Find: set $R$ of vertices reachable from $s \in V$

Reachable($s$):

1. $R \leftarrow \{s\}$
2. While there is a $(u, v) \in E$ where $u \in R$ and $v \notin R$
   - Add $v$ to $R$
3. Return $R$
Graph Traversal

- Learn the basic structure of a graph
- Walk from a fixed starting vertex $s$ to find all vertices reachable from $s$

- Three states of vertices
  - unvisited
  - visited/discovered (in $R$)
  - fully-explored (in $R$ and all neighbors in $R$)
Last time: Breadth-First Search

- Completely explore the vertices in order of their distance from \( s \)

- Naturally implemented using a queue
BFS(s)

Global initialization: mark all vertices “unvisited”

BFS(s)

- mark s “visited”; \( R \leftarrow \{s\} \); layer \( L_0 \leftarrow \{s\} \)
- while \( L_i \) not empty
  - \( L_{i+1} \leftarrow \emptyset \)
  - For each \( u \in L_i \)
    - for each edge \( \{u,v\} \)
      - if (v is “unvisited”)
        - mark v “visited”
        - Add v to set \( R \) and to layer \( L_{i+1} \)
      - mark u “fully-explored”
  - \( i \leftarrow i+1 \)
Properties of BFS

- BFS tree:
  - The rooted tree of edges followed when the BFS algorithm first discovers each vertex
  - Tree for BFS(s) gives the shortest paths (in # of hops) from s to each vertex reachable from s.

- On undirected graphs
  - All non-tree BFS edges join vertices on the same or adjacent layers
BFS Application: Shortest Paths

Tree gives shortest paths from start vertex

can label by distances from start vertex
BFS gives Bipartiteness

- Run BFS assigning all vertices from layer $L_i$ the color $i \mod 2$
  - i.e. red if they are in an even layer, blue if in an odd layer

- If there is an edge joining two vertices $u$ and $v$ from the same layer then output “Not Bipartite”
Why does it work?

\[ \text{Cycle length } 2(j-i)+1 \]

\( u \) and \( v \) have a common ancestor
Graph Search Application: Connected Components

- Want to answer questions of the form:
  - **Given**: vertices $u$ and $v$ in $G$
  - Is there a path from $u$ to $v$?

- **Idea**: create array $A$ such that $A[u] = \text{smallest numbered vertex that is connected to } u$

**Q**: Why not create an array $\text{Path}[u,v]$?
Graph Search Application: Connected Components

- initial state: all \( v \) unvisited
  for \( s \leftarrow 1 \) to \( n \) do
    if \( \text{state}(s) \neq \text{“fully-explored”} \) then
      BFS\((s)\): setting \( A[u] \leftarrow s \) for each \( u \) found
      (and marking \( u \) visited/fully-explored)
    endif
  endfor

- Total cost: \( O(n+m) \)
  - each vertex is touched once in this outer procedure and the edges examined in the different BFS runs are disjoint
  - works also with “Depth First Search”
DFS(u) – Recursive version

Global Initialization: mark all vertices "unvisited"

DFS(u)
mark u “visited” and add u to R
for each edge \{u,v\}
    if (v is “unvisited”)
        DFS(v)
end for
mark u “fully-explored”
Properties of DFS(s)

- Like BFS(s):
  - DFS(s) visits \( x \) if and only if there is a path in \( G \) from \( s \) to \( x \)
  - Edges into undiscovered vertices define a "depth first spanning tree" of \( G \)

- Unlike the BFS tree:
  - the DFS spanning tree isn't minimum depth
  - its levels don't reflect min distance from the root
  - non-tree edges never join vertices on the same or adjacent levels

- BUT...
Non-tree edges

- All non-tree edges join a vertex and one of its descendants/ancestors in the DFS tree.

- No cross edges.
No cross edges in DFS on undirected graphs

- **Claim:** During $\text{DFS}(x)$ every vertex marked visited is a descendant of $x$ in the DFS tree $T$

- **Claim:** For every $x, y$ in the DFS tree $T$, if $(x, y)$ is an edge not in $T$ then one of $x$ or $y$ is an ancestor of the other in $T$

- **Proof:**
  - One of $x$ or $y$ is visited first, suppose WLOG that $x$ is visited first and therefore $\text{DFS}(x)$ was called before $\text{DFS}(y)$
    - During $\text{DFS}(x)$, the edge $(x, y)$ is examined
  - Since $(x, y)$ is a not an edge of $T$, $y$ was visited when the edge $(x, y)$ was examined during $\text{DFS}(x)$
  - Therefore $y$ was visited during the call to $\text{DFS}(x)$ so $y$ is a descendant of $x$. 
DFS(v) for a directed graph
DFS(v)

- Tree edges
- Forward edges
- Back edges
- Cross edges

1. Start DFS at vertex v.
2. Visit each vertex in the order they are visited.
3. Mark each vertex as visited.
4. For each unvisited neighbor, call DFS recursively.
5. Backtrack when a cross edge is encountered.
Properties of Directed DFS

- Before $\text{DFS}(s)$ returns, it visits all previously unvisited vertices reachable via directed paths from $s$

- Every cycle contains a back edge in the DFS tree
Identify the Strongly Connected Components

Recall: \( u \) and \( v \) are in the same strongly connected component iff there is a path from \( u \) to \( v \) and from \( v \) to \( u \).
Strongly connected components can be found in $O(n+m)$ time

- But it’s tricky!
- Simpler problem: given a vertex $v$, compute the vertices in $v$’s scc in $O(n+m)$ time
Strongly Connected Components

- Tree edges
- Forward edges
- Back edges
- Cross edges

- Node 1
- Node 2
- Node 3
- Node 4
- Node 5
- Node 6
- Node 7
- Node 8
- Node 9
- Node 10
- Node 11
- Node 12
- Node 13

NO → cross edges
← cross edges
Directed Acyclic Graphs

- A directed graph $G=(V,E)$ is **acyclic** if it has no directed cycles.

- **Terminology**: A directed acyclic graph is also called a **DAG**.
Topological Sort

- **Given:** a directed acyclic graph (DAG) $G = (V, E)$
- **Output:** numbering of the vertices of $G$ with distinct numbers from 1 to $n$ so edges only go from lower number to higher numbered vertices

**Applications**
- nodes represent tasks
- edges represent precedence between tasks
- topological sort gives a sequential schedule for solving them
Directed Acyclic Graph
In-degree 0 vertices

- Every DAG has a vertex of in-degree 0
- **Proof:** By contradiction
  - Suppose every vertex has some incoming edge
  - Consider following procedure:
    - `while (true) do`
    - `v ← some predecessor of v`
  - After \( n+1 \) steps where \( n = |V| \) there will be a repeated vertex
    - This yields a cycle, contradicting that it is a DAG
Topological Sort

- Can do using DFS

- Alternative simpler idea:
  - Any vertex of in-degree 0 can be given number 1 to start
  - Remove it from the graph and then give a vertex of in-degree 0 number 2, etc.
Topological Sort
Topological Sort
Topological Sort
Topological Sort
Topological Sort
Topological Sort
Topological Sort
Topological Sort
Topological Sort

Diagram of a directed acyclic graph with nodes and directed edges showing a topological order.
Topological Sort
Topological Sort
Implementing Topological Sort

- Go through all edges, computing array with in-degree for each vertex \( O(m+n) \)
- Maintain a queue (or stack) of vertices of in-degree 0
- Remove any vertex in queue and number it
- When a vertex is removed, decrease in-degree of each of its neighbors by 1 and add them to the queue if their degree drops to 0

Total cost \( O(m+n) \)
Strongly Connected Components & Topological Sort work well together

- Shrinking each s.c.c. of directed graph to a vertex yields a DAG
- Can apply topological sort to the resulting DAG to get a good order to deal with the s.c.c.’s of the original graph one at a time