Graphs and Applications

Paul Fodor

CSE260, Computer Science B: Honors

Stony Brook University

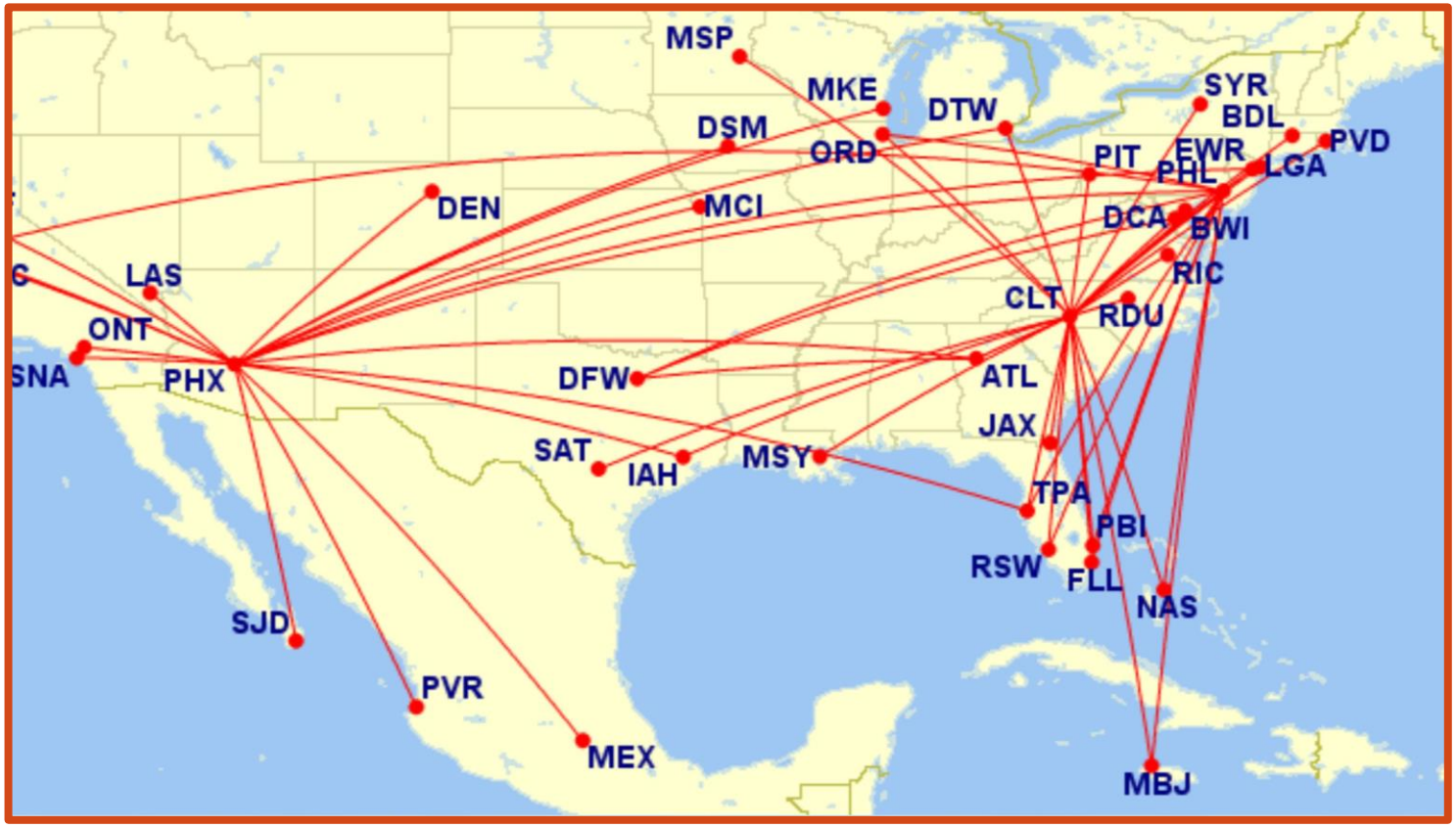
http://www.cs.stonybrook.edu/~cse260

Objectives

- To model real-world problems using graphs
  - Explain the Seven Bridges of Königsberg problem
- To describe the graph terminologies: ***vertices, edges, simple graphs, weighted/unweighted graphs, and directed/undirected graphs***
- To **represent vertices and edges** using lists, edge arrays, edge objects, adjacency matrices, and adjacency lists
- To model graphs using the **`Graph`** interface, the **`AbstractGraph`** class, and the **`UnweightedGraph`** class
- To display graphs visually
- To represent the traversal of a graph using the **`AbstractGraph.Tree`**
- To design and implement depth-first search
- To solve the connected-circle problem using depth-first search
- To design and implement breadth-first search
- To solve the nine-tail problem using breadth-first search

Modeling Using Graphs

- Graphs are useful in modeling and solving real-world problems
  - For example, the problem to find the least number of flights between two cities is to find a shortest path between two vertices in a graph



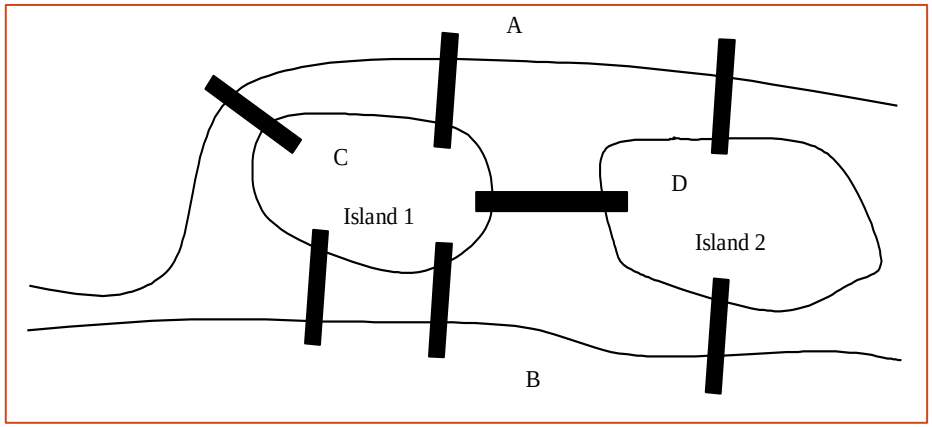
Modeling Using Graphs

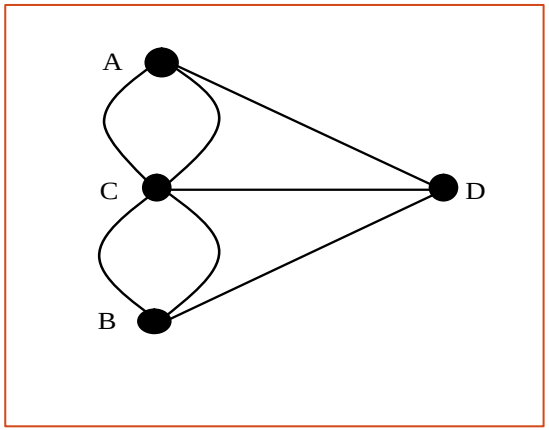
- Many practical problems can be represented by graphs because graphs are used to represent:
  - networks of communication
  - flow of computation
  - social media
  - travel
  - computer chip design
  - mapping the progression of neuro-degenerative diseases
- The development of algorithms to handle graphs is therefore of major interest in computer science.
- The transformation of graphs is often formalized and represented by graph rewrite systems.
- Understanding real-world systems as a network is called network science.

Seven Bridges of Königsberg

- The study of graph problems is known as ***graph theory.***
  - It was founded by Leonhard Euler in 1736, when he introduced graph terminology to solve the famous Seven Bridges of Königsberg problem"
    - The city of Königsberg, Prussia (now Kaliningrad, Russia), was divided by the Pregel River.
      - There were two islands on the river.
      - The city and islands were connected by seven bridges.
    - Euler replaced each land mass with a ***vertex*** or a ***node***, and each bridge with an ***edge***:

Leonhard Euler

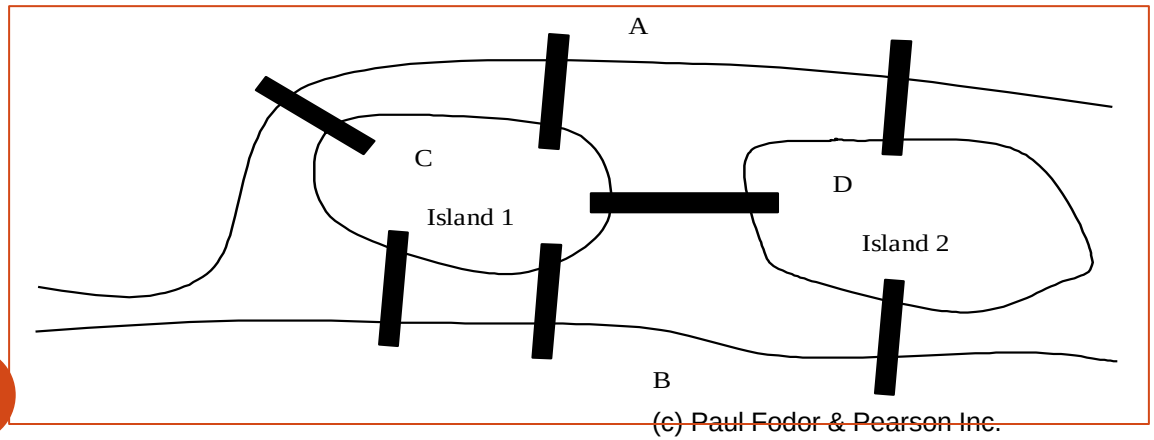


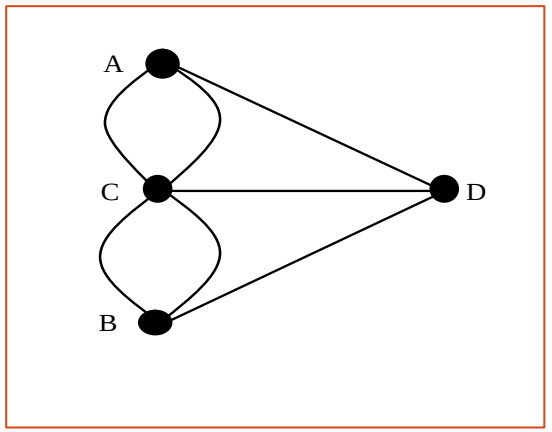


Seven Bridges of Königsberg

- The question was: can one take a walk, cross each bridge exactly once, and return to the starting point?
  - ***Is there a path starting from any vertex, traversing all edges exactly once, and returning to the starting vertex?***
    - Euler proved that for such a path to exist, each vertex must have an even number of edges
    - Therefore, the Seven Bridges of Königsberg problem has no solution!

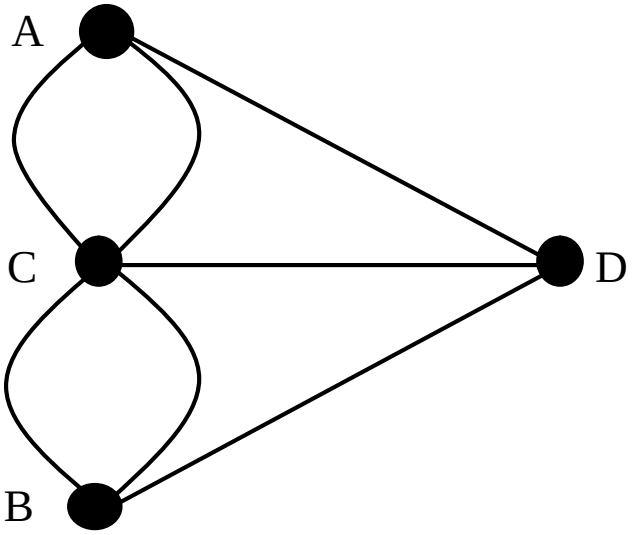
Swiss mathematician
Leonhard Euler
(15 April 1707 – 18 September 1783)

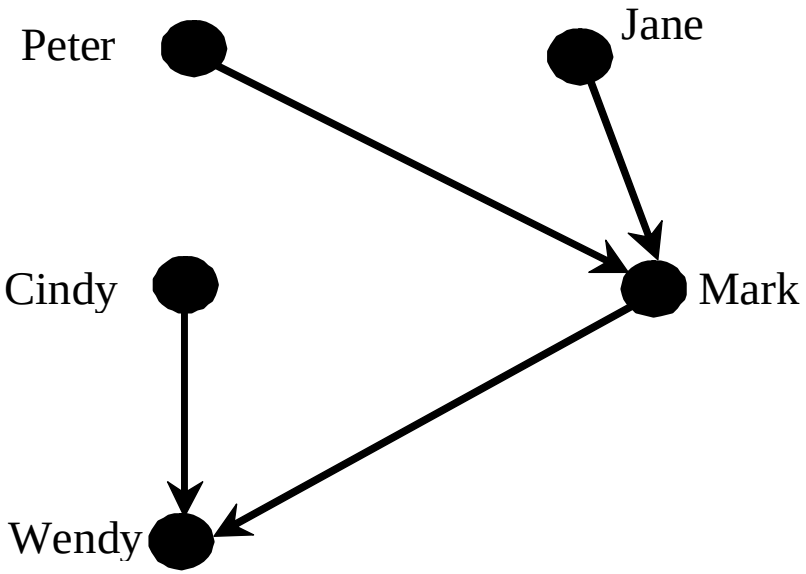




Basic Graph Terminologies

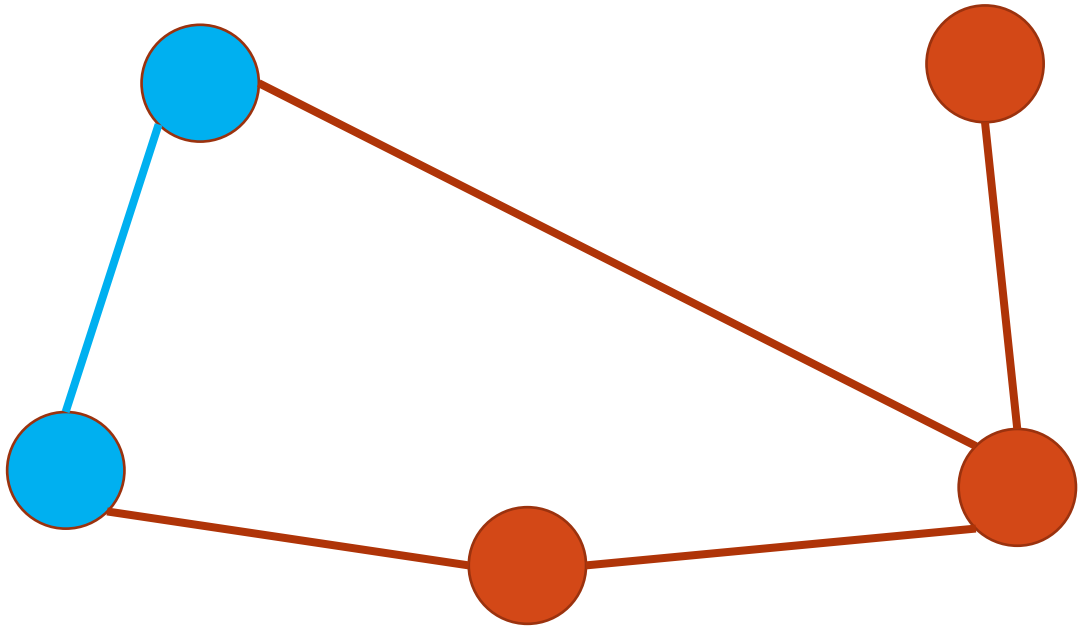
- A *graph* **G = (V, E),** where **V** represents a set of vertices and **E** represents a set of edges.
- A graph may be ***undirected*** or ***directed***





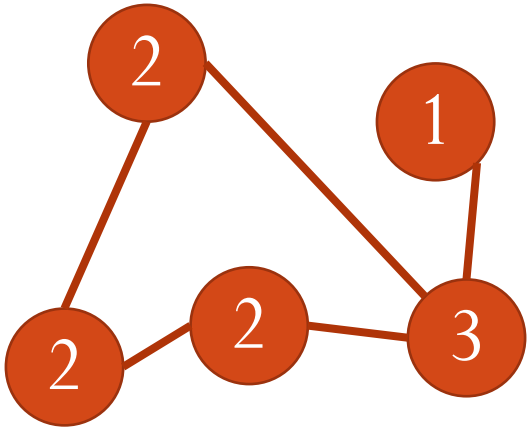
Adjacent Vertices

- Two vertices in a graph are said to be ***adjacent*** (or ***neighbors***) if they are connected by an edge
- An edge in a graph that joins two vertices is said to be ***incident*** to both vertices



Degree

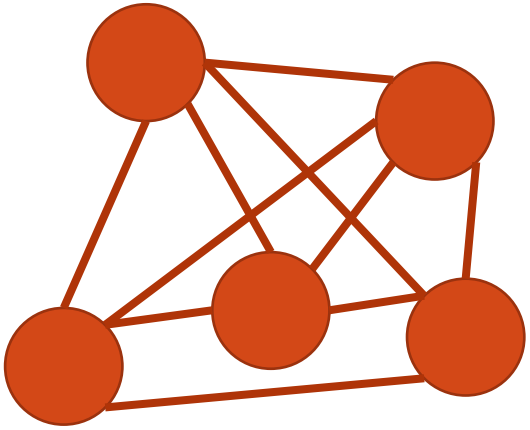
The ***degree*** of a vertex is the number of edges incident to it

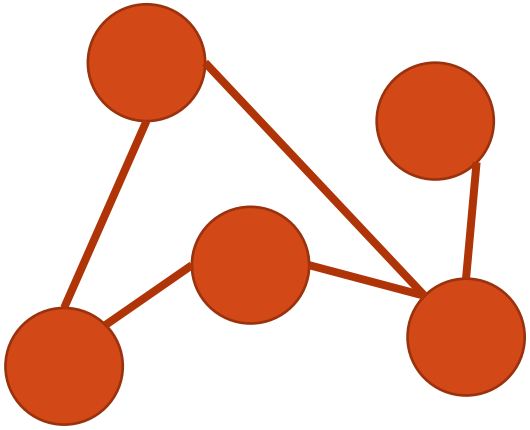


Complete

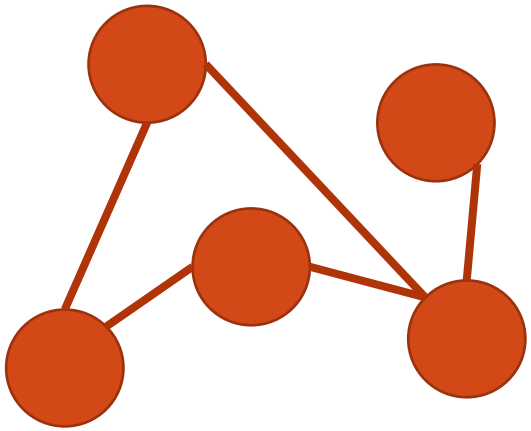
Incomplete

every two pairs of vertices are connected

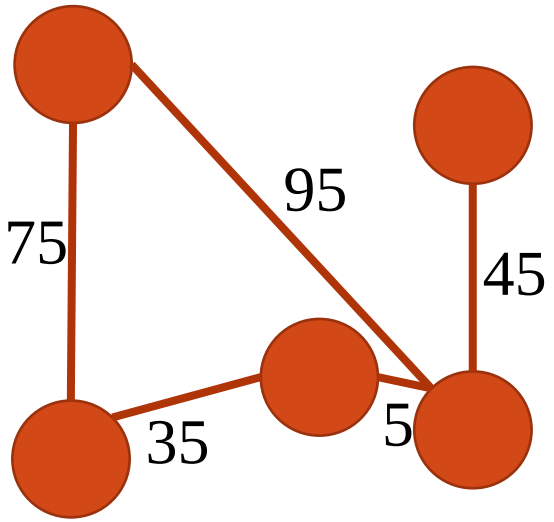




Unweighted

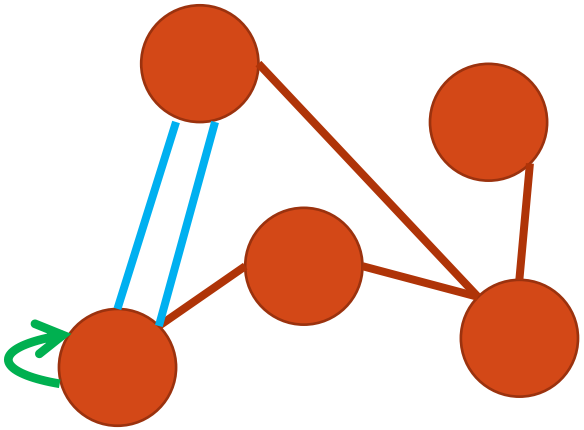


Weighted



Parallel Edges

If two vertices are connected by two or more edges, these edges are called ***parallel edges***

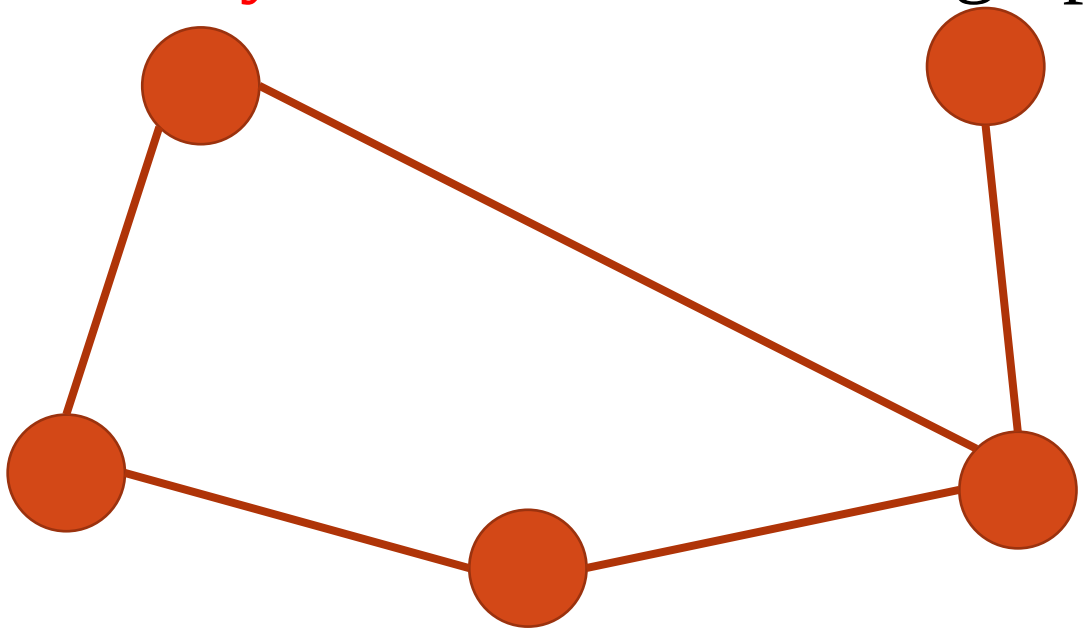


A ***loop*** is an edge that links a vertex to itself

A ***simple graph*** is one that has doesn't have any parallel edges or loops

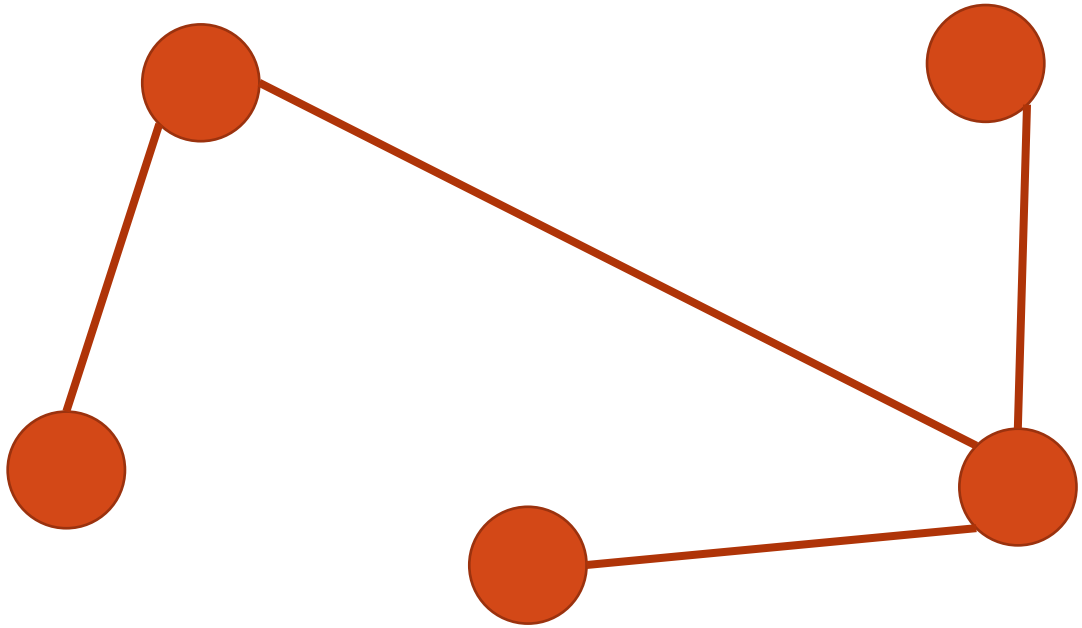
Connected graph

- A graph is ***connected*** if there exists a **path** between any two vertices in the graph



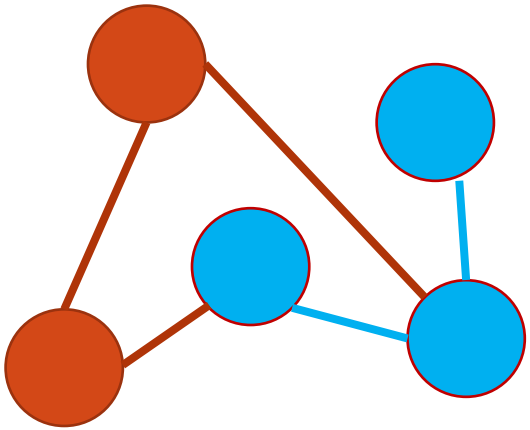
Tree

- A connected graph is a ***tree*** if it does not have cycles



Subgraph

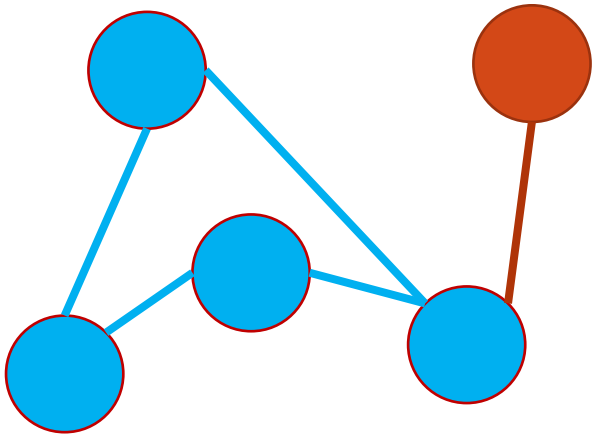
A ***subgraph*** of a graph G is a graph whose vertex set is a subset of that of G and whose edge set is a subset of that of G



Cycle

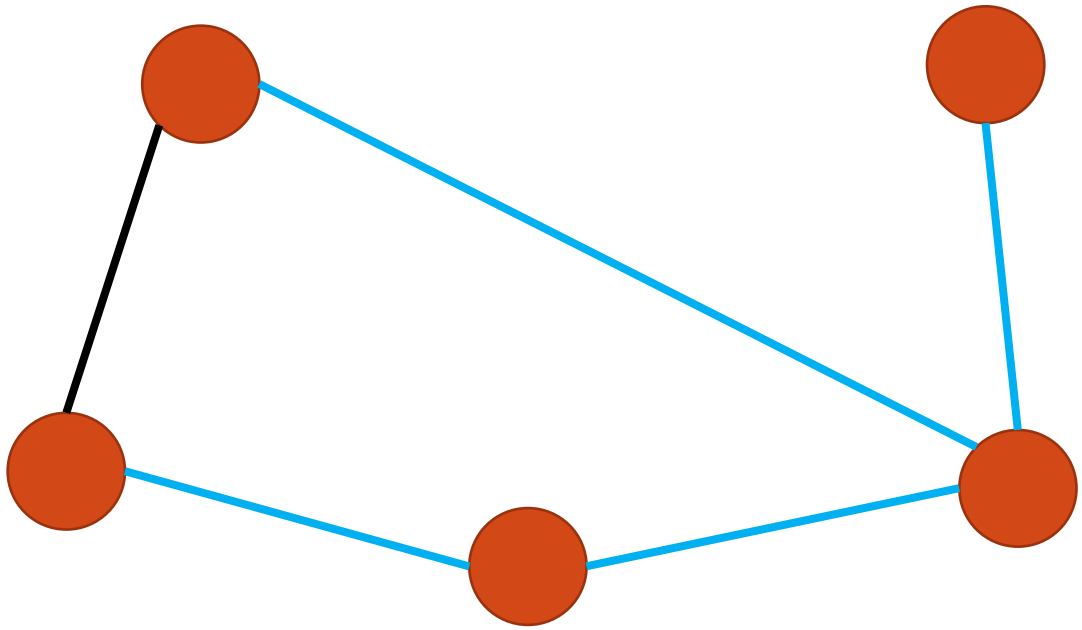
A ***closed path*** is a path where all vertices have 2 edges incident to them

A ***cycle*** is a closed path that starts from a vertex and ends at the same vertex



Spanning Tree

A ***spanning tree*** of a graph G is a connected subgraph of G and the subgraph **is a tree that contains all vertices in G**



Representing Graphs

- Representing Vertices
- Representing Edges: Edge Array
- Representing Edges: Edge Objects
- Representing Edges: Adjacency Matrices
- Representing Edges: Adjacency Lists

Representing Vertices

```
String[] vertices = {"Seattle",
"San Francisco", "Los Angles",
"Denver", "Kansas City", "Chicago",…}
```

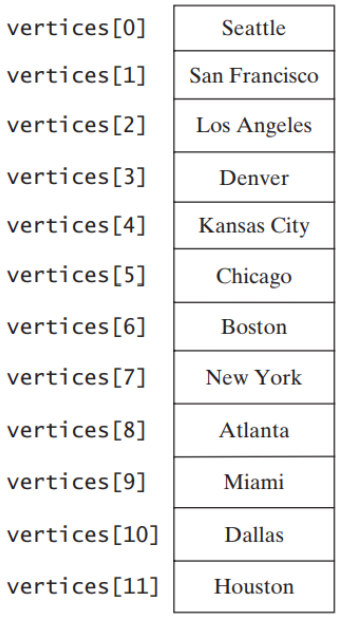
The vertices can be conveniently labeled using natural numbers 0, 1, 2, …, n–1, for a graph for n vertices

OR

```
List<String> vertices;
```

OR

```
public class City {
     private String cityName;
}
City[] vertices = {city0, city1, … };
```



Representing Edges: Edge Array

- The edges can be represented using a two-dimensional array of all the edges:

```
int[][] edges = {
 {0, 1}, {0, 3}, {0, 5},
 {1, 0}, {1, 2}, {1, 3},
 {2, 1}, {2, 3}, {2, 4}, {2, 10},
 {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
 {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
 {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
 {6, 5}, {6, 7},
 {7, 4}, {7, 5}, {7, 6}, {7, 8},
 {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
 {9, 8}, {9, 11},
 {10, 2}, {10, 4}, {10, 8}, {10, 11},
 {11, 8}, {11, 9}, {11, 10}
 };
```

Representing Edges: Edge Objects

```
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));
```

- Storing **`Edge`** objects in an **`ArrayList`** is useful if you don't know the edges in advance

Representing Edges: Adjacency Matrix

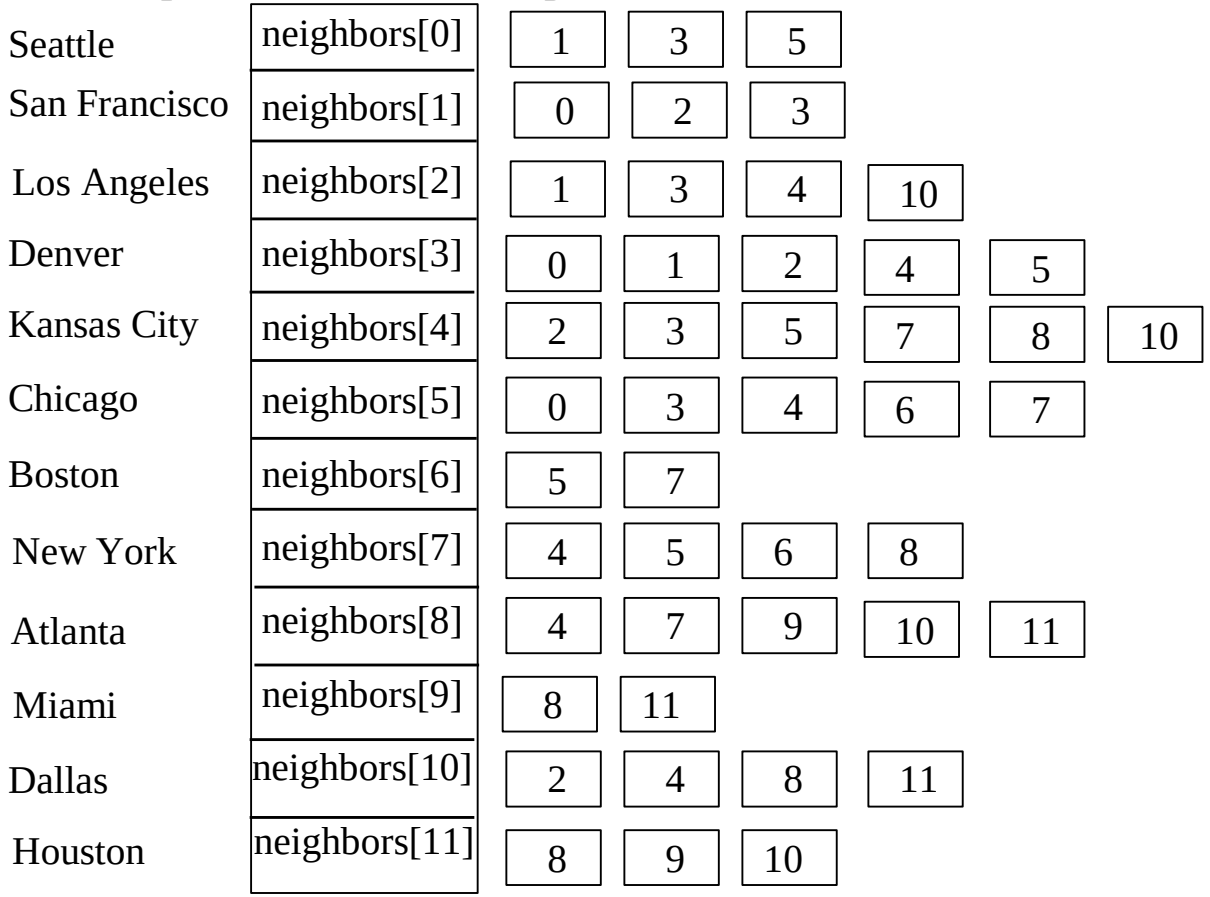
- Assume that the graph has **N** vertices and we can use a two-dimensional **N** * **N** matrix to represent the existence of edges

```
int[][] adjacencyMatrix = {
   {0, 1, 0, 1, 0, 1, 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, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0}, // Denver
   {0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0}, // Kansas City
   {1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0}, // Chicago
   {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0}, // Boston
   {0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0}, // New York
   {0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1}, // Atlanta
   {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1}, // Miami
   {0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1}, // Dallas
   {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0}  // Houston
};
```

- Since the matrix is symmetric for an undirected graph, to save storage we can use a ragged array

Representing Edges: Adjacency Vertex List

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

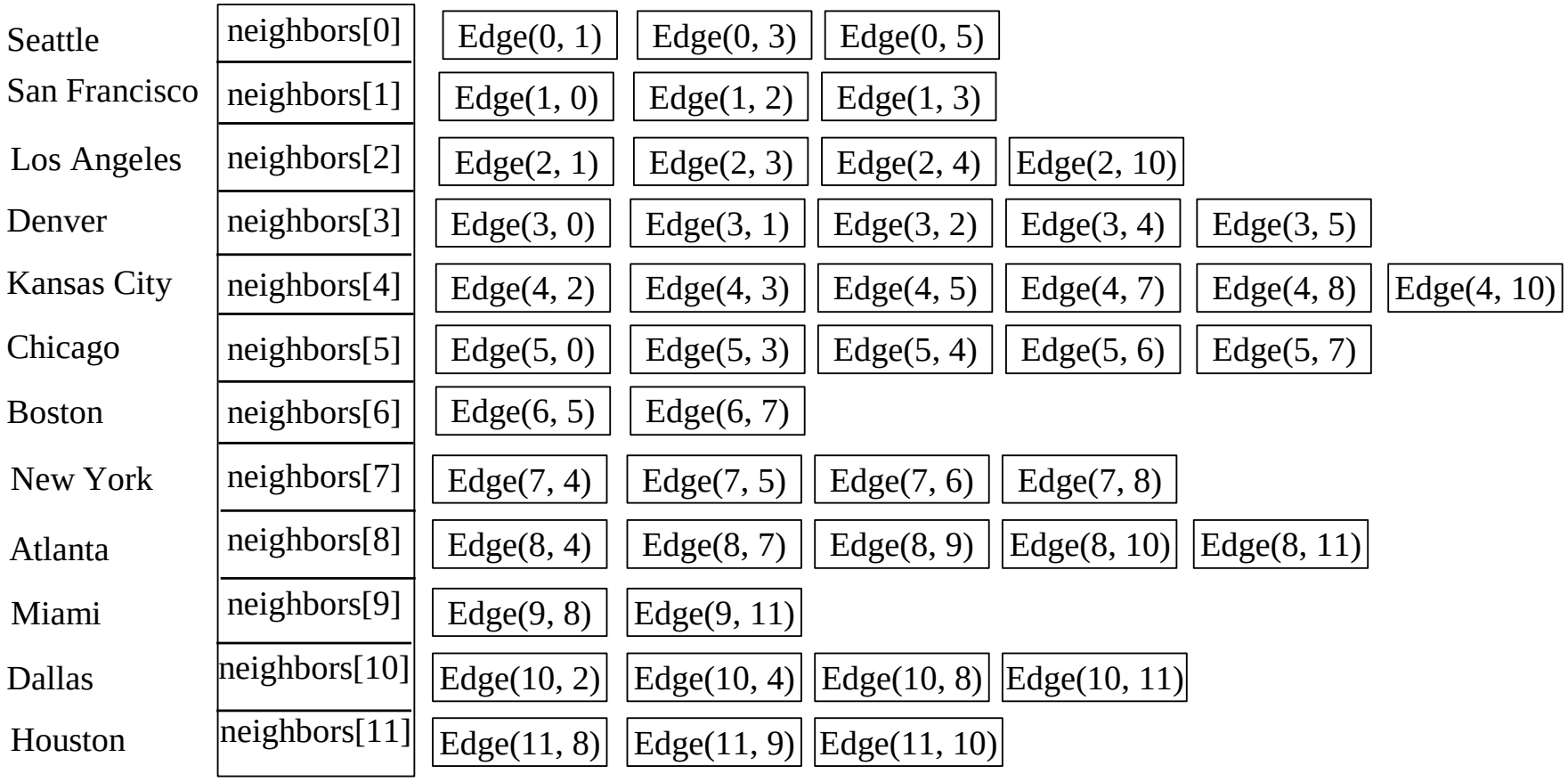


OR

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

Representing Edges: Adjacency Edge List

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

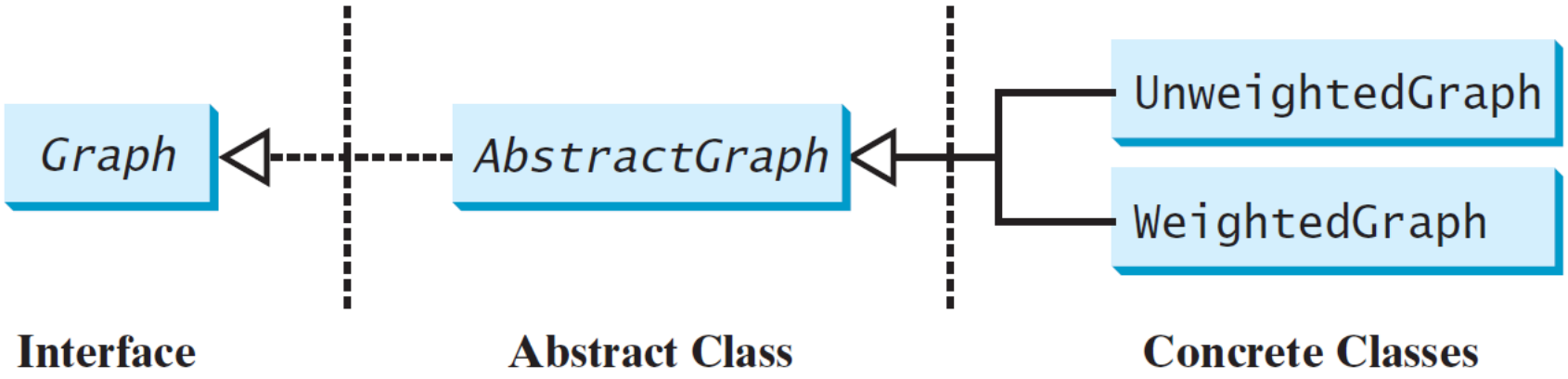


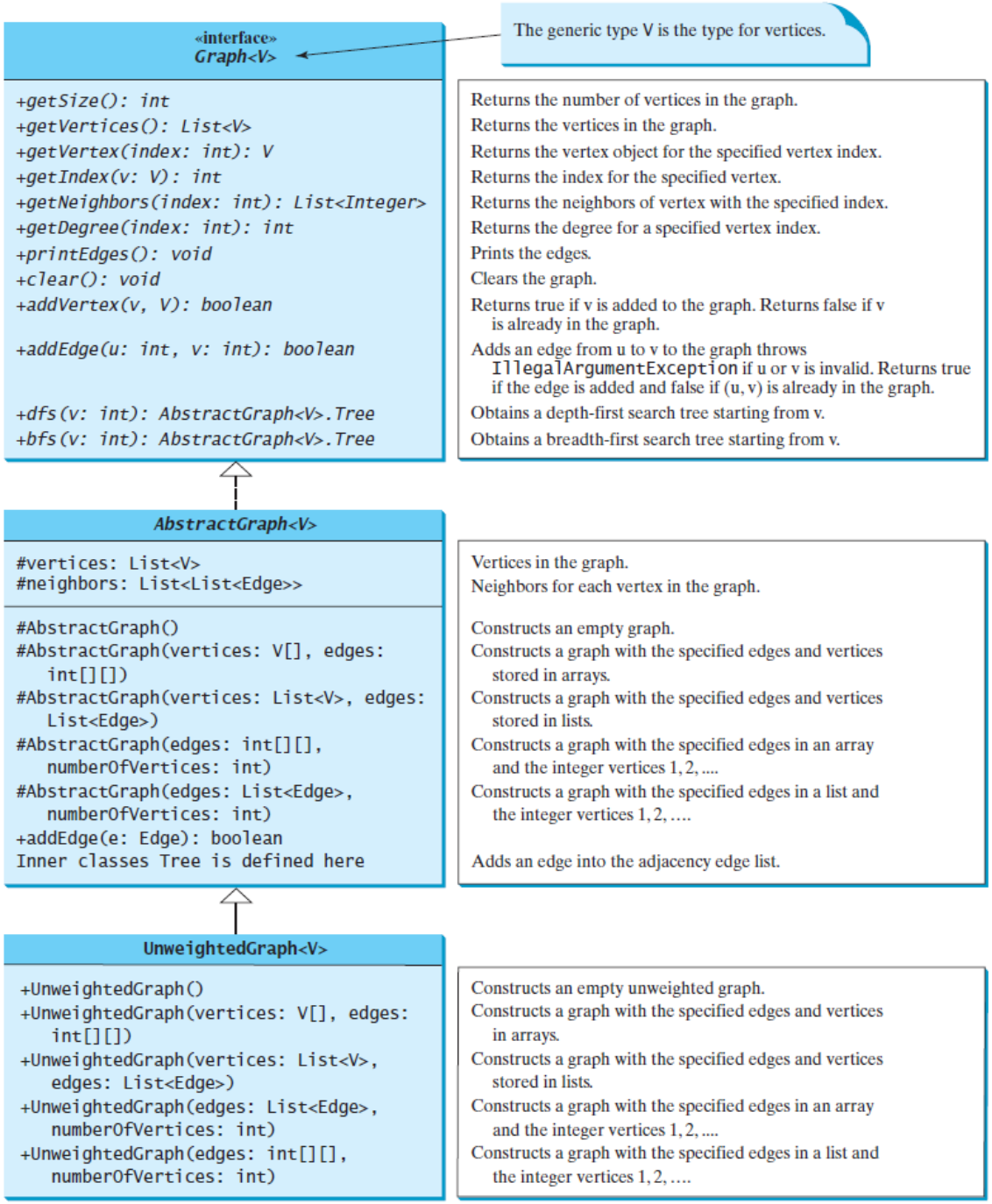
Representing Adjacency Edge List Using ArrayList

```
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.add(new ArrayList<Edge>());
neighbors.get(11).add(new Edge(11, 8));
neighbors.get(11).add(new Edge(11, 9));
neighbors.get(11).add(new Edge(11, 10));
```

Modeling Graphs

- The ***Graph*** interface defines the common operations for a graph
- An ***abstract*** class named ***AbstractGraph*** that partially implements the ***Graph*** interface




«interface»
Graph<V>
The generic type V is the type for vertices.
+getSize(): int
+getVertices(): List<V>
+getVertex(index: int): V
+getIndex(v: V): int
+getNeighbors(index: int): List<Integer>
+getDegree(index: int): int
+printEdges(): void
+clear(): void
+addVertex(v, V): boolean
+addEdge(u: int, v: int): boolean
+dfs(v: int): AbstractGraph<V>.Tree
+bfs(v: int): AbstractGraph<V>.Tree
Returns the number of vertices in the graph.
Returns the vertices in the graph.
Returns the vertex object for the specified vertex index.
Returns the index for the specified vertex.
Returns the neighbors of vertex with the specified index.
Returns the degree for a specified vertex index.
Prints the edges.
Clears the graph.
Returns true if v is added to the graph. Returns false if v is already in the graph.
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.
Obtains a depth-first search tree starting from v.
Obtains a breadth-first search tree starting from v.
AbstractGraph<V>
#vertices: List<V>
#neighbors: List<List<Edge>>
#AbstractGraph()
#AbstractGraph(vertices: V[], edges: int[][])
#AbstractGraph(vertices: List<V>, edges: List<Edge>)
#AbstractGraph(edges: int[][], numberOfVertices: int)
#AbstractGraph(edges: List<Edge>, numberOfVertices: int)
+addEdge(e: Edge): boolean
Inner classes Tree is defined here
Vertices in the graph.
Neighbors for each vertex in the graph.
Constructs an empty graph.
Constructs a graph with the specified edges and vertices stored in arrays.
Constructs a graph with the specified edges and vertices stored in lists.
Constructs a graph with the specified edges in an array and the integer vertices 1, 2, ...
Constructs a graph with the specified edges in a list and the integer vertices 1, 2, ...
Adds an edge into the adjacency edge list.
UnweightedGraph<V>
+UnweightedGraph()
+UnweightedGraph(vertices: V[], edges: int[][])
+UnweightedGraph(vertices: List<V>, edges: List<Edge>)
+UnweightedGraph(edges: List<Edge>, numberOfVertices: int)
+UnweightedGraph(edges: int[][], numberOfVertices: int)
Constructs an empty unweighted graph.
Constructs a graph with the specified edges and vertices in arrays.
Constructs a graph with the specified edges and vertices stored in lists.
Constructs a graph with the specified edges in an array and the integer vertices 1, 2, ...
Constructs a graph with the specified edges in a list and the integer vertices 1, 2, ...

```
public interface Graph<V> {
  /** Return the number of vertices in the graph */
  public int getSize();
  /** Return the vertices in the graph */
  public java.util.List<V> getVertices();
  /** Return the object for the specified vertex index */
  public V getVertex(int index);
  /** Return the index for the specified vertex object */
  public int getIndex(V v);
  /** Return the neighbors of vertex with the specified index */
  public java.util.List<Integer> getNeighbors(int index);
  /** Return the degree for a specified vertex */
  public int getDegree(int v);
  /** Print the edges */
  public void printEdges();
  /** Clear graph */
  public void clear();
  /** Add a vertex to the graph */
  public boolean addVertex(V vertex);
  /** Add an edge to the graph */
  public boolean addEdge(int u, int v);
  /** Obtain a depth-first search tree */
  public AbstractGraph<V>.Tree dfs(int v);
  /** Obtain a breadth-first search tree */
  public AbstractGraph<V>.Tree bfs(int v);
}
```

```
import java.util.ArrayList;
import java.util.List;

public abstract class AbstractGraph<V> implements Graph<V> {
  // Store vertices
  protected List<V> vertices = new ArrayList();
  // Adjacency lists
  protected List<List<Edge>> neighbors = new ArrayList();

  /** Construct an empty graph */
  protected AbstractGraph() {
  }

  /** Construct a graph from vertices and edges stored in arrays */
  protected AbstractGraph(V[] vertices, int[][] edges) {
    for (int i = 0; i < vertices.length; i++)
      addVertex(vertices[i]);
    createAdjacencyLists(edges, vertices.length);
  }

  /** Construct a graph from vertices and edges stored in List */
  protected AbstractGraph(List<V> vertices, List<Edge> edges) {
    for (int i = 0; i < vertices.size(); i++)
      addVertex(vertices.get(i));
    createAdjacencyLists(edges, vertices.size());
  }
```

```
/** Edge inner class inside the AbstractGraph class */
public static class Edge {
  public int u; // Starting vertex of the edge
  public int v; // Ending vertex of the edge
  /** Construct an edge for (u, v) */
  public Edge(int u, int v) {
    this.u = u;
    this.v = v;
  }
  public boolean equals(Object o) {
    return u == ((Edge)o).u && v == ((Edge)o).v;
  }
}

/** Add an edge to the graph */
protected boolean addEdge(Edge e) {
  if (e.u < 0 || e.u > getSize() - 1)
    throw new IllegalArgumentException("No such index: " + e.u);
  if (e.v < 0 || e.v > getSize() - 1)
    throw new IllegalArgumentException("No such index: " + e.v);
  if (!neighbors.get(e.u).contains(e)) {
    neighbors.get(e.u).add(e);
    return true;
  } else {
    return false;
  }
}
```

```
/** Construct a graph for integer vertices 0, 1, 2 and edge list */
protected AbstractGraph(List<Edge> edges, int numberOfVertices) {
  for (int i = 0; i < numberOfVertices; i++)
    addVertex((V)(new Integer(i))); // vertices is {0, 1, ...}
  createAdjacencyLists(edges, numberOfVertices);
}

/** Construct a graph from integer vertices 0, 1, and edge array */
protected AbstractGraph(int[][] edges, int numberOfVertices) {
  for (int i = 0; i < numberOfVertices; i++)
    addVertex((V)(new Integer(i))); // vertices is {0, 1, ...}
  createAdjacencyLists(edges, numberOfVertices);
}

/** Create adjacency lists for each vertex */
private void createAdjacencyLists(int[][] edges,int numberOfVertices){
  for (int i = 0; i < edges.length; i++) {
    addEdge(edges[i][0], edges[i][1]);
  }
}

/** Create adjacency lists for each vertex */
private void createAdjacencyLists(List<Edge> edges,int numberOfVertices){
  for (Edge edge: edges) {
    addEdge(edge.u, edge.v);
  }
}
```

```
@Override /** Add an edge to the graph */
public boolean addEdge(int u, int v) {
  return addEdge(new Edge(u, v));
}

@Override /** Return the number of vertices in the graph */
public int getSize() {
  return vertices.size();
}

@Override /** Return the vertices in the graph */
public List<V> getVertices() {
  return vertices;
}

@Override /** Return the object for the specified vertex */
public V getVertex(int index) {
  return vertices.get(index);
}

@Override /** Return the index for the specified vertex object */
public int getIndex(V v) {
  return vertices.indexOf(v);
}
```

```
@Override /** Return the neighbors of the specified vertex */
public List<Integer> getNeighbors(int index) {
  List<Integer> result = new ArrayList();
  for (Edge e: neighbors.get(index))
    result.add(e.v);
  return result;
}

@Override /** Return the degree for a specified vertex */
public int getDegree(int v) {
  return neighbors.get(v).size();
}

@Override /** Clear the graph */
public void clear() {
  vertices.clear();
  neighbors.clear();
}
```

```
@Override /** Print the edges */
public void printEdges() {
  for (int u = 0; u < neighbors.size(); u++) {
    System.out.print(getVertex(u) + " (" + u + "): ");
    for (Edge e: neighbors.get(u)) {
      System.out.print("(" + getVertex(e.u) + ", " +
        getVertex(e.v) + ") ");
    }
    System.out.println();
  }
}

@Override /** Add a vertex to the graph */
public boolean addVertex(V vertex) {
  if (!vertices.contains(vertex)) {
    vertices.add(vertex);
    neighbors.add(new ArrayList<Edge>());
    return true;
  } else {
    return false;
  }
}
```

```
import java.util.*;
public class UnweightedGraph<V> extends AbstractGraph<V> {
  /** Construct an empty graph */
  public UnweightedGraph() {
  }

  /** Construct a graph from vertices and edges stored in arrays */
  public UnweightedGraph(V[] vertices, int[][] edges) {
    super(vertices, edges);
  }

  /** Construct a graph from vertices and edges stored in List */
  public UnweightedGraph(List<V> vertices, List<Edge> edges) {
    super(vertices, edges);
  }

  /** Construct a graph for integer vertices 0, 1, 2 and edge list */
  public UnweightedGraph(List<Edge> edges, int numberOfVertices) {
    super(edges, numberOfVertices);
  }

  /** Construct a graph from integer vertices 0, 1, and edge array */
  public UnweightedGraph(int[][] edges, int numberOfVertices) {
    super(edges, numberOfVertices);
  }
}
```

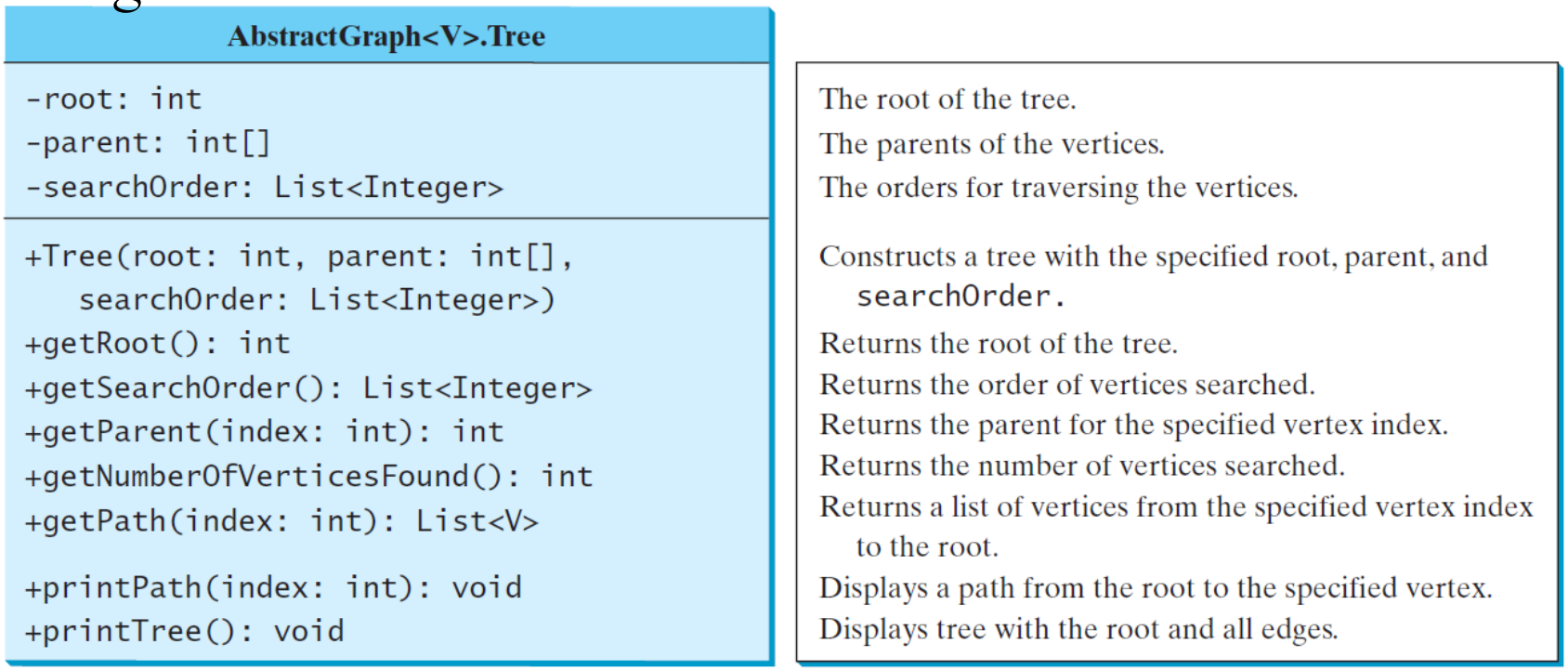
```
public class TestGraph {
  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};

    int[][] edges = {
      {0, 1}, {0, 3}, {0, 5},
      {1, 0}, {1, 2}, {1, 3},
      {2, 1}, {2, 3}, {2, 4}, {2, 10},
      {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
      {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
      {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
      {6, 5}, {6, 7},
      {7, 4}, {7, 5}, {7, 6}, {7, 8},
      {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
      {9, 8}, {9, 11},
      {10, 2}, {10, 4}, {10, 8}, {10, 11},
      {11, 8}, {11, 9}, {11, 10}
    };

    Graph<String> graph1 = new UnweightedGraph(vertices, edges);

    System.out.println("The number of vertices in graph1: "
      + graph1.getSize());
```

```
    System.out.println("The vertex with index 1 is "
      + graph1.getVertex(1));
    System.out.println("The index for Miami is " +
      graph1.getIndex("Miami"));

    System.out.println("The edges for graph1:");
    graph1.printEdges();

    String[] names = {"Peter", "Jane", "Mark", "Cindy", "Wendy"};
    java.util.ArrayList<AbstractGraph.Edge> edgeList
      = new java.util.ArrayList();
    edgeList.add(new AbstractGraph.Edge(0, 2));
    edgeList.add(new AbstractGraph.Edge(1, 2));
    edgeList.add(new AbstractGraph.Edge(2, 4));
    edgeList.add(new AbstractGraph.Edge(3, 4));

    // Create a graph with 5 vertices
    Graph<String> graph2 = new UnweightedGraph(
      java.util.Arrays.asList(names), edgeList);

    System.out.println("\nThe number of vertices in graph2: "
      + graph2.getSize());

    System.out.println("The edges for graph2:");
    graph2.printEdges();
  }
}
```

```
The number of vertices in graph1: 12
The vertex with index 1 is San Francisco
The index for Miami is 9
The edges for graph1:
Seattle (0): (0, 1) (0, 3) (0, 5)
San Francisco (1): (1, 0) (1, 2) (1, 3)
Los Angeles (2): (2, 1) (2, 3) (2, 4) (2, 10)
Denver (3): (3, 0) (3, 1) (3, 2) (3, 4) (3, 5)
Kansas City (4): (4, 2) (4, 3) (4, 5) (4, 7) (4, 8) (4, 10)
Chicago (5): (5, 0) (5, 3) (5, 4) (5, 6) (5, 7)
Boston (6): (6, 5) (6, 7)
New York (7): (7, 4) (7, 5) (7, 6) (7, 8)
Atlanta (8): (8, 4) (8, 7) (8, 9) (8, 10) (8, 11)
Miami (9): (9, 8) (9, 11)
Dallas (10): (10, 2) (10, 4) (10, 8) (10, 11)
Houston (11): (11, 8) (11, 9) (11, 10)

The number of vertices in graph2: 5
The edges for graph2:
Peter (0): (0, 2)
Jane (1): (1, 2)
Mark (2): (2, 4)
Cindy (3): (3, 4)
Wendy (4):
```

Graph Traversals

- ***Graph traversal*** is the process of visiting each vertex in the graph exactly once
- There are two popular ways to traverse a graph: ***depth-first traversal*** (or ***depth-first search***) and ***breadth-first traversal*** (or ***breadth-first search***)
- Both traversals result in a spanning tree, which can be modeled using a class:



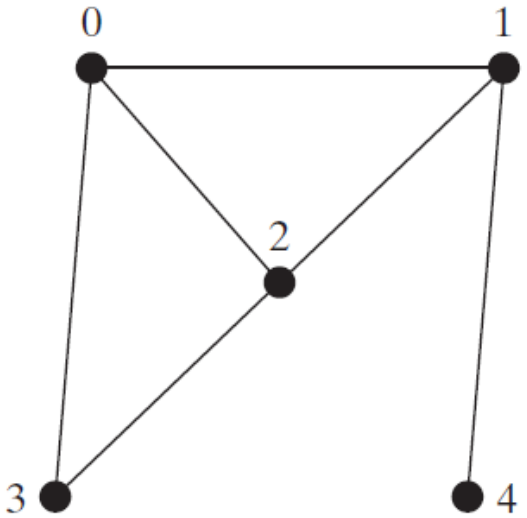
Depth-First Search

- The ***depth-first search*** of a graph starts from a vertex in the graph and visits all vertices in the graph as far as possible before backtracking

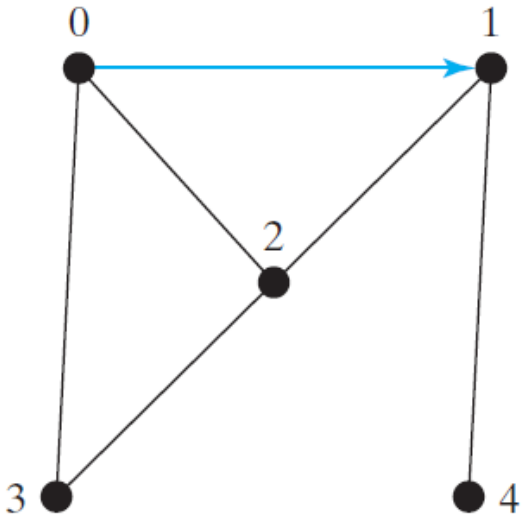
```
Input: G = (V, E) and a starting vertex v
Output: a DFS tree rooted at v
Tree dfs(vertex v) {
  visit v;
  for each neighbor w of v
    if (w has not been visited) {
      set v as the parent for w;
      dfs(w);
    }
}
```

- Since each edge and each vertex is visited only once, the time complexity of the dfs method is **O(|E| + |V|)**, where **|E|** denotes the number of edges and **|V|** the number of vertices

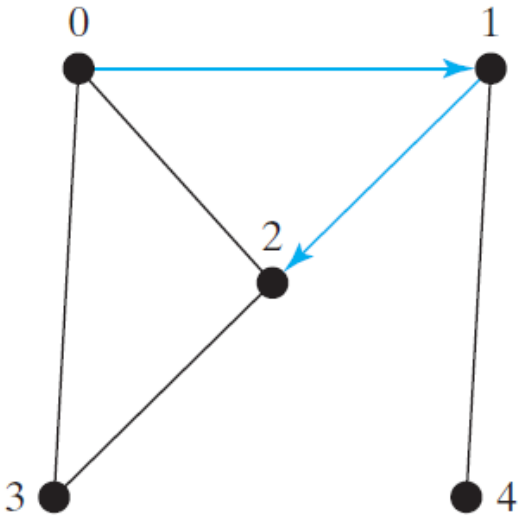
Depth-First Search Example



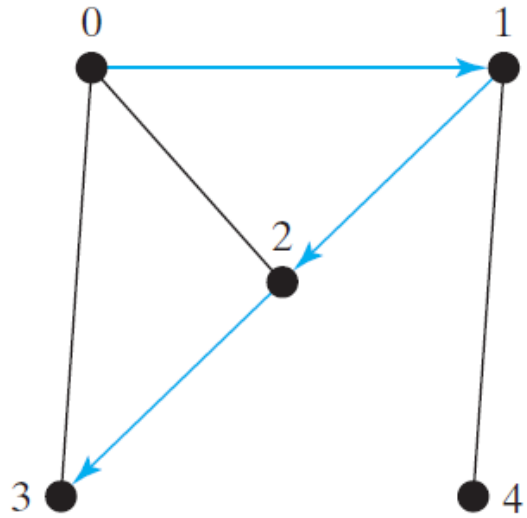
(a)



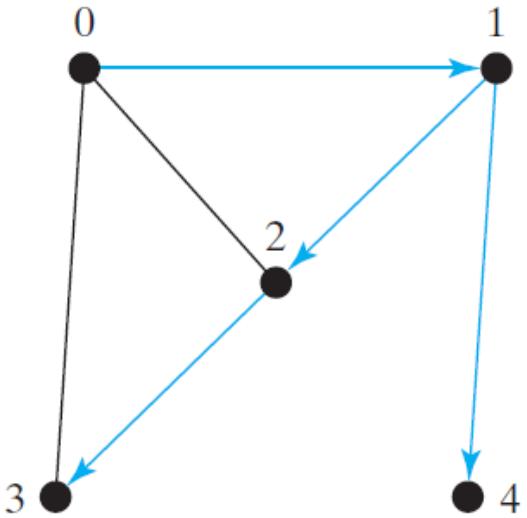
(b)



(c)

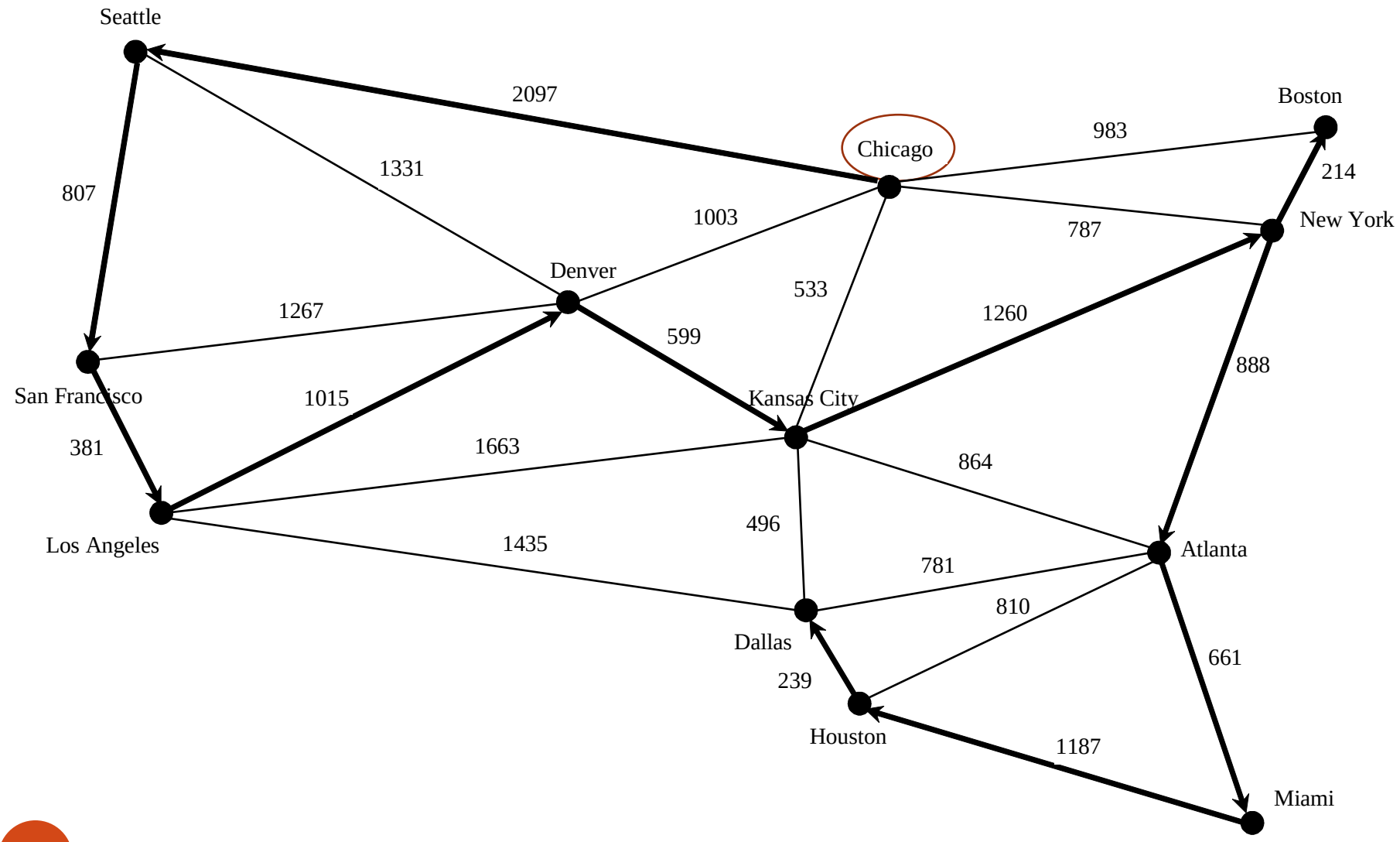


(d)



(e)

Depth-First Search Example



```
// Add the inner class Tree in the AbstractGraph class
public class Tree {
  private int root; // The root of the tree
  private int[] parent; // Store the parent of each vertex
  private List<Integer> searchOrder; // Store the search order

  /** Construct a tree with root, parent, and searchOrder */
  public Tree(int root, int[] parent, List<Integer> searchOrder){
    this.root = root;
    this.parent = parent;
    this.searchOrder = searchOrder;
  }

  /** Return the root of the tree */
  public int getRoot() {
    return root;
  }

  /** Return the parent of vertex v */
  public int getParent(int v) {
    return parent[v];
  }
```

```
/** Return an array representing search order */
public List<Integer> getSearchOrder() {
  return searchOrder;
}
/** Return number of vertices found */
public int getNumberOfVerticesFound() {
  return searchOrder.size();
}

/** Return the path of vertices from a vertex to the root */
public List<V> getPath(int index) {
  ArrayList<V> path = new ArrayList();
  do {
    path.add(vertices.get(index));
    index = parent[index];
  } while (index != -1);
  return path;
}

/** Print a path from the root to vertex v */
public void printPath(int index) {
  List<V> path = getPath(index);
  System.out.print("A path from " + vertices.get(root) + " to " +
    vertices.get(index) + ": ");
  for (int i = path.size() - 1; i >= 0; i--)
    System.out.print(path.get(i) + " ");
}
```

```
  /** Print the whole tree */
  public void printTree() {
    System.out.println("Root is: " + vertices.get(root));
    System.out.print("Edges: ");
    for (int i = 0; i < parent.length; i++)
      if (parent[i] != -1) {
        // Display an edge
        System.out.print("(" + vertices.get(parent[i]) + ", " +
          vertices.get(i) + ") ");
      }
    System.out.println();
  }
}
```

```
@Override /** Obtain a DFS tree starting from vertex v */
public Tree dfs(int v) {
  List<Integer> searchOrder = new ArrayList();
  int[] parent = new int[vertices.size()];
  for (int i = 0; i < parent.length; i++)
    parent[i] = -1; // Initialize parent[i] to -1
  // Mark visited vertices (default false)
  boolean[] isVisited = new boolean[vertices.size()];
  // Recursively search
  dfs(v, parent, searchOrder, isVisited);
  // Return a search tree
  return new Tree(v, parent, searchOrder);
}

/** Recursive method for DFS search */
private void dfs(int u, int[] parent, List<Integer> searchOrder,
    boolean[] isVisited) {
  // Store the visited vertex
  searchOrder.add(u);
  isVisited[u] = true; // Vertex v visited
  for (Edge e : neighbors.get(u))
    if (!isVisited[e.v]) {
      parent[e.v] = u; // The parent of vertex e.v is u
      dfs(e.v, parent, searchOrder, isVisited); // Recursive search
    }
}
```

```
public class TestDFS {

  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};

    int[][] edges = {
      {0, 1}, {0, 3}, {0, 5},
      {1, 0}, {1, 2}, {1, 3},
      {2, 1}, {2, 3}, {2, 4}, {2, 10},
      {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
      {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
      {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
      {6, 5}, {6, 7},
      {7, 4}, {7, 5}, {7, 6}, {7, 8},
      {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
      {9, 8}, {9, 11},
      {10, 2}, {10, 4}, {10, 8}, {10, 11},
      {11, 8}, {11, 9}, {11, 10}
    };

    Graph<String> graph = new UnweightedGraph(vertices, edges);

    AbstractGraph<String>.Tree dfs = graph.dfs(graph.getIndex("Chicago"));
```

```
    java.util.List<Integer> searchOrders = dfs.getSearchOrder();

    System.out.println(dfs.getNumberOfVerticesFound() +
      " vertices are searched in this DFS order:");

    for (int i = 0; i < searchOrders.size(); i++)
      System.out.print(graph.getVertex(searchOrders.get(i)) + " ");
    System.out.println();

    for (int i = 0; i < searchOrders.size(); i++)
      if (dfs.getParent(i) != -1)
        System.out.println("parent of " + graph.getVertex(i) +
          " is " + graph.getVertex(dfs.getParent(i)));
  }
}
```

```
12 vertices are searched in this DFS order:
   Chicago Seattle San Francisco Los Angeles Denver
   Kansas City New York Boston Atlanta Miami Houston Dallas
parent of Seattle is Chicago
parent of San Francisco is Seattle
parent of Los Angeles is San Francisco
parent of Denver is Los Angeles
parent of Kansas City is Denver
parent of Boston is New York
parent of New York is Kansas City
parent of Atlanta is New York
parent of Miami is Atlanta
parent of Dallas is Houston
parent of Houston is Miami
```

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 AND find it (not the shortest)
- Finding all connected components:
  - A ***connected component*** is a maximal connected subgraph in which every pair of vertices are connected by a path

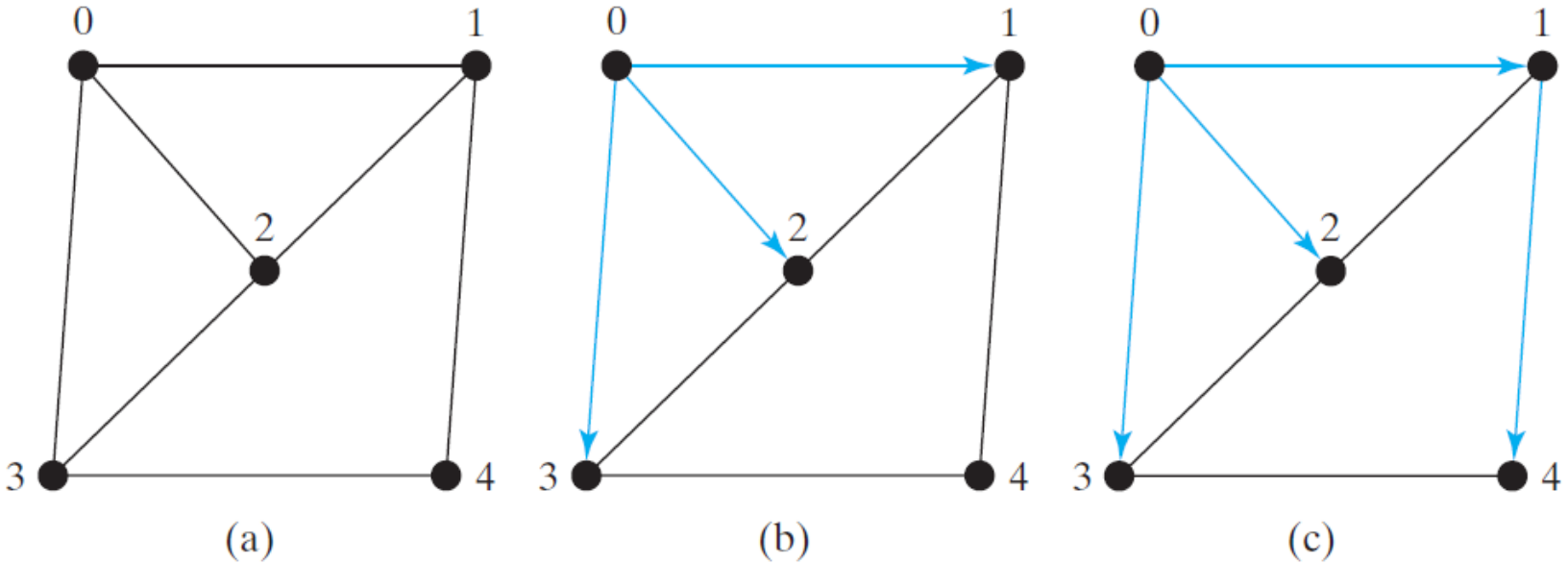
Breadth-First Search

- The ***breadth-first search*** of a graph visits the vertices level by level
  - The first level consists of the starting vertex (root)
  - Each next level consists of the vertices adjacent to the vertices in the preceding 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
- To ensure that each vertex is visited only once, it skips a vertex if it has already been visited

Breadth-First Search Algorithm

```
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

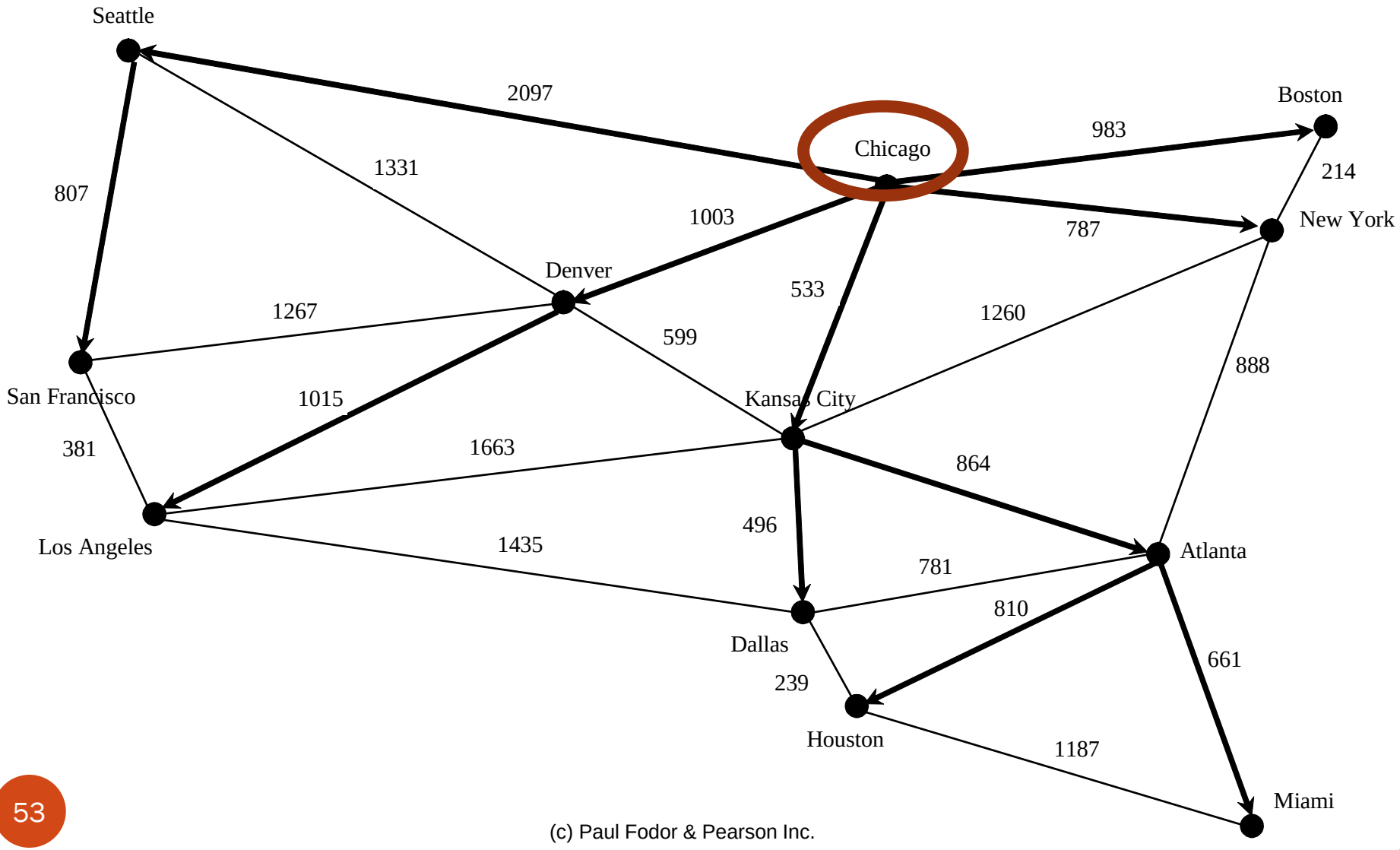
Queue: 1 2 3

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

Queue: 2 3 4

isVisited[4] = true

Breadth-First Search Example



```
@Override /** Starting bfs search from vertex v */
public Tree bfs(int v) {
  List<Integer> searchOrder = new ArrayList();
  int[] parent = new int[vertices.size()];
  for (int i = 0; i < parent.length; i++)
    parent[i] = -1; // Initialize parent[i] to -1
  java.util.LinkedList<Integer> queue =
    new java.util.LinkedList(); // list used as a queue
  queue.offer(v); // Enqueue v
  boolean[] isVisited = new boolean[vertices.size()];
  isVisited[v] = true; // Mark it visited

  while (!queue.isEmpty()) {
    int u = queue.poll(); // Dequeue to u
    searchOrder.add(u); // u searched
    for (Edge e: neighbors.get(u))
      if (!isVisited[e.v]) {
        queue.offer(e.v); // Enqueue v
        parent[e.v] = u; // The parent of w is u
        isVisited[e.v] = true; // Mark it visited
      }
  }
  return new Tree(v, parent, searchOrder);
}
```

```
public class TestBFS {

  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};

    int[][] edges = {
      {0, 1}, {0, 3}, {0, 5},
      {1, 0}, {1, 2}, {1, 3},
      {2, 1}, {2, 3}, {2, 4}, {2, 10},
      {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
      {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
      {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
      {6, 5}, {6, 7},
      {7, 4}, {7, 5}, {7, 6}, {7, 8},
      {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
      {9, 8}, {9, 11},
      {10, 2}, {10, 4}, {10, 8}, {10, 11},
      {11, 8}, {11, 9}, {11, 10}
    };

    Graph<String> graph = new UnweightedGraph(vertices, edges);

    AbstractGraph<String>.Tree bfs = graph.bfs(graph.getIndex("Chicago"));
```

```
    java.util.List<Integer> searchOrders = bfs.getSearchOrder();
    System.out.println(bfs.getNumberOfVerticesFound() +
      " vertices are searched in this order:");
    for (int i = 0; i < searchOrders.size(); i++)
      System.out.println(graph.getVertex(searchOrders.get(i)));
    for (int i = 0; i < searchOrders.size(); i++)
      if (bfs.getParent(i) != -1)
        System.out.println("parent of " + graph.getVertex(i) +
          " is " + graph.getVertex(bfs.getParent(i)));
  }
}
```

```
12 vertices are searched in this order:
 Chicago Seattle Denver Kansas City Boston New York
 San Francisco Los Angeles Atlanta Dallas Miami Houston
parent of Seattle is Chicago
parent of San Francisco is Seattle
parent of Los Angeles is Denver
parent of Denver is Chicago
parent of Kansas City is Chicago
parent of Boston is Chicago
parent of New York is Chicago
parent of Atlanta is Kansas City
parent of Miami is Atlanta
parent of Dallas is Kansas City
parent of Houston is Atlanta
```

Applications of the BFS

- Detecting whether a graph is connected (i.e., 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 - we can prove that the path between the root and any node in the BFS tree is the shortest path between the root and that 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 by modifying BFS (if a node was seen before, then there is a cycle - you can also extract the cycle)

Applications of the BFS

- 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
    - A graph is bipartite graph if and only if it is 2-colorable.
  - While doing **BFS** traversal, each node in the **BFS** tree is given the opposite color to its parent.
    - If there exists an edge connecting current vertex to a previously-colored vertex with the same color, then we can safely conclude that the **graph** is **NOT bipartite**.
  - If the graph is bipartite, then one partition is the union of all odd number stratas, another is the union of even number stratas

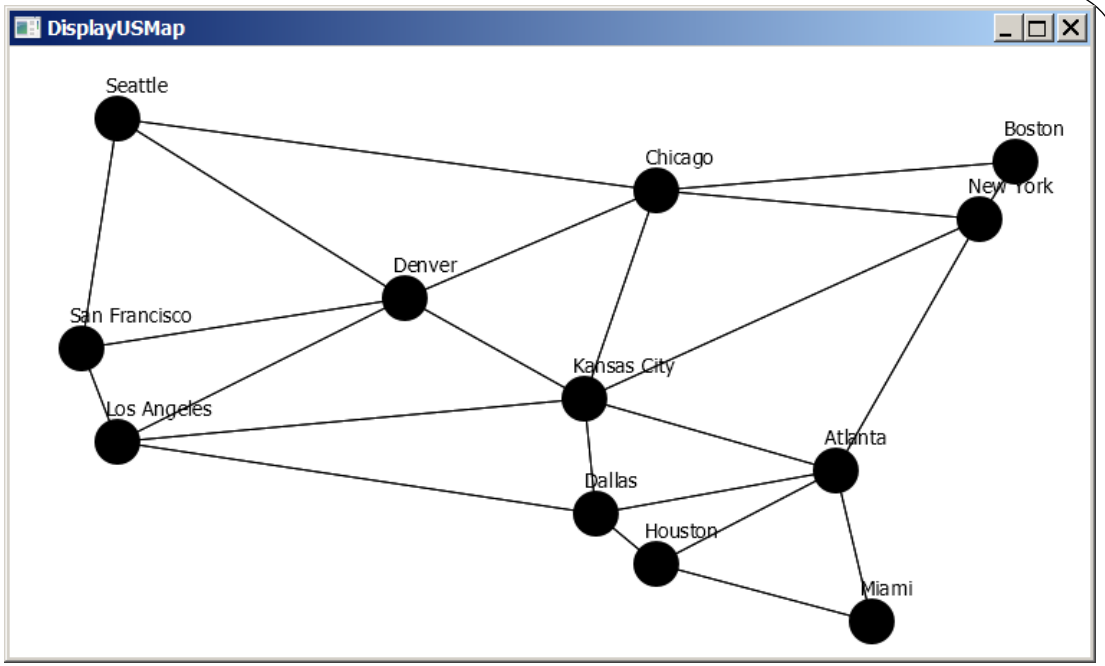
Graph Visualization

```
public interface Displayable {
  public int getX(); // Get x-coordinate of the vertex
  public int getY(); // Get x-coordinate of the vertex
  public String getName(); // Get display name of the vertex
}

import javafx.scene.layout.Pane;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Line;
import javafx.scene.text.Text;
public class GraphView extends Pane {
  private Graph<? extends Displayable> graph;
  public GraphView(Graph<? extends Displayable> graph) {
    this.graph = graph;
    // Draw vertices
    java.util.List<? extends Displayable> vertices=graph.getVertices();
    for (int i = 0; i < graph.getSize(); i++) {
      int x = vertices.get(i).getX();
      int y = vertices.get(i).getY();
      String name = vertices.get(i).getName();
      getChildren().add(new Circle(x, y, 16)); // Display a vertex
      getChildren().add(new Text(x - 8, y - 18, name));
    }
```

```
    // Draw edges for pair of vertices
    for (int i = 0; i < graph.getSize(); i++) {
      java.util.List<Integer> neighbors = graph.getNeighbors(i);
      int x1 = graph.getVertex(i).getX();
      int y1 = graph.getVertex(i).getY();
      for (int v: neighbors) {
        int x2 = graph.getVertex(v).getX();
        int y2 = graph.getVertex(v).getY();
        // Draw an edge for (i, v)
        getChildren().add(new Line(x1, y1, x2, y2));
      }
    }
  }
}
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;
public class DisplayUSMap extends Application {
  @Override
  public void start(Stage primaryStage) {
    City[] vertices = {new City("Seattle", 75, 50),
      new City("San Francisco", 50, 210),
      new City("Los Angeles", 75, 275), new City("Denver", 275, 175),
      new City("Kansas City", 400, 245),
      new City("Chicago", 450, 100), new City("Boston", 700, 80),
      new City("New York", 675, 120), new City("Atlanta", 575, 295),
```

```
      new City("Miami", 600, 400), new City("Dallas", 408, 325),
      new City("Houston", 450, 360) };
    int[][] edges = {
      {0, 1}, {0, 3}, {0, 5}, {1, 0}, {1, 2}, {1, 3},
      {2, 1}, {2, 3}, {2, 4}, {2, 10},
      {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
      {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
      {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
      {6, 5}, {6, 7}, {7, 4}, {7, 5}, {7, 6}, {7, 8},
      {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
      {9, 8}, {9, 11}, {10, 2}, {10, 4}, {10, 8}, {10, 11},
      {11, 8}, {11, 9}, {11, 10}
    };
    Graph<City> graph = new UnweightedGraph(vertices, edges);
    // Create a scene and place it in the stage
    Scene scene = new Scene(new GraphView(graph), 750, 450);
    primaryStage.setTitle("DisplayUSMap");
    primaryStage.setScene(scene);
    primaryStage.show();
  }
  static class City implements Displayable {
    private int x, y;
    private String name;
    City(String name, int x, int y) {
      this.name = name;
      this.x = x;
      this.y = y;
```

```
    @Override
    public int getX() {
      return x;
    }

    @Override
    public int getY() {
      return y;
    }

    @Override
    public String getName() {
      return name;
    }
  }

  public static void main(String[] args) {
    launch(args);
  }
}
```



The Nine Tail Problem

- Nine coins are placed in a three by three matrix with some face up (H) and some face down (T)
- A legal move is to take any coin that is face up (H) and reverse it, together with the coins adjacent to it (this does not include coins that are diagonally adjacent)
- **Your task is to find the minimum number of the moves that lead to all coins face down (T)**

H	H	H
T	T	T
H	H	H

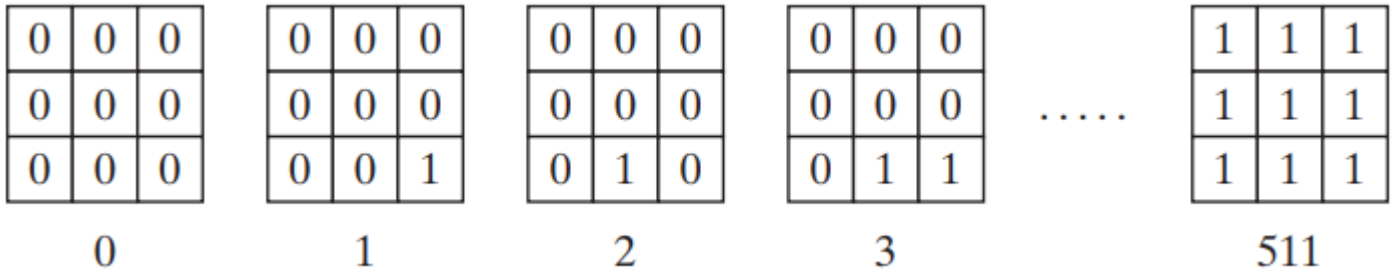
H	H	H
T	H	T
T	T	T

T	T	T
T	T	T
T	T	T

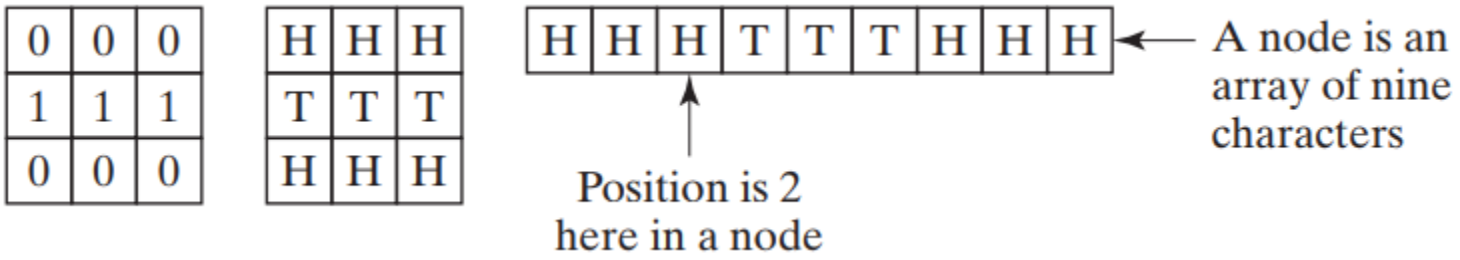
- The nine tails problem can be reduced to the shortest path problem

Representing Nine Coins

- Each state of the nine coins represents a node in the graph
- Since there are nine cells and each cell is either 0 or 1, there are a total of 2^9 (512) nodes, labeled 0, 1, . . . , and 511



- Represent the node as an array of char:

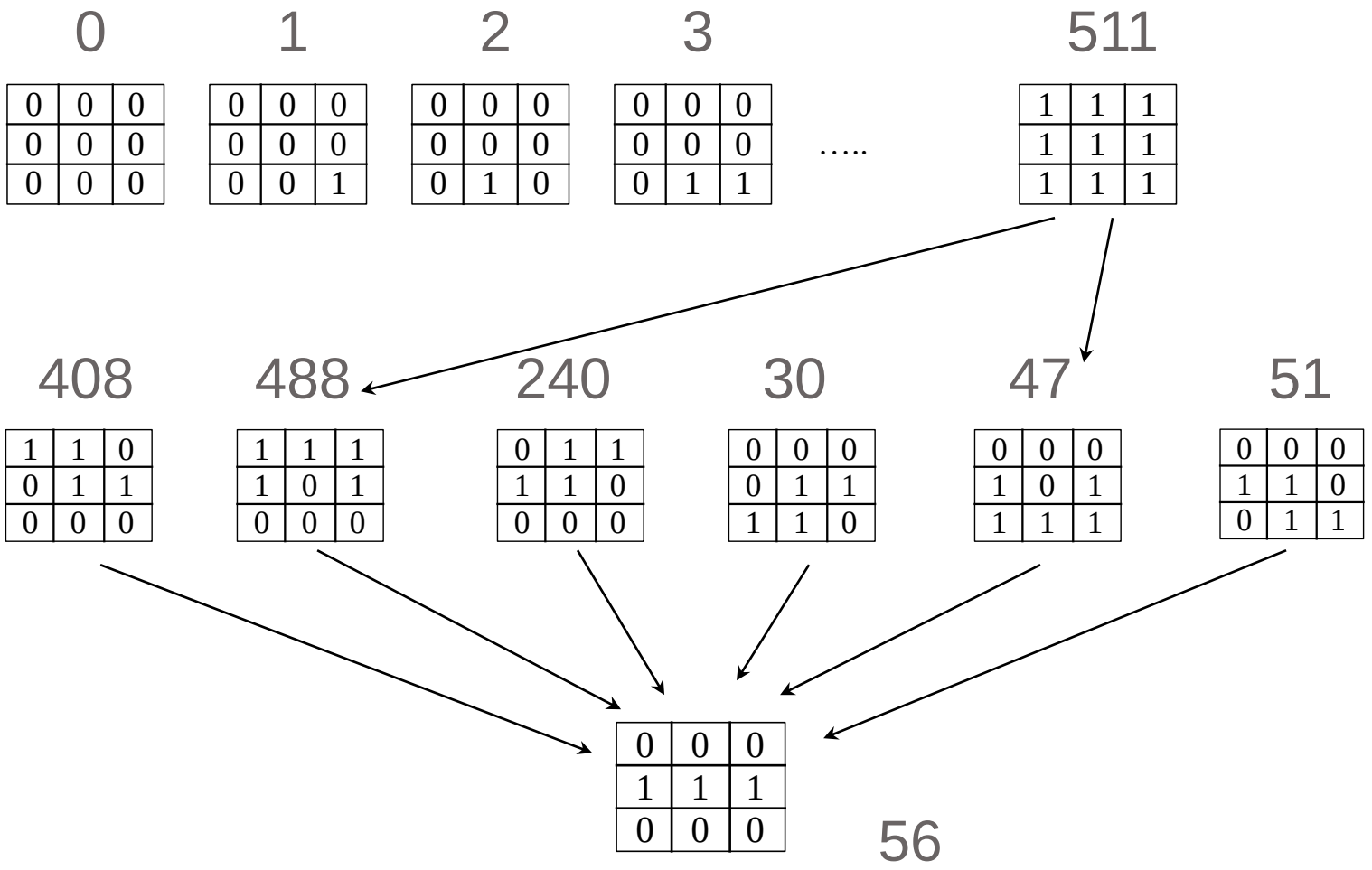


and convert that from binary to decimal

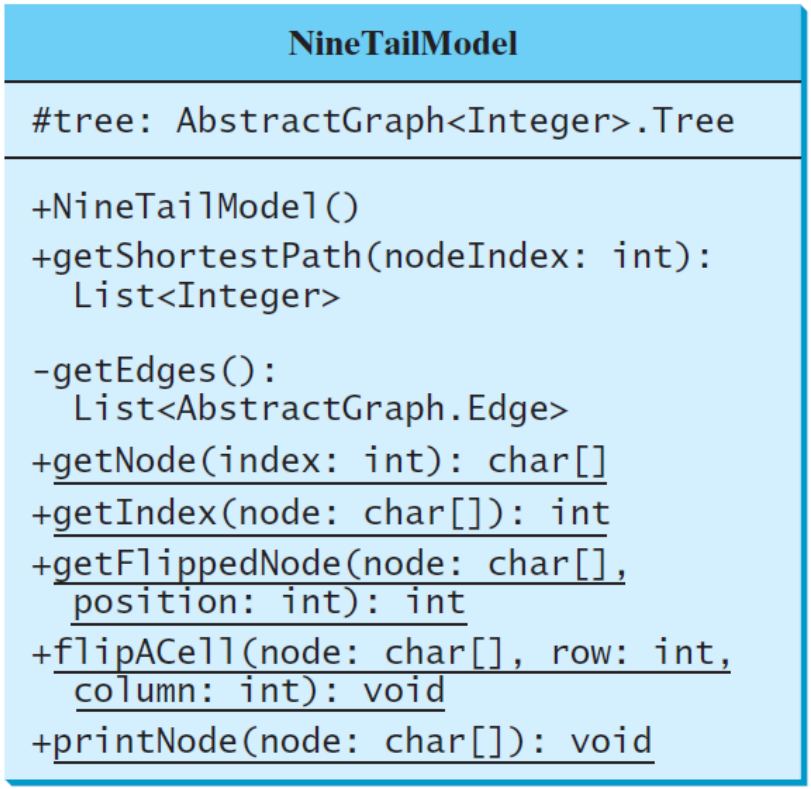
- 511 represents the state of nine face-down coins (we call this last node the ***target node***)

Model the Nine Tail Problem

- We assign an edge from node **v** to **u** if there is a legal move from **u** to **v**
  - The task is to build a graph that consists of 512 nodes labeled 0, 1, 2,…, 511, and edges among the nodes – We start from 511:



NineTailModel



A tree rooted at node 511.

Constructs a model for the nine tails problem and obtains the tree.

Returns a path from the specified node to the root. The path returned consists of the node labels in a list.

Returns a list of Edge objects for the graph.

Returns a node consisting of nine characters of Hs and Ts.

Returns the index of the specified node.

Flips the node at the specified position and its adjacent positions and returns the index of the flipped node.

Flips the node at the specified row and column.

Displays the node on the console.

```
import java.util.*;
public class NineTailModel {
  public final static int NUMBER_OF_NODES = 512;
  protected AbstractGraph<Integer>.Tree tree; // Define a tree
  /** Construct a model */
  public NineTailModel() {
    // Create edges
    List<AbstractGraph.Edge> edges = getEdges();
    // Create a graph
    UnweightedGraph<Integer> graph = new UnweightedGraph(
      edges, NUMBER_OF_NODES);
    // Obtain a BSF tree rooted at the target node
    tree = graph.bfs(511);
  }
  /** Create all edges for the graph */
  private List<AbstractGraph.Edge> getEdges() {
    List<AbstractGraph.Edge> edges =
      new ArrayList(); // Store edges
    for (int u = 0; u < NUMBER_OF_NODES; u++)
      for (int k = 0; k < 9; k++) {
        char[] node = getNode(u); // Get the node for vertex u
        if (node[k] == 'H') {
          int v = getFlippedNode(node, k);
          // Add edge (v, u) for a legal move from node u to node v
          edges.add(new AbstractGraph.Edge(v, u));
        }
      }
    return edges;
  }
```

```
public static int getFlippedNode(char[] node, int position) {
  int row = position / 3;
  int column = position % 3;
  flipACell(node, row, column);
  flipACell(node, row - 1, column);
  flipACell(node, row + 1, column);
  flipACell(node, row, column - 1);
  flipACell(node, row, column + 1);
  return getIndex(node);
}

public static void flipACell(char[] node, int row, int column){
  if (row >= 0 && row <= 2 && column >= 0 && column <= 2) {
    // Within the boundary
    if (node[row * 3 + column] == 'H')
      node[row * 3 + column] = 'T'; // Flip from H to T
    else
      node[row * 3 + column] = 'H'; // Flip from T to H
  }
}

public List<Integer> getShortestPath(int nodeIndex) {
  return tree.getPath(nodeIndex);
}
```

```
public static int getIndex(char[] node) {
  int result = 0;
  for (int i = 0; i < 9; i++)
    if (node[i] == 'T')
      result = result * 2 + 1;
    else
      result = result * 2 + 0;
  return result;
}

public static char[] getNode(int index) {
  char[] result = new char[9];
  for (int i = 0; i < 9; i++) {
    int digit = index % 2;
    if (digit == 0)
      result[8 - i] = 'H';
    else
      result[8 - i] = 'T';
    index = index / 2;
  }
  return result;
}

public static void printNode(char[] node) {
  for (int i = 0; i < 9; i++)
    if (i % 3 != 2)
      System.out.print(node[i]);
    else
      System.out.println(node[i]);
  System.out.println();
}
```

```
import java.util.Scanner;

public class NineTail {
  public static void main(String[] args) {
    // Prompt the user to enter nine coins H and T's
    System.out.print("Enter the initial nine coins Hs and Ts: ");
    Scanner input = new Scanner(System.in);
    String s = input.nextLine();

    char[] initialNode = s.toCharArray();

    NineTailModel model = new NineTailModel();
    java.util.List<Integer> path =
      model.getShortestPath(NineTailModel.getIndex(initialNode));

    System.out.println("The steps to flip the coins are ");
    for (int i = 0; i < path.size(); i++)
      NineTailModel.printNode(NineTailModel.getNode(path.get(i)));
  }

}
```

```
Enter the initial nine coins Hs and Ts: H H H T T T H T H
The steps to flip the coins are
HHH
HHH
THT

THH
TTH
HHT

THH
HTH
TTT

THT
HHT
TTH

THT
HHH
THT

TTT
TTT
TTT
```