CS062
DATA STRUCTURES AND ADVANCED PROGRAMMING

22: Graphs

Alexandra Papoutsaki
she/her/hers
Lecture 22: Graphs

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

Some slides adopted from Algorithms 4th Edition or COS226
Why study graphs?

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

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

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

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components
Graph representation

- **Vertex representation**: Here, integers between 0 and V-1.
  - We will use a 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.
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 ArrayList< ArrayList<Integer> > adj;

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

    //Returns the vertices adjacent to vertex v.
    public Iterable<Integer> adj(int v) {
        return adj.get(v);
    }
}
```
Lecture 22: Graphs

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

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - 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).
Implementation of 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);
            }
        }
    }
}
```
PRACTICE TIME

- Run DFS on the following graph starting at vertex 0 and return the vertices in the order of being marked. Assume that the adj method returns back the adjacent vertices in increasing numerical order.
**DEPTH-FIRST SEARCH**

**ANSWER**

- Vertices marked as visited: 0, 2, 3, 4, 1, 5

<table>
<thead>
<tr>
<th>V</th>
<th>marked</th>
<th>edgeTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>1</td>
</tr>
</tbody>
</table>
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 entries (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 22: Graphs

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

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components
BREADTH-FIRST SEARCH

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 BreadthFirstSearch {
    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 BreadthFirstSearch(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);
                }
            }
        }
    }
}
```
PRACTICE TIME

- Run the BFS on the following graph starting at vertex 0 and return the vertices in the order of being marked. Assume that the adj method returns back the adjacent vertices in increasing numerical order.
BREADTH-FIRST SEARCH

ANSWER

- Vertices marked as visited: 0, 2, 4, 5, 3, 1

```
0  T  -  0
1  T  4  2
2  T  0  1
3  T  2  2
4  T  0  1
5  T  0  1
```
Run DFS and BFS on the following graph starting at vertex s. Assume that the adj method returns back the adjacent vertices in lexicographic order.
Run DFS and BFS on the following graph starting at vertex s. Assume that the adj method returns back the adjacent vertices in lexicographic order.

- **DFS:** s->a->b->e->d->c->f->g->h
- **BFS:** s->a->c->b->d->f->e->g->h
Summary

- **DFS**: Uses recursion.
- **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 22: Graphs

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

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components
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
## 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 22: Graphs

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

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components
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 ArrayList<ArrayList<Integer>> adj;

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

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

    //Returns the vertices adjacent from vertex v.
    public Iterable<Integer> adj(int v) {
        return adj.get(v);
    }
}
```
Lecture 22: Graphs

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

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

- Find all vertices reachable from $s$ along a directed path.
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);
            }
        }
    }
}
```
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 22: Graphs

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

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

- Single-source reachability in a digraph: DFS/BFS.
- Shortest path in a digraph: BFS.
Lecture 22: Graphs

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

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components
Is a digraph strongly connected?

- A **strongly connected digraph** is a directed graph in which it is possible to reach any vertex starting from any other vertex by traversing edges.

- 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 22: Graphs

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

- Directed Graphs
  - Digraph API
  - Depth-First Search
  - Breadth-First Search
  - Strongly Connected Components
Readings:

- Recommended 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/)

Visualization

- [https://visualgo.net/en/dfs bfs](https://visualgo.net/en/dfs bfs)
Problem 1

- What is the maximum number of edges in an undirected graph with \( V \) vertices and no parallel edges?

- What is the minimum number of edges in an undirected graph with \( V \) vertices, none of which are isolated (have degree 0)?

- What is the maximum number of edges in a digraph with \( V \) vertices and no parallel edges?

- What is the minimum number of edges in a digraph with \( V \) vertices, none of which are isolated?
Problem 2

Assume you are given the following 16 edges of an undirected graph with 12 vertices, inserted in an adjacency list in this order:

- 8-4
- 2-3
- 1-11
- 0-6
- 3-6
- 5-2
- 5-10
- 5-0
- 8-1
- 4-1
Problem 3

- Run DFS and BFS on the following digraph starting at vertex 0.
Answer 1

- What is the maximum number of edges in an undirected graph with $V$ vertices and no parallel edges?
  - $n(n - 1)/2$, where $n = |V|$.

- What is the minimum number of edges in an undirected graph with $V$ vertices, none of which are isolated (have degree 0)?
  - $n - 1$.

- What is the maximum number of edges in a digraph with $V$ vertices and no parallel edges?
  - $n(n - 1)$, where $n = |V|$.

- What is the minimum number of edges in a digraph with $V$ vertices, none of which are isolated?
  - $n - 1$. 
Assume you are given the following 16 edges of an undirected graph with 12 vertices, inserted in an adjacency list in this order:

- 8-4
- 11-8
- 2-3
- 2-0
- 1-11
- 6-2
- 0-6
- 5-2
- 3-6
- 5-10
- 10-3
- 5-0
- 7-11
- 8-1
- 7-8
- 4-1
- ...

0 -> 5 -> 2 -> 6
1 -> 4 -> 8 -> 11
2 -> 5 -> 6 -> 0 -> 3
3 -> 10 -> 6 -> 2
4 -> 1 -> 8
5 -> 0 -> 10 -> 2
6 -> 2 -> 3 -> 0
7 -> 8 -> 11
8 -> 1 -> 11 -> 7 -> 4
9 ->
10 -> 5 -> 3
11 -> 8 -> 7 -> 1
**Answer 3**

- DFS - Order of visit: 0, 1, 3, 2, 4, 5, 7, 6

![Graph Diagram]

<table>
<thead>
<tr>
<th>V</th>
<th>marked</th>
<th>edgeTo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>T</td>
<td>5</td>
</tr>
</tbody>
</table>
Answer 3

- BFS - Order of visit: 0, 1, 3, 2, 4, 5, 7, 6

```
V  marked  edgeTo  distTo
---  ------  ------  ------
0   T      -       0
1   T      1       1
2   T      3       2
3   T      1       2
4   T      3       3
5   T      4       4
6   T      7       6
7   T      5       5
```