CS062 DATA STRUCTURES AND ADVANCED PROGRAMMING

24–25: Graphs

Alexandra Papoutsaki she/her/hers

Tom Yeh he/him/his
Lecture 24-25: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
  - Connected Components

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Topological Sort
  - Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226
Undirected Graphs

- **Graph**: A set of *vertices* connected pairwise by *edges*.

https://www.wikiwand.com/simple/Graph_(mathematics)
Why study graphs?

- Thousands of practical applications.
- Hundreds of graph algorithms known.
- Interesting and broadly useful abstraction.
- Challenging branch of theoretical computer science.
Protein-protein interaction graph

The Internet

https://www.opte.org/the-internet
Social media

Graph terminology

- **Path**: Sequence of vertices connected by edges
- **Cycle**: Path whose first and last vertices are the same
- Two vertices are **connected** if there is a path between them
Examples of graph-processing problems

- Is there a path between vertex s and t?
- What is the shortest path between s and t?
- Is there a cycle in the graph?
- **Euler Tour**: Is there a cycle that uses each edge exactly once?
- **Hamilton Tour**: Is there a cycle that uses each vertex exactly once?
- Is there a way to connect all vertices?
- What is the shortest way to connect all vertices?
- Is there a vertex whose removal disconnects the graph?
Lecture 24-25: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
  - Connected Components
- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Topological Sort
  - Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226
Graph representation

- **Vertex representation**: Here, integers between 0 and V-1.
  - We will use a symbol table (dictionary) to map between names of vertices and integers (indices).
Basic Graph API

- **public class Graph**
  - `Graph(int V)`: create an empty graph with V vertices.
  - `void addEdge(int v, int w)`: add an edge v-w.
  - `Iterable<Integer> adj(int v)`: return vertices adjacent to v.
  - `int V()`: number of vertices.
  - `int E()`: number of edges.
Example of how to use the Graph API to process the graph

```java
public static int degree(Graph g, int v) {
    int count = 0;
    for (int w : g.adj(v))
        count++;
    return count;
}
```
Graph density

- In a simple graph (no parallel edges or loops), if $|V| = n$, then:
  - minimum number of edges is 0 and
  - maximum number of edges is $n(n - 1)/2$.
- Dense graph -> edges closer to maximum.
- Sparse graph -> edges closer to minimum.
Graph representation: adjacency matrix

- Maintain a $|V|$-by-$|V|$ boolean array; for each edge $v-w$:
  - $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$;
- Good for dense graphs (edges close to $|V|^2$).
- Constant time for lookup of an edge.
- Constant time for adding an edge.
- $|V|$ time for iterating over vertices adjacent to $v$.
- Symmetric, therefore wastes space in undirected graphs ($|V|^2$).
- Not widely used in practice.

\[
\begin{array}{cccc}
A & B & C & D \\
A & 0 & 1 & 1 & 1 \\
B & 1 & 0 & 0 & 1 \\
C & 1 & 0 & 0 & 0 \\
D & 1 & 1 & 0 & 0 \\
\end{array}
\]
Graph representation: adjacency list

- Maintain vertex-indexed array of lists.
- Good for sparse graphs (edges proportional to $|V|$) which are much more common in the real world.
- Algorithms based on iterating over vertices adjacent to $v$.
- Space efficient ($|E| + |V|$).
- Constant time for adding an edge.
- Lookup of an edge or iterating over vertices adjacent to $v$ is $\text{degree}(v)$. 
UNDIRECTED GRAPHS

Adjacency-list graph representation in Java

```java
public class Graph {

    private final int V;
    private int E;
    private Bag<Integer>[] adj;

    //Initializes an empty graph with V vertices and 0 edges.
    public Graph(int V) {
        this.V = V;
        this.E = 0;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new Bag<Integer>();
        }
    }

    //Adds the undirected edge v-w to this graph. Parallel edges and self-loops allowed
    public void addEdge(int v, int w) {
        E++;
        adj[v].add(w);
        adj[w].add(v);
    }

    //Returns the vertices adjacent to vertex v.
    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```

A bag is a collection where removing items is not supported—its purpose is to provide clients with the ability to collect items and then to iterate through the collected items.
Lecture 24-25: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
  - Connected Components

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Topological Sort
  - Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226
Mazes as graphs

- Vertex = intersection; edge = passage

How to survive a maze: a lesson from a Greek myth

- Theseus escaped from the labyrinth after killing the Minotaur with the following strategy instructed by Ariadne:
  - Unroll a ball of string behind you.
  - Mark each newly discovered intersection and passage.
  - Retrace steps when no unmarked options.
- Also known as the Trémaux algorithm.
Depth-first search

- **Goal**: Systematically traverse a graph.
- **DFS** (to visit a vertex $v$)
  - Mark vertex $v$.
  - Recursively visit all unmarked vertices $w$ adjacent to $v$.

- **Typical applications**:
  - Find all vertices connected to a given vertex.
  - Find a path between two vertices.
4.1 Depth-First Search Demo
Depth-first search

- **Goal**: Find all vertices connected to \( s \) (and a corresponding path).
- **Idea**: Mimic maze exploration.
- **Algorithm**:
  - Use recursion (ball of string).
  - Mark each visited vertex (and keep track of edge taken to visit it).
  - Return (retrace steps) when no unvisited options.
- When started at vertex \( s \), DFS marks all vertices connected to \( s \) (and no other).
Depth-first search in Java

```java
public class DepthFirstSearch {
    private boolean[] marked; // marked[v] = is there an s-v path?
    private int[] edgeTo; // edgeTo[v] = previous vertex on path from s to v

    public DepthFirstSearch(Graph G, int s) {
        marked = new boolean[G.V()];
        edgeTo = new int[G.V()];
        dfs(G, s);
    }

    // depth first search from v
    private void dfs(Graph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                edgeTo[w] = v;
                dfs(G, w);
            }
        }
    }
}
```
Depth-first search Analysis

- DFS marks all vertices connected to $s$ in time proportional to $|V| + |E|$ in the worst case.

- Initializing arrays marked and edgeTo takes time proportional to $|V|$.

- Each adjacency-list entry is examined exactly once and there are $2|E|$ such edges (two for each edge).

- Once we run DFS, we can check if vertex $v$ is connected to $s$ in constant time. We can also find the $v$-$s$ path (if it exists) in time proportional to its length.
Lecture 24-25: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
  - Connected Components

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Topological Sort
  - Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226
Breadth-first search

- **BFS** (from source vertex $s$)
  - Put $s$ on a queue and mark it as visited.
  - Repeat until the queue is empty:
    - Dequeue vertex $v$.
    - Enqueue each of $v$’s unmarked neighbors and mark them.

- Basic idea: BFS traverses vertices in order of distance from $s$. 
4.1 Breadth-First Search Demo
BREADTH-FIRST SEARCH

Breadth-first search in Java

```java
public class BreadthFirstPaths {
    private boolean[] marked; // marked[v] = is there an s-v path
    private int[] edgeTo; // edgeTo[v] = previous edge on shortest s-v path
    private int[] distTo; // distTo[v] = number of edges shortest s-v path

    public BreadthFirstPaths(Graph G, int s) {
        marked = new boolean[G.V()];
        distTo = new int[G.V()];
        edgeTo = new int[G.V()];
        bfs(G, s);
    }

    private void bfs(Graph G, int s) {
        Queue<Integer> q = new Queue<Integer>();
        distTo[s] = 0;
        marked[s] = true;
        q.enqueue(s);

        while (!q.isEmpty()) {
            int v = q.dequeue();
            for (int w : G.adj(v)) {
                if (!marked[w]) {
                    edgeTo[w] = v;
                    distTo[w] = distTo[v] + 1;
                    marked[w] = true;
                    q.enqueue(w);
                }
            }
        }
    }
}
```
Breadth-first search

- **DFS**: Put unvisited vertices on a stack.

- **BFS**: Put unvisited vertices on a queue.

- **Shortest path problem**: Find path from \( s \) to \( t \) that uses the fewest number of edges.
  - E.g., calculate the fewest numbers of hops in a communication network.
  - E.g., calculate the Kevin Bacon number or Erdös number.

- BFS computes shortest paths from \( s \) to all vertices in a graph in time proportional to \(|E| + |V|\)
  - The queue always consists of zero or more vertices of distance \( k \) from \( s \), followed by zero or more vertices of \( k+1 \).
Lecture 24-25: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
  - Connected Components

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Topological Sort
  - Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226
Connectivity queries

- **Goal**: Preprocess graph to answer questions of the form “is \( v \) connected to \( w \)” in constant time.

- **public class CC**
  - `CC(Graph G)`: find connected components in \( G \).
  - `boolean connected(int v, int w)`: are \( v \) and \( w \) connected?
  - `int count()`: number of connected components.
  - `int id(int v)`: component identifier for vertex \( v \).
Connected components

- **Goal**: Partition vertices into connected components.

- **Connected Components**
  - Initialize all vertices as unmarked.
  - For each unmarked vertex, run DFS to identify all vertices discovered as part of the same component.
4.1 CONNECTED COMPONENTS DEMO
public class CC {
    private boolean[] marked; // marked[v] = has vertex v been marked?
    private int[] id; // id[v] = id of connected component containing v
    private int[] size; // size[id] = number of vertices in given component
    private int count; // number of connected components

    public CC(Graph G) {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        size = new int[G.V()];
        for (int v = 0; v < G.V(); v++) {
            if (!marked[v]) {
                dfs(G, v);
                count++;
            }
        }
    }

    private void dfs(Graph G, int v) {
        marked[v] = true;
        id[v] = count;
        size[count]++;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
    }
}
Lecture 24-25: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
  - Connected Components

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Topological Sort
  - Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226
Directed Graph Terminology

- **Directed Graph (digraph)**: A set of vertices $V$ connected pairwise by a set of directed edges $E$.
  
  - E.g., $V = \{0,1,2,3,4,5,6,7,8,9,10,11,12\}$,
    $E = \{\{0,1\}, \{0,5\}, \{2,0\}, \{2,3\}, \{3,2\}, \{3,5\}, \{4,2\}, \{4,3\}, \{5,4\}, \{6,0\}, \{6,4\}, \{6,9\}, \{7,6\}, \{7,8\}, \{8,7\}, \{8,9\},
    \{9,10\}, \{9,11\}, \{10,12\}, \{11,4\}, \{11,12\}, \{12,9\}\}$.  

- **Directed path**: A sequence of vertices in which there is a directed edge pointing from each vertex in the sequence to its successor in the sequence, with no repeated edges.
  
  - A simple directed path is a directed path with no repeated vertices.

- **Directed cycle**: A directed path with at least one edge whose first and last vertices are the same.
  
  - A simple directed cycle is a directed cycle with no repeated vertices (other than the first and last).

- The **length** of a cycle or a path is its number of edges.
Directed Graph Terminology

- **Self-loop**: an edge that connects a vertex to itself.
- Two edges are **parallel** if they connect the same pair of vertices.
- The **outdegree** of a vertex is the number of edges pointing from it.
- The **indegree** of a vertex is the number of edges pointing to it.
- A vertex \( w \) is **reachable** from a vertex \( v \) if there is a directed path from \( v \) to \( w \).
- Two vertices \( v \) and \( w \) are **strongly connected** if they are mutually reachable.
Directed Graph Terminology

- A digraph is **strongly connected** if there is a directed path from every vertex to every other vertex.
- A digraph that is not strongly connected consists of a set of strongly connected components, which are maximal strongly connected subgraphs.
- A **directed acyclic graph (DAG)** is a digraph with no directed cycles.
Anatomy of a digraph

Anatomy of a digraph

A digraph and its strong components
## Digraph Applications

<table>
<thead>
<tr>
<th>Digraph</th>
<th>Vertex</th>
<th>Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web</td>
<td>Web page</td>
<td>Link</td>
</tr>
<tr>
<td>Cell phone</td>
<td>Person</td>
<td>Placed call</td>
</tr>
<tr>
<td>Financial</td>
<td>Bank</td>
<td>Transaction</td>
</tr>
<tr>
<td>Transportation</td>
<td>Intersection</td>
<td>One-way street</td>
</tr>
<tr>
<td>Game</td>
<td>Board</td>
<td>Legal move</td>
</tr>
<tr>
<td>Citation</td>
<td>Article</td>
<td>Citation</td>
</tr>
<tr>
<td>Infectious Diseases</td>
<td>Person</td>
<td>Infection</td>
</tr>
<tr>
<td>Food web</td>
<td>Species</td>
<td>Predator-prey relationship</td>
</tr>
</tbody>
</table>
### Popular digraph problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s-&gt;t path</td>
<td>Is there a path from s to t?</td>
</tr>
<tr>
<td>Shortest s-&gt;t path</td>
<td>What is the shortest path from s to t?</td>
</tr>
<tr>
<td>Directed cycle</td>
<td>Is there a directed cycle in the digraph?</td>
</tr>
<tr>
<td>Topological sort</td>
<td>Can vertices be sorted so all edges point from earlier to later vertices?</td>
</tr>
<tr>
<td>Strong connectivity</td>
<td>Is there a directed path between every pair of vertices?</td>
</tr>
</tbody>
</table>
Lecture 24-25: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
  - Connected Components

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Topological Sort
  - Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226
Basic Graph API

- **public class** Digraph
  - `Digraph(int V)`: create an empty digraph with `V` vertices.
  - `void addEdge(int v, int w)`: add an edge `v->w`.
  - `Iterable<Integer> adj(int v)`: return vertices adjacent from `v`.
  - `int V()`: number of vertices.
  - `int E()`: number of edges.
  - `Digraph reverse()`: reverse edges of digraph.
DIRECTED GRAPHS

Digraph representation: adjacency list

- Maintain vertex-indexed array of lists.
- Good for sparse graphs (edges proportional to $|V|$) which are much more common in the real world.
- Algorithms based on iterating over vertices adjacent from $v$.
- Space efficient ($|E| + |V|$).
- Constant time for adding a directed edge.
- Lookup of a directed edge or iterating over vertices adjacent from $v$ is $\text{outdegree}(v)$. 
Adjacency-list digraph representation in Java

```java
public class Digraph {

    private final int V;
    private int E;
    private Bag<Integer>[] adj;

    // Initializes an empty digraph with V vertices and 0 edges.
    public Digraph(int V) {
        this.V = V;
        this.E = 0;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new Bag<Integer>();
        }
    }

    // Adds the directed edge v->w to this digraph.
    public void addEdge(int v, int w) {
        E++;
        adj[v].add(w);
    }

    // Returns the vertices adjacent from vertex v.
    public Iterable<Integer> adj(int v) {
        return adj[v];
    }
}
```
Lecture 24-25: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
  - Connected Components
- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Topological Sort
  - Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226
Reachability

- Find all vertices reachable from $s$ along a directed path.

Is $w$ reachable from $v$ in this digraph?

https://apprize.info/science/algorithms_2/2.html
Depth-first search in digraphs

- Same method as for undirected graphs.
  - Every undirected graph is a digraph with edges in both directions.
  - Maximum number of edges in a simple digraph is $n(n - 1)$.

- DFS (to visit a vertex $v$)
  - Mark vertex $v$.
  - Recursively visit all unmarked vertices $w$ adjacent from $v$.

- Typical applications:
  - Find a directed path from source vertex $S$ to a given target vertex $V$.
  - Topological sort.
  - Directed cycle detection.
4.2 Directed DFS Demo
Directed depth-first search in Java

```java
public class DirectedDFS {
    private boolean[] marked; // marked[v] = is there an s->v path?

    public DirectedDFS(Digraph G, int s) {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    // directed depth first search from v
    private void dfs(Digraph G, int v) {
        marked[v] = true;
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
    }
}
```
Alternative iterative implementation with a stack

```java
public class DirectedDFS {
    private boolean[] marked;  // marked[v] = is there an s->v path?

    public DirectedDFS(Digraph G, int s) {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    // iterative dfs that uses a stack
    private void dfs(Digraph G, int v) {
        Stack stack = new Stack();
        s.push(v);
        while (!stack.isEmpty()) {
            int vertex = stack.pop();
            if (!marked[vertex]) {
                marked[vertex] = true;
                while (int w : G.adj(vertex)) {
                    if (!marked[w])
                        stack.push(w);
                }
            }
        }
    }
}
```
Depth-first search Analysis

- DFS marks all vertices reachable from $s$ in time proportional to $|V| + |E|$ in the worst case.
- Initializing arrays marked takes time proportional to $|V|$.
- Each adjacency-list entry is examined exactly once and there are $E$ such edges.
- Once we run DFS, we can check if vertex $v$ is reachable from $s$ in constant time. We can also find the $s \rightarrow v$ path (if it exists) in time proportional to its length.
Lecture 24-25: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
  - Connected Components

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Topological Sort
  - Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226
Breadth-first search

- Same method as for undirected graphs.
  - Every undirected graph is a digraph with edges in both directions.
- BFS (from source vertex $s$)
  - Put $s$ on queue and mark $s$ as visited.
  - Repeat until the queue is empty:
    - Dequeue vertex $v$.
    - Enqueue all unmarked vertices adjacent from $v$, and mark them.
- Typical applications:
  - Find the shortest (in terms of number of edges) directed path between two vertices in time proportional to $|E| + |V|$.
4.2 Directed BFS Demo
Lecture 24-25: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
  - Connected Components

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Topological Sort
  - Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226
Depth-first orders

- If we save the vertex given as argument to recursive dfs in a data structure, we have three possible orders of seeing the vertices:
  - **Preorder**: Put the vertex on a queue before the recursive calls.
  - **Postorder**: Put the vertex on a queue after the recursive calls.
  - **Reverse postorder**: Put the vertex on a stack after the recursive calls.
Depth-first orders

```java
public class DepthFirstOrder {
    private boolean[] marked; // marked[v] = has v been marked in dfs?
    private Queue<Integer> preorder; // vertices in preorder
    private Queue<Integer> postorder; // vertices in postorder
    private Stack<Integer> reversePostOrder; // vertices in reverse postorder

    /**
     * Determines a depth-first order for the digraph (code G).
     * @param G the digraph
     */
    public DepthFirstOrder(Digraph G) {
        postorder = new Queue<Integer>();
        preorder = new Queue<Integer>();
        reversePostOrder = new Stack<Integer>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
    }

    // run DFS in digraph G from vertex v and compute preorder/postorder
    private void dfs(Digraph G, int v) {
        marked[v] = true;
        preorder.enqueue(v);
        for (int w : G.adj(v)) {
            if (!marked[w]) {
                dfs(G, w);
            }
        }
        postorder.enqueue(v);
        reversePostOrder.push(v);
    }
```
Depth-first orders

preorder is order of dfs() calls
postorder is order in which vertices are done

reversePost

pre
0
0 5
0 5 4

post
4
4 5
4 5 1

reversePost
12 1 5 4
11 12 1 5 4
10 11 12 1 5 4
9 10 11 12 1 5 4
8 7 2 3 0 6 9 10 11 12 1 5 4

stack

0 done
check 1
dfs(2)
check 0
dfs(3)
check 5
3 done
2 done
check 3
check 4
check 6
dfs(7)
check 6
7 done
dfs(8)
check 7
8 done
check 9
check 10
check 11
check 12

6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
check 12
9 done
check 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
check 12
9 done
check 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
check 12
9 done
check 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
check 12
9 done
check 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
check 12
9 done
check 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
check 12
9 done
check 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
check 12
9 done
check 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
check 12
9 done
check 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
check 12
9 done
check 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
check 12
9 done
check 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
check 12
9 done
check 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
check 12
9 done
check 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
check 12
9 done
ccheck 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
ccheck 12
9 done
ccheck 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
ccheck 12
9 done
ccheck 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
ccheck 12
9 done
ccheck 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
dfs(12)
12 done
11 done
dfs(10)
10 done
ccheck 12
9 done
ccheck 12
6 done
check 4
dfs(0)
dfs(5)
4 done
5 done
dfs(1)
1 done
dfs(6)
dfs(9)
dfs(11)
Topological sort

- **Goal**: Order the vertices of a DAG so that all edges point from an earlier vertex to a later vertex.
  - Think of modeling major requirements as a DAG.
- Reverse postorder in DAG is a topological sort.
- With DFS, we can topologically sort a DAG in $|E| + |V|$ time.
4.2 Topological Sort Demo
Summary

- Single-source reachability in a digraph: DFS/BFS.
- Shortest path in a digraph: BFS.
- Topological sort in a DAG: DFS.
Lecture 24-25: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
  - Connected Components

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Topological Sort
  - Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226
Is a digraph strongly connected?

- Pick a random starting vertex $s$.

- Run DFS/BFS starting at $s$.
  - If have not reached all vertices, return false.

- Reverse edges.

- Run DFS/BFS again on reversed graph.
  - If have not reached all vertices, return false.
  
- Else return true.
Lecture 24-25: Graphs

- Undirected Graphs
  - Graph API
  - Depth-First Search
  - Breadth-First Search
  - Connected Components
- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Topological Sort
  - Strongly Connected Components

Some slides adopted from Algorithms 4th Edition or COS226
Readings:

- Textbook: Chapter 4.1 (Pages 522-556), Chapter 4.2 (Pages 566-594)
- Website:
  - [https://algs4.cs.princeton.edu/41graph/](https://algs4.cs.princeton.edu/41graph/)
  - [https://algs4.cs.princeton.edu/42digraph/](https://algs4.cs.princeton.edu/42digraph/)

Practice Problems:

- 4.1.1-4.1.6, 4.1.9, 4.1.11
- 4.2.1-4.27