Lecture 3

Graph traversal. Depth- and breadth-first search

Chinmay Nirkhe | CSE 421 Winter 2026

Previously on CSE 421 …

A writeup for breadth-first search

- **Input**: an undirected graph $G = (V, E)$ and a starting root s
- **Output**: A tree T such that $d_T(s, v) = d_G(s, v)$ for any vertex $v \in G$. (For any unreachable vertex v , by convention, $d_G(s, v) = \infty$ and v is not included in T.)
- **Algorithm:**
  - **Details:** Initialize a queue Q with s and empty tree T. While Q isn't empty, pop v off and mark as visited. Then and add all unvisited neighbors w of v to the queue and add edge (v, w) to T.
  - **Runtime:** Each edge and vertex is visited/referenced at most $O(1)$ times so total complexity is at most $O(|V| + |E|)$.

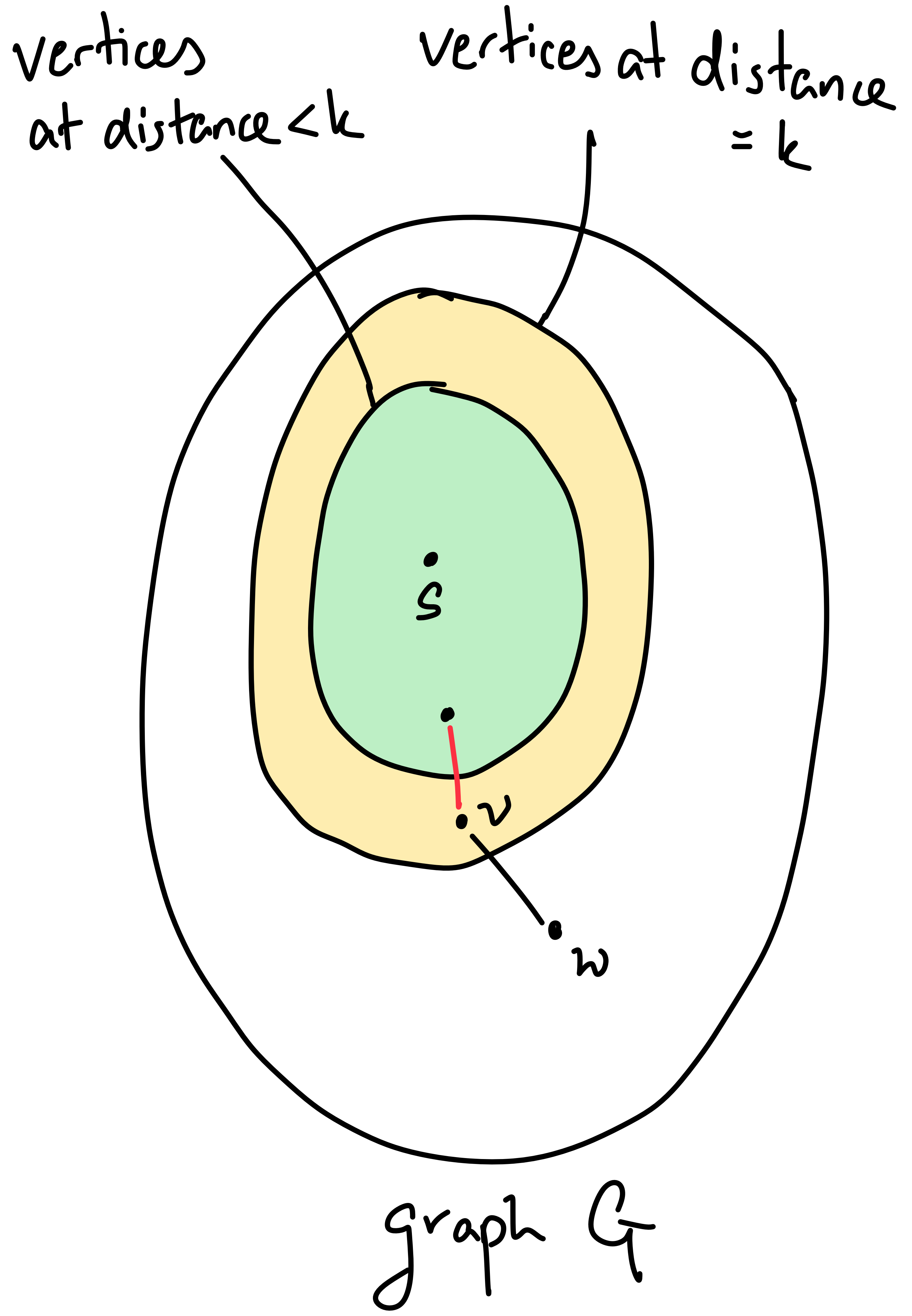
Today

A writeup for breadth-first search

Correctness argument

- **Claim:** A tree T such that $d_T(s, v) = d_G(s, v)$ for any vertex $v \in G$.
- **Stronger claim:** A tree T such that $d_T(s, v) = d_G(s, v)$ for any vertex $v \in G$ **and** BFS dequeues vertices in monotonically increasing order of distance.
- **Proof:** (Induction)
    - Base case: s is the only vertex at distance 0 and it is dequeued first. Also $d_T(s, s) = d_G(s, s)$.
    - Induction: Assume that for all vertices v with $d_G(s, v) = k$, that $d_T(s, v) = d_G(s, v) = k$ and that they are dequeued before vertices at distance $> k$.
    - Let w be a vertex at distance $d_G(s, w) = k + 1$ and v its predecessor on the shortest G-path to s. Then, $d_G(s, v) = k$.
    - When BFS dequeues v, it observes the edge (v, w) with w unvisited (by induction) and adds (v, w) to T. So,

    $$d_T(s, w) = d_T(s, v) + 1 = d_G(s, v) + 1 = d_G(s, w).$$



Connected components

- For a undirected graph G, a connected component $C \subseteq V$ is a **maximal set** such that
  - For all pairs $u, v \in C$, there exists a path $u \rightsquigarrow v$
  - There are no edges between C and $V \backslash C$.
- Then, $u \rightsquigarrow v$ iff u, v in the same connected component

Connected components

- **Algorithm for computing connected components:**
  - Idea: Let $V = \{1, \ldots, n\}$. Create an array $A(u) =$ smallest numbered vertex connected to u. A canonical name for the connected component.
  - Then u and v are connected iff $A(u) = A(v)$. Better than storing all pairs of paths $p(u, v)$.

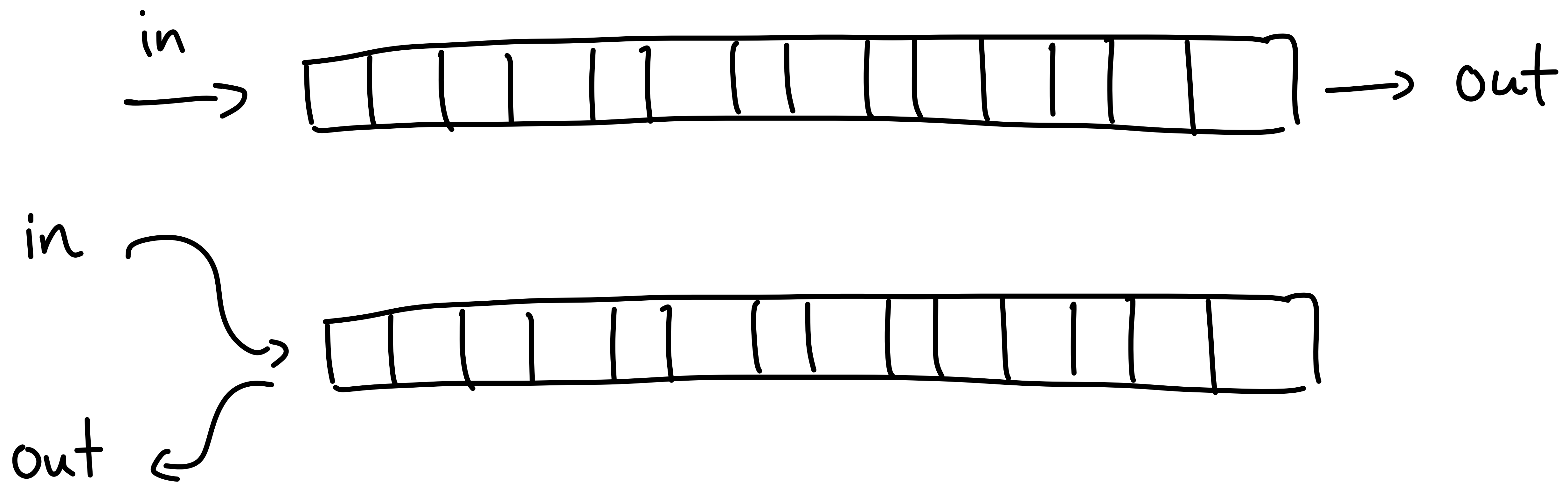
Faster when all pairs are being compared.

Connected components

- **Algorithm for computing connected components:**
  - Initialize all vertex as not visited.
  - For $s \leftarrow 1$ till n,
    - If s is not visited, then run subroutine BFS(s) and set $A(u) \leftarrow s$ for every vertex visited by the BFS and mark each vertex as visited.
- **Correctness:** (sketch) Prove by induction on vertex number u, that $A(u)$ equals the smallest numbered vertex connected to u.
- **Total runtime:** $O(n + m)$ because
  - Each vertex is visited once by outer routine and the BFS runs are disjoint and observes each edge a constant number of times.
  - Could have run any generic graph traversal actually as long as it is efficient

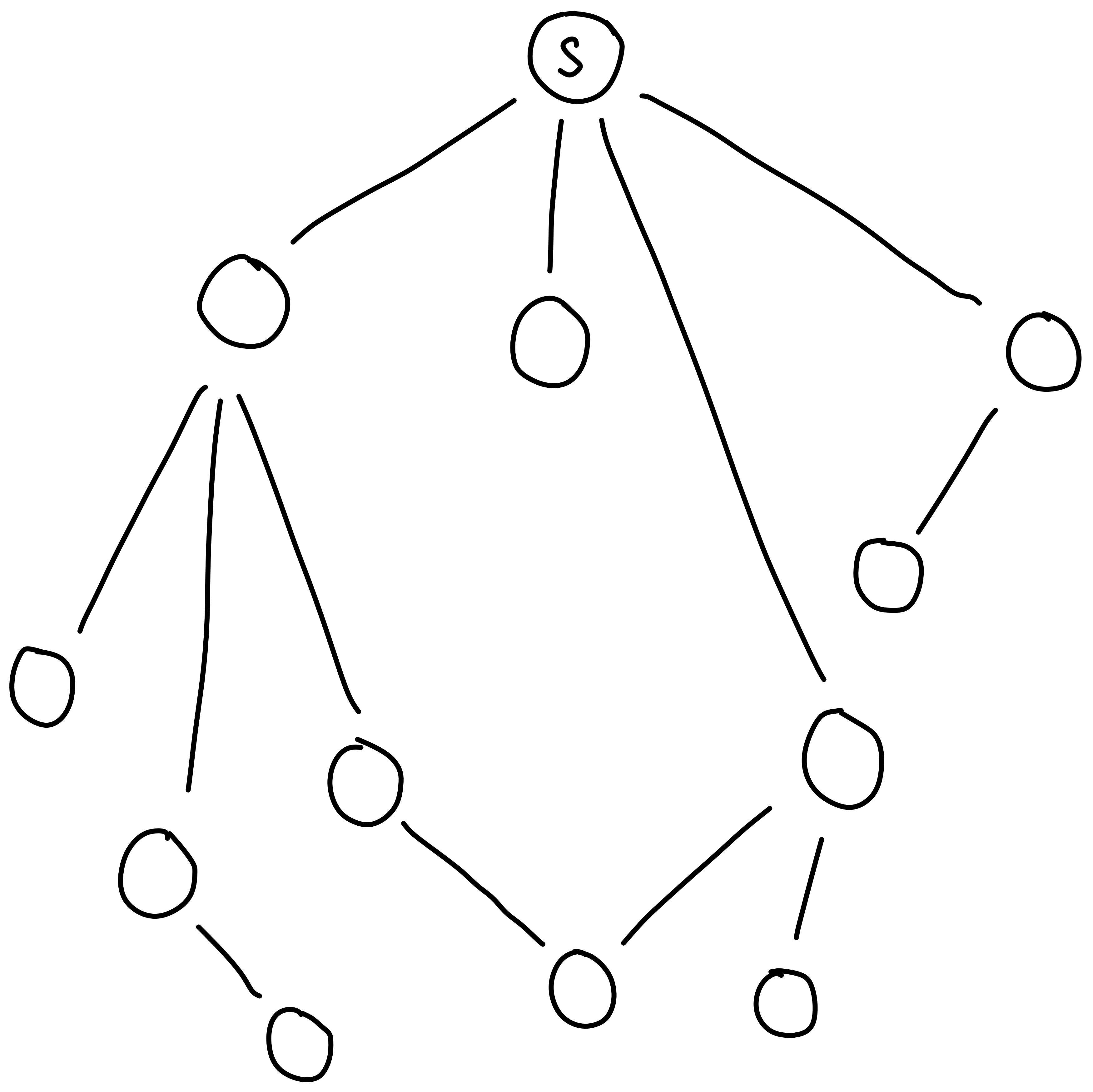
Depth-first search

- Breadth-first search visits all the neighbors before diving in deeper
- Depth-first search visits as deep as possible
- The trees formed by the visiting order look quite different!
- Generated by different data structures but similar algorithm!
  - BFS: Queue — *first in, first out*
  - DFS: Stack — *first in, last out*



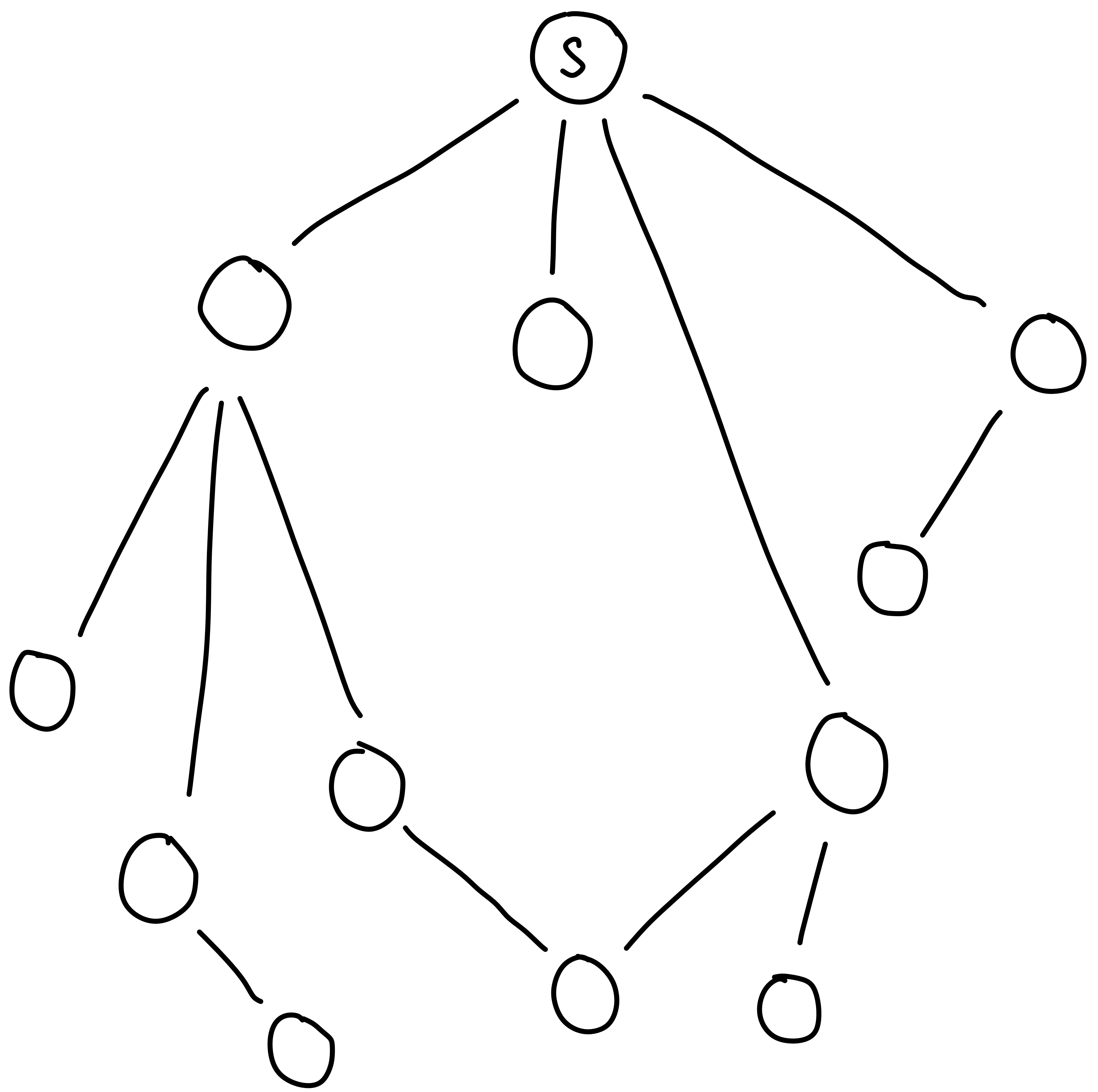
Breadth-first search (BFS)

- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. Set s as visited.
  - While Q isn't empty, pop v off the queue.
    - For every neighbor u of v that is not visited,
      - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
      - Set $R \leftarrow R \cup \{u\}$.



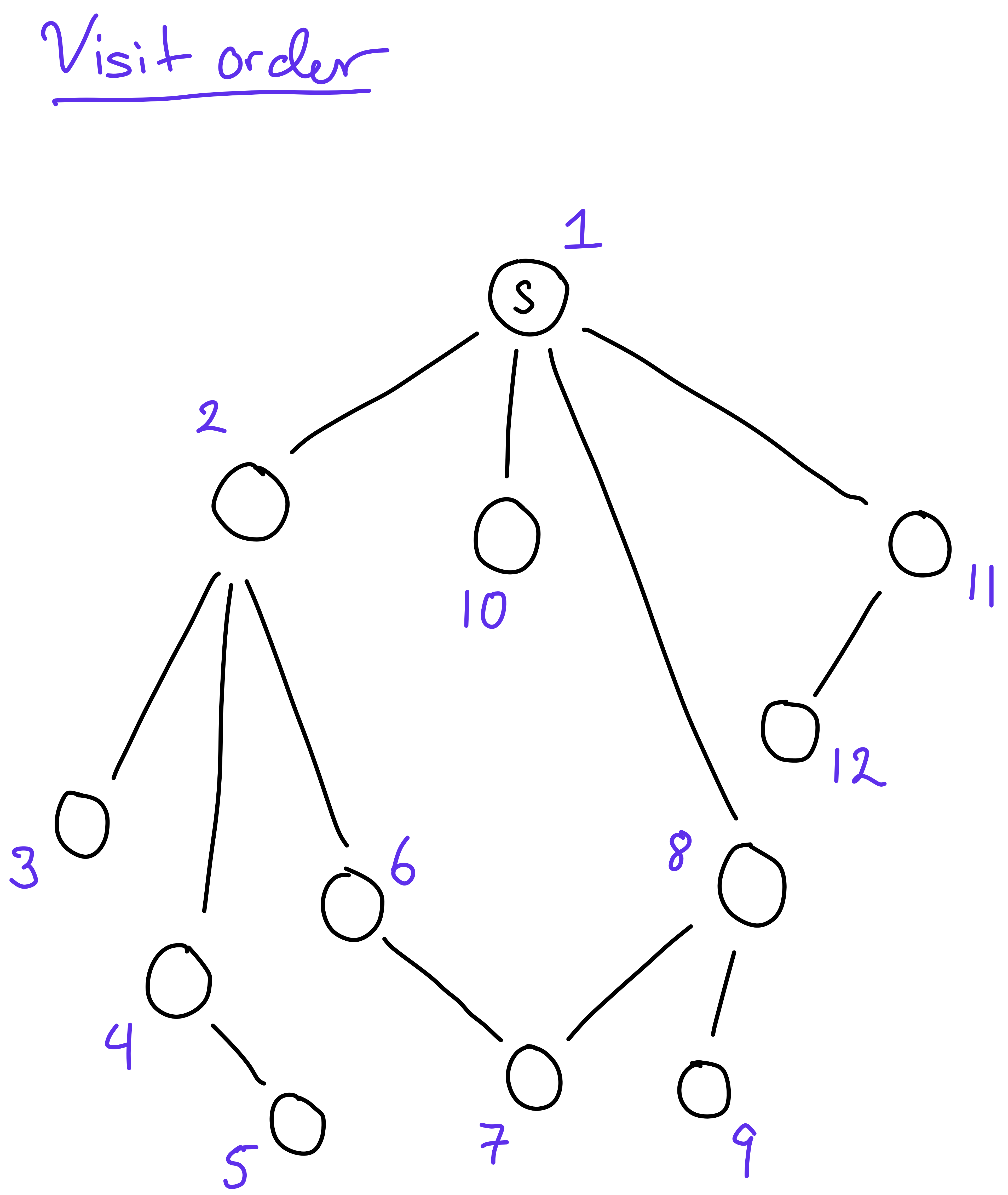
Depth-first search (BFS)

- Assign a bit to every vertex as visited/not visited.
- **Algorithm**:
    - Initialize set $R \leftarrow \{s\}$ and stack $S \leftarrow \{s\}$.
    - Set all vertices to not visited.
    - While S isn't empty, pop v off the stack.
        - If v is not visited, set v to visited
            - For every neighbor u of v that is not visited,
                - $S \leftarrow S \cup \{u\}$.
                - Set $R \leftarrow R \cup \{u\}$.



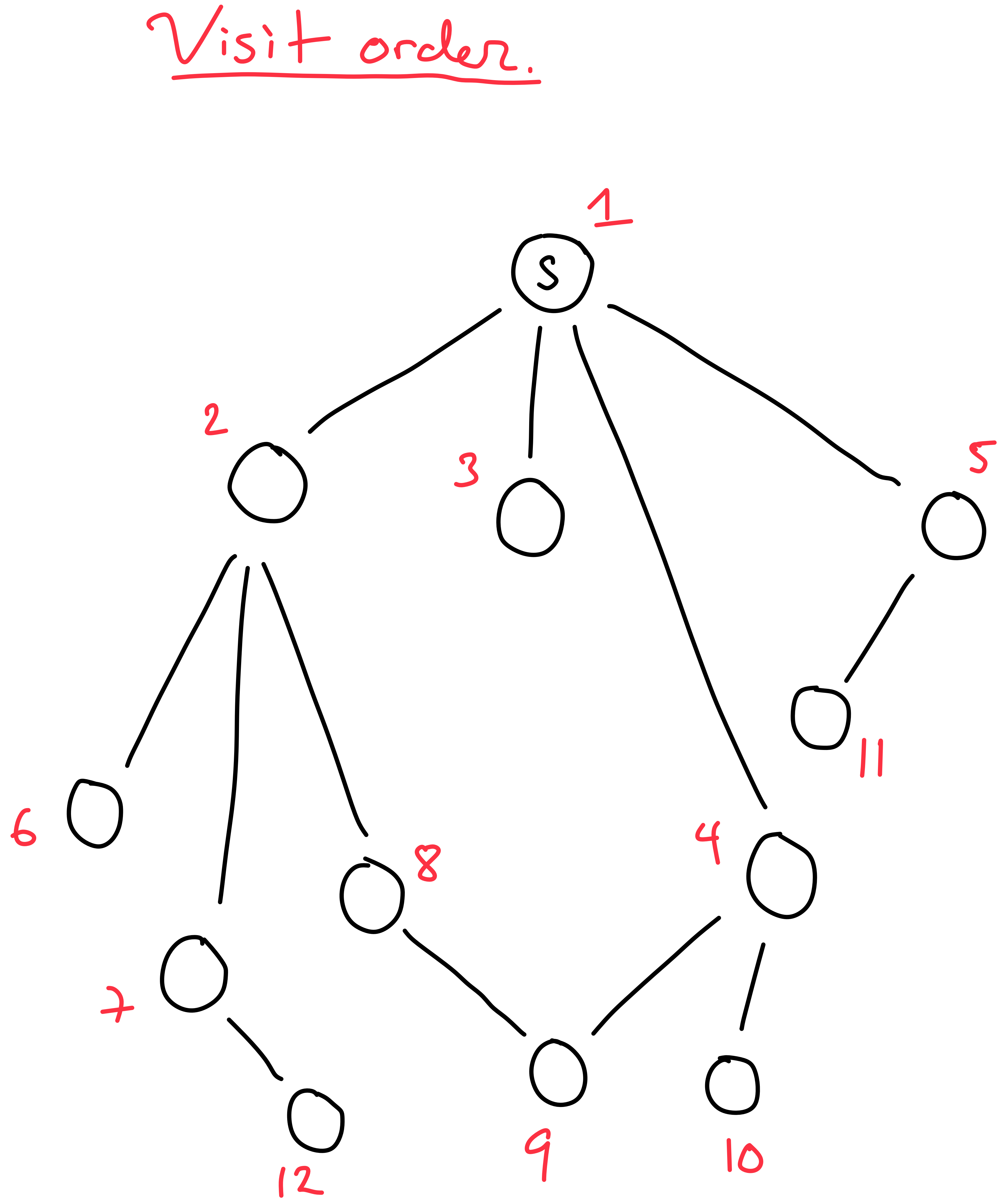
Depth-first search (BFS)

- Assign a bit to every vertex as visited/not visited.
- **Algorithm**:
  - Initialize set $R \leftarrow \{s\}$ and stack $S \leftarrow \{s\}$.
  - Set all vertices to not visited.
  - While S isn't empty, pop v off the stack.
    - If v is not visited, set v to visited
      - For every neighbor u of v that is not visited,
        - $S \leftarrow S \cup \{u\}$.
        - Set $R \leftarrow R \cup \{u\}$.



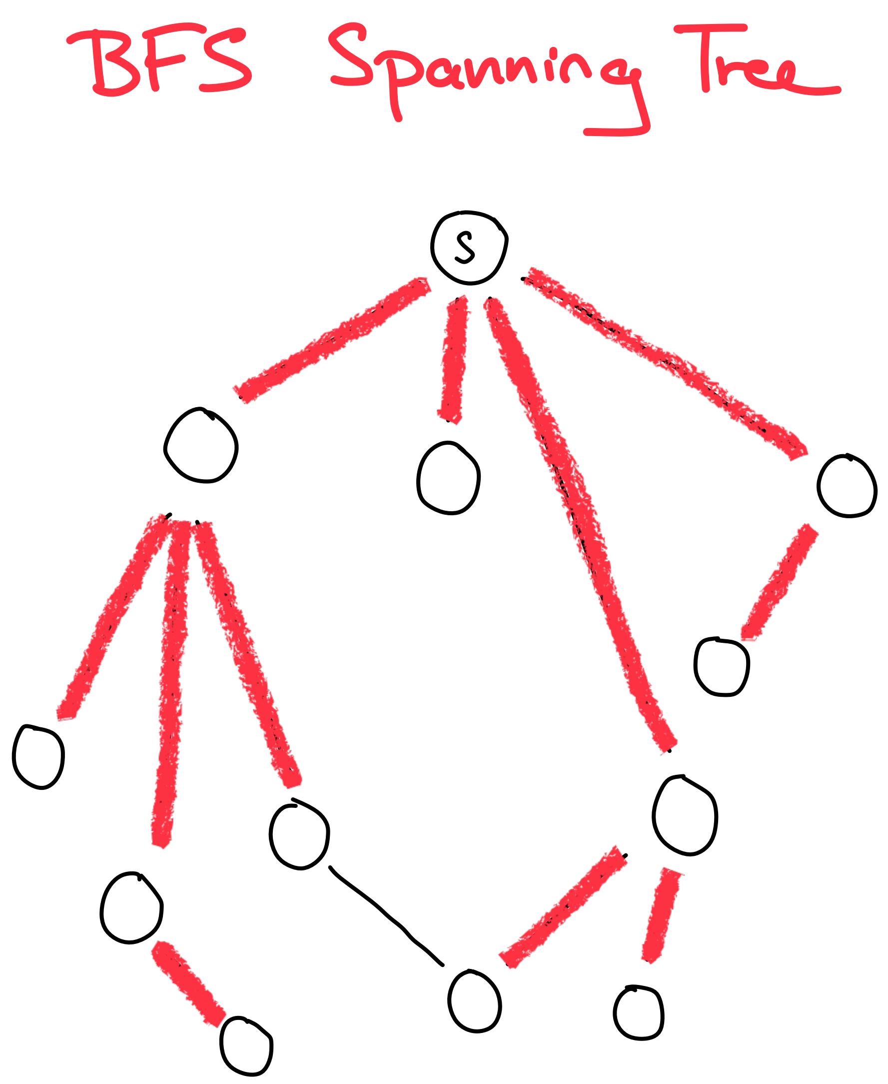
Breadth-first search (BFS)

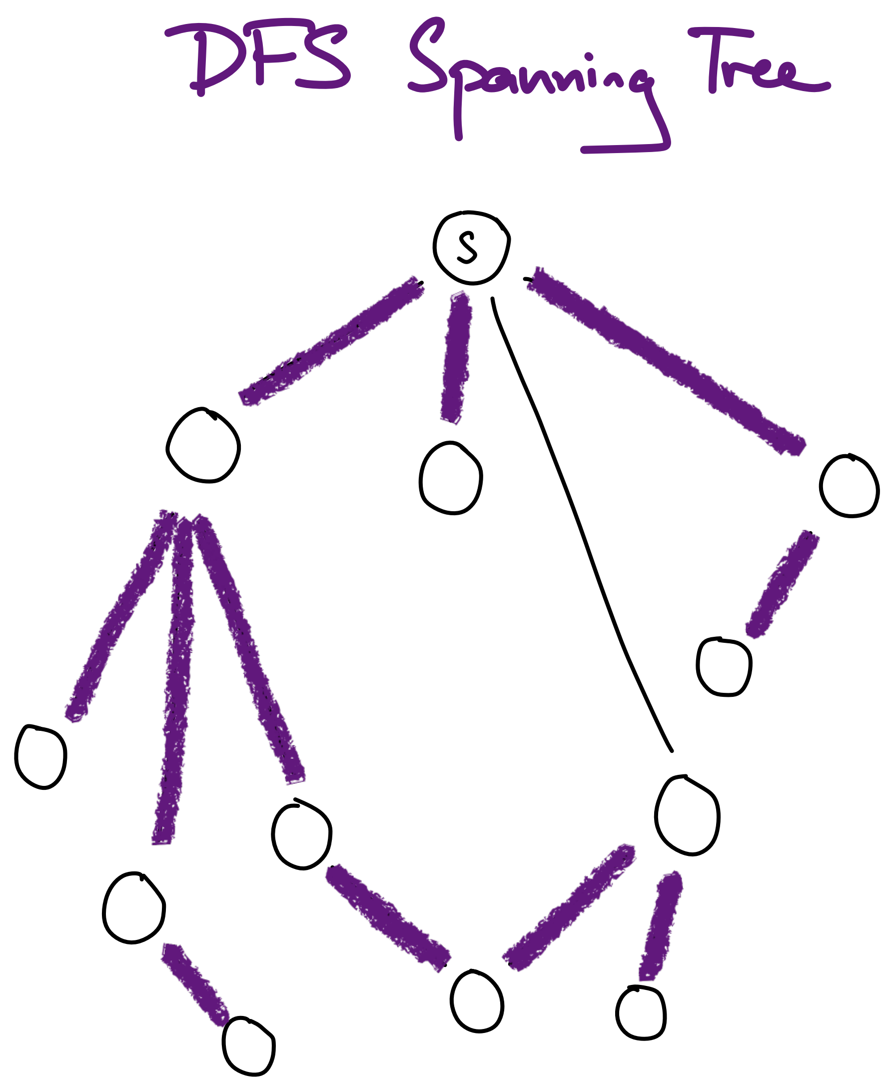
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
    - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
    - Set all vertices to not visited. Set s as visited.
    - While Q isn't empty, pop v off the queue.
        - For every neighbor u of v that is not visited,
            - $Q \leftarrow Q \cup \{u\}$ and set u to visited.
            - Set $R \leftarrow R \cup \{u\}$.



Spanning trees

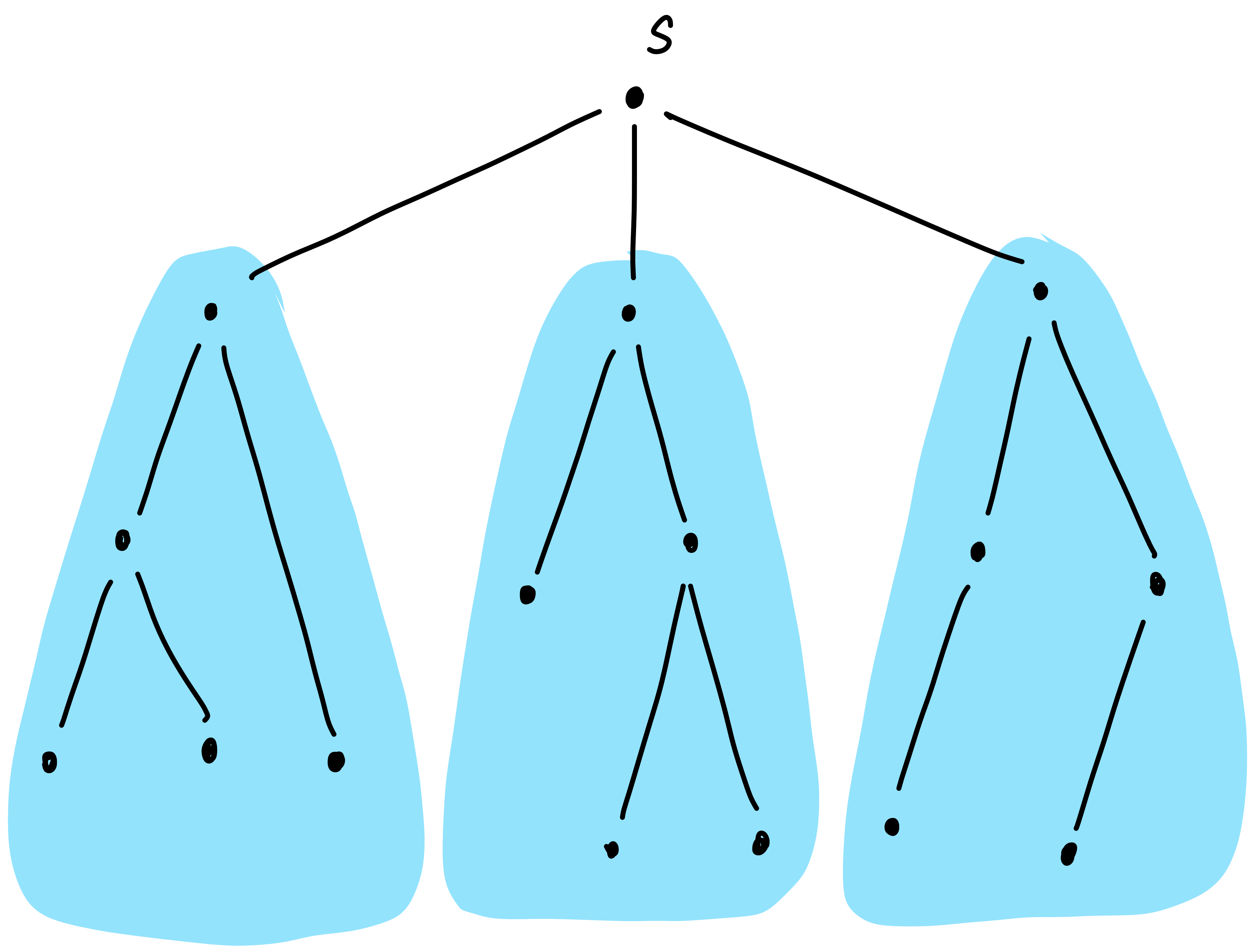
- A spanning tree $T \subseteq E$ is a tree (no cycles) for a connected component such that every vertex in the component touches T.
- BFS and DFS both generate spanning trees but they are different!





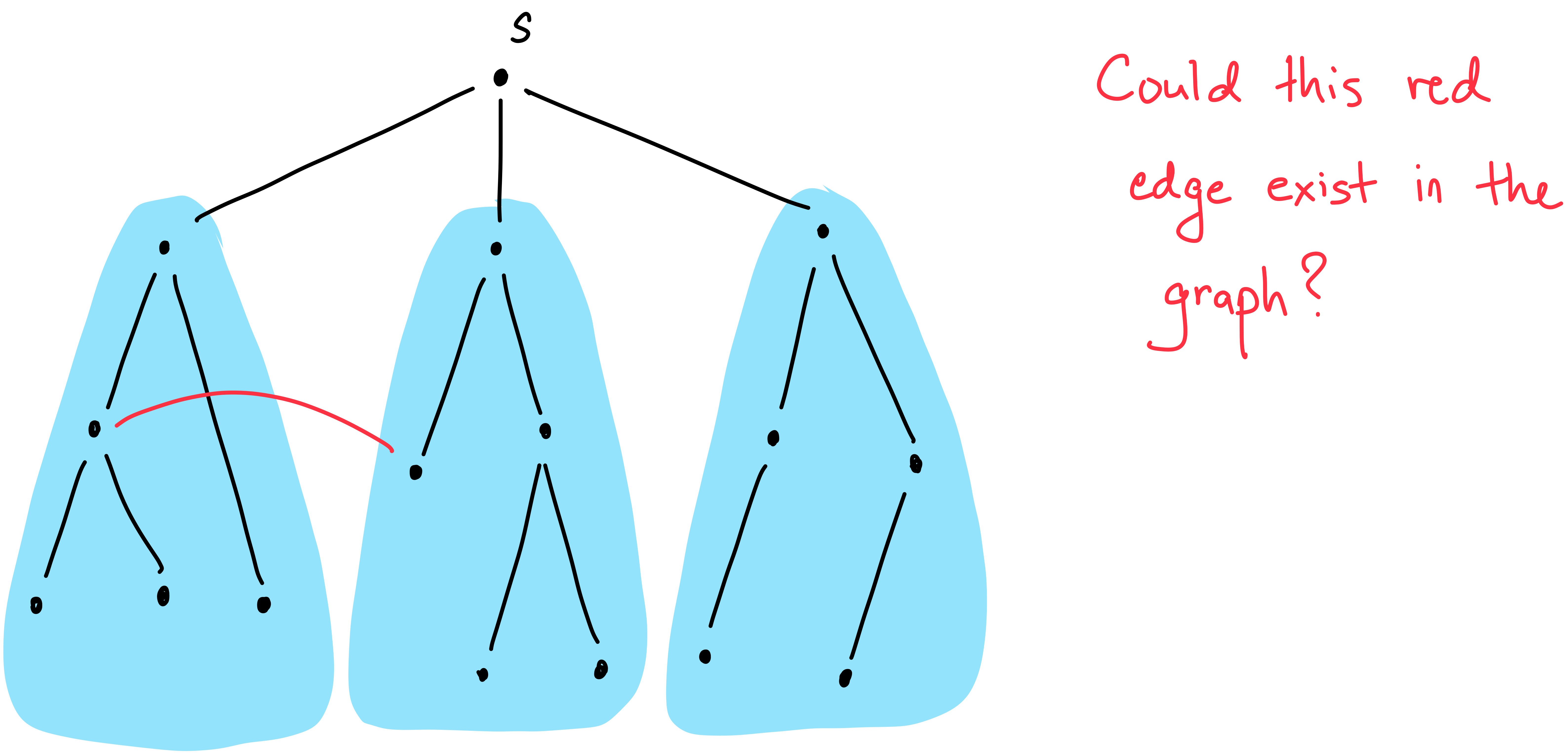
Understanding the DFS spanning tree

- What do the edges not included in the spanning tree look like?



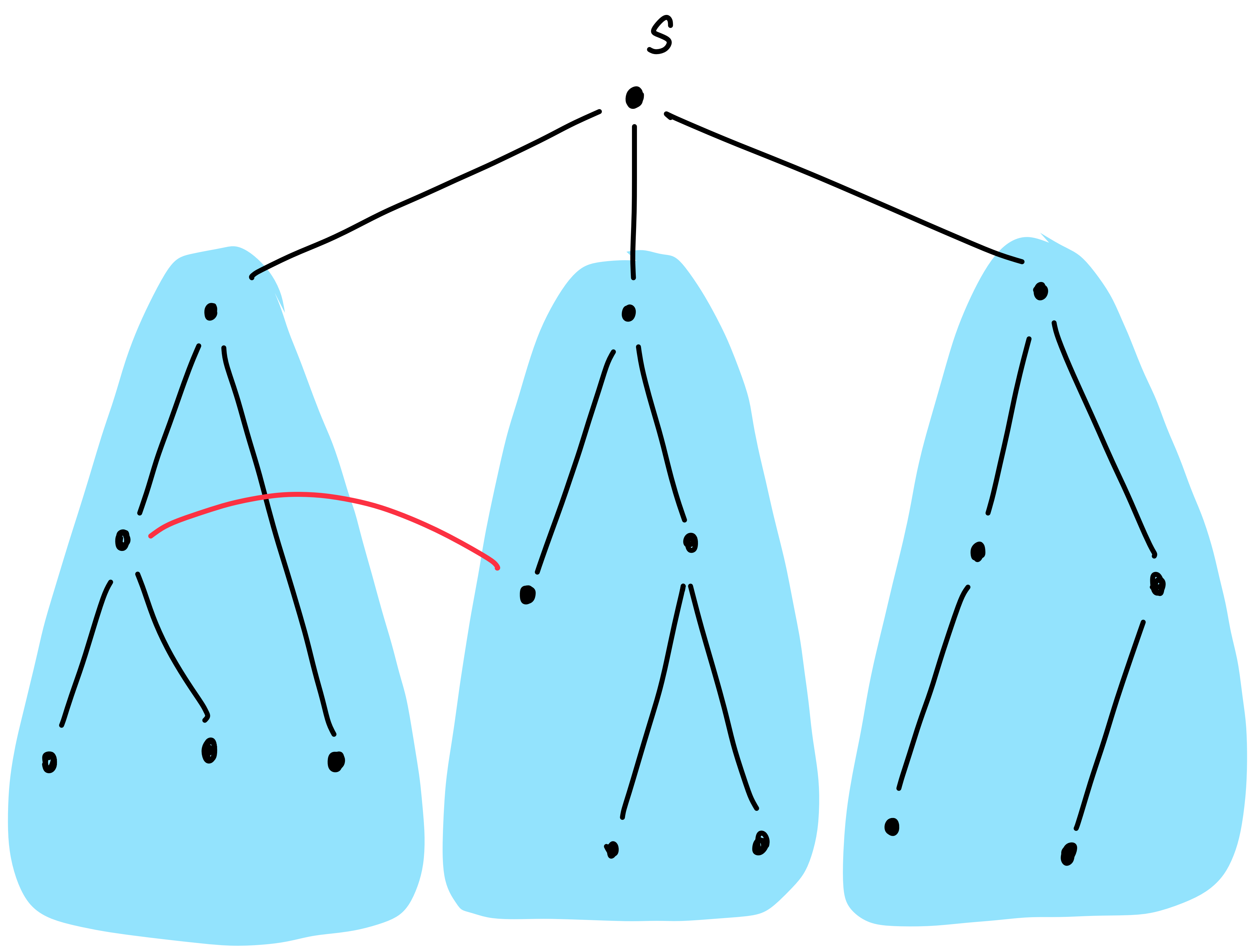
Understanding the DFS spanning tree

- What do the edges not included in the spanning tree look like?



Understanding the DFS spanning tree

- What do the edges not included in the spanning tree look like?



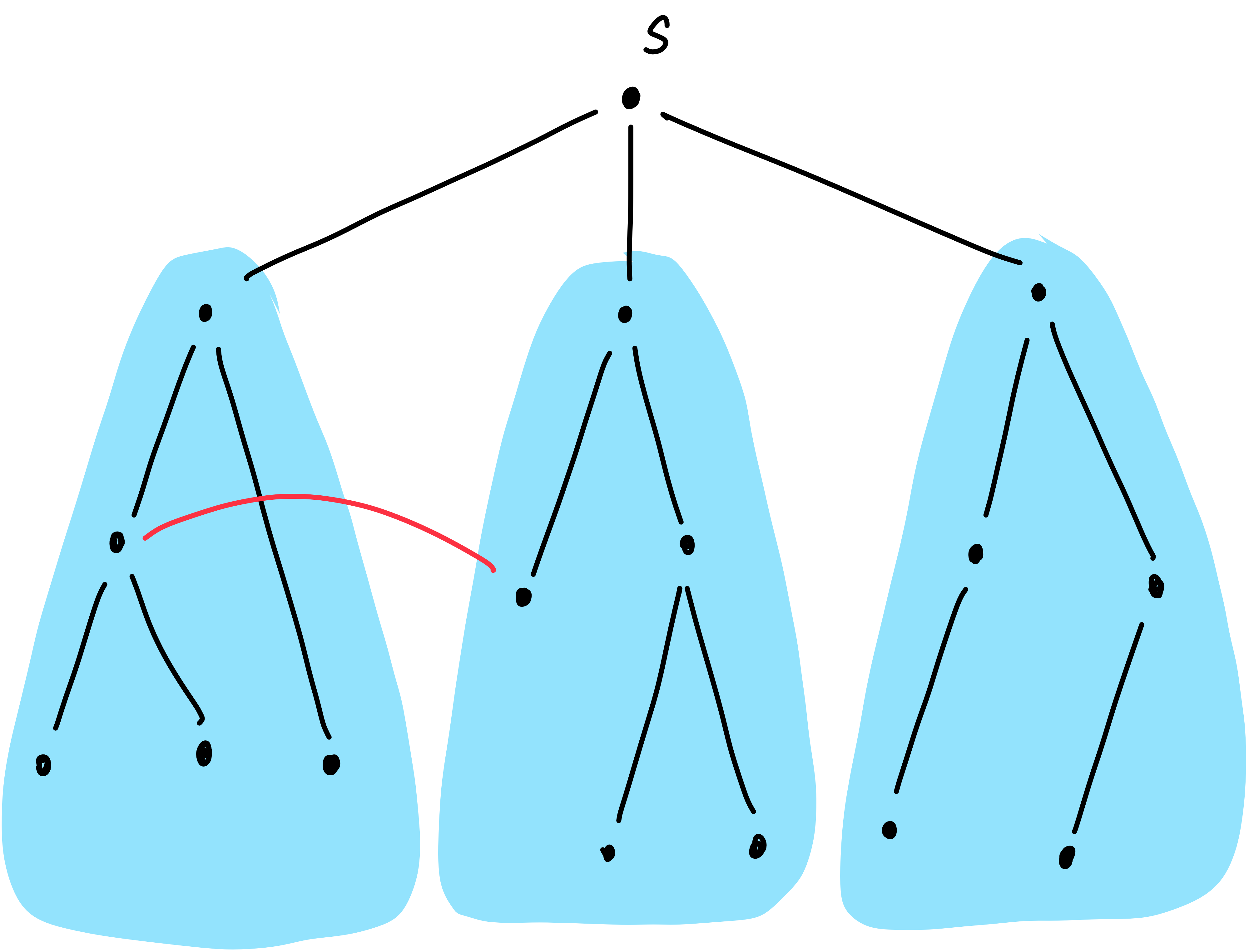
Could this red edge exist in the graph?

No. When adding a , we must have added the red edge.

Understanding the DFS spanning tree

- What do the edges not included in the spanning tree look like?

Such an edge is called a cross edge.

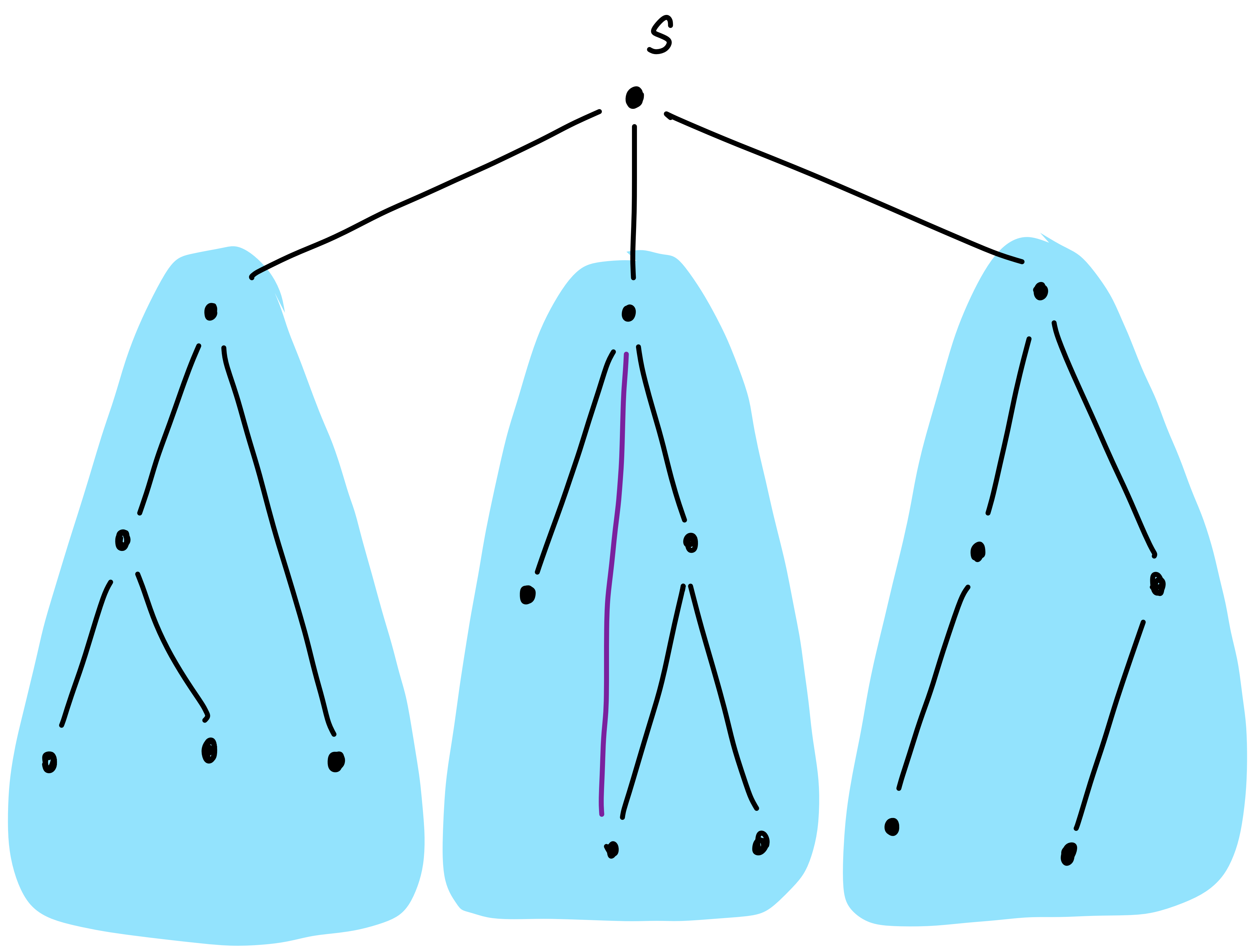


Could this red edge exist in the graph?

No. When adding a , we must have added the red edge.

Understanding the DFS spanning tree

- What do the edges not included in the spanning tree look like?

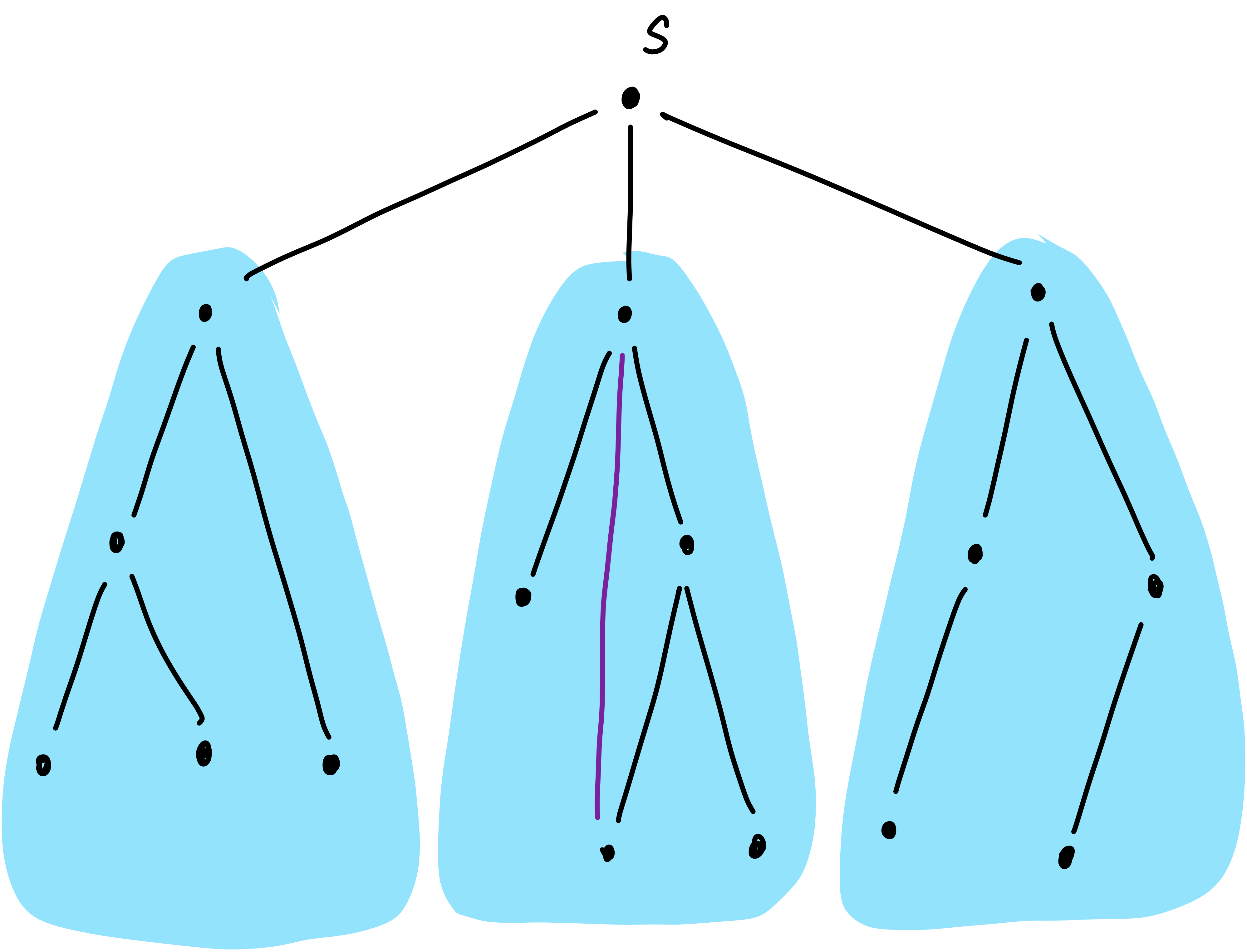


What about this purple edge?

Understanding the DFS spanning tree

- What do the edges not included in the spanning tree look like?

Def. For undirected graphs, a <u>back edge</u> is an edge that connects a vertex to a (proper) ancestor in the tree.



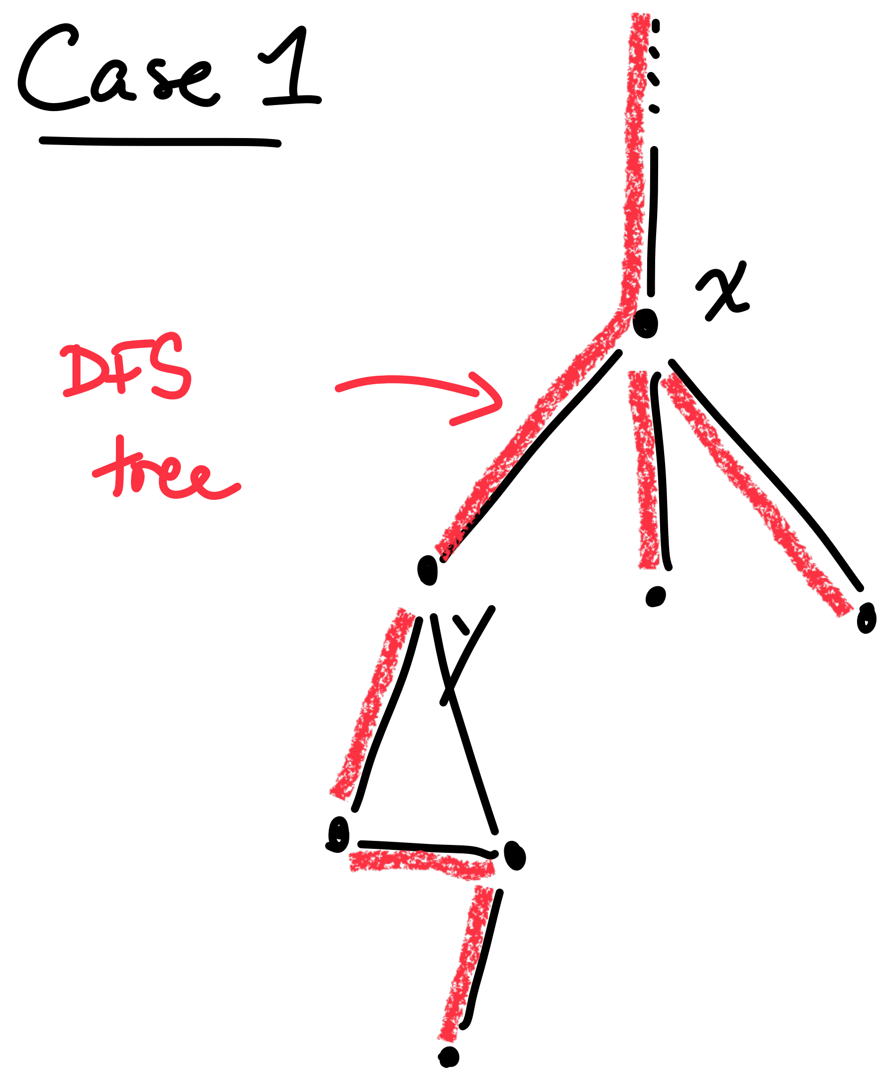
What about this purple edge?

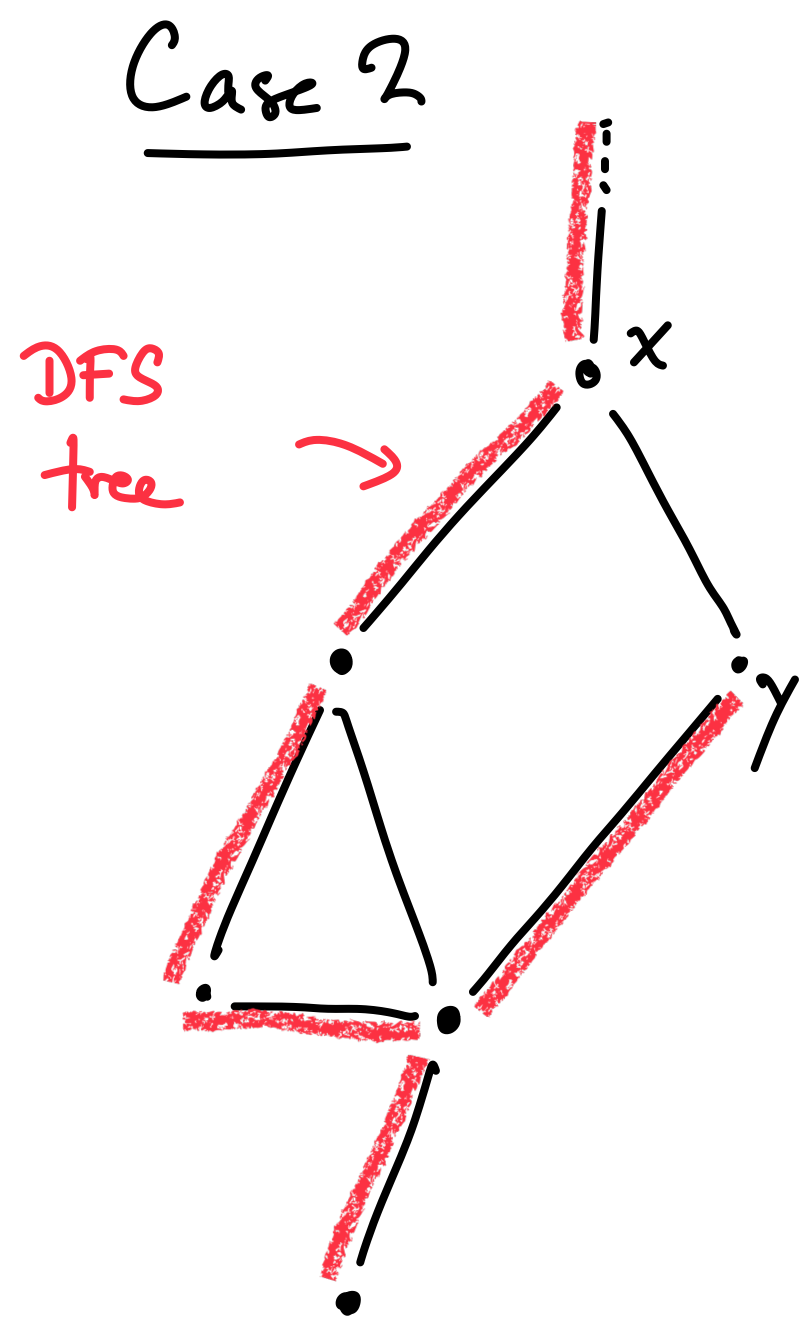
Yes. Edges can connect down tree.

No cross edges in DFS

(for undirected graphs)

- **Claim:** For every edge $(x, y) \in E$, either (x, y) is an edge in T (tree edge), or else x or y is an ancestor of the other in T (back edge).
- **Proof:**
  - DFS is called recursively as we explore. Wlog, assume DFS(x) is called before DFS(y).
  - Case 1: y was marked "not visited" when (x, y) edge is examined. Then $(x, y) \in T$ (see figure).
  - Case 2: y was marked "visited" when (x, y) edge is examined. Was visited in some other branch of the DFS(x) call. So y is a descendant of x.

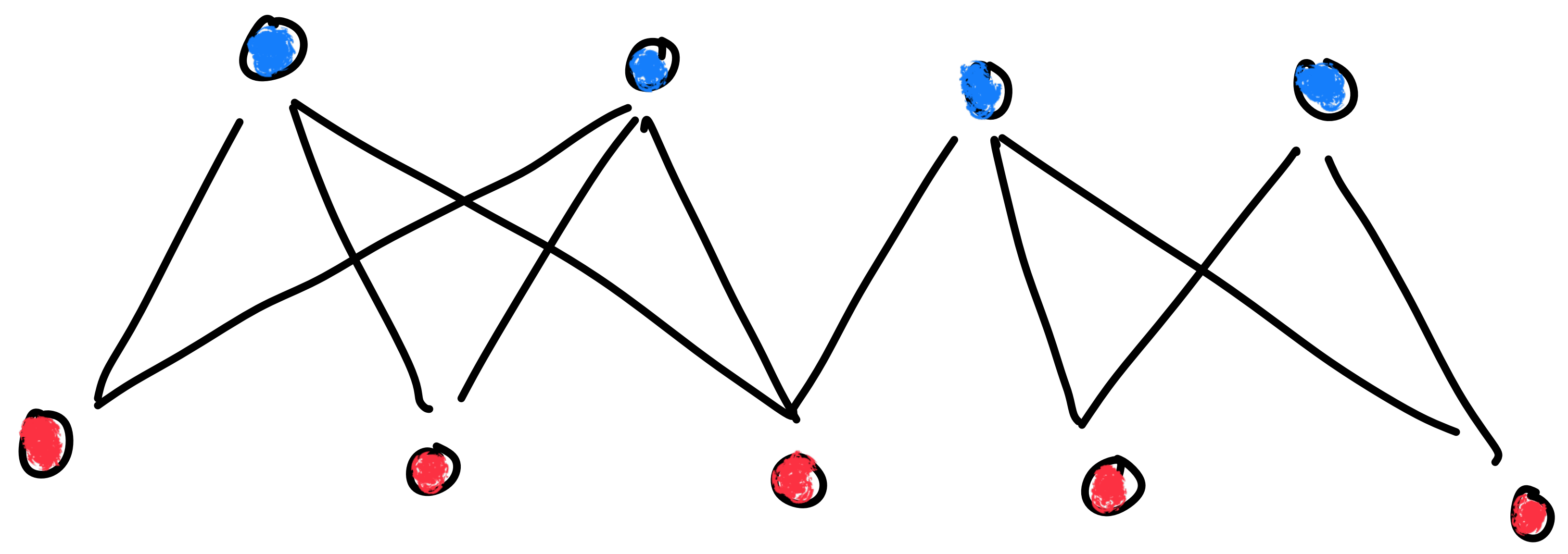




Applications of graph traversal

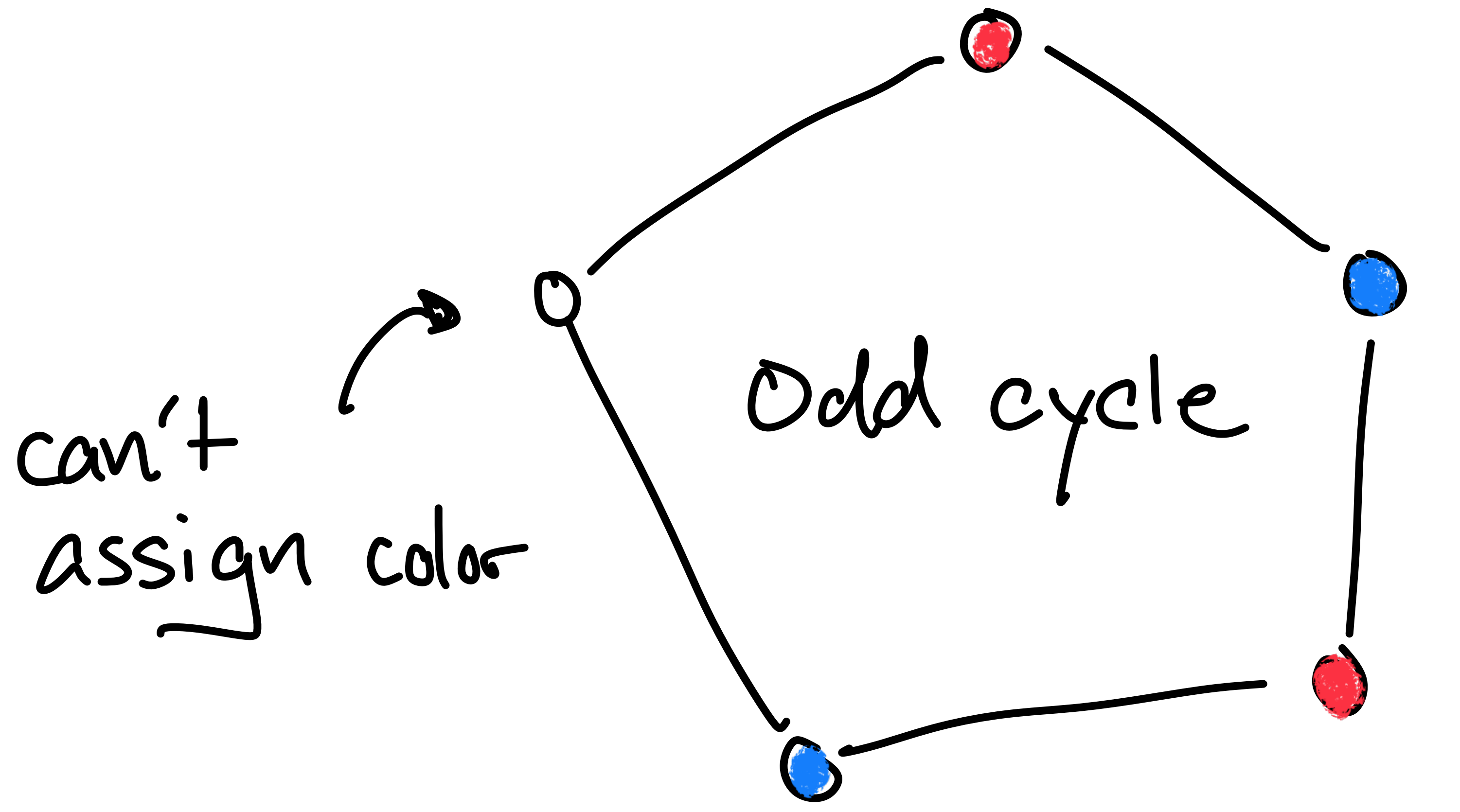
Bipartiteness testing

Application of graph traversal

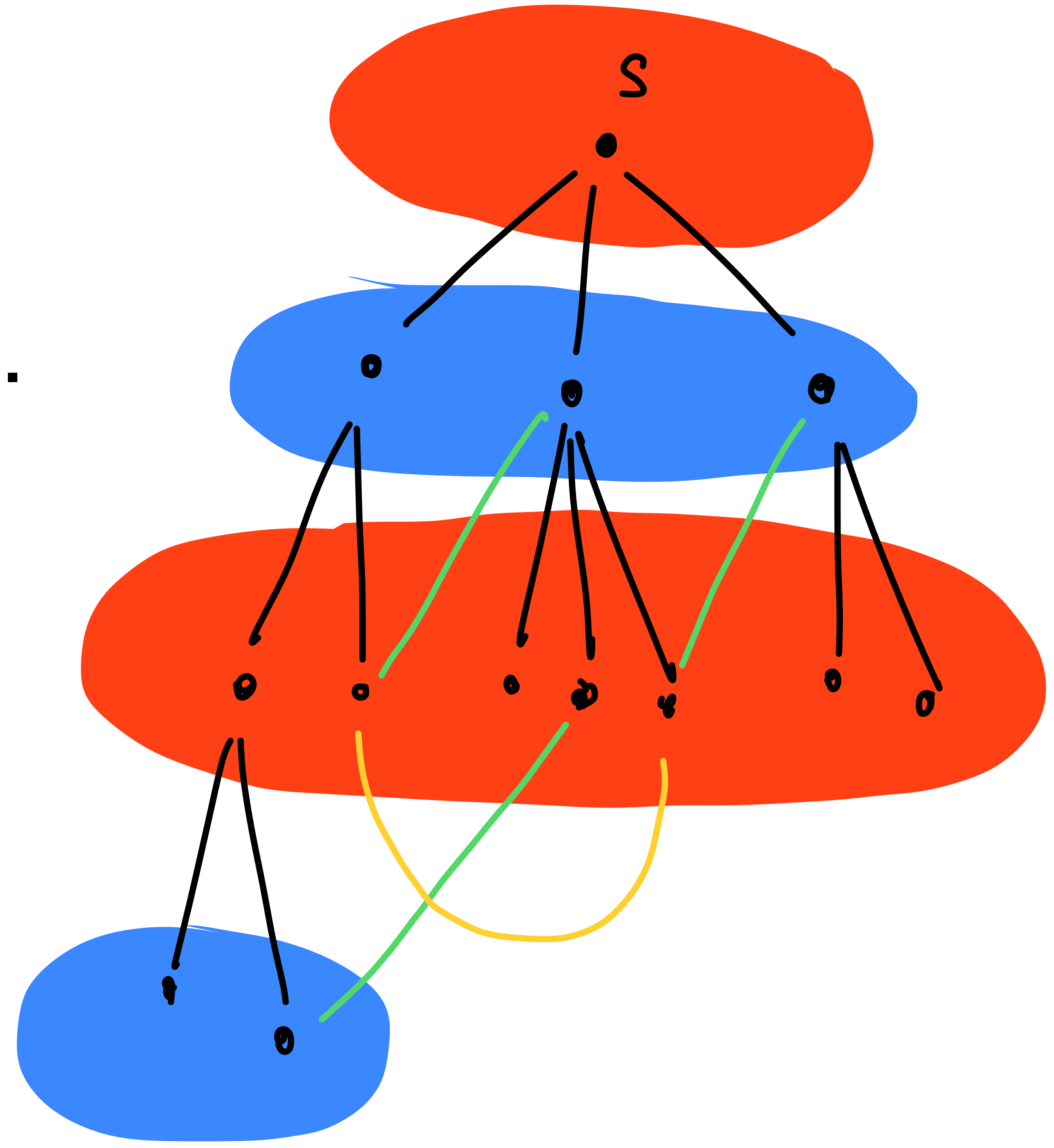


- Recall, a graph is bipartite iff we can split $V = X \sqcup Y$ such that every edge is between $(x, y) \in X \times Y$.
- Equivalently, a graph is bipartite if we can color every vertex either *red* or *blue* such that each edge is between a red and a blue vertex.
- **Input:** Undirected graph G
- **Output:** A coloring $c : V \to \{$red, blue$\}$if G is bipartite; else “not bipartite”

Bipartite graph property

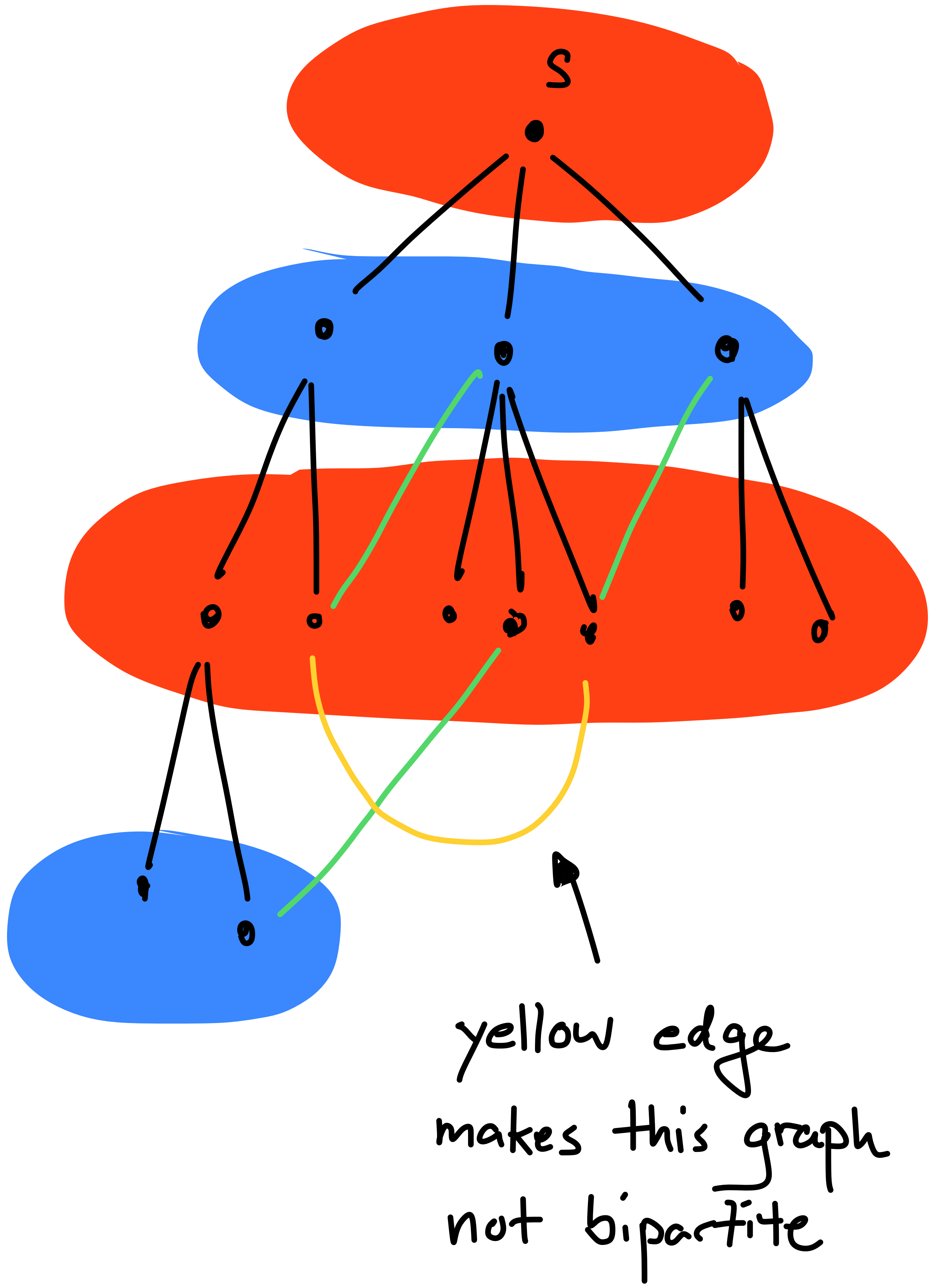


- **Claim:** A graph is bipartite iff it contains no *odd cycles*.
- **Proof:**
  - If it contains an odd cycle, we can't color the cycle let alone the rest of the graph.
  - If it contains no odd cycles, run BFS starting from some vertex s.
    - Color according to length from s in BFS tree with even = red, odd = blue.
    - If there exists an edge between colors, we found an odd cycle, (a ⊥ to our assumption).



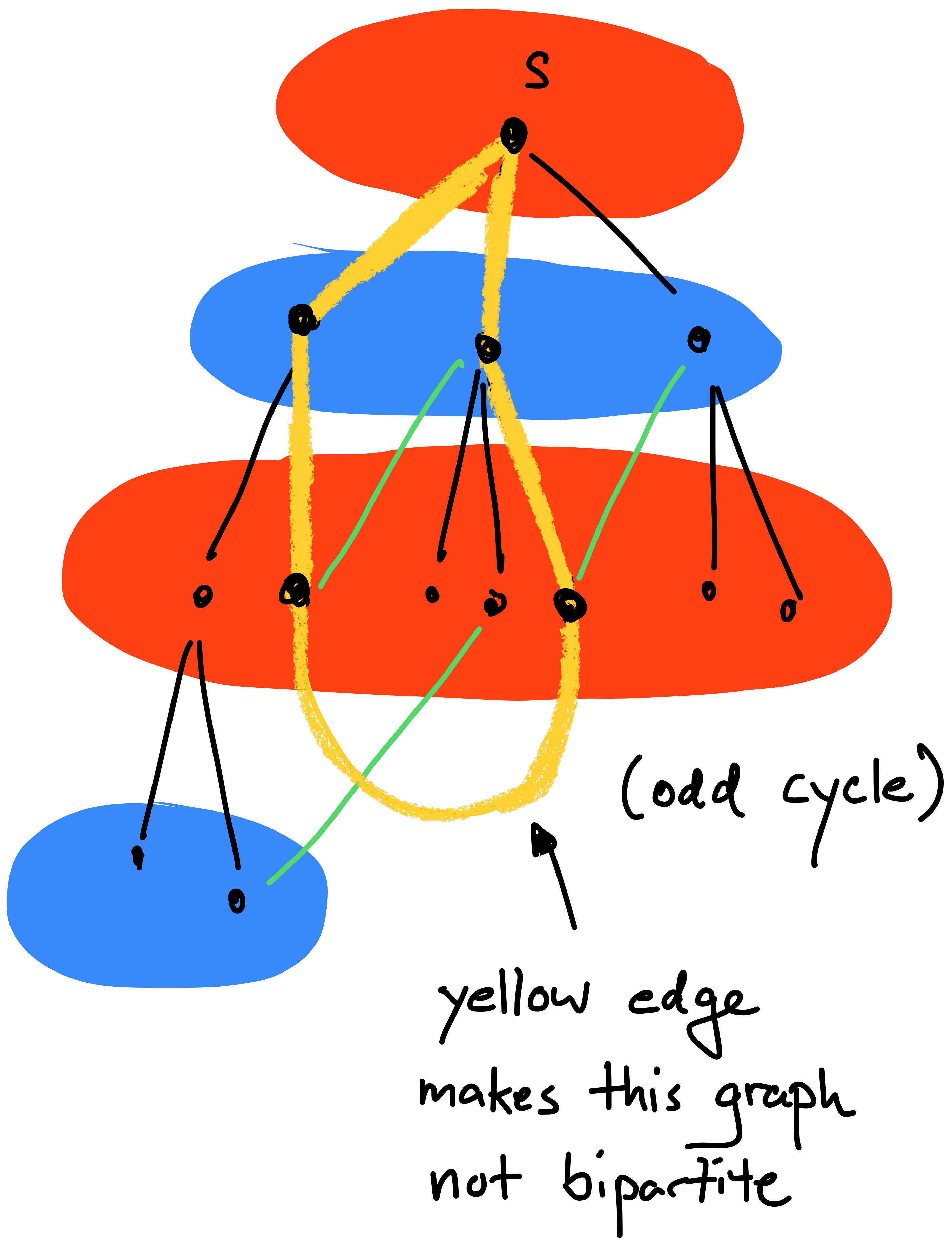
Bipartiteness testing

- **Claim:** A graph is bipartite iff it contains no *odd cycles*.
- **Algorithm:**
  - Start BFS from some vertex s. Instead of marking vertices as visited or not, marked them as "red", "blue", or "not visited". Mark s as red and add s to queue Q.
  - Pop vertex u from queue Q.
    - Check all neighbors v of u and make sure they are either "not visited" or the opposite color of u.
    - If not, abort and output "not bipartite".
    - If so, add the "not visited" neighbors v to the queue Q and color them with opposite color.
  - If queue Q is empty, output coloring generated.
- **Runtime:** Same as BFS, $O(n + m)$.

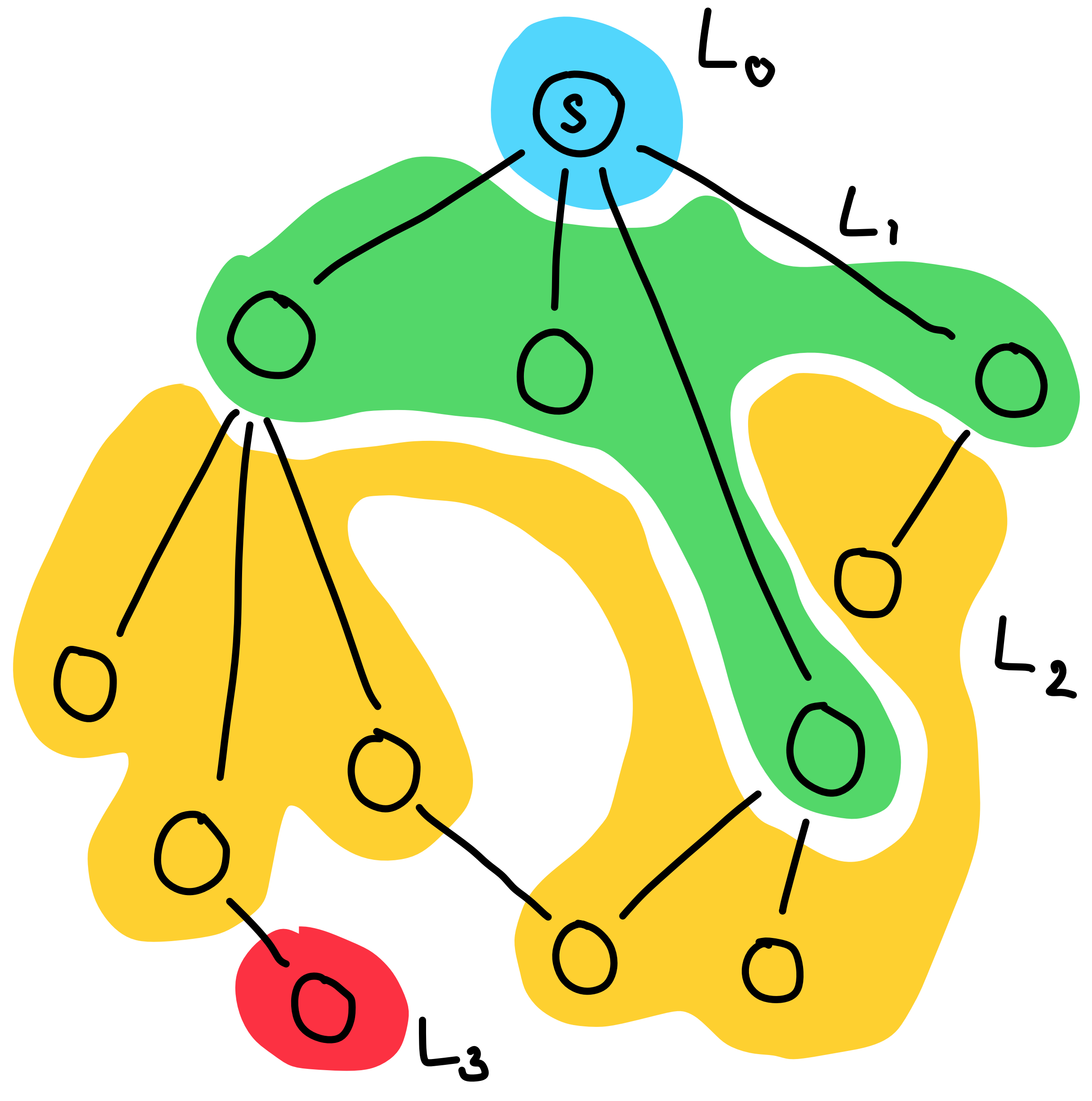


Bipartiteness testing

- **Claim:** A graph is bipartite iff it contains no *odd cycles*.
- **Algorithm:**
  - Start BFS from some vertex s. Instead of marking vertices as visited or not, marked them as "red", "blue", or "not visited". Mark s as red and add s to queue Q.
  - Pop vertex u from queue Q.
    - Check all neighbors v of u and make sure they are either "not visited" or the opposite color of u.
    - If not, abort and output "not bipartite".
    - If so, add the "not visited" neighbors v to the queue Q and color them with opposite color.
  - If queue Q is empty, output coloring generated.
- **Runtime:** Same as BFS, $O(n + m)$.

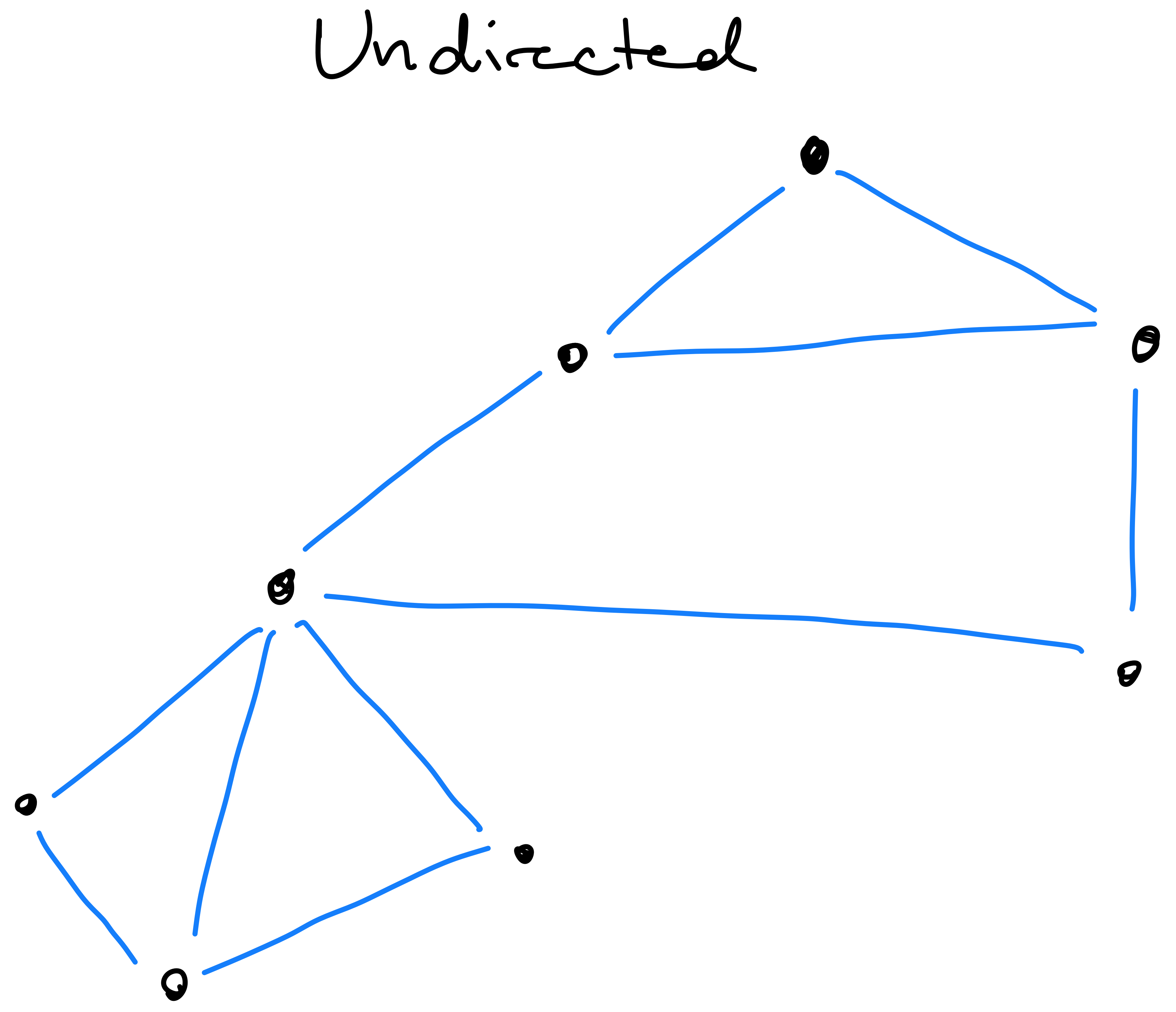


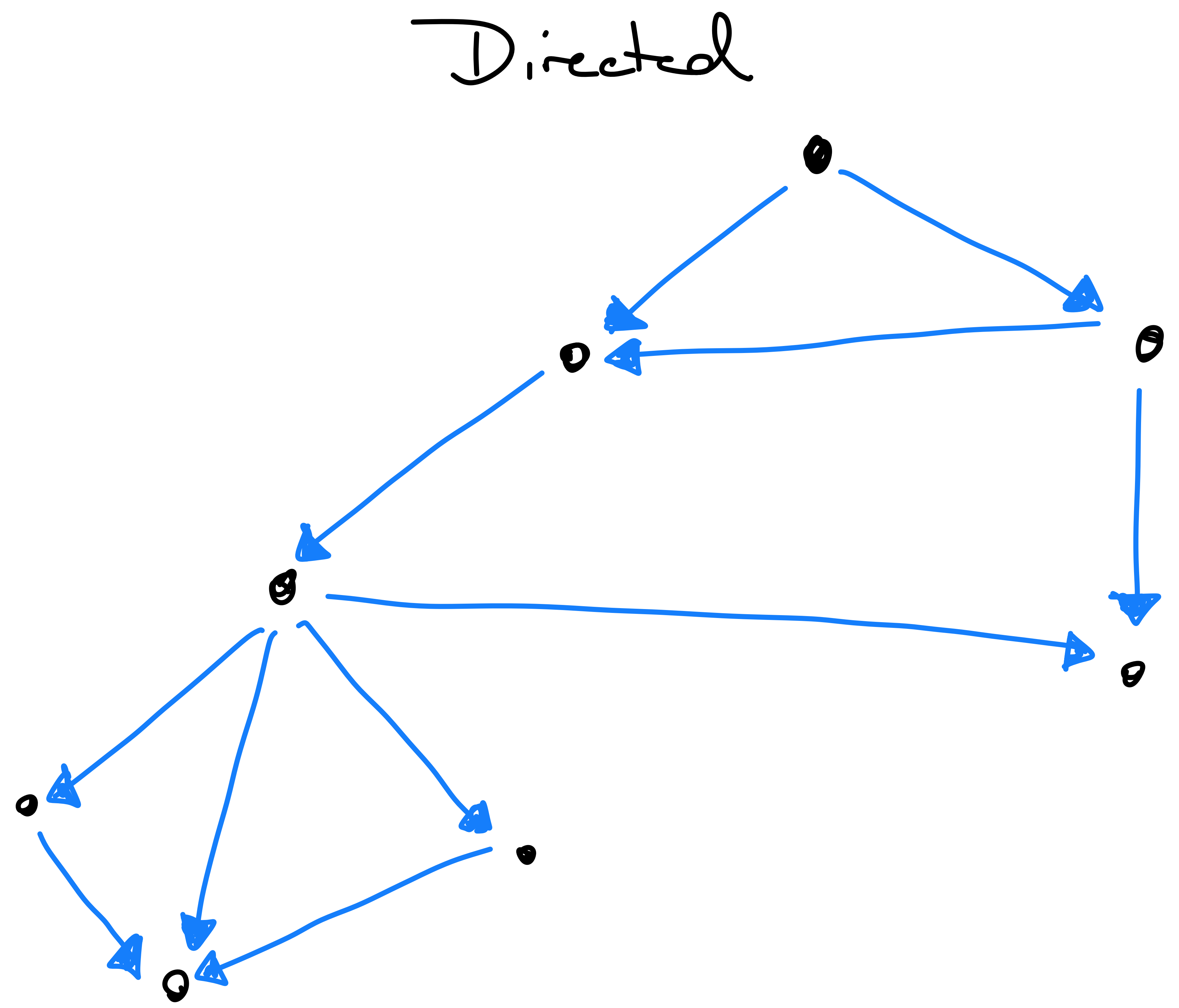
BFS edge property



- The BFS algorithm generates a tree T starting from root s.
- Let layer $L_i \subseteq V$ be the set of vertices distance i from s in T.
- **Claim:** The edges E only occur between adjacent layers or the same layer.
- **Proof**: If there is an edge $(u, v) \in L_i \times L_{\geq i+2}$, then v should have been in L_{i+1} because it was added to the queue after u was analyzed.
- Therefore, "bad edges" for bipartite testing only occur within the same layer. This finds an odd cycle.

Directed graphs

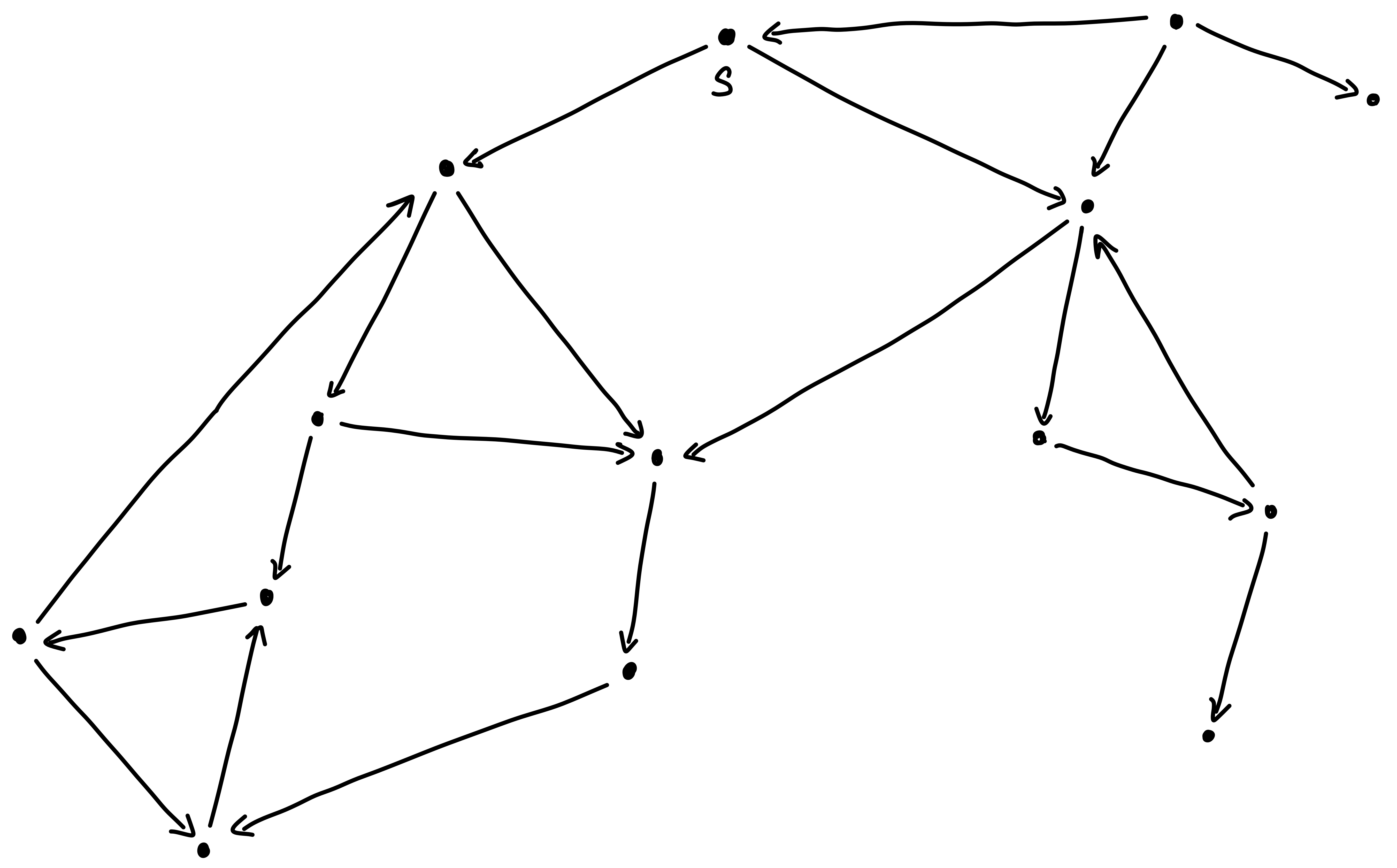




Depth-first search on directed graphs

- Same as DFS on undirected graphs except we only add neighbor v if an edge points from $u \to v$.

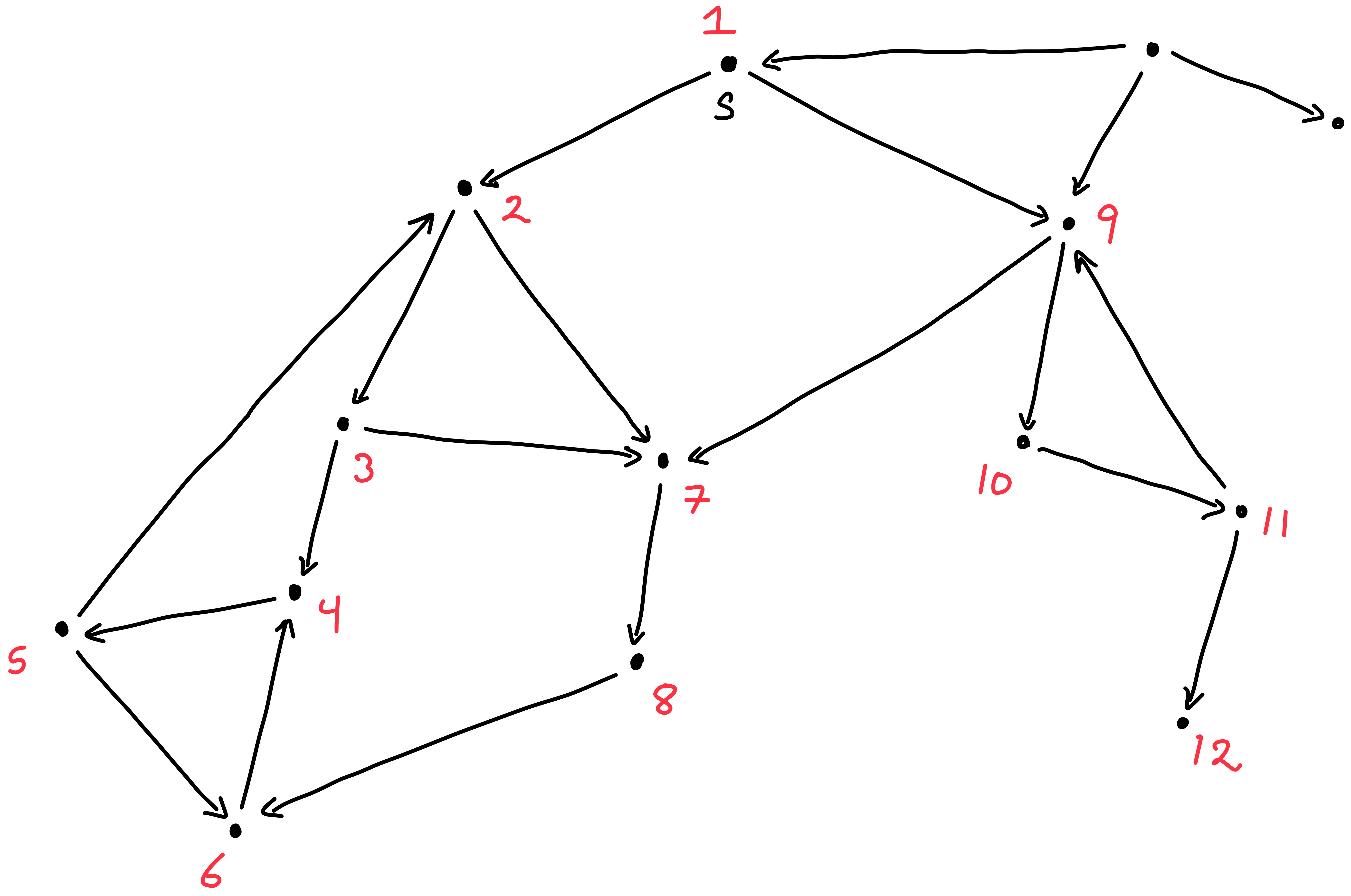
- DFS starting from s will visit all vertices u reachable by a *directed* path $s \rightsquigarrow u$.



Depth-first search on directed graphs

- Same as DFS on undirected graphs except we only add neighbor v if an edge points from $u \to v$.

- DFS starting from s will visit all vertices u reachable by a *directed* path $s \rightsquigarrow u$.



Depth-first search on directed graphs

- Same as DFS on undirected graphs except we only add neighbor v if an edge points from $u \to v$.
- DFS starting from s will visit all vertices u reachable by a *directed* path $s \rightsquigarrow u$.