CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

24–25: Undirected / Directed Graphs

Tom Yeh
he/him/his

Midterm 2 Stats

Lecture 34: Undirected Graphs

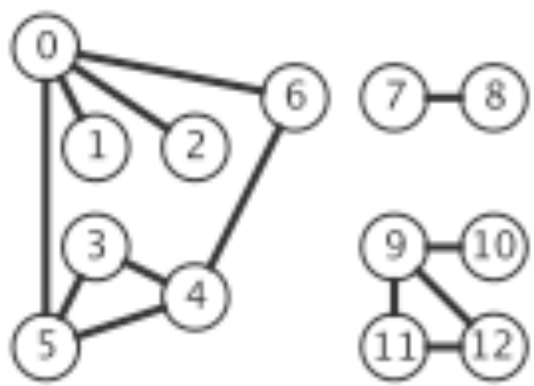
- Graph API
- Depth-First Search
- Breadth-First Search
- Connected Components

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 to map between names and integers.

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



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

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

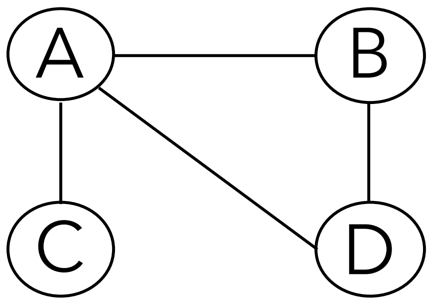
Graph density

- 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.

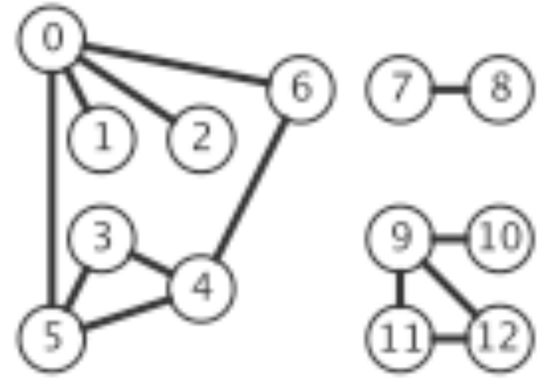
Graph representation: adjacency matrix

- Maintain a $|V|$-by-$|V|$ boolean array; for each edge `v-w`:
  - `adj[v][w] = adj[w][v] = true;` (1).
- 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.

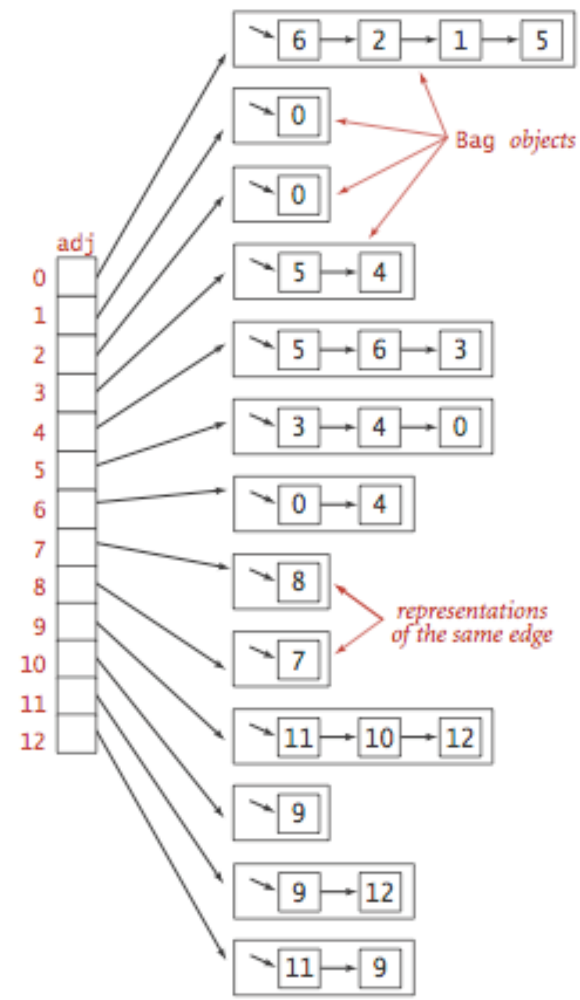
	A	**B**	**C**	**D**
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0



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 $degree(v)$.



Adjacency-list graph representation in 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];
    }
```

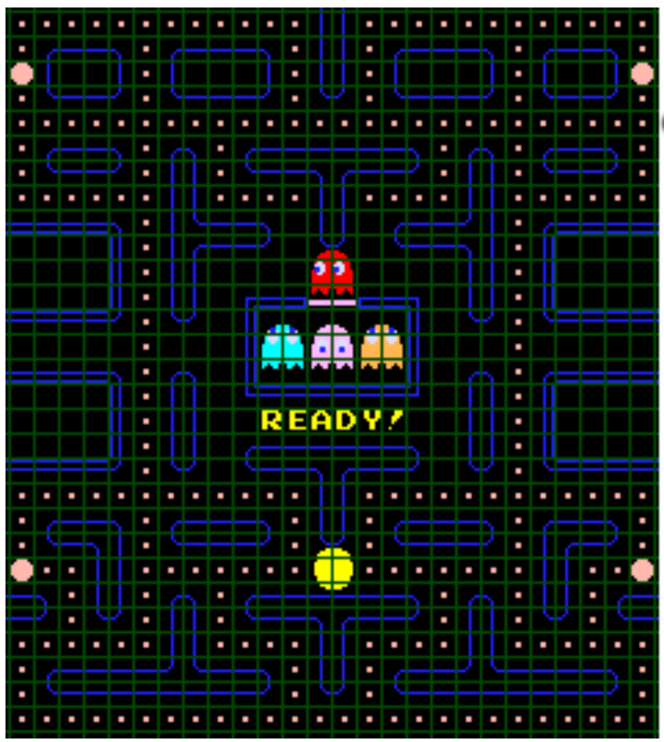
A bag is a collection where removing items is not supported—its purpose is to provide clients with the ability to collect items and then to iterate through the collected items

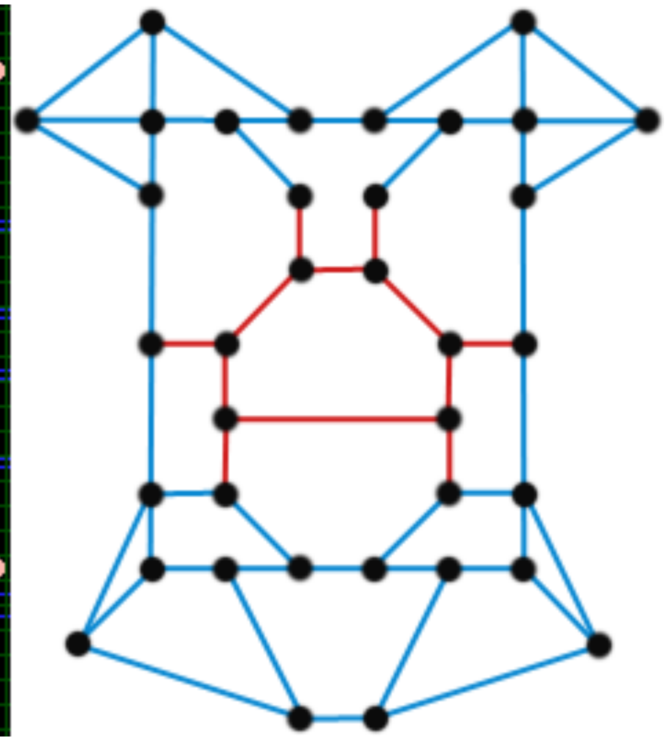
Lecture 34: Undirected Graphs

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

Mazes as graphs

- Vertex = intersection; edge = passage

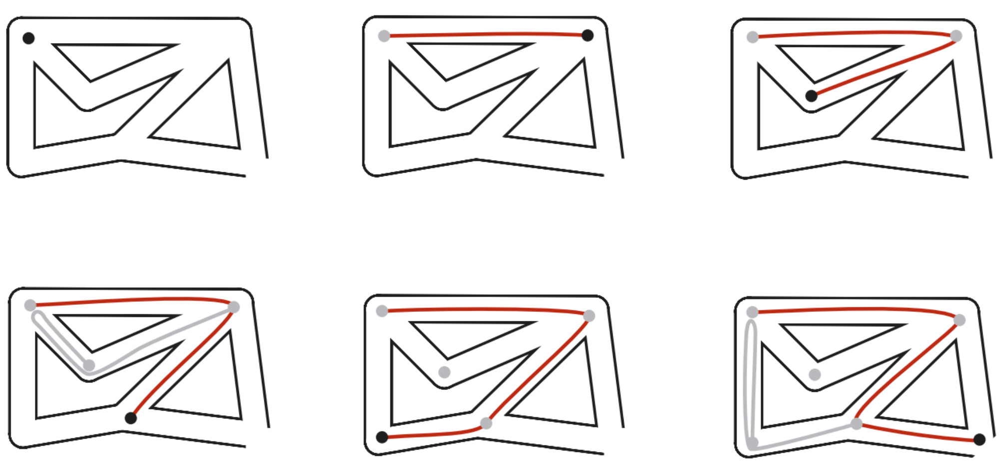




http://oatzy.blogspot.com/2011/09/playing-with-pac-man.html

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.
  - 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.

http://algs4.cs.princeton.edu

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

```
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 $2E$ 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 34: Undirected Graphs

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

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.

http://algs4.cs.princeton.edu

4.1 BREADTH-FIRST SEARCH DEMO

Breadth-first search in 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 34: Undirected Graphs

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

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.

http://algs4.cs.princeton.edu

4.1 CONNECTED COMPONENTS DEMO

Connected Components in Java

```java
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 34: Undirected Graphs

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

Readings:

- Textbook: Chapter 4.1 (Pages 522-556)
- Website:
  - https://algs4.cs.princeton.edu/41graph/

Practice Problems:

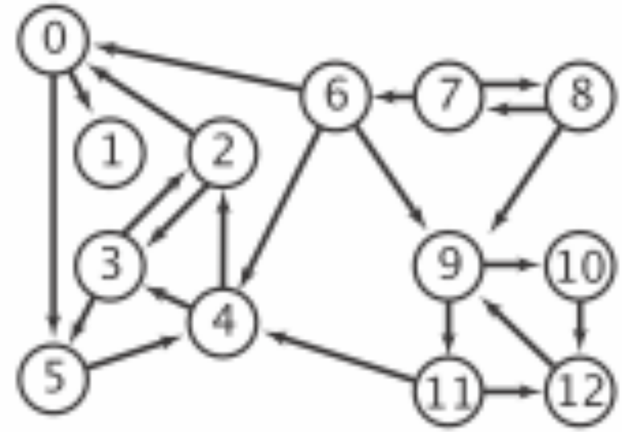
- 4.1.1-4.1.6, 4.1.9, 4.1.11

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

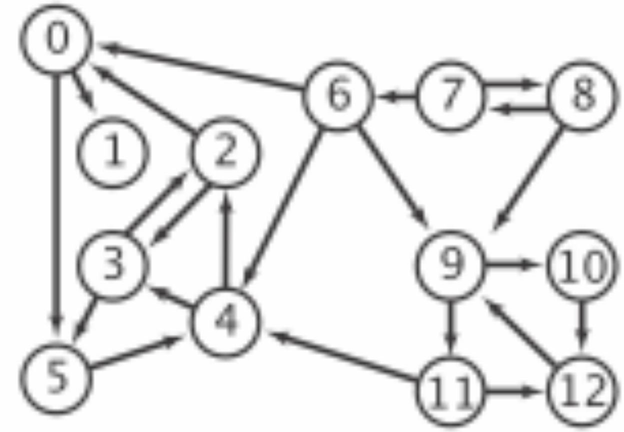
Some slides adopted from Algorithms 4th Edition or COS226

Directed Graph Terminology



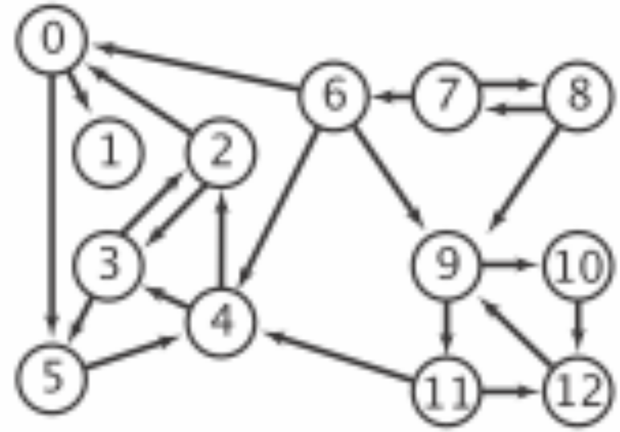
- Directed Graph (digraph) : a set of vertices V connected pairwise by a set of directed edges E.
  - E.g., V = {0,1,2,3,4,5,6,7,8,9,10,11,12},
    E = {{0,1}, {0,5}, {2,0}, {2,3},{3,2},{3,5},{4,2},{4,3},{5,4},{6,0},{6,4},{6,9},{7,6}{7,8},{8,7},{8,9},{9,10},{9,11},{10,12},{11,4},{11,12},{12,9}}.
- Directed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



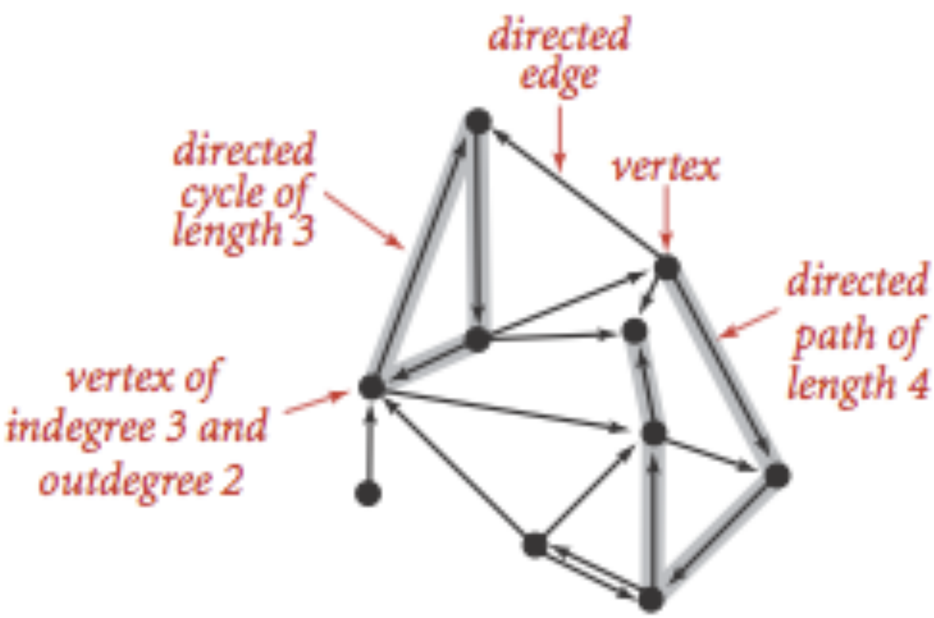
- Self-loop: an edge that connects a vertex to itself.
- Two edges are parallel if they connect the same pair of vertices.
- The outdegree of a vertex is the number of edges pointing from it.
- The indegree of a vertex is the number of edges pointing to it.
- A vertex w is reachable from a vertex v if 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

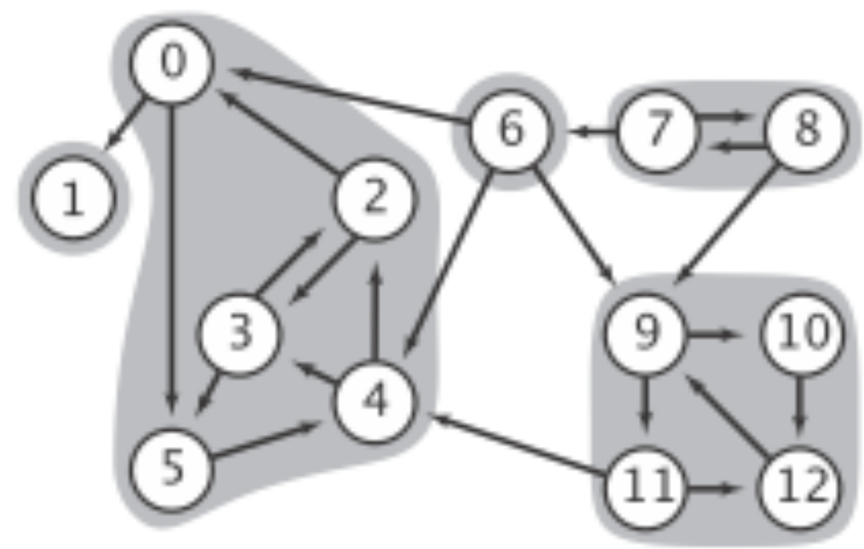


- A digraph is strongly connected if there is a directed path from every vertex to every other vertex.
- A digraph that is not strongly connected consists of a set of strongly connected components, which are maximal strongly connected subgraphs.
- A directed acyclic graph (DAG) is a digraph with no directed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

Anatomy of a digraph



Anatomy of a digraph



A digraph and its strong components

Digraph Applications

Digraph	Vertex	Edge
Web	Web page	Link
Cell phone	Person	Placed call
Financial	Bank	Transaction
Transportation	Intersection	One-way street
Game	Board	Legal move
Citation	Article	Citation
Infectious Diseases	Person	Infection
Food web	Species	Predator-prey relationship

Popular digraph problems

Problem	Description
s->t path	Is there a path from s to t?
Shortest s->t path	What is the shortest path from s to t?
Directed cycle	Is there a directed cycle in the digraph?
Topological sort	Can vertices be sorted so all edges point from earlier to later vertices?
Strong connectivity	Is there a directed path between every pair of vertices?

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

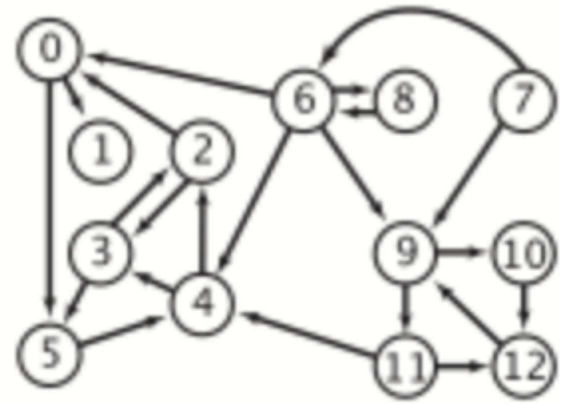
Some slides adopted from Algorithms 4th Edition or COS226

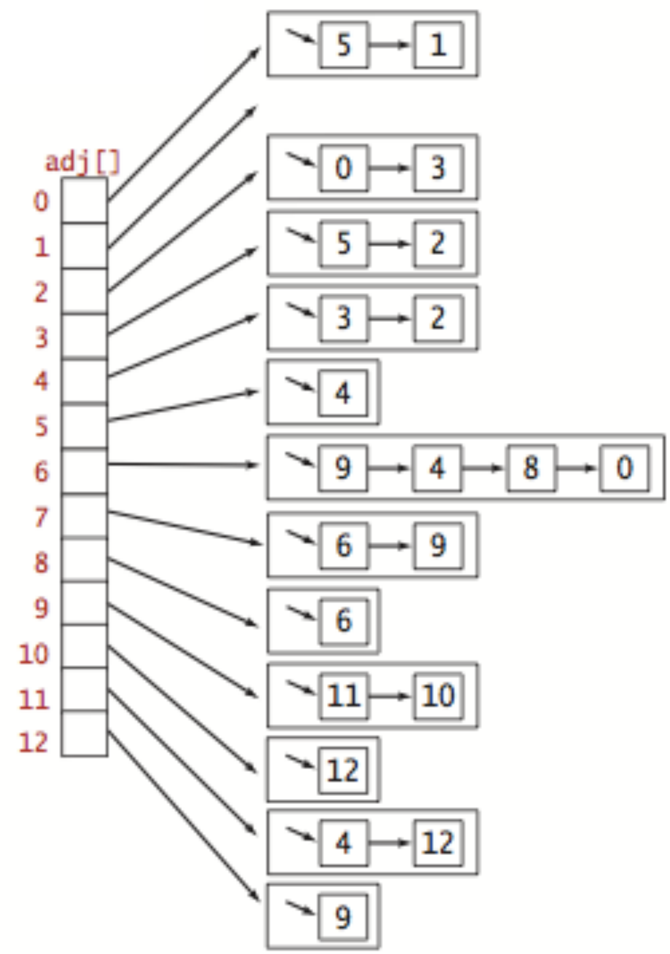
Basic Graph API

- `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.

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 $outdegree(v)$.





Adjacency-list digraph representation in 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];
    }
```

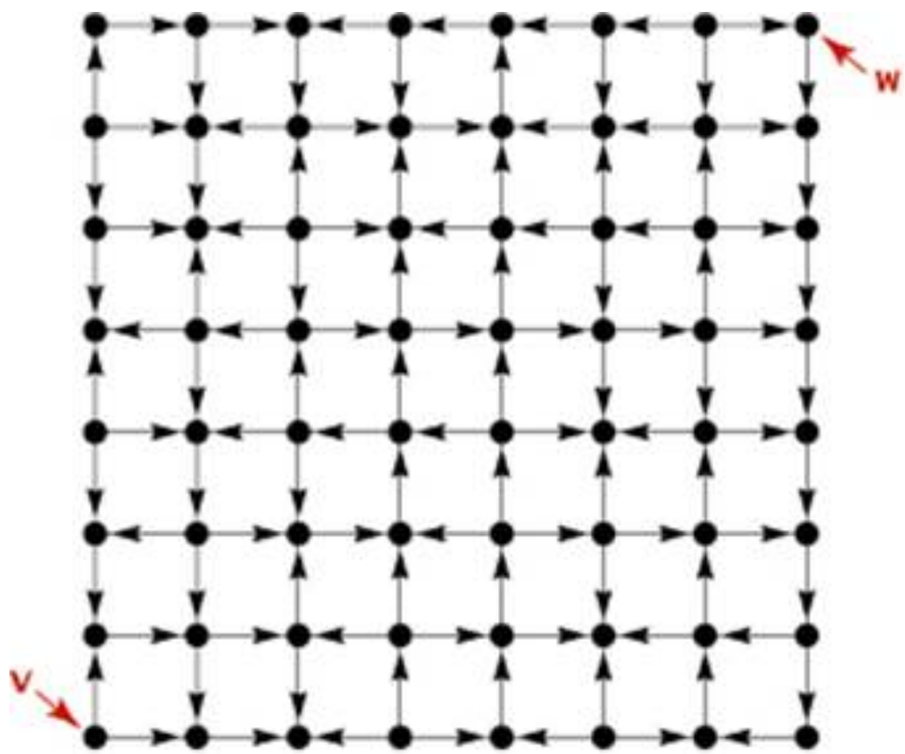
Lecture 24-25: Graphs

- 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.

http://algs4.cs.princeton.edu

4.2 Directed DFS Demo

Directed depth-first search in 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

```
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->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 $|E| + |V|$.

http://algs4.cs.princeton.edu

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.

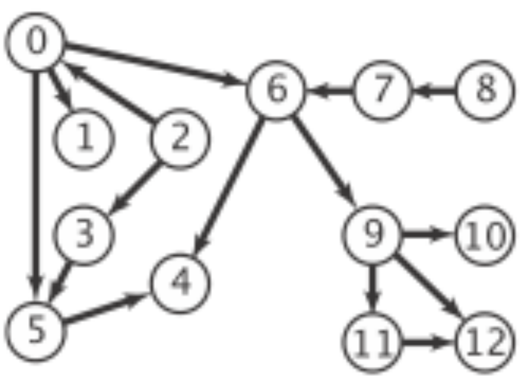
Depth-first orders

```
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*

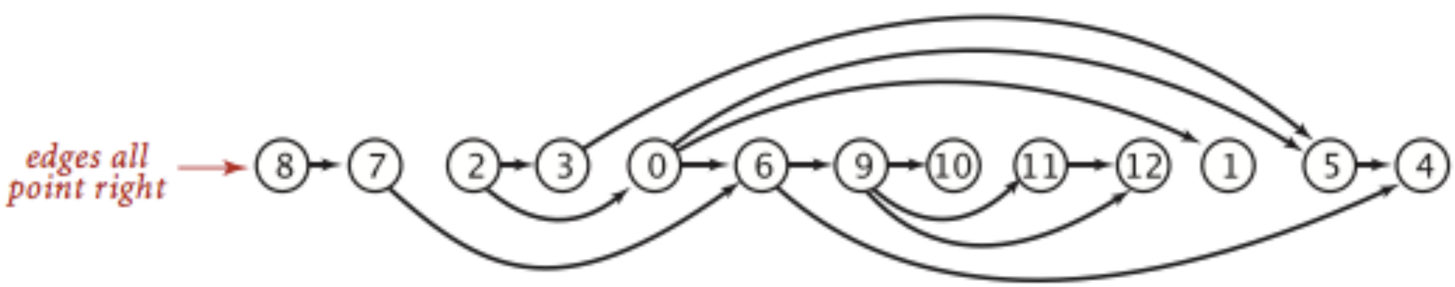
postorder is order in which vertices are done

	pre (*queue*)	post (*queue*)	reversePost (*stack*)
dfs(0)	0		
dfs(5)	0 5		
dfs(4)	0 5 4		
4 done		4	4
5 done		4 5	5 4
dfs(1)	0 5 4 1		
1 done		4 5 1	1 5 4
dfs(6)	0 5 4 1 6		
dfs(9)	0 5 4 1 6 9		
dfs(11)	0 5 4 1 6 9 11		
dfs(12)	0 5 4 1 6 9 11 12		
12 done		4 5 1 12	12 1 5 4
11 done		4 5 1 12 11	11 12 1 5 4
dfs(10)	0 5 4 1 6 9 11 12 10		
10 done		4 5 1 12 11 10	10 11 12 1 5 4
check 12			
9 done		4 5 1 12 11 10 9	9 10 11 12 1 5 4
check 4			
6 done		4 5 1 12 11 10 9 6	6 9 10 11 12 1 5 4
0 done		4 5 1 12 11 10 9 6 0	0 6 9 10 11 12 1 5 4
check 1			
dfs(2)	0 5 4 1 6 9 11 12 10 2		
check 0			
dfs(3)	0 5 4 1 6 9 11 12 10 2 3		
check 5			
3 done		4 5 1 12 11 10 9 6 0 3	3 0 6 9 10 11 12 1 5 4
2 done		4 5 1 12 11 10 9 6 0 3 2	2 3 0 6 9 10 11 12 1 5 4
check 3			
check 4			
check 5			
check 6			
dfs(7)	0 5 4 1 6 9 11 12 10 2 3 7		
check 6			
7 done		4 5 1 12 11 10 9 6 0 3 2 7	7 2 3 0 6 9 10 11 12 1 5 4
dfs(8)	0 5 4 1 6 9 11 12 10 2 3 7 8		
check 7			
8 done		4 5 1 12 11 10 9 6 0 3 2 7 8	8 7 2 3 0 6 9 10 11 12 1 5 4
check 9			
check 10			
check 11			
check 12			

reverse postorder

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.



http://algs4.cs.princeton.edu

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/42digraph/

Practice Problems:

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