Object-oriented programming and data-structures

CS/ENGRD 2110 SUMMER 2018

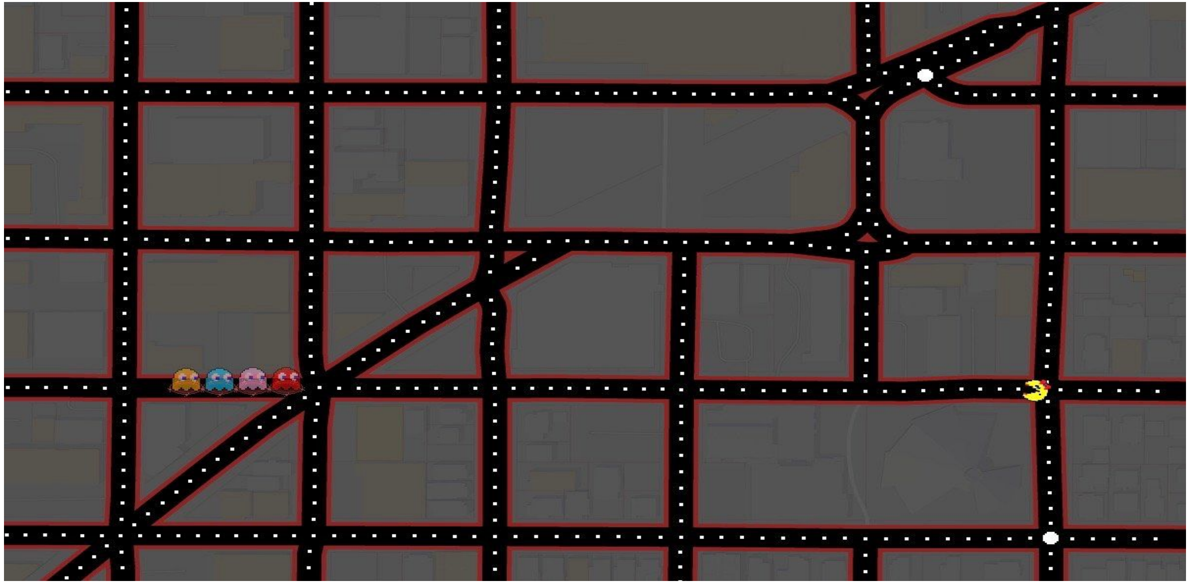
Lecture 12: Graphs Search

http://courses.cs.cornell.edu/cs2110/2018su

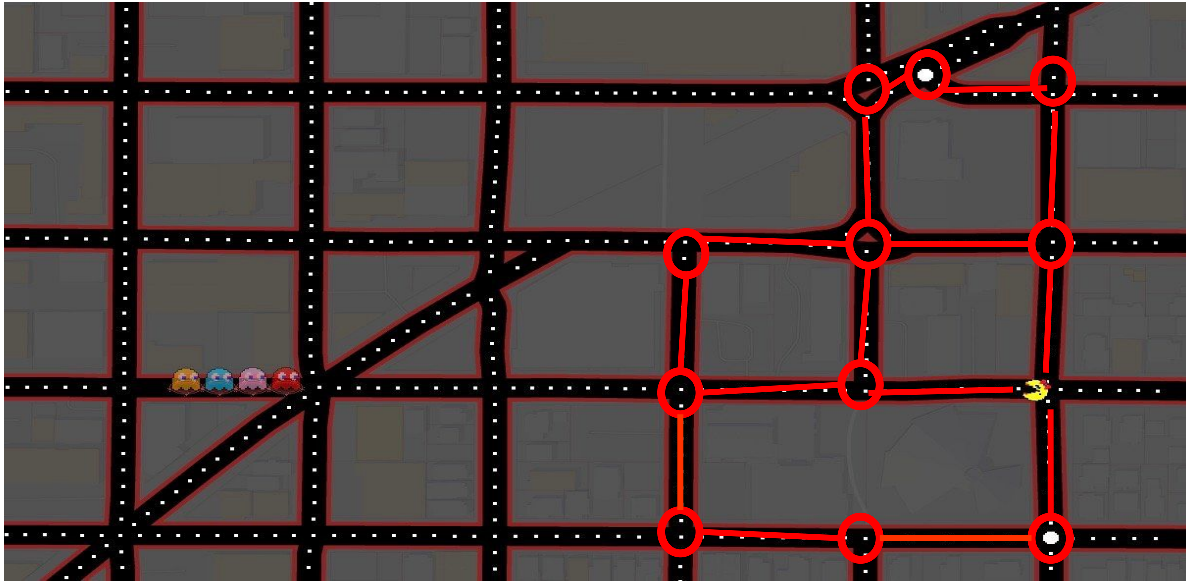
Graph Algorithms

- Search
  - Depth-first search
  - Breadth-first search
- Shortest paths
  - Dijkstra's algorithm
- Spanning trees
  - Algorithms based on properties
  - Minimum spanning trees
  - Prim's algorithm

Search (Again)

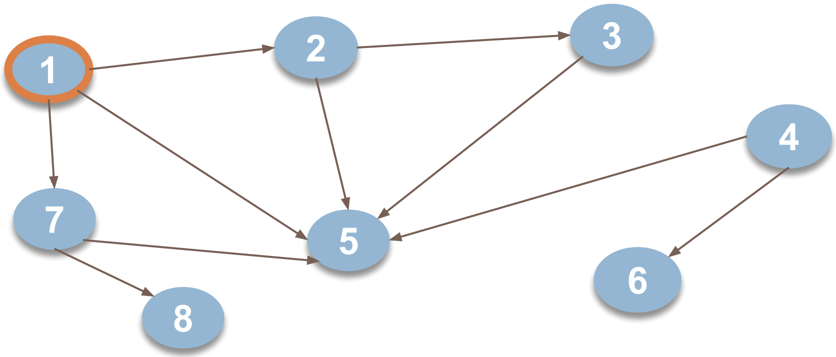


Search (Again)



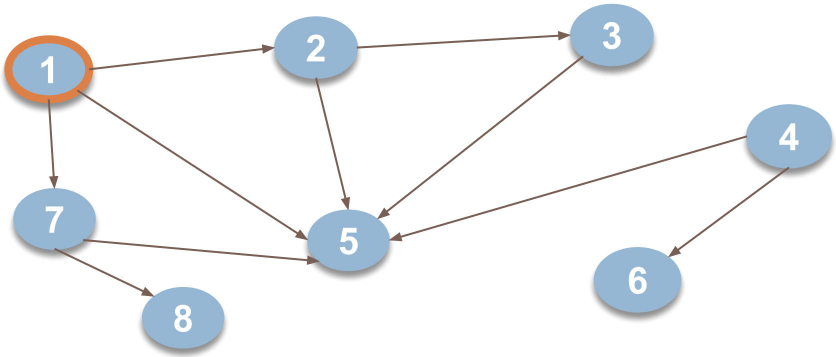
Search on Graphs

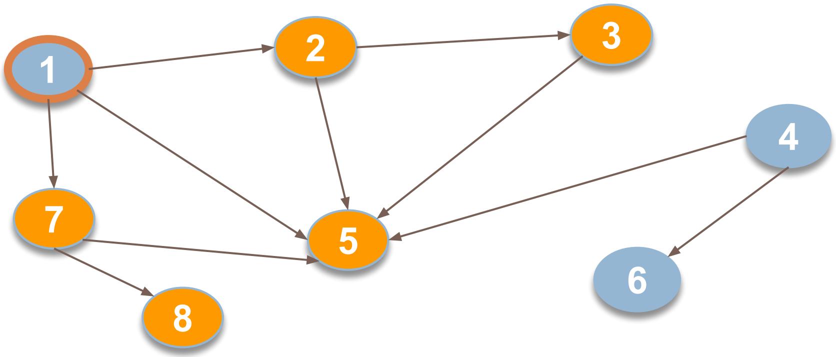
- Given a graph (V,E) and a vertex u ∈ V, want to visit every node that is reachable from u



Search on Graphs

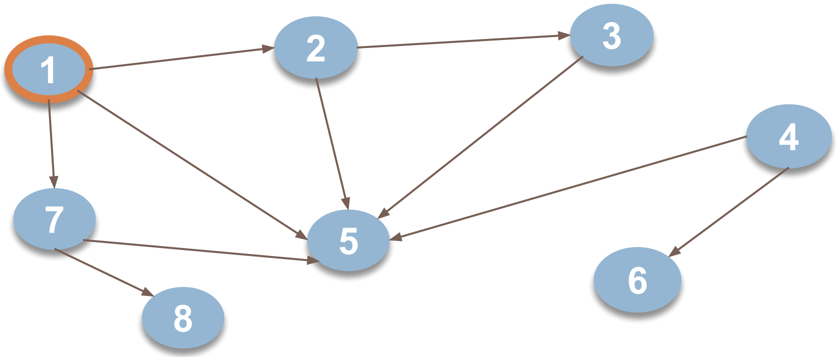
- Given a graph (V,E) and a vertex u ∈ V, want to visit every node that is reachable from u





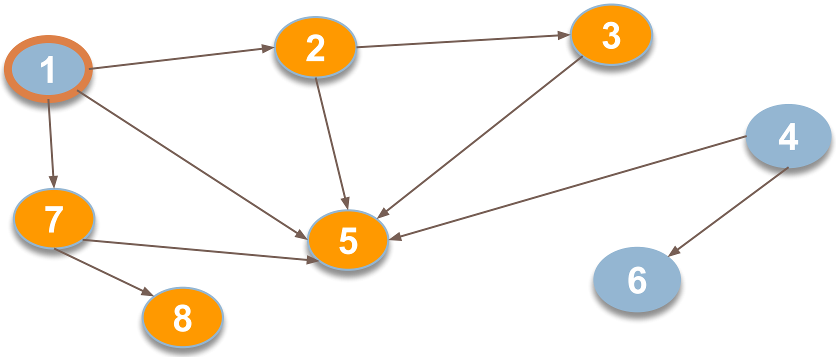
Search on Graphs

- Given a graph (V,E) and a vertex u ∈ V, want to visit every node that is reachable from u



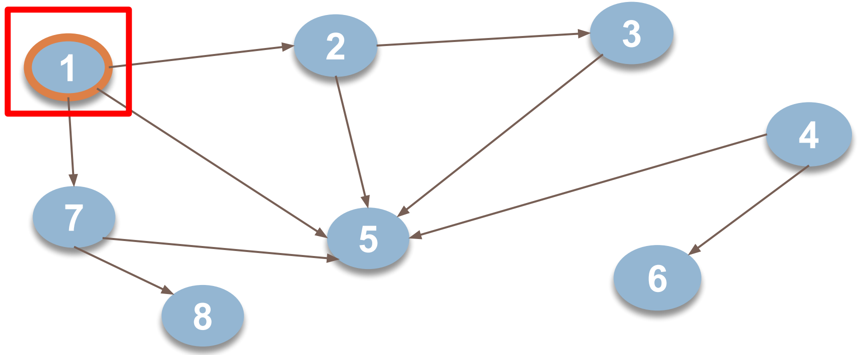
There are many paths to some nodes.

How do we visit all nodes efficiently, without doing extra work?



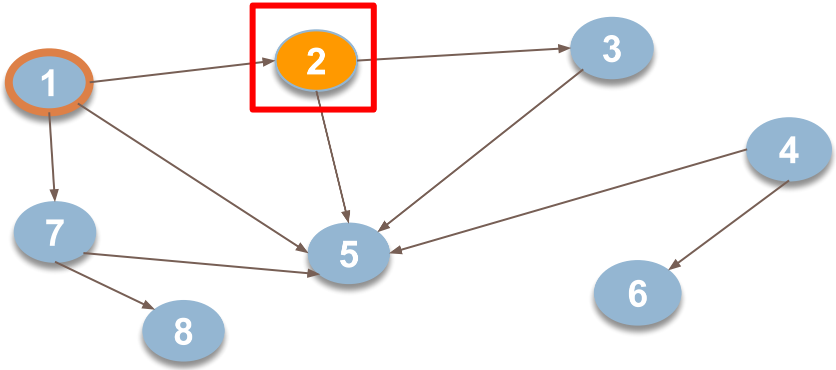
Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



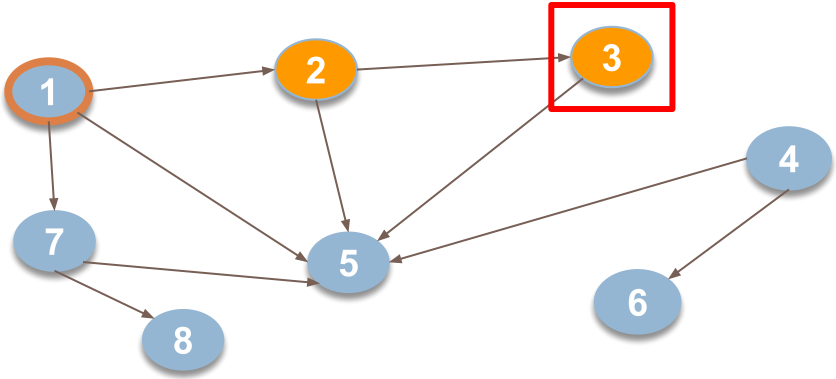
Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



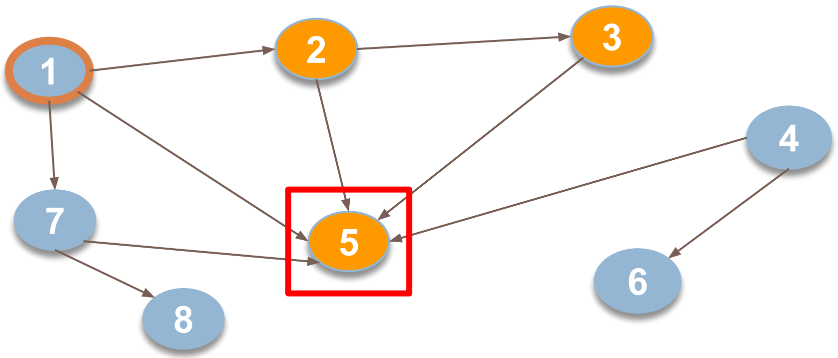
Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



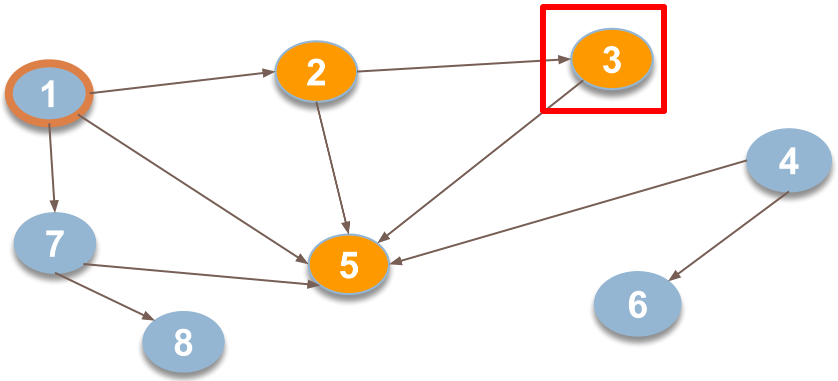
Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



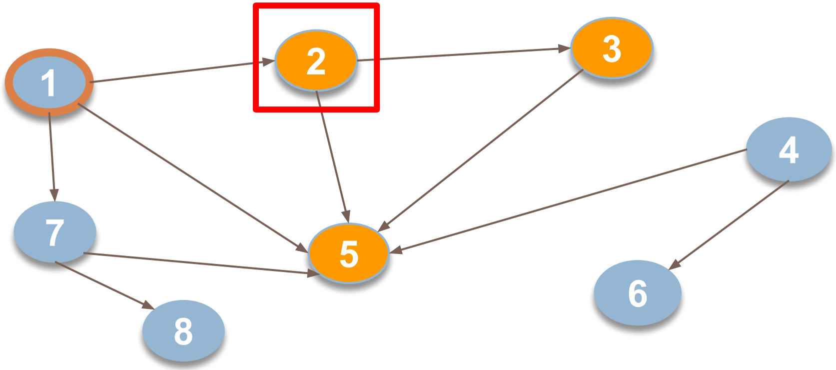
Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



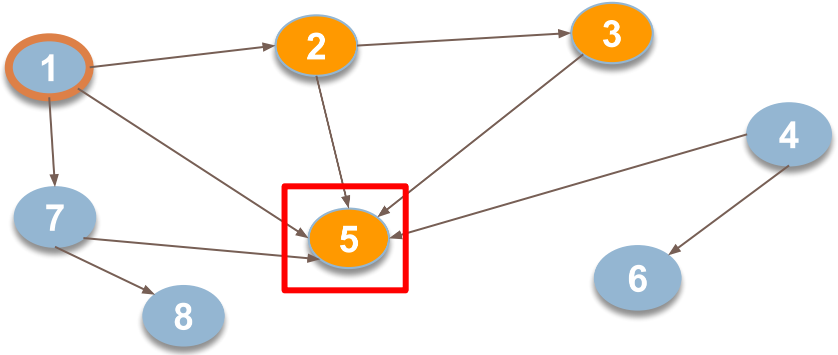
Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



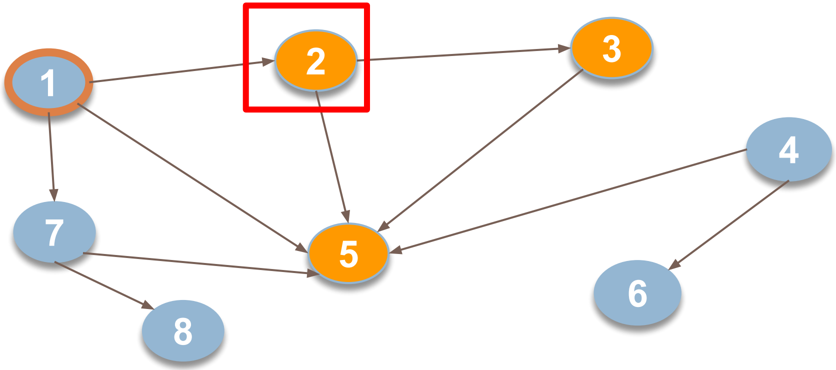
Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



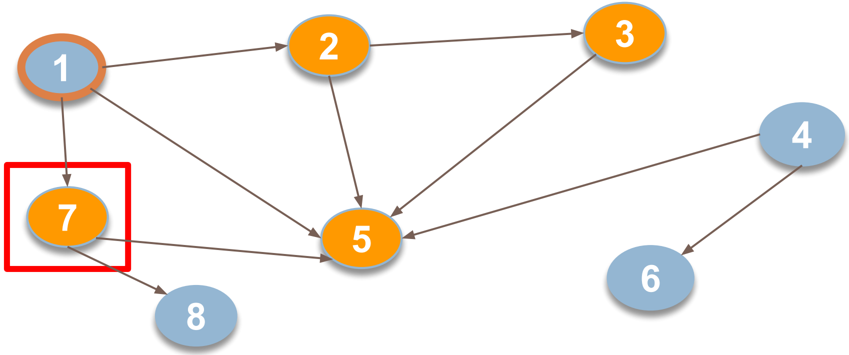
Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



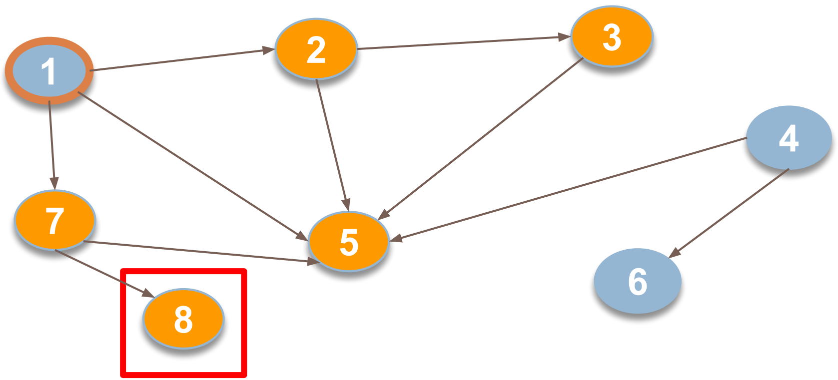
Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



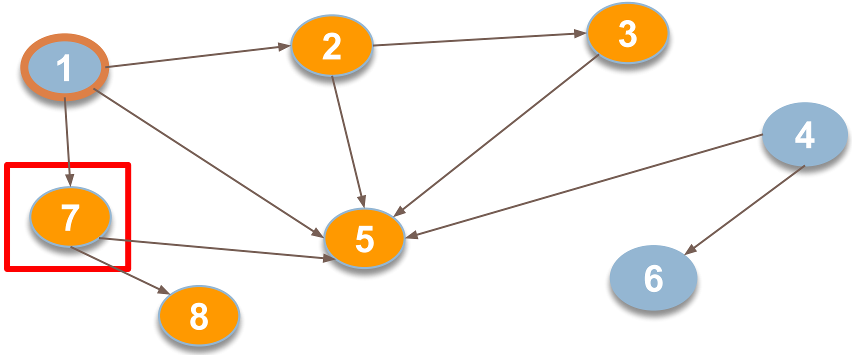
Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



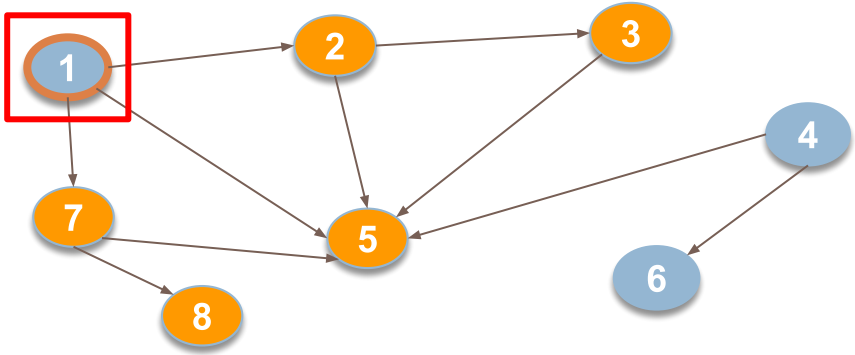
Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



Depth-First Search

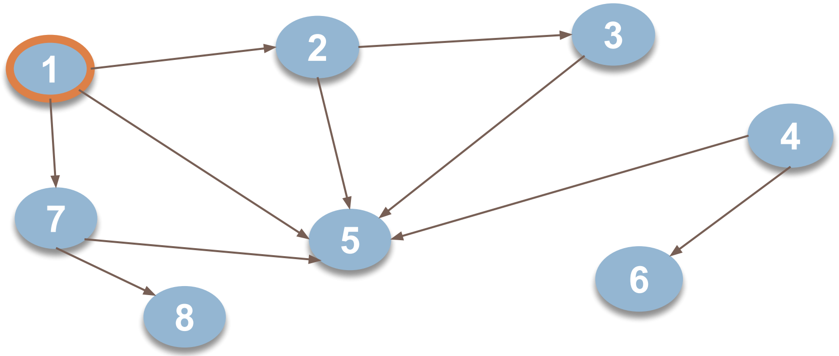
Intuition: Recursively visit all vertices that are reachable along unvisited paths.



Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.

```
/** Visit all nodes reachable on unvisited paths
from u.
Precondition: u is unvisited. */
public static void dfs(int u) {
    visit(u);
    for all edges (u,v):
        if(!visited[v]):
            dfs(v);
}
```



dfs(1) visits the nodes in this order: 1, 2, 3, 5, 7, 8

Depth-First Search in Java

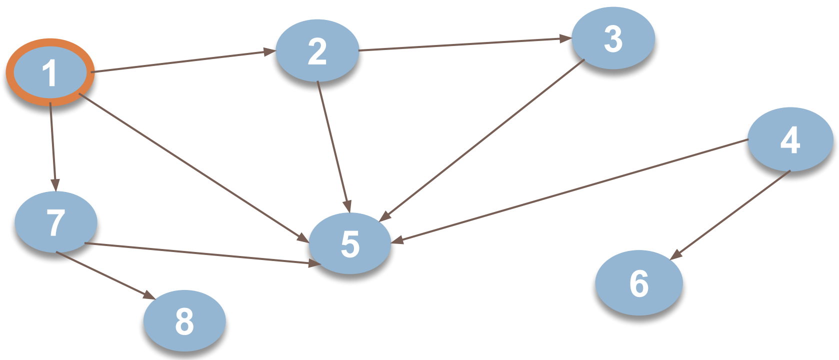

```
public class Node {
    boolean visited;
    List<Node> neighbours;

/** Visit all nodes reachable on unvisited paths from this node.
Precondition: this node is unvisited. */
    public void dfs() {
        visited= true;
        for (Node n: neighbours) {
            if (!n.visited) n.dfs();
        }
    }
}
```

Each vertex of the graph is an object of type Node

No need for a parameter. The object is the node.

Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.

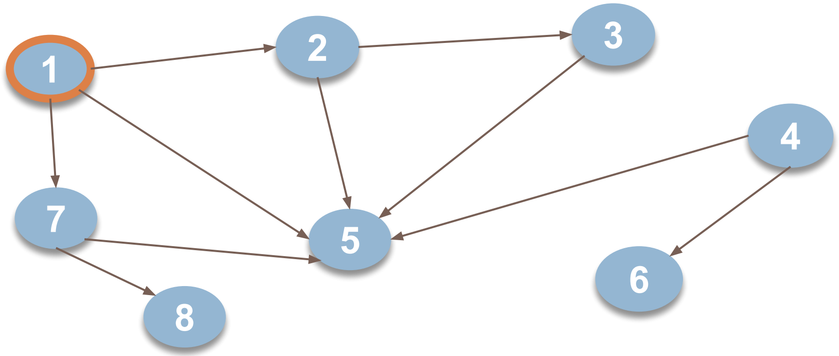


dfs(1) visits the nodes in this order: 1, 2, 3, 5, 7, 8

Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.

Suppose there are n vertices that are reachable along unvisited paths, and m edges



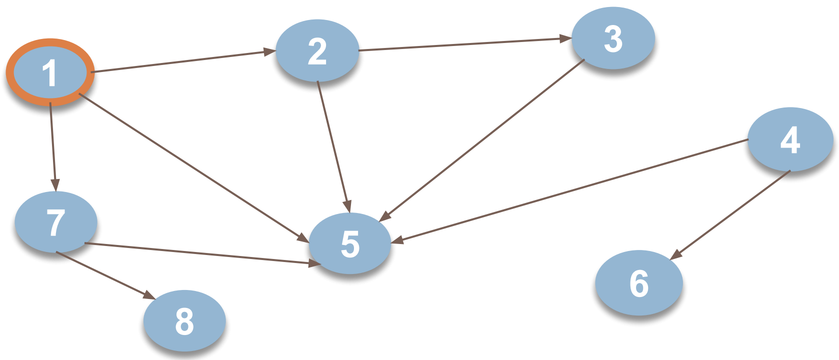
dfs(1) visits the nodes in this order: 1, 2, 3, 5, 7, 8

Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.

Suppose there are n vertices that are reachable along unvisited paths, and m edges

Visits every vertex in the graph exactly once and every edge exactly once



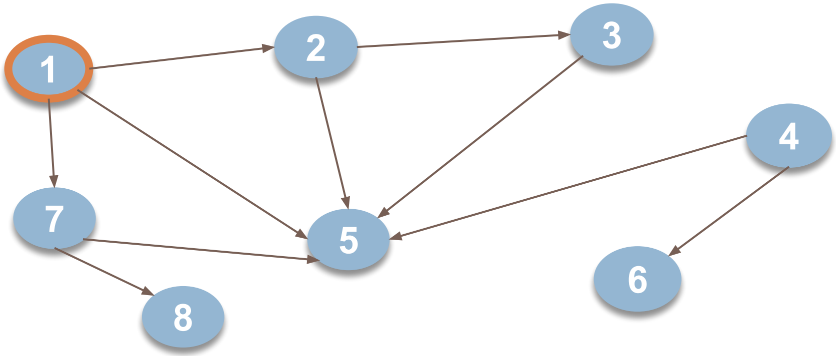
dfs(1) visits the nodes in this order: 1, 2, 3, 5, 7, 8

Depth-First Search

Intuition: Recursively visit all vertices that are reachable along unvisited paths.

Suppose there are n vertices that are reachable along unvisited paths, and m edges

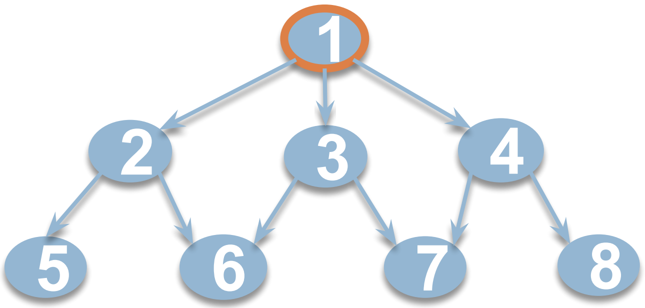
Worst-case time complexity:
O(n + m)



dfs(1) visits the nodes in this order: 1, 2, 3, 5, 7, 8

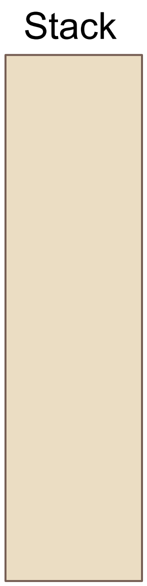
DFS Quiz

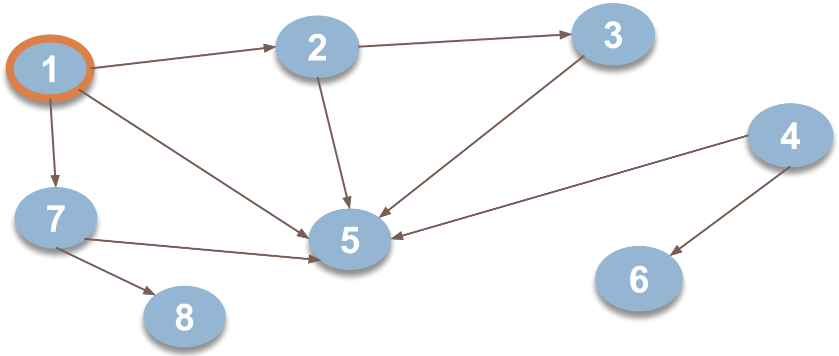
- In what order would a DFS visit the vertices of this graph? Break ties by visiting the lower-numbered vertex first.
  - 1, 2, 3, 4, 5, 6, 7, 8
  - 1, 2, 5, 6, 3, 6, 7, 4, 7, 8
  - 1, 2, 5, 3, 6, 4, 7, 8
  - 1, 2, 5, 6, 3, 7, 4, 8



Depth-First Search Iteratively

Intuition: Recursively visit all vertices that are reachable along unvisited paths.

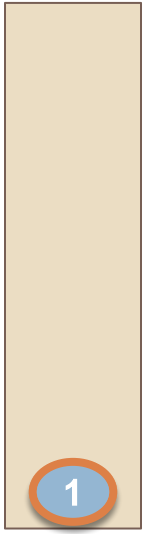


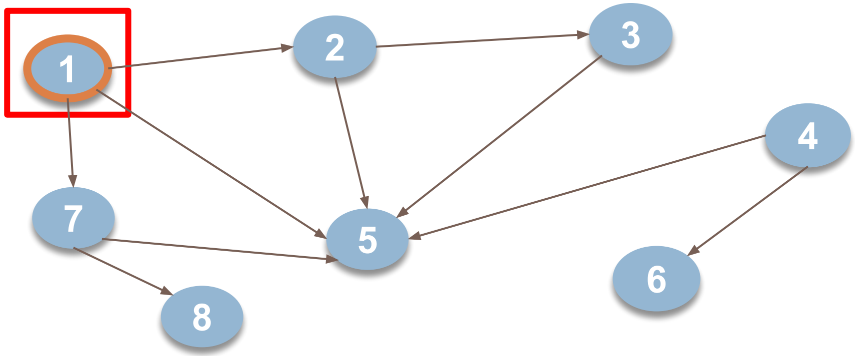


Depth-First Search Iteratively

Intuition: Recursively visit all vertices that are reachable along unvisited paths.

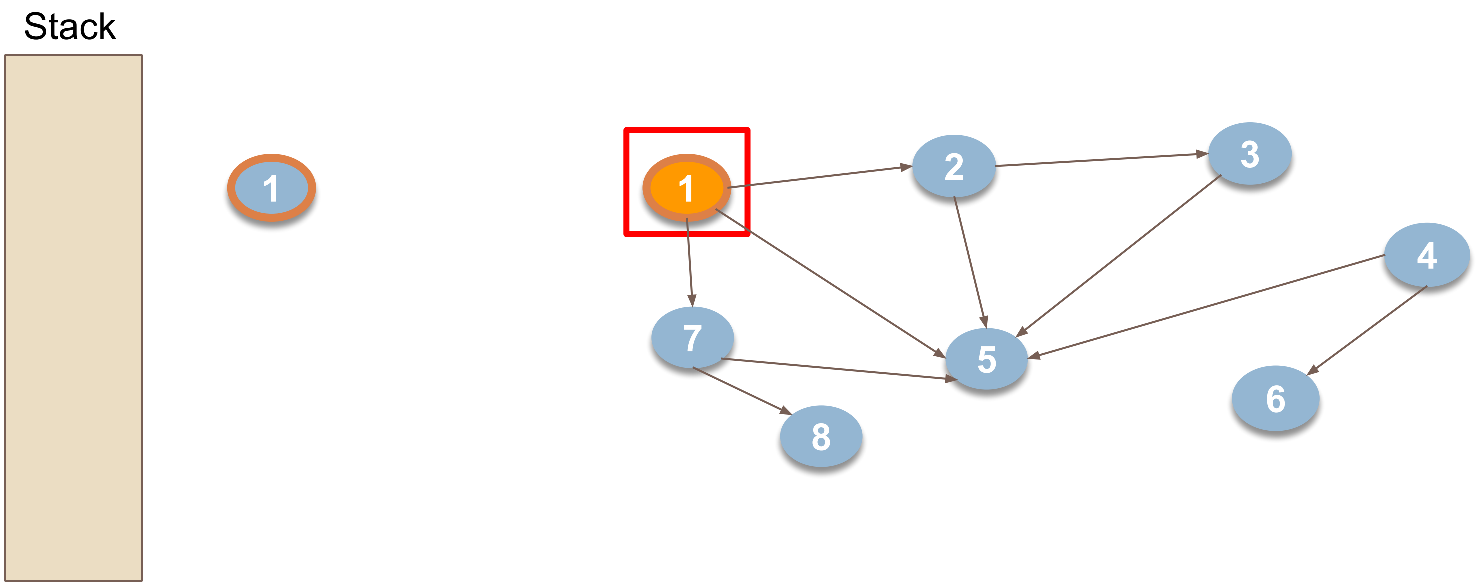
Stack





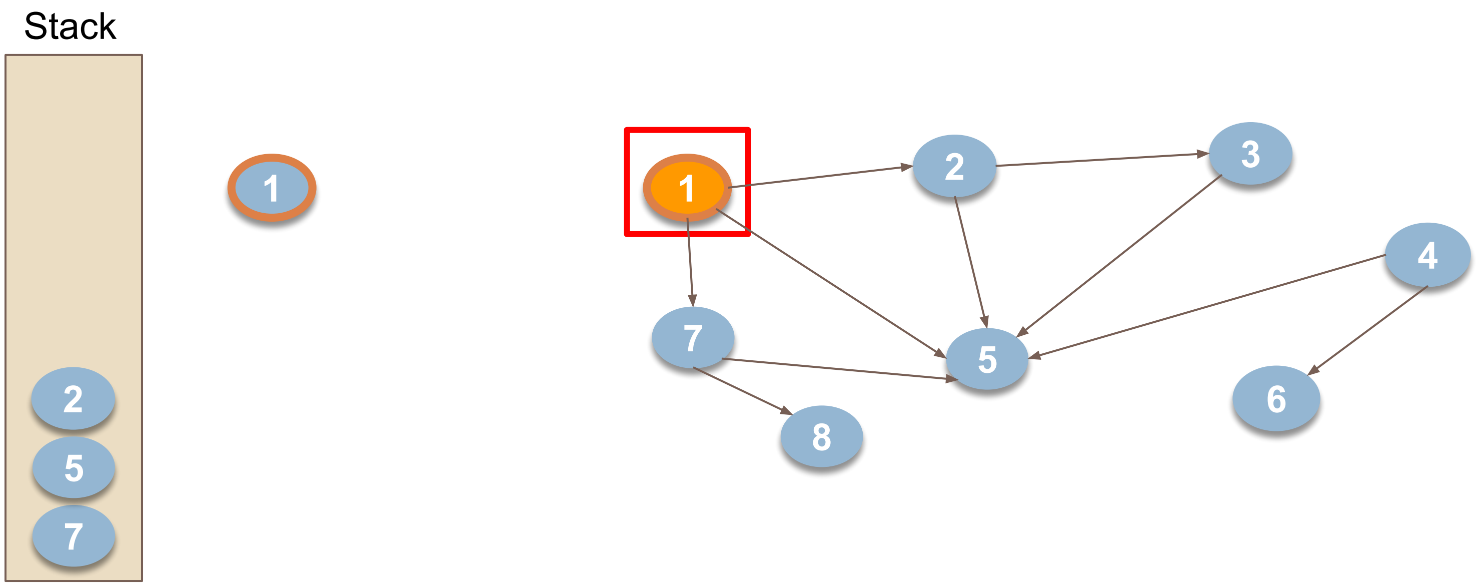
Depth-First Search Iteratively

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



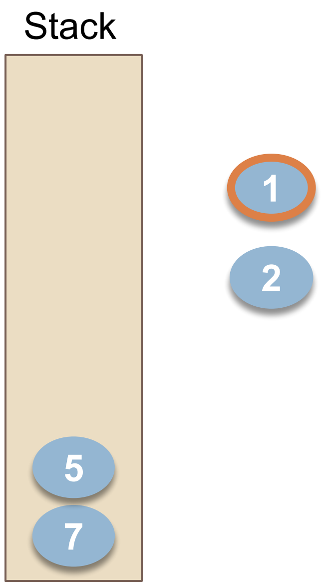
Depth-First Search Iteratively

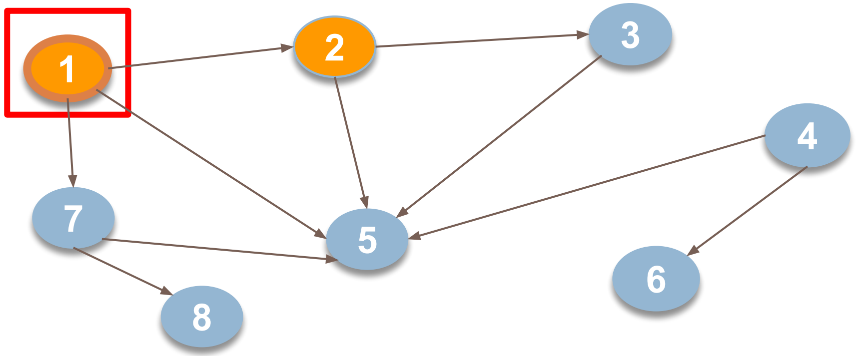
Intuition: Recursively visit all vertices that are reachable along unvisited paths.



Depth-First Search Iteratively

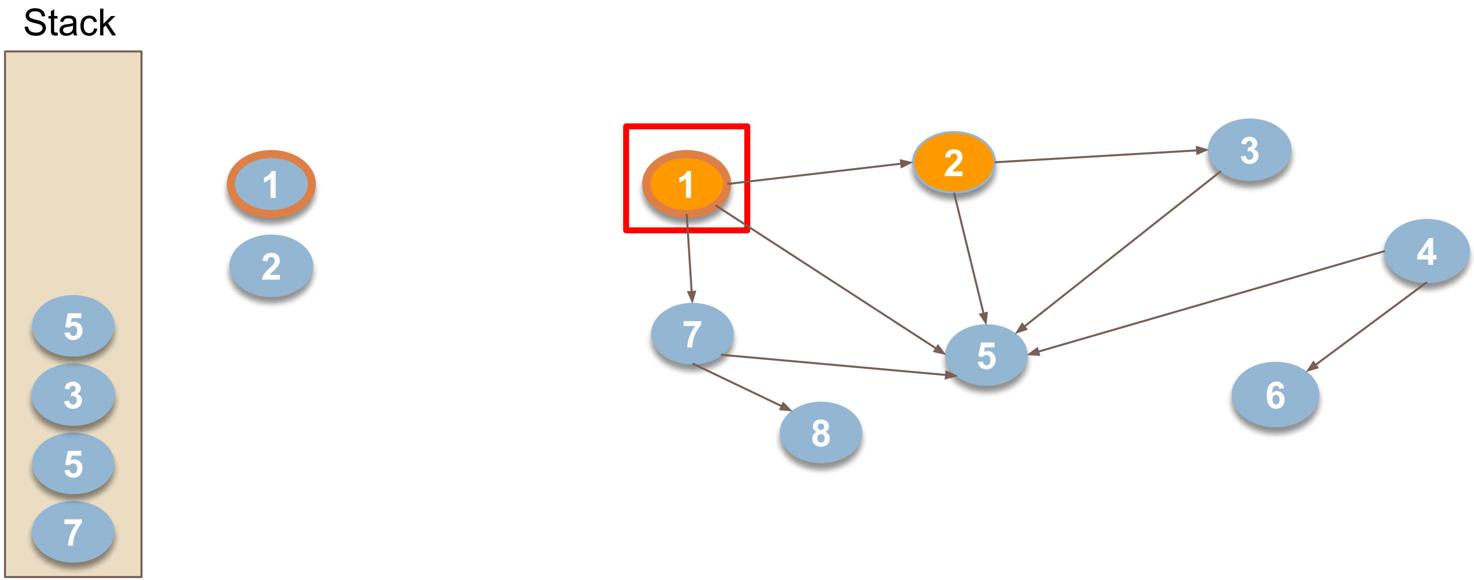
Intuition: Recursively visit all vertices that are reachable along unvisited paths.





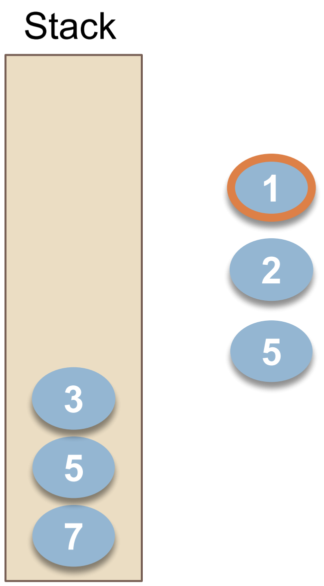
Depth-First Search Iteratively

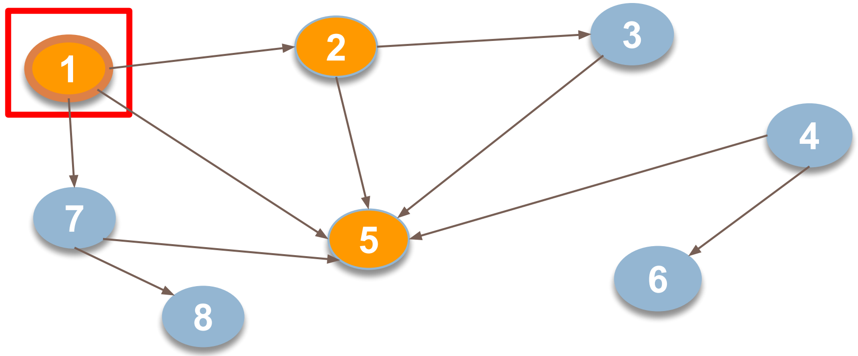
Intuition: Recursively visit all vertices that are reachable along unvisited paths.



Depth-First Search Iteratively

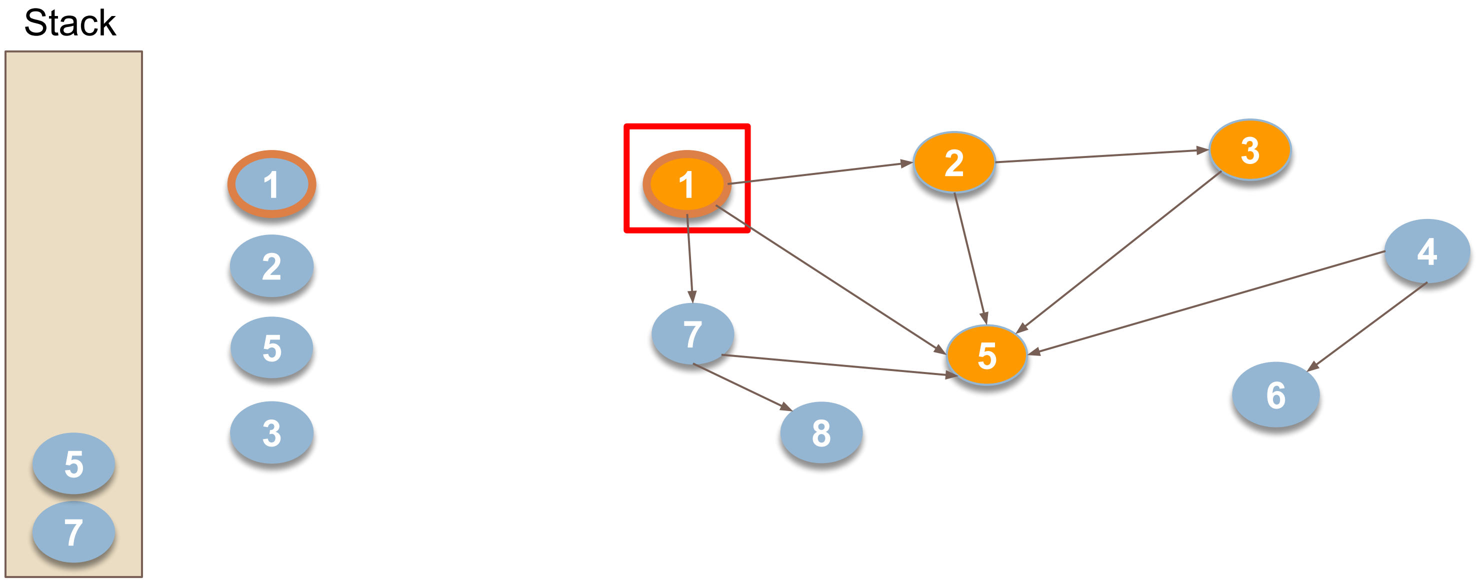
Intuition: Recursively visit all vertices that are reachable along unvisited paths.





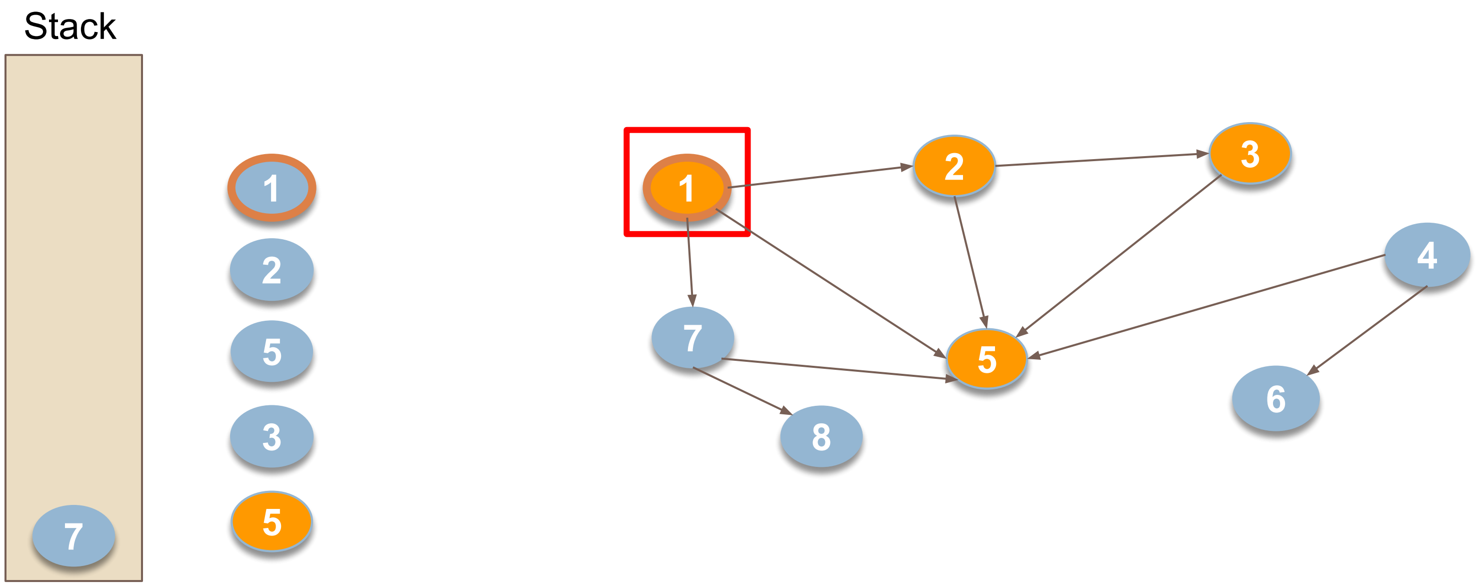
Depth-First Search Iteratively

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



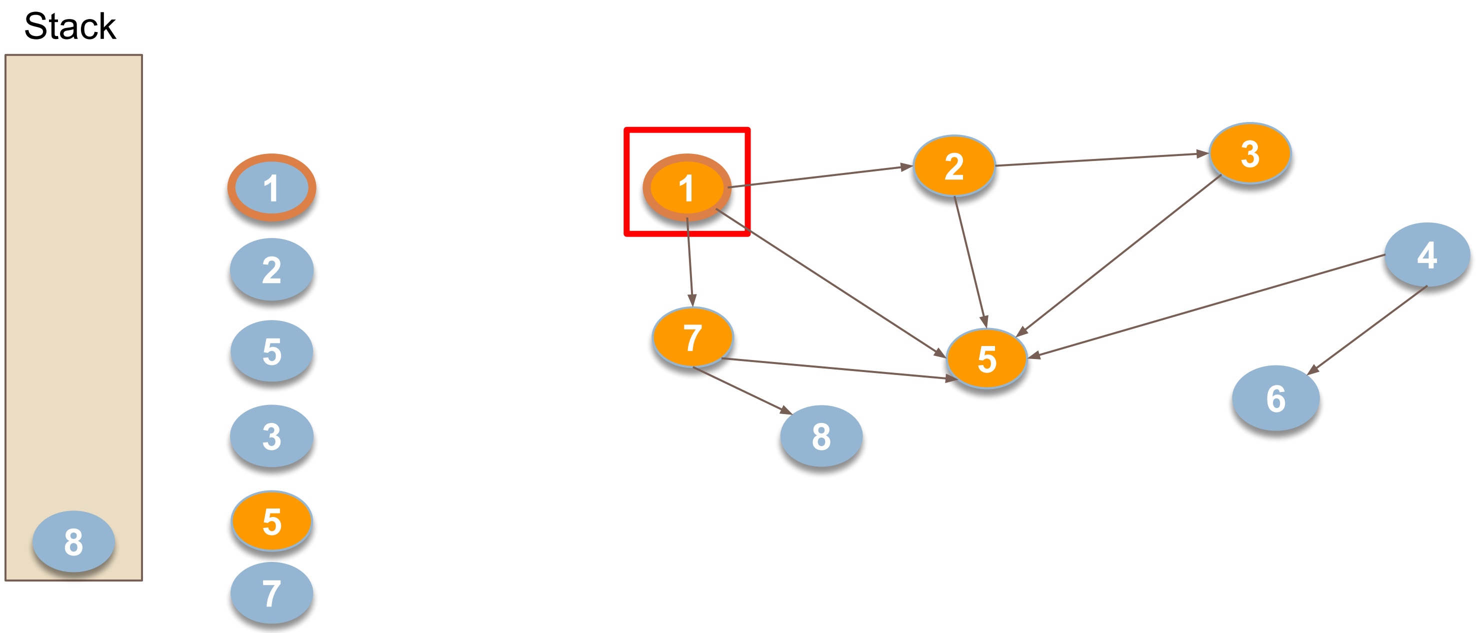
Depth-First Search Iteratively

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



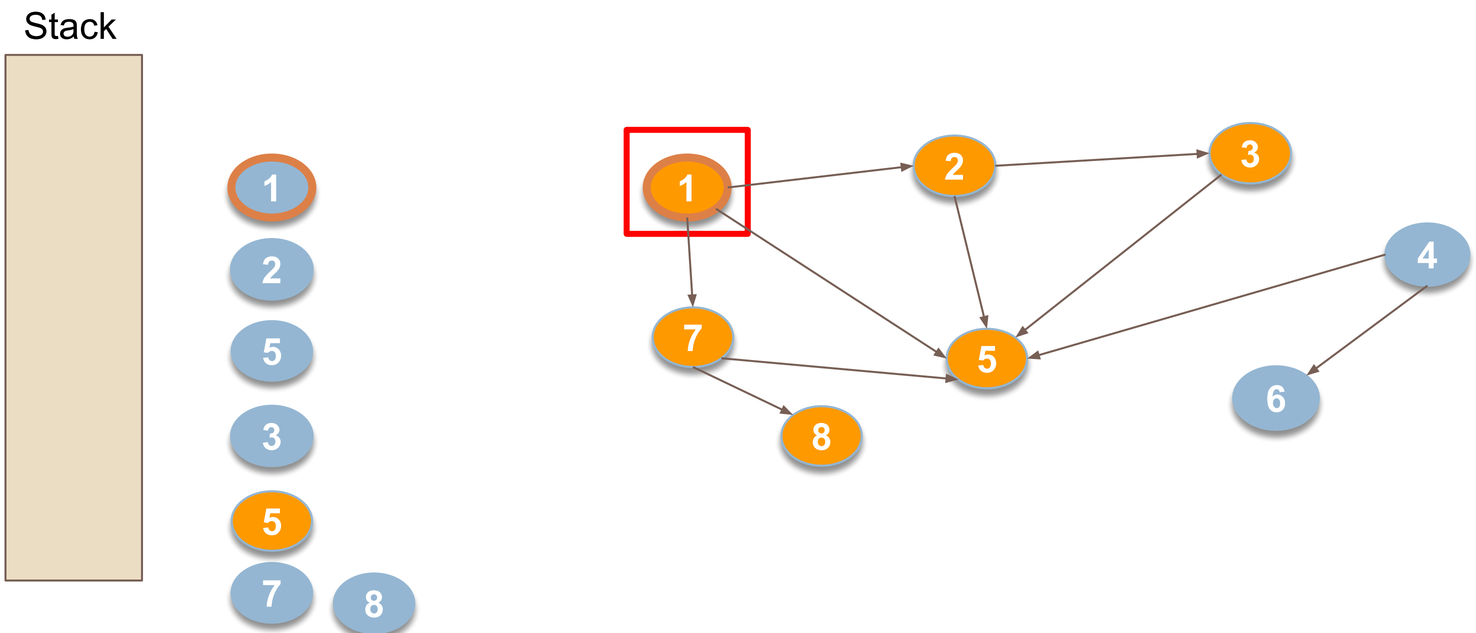
Depth-First Search Iteratively

Intuition: Recursively visit all vertices that are reachable along unvisited paths.



Depth-First Search Iteratively

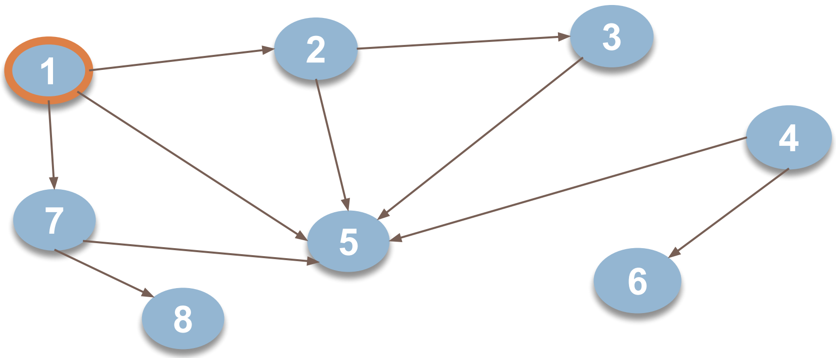
Intuition: Recursively visit all vertices that are reachable along unvisited paths.



Depth-First Search Iteratively

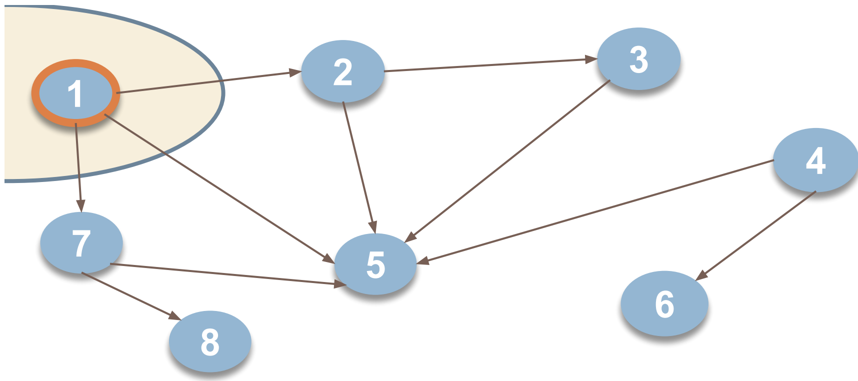
Intuition: Visit all vertices that are reachable along unvisited paths from the current node.

```
/** Visit all nodes reachable on unvisited paths from u.
Precondition: u is unvisited. */
public static void dfs(int u) {
    Stack s= (u);// Not Java!
    while (s is not empty) {
        u= s.pop();
        if (u not visited) {
            visit u;
            for each edge (u, v):
                s.push(v);
        }
    }
}
```



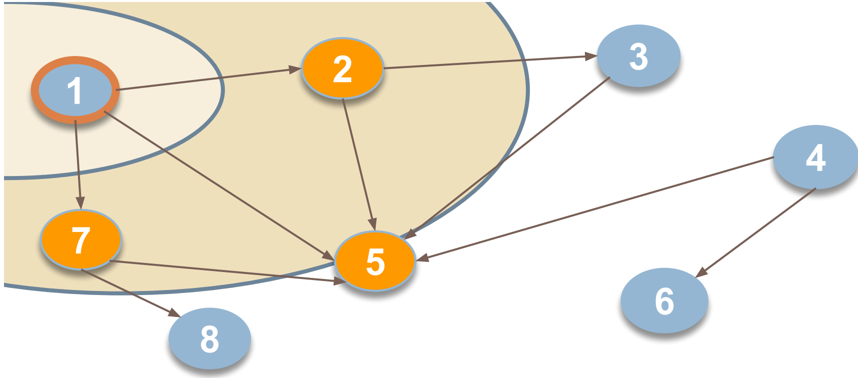
Breadth-First Search

Intuition: Iteratively process the graph in "layers" moving further away from the source node.



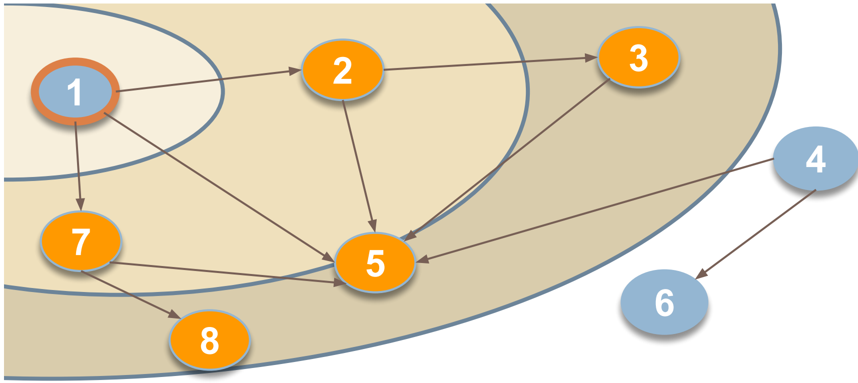
Breadth-First Search

Intuition: Iteratively process the graph in "layers" moving further away from the source node.



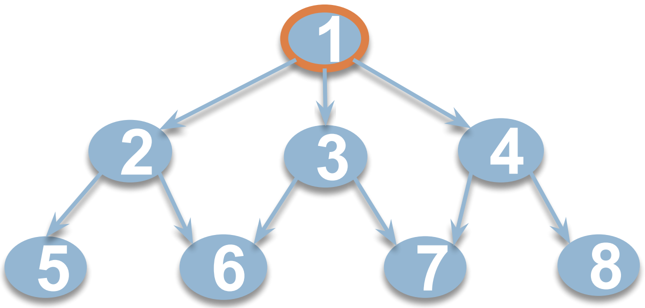
Breadth-First Search

Intuition: Iteratively process the graph in "layers" moving further away from the source node.



BFS Quiz

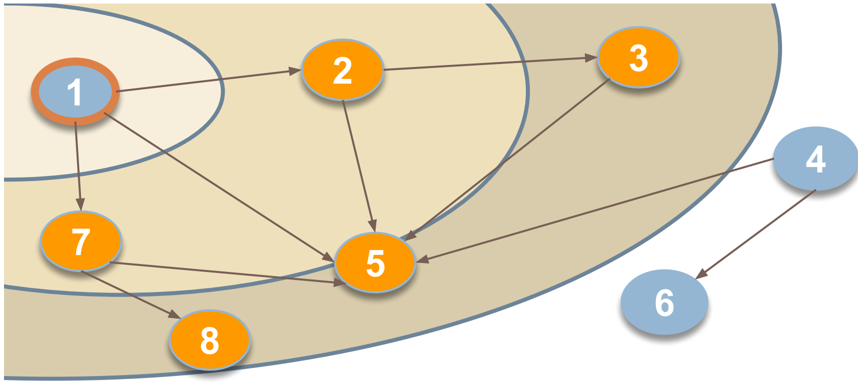
- In what order would a BFS visit the vertices of this graph? Break ties by visiting the lower-numbered vertex first.
  - 1, 2, 3, 4, 5, 6, 7, 8
  - 1, 2, 3, 4, 5, 6, 6, 7, 7, 8
  - 1, 2, 5, 3, 6, 4, 7, 8
  - 1, 2, 5, 6, 3, 7, 4, 8



Breadth-First Search

Intuition: Iteratively process the graph in "layers" moving further away from the source node.

```
/** Visit all nodes reachable on
unvisited paths from u.
Precondition: u is unvisited. */
public static void bfs(int u) {
    Queue q= (u);// Not Java!
    while ( q is not empty ) {
        u= q.remove();
        if (u not visited) {
            visit u;
          for each (u, v):
                q.add(v);
          }
     }
}
```

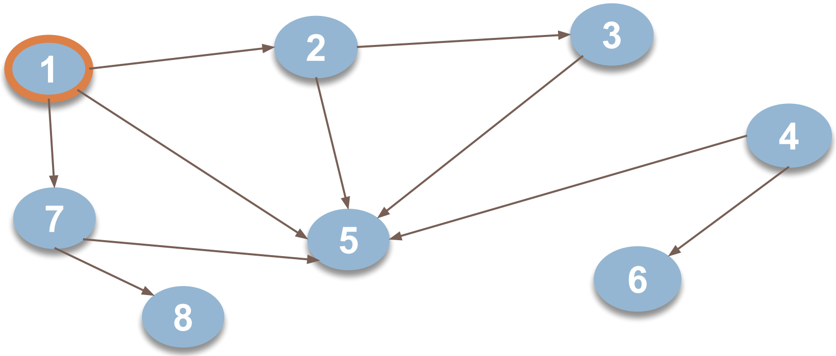


Analysing BFS

Intuition: Iteratively process the graph in "layers" moving further away from the source node.

Suppose there are n vertices that are reachable along unvisited paths, and m edges

Worst-case time complexity:
O(n + m)



bfs(1) visits the nodes in this order: 1, 2, 7, 3, 5, 8

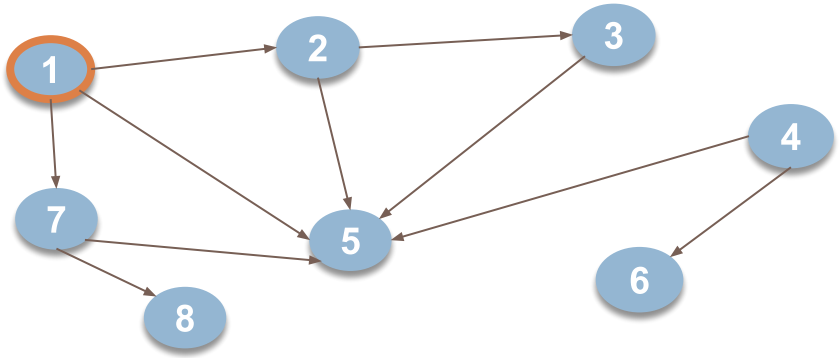
Comparing Search Algorithms

DFS

- Visits: 1,2,3,5,7,8
- Time: O(n + m)
- Space: O(n)

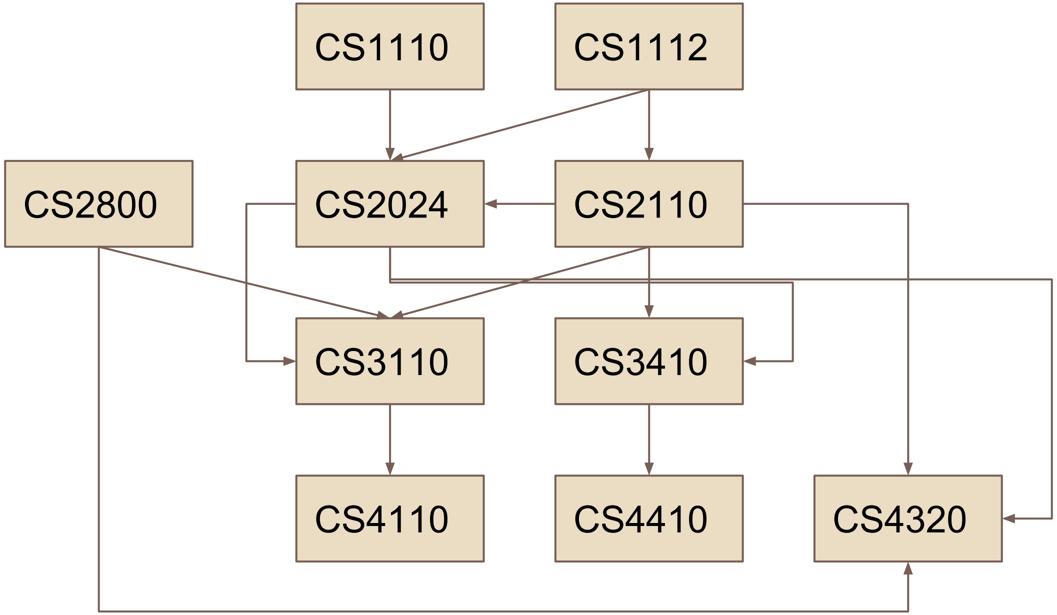
BFS

- Visits: 1,2,5,7,3,8
- Time: O(n + m)
- Space: O(n)



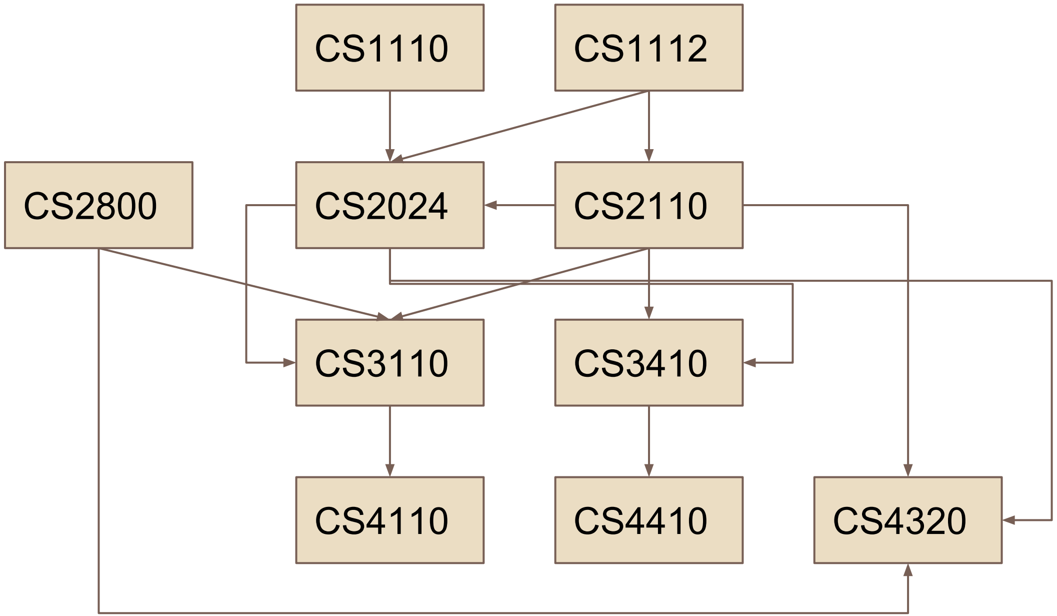
Topological Sort

- Problem: In what order should I take CS classes at Cornell?



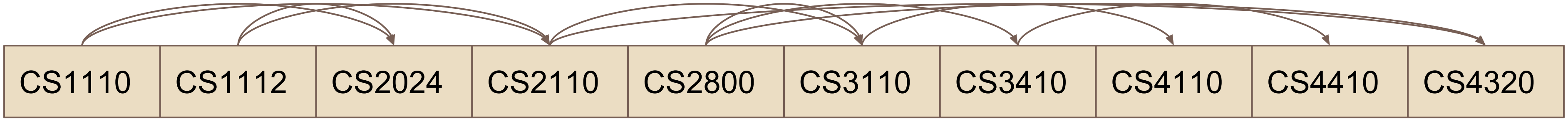
Topological Sort

- Can I get a **linear ordering** of the graph such that all courses that are prereqs happen before courses that are not



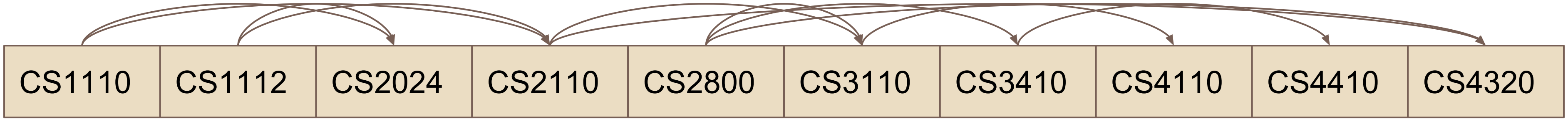
Topological Sort

- Can I get a **linear ordering** of the graph such that all courses that are prereqs happen before courses that are not



Topological Sort

- Can I get a **linear ordering** of the graph such that all courses that are prereqs happen before courses that are not



- Graphically: can I arrange all the nodes such that edges all point to the right?

Topological Sort, Formally

- A topological sort of a graph **G** is a linear ordering of all its vertices such that i
  - if G contains an edge (u,v) then u appears before v in the ordering.

Topological Sort, Formally

- A topological sort of a graph **G** is a linear ordering of all its vertices such that i
  - if G contains an edge (u,v) then u appears before v in the ordering.

- Can be computed efficiently using DFS

Topological Sort

- Let's revisit our DFS algorithm
  - Every node has a **discovery time u**
    - The time when we mark it as visited for the first time
  - Every node has a **finishing time f**
    - The time when we explore the last of its edge

Topological Sort


```
public class Node {
      boolean visited; List<Node> neighbours;
      int discoveryTime; int finishingTime;

      public void dfs() {
            visited= true;
            discoveringTime = time;
            for (Node n: neighbours) {
                  if (!n.visited) n.dfs();
            }
            time++;
            finishingTime = time;
      }
}
```

Topological Sort

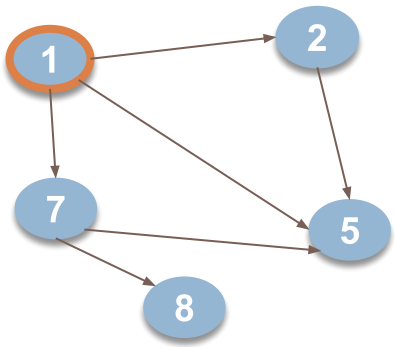
- Revisit DFS as follows:
  - For every node u in G, run u.dfs();
  - As each vertex is finished, insert it into the front of a linked list
  - Return the linked list of vertices

Topological Sort

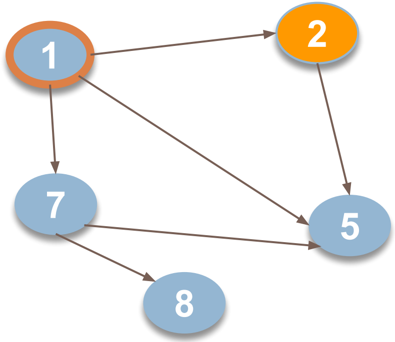
- Revisit DFS as follows:
  - For every node u in G, run u.dfs();
  - As each vertex is finished, insert it into the front of a linked list
  - Return the linked list of vertices

- Key idea: inserting a vertex in front of the list when finished ensures that vertices v with an edge (u,v) always appear **before** vertices v in the linked list (as they will marked as finished after v)

Topological Sort

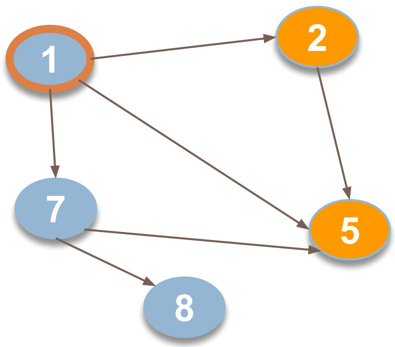


Topological Sort



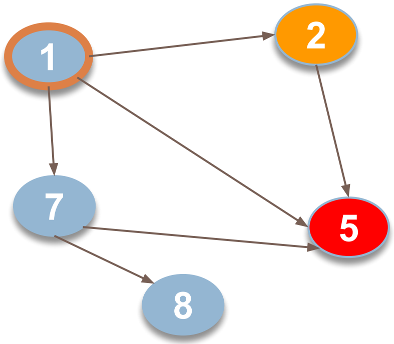
Time = 2

Topological Sort



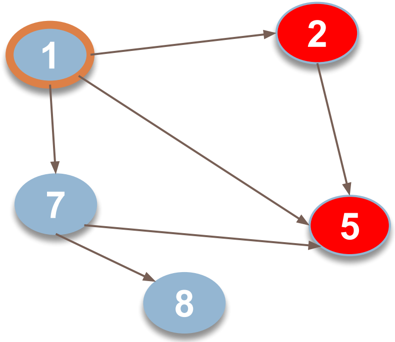
Time = 3

Topological Sort

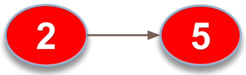


Time = 4

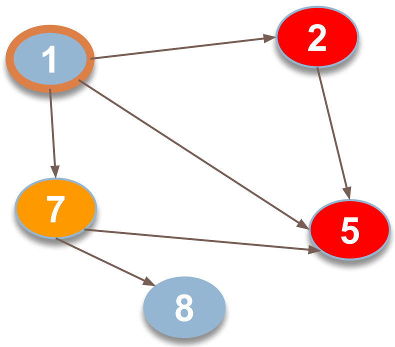
Topological Sort



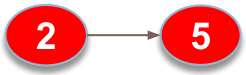
Time = 5



Topological Sort

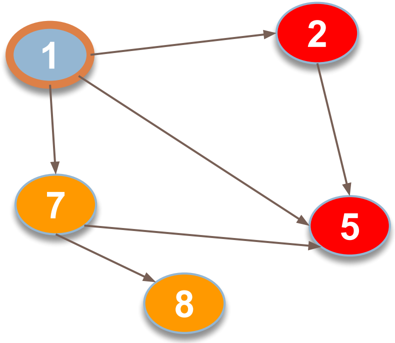


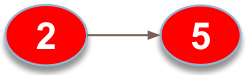
Time = 6



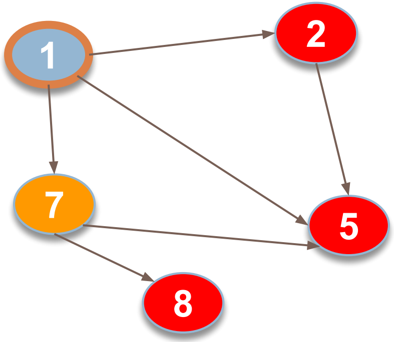
Topological Sort

Time = 7





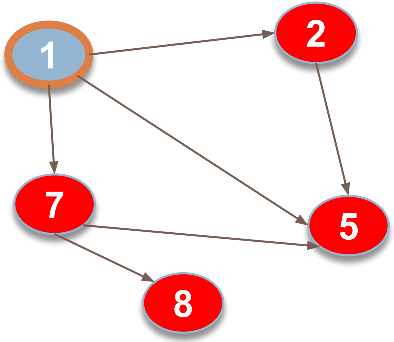
Topological Sort

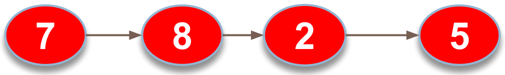


Time = 8

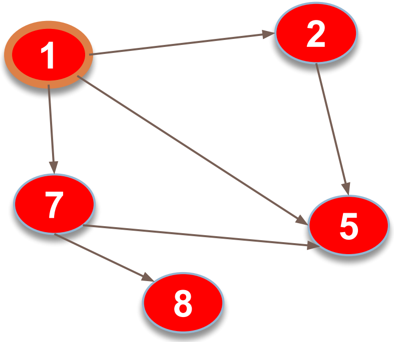
Topological Sort

Time = 9





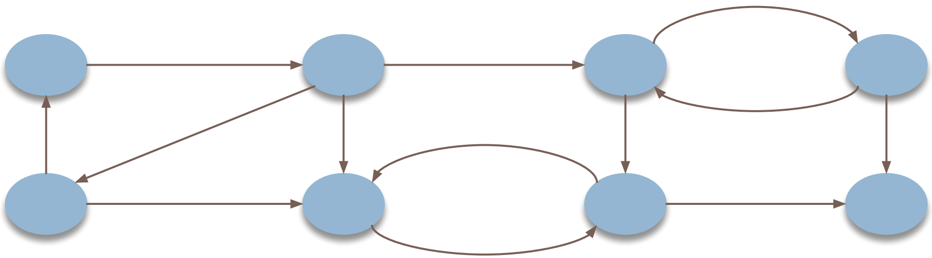
Topological Sort



Time = 10

Strongly Connected Components

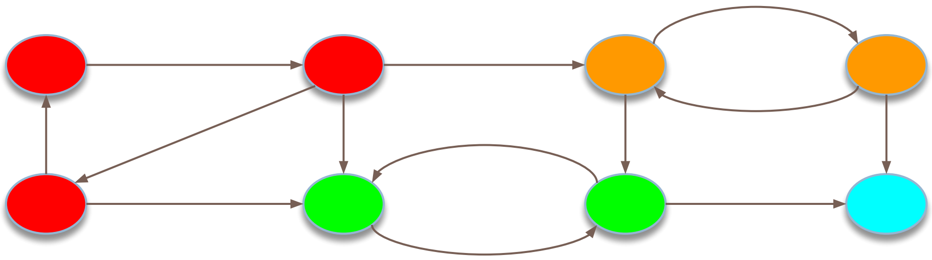
- **Strongly Connected Component**
- A strongly connected component of a directed graph G = (V,E) is a maximal set of vertices C such that for every pair of vertices u and v in C, we have both **v is reachable from u** and **u is reachable from v**. That is u and v are reachable from each other



Strongly Connected Components

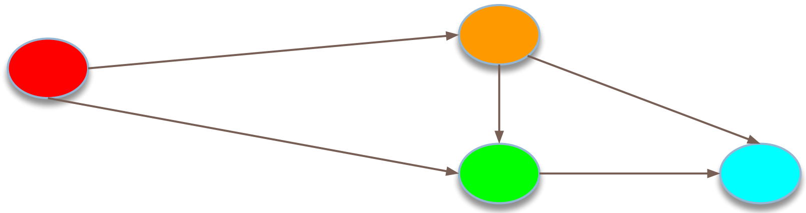
- **Strongly Connected Component**

- A strongly connected component of a directed graph G = (V,E) is a maximal set of vertices C such that for every pair of vertices u and v in C, we have both **v is reachable from u** and **u is reachable from v**. That is u and v are reachable from each other



Strongly Connected Components

- **Strongly Connected Component**
- A strongly connected component of a directed graph G = (V,E) is a maximal set of vertices C such that for every pair of vertices u and v in C, we have both **v is reachable from u** and **u is reachable from v**. That is u and v are reachable from each other



Reduce the graph to its SCC
=> the **component graph**

Strongly Connected Components

- Often used as a subprocedure: partition the graph into its SCC and run an algorithm on each partition
- Used to identify **communities** of people on social networks
- Used to identify **bots/spam pages**

Kosaraju’s algorithm

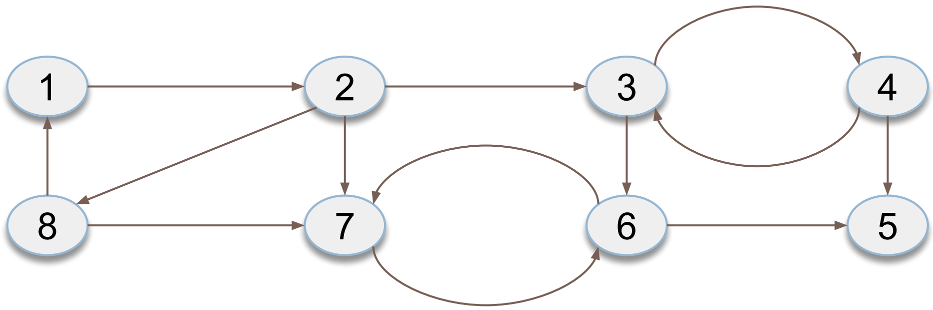
- Leverages observation that, if there exists a number of SCC in the graph G, then those SCC stay the same in the graph G^T (with all of its edges flipped)

- Idea is to compute DFS of the graph to get finishing times, transpose that graph, then run DFS(u) for every node in that order

  - The first node that we traverse is either
    - Already part of a strongly connected component
    - The root of a new connected component.

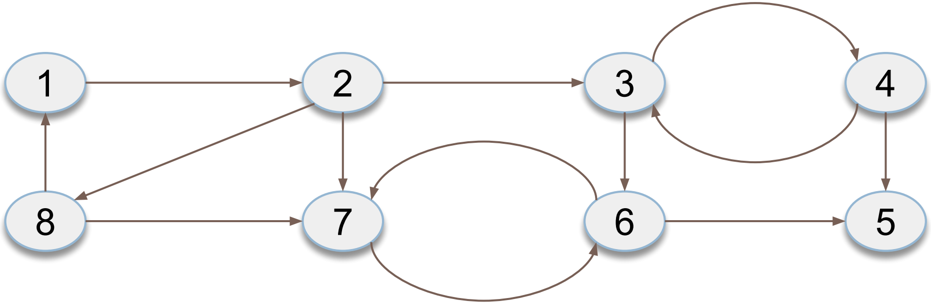
Kosaraju's algorithm

- First compute finishing times of all vertices



Kosaraju’s algorithm

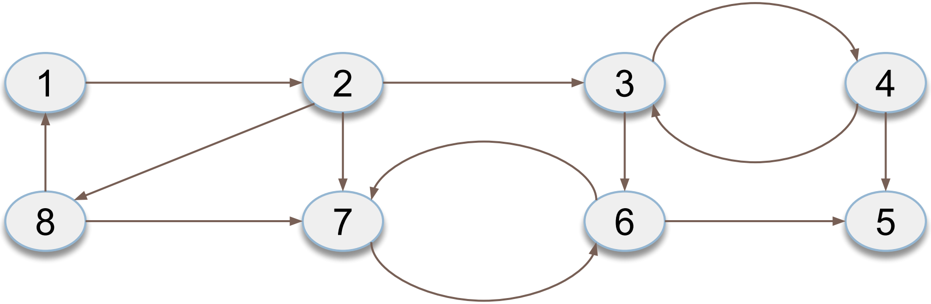
- First compute finishing times of all vertices



1	7
2	6
3	5
4	4
5	0
6	1
7	2
8	3

Kosaraju's algorithm

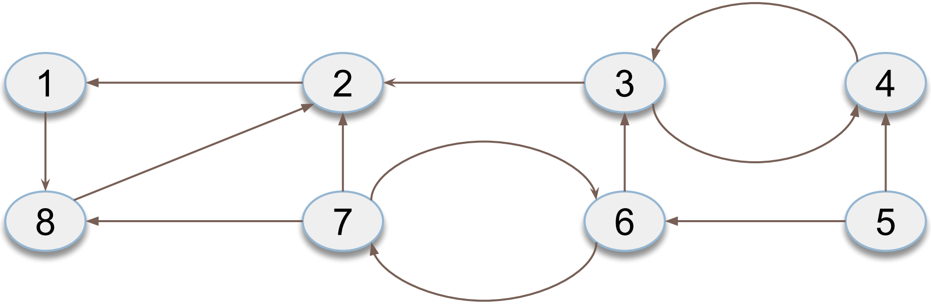
- Compute transpose of G (flip all edges)



1	7
2	6
3	5
4	4
5	0
6	1
7	2
8	3

Kosaraju's algorithm

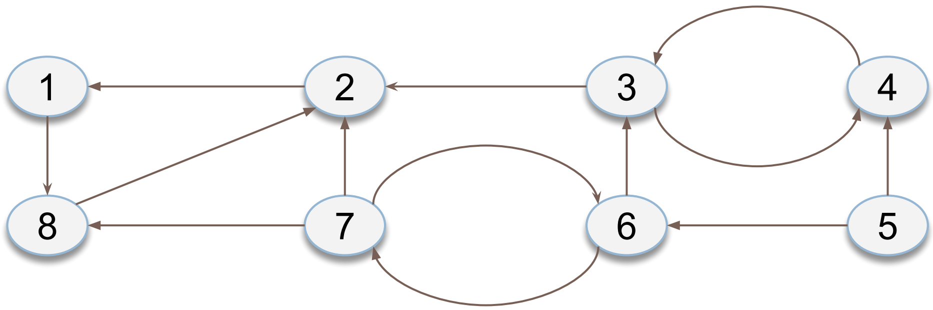
- Compute transpose of G (flip all edges)



1	7
2	6
3	5
4	4
5	0
6	1
7	2
8	3

Kosaraju's algorithm

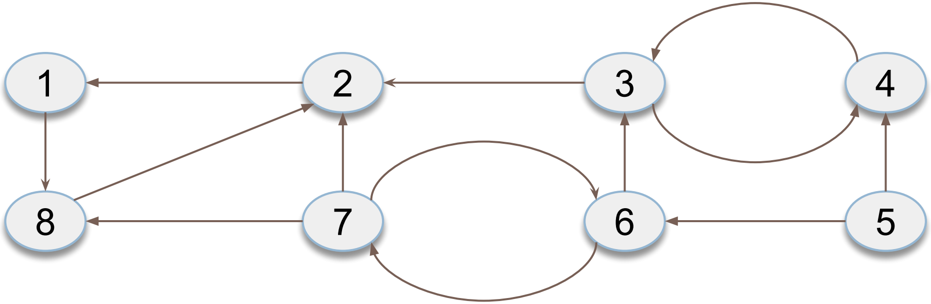
- Sort vertices in reverse order of their finishing time



1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

Kosaraju’s algorithm

- Go through each vertex v

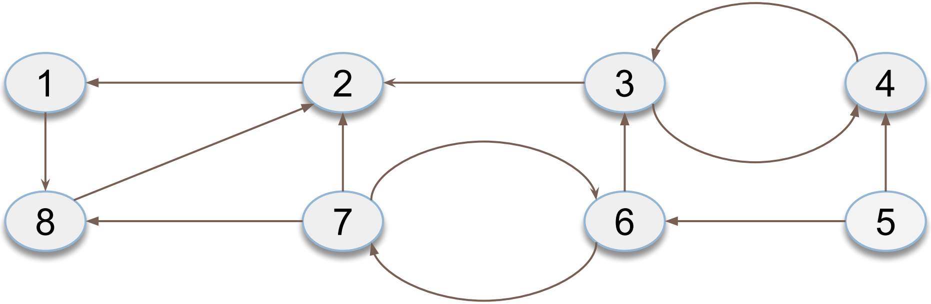


- Set v.scc = v. Then run DFS(v)
- For all reachable v’
  - If v’.scc = null, then assign v’.scc = v

1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

Kosaraju's algorithm

- Go through each vertex v

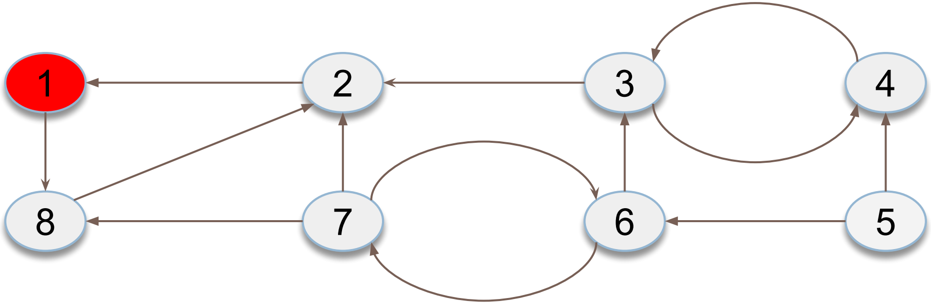


- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

Kosaraju's algorithm

- Go through each vertex v

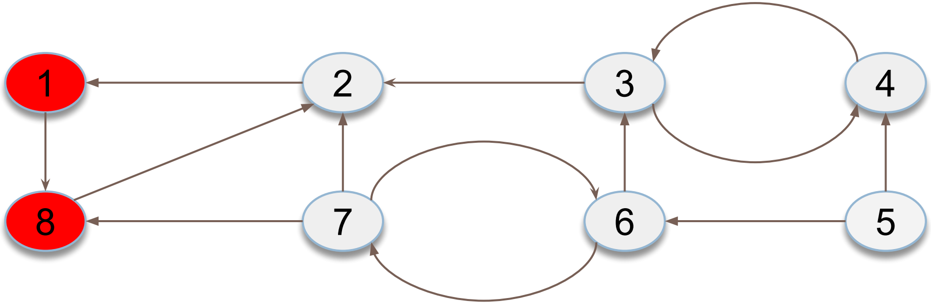


- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

Kosaraju's algorithm

- Go through each vertex v

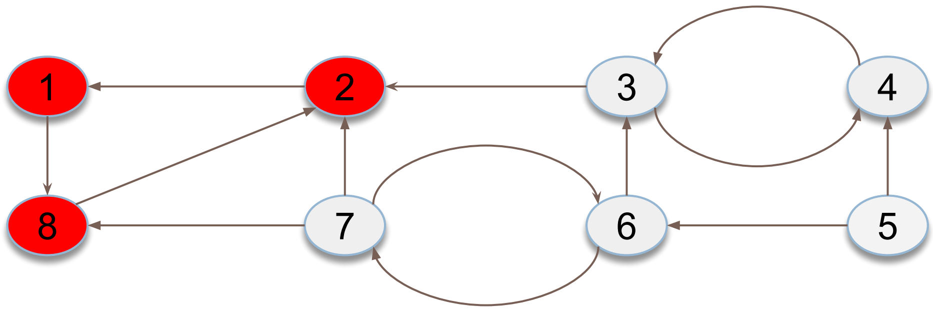


- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

Kosaraju's algorithm

- Go through each vertex v

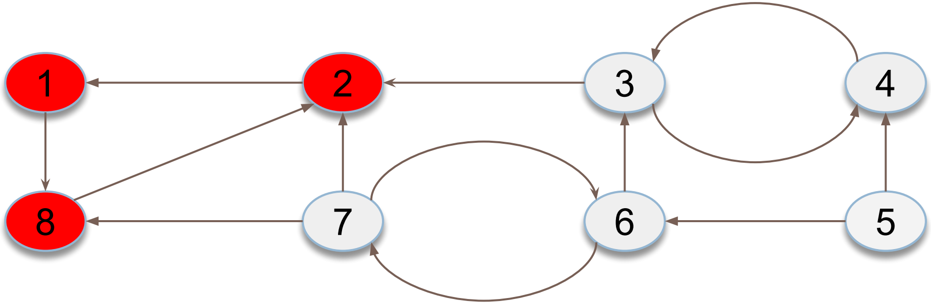


1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

Kosaraju's algorithm

- Go through each vertex v

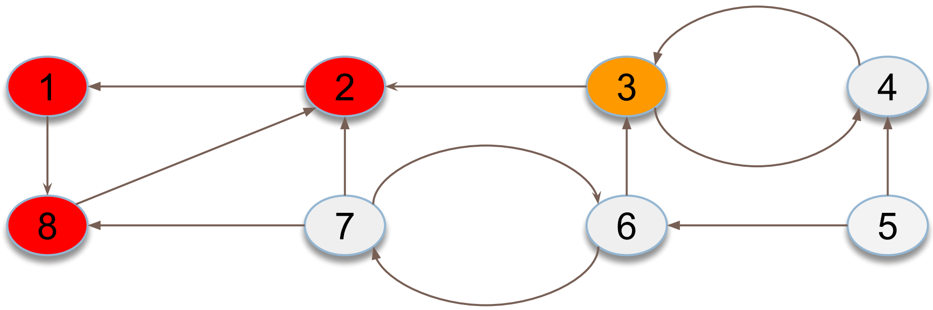


- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

Kosaraju's algorithm

- Go through each vertex v

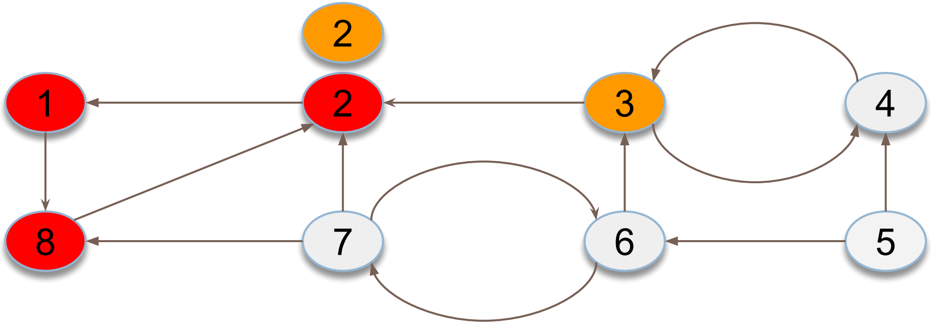


1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

Kosaraju's algorithm

- Go through each vertex v

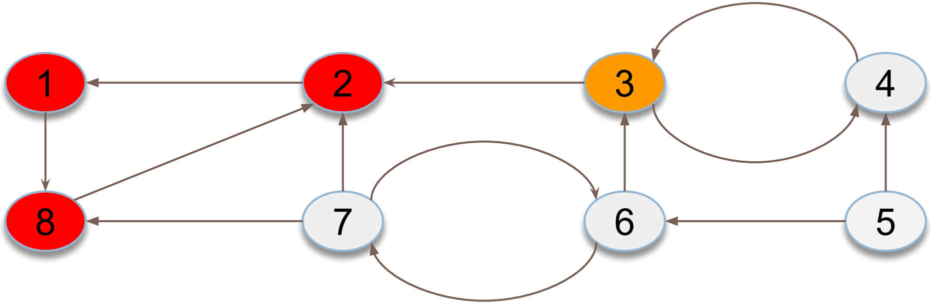


1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

Kosaraju's algorithm

- Go through each vertex v

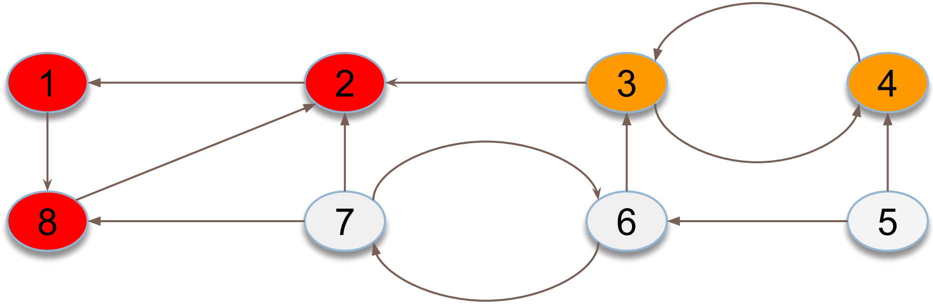


1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

Kosaraju’s algorithm

- Go through each vertex v

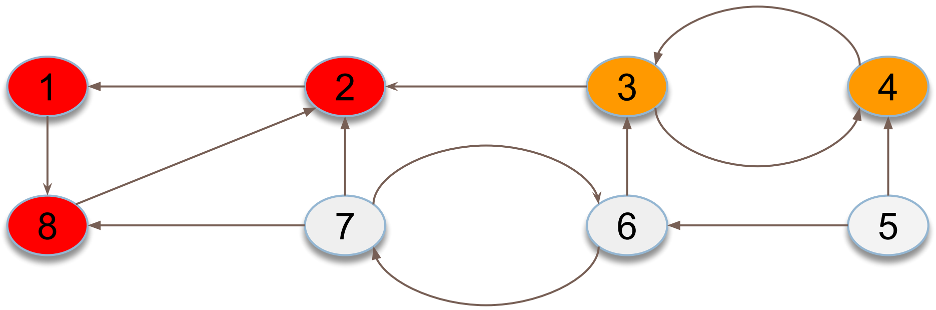


- Set v.scc = v. Then run DFS(v)
- For all reachable v’
  - If v’.scc = null, then assign v’.scc = v

1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

Kosaraju’s algorithm

- Go through each vertex v

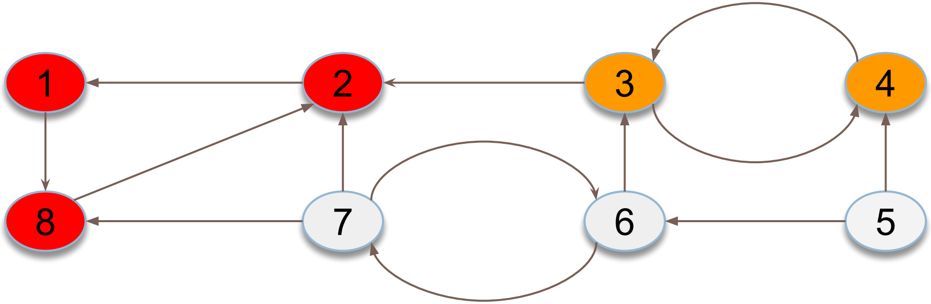


- Set v.scc = v. Then run DFS(v)
- For all reachable v’
  - If v’.scc = null, then assign v’.scc = v

1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

Kosaraju's algorithm

- Go through each vertex v

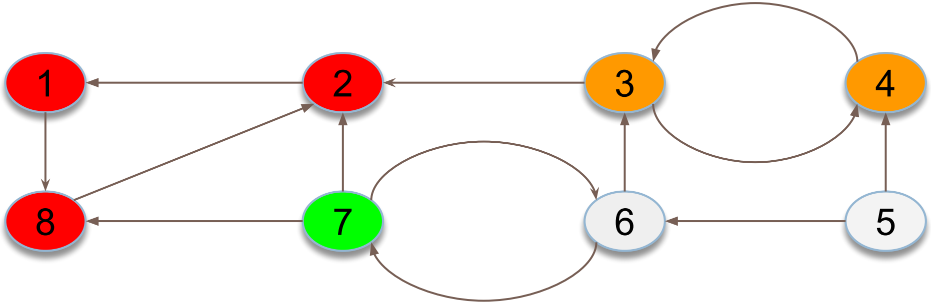


1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

Kosaraju's algorithm

- Go through each vertex v

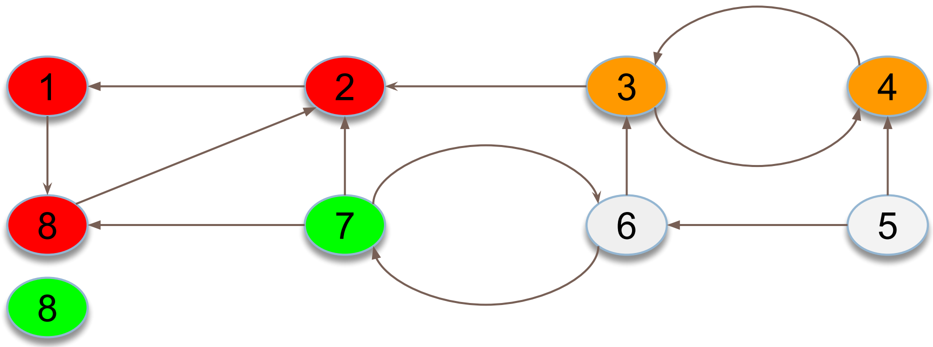


1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

Kosaraju's algorithm

- Go through each vertex v

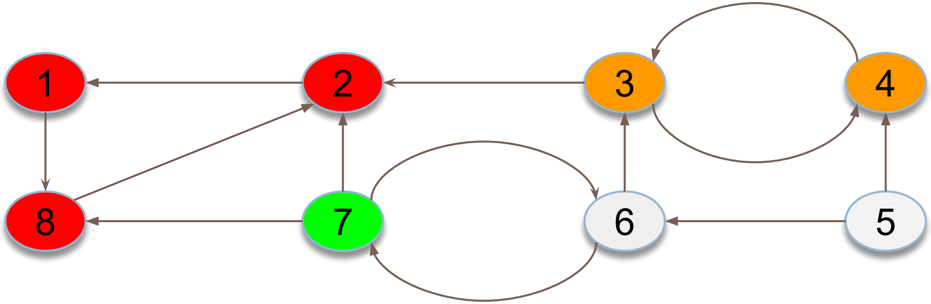


1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

Kosaraju's algorithm

- Go through each vertex v

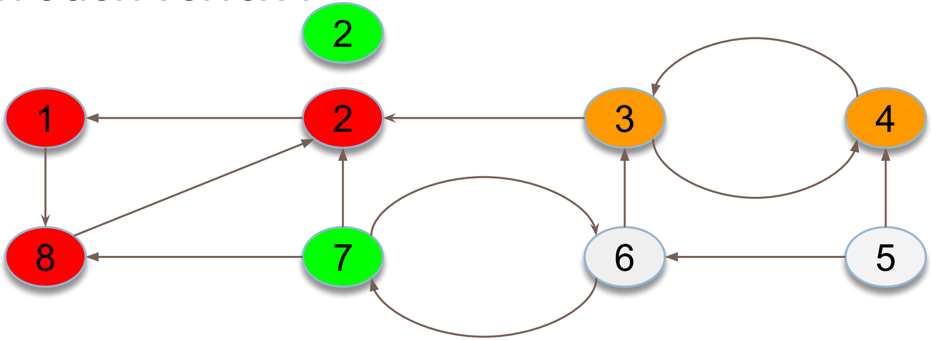


- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

Kosaraju's algorithm

- Go through each vertex v

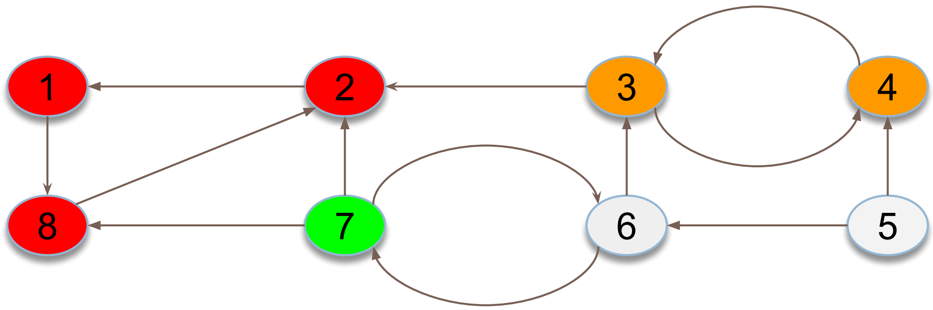


1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

Kosaraju's algorithm

- Go through each vertex v

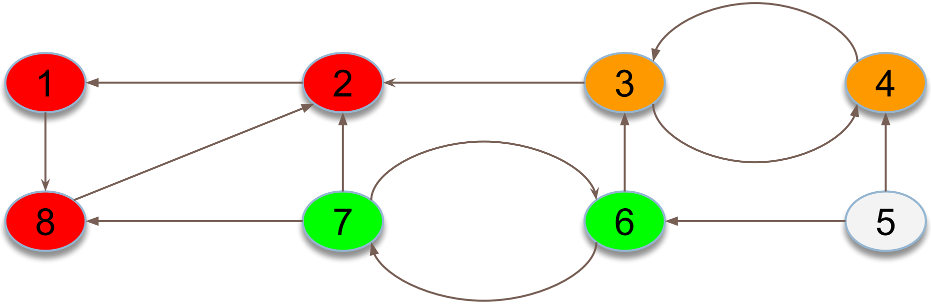


- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

Kosaraju's algorithm

- Go through each vertex v

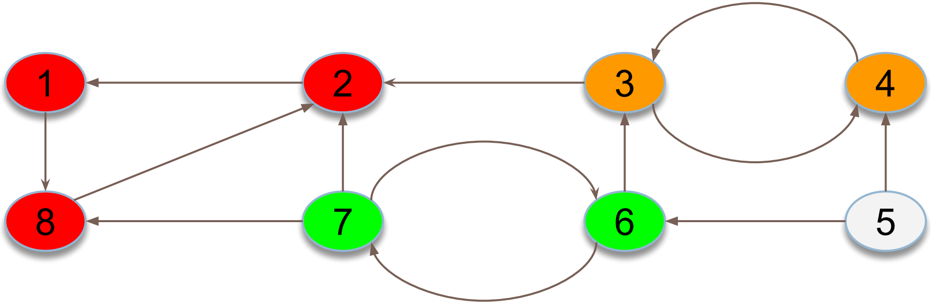


1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

- Set v.scc = v. Then run DFS(v)
- For all reachable v'
  - If v'.scc = null, then assign v'.scc = v

Kosaraju’s algorithm

- Go through each vertex v

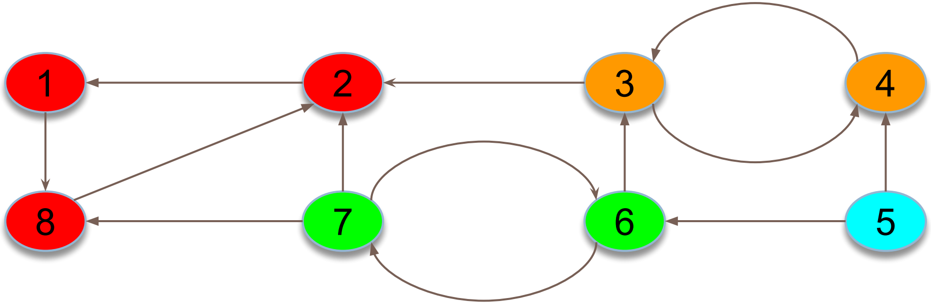


1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

- Set v.scc = v. Then run DFS(v)
- For all reachable v’
  - If v’.scc = null, then assign v’.scc = v

Kosaraju’s algorithm

- Go through each vertex v



1	7
2	6
3	5
4	4
8	3
7	2
6	1
5	0

- Set v.scc = v. Then run DFS(v)
- For all reachable v’
  - If v’.scc = null, then assign v’.scc = v

Intuition revisited

- Once visit a node in a strongly connected component, will visit:
  - All nodes **n** in that strongly connected nodes
  - Nodes **n'** that leave the strongly connected components

- When compute the transpose, switching the edges
  - Has no effects on nodes **n** in the SCC (because (u,v) and (v,u) are both paths in the SCC)
  - Means that nodes **n'** are no longer reachable

Kosaraju's algorithm


```
findSCC(Graph<T>  g) {
      List<GraphNode<T> topoSort =  DFS(G);
      topoSort.sort(//reverse finishing time);
      transpose(G);
      for (GraphNode u: topoSortReverse) {
            if (u.scc == null) assignSCC(u,u);
      }
assignSCC(GraphNode<T> u, GraphNode<T> root) {
      assert(u.scc == null);
      u.scc = root;
      for (GraphNode<T> n: u.neighbours) {
            assignSCC(n,root);
      }}
```

Add a parameter **GraphNode<T> scc** to every graph node.

Other SCC algorithms

- Kosaraju's algorithm easy to understand, but requires two DFS calls

- Tarjan’s algorithm (former Cornell prof!) and Djikstra’s algorithm are harder to reason about but require only one DFS call and one or more stacks
  - Read up if you’re interested!