Objectives

- To model real-world problems using graphs and explain the Seven Bridges of Königsberg problem (§28.1).
- To describe the graph terminologies: vertices, edges, simple graphs, weighted/unweighted graphs, and directed/undirected graphs (§28.2).
- To represent vertices and edges using lists, edge arrays, edge objects, adjacency matrices, and adjacency lists (§28.3).
- To model graphs using the Graph interface, the AbstractGraph class, and the UnweightedGraph class (§28.4).
- To display graphs visually (§28.5).
- To represent the traversal of a graph using the AbstractGraph.Tree class (§28.6).
- To design and implement depth-first search (§28.7).
- To solve the connected-circle problem using depth-first search (§28.8).
- To design and implement breadth-first search (§28.9).
- To solve the nine-tail problem using breadth-first search (§28.10).
Seven Bridges of Königsberg

Basic Graph Terminologies
What is a graph? G=(V, E) = vertices (nodes) and edges
- **Weighted** vs. **Unweighted** graphs
- **Directed** vs. **Undirected** graphs
  - Adjacent vertices share an edge (Adjacent edges)
  - A vertex is **Incident** to an edge that it joins
  - Degree of a vertex = number of edges it joins
  - Neighborhood = subgraph with all adjacent vertices
  - A **Loop** is an edge that connects a vertex to itself
  - A **Cycle** is a path from a vertex to itself via other vertices

Weighted vs Unweighted Graph

Directed vs Undirected Graph
More Graph Terminology

**Parallel edge**: two edges that share the same vertices, also called multiple edges:
- Multiple edges in red, loops in blue

https://en.wikipedia.org/wiki/Multigraph

More Graph Terminology

**Simple graph**: undirected, unweighted, no loops, no parallel edges:

http://mathworld.wolfram.com/SimpleGraph.html

More Graph Terminology

**Complete graph**: simple graph where every pair of vertices are connected by an edge:

https://en.wikipedia.org/wiki/Complete_graph

Representing Graphs

- Representing Vertices
- Representing Edges: Edge Array
- Representing Edges: Edge Objects
- Representing Edges: Adjacency Matrices
- Representing Edges: Adjacency Lists
Representing Vertices (Nodes)

String[] vertices = {"Seattle", "San Francisco", "Los Angles", ... };
List<String> vertices;
or
public class City {
    String name;
}
City[] vertices = {city0, city1, ... };

Representing Edges (Arcs)

int[][] edges = {{0, 1}, {0, 3} {0, 5}, {1, 0}, {1, 2}, ... };
or
public class Edge {
    int u, v;
    public Edge(int u, int v) {
        this.u = u;
        this.v = v;
    }
}
List<Edge> list = new ArrayList<>();
list.add(new Edge(0, 1)); list.add(new Edge(0, 3)); ...

Representing Edges: Adjacency Matrix

int[][] adjacencyMatrix = {
    {0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0}, // Seattle
    {1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0}, // San Francisco
    {0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0}, // Los Angeles
    {1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0}, // Denver
    {0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0}, // Kansas City
    {1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0}, // Chicago
    {0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0}, // Boston
    {0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0}, // New York
    {0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0}, // Atlanta
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0}, // Miami
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1}} // Houston
};

Representing Edges: Adjacency Vertex List

List<Integer>[] neighbors = new List[12];
Seattle neighbors[0] = [0, 1, 2, 10, 11]
San Francisco neighbors[1] = [0, 2, 3, 9, 10]
Los Angeles neighbors[2] = [0, 1, 3, 4, 5]
Denver neighbors[3] = [0, 1, 2, 4, 5, 8, 9, 10]
Kansas City neighbors[4] = [0, 1, 2, 5, 6, 7]
Chicago neighbors[5] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
Boston neighbors[6] = [0, 1, 2]
New York neighbors[7] = [0, 1, 2, 3, 4, 5, 6, 7]
Atlanta neighbors[8] = [0, 1, 2, 3, 4, 5, 6, 7]
Miami neighbors[9] = [0, 1, 2, 3, 4, 5, 6, 7]
Dallas neighbors[10] = [0, 1, 2, 3, 4, 5, 6, 7]
Houston neighbors[11] = [0, 1, 2]
### Representing Edges: Adjacency Edge List

```java
List<Edge>[] neighbors = new List[12];
```

<table>
<thead>
<tr>
<th>City</th>
<th>neighbors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
<td>(Edge 11)</td>
</tr>
<tr>
<td>San Francisco</td>
<td>(Edge 12)</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>(Edge 10)</td>
</tr>
<tr>
<td>Denver</td>
<td>(Edge 11)</td>
</tr>
<tr>
<td>Kansas City</td>
<td>(Edge 9)</td>
</tr>
<tr>
<td>Chicago</td>
<td>(Edge 10)</td>
</tr>
<tr>
<td>Boston</td>
<td>(Edge 9)</td>
</tr>
<tr>
<td>New York</td>
<td>(Edge 8)</td>
</tr>
<tr>
<td>Atlanta</td>
<td>(Edge 11)</td>
</tr>
<tr>
<td>Miami</td>
<td>(Edge 9)</td>
</tr>
<tr>
<td>Dallas</td>
<td>(Edge 10)</td>
</tr>
<tr>
<td>Houston</td>
<td>(Edge 11)</td>
</tr>
</tbody>
</table>

### Representing Adjacency Edge List Using ArrayList

```java
List<List<Edge>> neighbors = new ArrayList<>();
```

```java
neighbors.add(new ArrayList<>());
neighbors.add(new ArrayList<>());
neighbors.add(new ArrayList<>());
neighbors.add(new ArrayList<>());
neighbors.add(new ArrayList<>());
neighbors.add(new ArrayList<>());
neighbors.add(new ArrayList<>());
neighbors.add(new ArrayList<>());
neighbors.add(new ArrayList<>());
neighbors.add(new ArrayList<>());
neighbors.add(new ArrayList<>());
```

### Modeling Graphs

- **Graph**
- **UnweightedGraph**
- **WeightedGraph**

### UnweightedGraph

```java
Graph vertices = new ArrayList<>();
Graph edges = new ArrayList<>();
```

### WeightedGraph

```java
WeightedGraph vertices = new ArrayList<>();
WeightedGraph edges = new ArrayList<>();
```

### Adjacency Edge List

```java
List<Edge> neighbors = new List[12];
```

- `neighbors.add(new Edge(11, 9));`
- `neighbors.add(new Edge(9, 11));`
- `neighbors.add(new Edge(8, 7));`
- `neighbors.add(new Edge(7, 5));`
- `neighbors.add(new Edge(5, 3));`
- `neighbors.add(new Edge(4, 3));`
- `neighbors.add(new Edge(3, 1));`
- `neighbors.add(new Edge(2, 3));`
- `neighbors.add(new Edge(1, 2));`
- `neighbors.add(new Edge(0, 3));`
- `neighbors.add(new Edge(11, 10));`
- `neighbors.add(new Edge(8, 10));`
- `neighbors.add(new Edge(2, 10));`
- `neighbors.add(new Edge(5, 6));`
- `neighbors.add(new Edge(4, 7));`
- `neighbors.add(new Edge(3, 4));`
- `neighbors.add(new Edge(2, 4));`
- `neighbors.add(new Edge(1, 2));`
- `neighbors.add(new Edge(0, 1));`
- `neighbors.add(new Edge(1, 0));`
- `neighbors.add(new Edge(2, 1));`
- `neighbors.add(new Edge(1, 3));`
- `neighbors.add(new Edge(11, 8));`
- `neighbors.add(new Edge(11, 11));`
- `neighbors.add(new Edge(11, 10));`
Graph Visualization

Graph Traversals
Depth-first search and breadth-first search

Both traversals result in a spanning tree, which can be modeled using a class.

```
UnWeightedGraph<V> graph;

for (int i = 0; i < vertexCount; i++)
    if (vertexStates[i] == 0)
        DFS(i);
```

The root of the tree.
The parent of the vertices.
The order for traversing the vertices.
Constructs a tree with the specified root, parent, and search order.
Returns the root of the tree.
Returns the order of vertices searched.
Returns the parent for the specified vertex index.
Returns the number of vertices searched.
Returns a list of vertices from the specified vertex index to the root.
Displays a path from the root to the specified vertex.
Displays tree with the root and all edges.

Depth-First Search

The depth-first search of a graph is like the depth-first search of a tree discussed in §25.2.3, “Tree Traversal.” For trees, the search starts from the root. In a graph, the search can start from any vertex:

**Input:** G = (V, E) and a starting vertex v
**Output:** a DFS tree rooted at v

Tree dfs(vertex v) {
    visit v;
    for each neighbor w of v
        if (w has not been visited) {
            set v as the parent for w;
            dfs(w);
        }
}

Depth-First Search Example
Depth-First Search Example

Applications of the DFS
- Detecting whether a graph is connected. Search the graph starting from any vertex. If the number of vertices searched is the same as the number of vertices in the graph, the graph is connected. Otherwise, the graph is not connected.
- Detecting whether there is a path between two vertices.
- Finding a path between two vertices.
- Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.
- Detecting whether there is a cycle in the graph, and finding a cycle in the graph.

Breadth-First Search
The breadth-first traversal of a graph is like the breadth-first traversal of a tree discussed in §25.2.3, “Tree Traversal.” With breadth-first traversal of a tree, the nodes are visited level by level. First the root is visited, then all the children of the root, then the grandchildren of the root from left to right, and so on.
Breadth-First Search

Input: \( G = (V, E) \) and a starting vertex \( v \)
Output: a BFS tree rooted at \( v \)

\[
\text{bfs}(\text{vertex } v) \{ \\
\quad \text{create an empty queue for storing vertices to be visited;} \\
\quad \text{add } v \text{ into the queue;} \\
\quad \text{mark } v \text{ visited;} \\
\quad \text{while the queue is not empty} \{ \\
\quad \quad \text{dequeue a vertex, say } u, \text{ from the queue} \\
\quad \quad \text{process } u; \\
\quad \quad \text{for each neighbor } w \text{ of } u \\
\quad \quad \quad \text{if } w \text{ has not been visited} \{ \\
\quad \quad \quad \quad \text{add } w \text{ into the queue;} \\
\quad \quad \quad \quad \text{set } u \text{ as the parent for } w; \\
\quad \quad \quad \quad \text{mark } w \text{ visited;} \\
\quad \quad \} \\
\quad \} \\
\}
\]

Breadth-First Search Example

Queue: 0
Queue: 1 2 3
Queue: 2 3 4
isVisited[0] = true
isVisited[1] = true, isVisited[2] = true,
isVisited[3] = true
isVisited[4] = true

Applications of the BFS

- Detecting whether a graph is connected. A graph is connected if there is a path between any two vertices in the graph.
- Detecting whether there is a path between two vertices.
- Finding a shortest path between two vertices. You can prove that the path between the root and any node in the BFS tree is the shortest path between the root and the node.
- Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.
- Detecting whether there is a cycle in the graph, and finding a cycle in the graph.
- Testing whether a graph is bipartite. A graph is bipartite if the vertices of the graph can be divided into two disjoint sets such that no edges exist between vertices in the same set.