Chapter 28 Graphs and Applications

CS2: Data Structures and Algorithms
Colorado State University

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).
Modeling Using Graphs

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

![Unweighted Graph](https://www.slideshare.net/emersonferr/20-intro-graphs)

![Weighted Graph](https://www.slideshare.net/emersonferr/20-intro-graphs)
Directed vs Undirected Graph

Directed Graph (a)

Undirected Graph (b)

Adjacency Matrix Representation (a)

Adjacency Matrix Representation (b)

More Graph Terminology

Parallel edge: two edges that share the same vertices, also called multiple edges:

- Multiple edges in red, loops in blue

Parallel edge: two edges that share the same vertices, also called multiple edges:

- Multiple edges in red, loops in blue
More Graph Terminology

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

![Simple Graph](http://mathworld.wolfram.com/SimpleGraph.html)

More Graph Terminology

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

![Complete Graph](https://en.wikipedia.org/wiki/Complete_graph)
Representing Vertices (Nodes)

String[] vertices = {“Seattle“, “San Francisco“, “Los Angeles”, … };  
List<String> vertices;  
or

public class City {
    String name;
}

City[] vertices = {city0, city1, … };
Representing Edges (Arcs)

```java
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

```java
int[][] adjacencyMatrix = {
    {0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0}; // Seattle
    {1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0}; // San Francisco
    {0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0}; // Los Angeles
    {1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0}; // Denver
    {0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0}; // Kansas City
    {1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0}; // Chicago
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; // Boston
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; // New York
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; // Atlantic
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; // Miami
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; // Dallas
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; // Houston
};
```
Representing Edges: Adjacency Vertex List

List<Integer>[] neighbors = new List[12];

Seattle
San Francisco
Los Angeles
Denver
Kansas City
Chicago
Boston
New York
Atlanta
Miami
Dallas
Houston

neighbors[0] 1 3 5
neighbors[1] 0 2 3
neighbors[2] 1 3 4 10
neighbors[3] 0 1 2 4 5
neighbors[4] 2 3 5 7 8 10
neighbors[5] 0 3 4 6 7
neighbors[6] 5 7
neighbors[7] 4 5 6 8
neighbors[8] 4 7 9 10 11
neighbors[9] 8 11
neighbors[10] 2 4 8 11

List<List<Integer>> neighbors = new ArrayList<>();

Liang, Introduction to Java Programming, Tenth Edition, (c) 2013 Pearson Education, Inc. All rights reserved.

Representing Edges: Adjacency Edge List

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

Seattle
San Francisco
Los Angeles
Denver
Kansas City
Chicago
Boston
New York
Atlanta
Miami
Dallas
Houston

neighbors[0] Edge(0, 11) Edge(0, 3) Edge(0, 5)
neighbors[1] Edge(1, 0) Edge(1, 2) Edge(1, 3)
neighbors[2] Edge(2, 1) Edge(2, 3) Edge(2, 4) Edge(2, 10)
neighbors[3] Edge(3, 0) Edge(3, 1) Edge(3, 2) Edge(3, 4) Edge(3, 5)
neighbors[5] Edge(5, 0) Edge(5, 3) Edge(5, 4) Edge(5, 6) Edge(5, 7)
neighbors[6] Edge(6, 5)
neighbors[7] Edge(7, 4) Edge(7, 5) Edge(7, 6) Edge(7, 8)
neighbors[8] Edge(8, 4) Edge(8, 7) Edge(8, 9) Edge(8, 10) Edge(8, 11)
neighbors[9] Edge(9, 8) Edge(9, 11)
neighbors[10] Edge(10, 2) Edge(10, 4) Edge(10, 8) Edge(10, 11)
Representing Adjacency Edge List Using ArrayList

```java
List<ArrayList<Edge>> neighbors = new ArrayList<>();
neighbors.add(new ArrayList<Edge>());
neighbors.get(0).add(new Edge(0, 1));
neighbors.get(0).add(new Edge(0, 3));
neighbors.get(0).add(new Edge(0, 5));
neighbors.add(new ArrayList<Edge>());
neighbors.get(1).add(new Edge(1, 0));
neighbors.get(1).add(new Edge(1, 2));
neighbors.get(1).add(new Edge(1, 3));
neighbors.get(1).add(new Edge(1, 8));
neighbors.get(11).add(new Edge(11, 8));
neighbors.get(11).add(new Edge(11, 9));
neighbors.get(11).add(new Edge(11, 10));
```

The generic type `V` is the type for vertices.

### Graph

- `getSize()`: Returns the number of vertices in the graph.
- `getVertices()`: Returns the vertices in the graph.
- `getVertex(index: int)`: Returns the vertex object for the specified vertex index.
- `getIndex(v: V)`: Returns the index for the specified vertex.
- `getNeighbors(index: int)`: Returns the neighbors of vertex with the specified index.
- `getDegree(index: int)`: Returns the degree for a specified vertex index.
- `printEdges()`: Prints the edges.
- `clear()`: Clears the graph.
- `addVertex(v: V)`: Returns true if v is added to the graph. Returns false if v is already in the graph.
- `addEdge(u: int, v: int)`: Adds an edge from u to v to the graph throws IllegalArgumentException if u or v is invalid. Returns true if the edge is added and false if (u, v) is already in the graph.
- `remove(v: V)`: Removes a vertex from the graph.
- `remove(u: int, v: int)`: Removes an edge from the graph.
- `dfs(v: int)`: Obtains a depth-first search tree starting from v.
- `bfs(v: int)`: Obtains a breadth-first search tree starting from v.

### UnweightedGraph

- `vertices: List<V>`: Vertices in the graph.
- `neighbors: List<List<Edge>>`: Neighbors for each vertex in the graph.
- `UnweightedGraph()`: Constructs an empty graph.
- `UnweightedGraph(vertices: V[], edges: int[][][])`: Constructs a graph with the specified edges and vertices stored in arrays.
- `UnweightedGraph[vertices: List<V>, edges: List<List<Edge>>]`: Constructs a graph with the specified edges and vertices stored in lists.
- `UnweightedGraph[edges: int[][]], numberOfVertices: int]`: Constructs a graph with the specified edges in an array and the integer vertices 1, 2, ...
- `UnweightedGraph[edges: List<Edge>, numberOfVertices: int]`: Constructs a graph with the specified edges in a list and the integer vertices 1, 2, ...
Graph Traversals

Depth-first search and breadth-first search

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

```java
UnweightedGraph<V>.SearchTree

-root: int
-parent: int[]
-searchOrder: List<Integer>

+SearchTree(root: int, parent: int[], searchOrder: List<Integer>)
+getRoot(): int
+getSearchOrder(): List<Integer>
+getParent(index: int): int
+getNumberOfVerticesFound(): int
+getPath(index: int): List<V>
+printPath(index: int): void
+printTree(): void

The root of the tree.
The parents of the vertices.
The orders for traversing the vertices.
Constructs a tree with the specified root, parent, and searchOrder.
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 builds a “spanning tree” of all of the reachable edges from the starting vertex v, using marking of the visited nodes:

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

Tree \( \text{dfs}(\text{vertex } v) \) {  
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.
DFS: Depth First Search

- Explores edges from the most recently discovered node; backtracks when reaching a dead-end. The book does not use white, grey, black, but uses visited (and implicitly unexplored). Recursive

```java
DFS(u):
    mark u as visited
    for each edge (u,v) :
        if v is not marked visited :
            DFS(v)
```

DFS and cyclic graphs

- When DFS visits a node for the first time it is white. There are two ways DFS can revisit a node:
  
  1. DFS has already fully explored the node. What color does it have then? Is there a cycle then?

  2. DFS is still exploring this node. What color does it have in this case? Is there a cycle then?
DFS and cyclic graphs

- There are two ways DFS can revisit a node:
  1. DFS has already fully explored the node. **What color does it have then?** 
     - Is there a cycle then?

  2. DFS is still exploring this node. **What color does it have in this case?** 
     - Is there a cycle then?

So DFS with the white, grey, black coloring scheme detects a cycle when a GREY node is visited
Cycle detection: DFS + coloring

When a grey (frontier) node is visited, a cycle is detected.

Recursive / node coloring version

DFS(u):

# c: color, p: parent

c[u]=grey

forall v in Adj(u):
    if c[v]==white:
        parent[v]=u
        DFS(v)

c[u]=black
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

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

Queue: 0
isVisited[0] = true

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

Queue: 2 3 4
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.

Graphs Describing Precedence

❖ Edge from \( x \) to \( y \) indicates \( x \) should come before \( y \), e.g.:
  – prerequisites for a set of courses
  – dependences between programs
  – set of tasks, e.g. building a computer
Graphs Describing Precedence

- Want an ordering of the vertices of the graph that respects the precedence relation
  - Example: An ordering of CS courses
- The graph must not contain cycles. **WHY?**
Topological Sorting of DAGs

- **DAG**: Directed Acyclic Graph
- **Topological sort**: listing of nodes such that if \((a, b)\) is an edge, \(a\) appears before \(b\) in the list
- **Is a topological sort unique?**

*Question: Is a topological sort unique?*
A directed graph without cycles

\[ a \rightarrow b \rightarrow c \]
\[ d \rightarrow e \rightarrow f \]
\[ g \]

\[ a, g, d, b, e, c, f \]
\[ a, b, g, d, e, f, c \]

Topological Sort Algorithm

- Modification of DFS: Traverse tree using DFS starting from all nodes that have no predecessor.

- Add a node to the list when ready to backtrack.
Topological Sort Algorithm

List toppoSor{t(Graph theGraph)
    // use stack stck and list lst
    // push all roots
    for (all vertices v in the graph theGraph)
        if (v has no predecessors)
            stck.push(v)
            Mark v as visited
    // DFS
    while (!stck.isEmpty())
        if (all vertices adjacent to the vertex on top of the stack have been visited)
            v = stck.pop()
            lst.add(0, v)
        else
            Select an unvisited vertex u adjacent to vertex v on top of the stack
            stck.push(u)
            Mark u as visited
            Set v as parent of u
    return lst

Algorithm 2: Example 1

![Diagram of a directed graph with vertices labeled a, b, c, d, e, f, g, and edges connecting them in a specific order.](image-url)
Topological sorting solution

Red edges represent spanning tree.