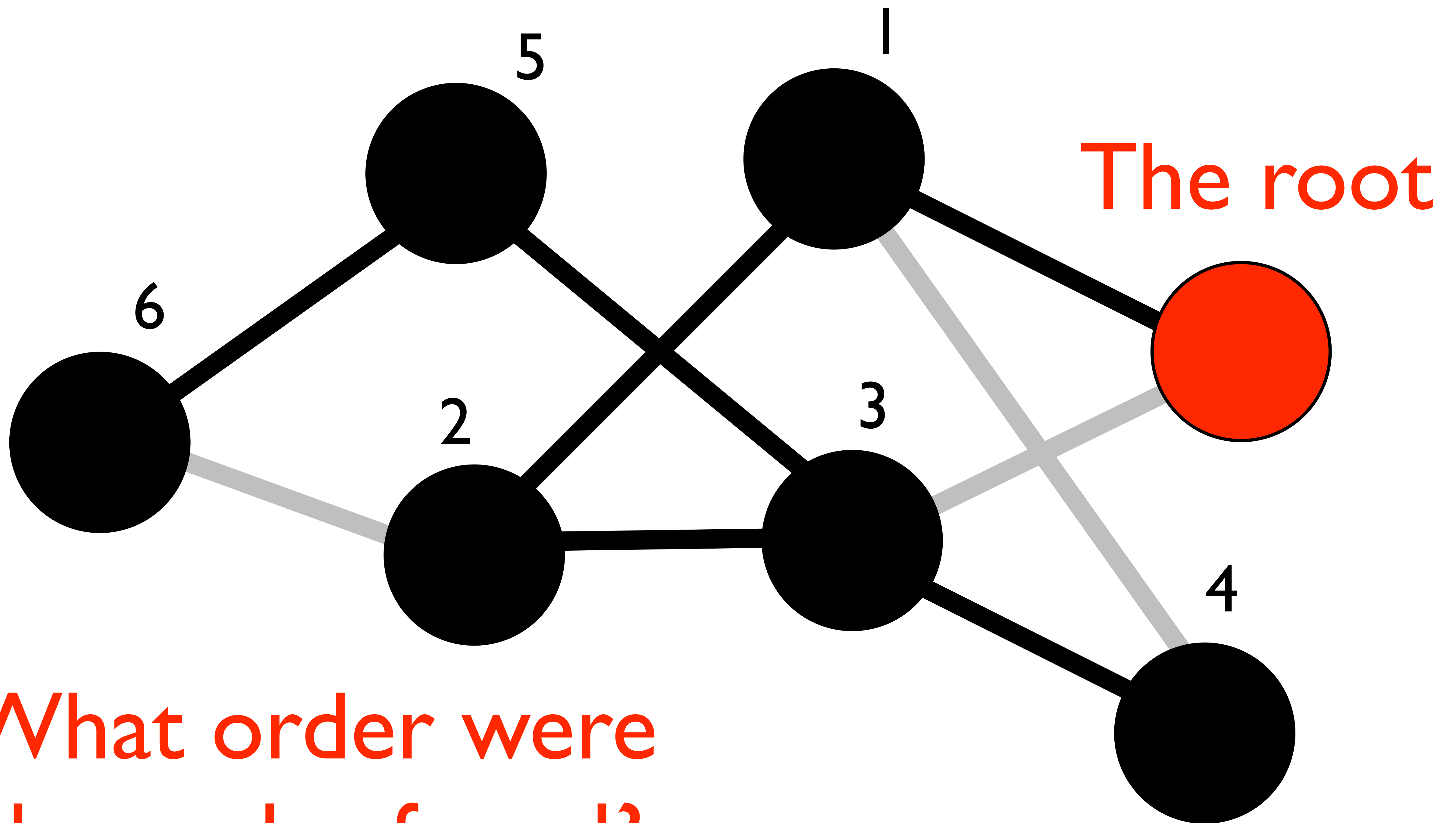
Where is the root (DFS)?



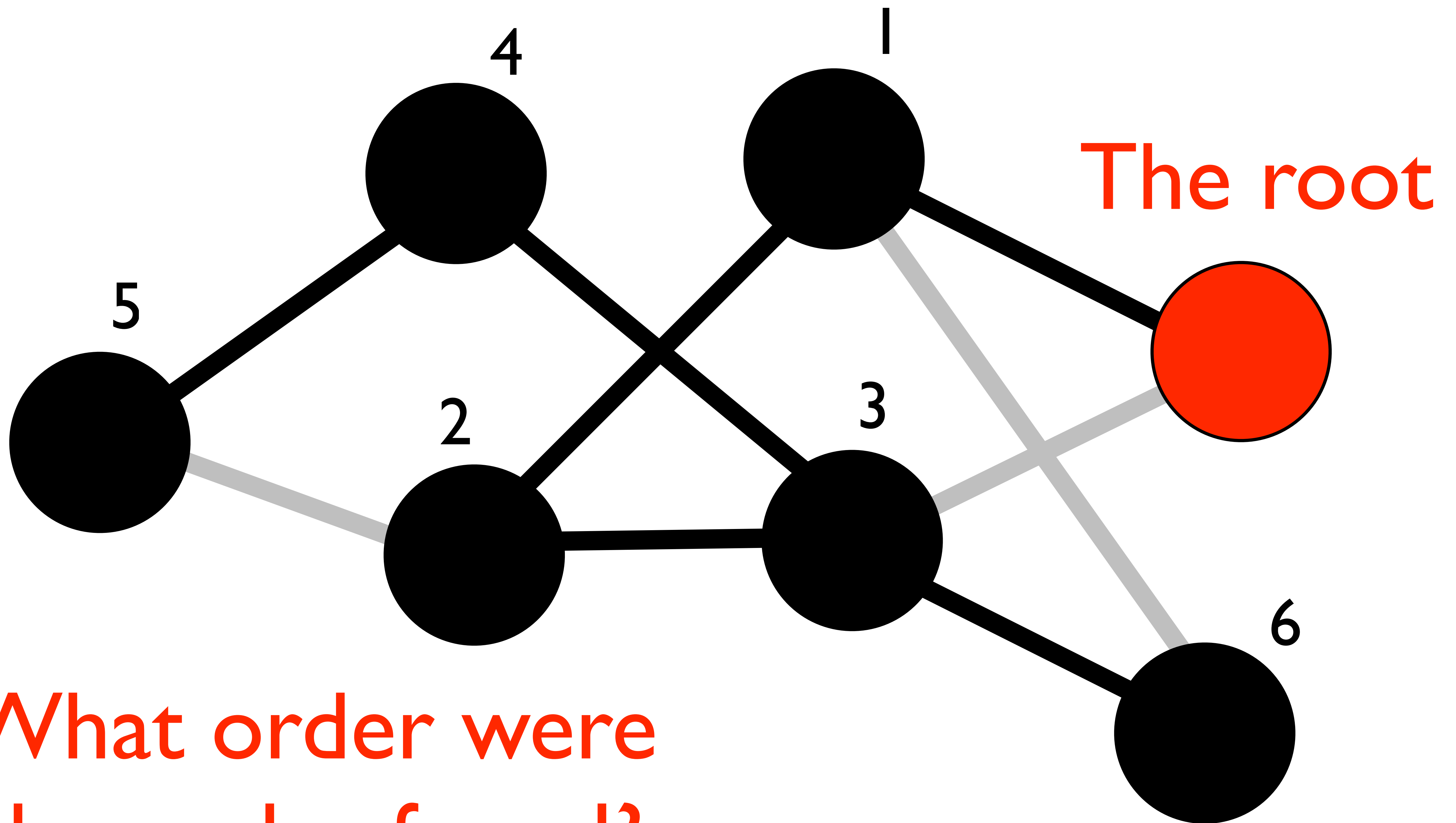
Where is the root (DFS)?



Where is the root (DFS)?



Where is the root (DFS)?



Problem 3.6

Given: Connected graph G and vertex u. DFS and BFS, both started from u, give us the same tree, T.

Prove: G = T.

Problem 3.6

Given: Connected graph G and vertex u. DFS and BFS, both started from u, give us the same tree, T.

Prove: G = T.

Proof by contradiction: Suppose $G \neq T$.

Problem 3.6

Given: Connected graph G and vertex u. DFS and BFS, both started from u, give us the same tree, T.

Prove: G = T.

Proof by contradiction: Suppose G≠T. Let e = {v,w} be an edge in G, not in T.

Problem 3.6

Given: Connected graph G and vertex u. DFS and BFS, both started from u, give us the same tree, T.

Prove: G = T.

Proof by contradiction: Suppose G≠T. Let e = {v,w} be an edge in G, not in T. Since T is a BFST, we know: ....

Since T is a DFST, we know: ....

Problem 3.6

Given: Connected graph G and vertex u. DFS and BFS, both started from u, give us the same tree, T.

Prove: G = T.

Proof by contradiction: Suppose G≠T. Let e = {v,w} be an edge in G, not in T. Since T is a BFST, we know: ....

Since T is a DFST, we know: ....

Homework: Finish this proof.

Problem 3.9

Given: Graph G with nodes s and t such that distance(s,t) = (n/2 +1). (n = #nodes)

Prove: We can disconnect this graph if we are allowed to delete one node.

Problem 3.9

Given: Graph G with nodes s and t such that distance(s,t) $\geq$ n/2. (n = #nodes)

Prove: We can disconnect this graph if we are allowed to delete one node.

Proof: Let T be a BFS tree for G, rooted at s. What can we say about the levels?

Problem 3.9

Given: Graph G with nodes s and t such that distance(s,t) $\geq n/2$. (n = #nodes)

Prove: We can disconnect this graph if we are allowed to delete one node.

Proof: Let T be a BFS tree for G, rooted at s. What can we say about the levels?

Level 0 = {s}.

Problem 3.9

Given: Graph G with nodes s and t such that distance(s,t) = n/2 + 1. (n = #nodes)

Prove: We can disconnect this graph if we are allowed to delete one node.

Proof: Let T be a BFS tree for G, rooted at s. What can we say about the levels?

Level 0 = {s}. Level(n/2 + 1) contains t.

Problem 3.9

Given: Graph G with nodes s and t such that distance(s,t) = n/2 + 1. (n = #nodes)

Prove: We can disconnect this graph if we are allowed to delete one node.

Proof: Let T be a BFS tree for G, rooted at s. What can we say about the levels?

Level 0 = {s}. Level(n/2 + 1) contains t.

No two levels overlap.

Problem 3.9

Given: Graph G with nodes s and t such that distance(s,t) = n/2 + 1. (n = #nodes)

Prove: We can disconnect this graph if we are allowed to delete one node.

Proof: Let T be a BFS tree for G, rooted at s. What can we say about the levels?

Level 0 = {s}. Level(n/2 + 1) contains t.

No two levels overlap. Some level i has size 1!

Problem 3.9

Given: Graph G with nodes s and t such that distance(s,t) = $n/2 + 1$. (n = #nodes)

Prove: We can disconnect this graph if we are allowed to delete one node.

Proof: Let T be a BFS tree for G, rooted at s. What can we say about the levels?

Level 0 = {s}. Level($n/2 + 1$) contains t.

No two levels overlap. Some level i has size 1! ($1 \le i \le n/2$) HW: Finish this proof.

Problem 3.2

Input: An undirected graph G.

Output: A cycle in G, if one exists, else throw new UnexpectedTreeException( )

Problem 3.2

Input: An undirected graph G.

Output: A cycle in G, if one exists, else throw new UnexpectedTreeException( )

BFS or DFS?

Problem 3.2

Input: An undirected graph G.

Output: A cycle in G, if one exists, else throw new UnexpectedTreeException( )

Let T be a DFS tree for G, rooted at u.
Now what?

Problem 3.2

Input: An undirected graph G.

Output: A cycle in G, if one exists, else throw new UnexpectedTreeException( )

Let T be a DFS tree for G, rooted at u. Now what? Suppose {v,w} is a back-edge for T.

Problem 3.2

Input: An undirected graph G.

Output: A cycle in G, if one exists, else throw new UnexpectedTreeException( )

Let T be a DFS tree for G, rooted at u. Now what? Suppose {v,w} is a back-edge for T. w or v is ancestor of the other

Problem 3.2

Input: An undirected graph G.

Output: A cycle in G, if one exists, else throw new UnexpectedTreeException( )

Let T be a DFS tree for G, rooted at u. Now what? Suppose {v,w} is a back-edge for T. Say w is ancestor of v.

Problem 3.2

Input: An undirected graph G.

Output: A cycle in G, if one exists, else throw new UnexpectedTreeException( )

Let T be a DFS tree for G, rooted at u. Now what? Suppose {v,w} is a back-edge for T. Say w is ancestor of v. Found cycle: v, p(v), p(p(v)), ..., w, v.

Problem 3.2

Input: An undirected graph G.

Output: A cycle in G, if one exists, else throw new UnexpectedTreeException( )

Let T be a DFS tree for G, rooted at u. Now what? Suppose {v,w} is a back-edge for T. Say w is ancestor of v. Found cycle: v, p(v), p(p(v)), ..., w, v.

What's left?

Problem 3.2

Input: An undirected graph G.

Output: A cycle in G, if one exists, else throw new UnexpectedTreeException( )

Let T be a DFS tree for G, rooted at u. Now what? Suppose {v,w} is a back-edge for T. Say w is ancestor of v. Found cycle: v, p(v), p(p(v)), ..., w, v.

What's left? 1) Why is this cycle?

Problem 3.2

Input: An undirected graph G.

Output: A cycle in G, if one exists, else throw new UnexpectedTreeException( )

Let T be a DFS tree for G, rooted at u. Now what? Suppose {v,w} is a back-edge for T. Say w is ancestor of v. Found cycle: v, p(v), p(p(v)), ..., w, v.

What's left? 1) Why is this cycle? 2) What if there is no back-edge?

Problem 3.2

Input: An undirected graph G.

Output: A cycle in G, if one exists, else throw new UnexpectedTreeException( )

Let T be a DFS tree for G, rooted at u. Now what? Suppose {v,w} is a back-edge for T. Say w is ancestor of v. Found cycle: v, p(v), p(p(v)), ..., w, v.

What's left? 1) Why is this cycle?
2) What if there is no back-edge?
3) What if G is not connected?

Problem 3.2

Input: An undirected graph G.

Output: A cycle in G, if one exists, else throw new UnexpectedTreeException( )

Let T be a DFS tree for G, rooted at u. Now what? Suppose {v,w} is a back-edge for T. Say w is ancestor of v. Found cycle: v, p(v), p(p(v)), ..., w, v.

HW: Finish proof

DFS: recursive version

```
Initialize: Mark all nodes Unfound
DFS(v):
  Mark v as Found
  For (w in Adj[v]):
    If (w is Unfound)
      {parent(w) = v;  DFS(w);}
```

DFS: iterative version

```
Initialize:
Active = new stack with just s.
Mark all nodes Unfound
while (Active is nonempty) {
  v = Active.pop( );
  if (v is Unfound) {
    Mark v Found;
    For (w in Adj[v]):
        Active.push(w);
    }
```

Compare: BFS

```
Initialize:
Active = new queue with just s.
Mark all nodes Unfound
while (Active is nonempty) {
  v = Active.remove( );
  if (v is Unfound) {
    Mark v Found;
    For (w in Adj[v]):
        Active.add(w);
    }
```

Both versions of DFS are implemented using a Stack!

Both versions of DFS are implemented using a Stack!

(Recursive version uses the function-call stack)

Next Topic: Directed Graphs (aka "digraphs")

Directed Graphs

A directed graph is like a graph, but the edges are ORDERED pairs of vertices, rather than unordered pairs of vertices.

For normal graphs, we denote edges in curly braces: {u,v}   (unordered)

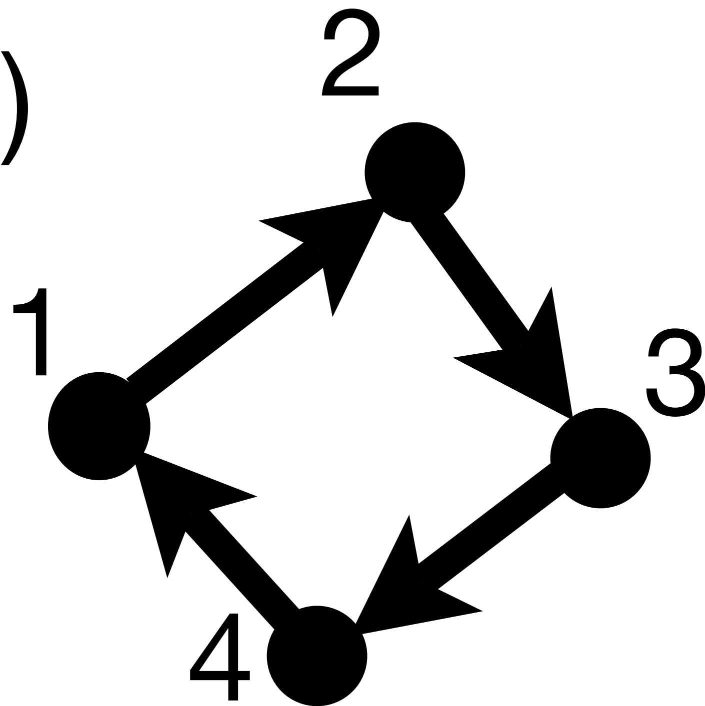
For directed graphs, parentheses: (u,v) (ordered: “from u to v”).
Directed edges are also called “arcs”

Directed Graphs

Example: $V = \{1,2,3,4\}$

$E = \{ (1,2), (2,3), (3,4), (4,1)\}$.

This example is known as a "directed cycle" or "oriented cycle" (of length 4)



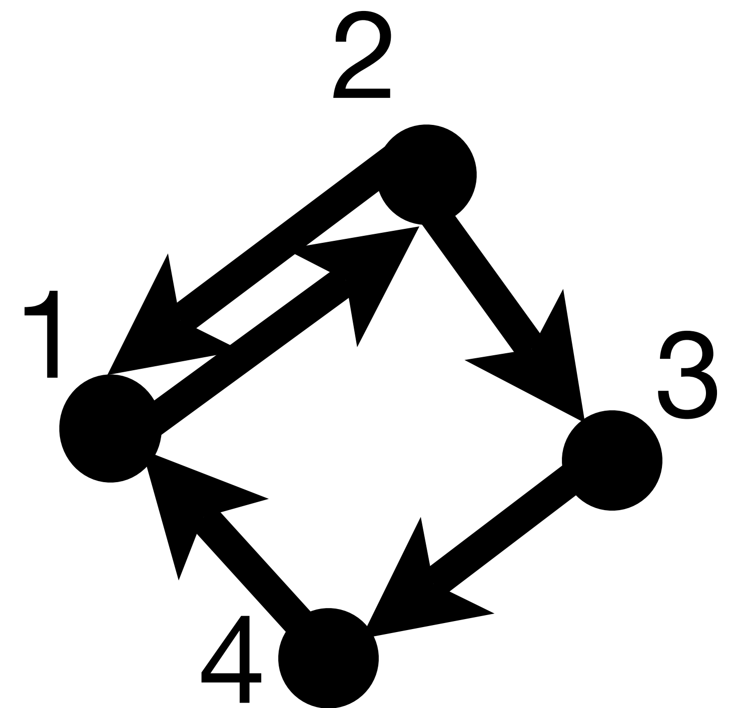
Representing DiGraphs

For directed graphs, we need 2 adjacency lists for each node. edges INTO v, and OUT FROM v.

2 arrays of Lists:

in-neighbors[2] = {1}

out-neighbors[2] = {1,3}

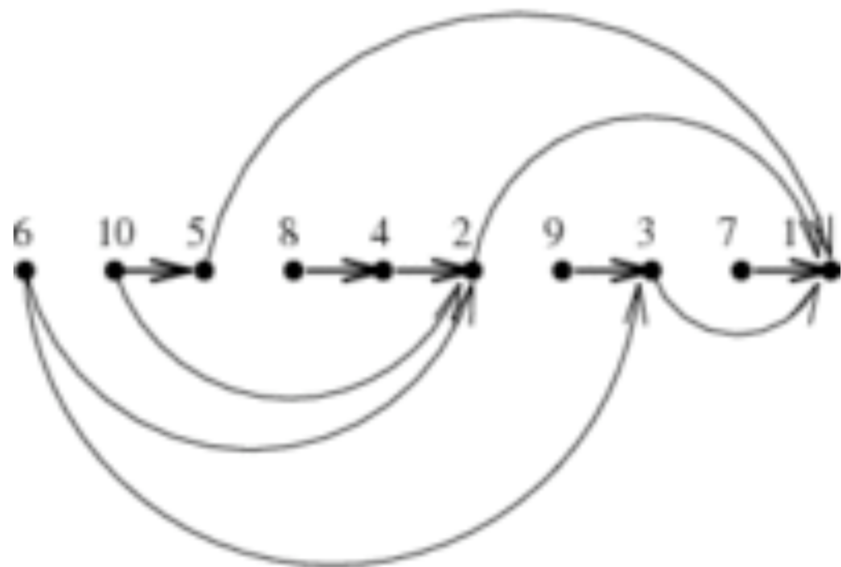


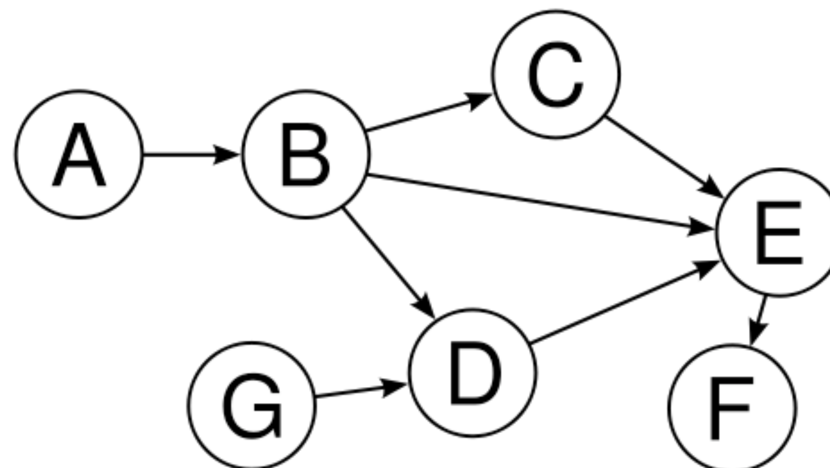
DAGs

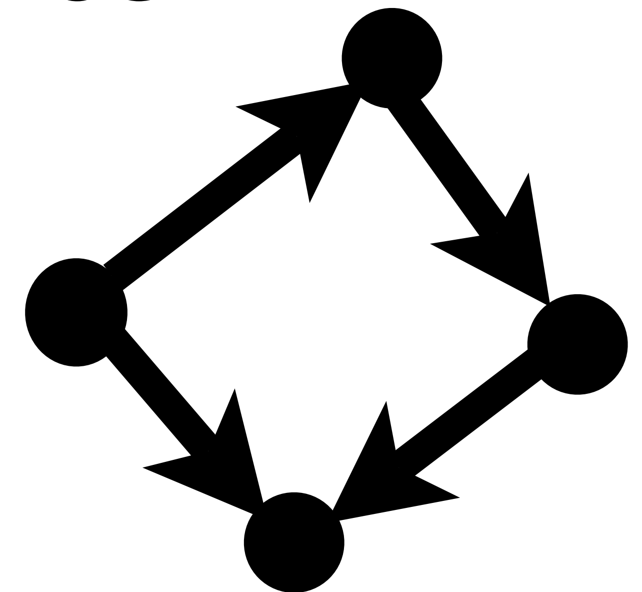
A directed graph is called acyclic if it has no oriented cycles.

Meaning: you can’t get back where you start if you always follow arrows.

The “underlying graph” (just lose all the orientation info) may have cycles.



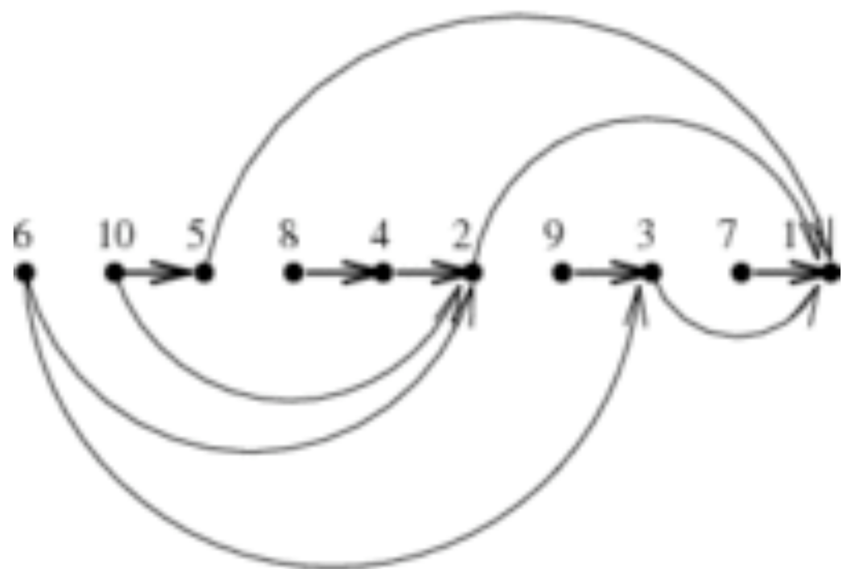


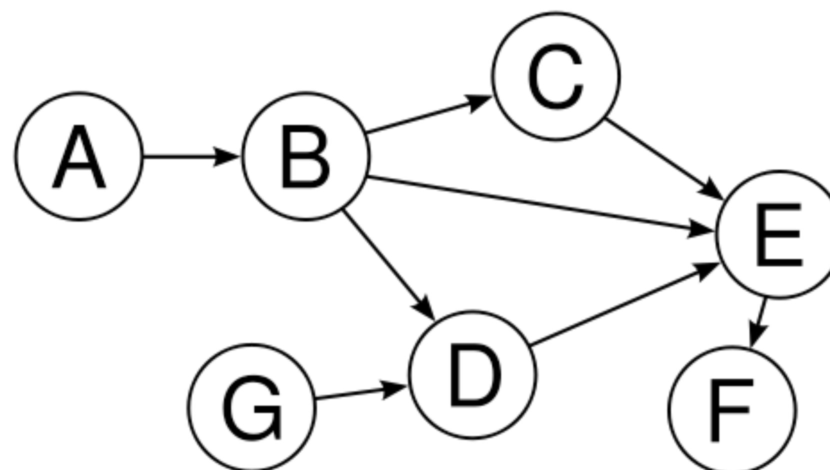


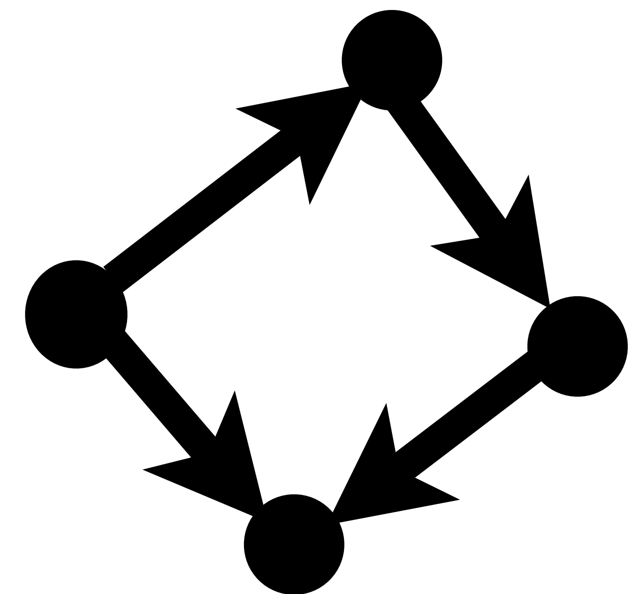
DAGs

Q: How do we tell if a graph is a DAG? Alternatively, how do we find an oriented cycle if there is one?

Look at the Left example. The nodes are in a line, all the edges go left-to-right. This is called a topological sort.





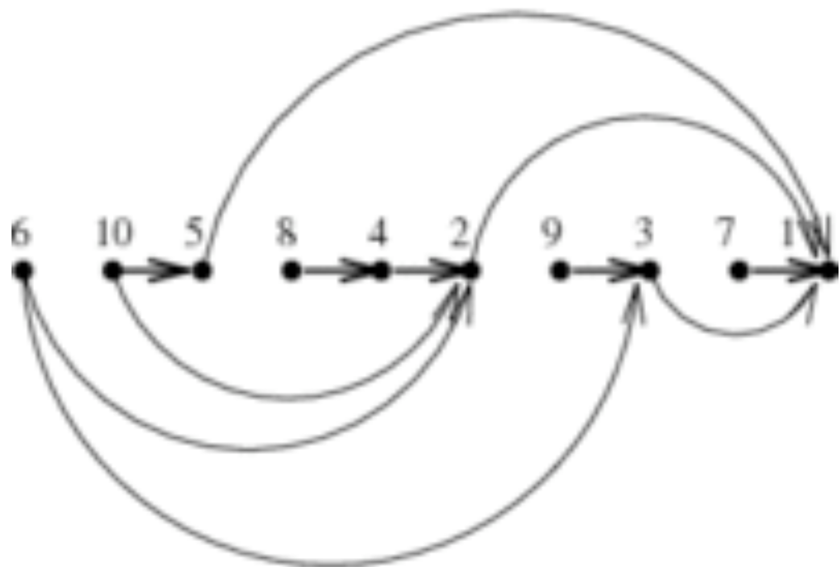


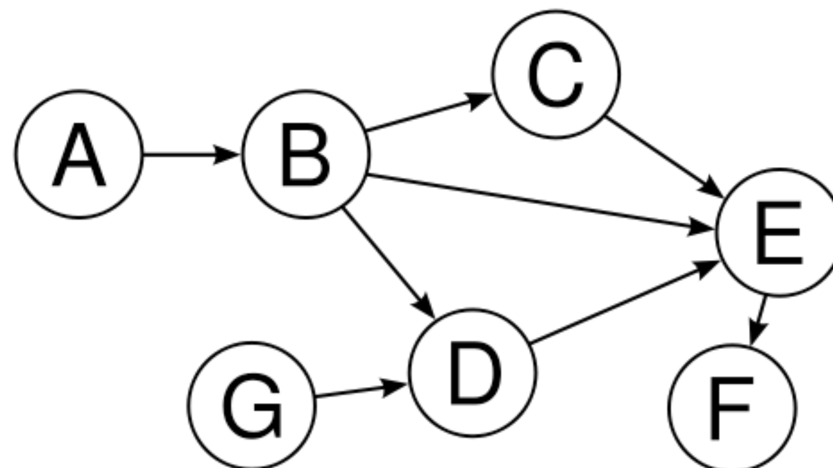
DAGs

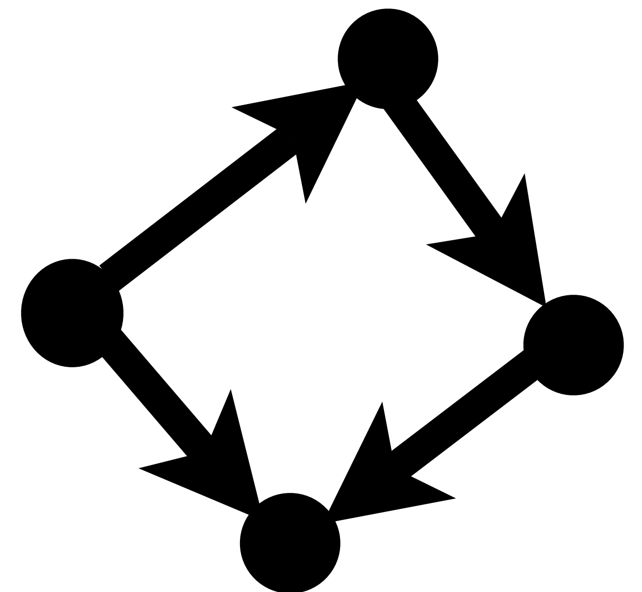
Q: How do we tell if a graph is a DAG? Alternatively, how do we find an oriented cycle if there is one?

Look at the Left example. The nodes are in a line, all the edges go left-to-right. This is called a topological sort.

Thm: G has a topological sort iff G is a DAG



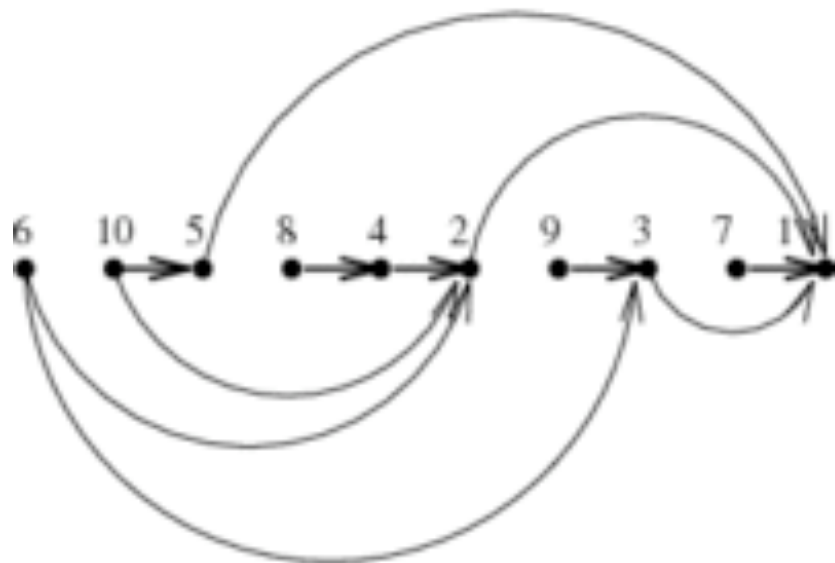


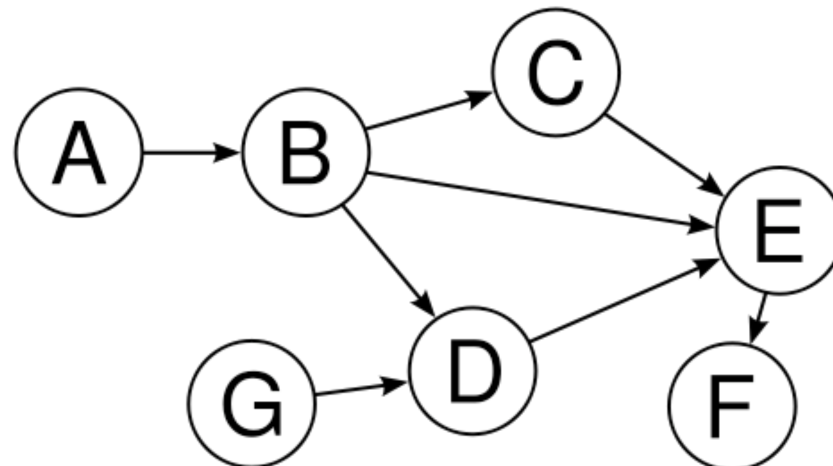


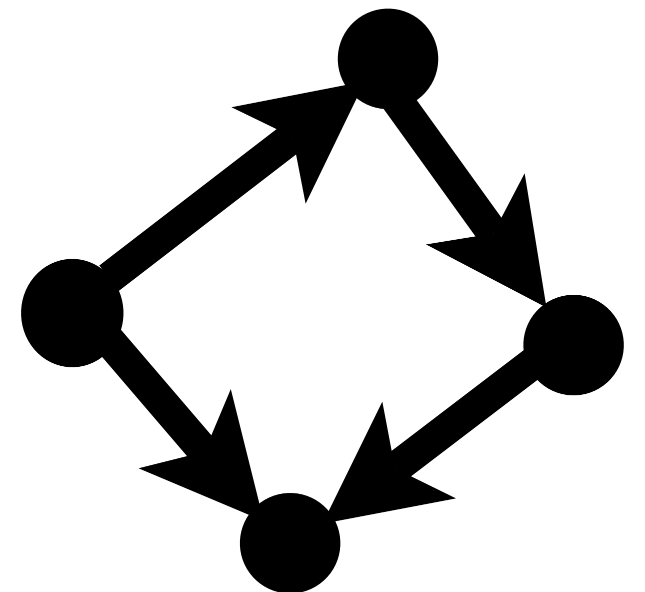
Topological Sorting

Q: How do we tell if a graph has a topo. sort?

Look at the left example. The leftmost node has no in-edges. Is this always the case?





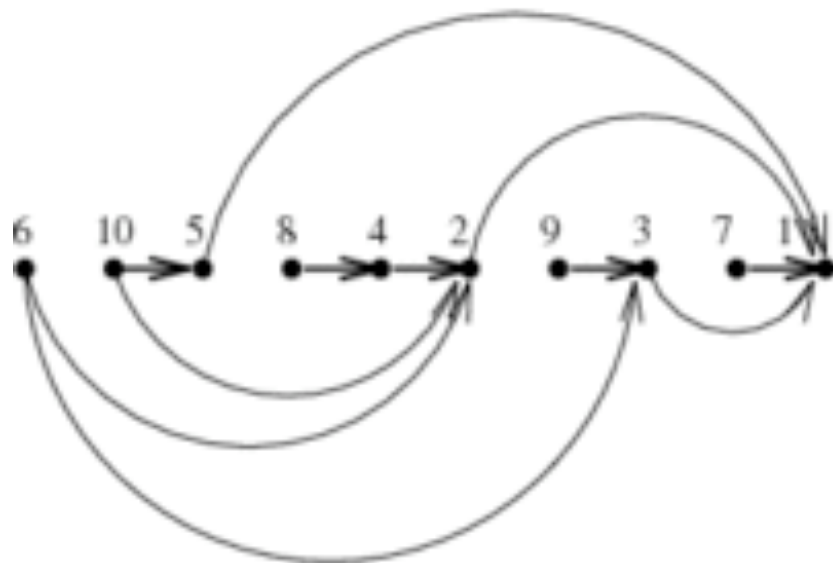


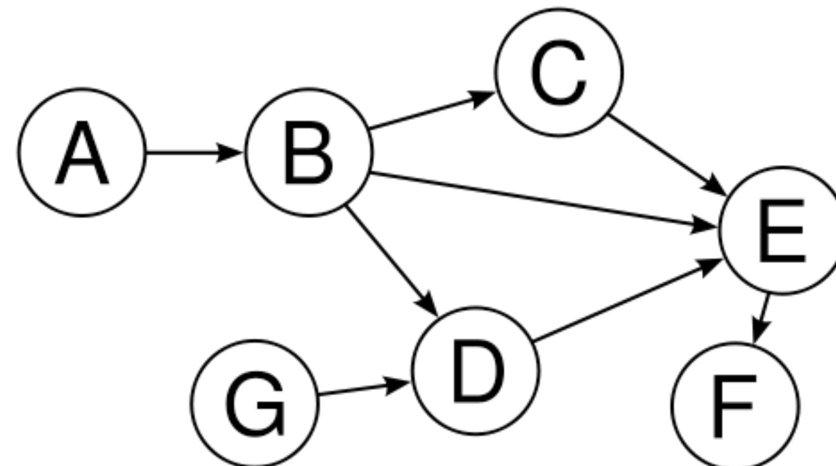
Topological Sorting

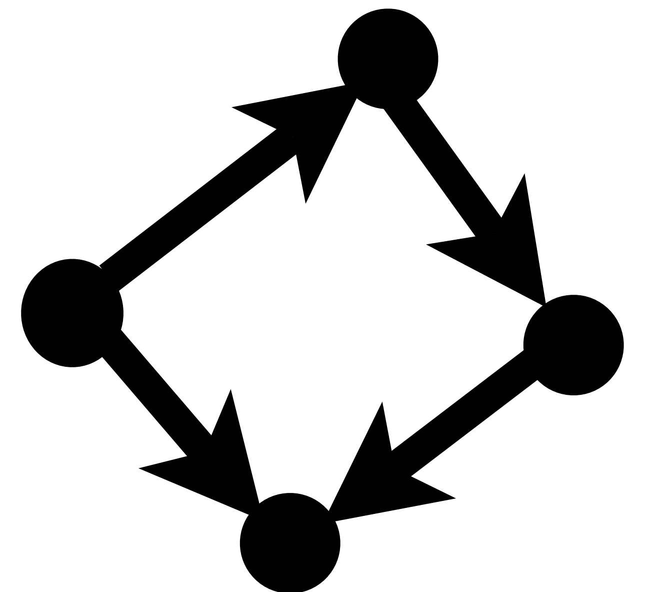
Q: How do we tell if a graph has a topo. sort?

Look at the left example. The leftmost node has no in-edges. Is this always the case? Yes.

Idea: Find the leftmost node. Recurse!







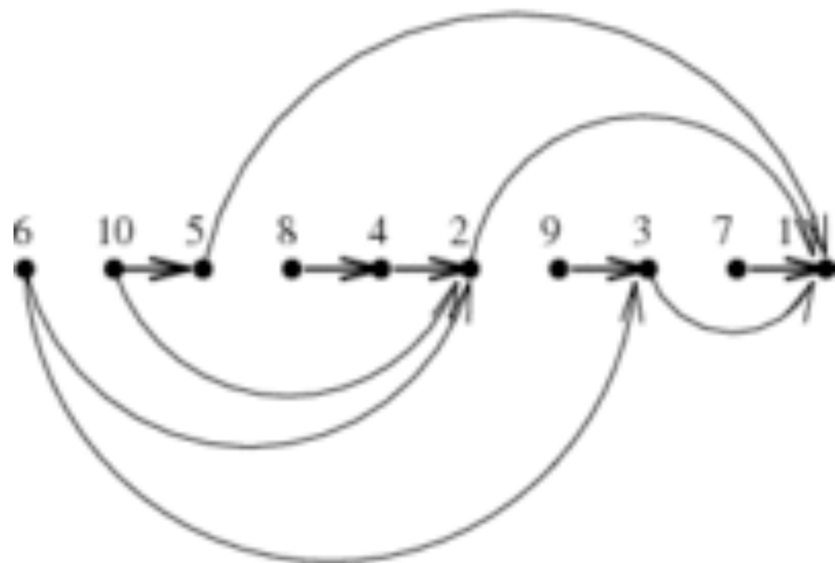
Topological Sorting

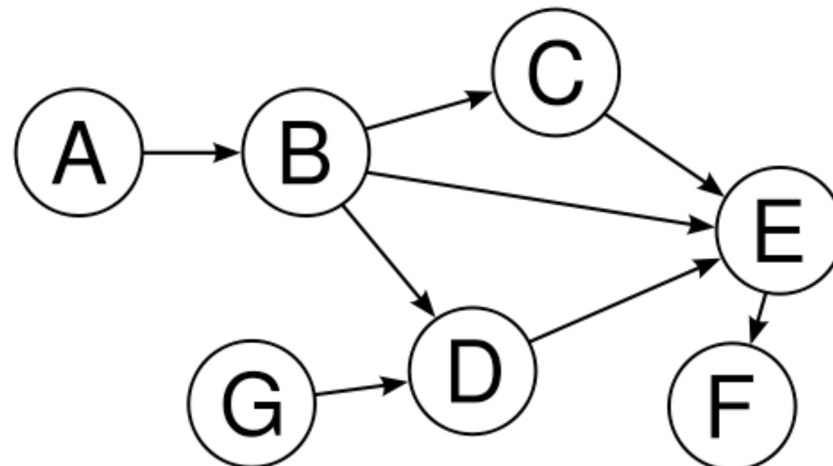
Q: How do we tell if a graph has a topo. sort?

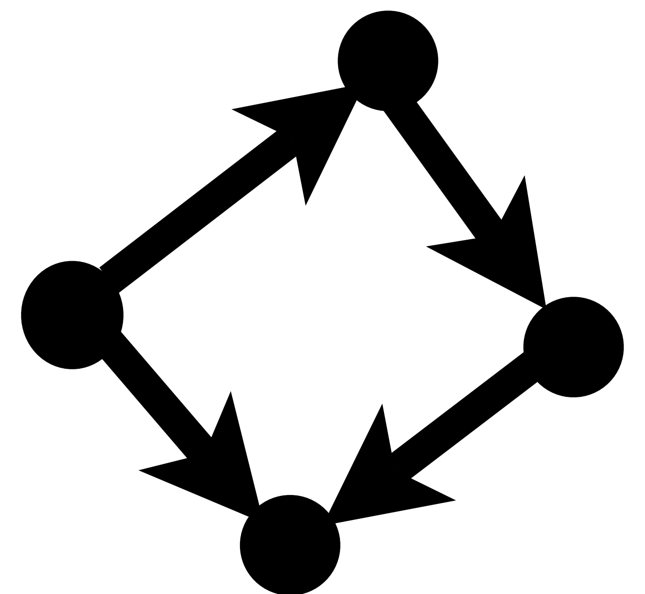
Look at the left example. The leftmost node has no in-edges. Is this always the case? Yes.

Idea: Find the leftmost node. Recurse!

Implementation issues?

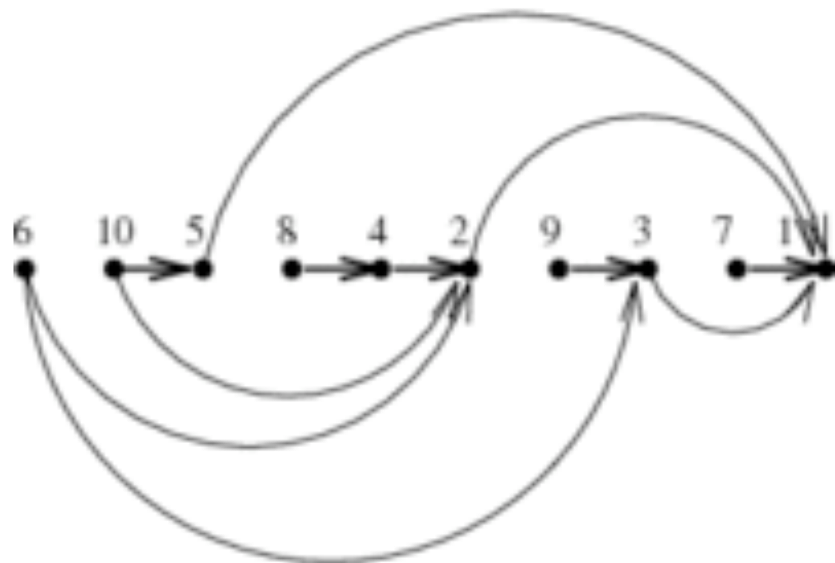


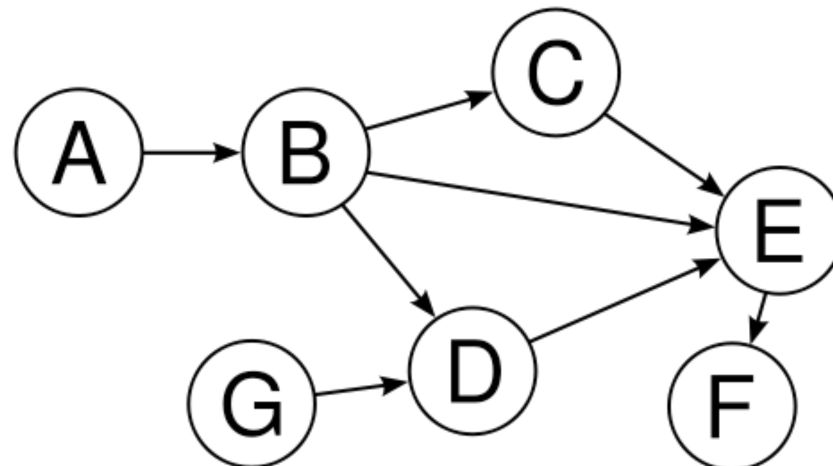


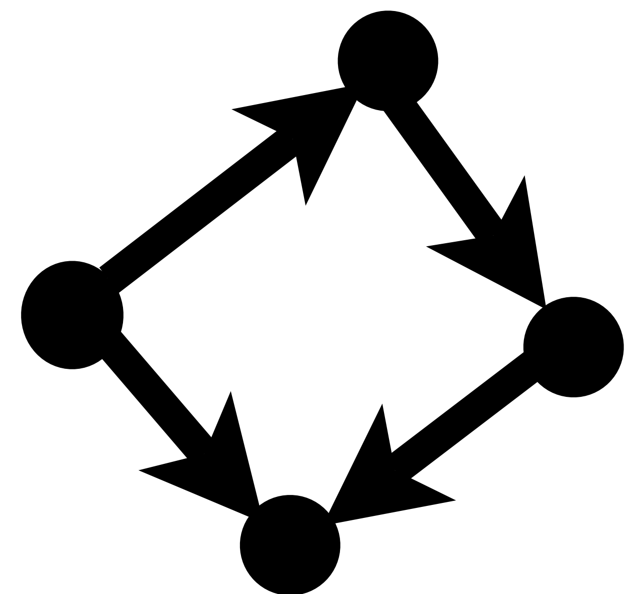


Searching DiGraphs

Question: Find all nodes reachable from s in a directed graph G.



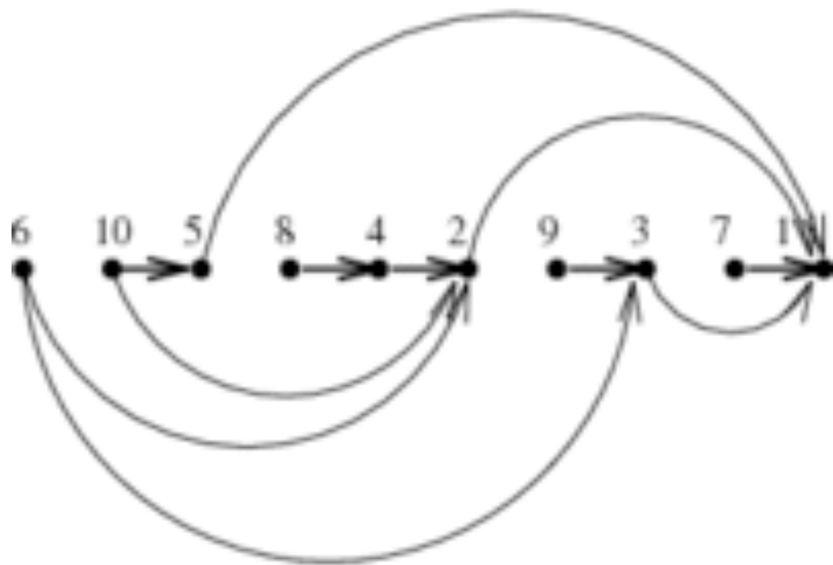


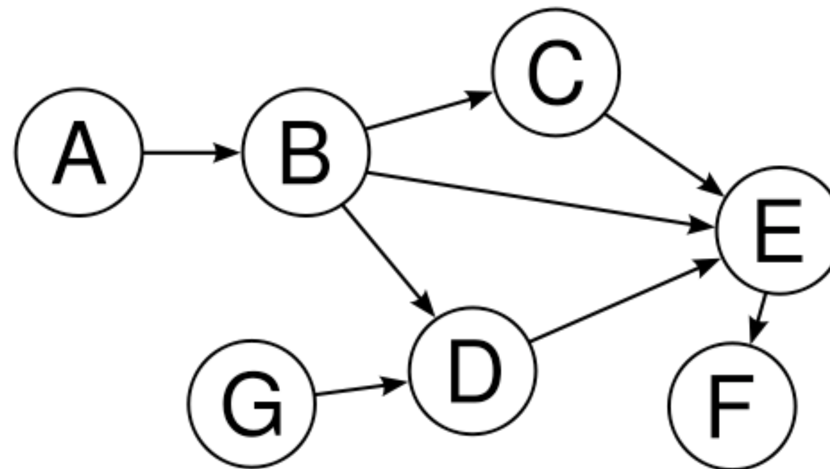


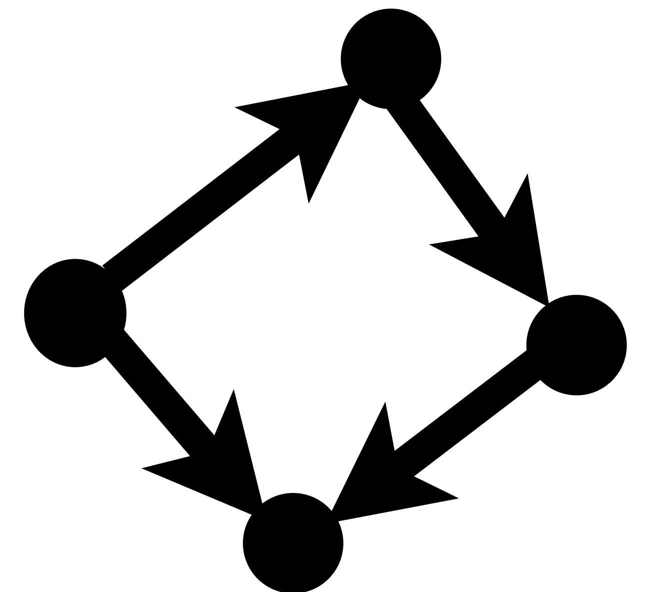
Searching DiGraphs

Question: Find all nodes reachable from s in a directed graph G.

Idea: Slightly modify BFS and DFS algorithms: they should only "find" nodes along out-edges.

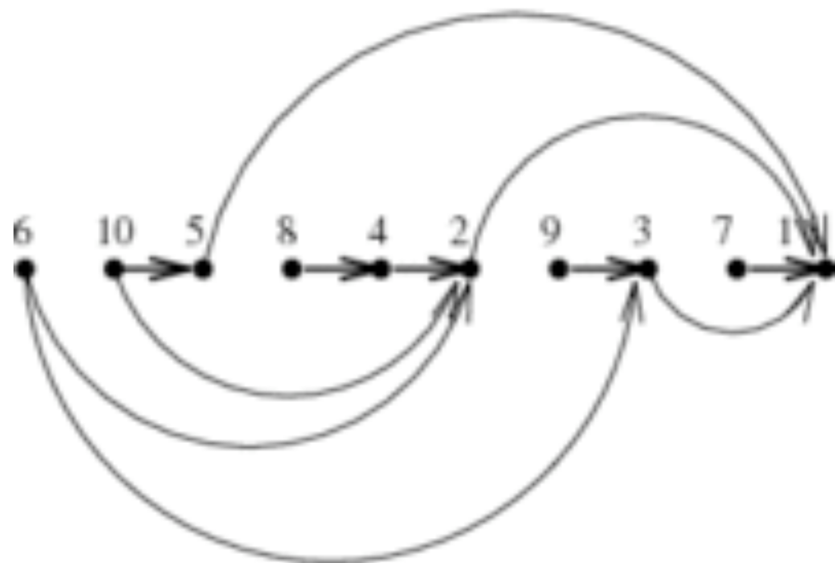


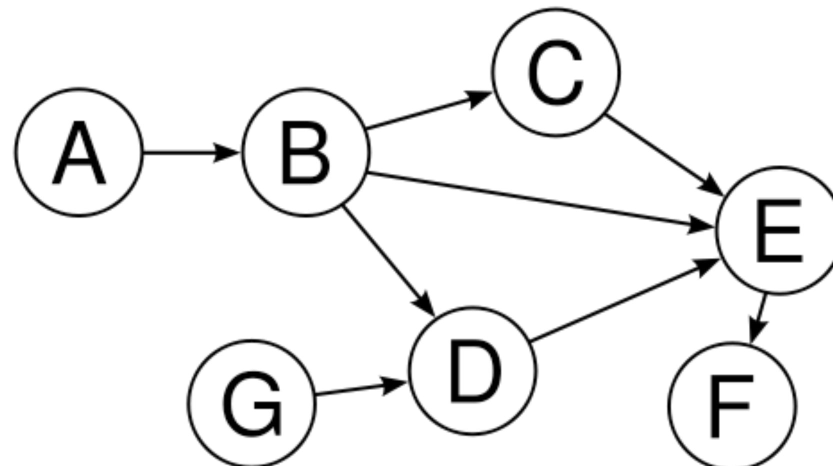


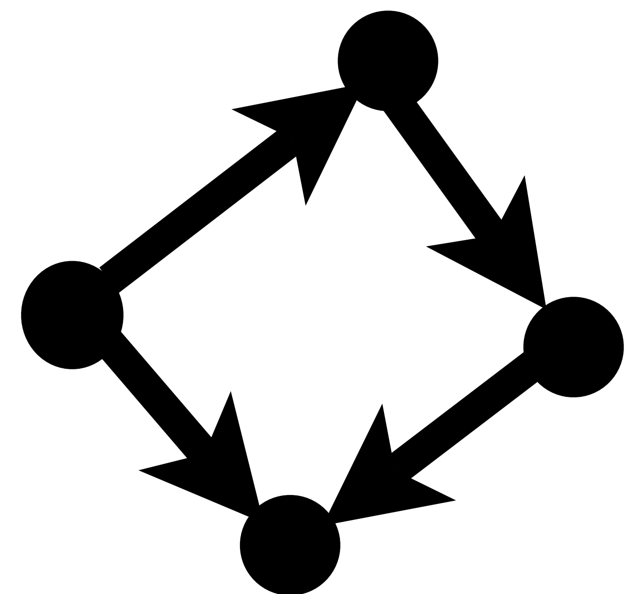


DFS and topo sort

Suppose we run DFS on a directed graph. What can we say about a node v when we mark it as INACTIVE?



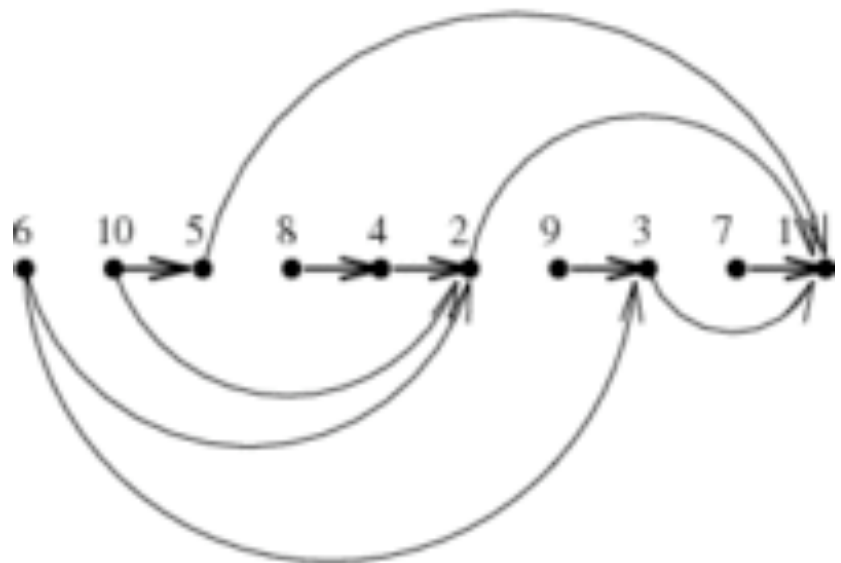


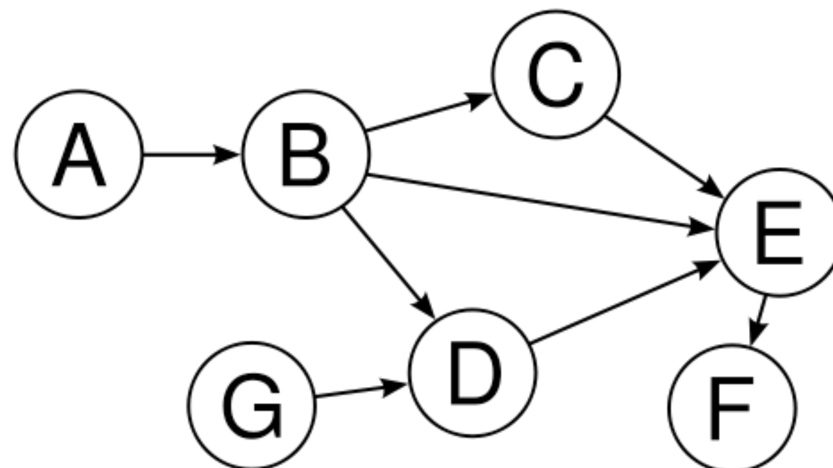


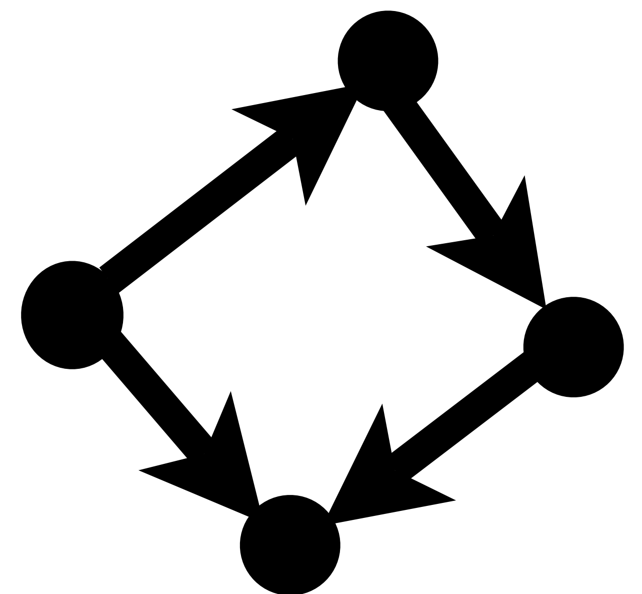
DFS and topo sort

Suppose we run DFS on a directed graph. What can we say about a node v when we mark it as INACTIVE?

Did we fully explore all nodes reachable from v?



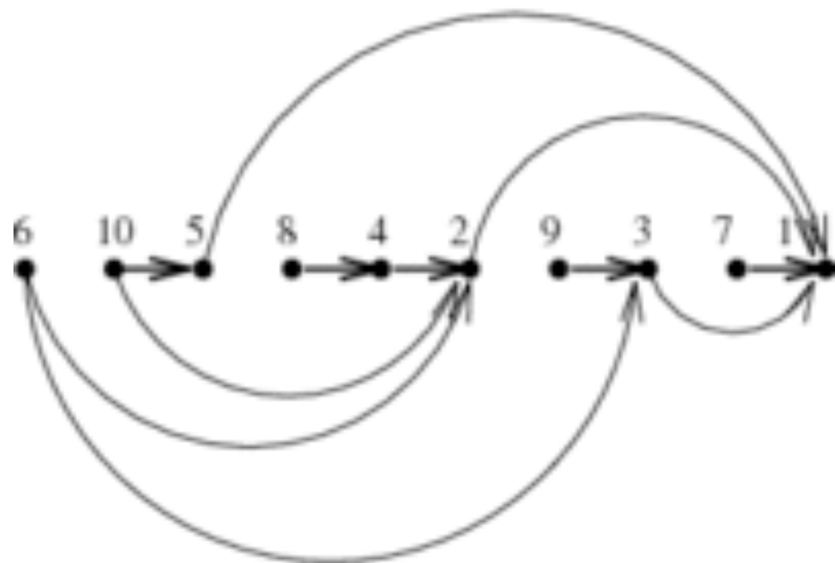


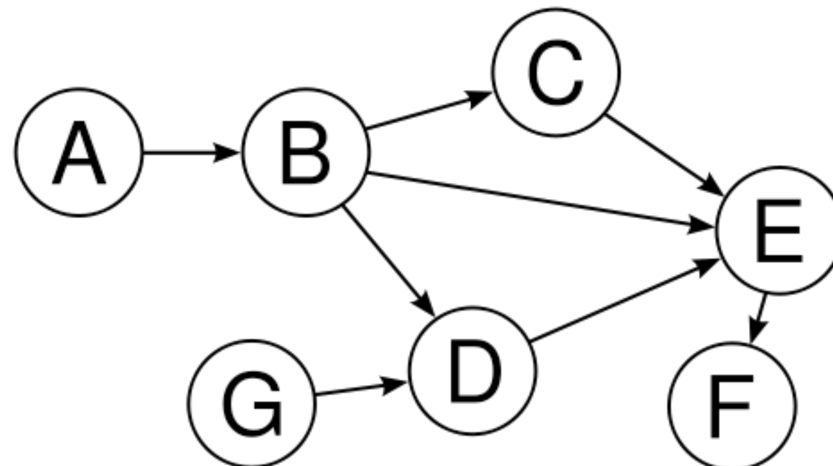


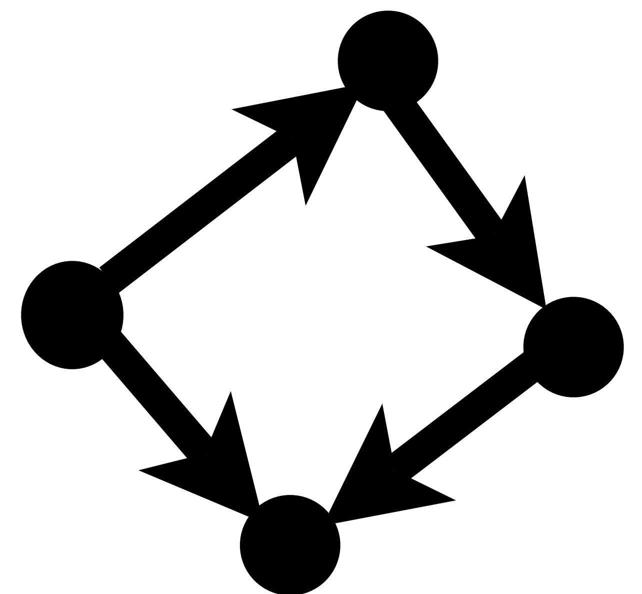
DFS and topo sort

Suppose we run DFS on a directed graph. What can we say about a node v when we mark it as INACTIVE?

Did we fully explore all nodes reachable from v? (to whiteboard)



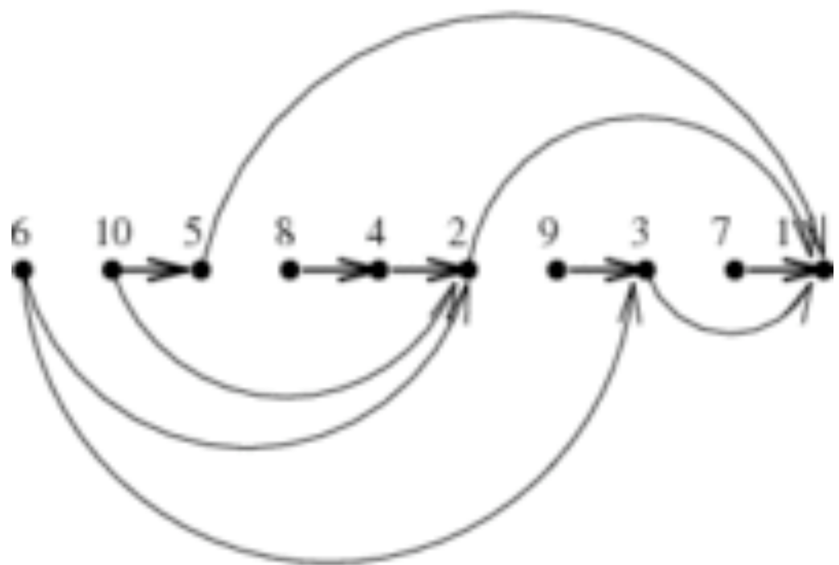


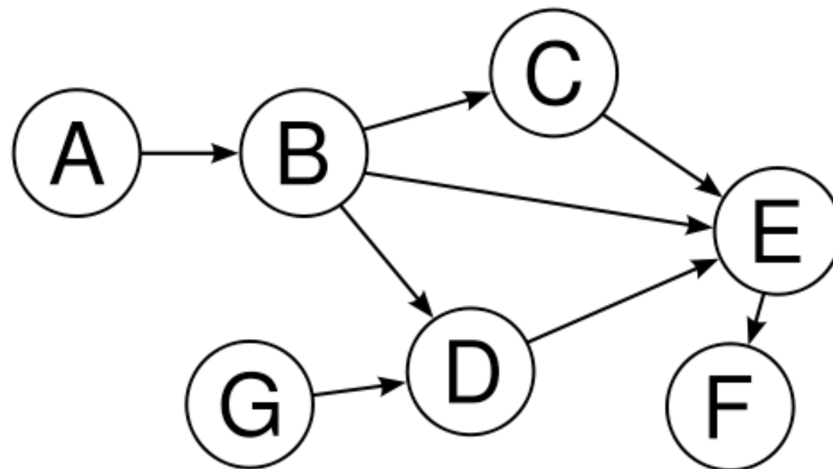


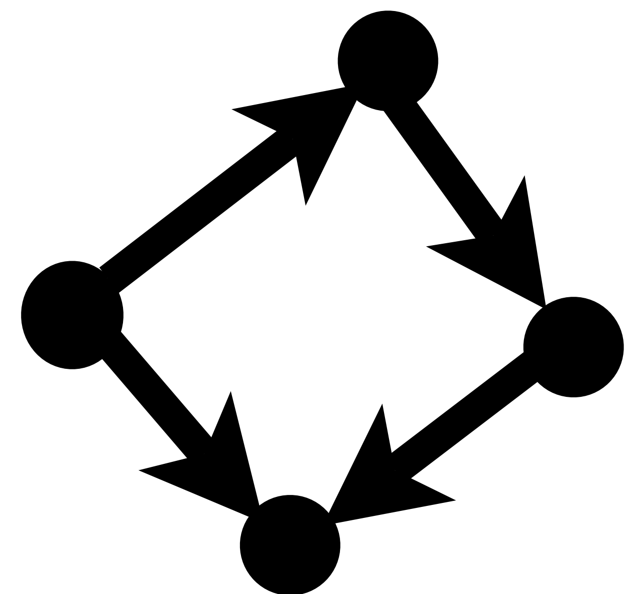
DFS and topo sort

Suppose we run DFS on a directed graph. What can we say about a node v when we mark it as INACTIVE?

Did we fully explore all nodes reachable from v? YES, unless there is a back-edge to an active node (an ancestor of the current node). In this case there is an oriented cycle!.







DFS and topo sort

Suppose we run DFS on a directed graph. What can we say about a node v when we mark it as INACTIVE?

Did we fully explore all nodes reachable from v? YES, unless there is a back-edge to an active node (an ancestor of the current node). In this case there is an oriented cycle!

So: IF we ever find a back-edge to an active node, output the cycle. Otherwise, we know all nodes reachable from v were marked INACTIVE before v was.

Algorithm

Start at a node s. Begin the DFS algorithm, following only out-edges.

Algorithm

Start at a node s. Begin the DFS algorithm, following only out-edges.

If we ever find an edge to an active node, there is a cycle. Each time we mark a node inactive, pre-pend it to the output.

Algorithm

Start at a node s. Begin the DFS algorithm, following only out-edges.

If we ever find an edge to an active node, there is a cycle. Each time we mark a node inactive, pre-pend it to the output.

If graph is a DAG, output will be a topological sort of the nodes reachable from s.

Algorithm

Start at a node s. Begin the DFS algorithm, following only out-edges.

If we ever find an edge to an active node, there is a cycle. Each time we mark a node inactive, pre-pend it to the output.

If graph is a DAG, output will be a topological sort of the nodes reachable from s.

If not given s: start by finding a source (node with no in-edges). A DAG always has one.

Algorithm

Start at a node s. Begin the DFS algorithm, following only out-edges.

If we ever find an edge to an active node, there is a cycle. Each time we mark a node inactive, pre-pend it to the output.

If graph is a DAG, output will be a topological sort of the nodes reachable from s.

If not given s: start by finding a source (node with no in-edges). A DAG always has one.

If not all nodes reached, go to next component.