34: Undirected Graphs
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.
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 $v-w$:
  - $\text{adj}[v][w] = \text{adj}[w][v] = \text{true}$; (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.

![Adjacency Matrix Example](image-url)
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 \(\text{degree}(v)\).
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 34: Undirected Graphs

- Graph API
- Depth-First Search
- Breadth-First Search
- Connected Components
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.
  - 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 \( \text{marked} \) and \( \text{edgeTo} \) 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 $s$. 

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 Erdős 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.
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 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