Lecture 4

Breadth- and depth-first search, topological sort

Chinmay Nirkhe | CSE 421 Spring 2025

Graph search and traversal

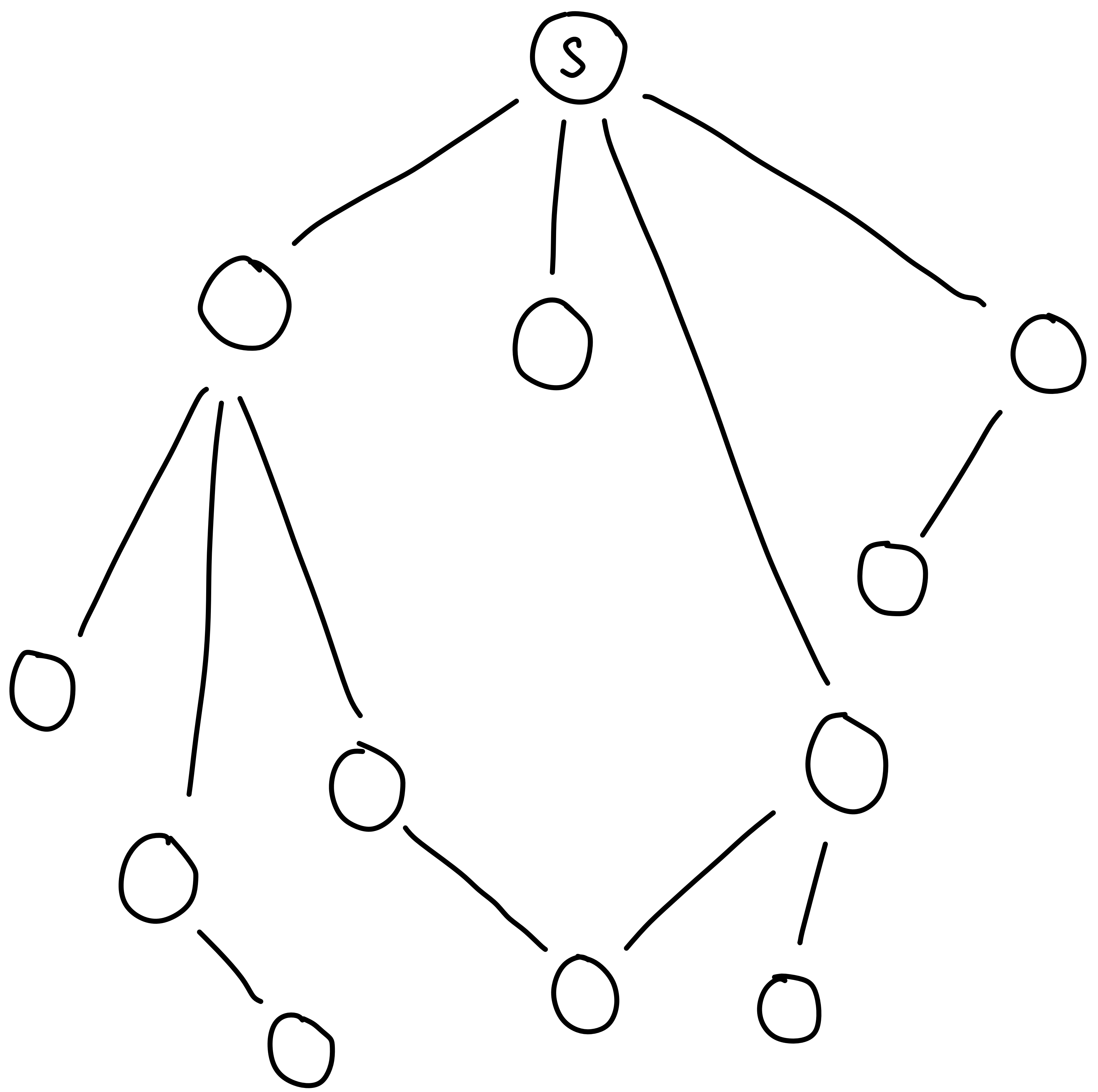
- Used to discover the structure of a graph
- "Walk" from a fixed starting vertex s ("the source") to find all the vertices reachable from s
- **Generic traversal algorithm.**
  - **Input:** Graph G and vertex $s \in V$
  - **Find:** set $R \subseteq V$ reachable from s

Reachable(s):

$R \leftarrow \{s\}$
While there exists a $(u, v) \in R \times (V \backslash R)$
    Add v to R: $R \leftarrow R \cup \{v\}$.
return R

Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.

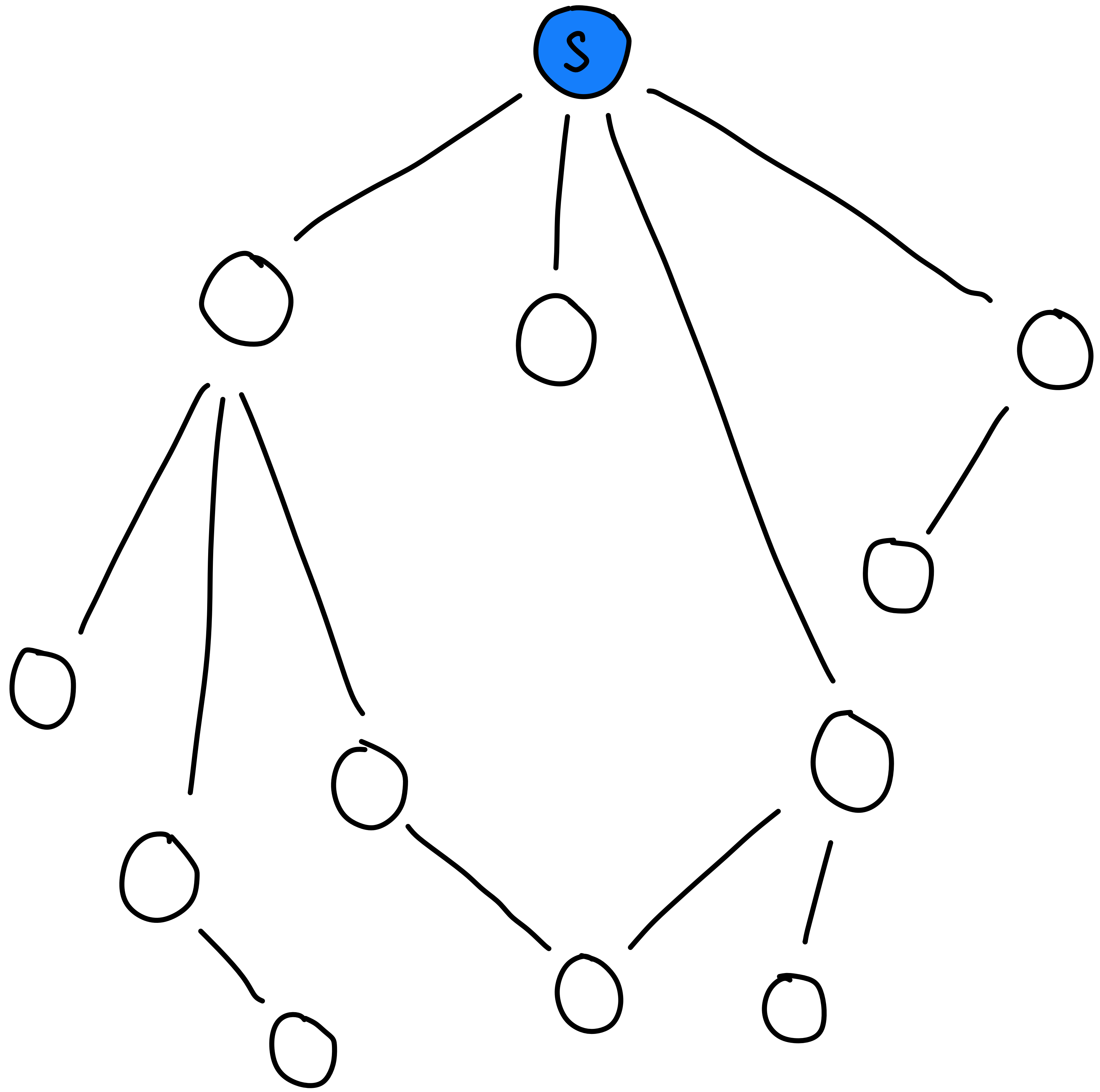


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.

queue Q

s

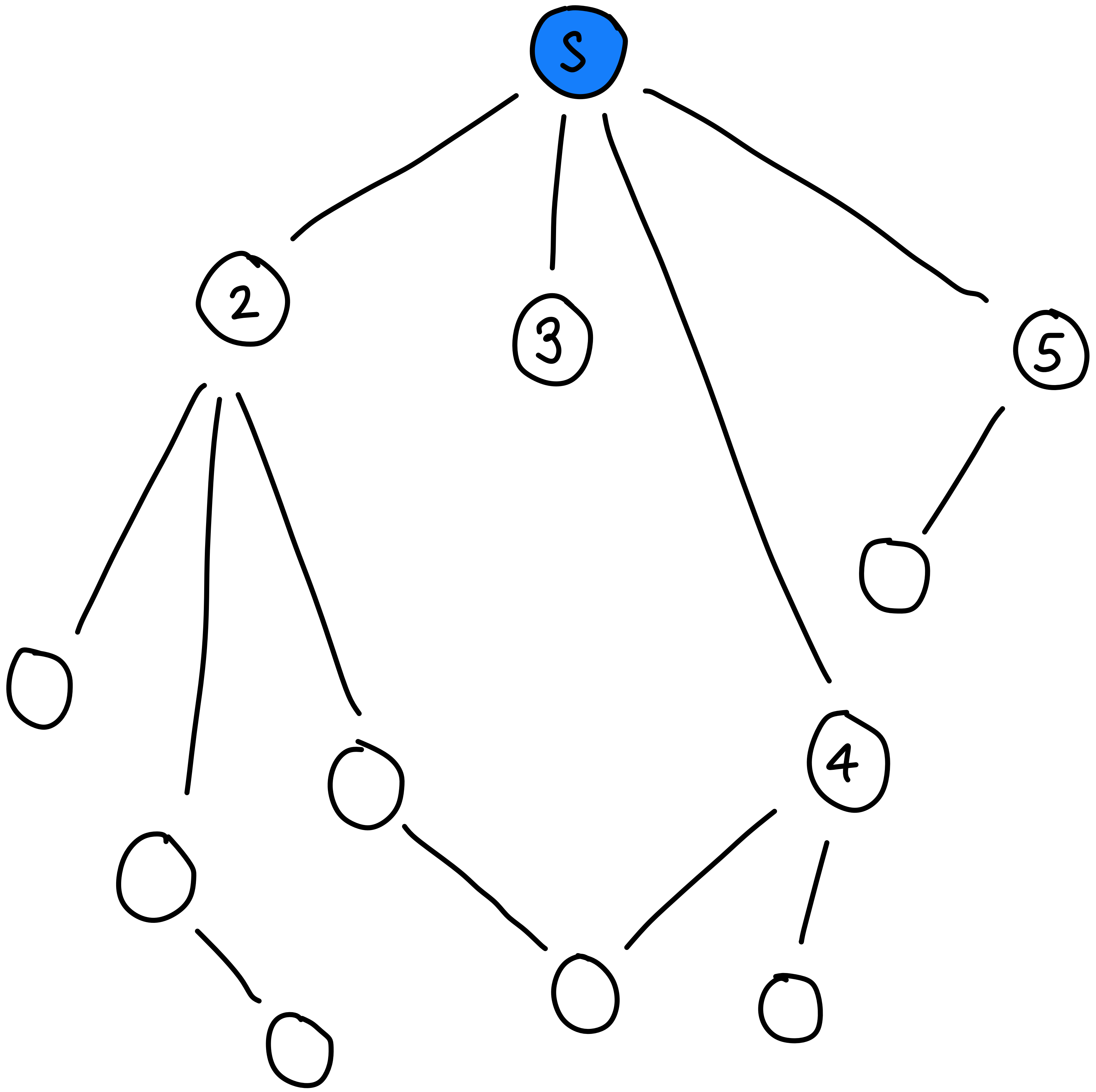


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
2
3
4
5

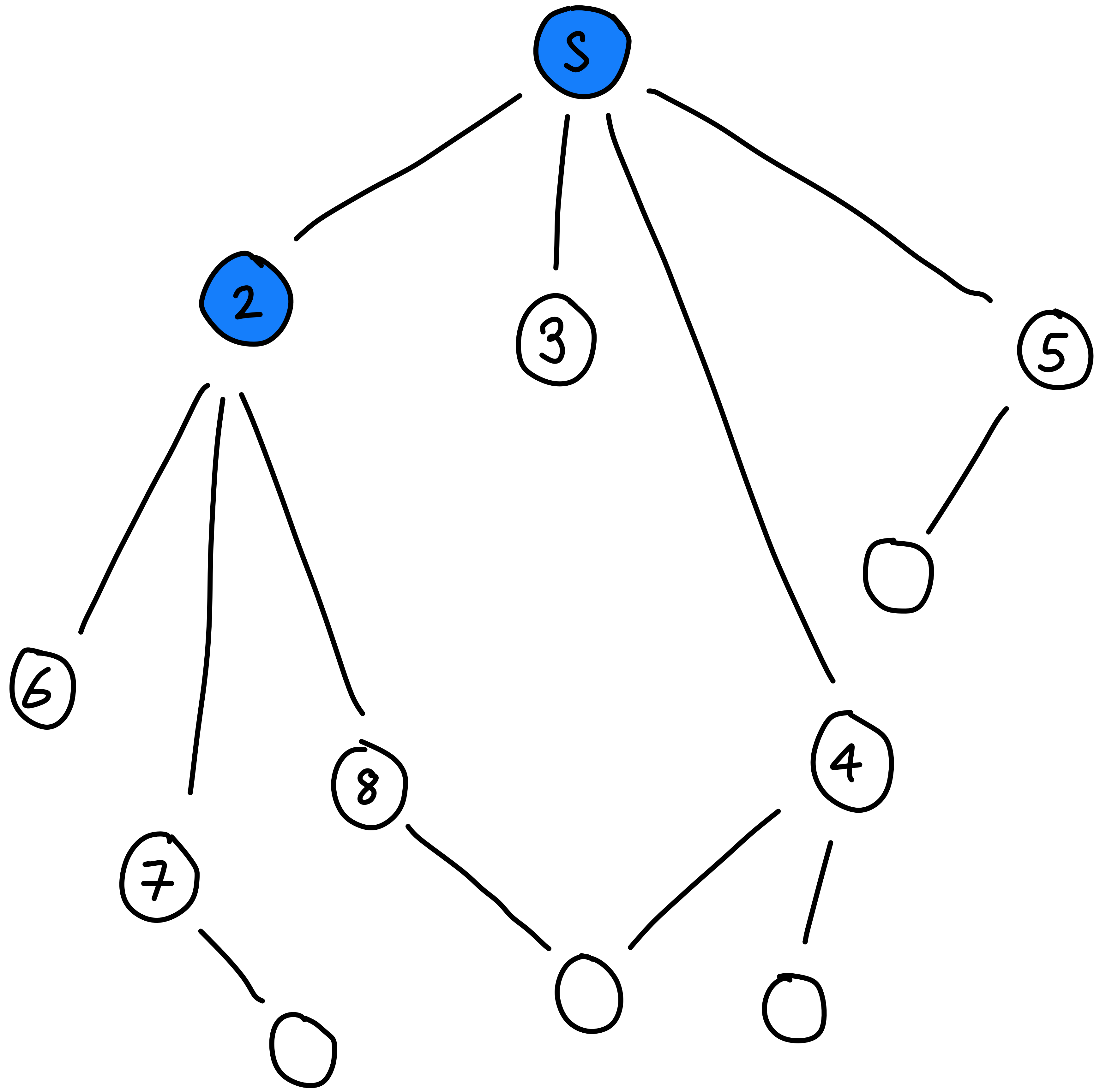


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
3
4
5
6
7
8

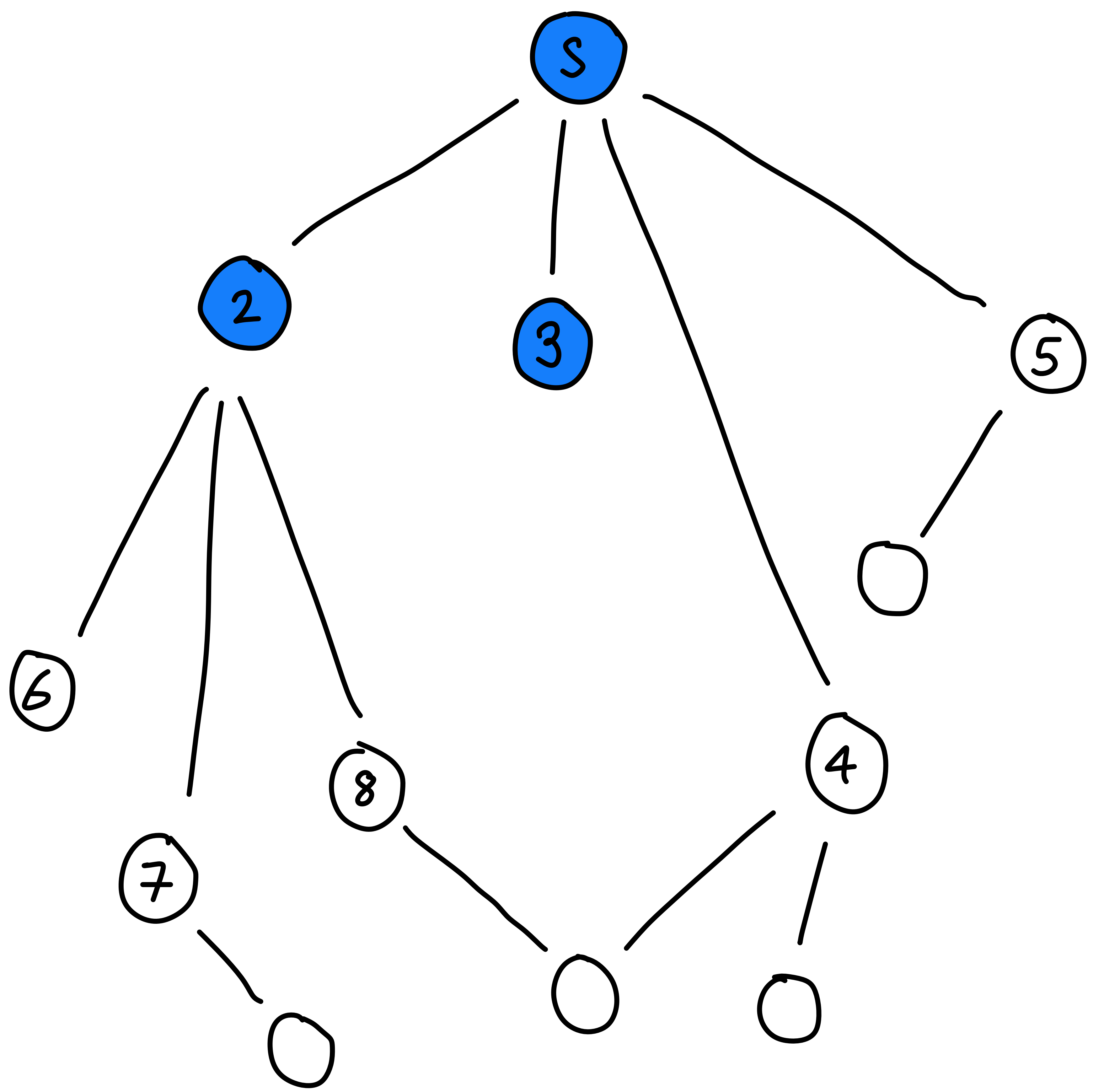


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
4
5
6
7
8

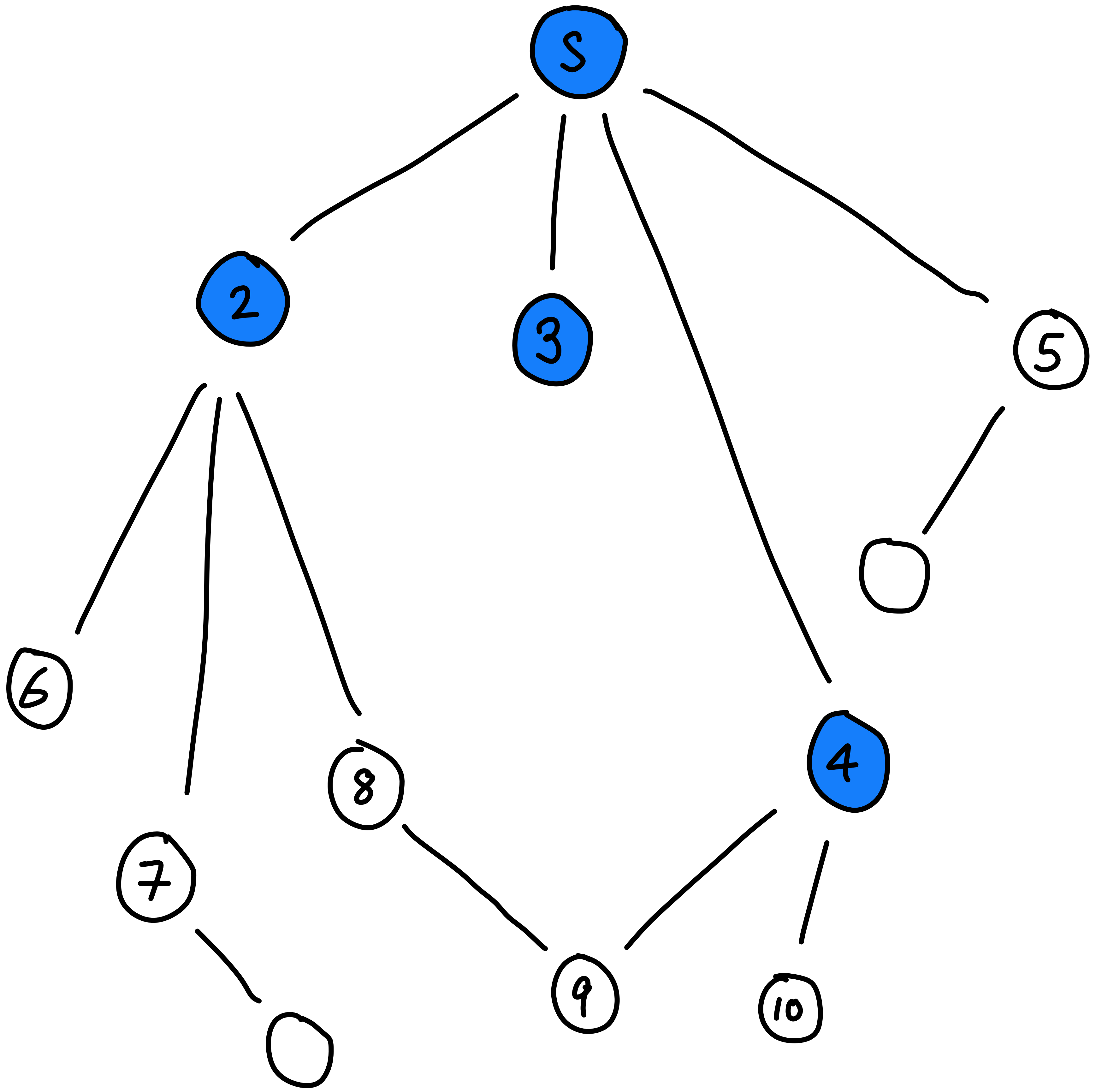


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
5
6
7
8
9
10

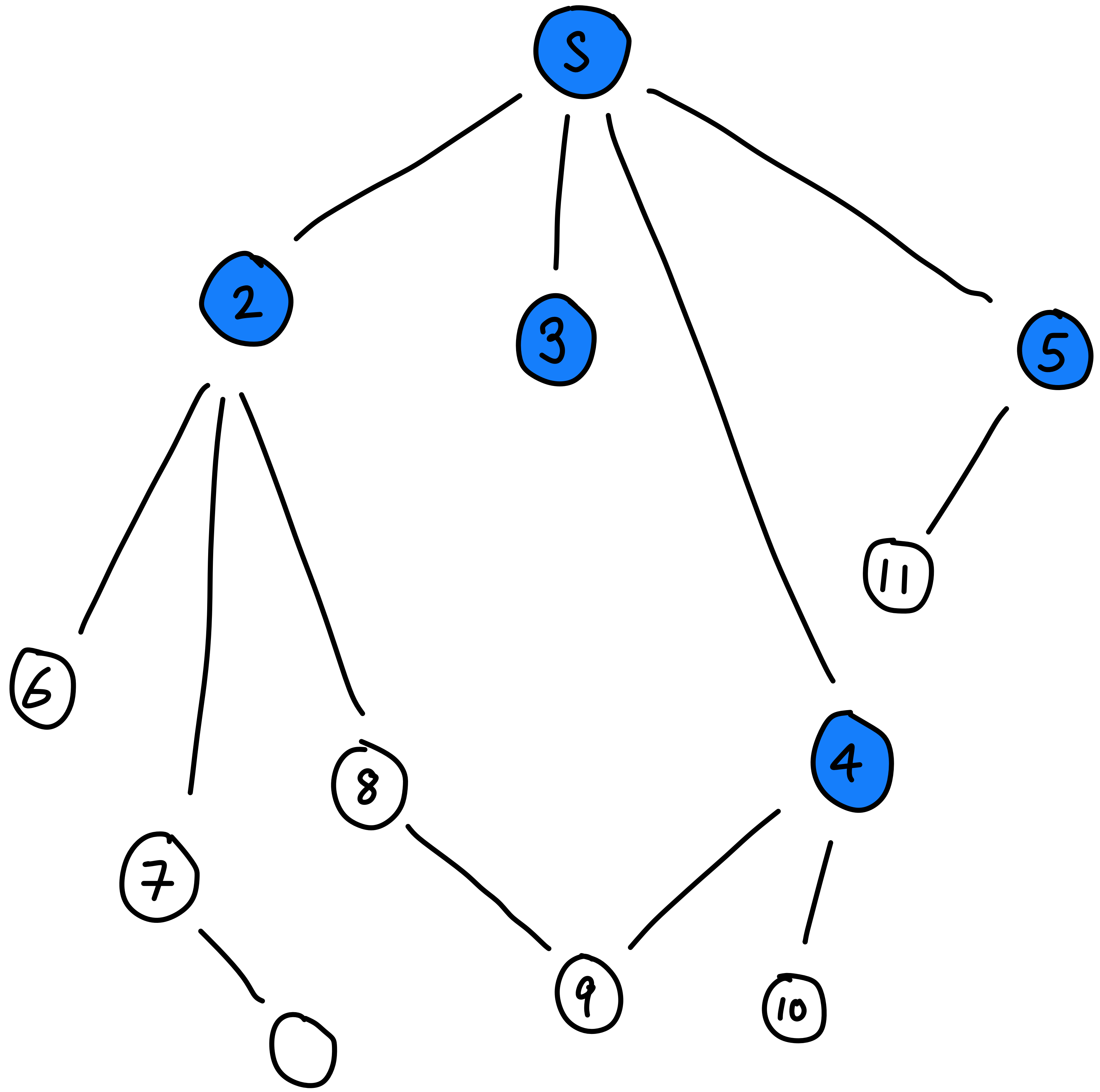


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
6
7
8
9
10
11

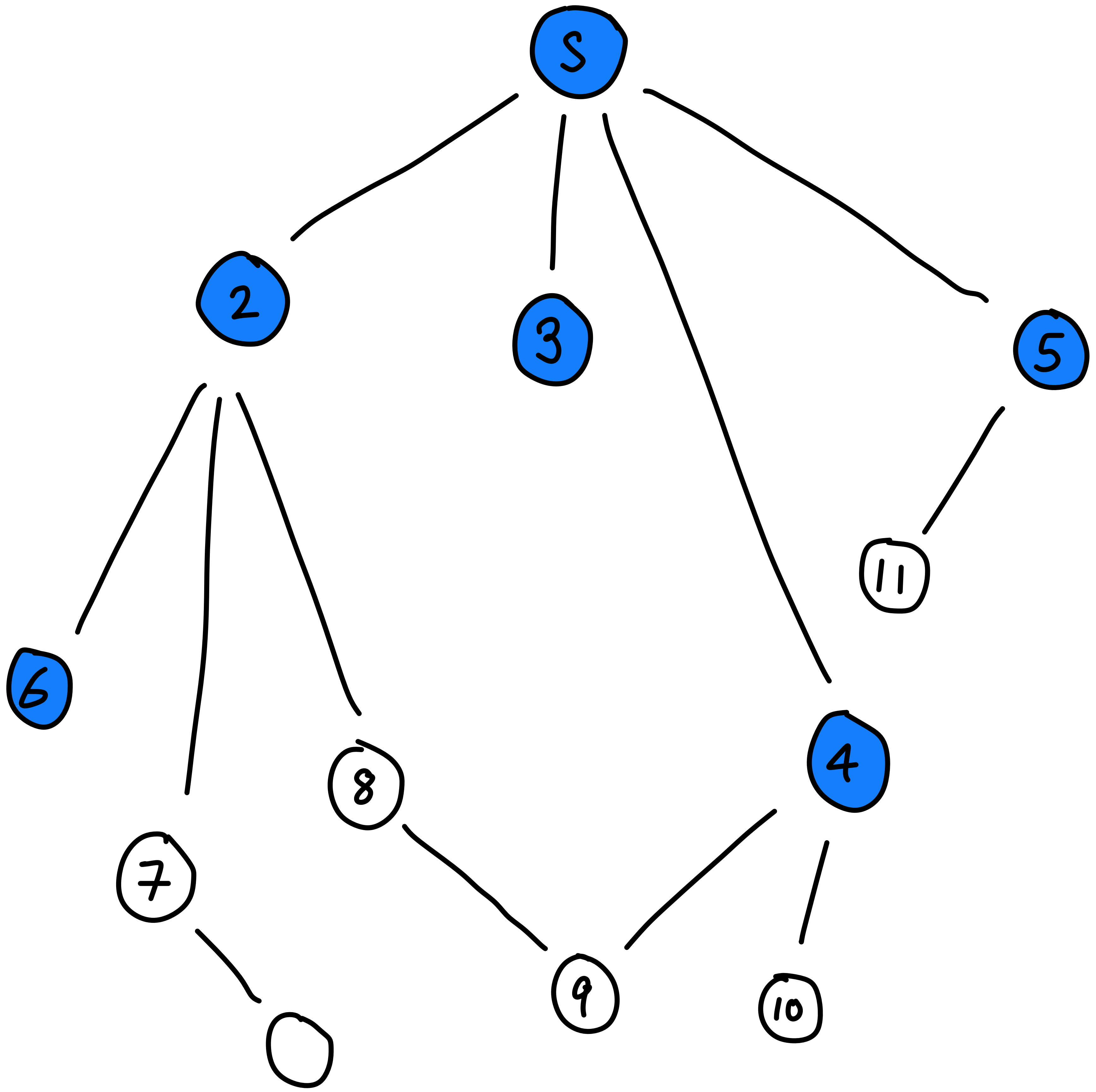


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
7
8
9
10
11

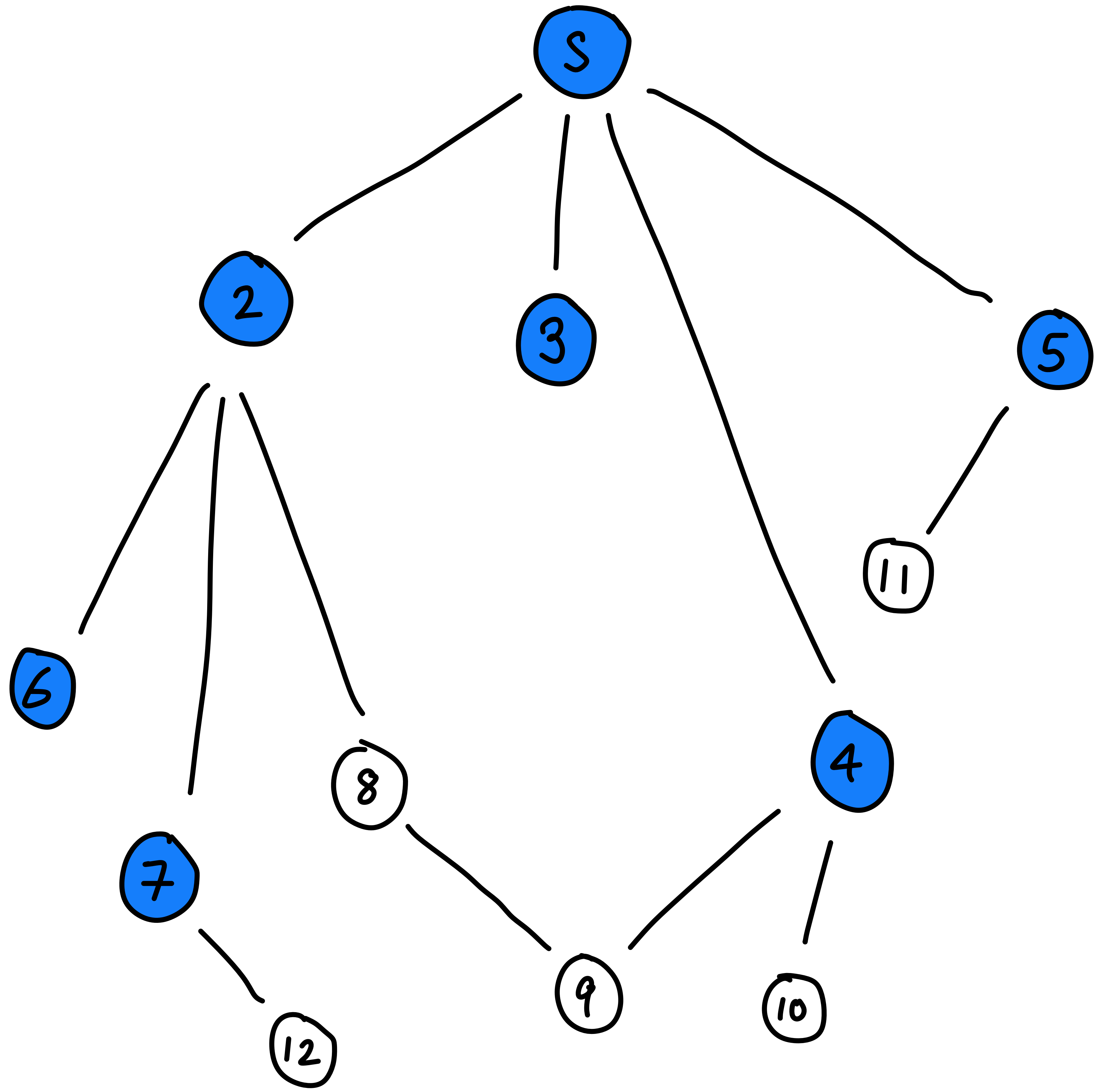


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
8
9
10
11
12

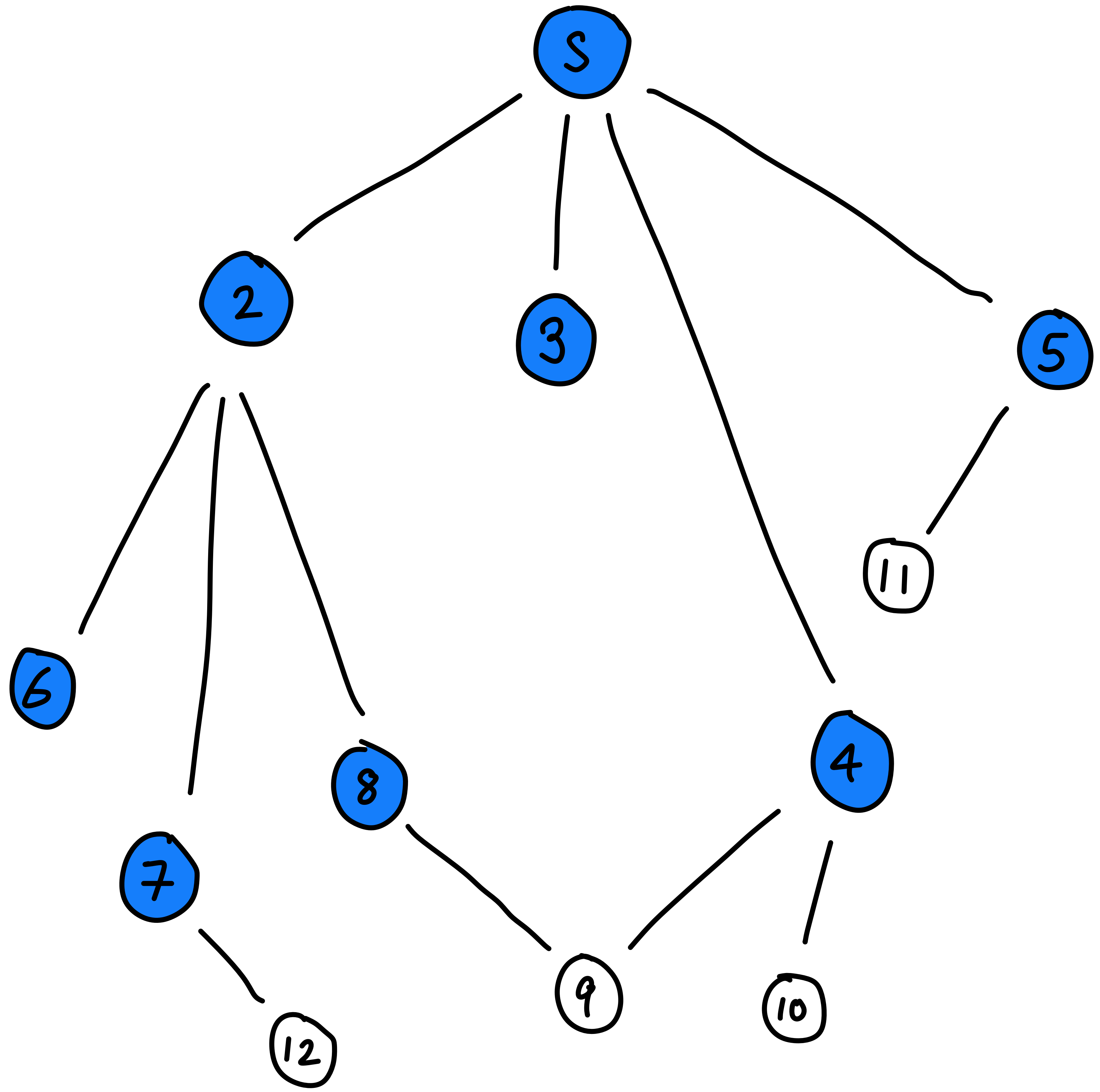


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
~~8~~
9
10
11
12

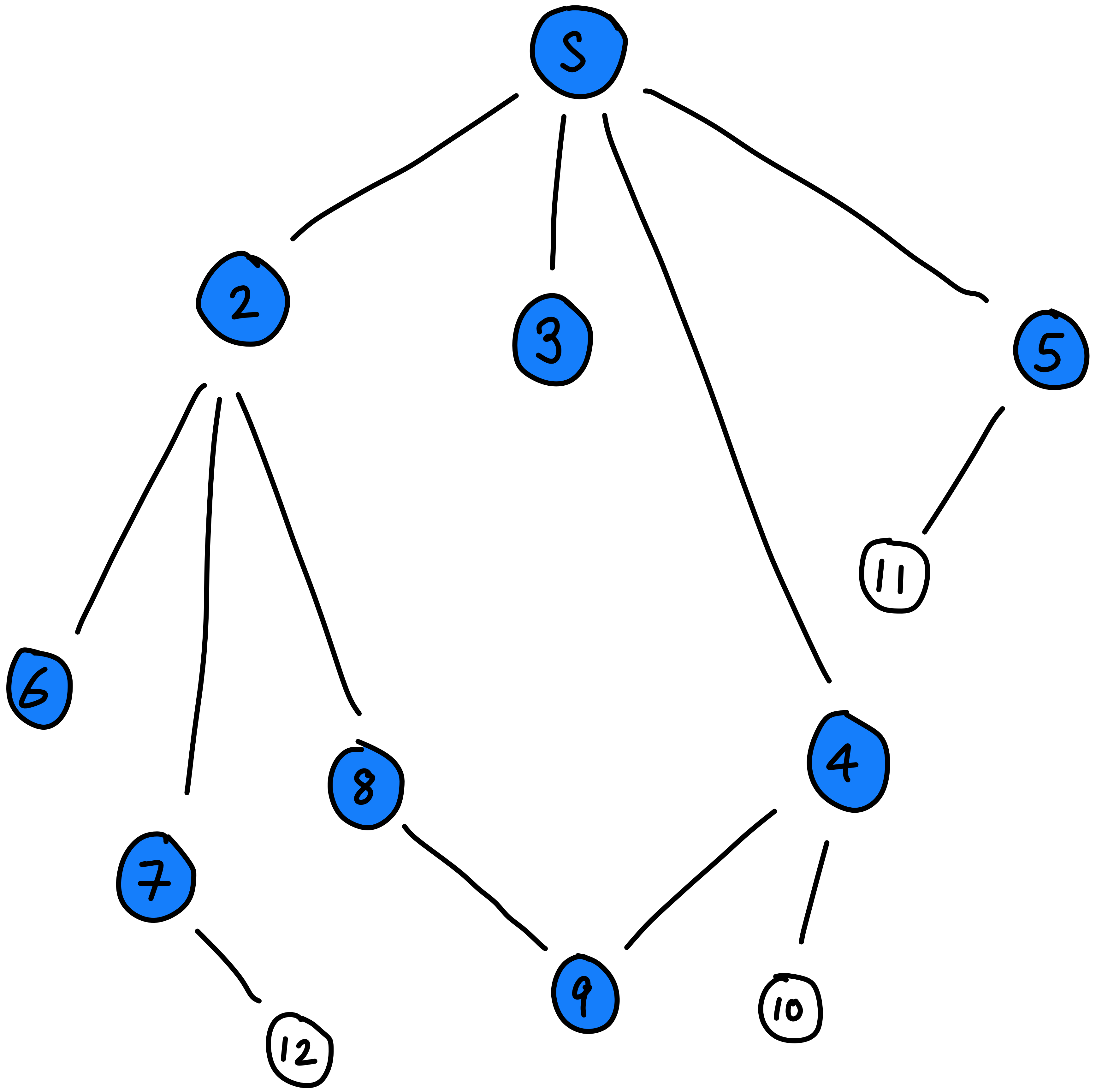


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
~~8~~
~~9~~
10
11
12

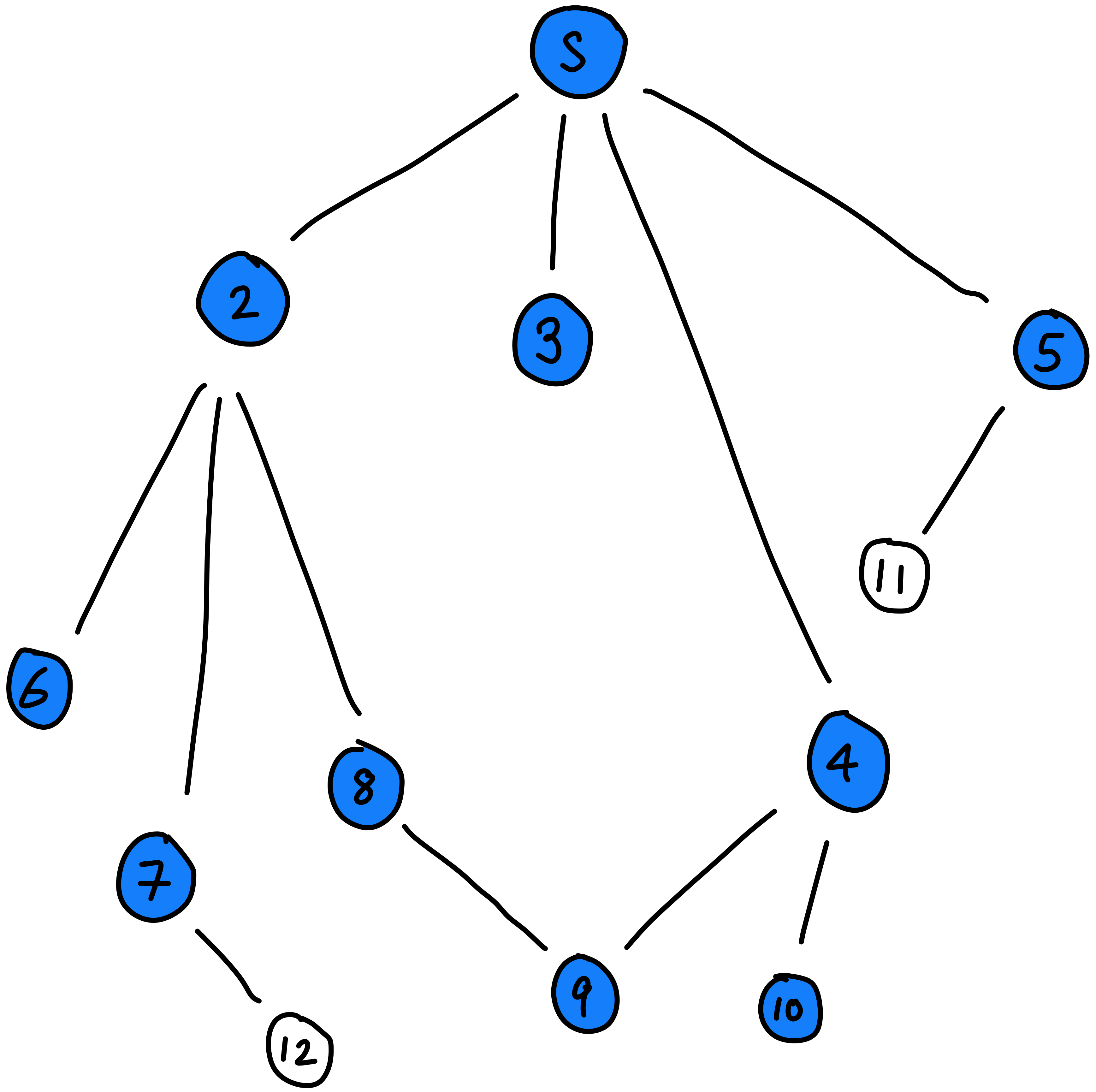


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
~~8~~
~~9~~
~~10~~
11
12

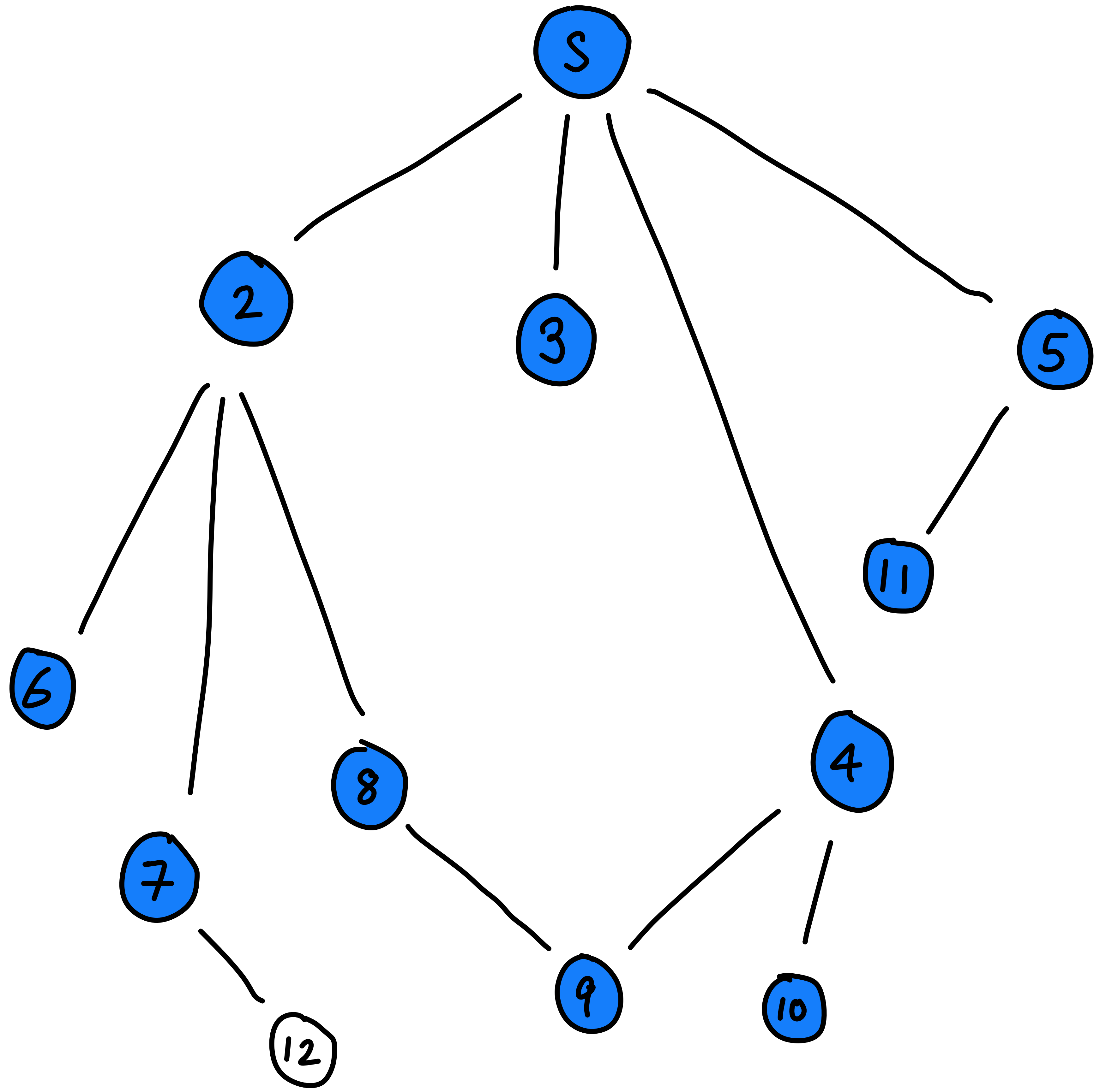


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.

queue Q

~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
~~8~~
~~9~~
~~10~~
~~11~~
12

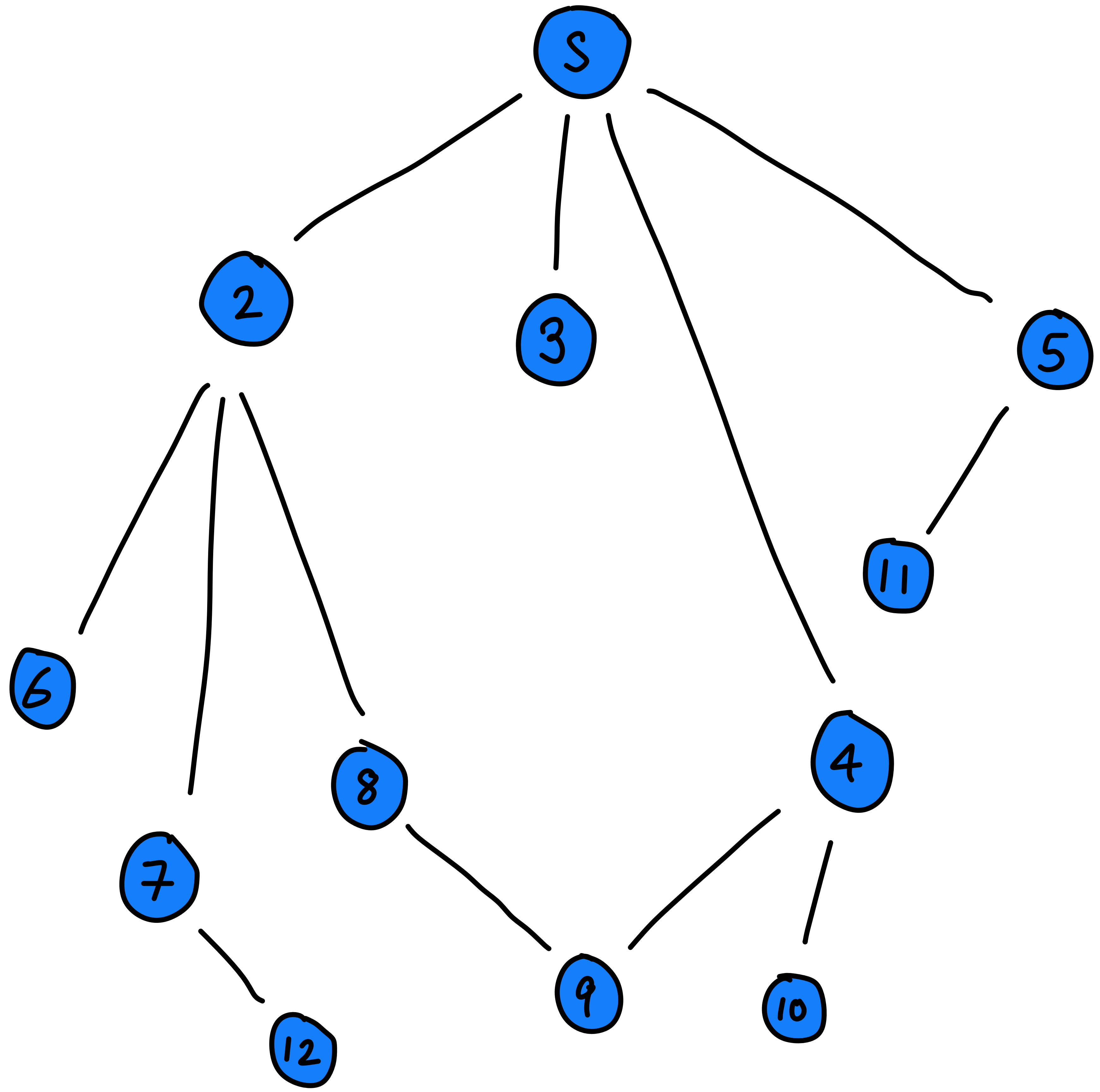


Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.

queue Q

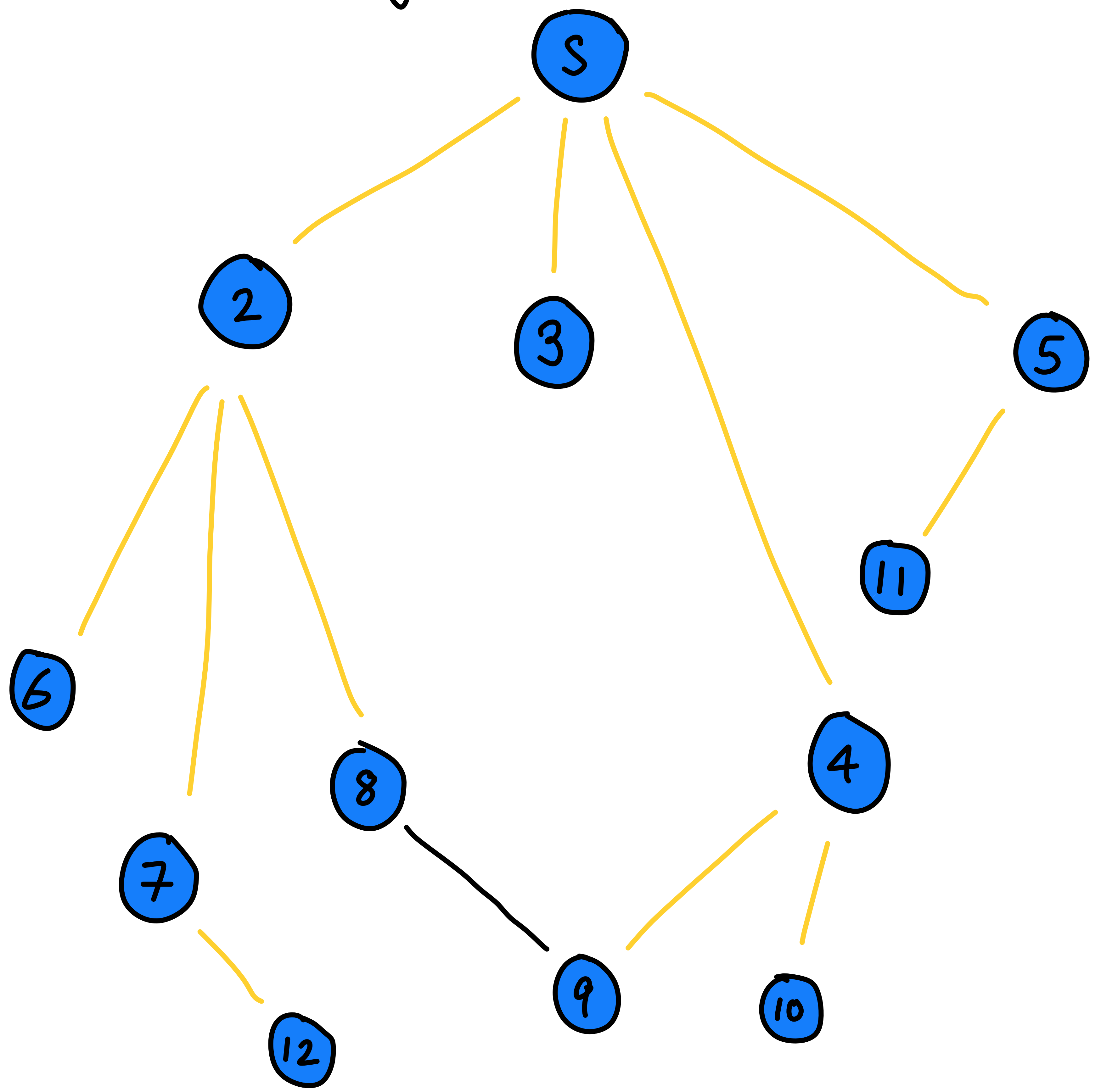
~~s~~
~~2~~
~~3~~
~~4~~
~~5~~
~~6~~
~~7~~
~~8~~
~~9~~
~~10~~
~~11~~
~~12~~



Breadth-first search (BFS)

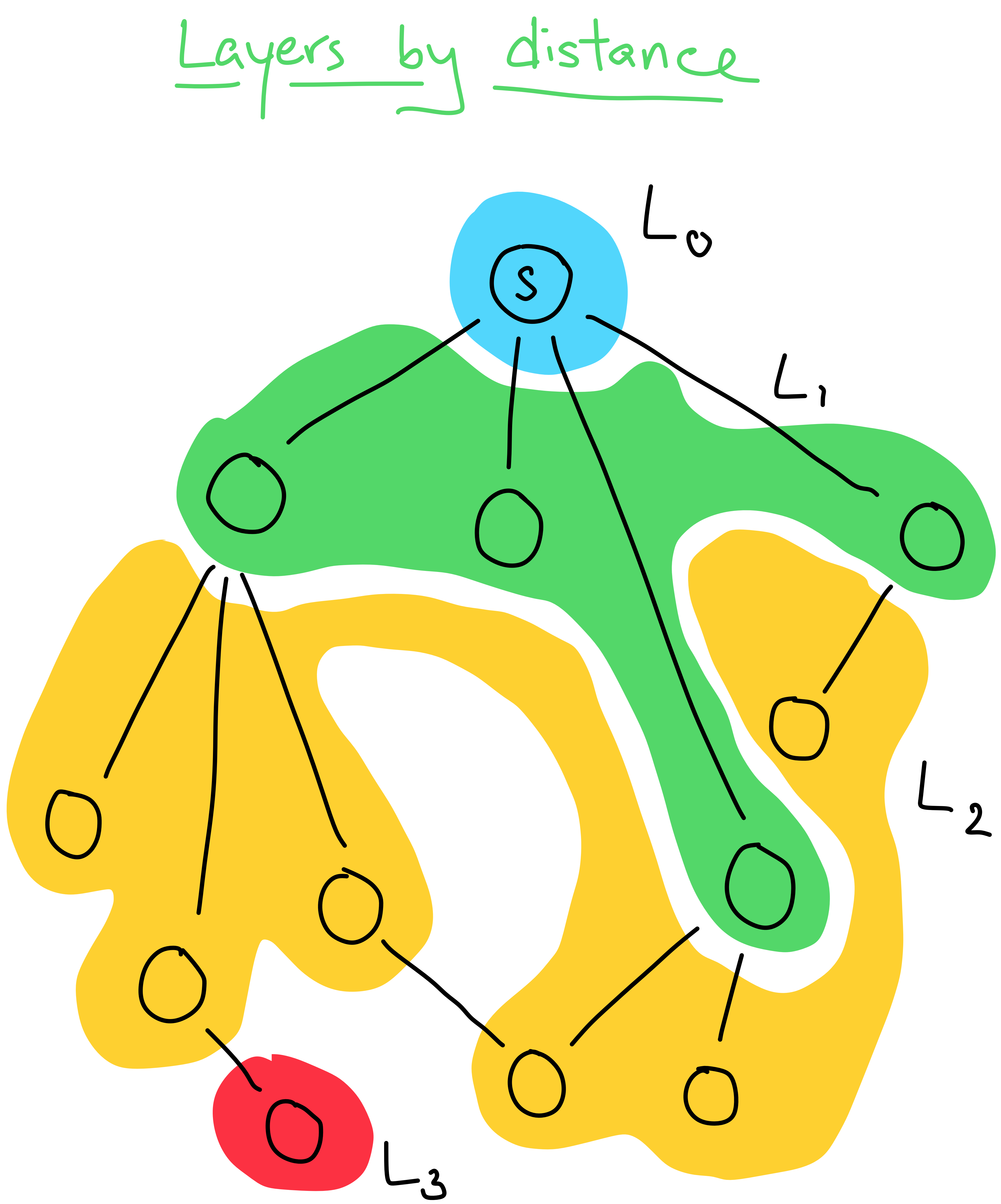
- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- Assign a bit to every vertex as visited/not visited.
- **Algorithm:**
  - Initialize set $R \leftarrow \{s\}$ and queue $Q \leftarrow \{s\}$.
  - Set all vertices to not visited. 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 $R \leftarrow R \cup \{u\}$.

BFS tree generated by tracking which edges are used



Breadth-first search (BFS)

- Used to explore the vertices in R according to their distance from s.
- Implemented using the *queue* data structure.
- 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 $R \leftarrow R \cup \{u\}$.



Graph search and traversal

- Used to discover the structure of a graph
- "Walk" from a fixed starting vertex s ("the source") to find all the vertices reachable from s
- **Generic traversal algorithm.**
  - **Input:** Graph G and vertex $s \in V$
  - **Find:** set $R \subseteq V$ reachable from s

Reachable(s):

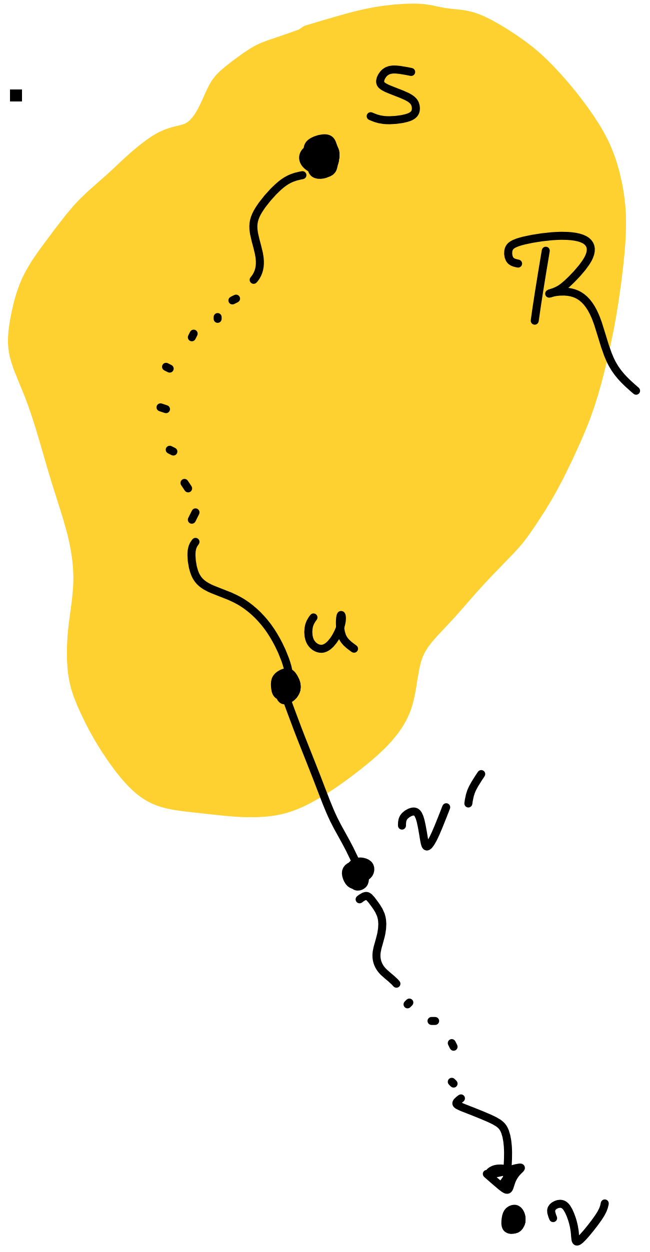
$R \leftarrow \{s\}$
While there exists a $(u, v) \in R \times (V \backslash R)$
  Add v to R: $R \leftarrow R \cup \{v\}$.
return R

Reachable(s):

$R \leftarrow \{s\}$
While there exists a $(u, v) \in R \times (V \backslash R)$
    Add v to R: $R \leftarrow R \cup \{v\}$.
return R

Generic graph traversa

- **Claim:** R is exactly the set of reachable vertices.
- **Proof:** We show both directions. (1): every vertex in R is reachable. (2): every reachable is in R.
    - **Direction 1**. For $v \in R$, there is a path $s \rightsquigarrow v$. Proved by induction on the generic graph traversal algorithm: If we added v by edge $(u, v) \in R \times (V \backslash R)$ then $s \rightsquigarrow u \rightarrow v$.
    - **Direction 2**. Assume (for $\perp$), there is a vertex v that is reachable but not $v \notin R$.
        - Let $p =$ the path $s \rightsquigarrow v$ and let v' be the **first** vertex on p such that $v' \notin R$.
        - Then u, the predecessor of v', satisfies $u \in R$ and $(u, v') \in R \times (V \backslash R)$.
        - Contradicts the definition of the generic graph traversal.



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 $p(u, v)$.

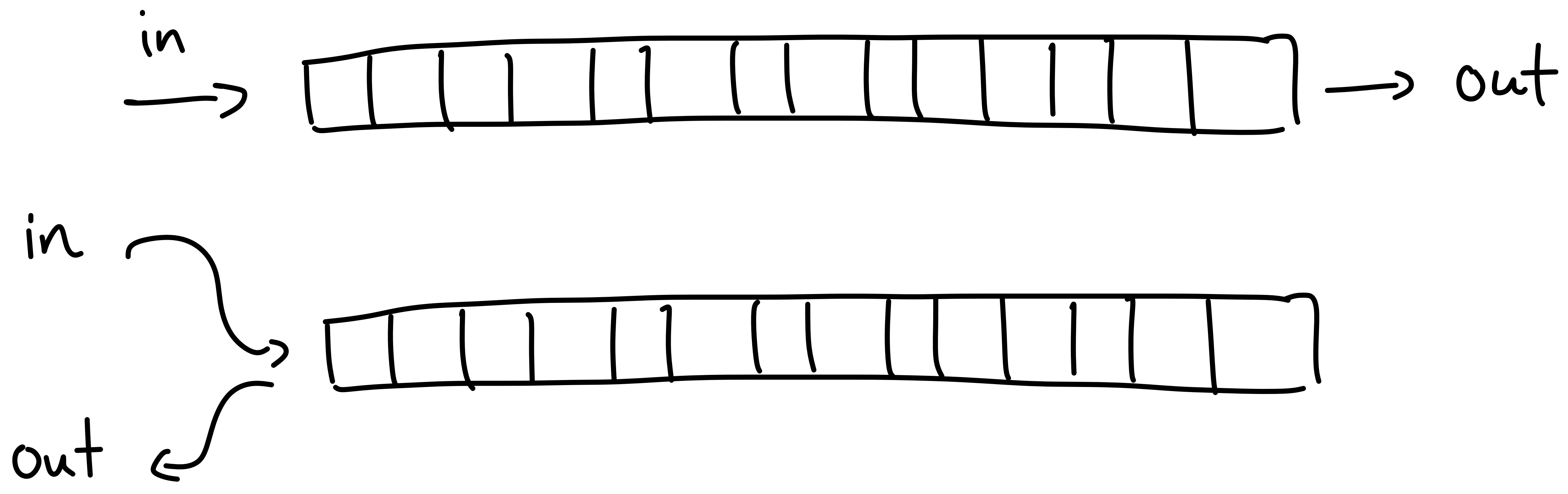
Faster when all pairs are being compared.

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.
- **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

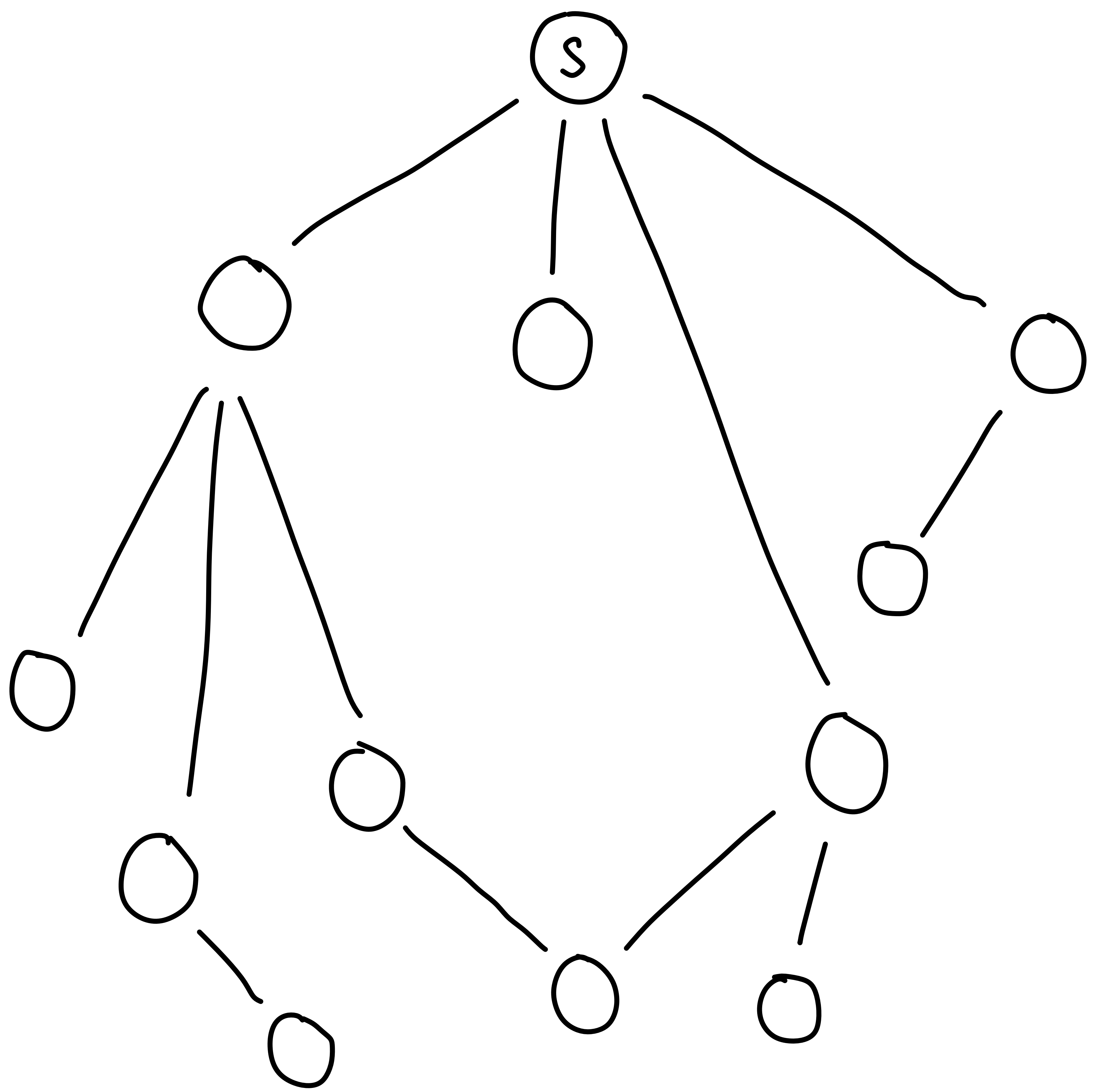
Depth-first search

- 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*



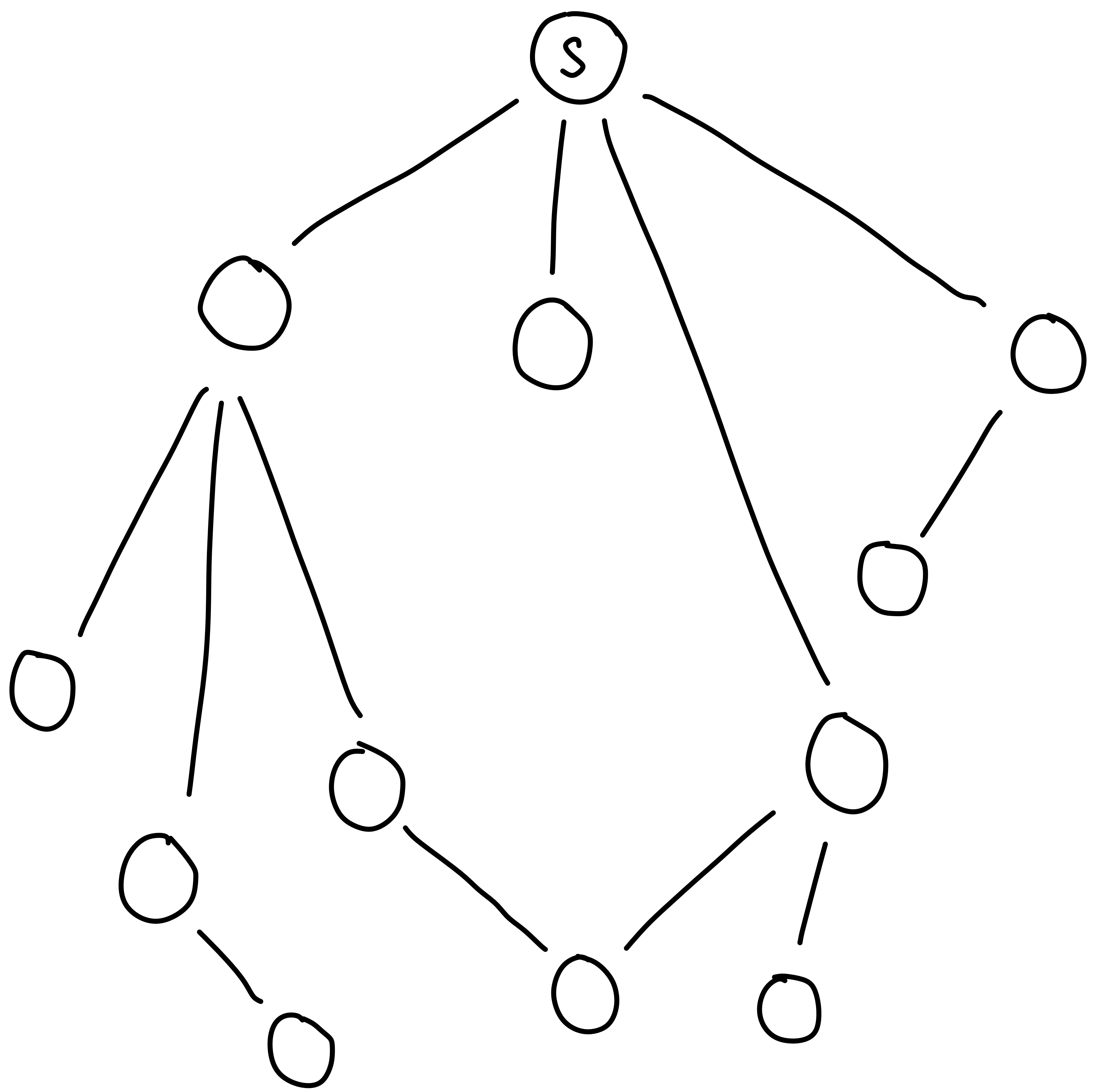
Breadth-first search (BFS)

- 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)

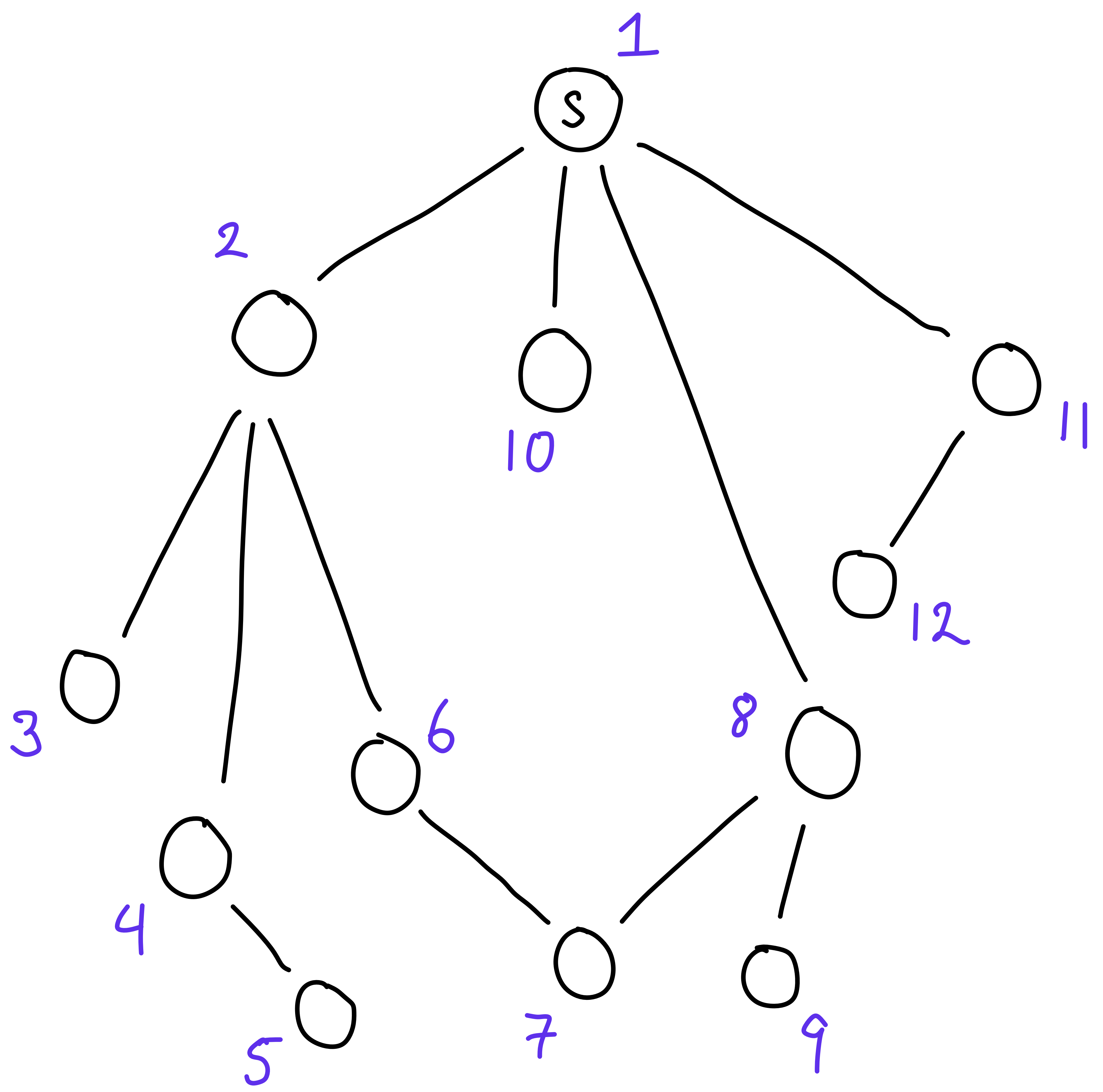
- Assign a bit to every vertex as visited/not visited.
- **Algorithm**:
  - Initialize set $R \leftarrow \{s\}$ and stack $S \leftarrow \{s\}$.
  - Set all vertices to not visited.
  - While S isn't empty, pop v off the stack.
    - If v is not visited, set v to visited
      - For every neighbor u of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



Depth-first search (BFS)

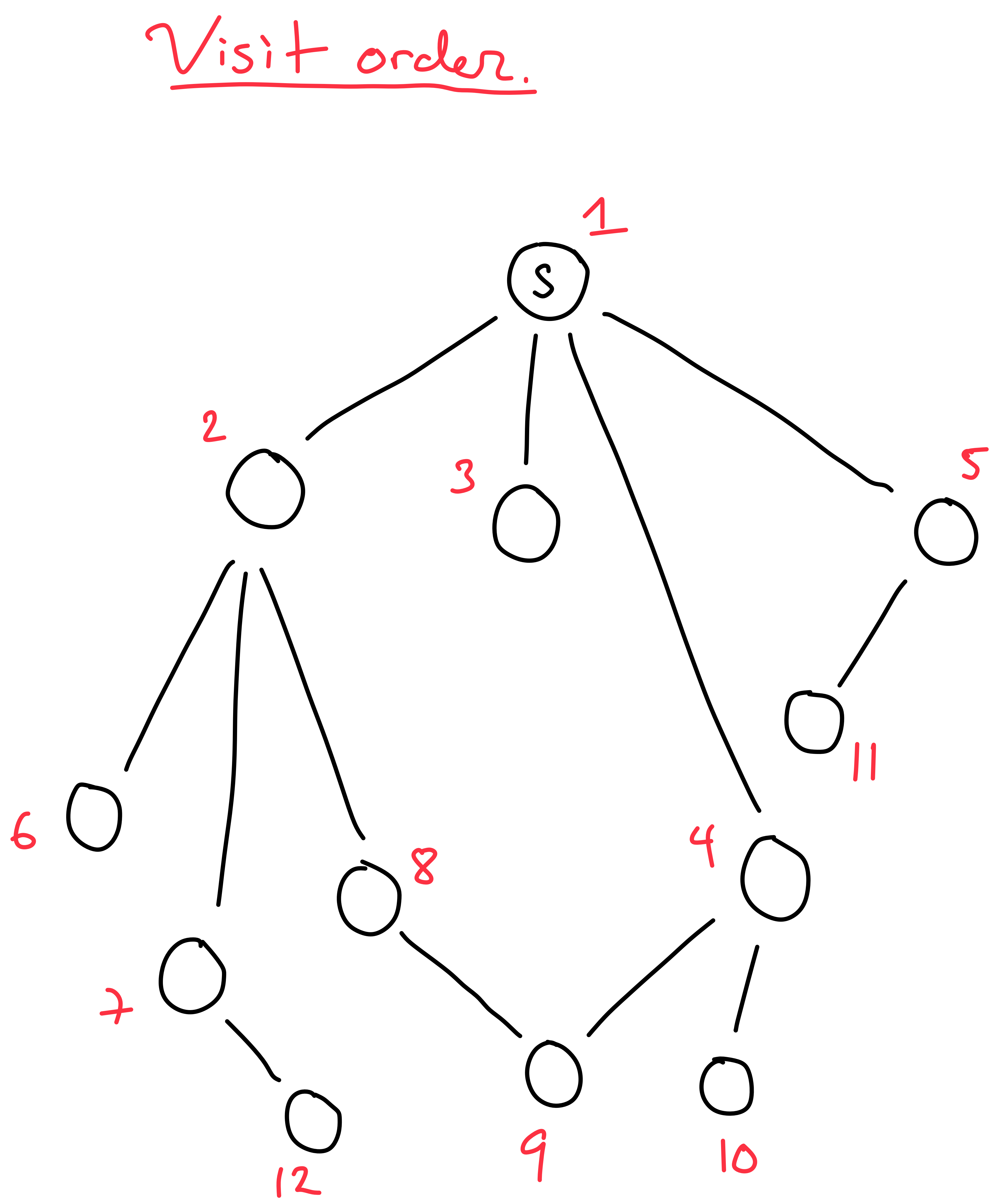
Visit order

- 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 $R \leftarrow R \cup \{u\}$.



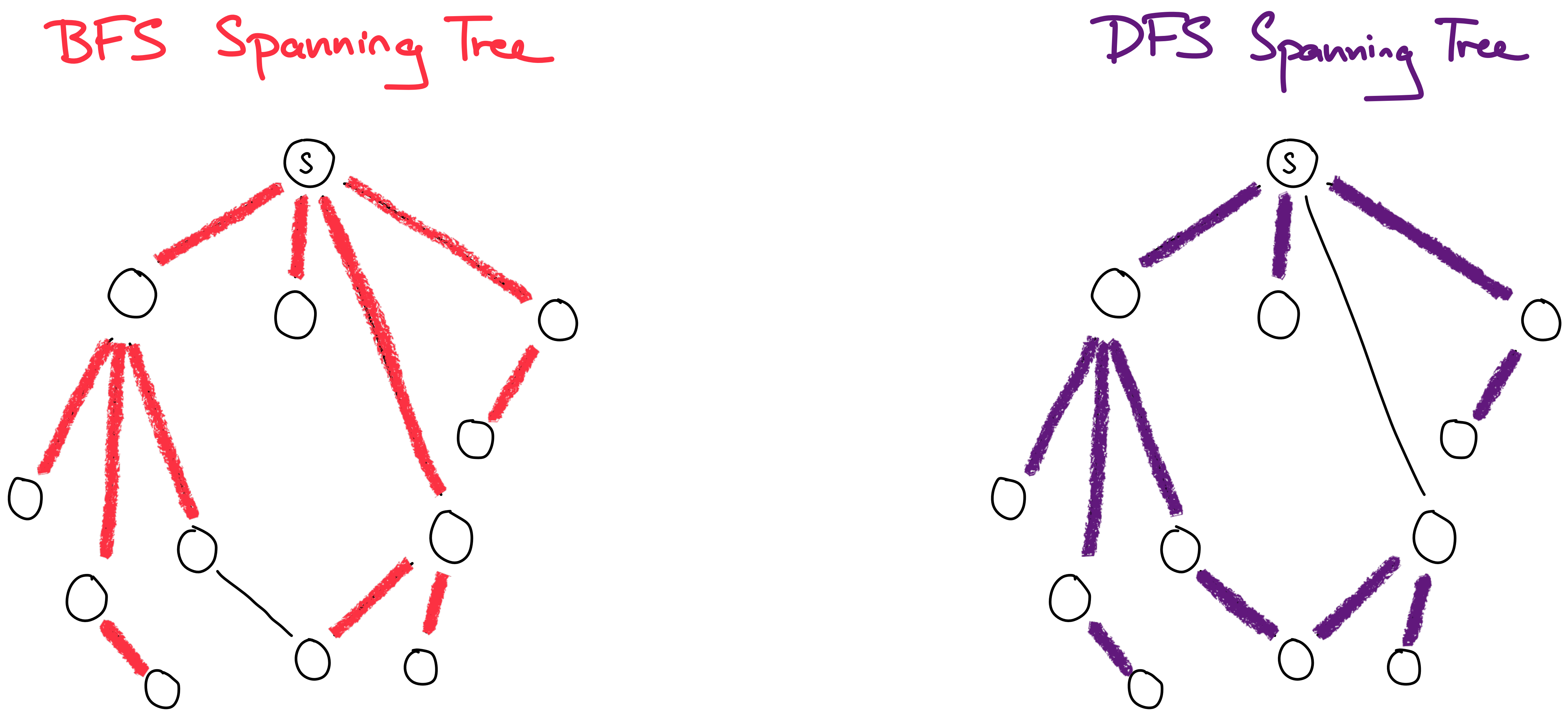
Breadth-first search (BFS)

- 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 $R \leftarrow R \cup \{u\}$.



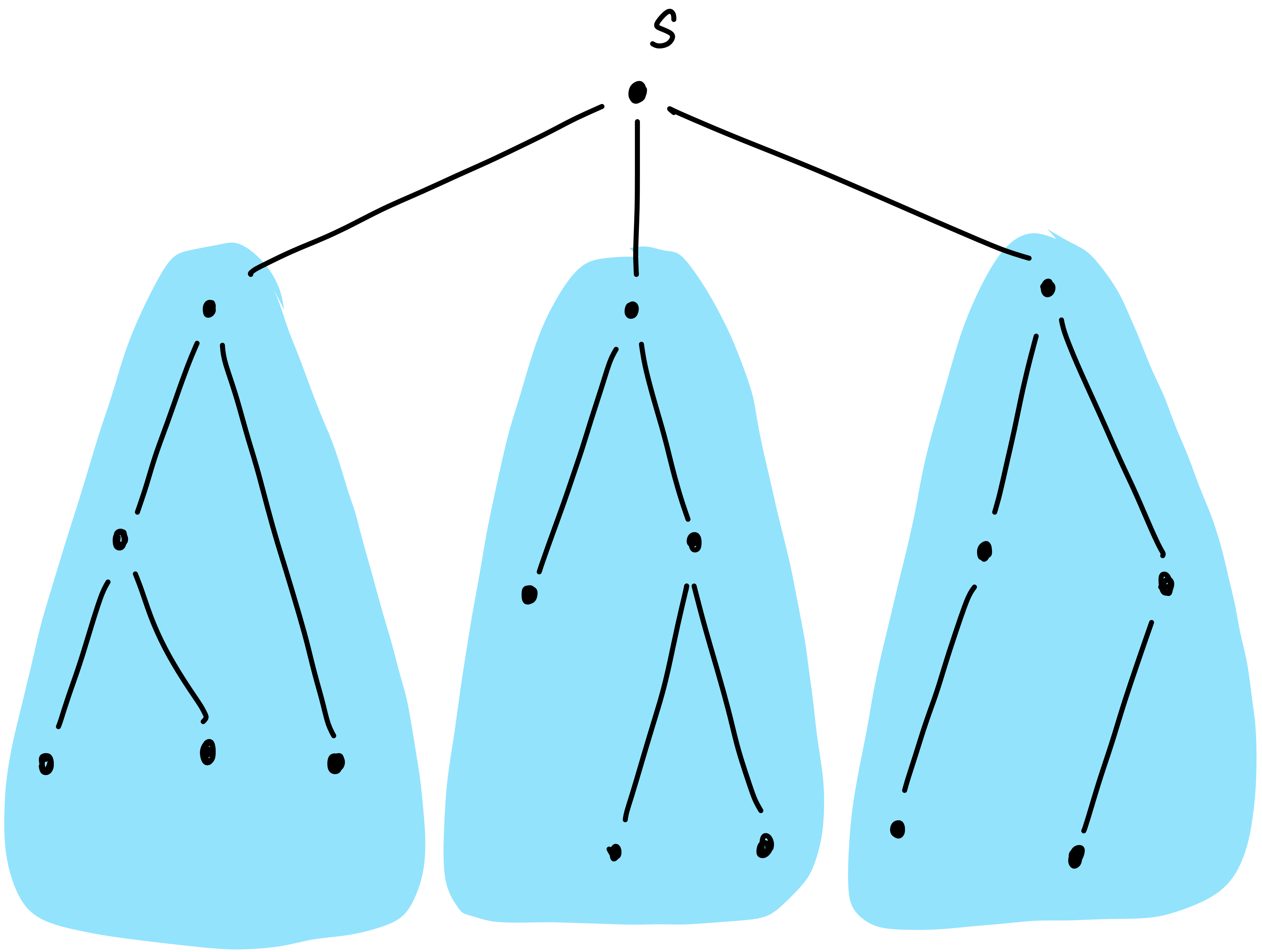
Spanning trees

- 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!



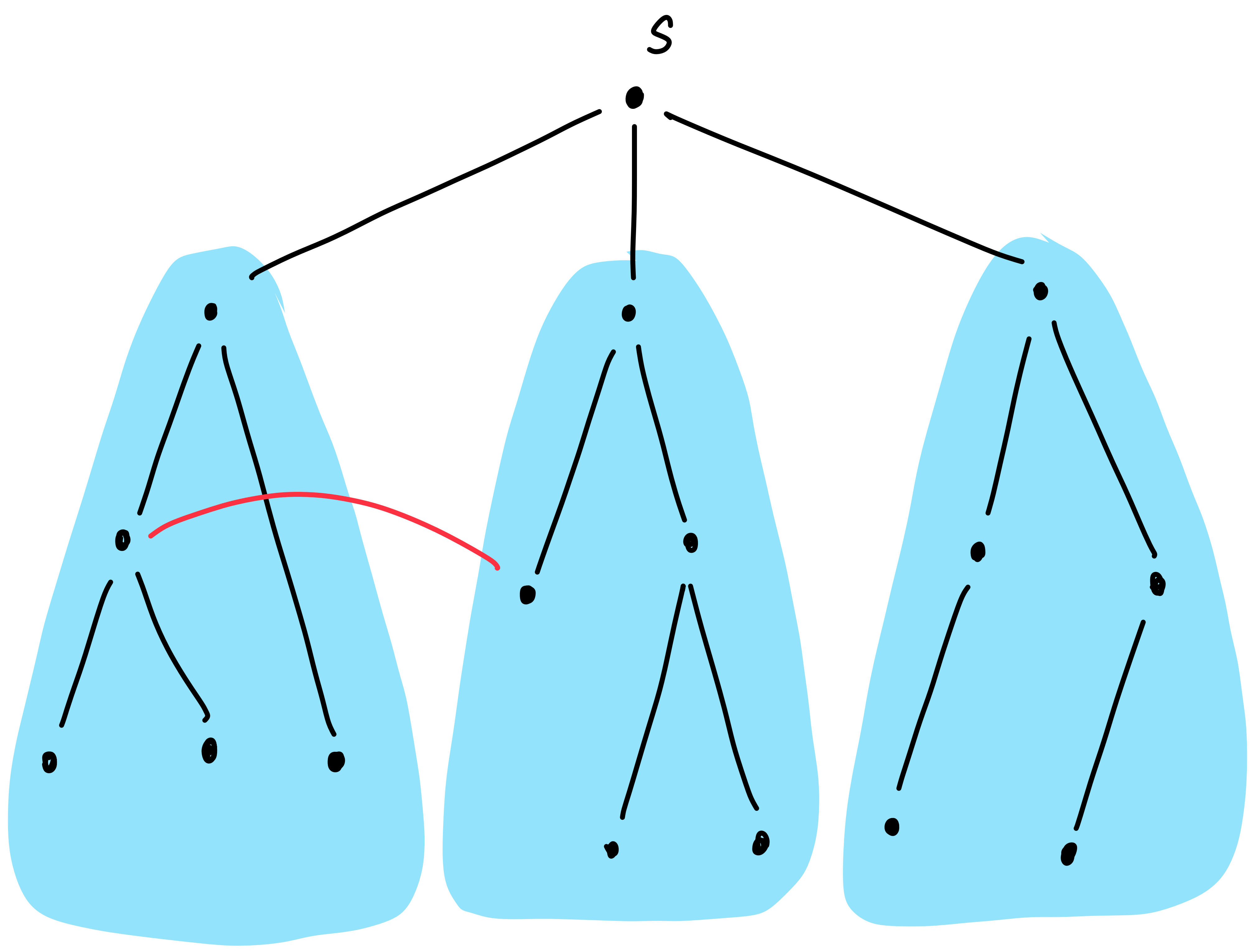
Understanding the DFS spanning tree

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



Understanding the DFS spanning tree

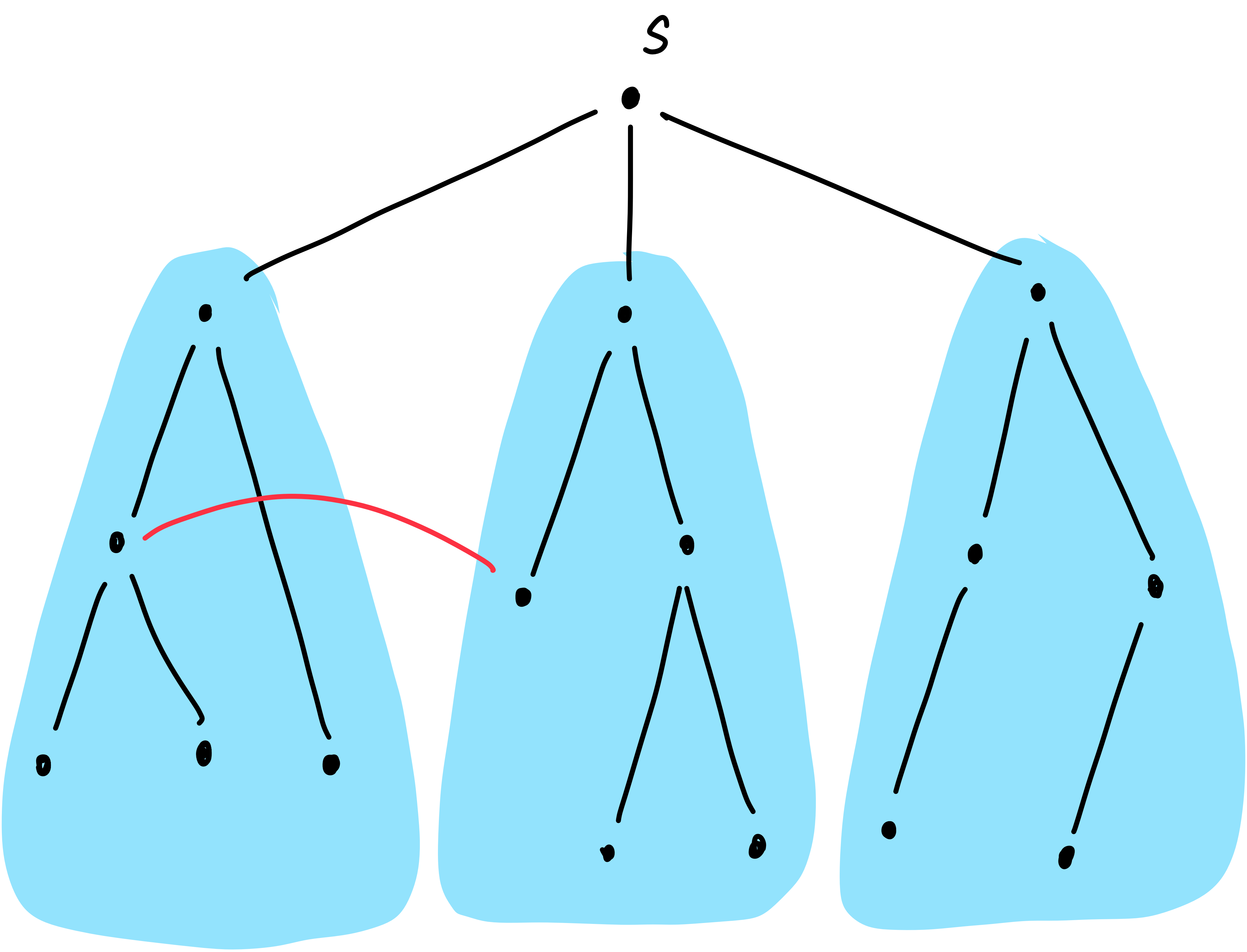
- What do the edges not included in the spanning tree look like?



Could this red edge exist in the graph?

Understanding the DFS spanning tree

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



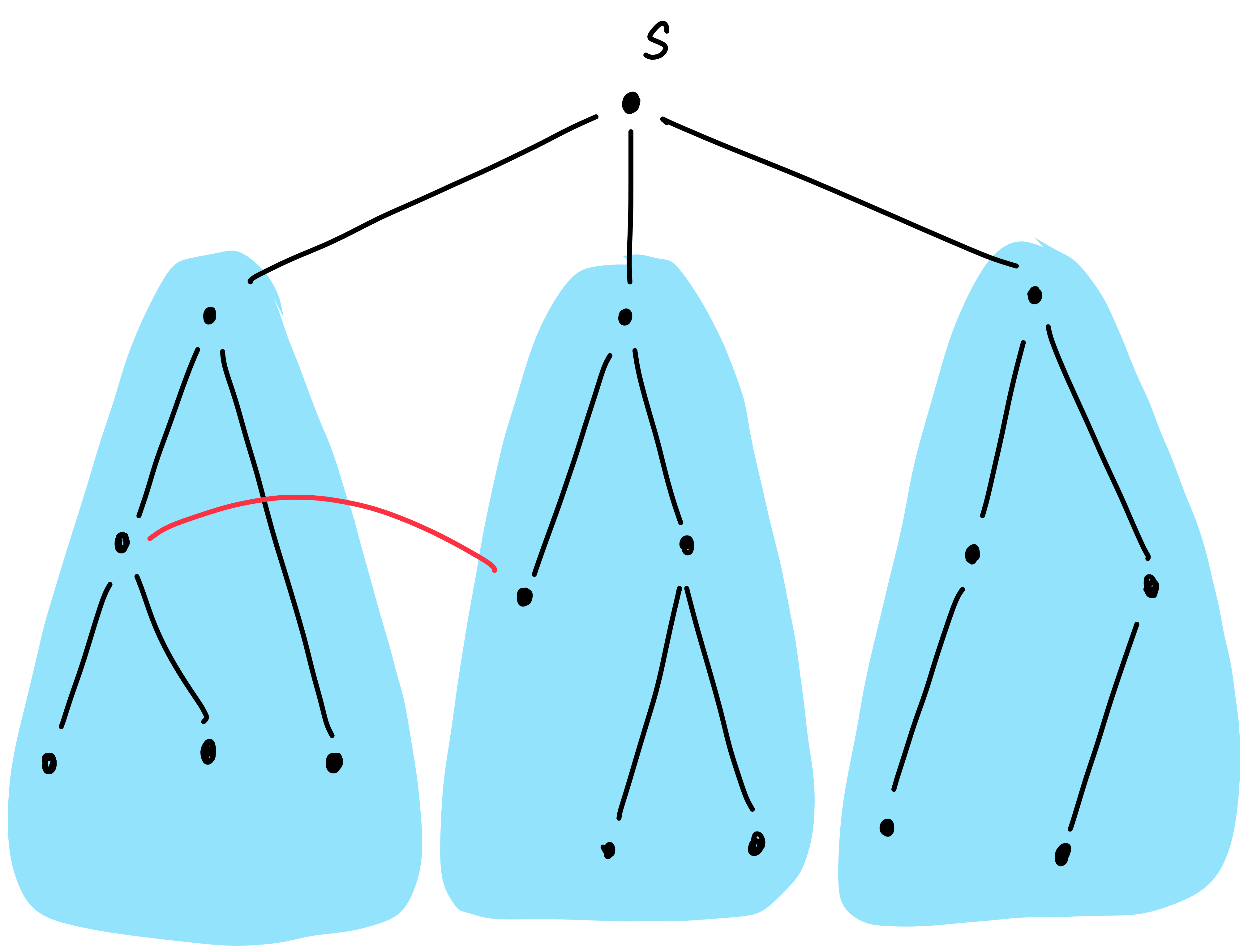
Could this red edge exist in the graph?

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?



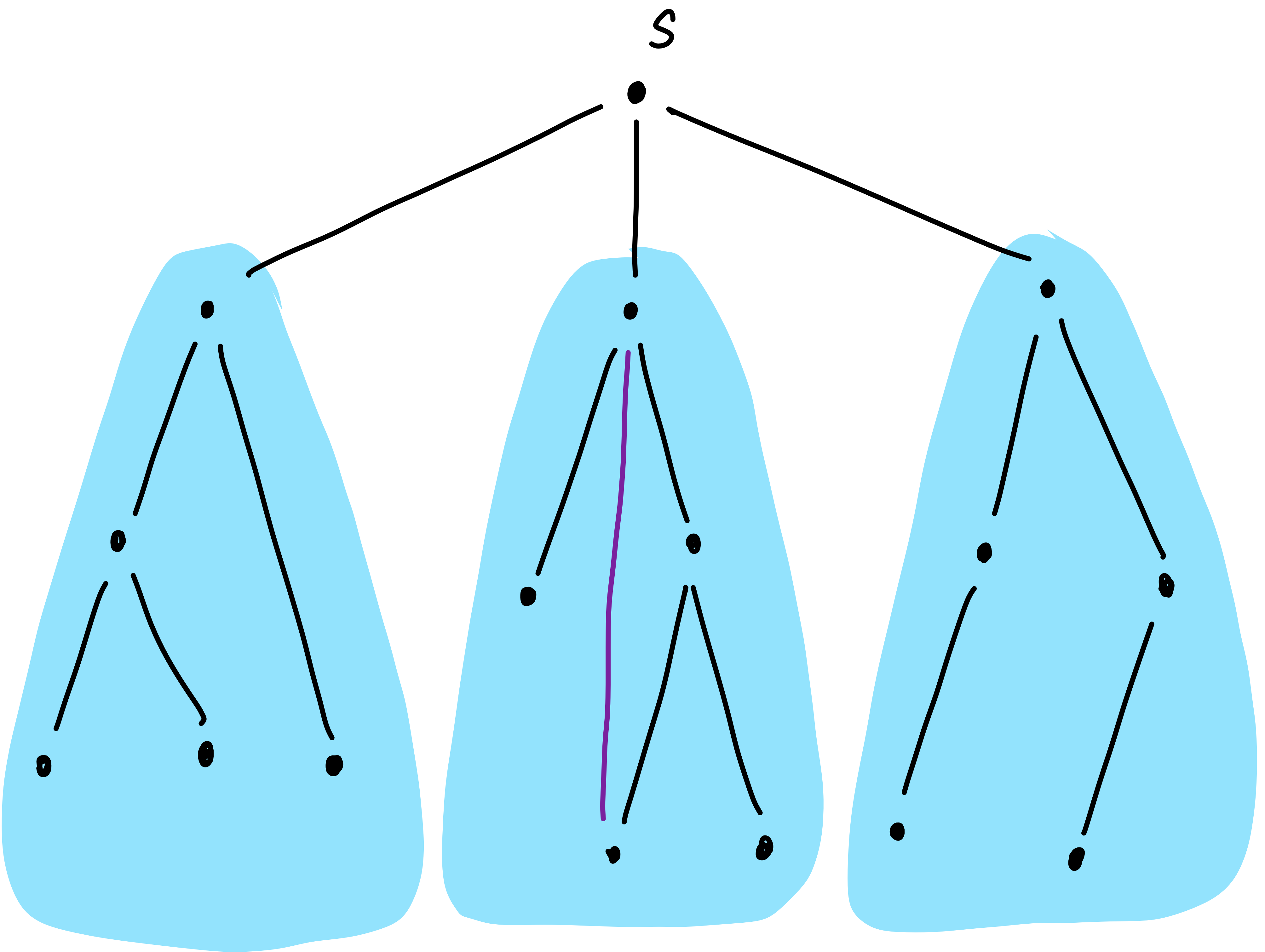
Could this red edge exist in the graph?

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

Such an edge is called a cross edge.

Understanding the DFS spanning tree

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

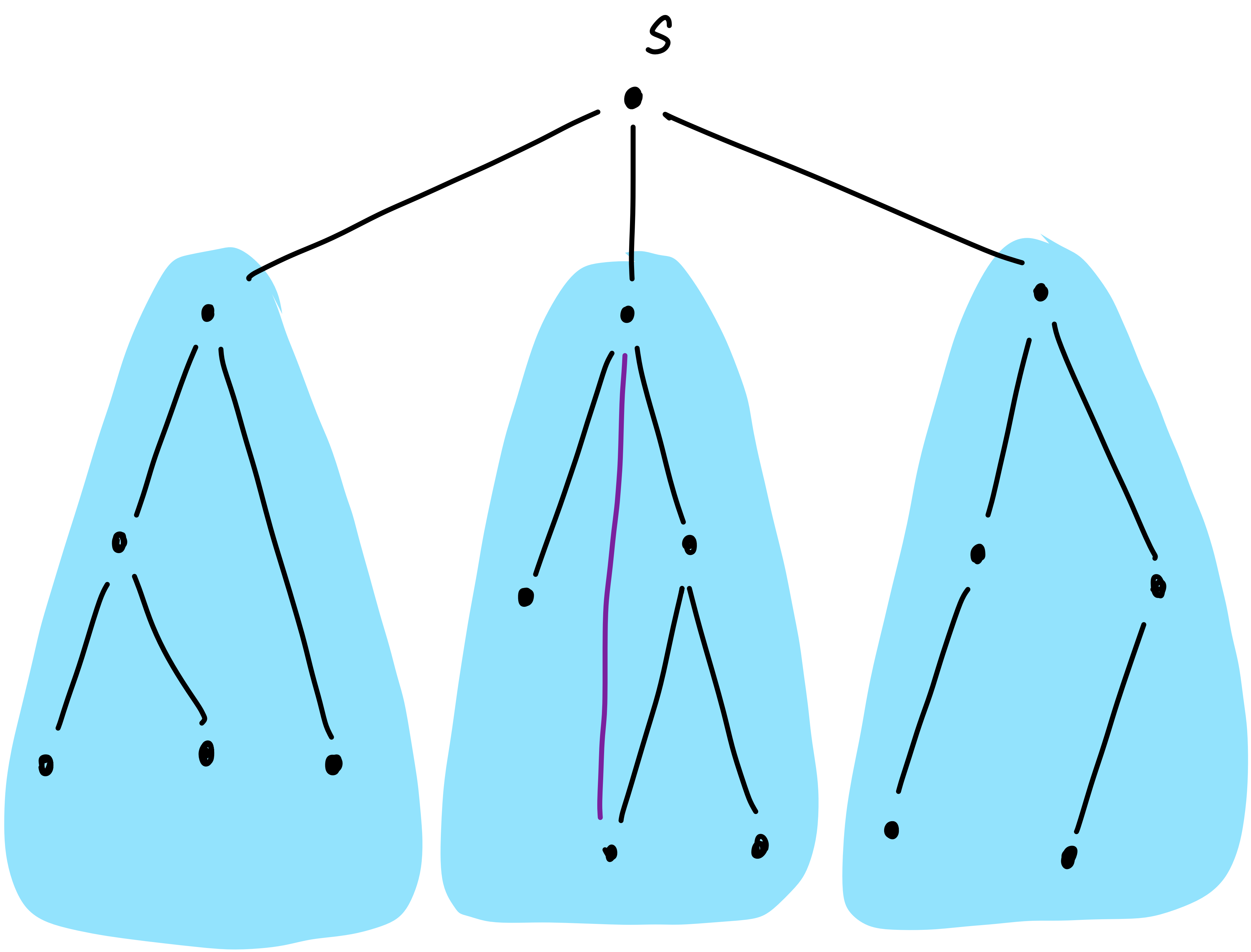


What about this purple edge?

Understanding the DFS spanning tree

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

Def. A <u>back edge</u> is an edge that connects a vertex to an ancestor that is not it's parent in the tree.

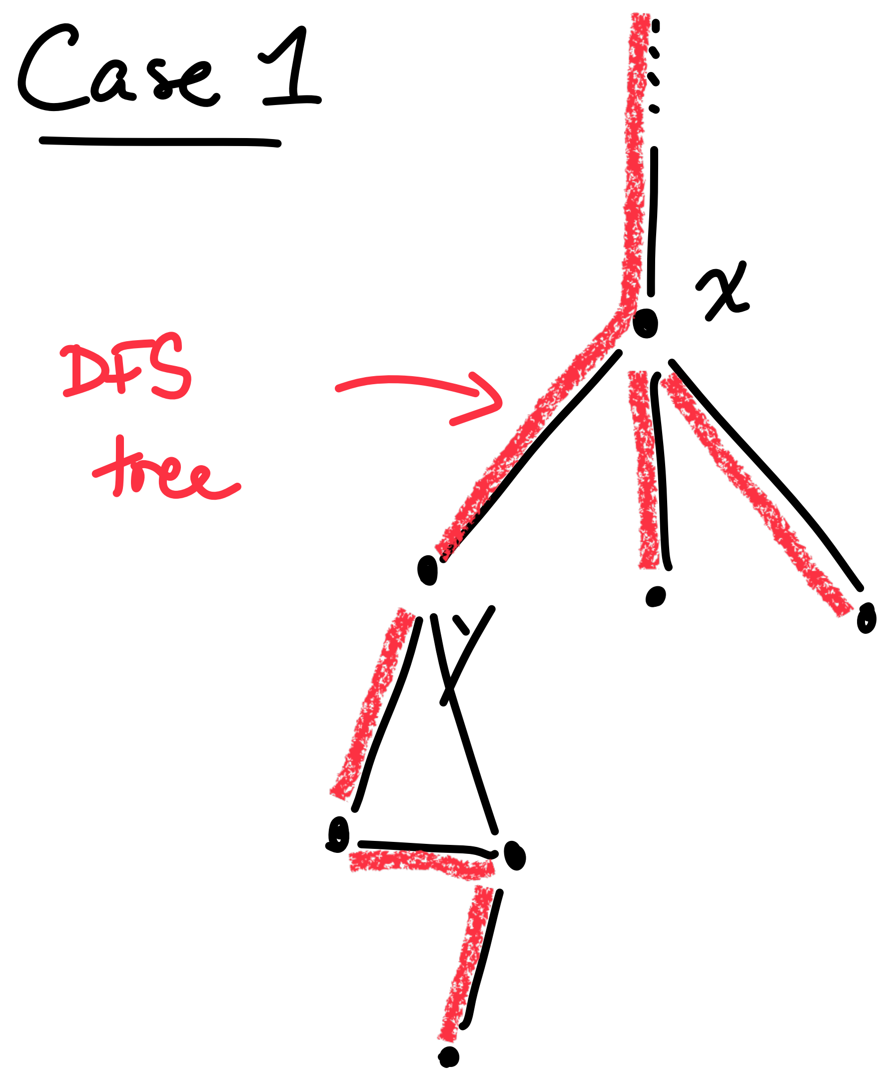


What about this purple edge?

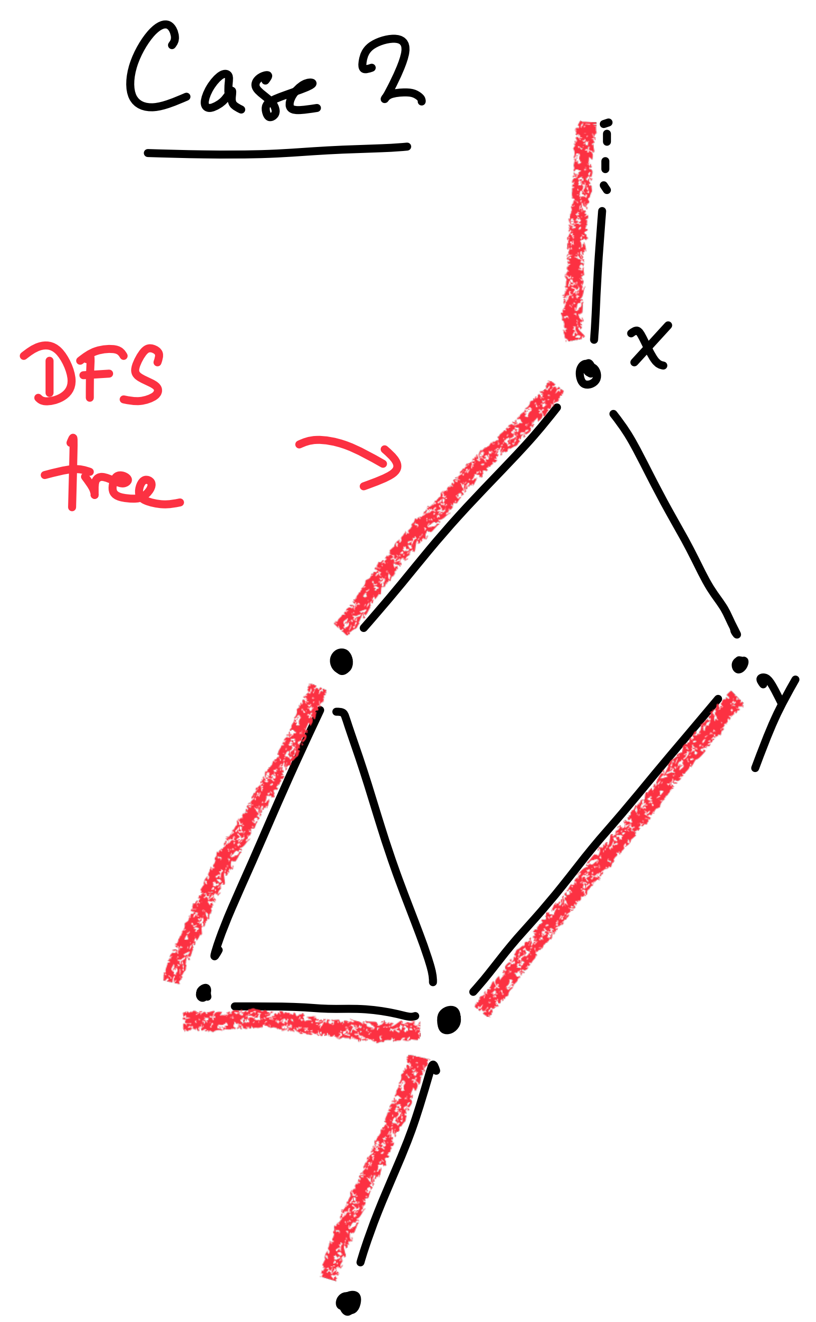
Yes. Edges can connect down tree.

No cross edges in DFS

(for undirected graphs)



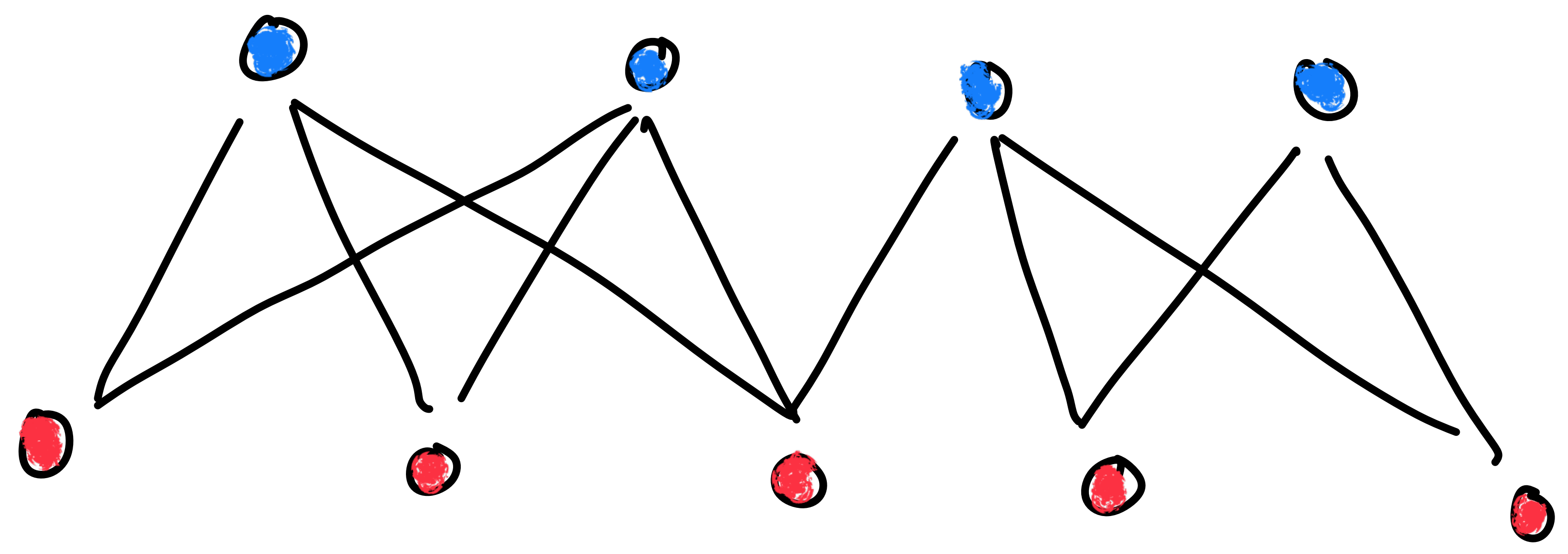
- **Claim:** During DFS(x), every vertex marked "visited" is a descendant of x in the DFS tree T.
- **Claim:** For every edge $(x, y) \in E$, either (x, y) is an edge in T, or else x or y is an ancestor of the other in T.
- **Proof:**
  - DFS is called recursively as we explore. Wlog, assume DFS(x) is 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

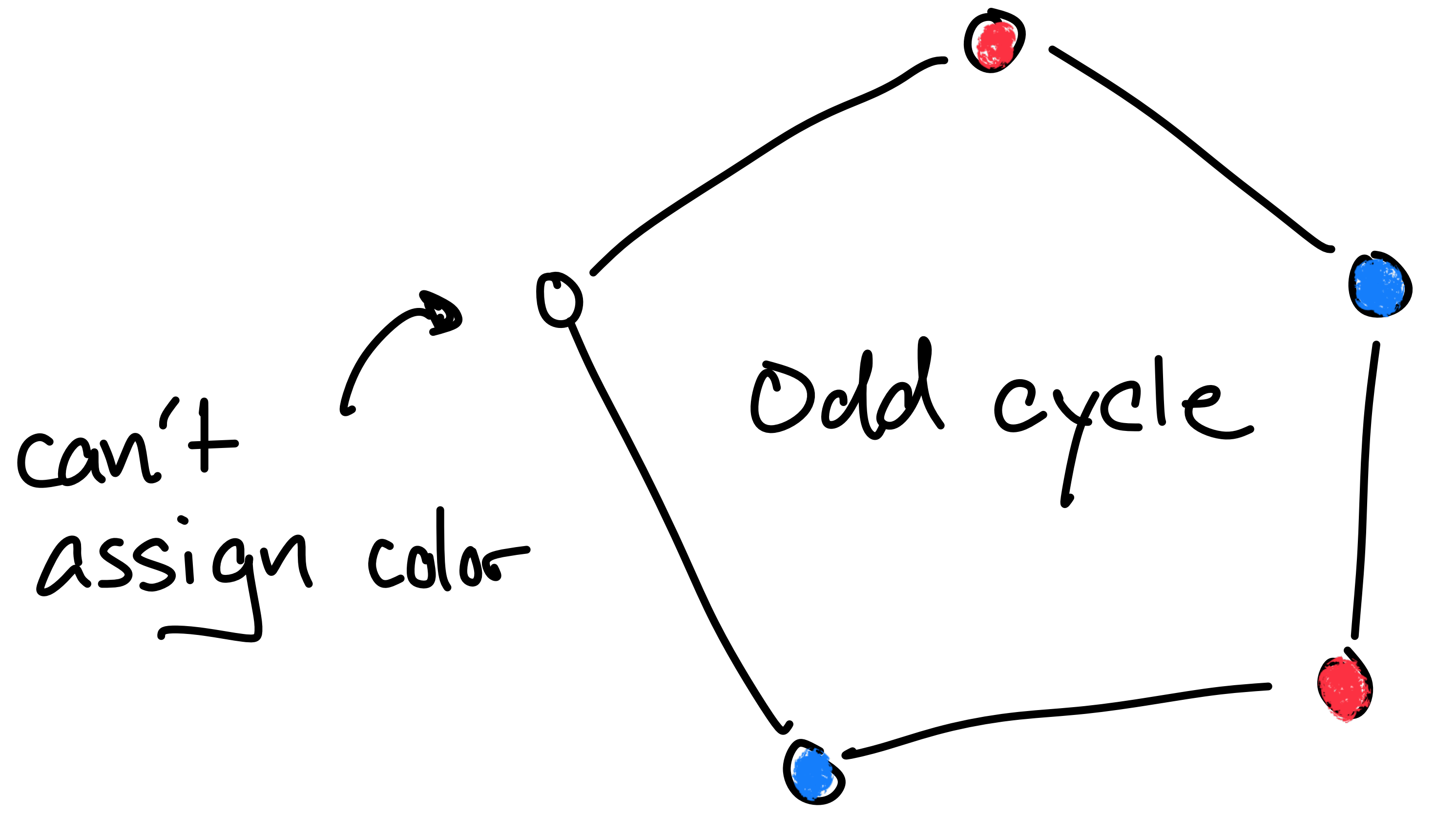
Bipartiteness testing

Application of graph traversal

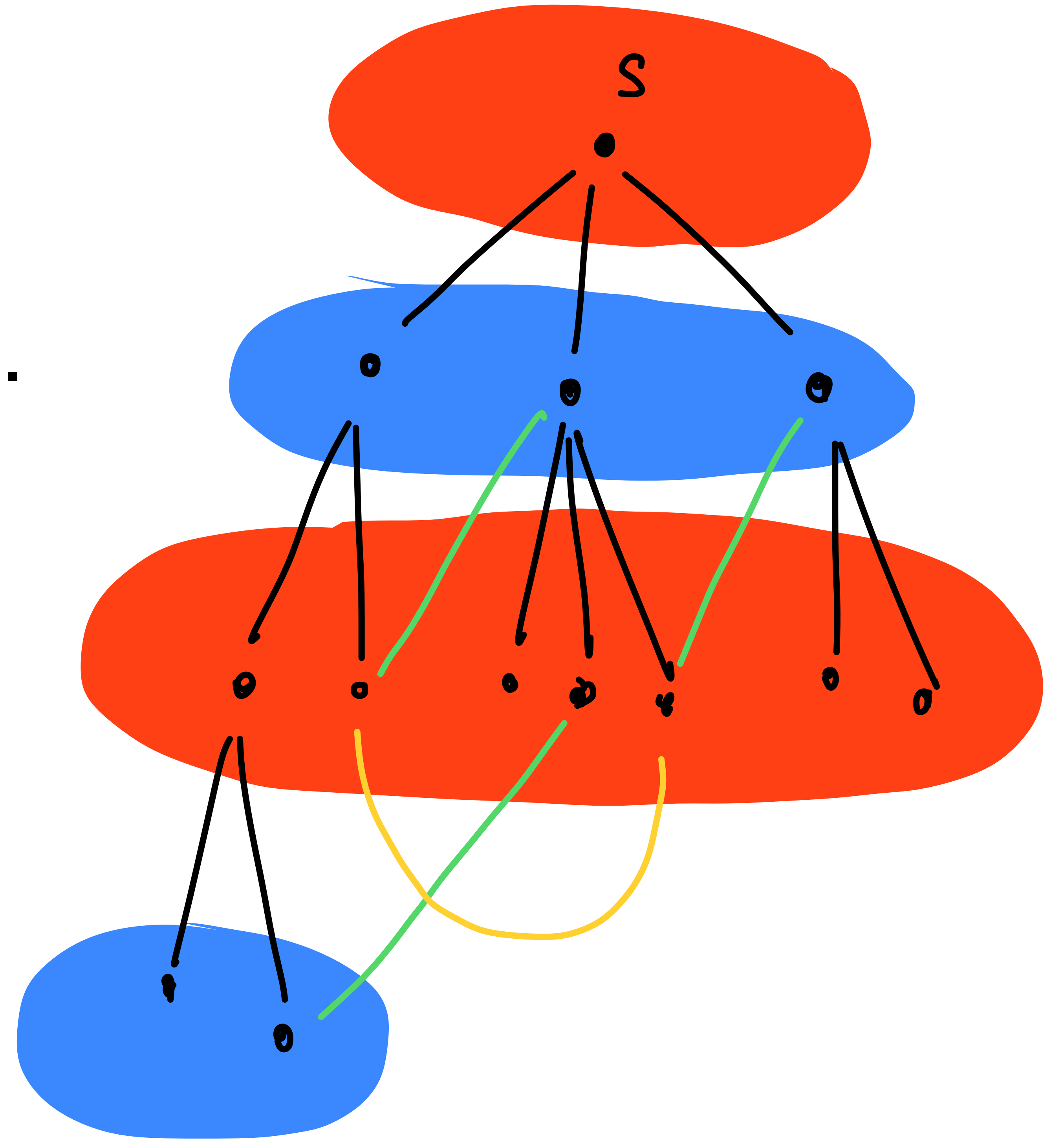


- Recall, a graph is bipartite iff we can split $V = X \sqcup Y$ such that every edge is between $(x, y) \in X \times Y$.
- Equivalently, a graph is bipartite if we can color every vertex either *red* or *blue* such that each edge is between a red and a blue vertex.
- **Input:** Undirected graph G
-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

Bipartite graph property

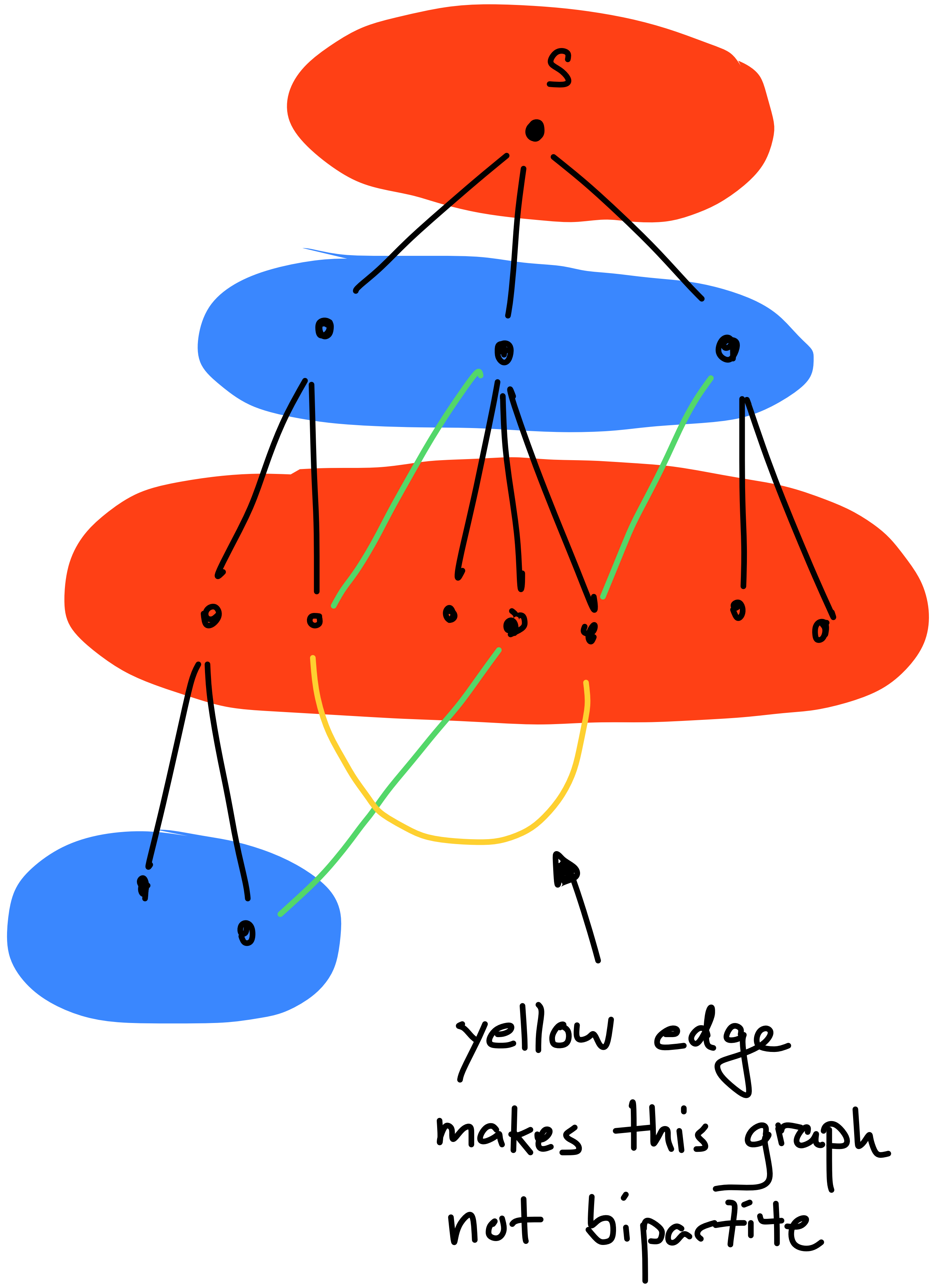


- **Claim:** A graph is bipartite iff it contains no *odd cycles*.
- **Proof:**
  - If it contains an odd cycle, we can't color the cycle let alone the rest of the graph.
  - If it contains no odd cycles, run BFS starting from some vertex s.
    - Color according to length from s in BFS tree with even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



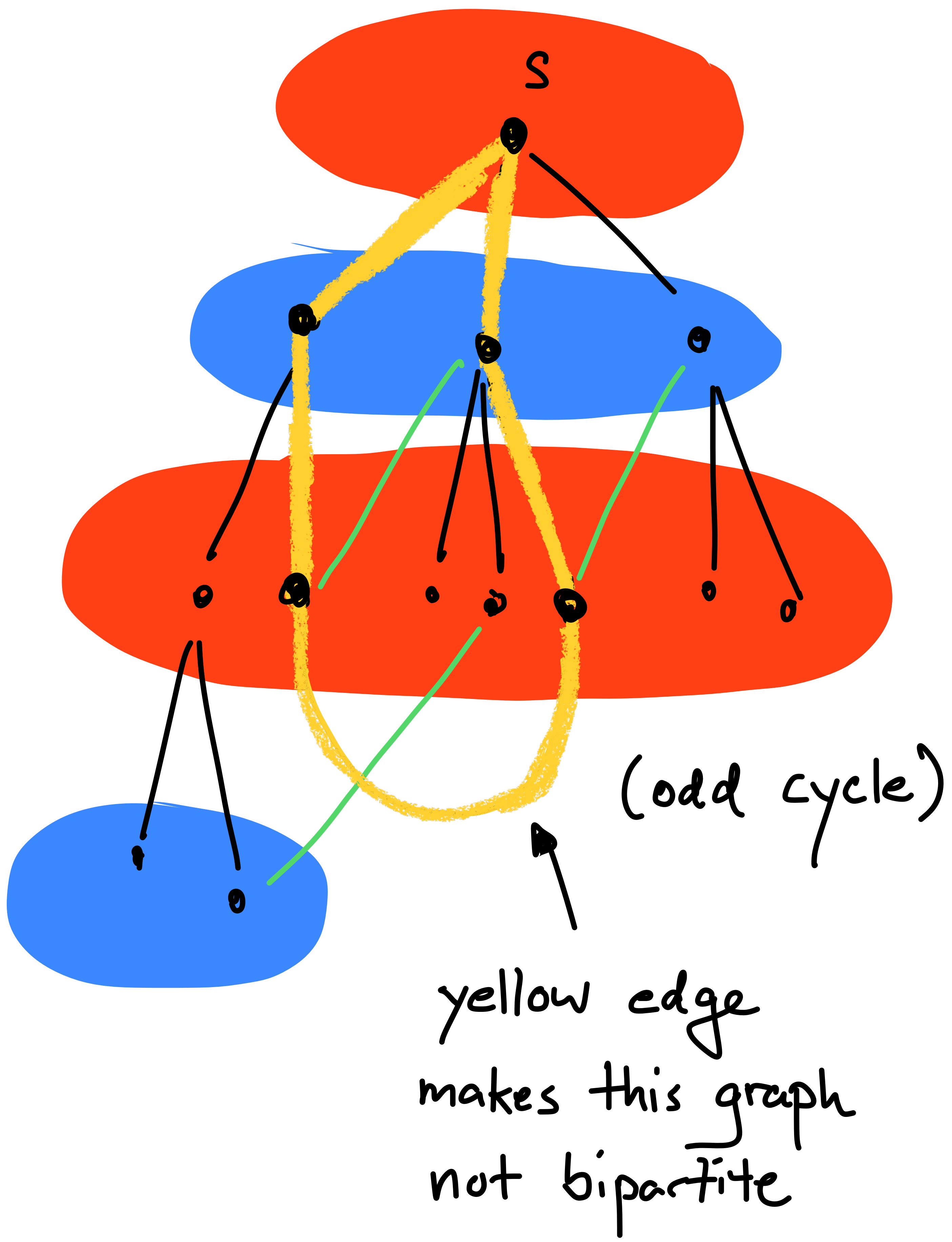
Bipartiteness testing

- **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, $O(n + m)$.

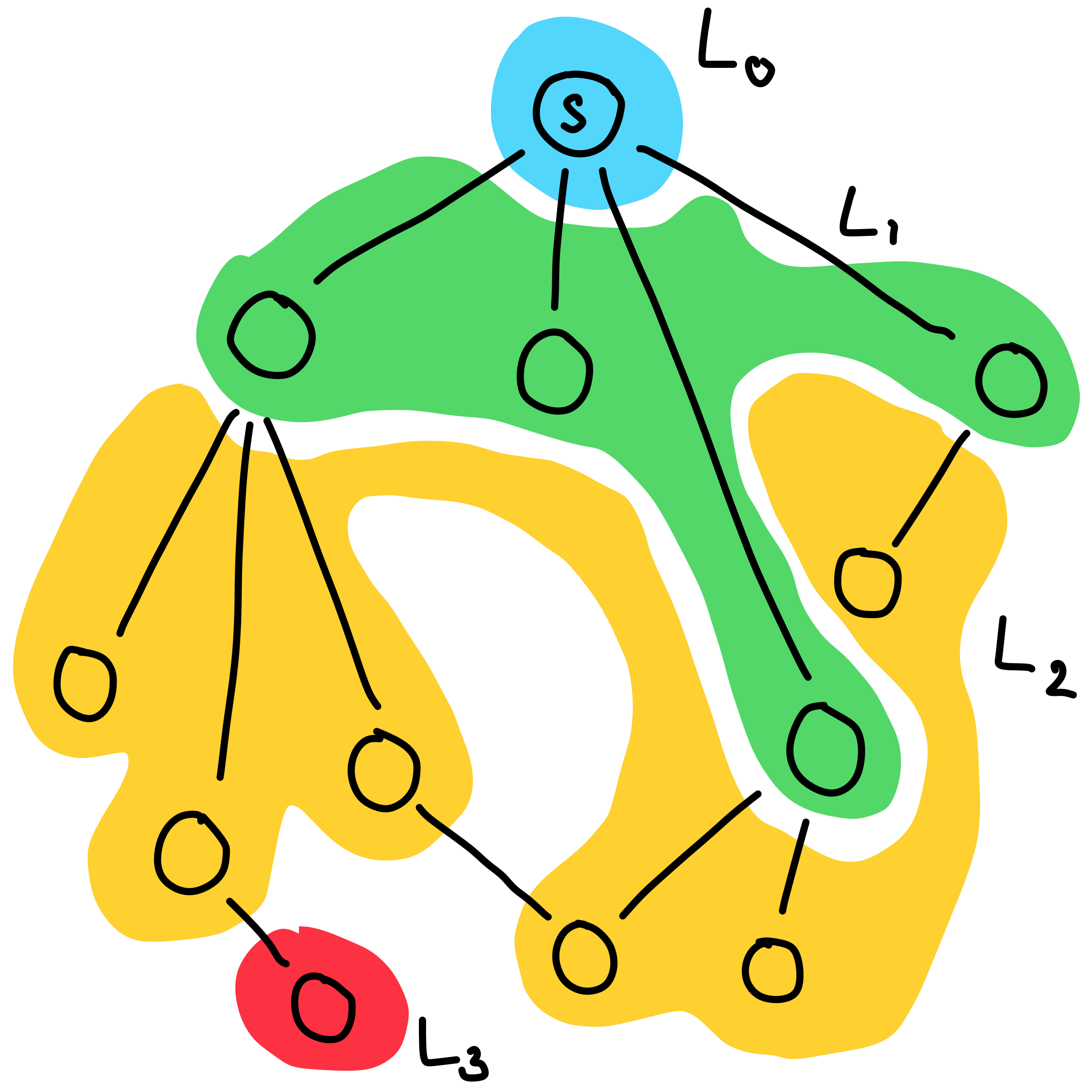


Bipartiteness testing

- **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, $O(n + m)$.

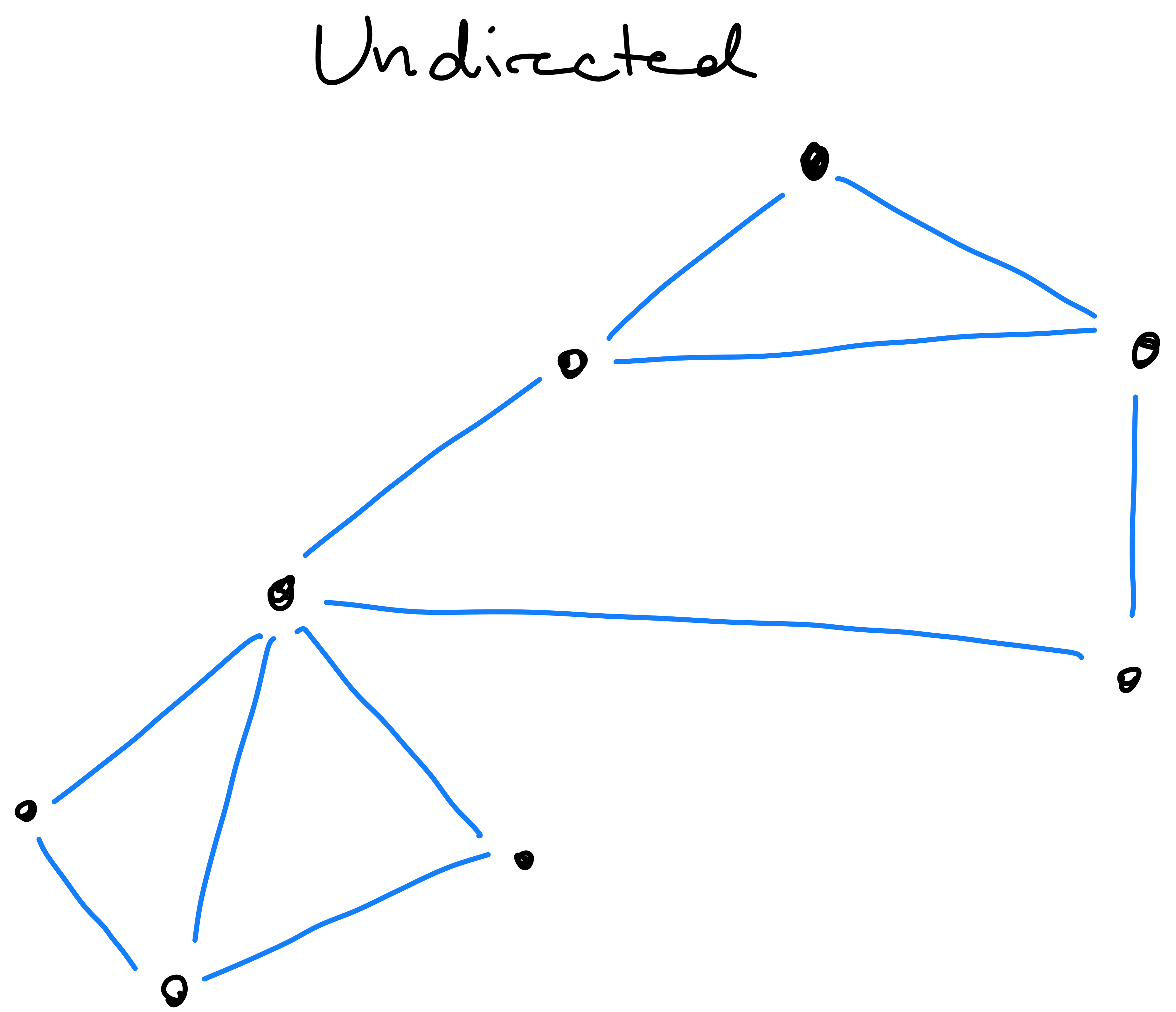


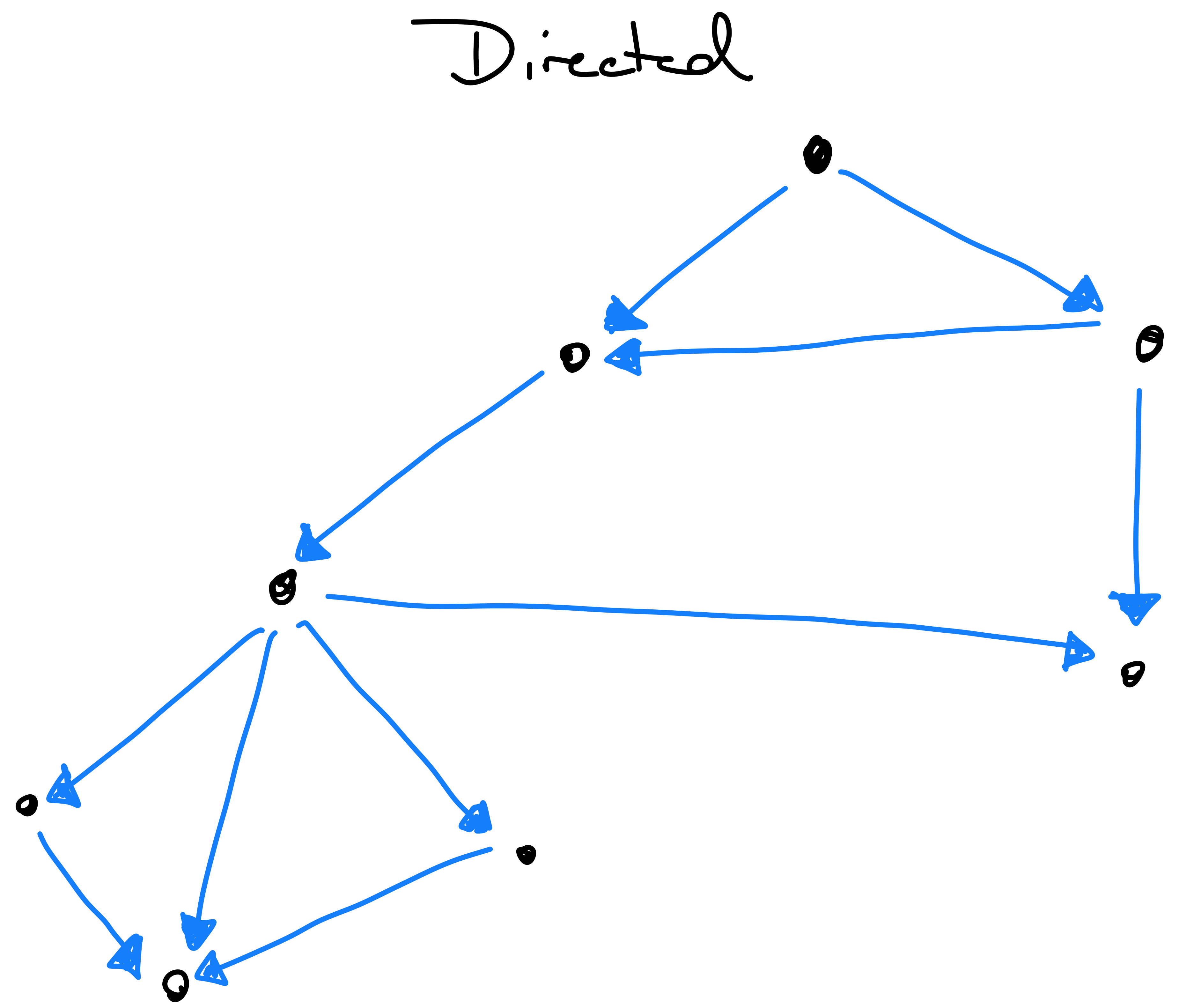
BFS edge property



- 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 $(u, v) \in L_i \times L_{i+2}$, 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

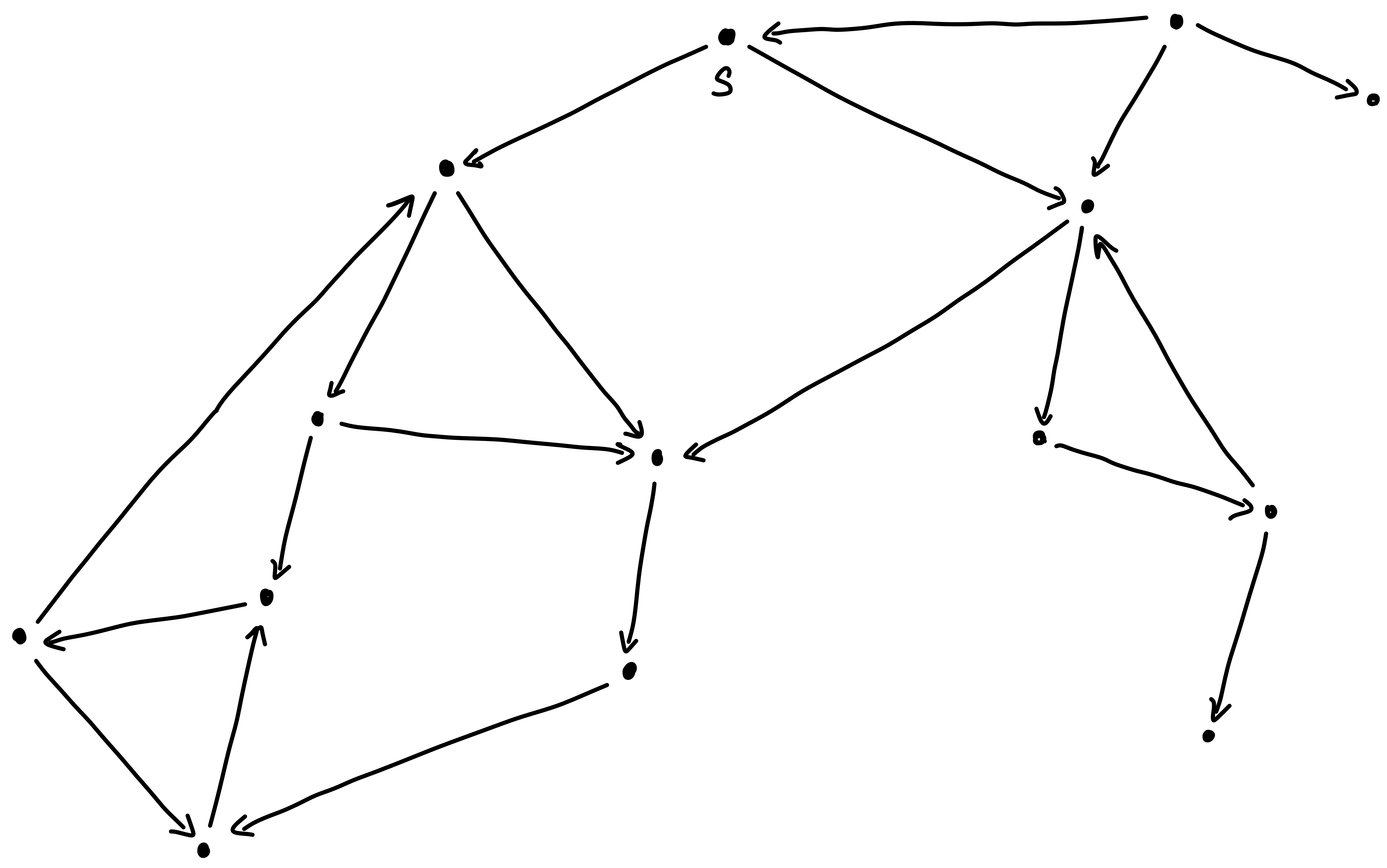




Depth-first search on directed graphs

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

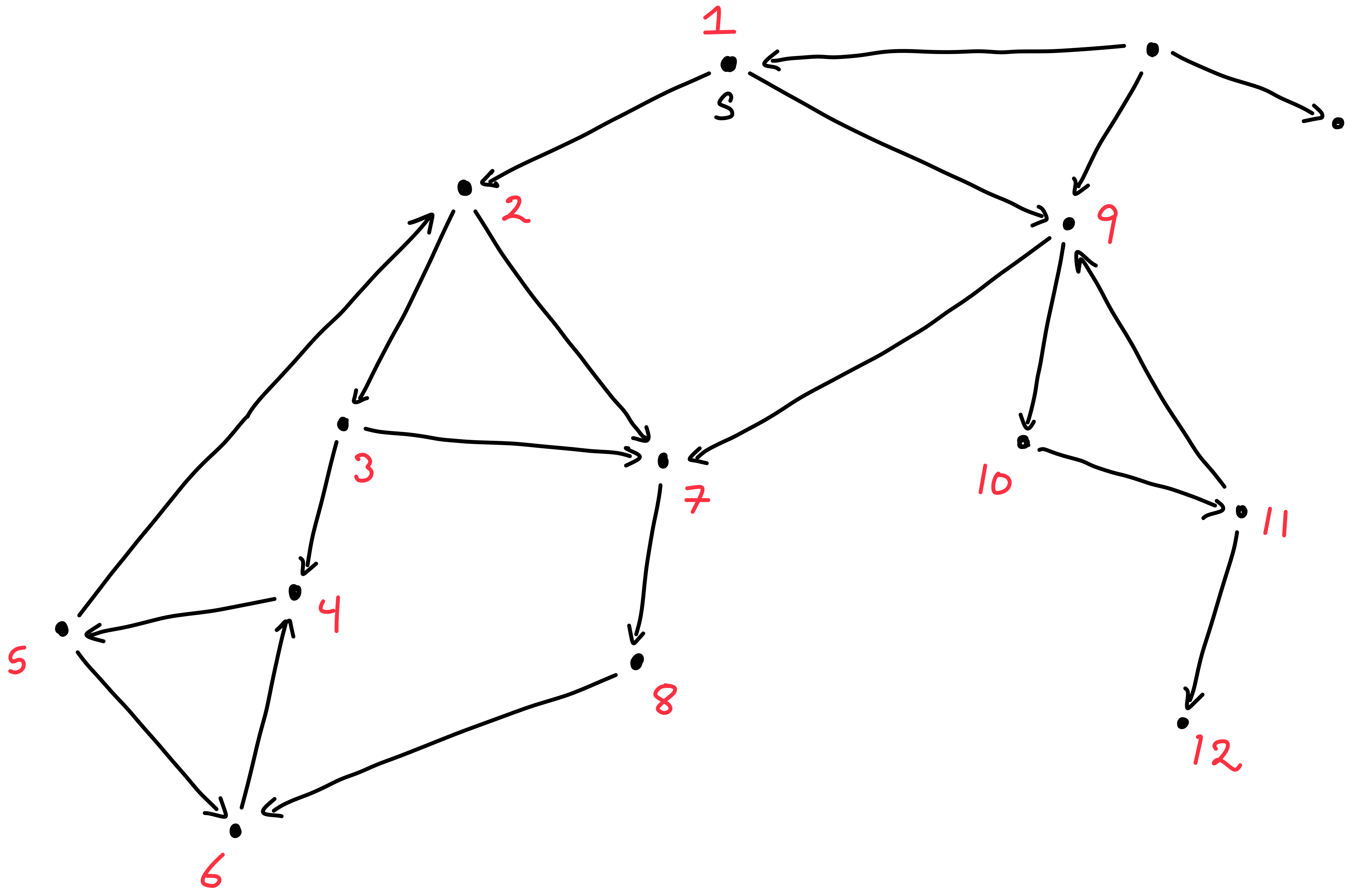
- DFS starting from s will visit all vertices u reachable by a *directed* path $s \rightsquigarrow u$.



Depth-first search on directed graphs

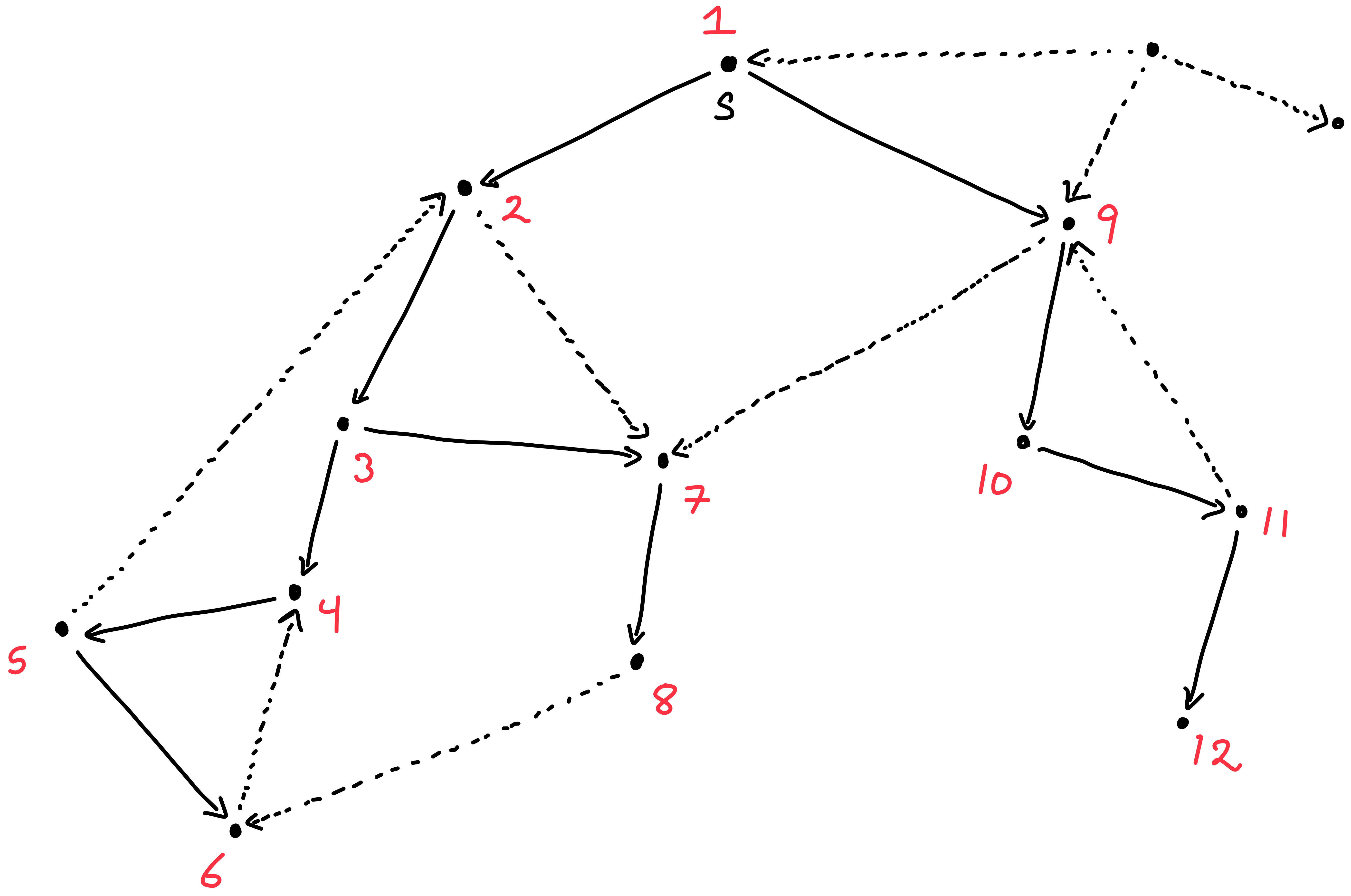
- Same as DFS on undirected graphs except we only add neighbor v if an edge points from $u \to v$.

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

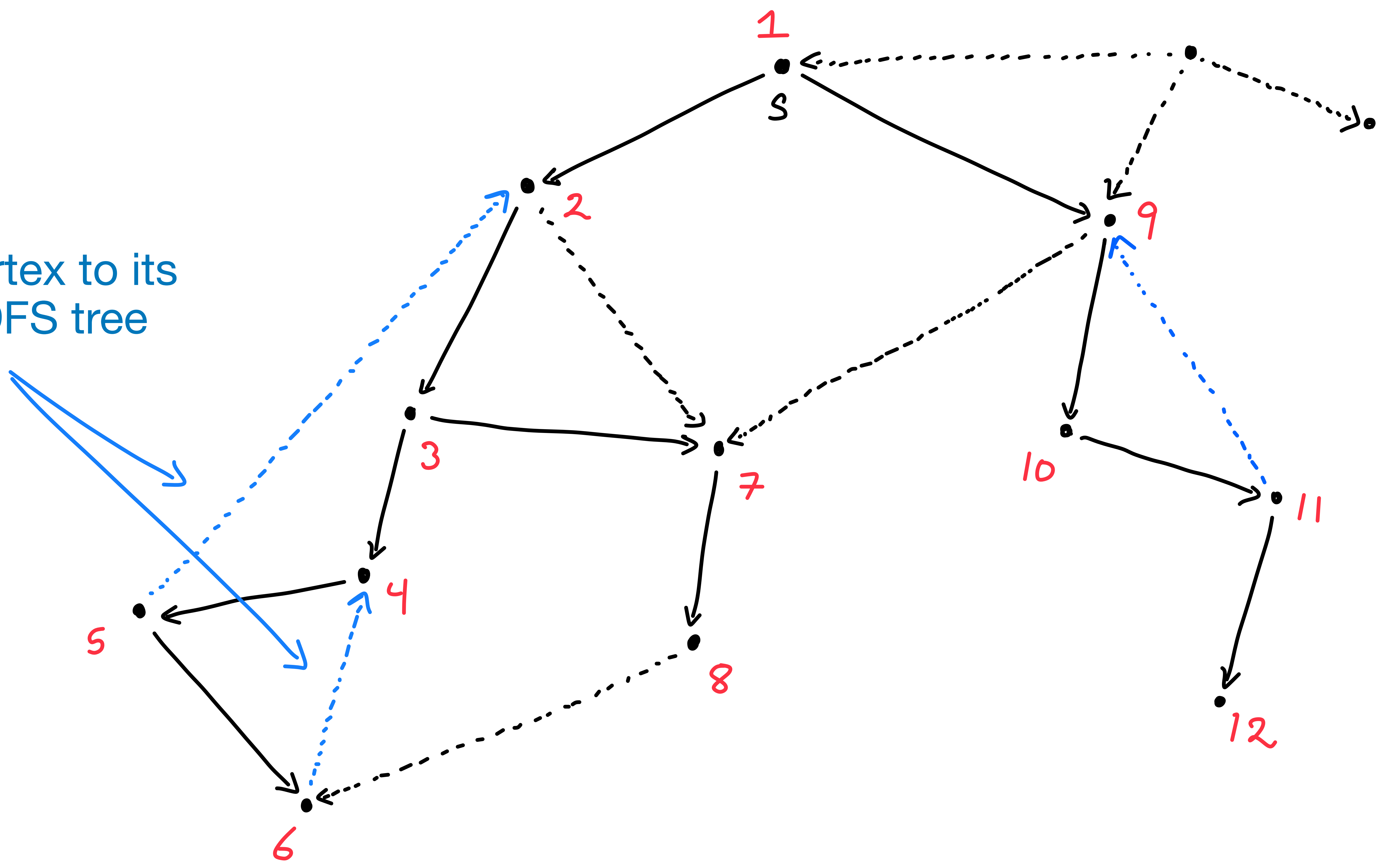


Depth-first search on directed graphs

- 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 $s \rightsquigarrow u$.



DFS edge nomenclature



Back edge

Connects vertex to its ancestor in DFS tree

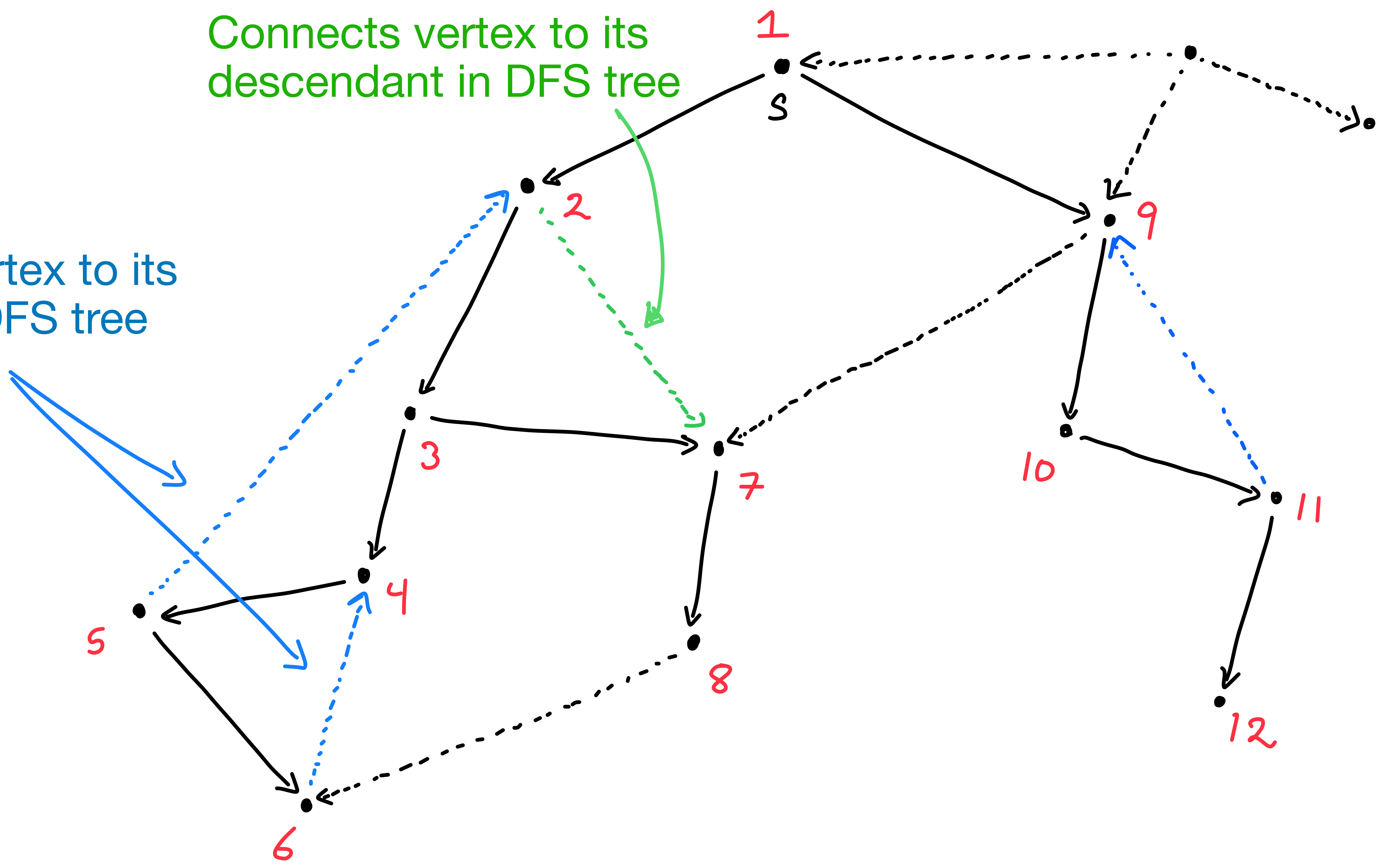
DFS edge nomenclature

Forward edge

Connects vertex to its descendant in DFS tree

Back edge

Connects vertex to its ancestor in DFS tree



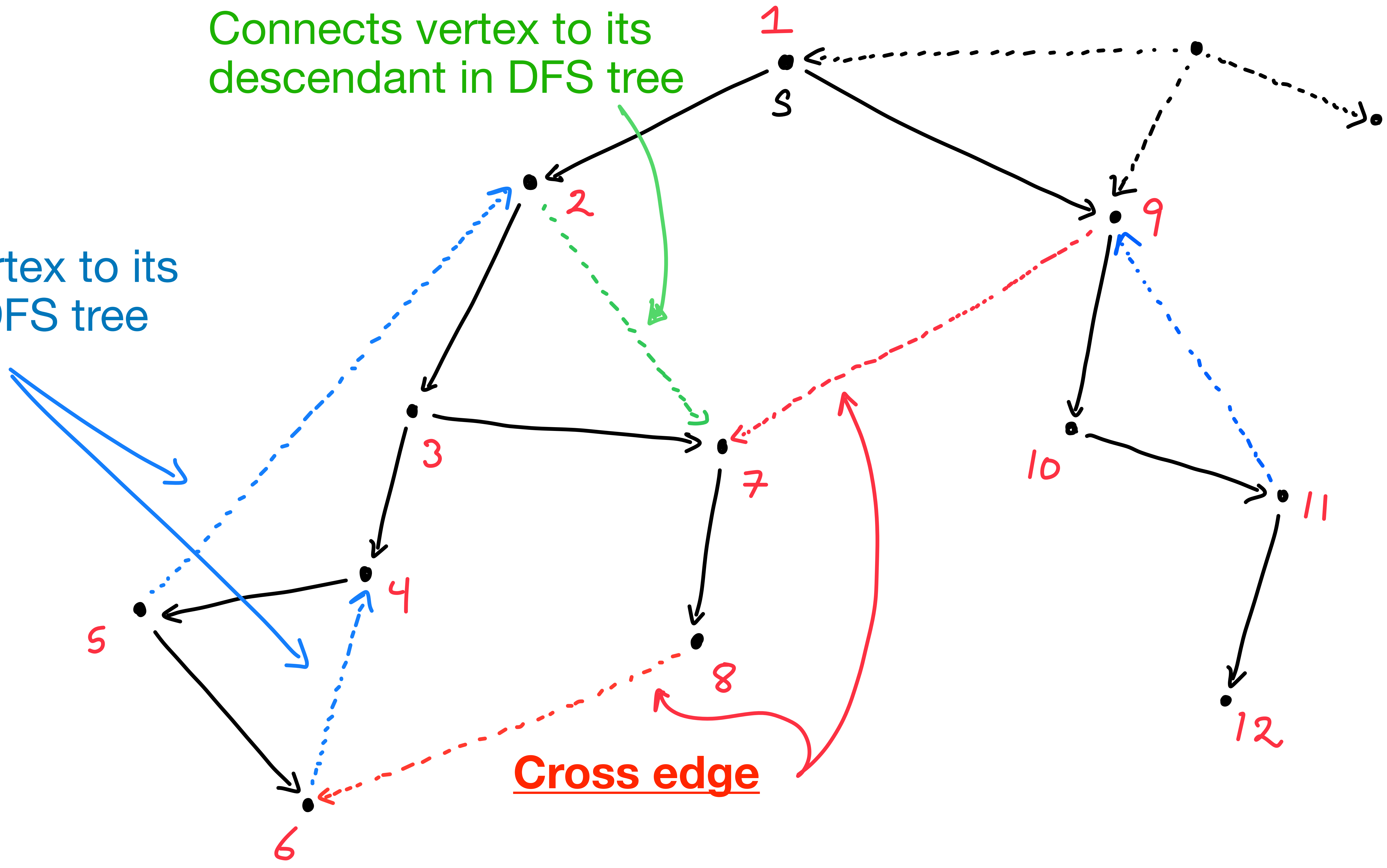
DFS edge nomenclature

Forward edge

Connects vertex to its descendant in DFS tree

Back edge

Connects vertex to its ancestor in DFS tree



Cross edge

Connects vertices across branches. Always high → low in DFS tree

DFS edge nomenclature

Unvisited edge

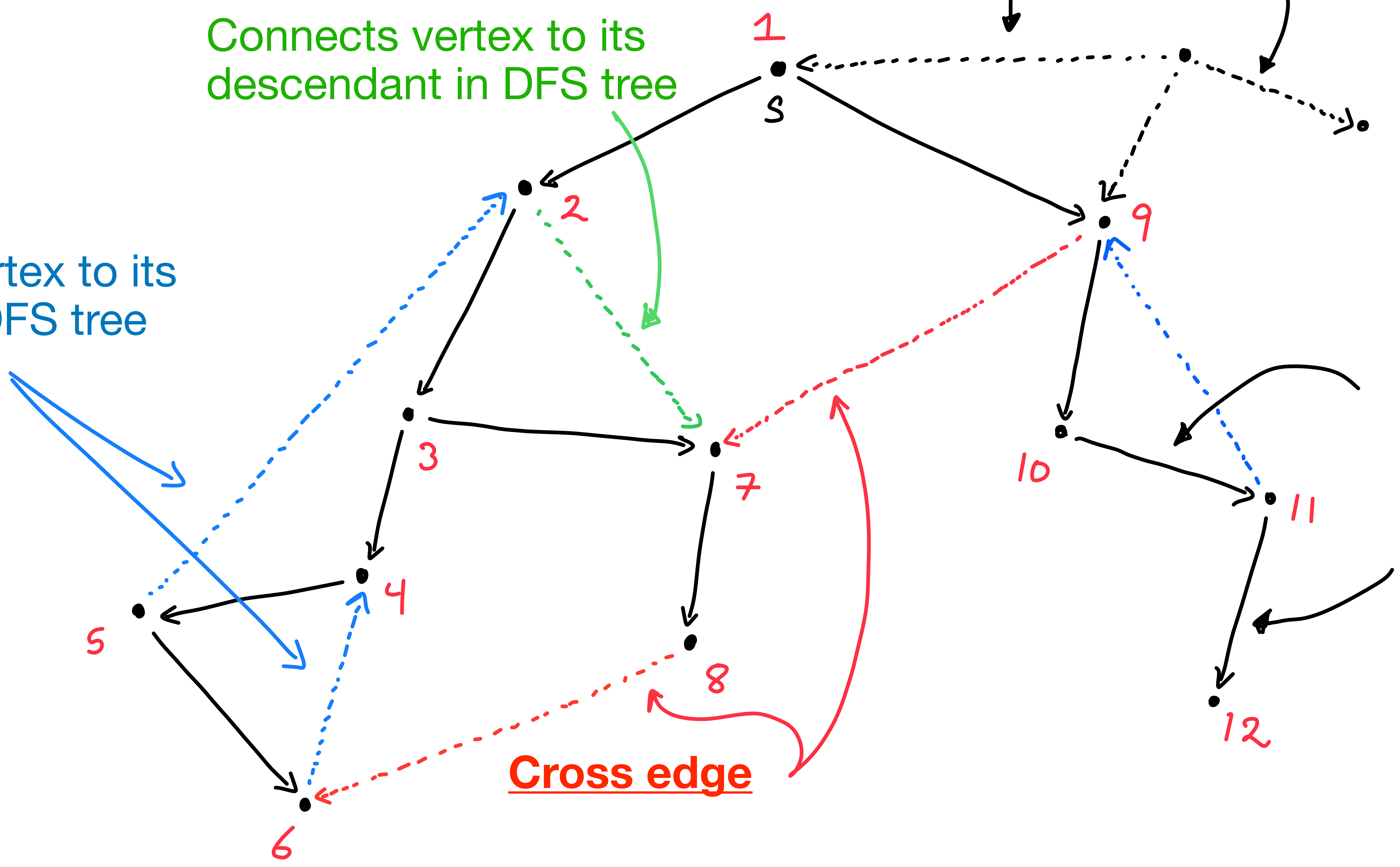
The edges not visited during DFS

Forward edge

Connects vertex to its descendant in DFS tree

Back edge

Connects vertex to its ancestor in DFS tree



Tree edge

The edges in the DFS tree

Cross edge

Connects vertices across branches. Always high → low in DFS tree

DFS edge nomenclature

Forward edge

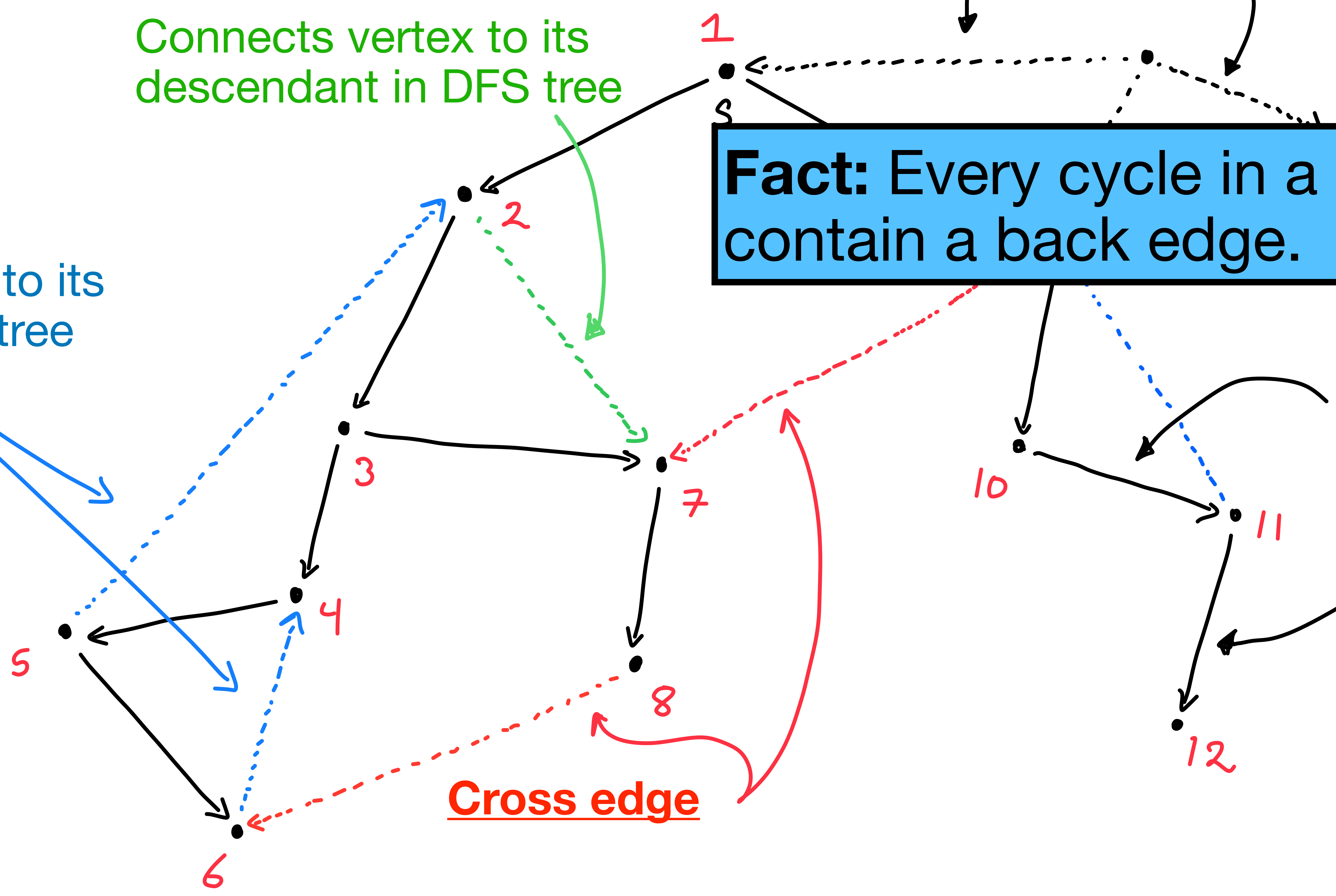
Connects vertex to its
descendant in DFS tree

Back edge

Connects vertex to its
ancestor in DFS tree

Unvisited edge

The edges not visited during DFS



Fact: Every cycle in a directed graph must contain a back edge.

Tree edge

The edges in the DFS tree

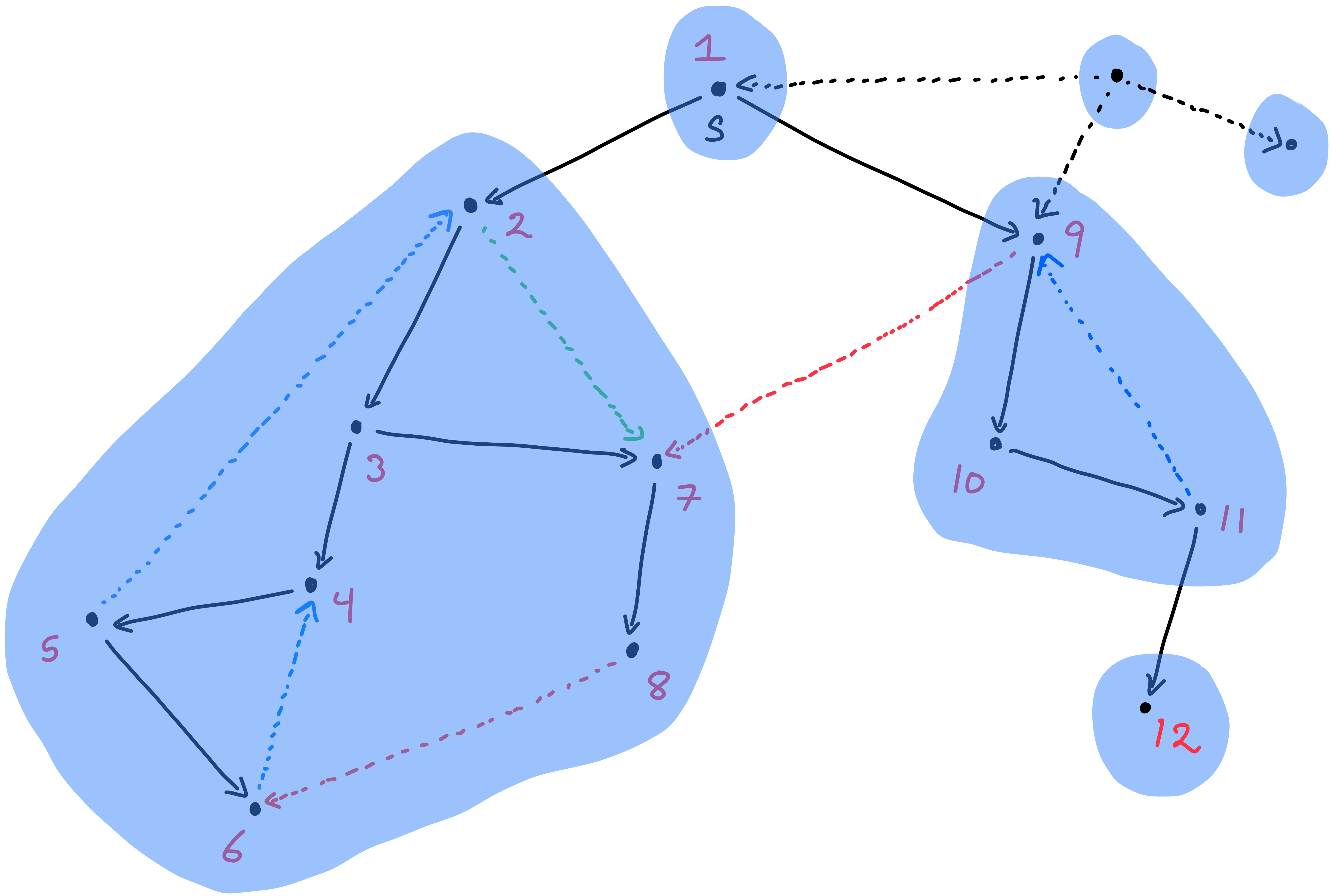
Cross edge

Connects vertices across branches. Always high → low in DFS tree

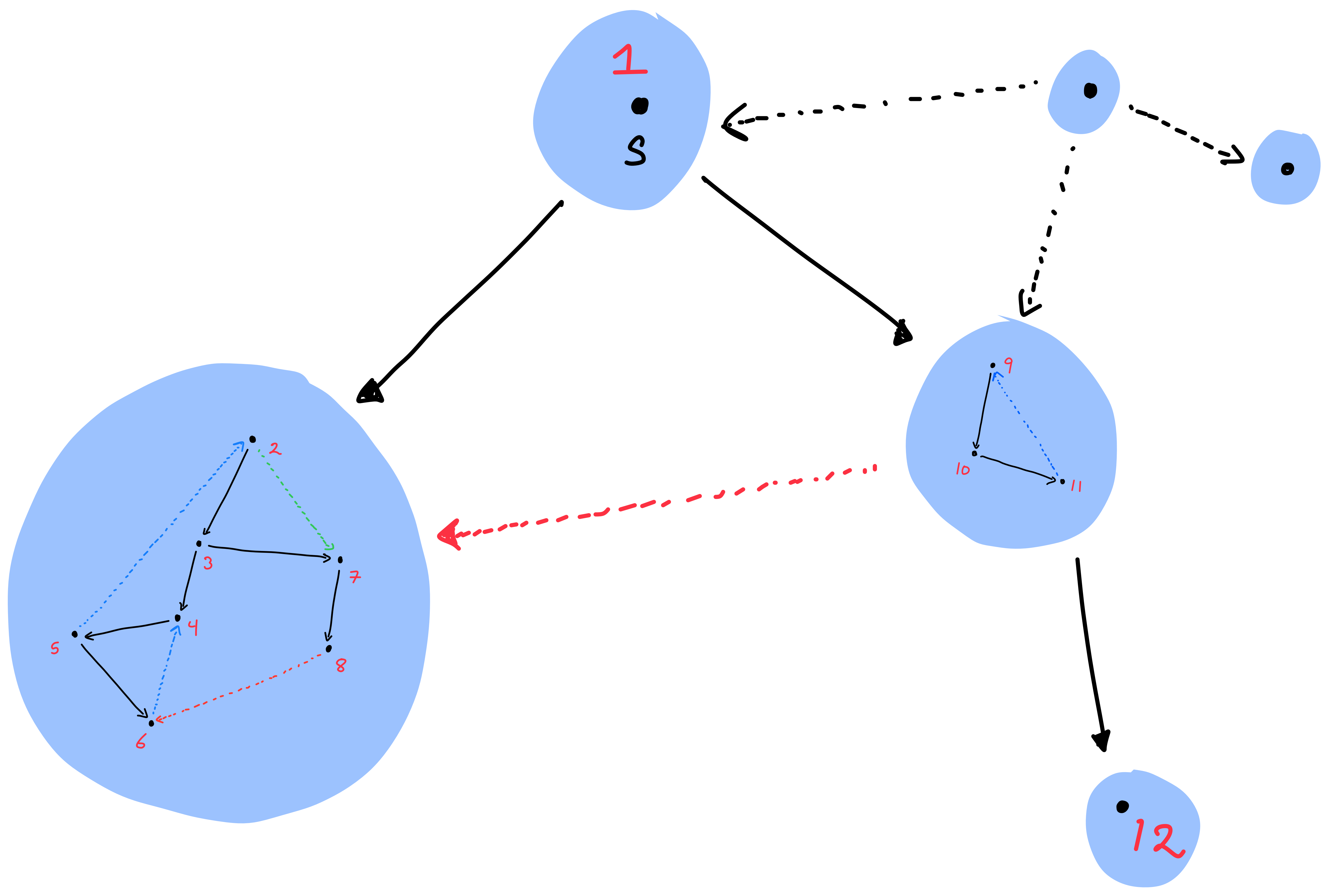
Strongly connected components

- Vertices u and v are **strongly connected** iff they are on a directed cycle. I.e. there is a directed path $u \rightsquigarrow v$ and a directed path $v \rightsquigarrow u$.
- Every directed graph's vertices can be partitioned into **strongly connected components** where all pairs of vertices in the component are *strongly connected*
- Strongly connected components (SCCs) can be stored efficiently just like connected components
- SCCs can be found by extending DFS algorithm in $O(n + m)$ time

Strongly connected components



Strongly connected components



Directed acyclic graphs

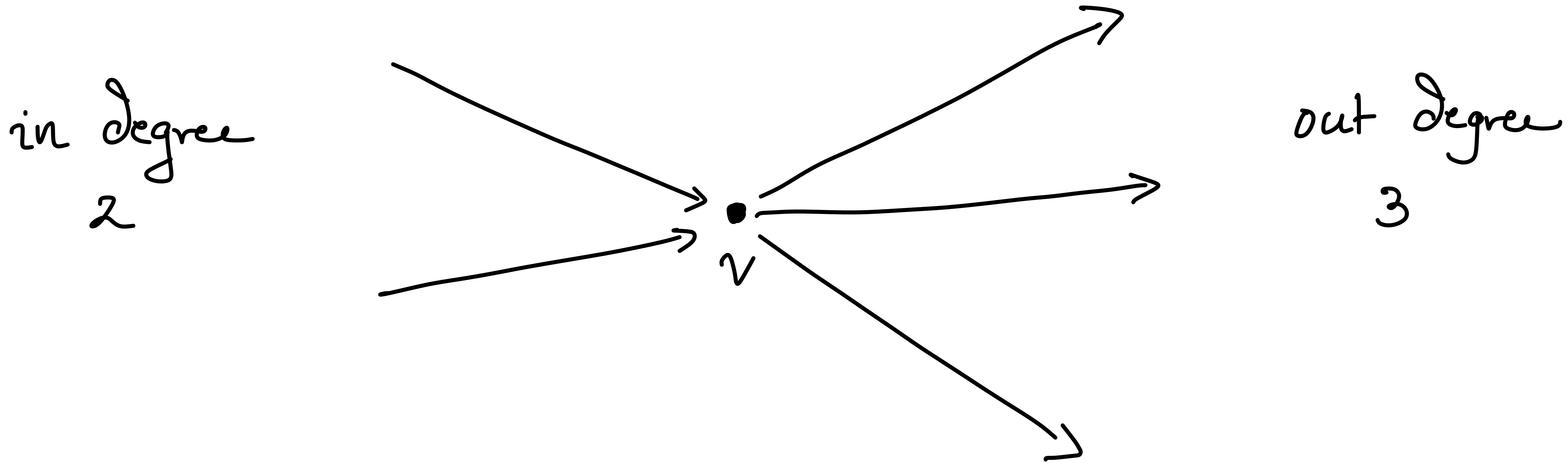
- A directed graph G is *acyclic* iff it has no directed cycles
- Also referred to as a DAG
- If we shrink every strongly connected component to a vertex, this converts the directed graph into a DAG

Topological sorting of graphs

- **Input:** a directed acyclic graph DAG $G = (V, E)$
- **Output:** An injective numbering $N : V \hookrightarrow \{1, \ldots, n\}$ such that edges only go from lower numbered to higher numbered vertices.

  i.e. for $u \to v$, we must have $N(u) < N(v)$.
- **Applications**
  - Vertices represents tasks and edges represent prerequisites
  - Topological sorts gives a sequential ordering for how to solve the system
- For general graphs, generate DAG by shrinking SCCs and then process SCCs in the order given by topological sort.

In-degree and out-degree



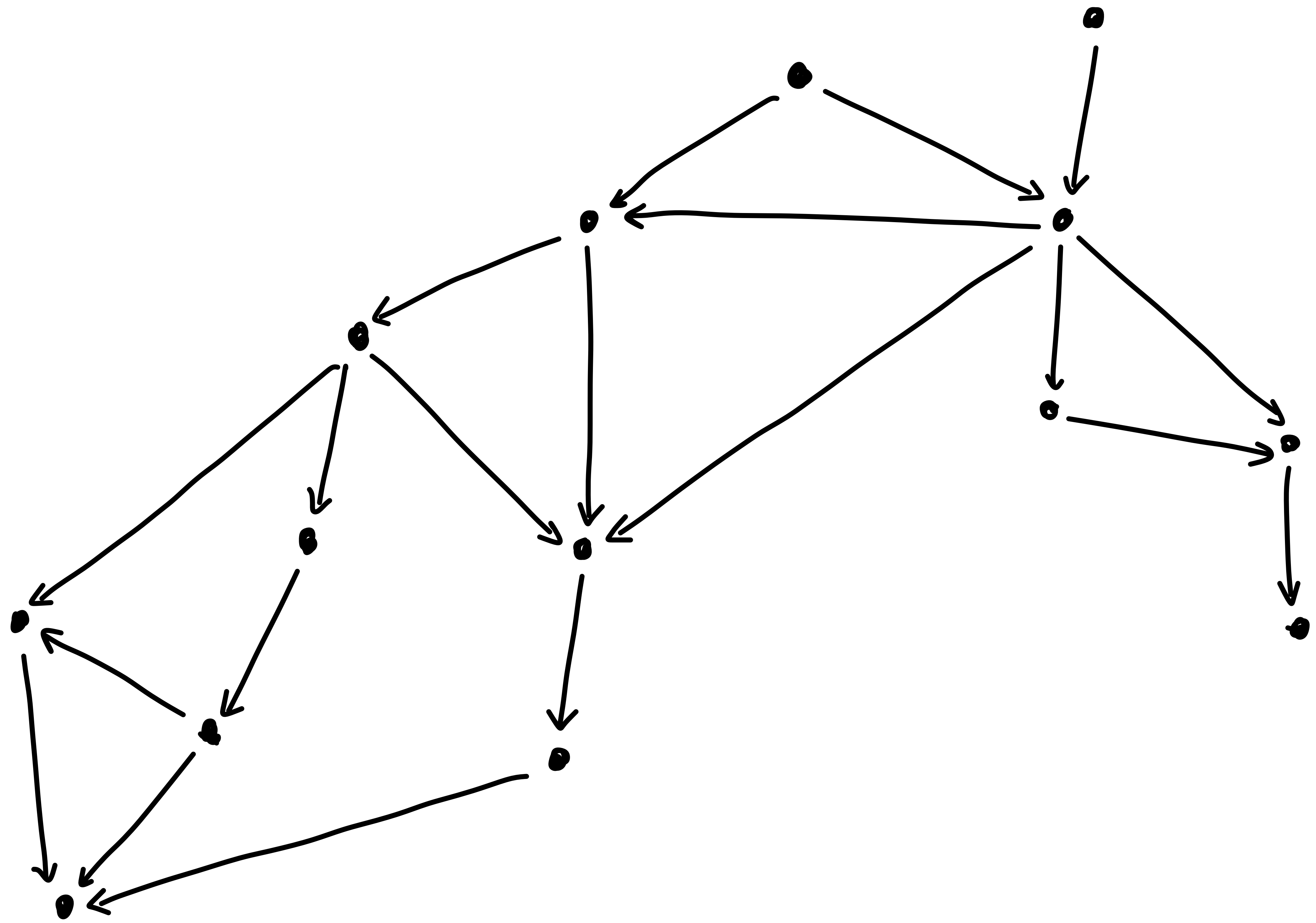
In-degree zero vertices

- **Claim:** Every DAG has at least one vertex of in-degree 0.
- **Proof:**
    - Assume every vertex has in-degree ≥ 1.
    - Starting with any vertex v pick an in-edge $u \to v$ and go in reverse to u. Repeat.
    - Since there are only n vertices, eventually a vertex will be repeated. This means there is a cycle, a contradiction.

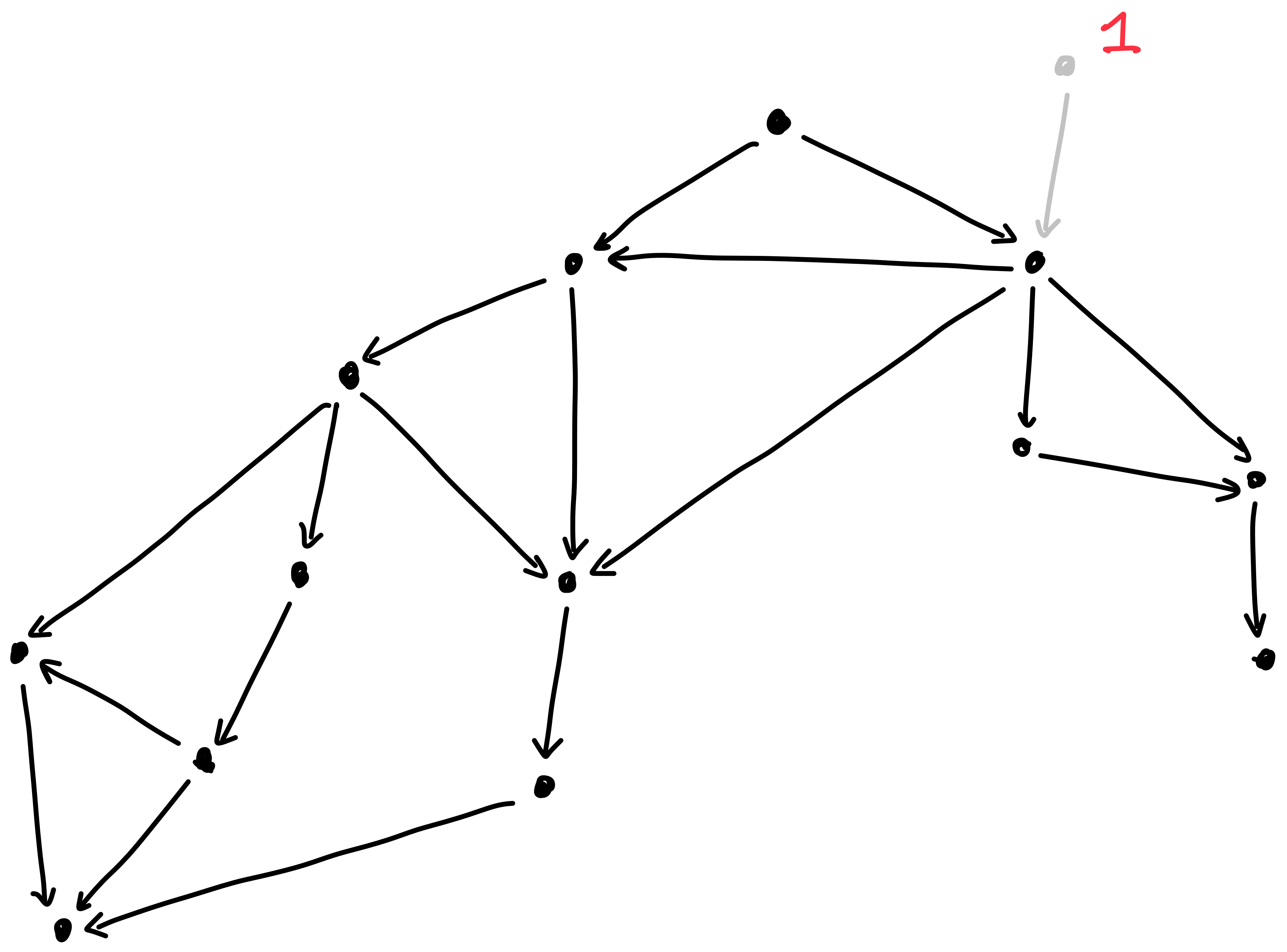
Algorithm for topological sort

- Any vertex v_1 of in-degree 0 can be numbered as 1
- Can run DFS starting from v_1
- Alternative simpler idea:
  - If we remove v_1 and assign $N(v_1) = 1$, then the rest is still a DAG
  - Then, there is a new vertex v_2 of in-degree 0
  - Repeat, until all vertices are exhausted

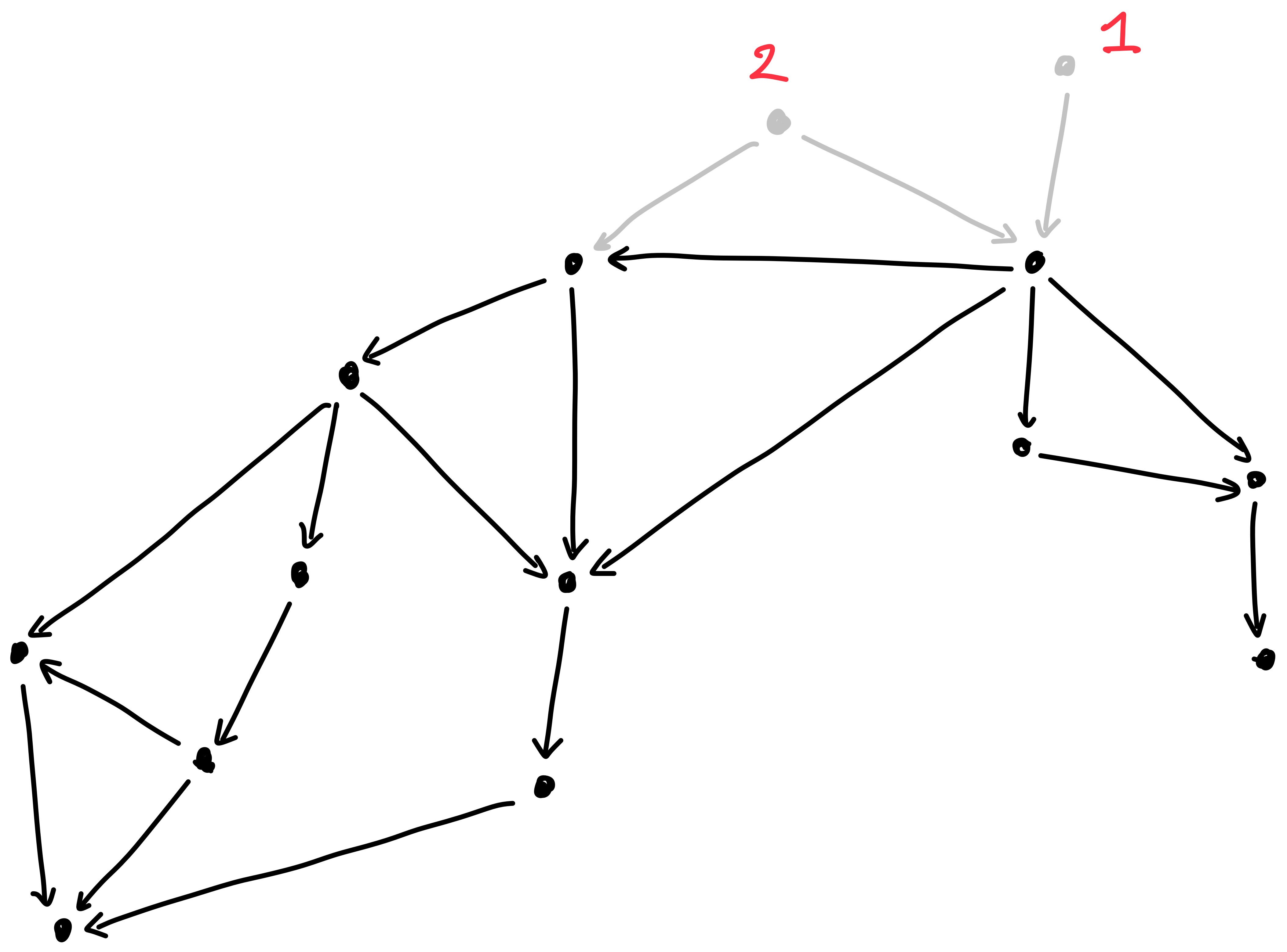
Implementing topological sort



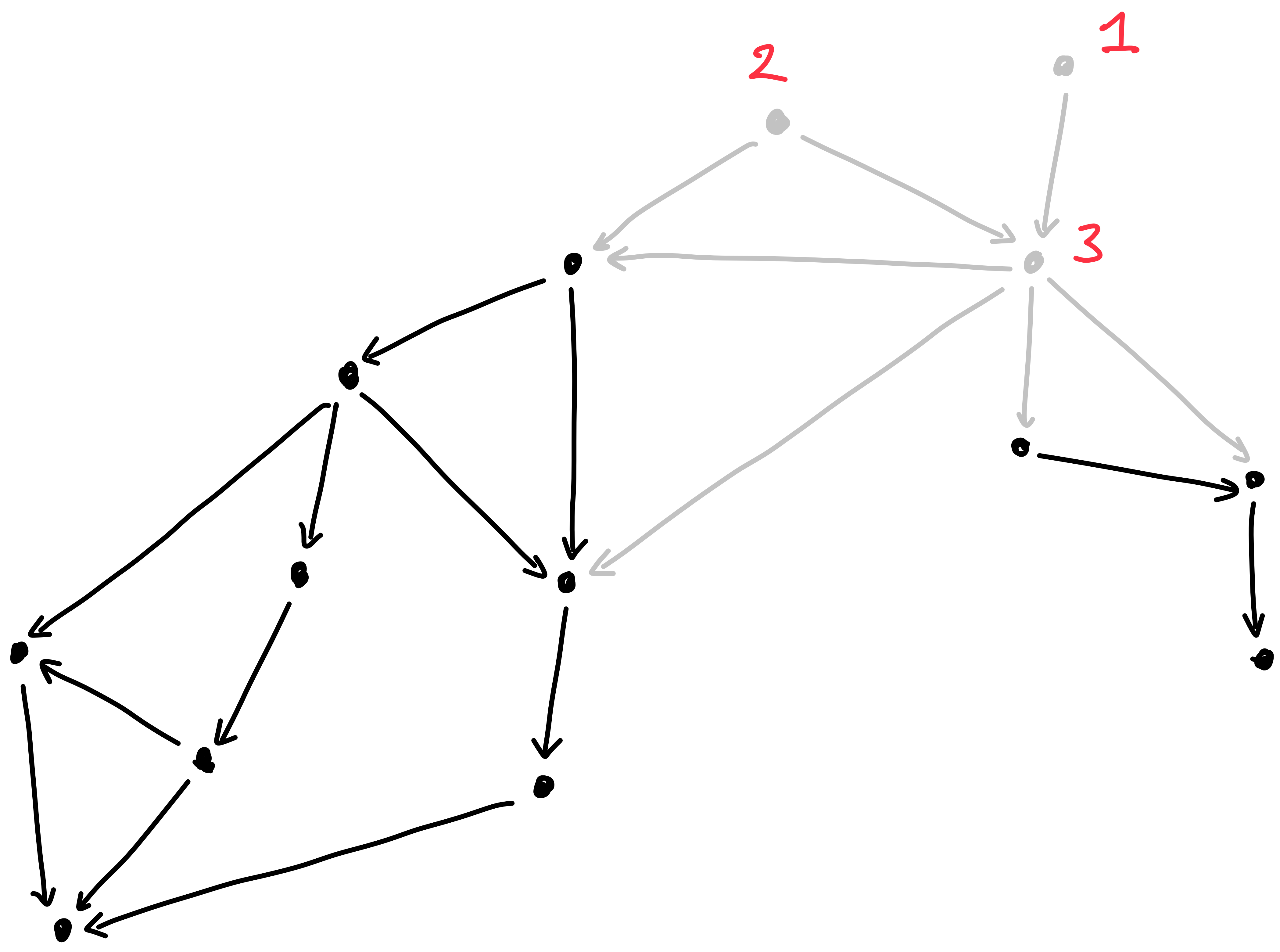
Implementing topological sort



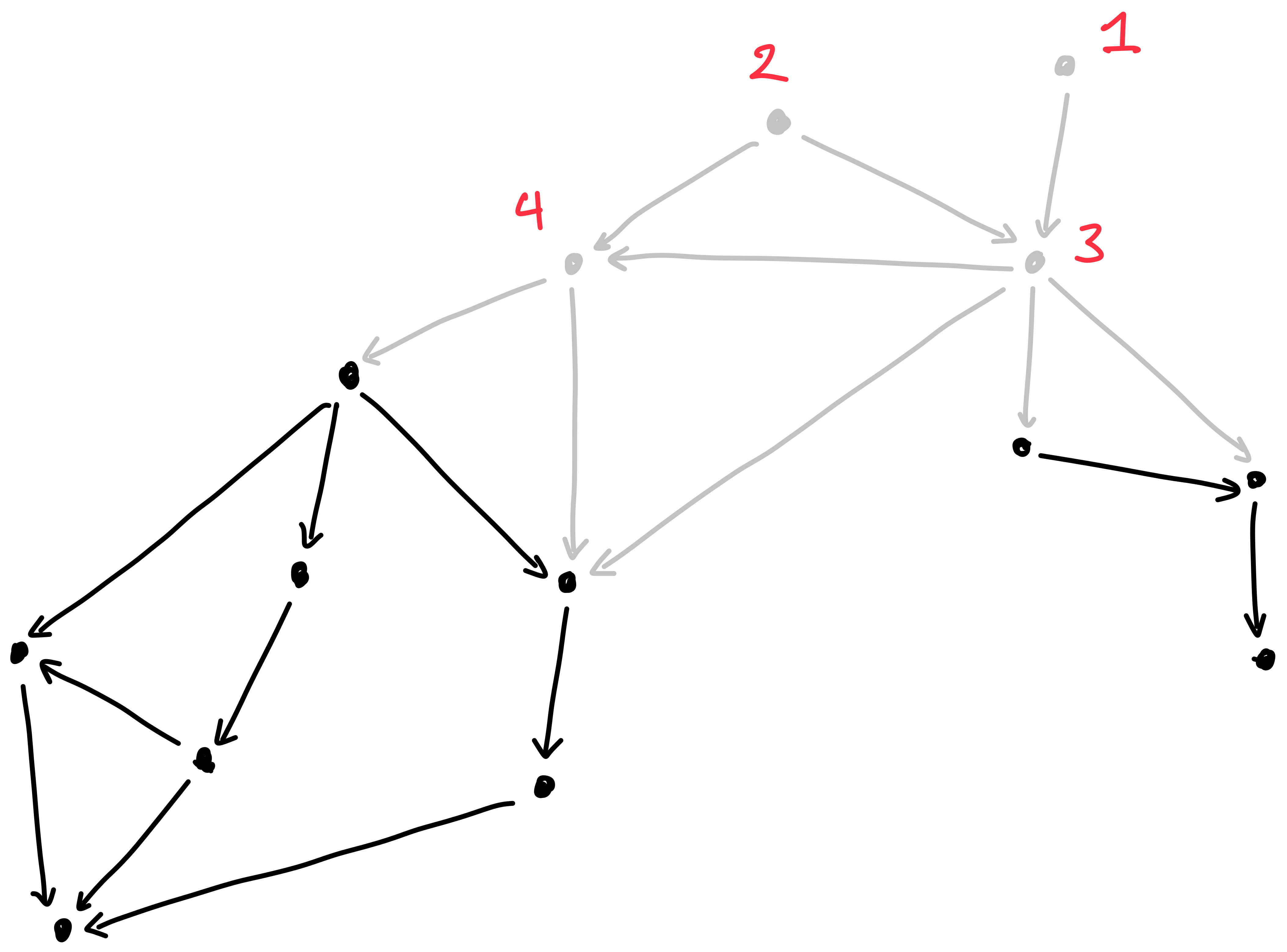
Implementing topological sort



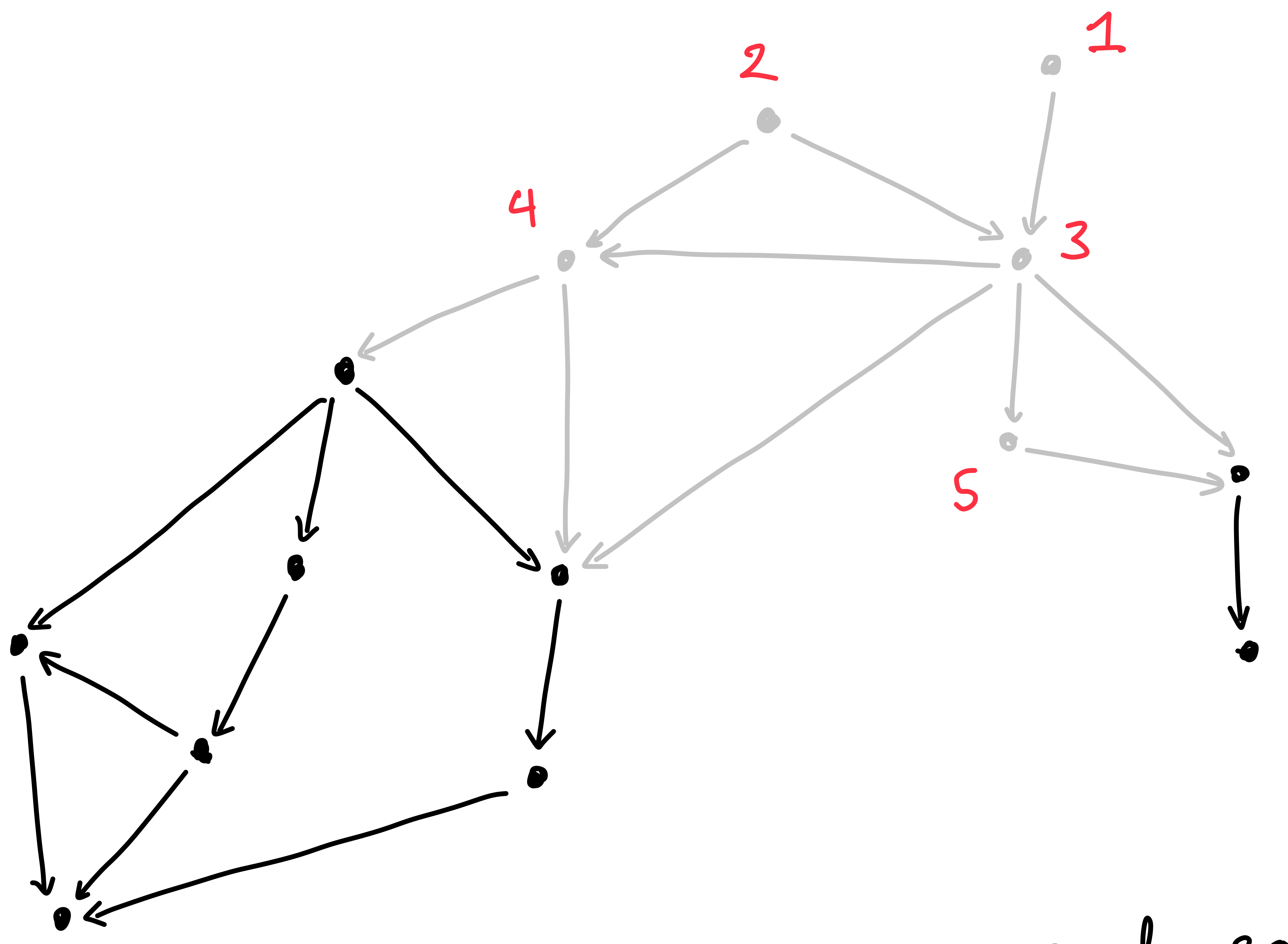
Implementing topological sort



Implementing topological sort



Implementing topological sort



and so on...

Implementing topological sort

- Issue is finding the next vertex that has in-degree 0. Can be algorithmically slow.
- Observe that when we remove the vertex v_j, the in-degree of only the out-neighbors of v_j will decrease.

Implementing topological sort

- **Algorithm:**
  - Iterate through all vertices and set $d(v) =$ in-degree of each vertex. Initialize queue Q with vertices such that $d(v) = 0$. Set $j \leftarrow 1$.
  - While Q is non-empty, pop vertex u off queue
    - Set $N(u) = j$. $j \leftarrow j + 1$.
    - Decrease $d(v) \leftarrow d(v) - 1$ for every neighbor v s.t. $u \rightarrow v$. If $d(v) = 0$, add v to Q.
  - **Runtime:** Each edge is visited only once. So $O(n + m)$ time.