Graphs and Applications

Paul Fodor

CSE260, Computer Science B: Honors

Stony Brook University

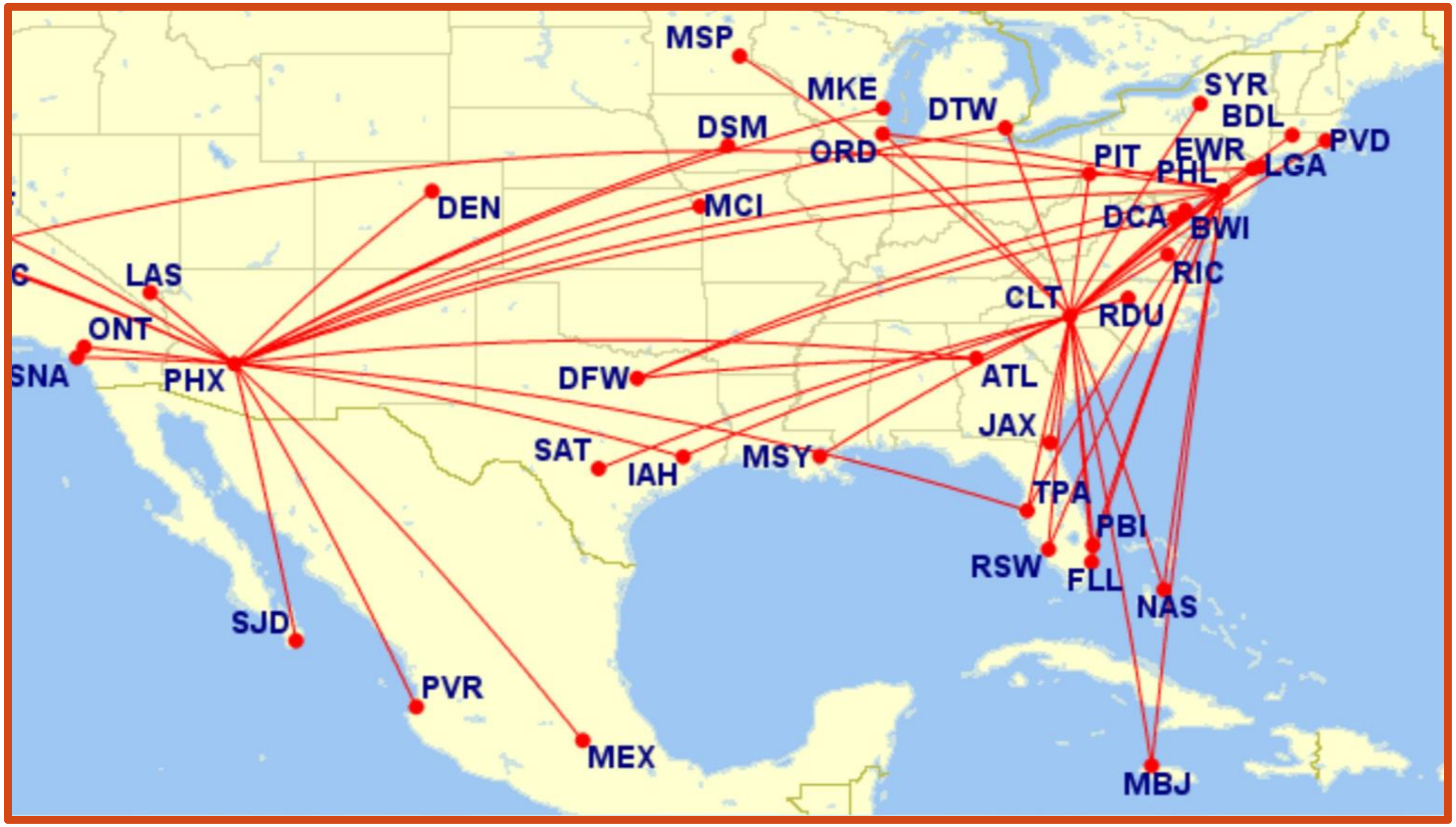
http://www.cs.stonybrook.edu/~cse260

Objectives

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

Modeling real-world problems 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



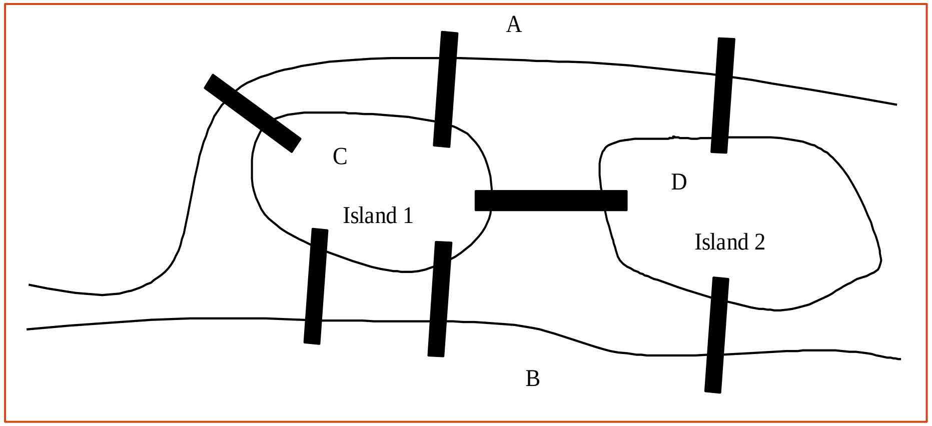
Modeling Problems Using Graphs

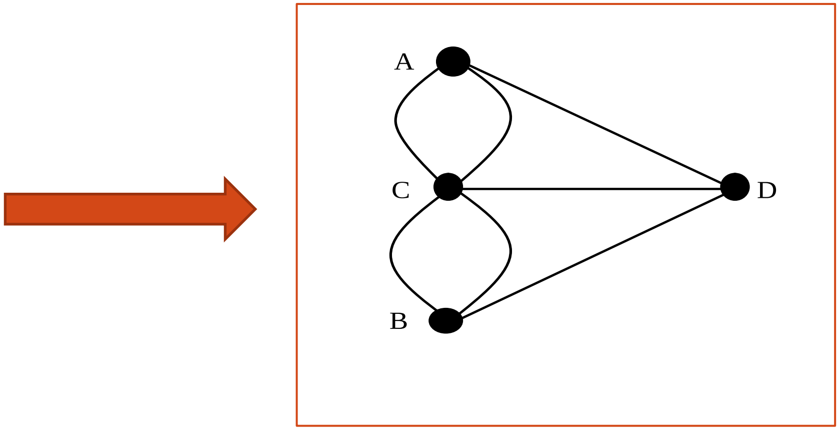
- Many practical problems can be represented by graphs because graphs are used to represent:
  - travel routes (airline scheduling), optimal mail/package delivery, supply chain implementation
  - networks of communication
    - ***routing*** is the selection of paths for traffic in a network
  - social media analysis: marketing (community detection), centrality measurement, information flow, maximizing influence, etc.
  - computer chip design (placement of electronic components into an electrical network on a monolithic semiconductor)
  - Search Engine Algorithms (e.g., PageRank algorithm)
- The development of algorithms to handle graphs is therefore of major interest in computer science.

How it all started?

Leonhard Euler

- 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***:

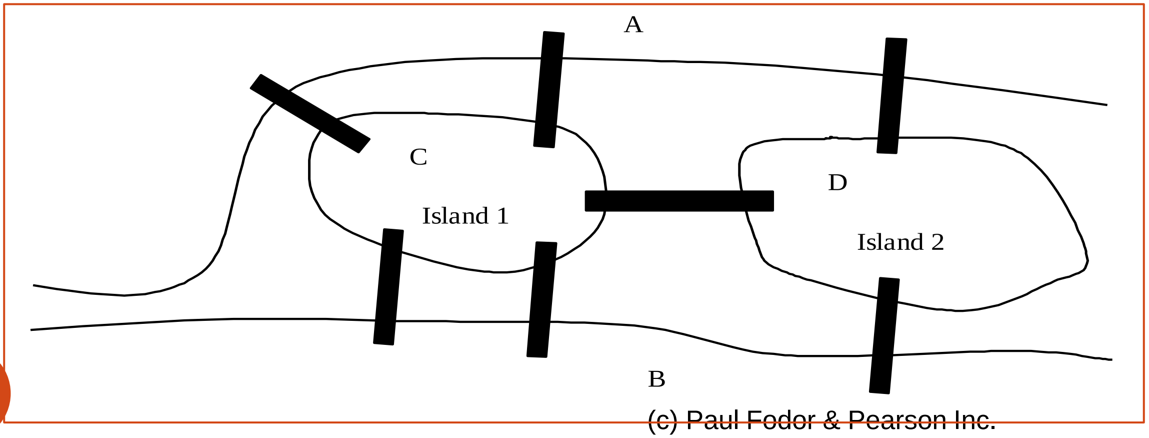


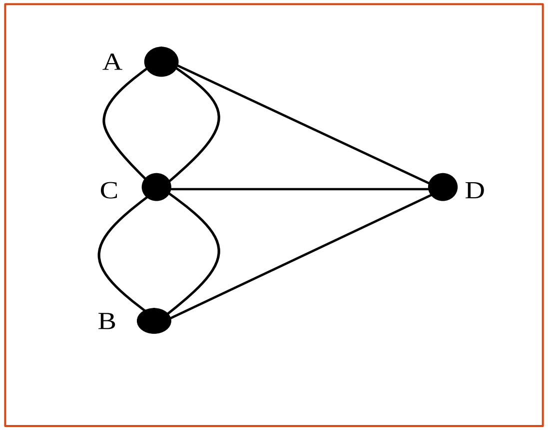


Seven Bridges of Königsberg

- Euler's question: ***can one take a walk, cross each bridge exactly once, and return to the starting point?***
  - That is: ***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!

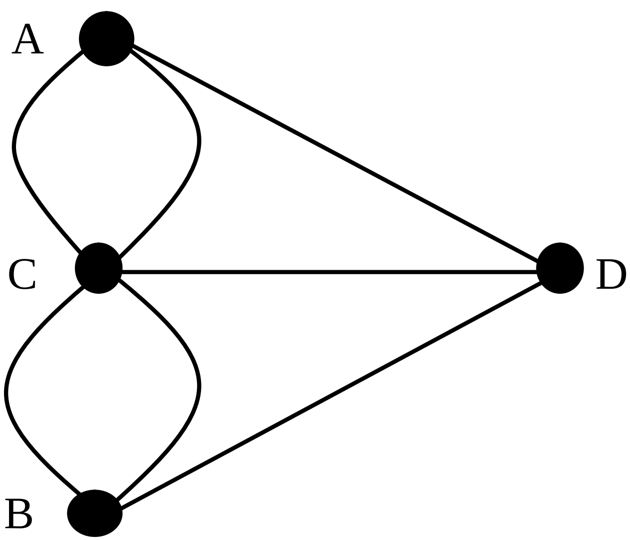
Swiss mathematician Leonhard Euler
(15 April 1707 – 18 September 1783)

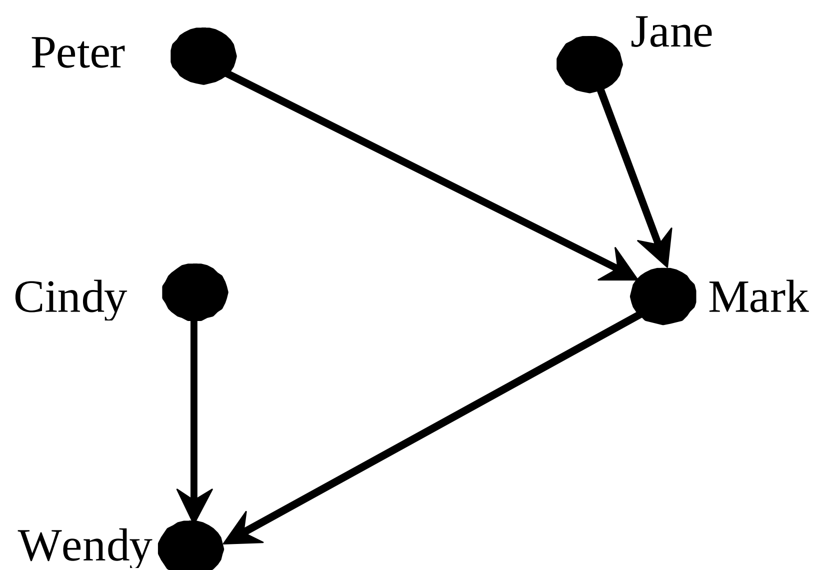




Basic Graph Terminology

- A ***graph*** **`G = (V, E)`**, where **`V`** represents a set of vertices (or nodes) and **`E`** represents a set of edges (or links).
- A graph may be ***undirected*** (i.e., if (x,y) is in E, then (y,x) is also in E) or ***directed***



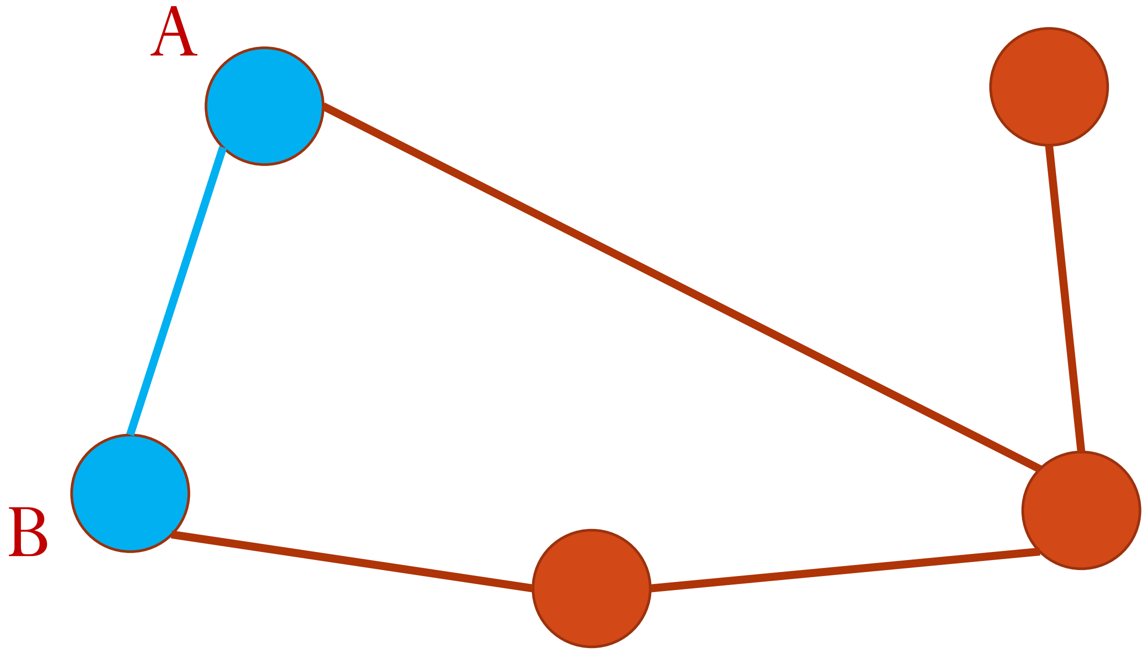


Peter likes Mark

Mark does not like Peter

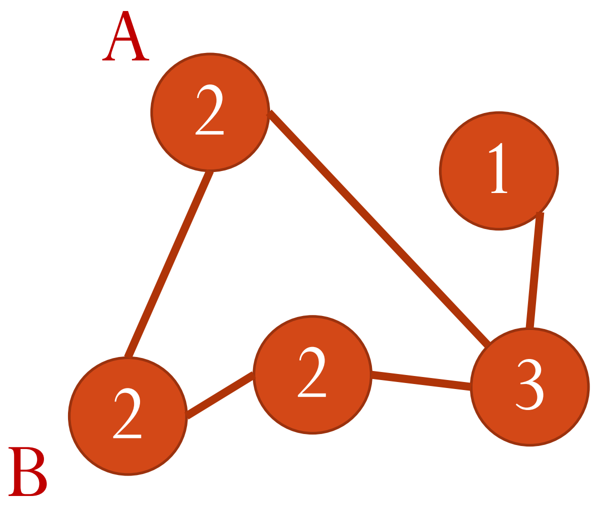
Adjacent Vertices

- 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
  - For example, A and B are adjacent



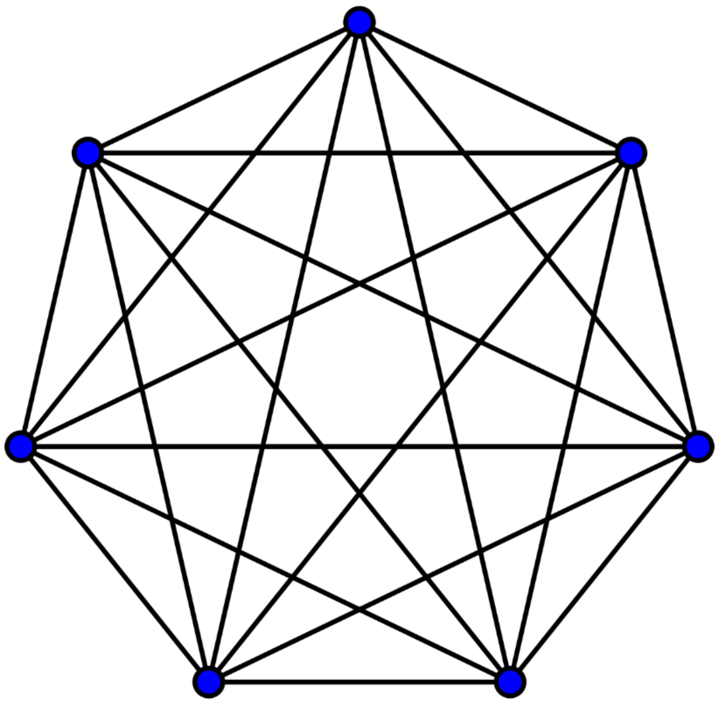
Degree

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

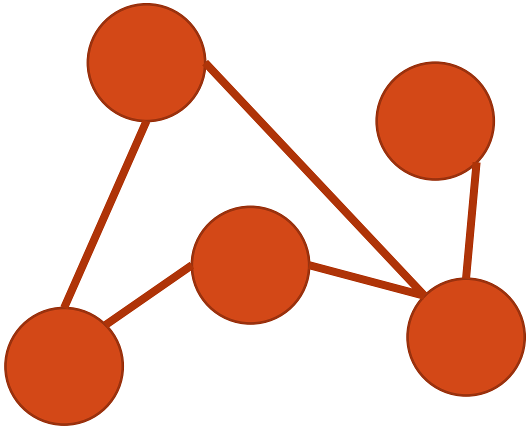


Complete graph

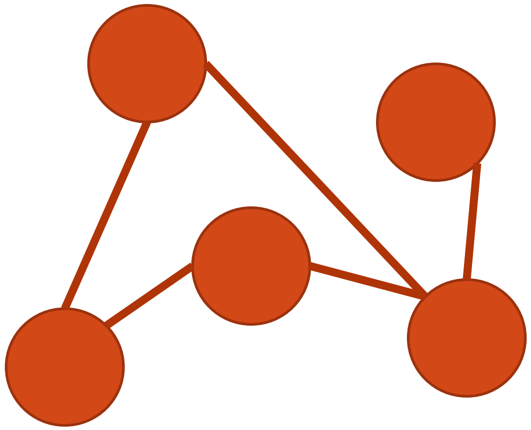
every two pairs of vertices are connected



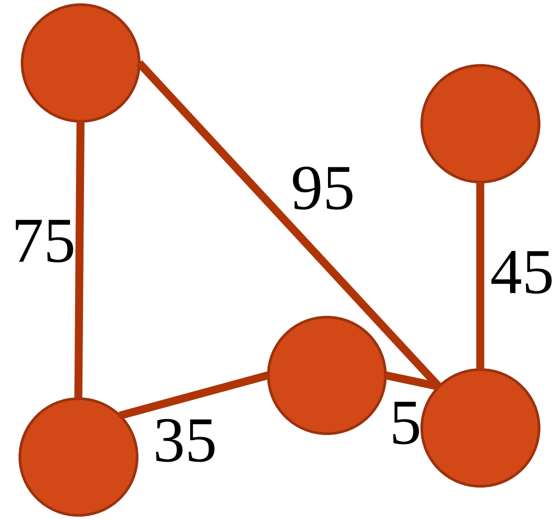
Incomplete graph



Unweighted

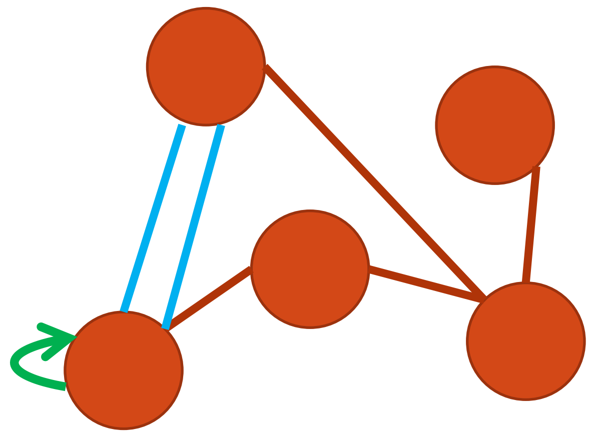


Weighted



Parallel Edges

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

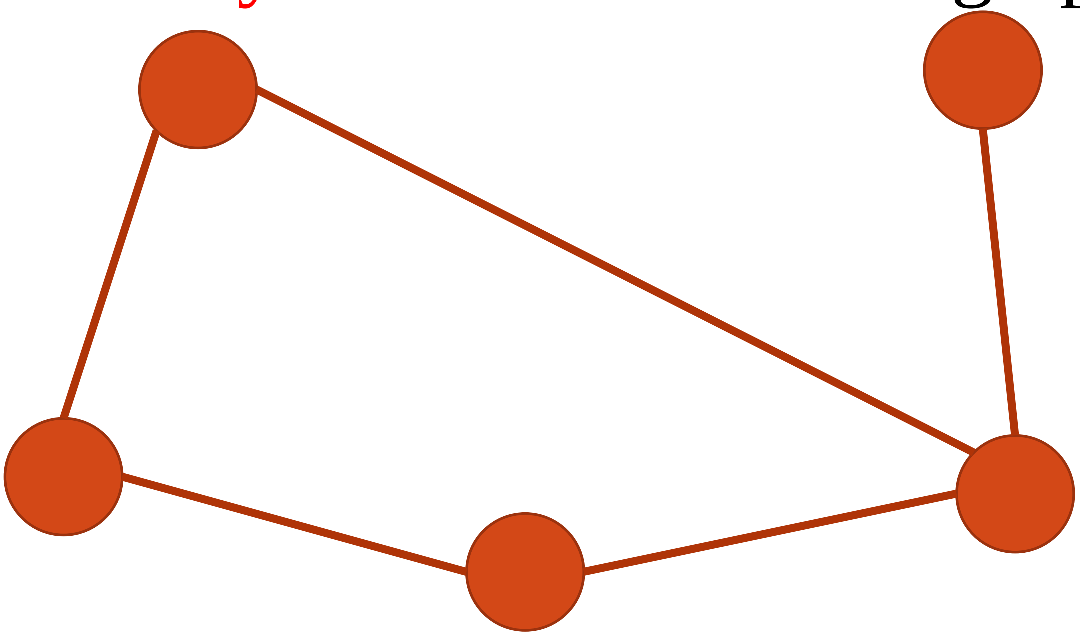


A ***loop*** is an edge that links a vertex to itself

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

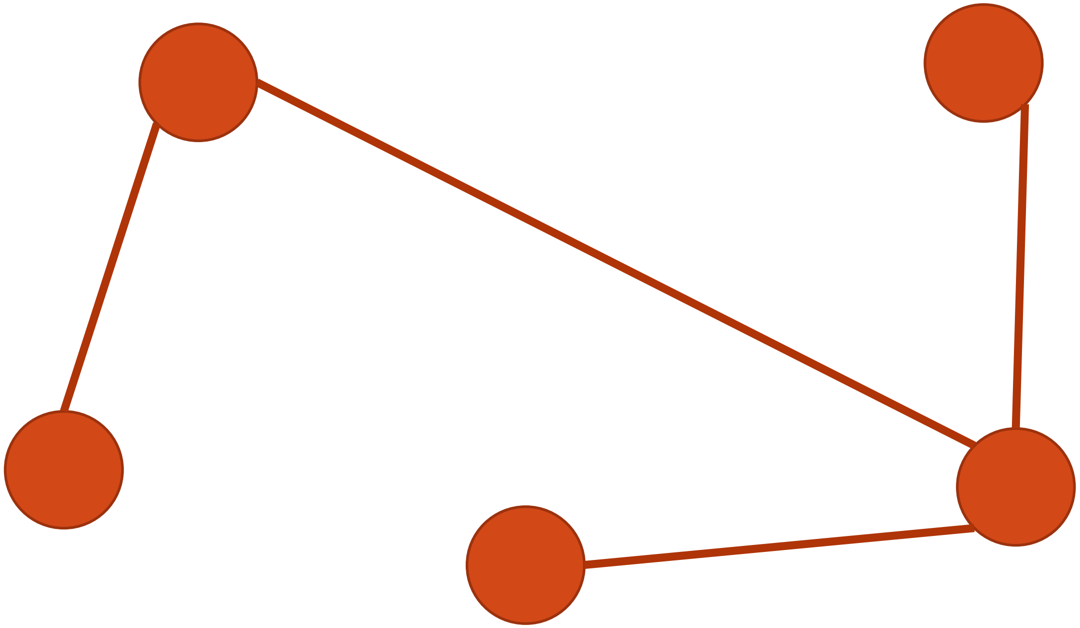
Connected graph

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



Tree

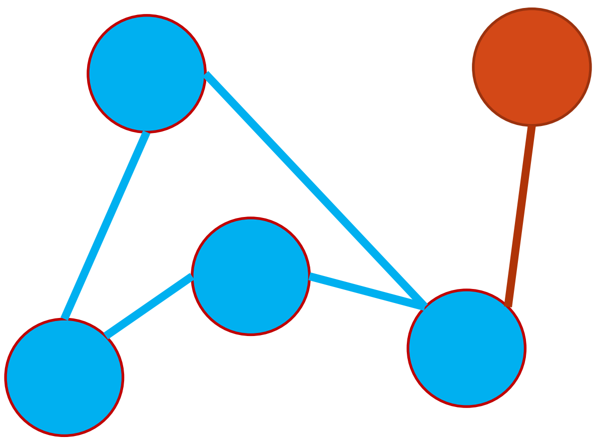
- A connected graph is a ***tree*** if it does not have cycles



Cycles

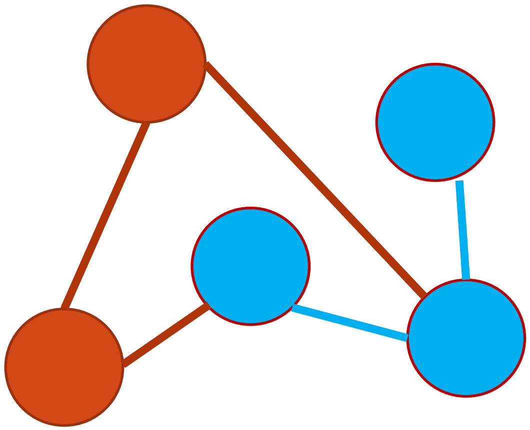
A ***closed path*** is a path where all vertices have 2 edges incident to them

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



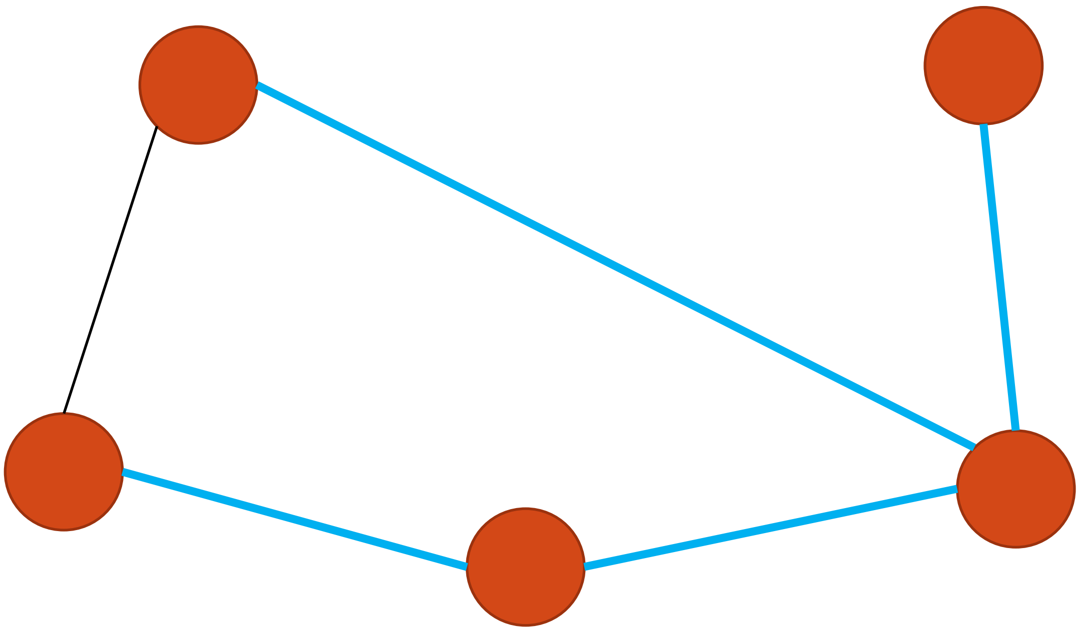
Subgraphs

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



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",…}
```

OR

```
List<String> vertices;
vertices.add("Seattle");…
```

OR

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

vertices[0]	Seattle
vertices[1]	San Francisco
vertices[2]	Los Angeles
vertices[3]	Denver
vertices[4]	Kansas City
vertices[5]	Chicago
vertices[6]	Boston
vertices[7]	New York
vertices[8]	Atlanta
vertices[9]	Miami
vertices[10]	Dallas
vertices[11]	Houston

In all these representations the vertices can be conveniently labeled using the indexes `0, 1, 2, …, n-1`, for a graph for `n` vertices.

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}, // edges starting from 0
  {1, 0}, {1, 2}, {1, 3}, // edges starting from 1
  {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 number of edges in advance

Representing Edges: Adjacency Matrix

- Knowing 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];
```

for Seattle	neighbors[0]	1	3	5			
San Francisco	neighbors[1]	0	2	3			
Los Angeles	neighbors[2]	1	3	4	10		
Denver	neighbors[3]	0	1	2	4	5	
Kansas City	neighbors[4]	2	3	5	7	8	10
Chicago	neighbors[5]	0	3	4	6	7	
Boston	neighbors[6]	5	7				
New York	neighbors[7]	4	5	6	8		
Atlanta	neighbors[8]	4	7	9	10	11	
Miami	neighbors[9]	8	11				
Dallas	neighbors[10]	2	4	8	11		
Houston	neighbors[11]	8	9	10			

OR

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

Representing Edges: Adjacency Edge List

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

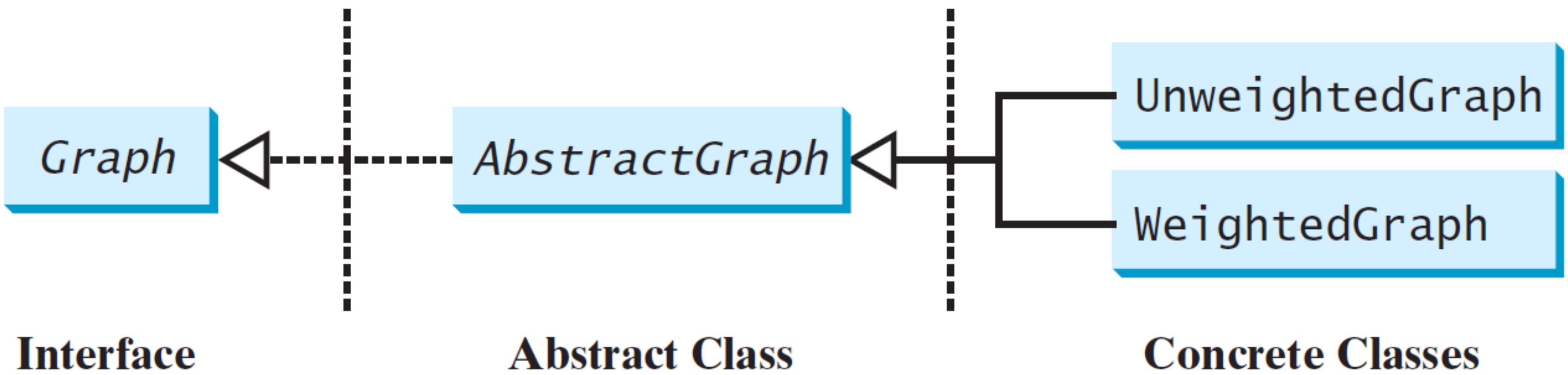
Seattle	neighbors[0]	Edge(0, 1)	Edge(0, 3)	Edge(0, 5)			
San Francisco	neighbors[1]	Edge(1, 0)	Edge(1, 2)	Edge(1, 3)			
Los Angeles	neighbors[2]	Edge(2, 1)	Edge(2, 3)	Edge(2, 4)	Edge(2, 10)		
Denver	neighbors[3]	Edge(3, 0)	Edge(3, 1)	Edge(3, 2)	Edge(3, 4)	Edge(3, 5)	
Kansas City	neighbors[4]	Edge(4, 2)	Edge(4, 3)	Edge(4, 5)	Edge(4, 7)	Edge(4, 8)	Edge(4, 10)
Chicago	neighbors[5]	Edge(5, 0)	Edge(5, 3)	Edge(5, 4)	Edge(5, 6)	Edge(5, 7)	
Boston	neighbors[6]	Edge(6, 5)	Edge(6, 7)				
New York	neighbors[7]	Edge(7, 4)	Edge(7, 5)	Edge(7, 6)	Edge(7, 8)		
Atlanta	neighbors[8]	Edge(8, 4)	Edge(8, 7)	Edge(8, 9)	Edge(8, 10)	Edge(8, 11)	
Miami	neighbors[9]	Edge(9, 8)	Edge(9, 11)				
Dallas	neighbors[10]	Edge(10, 2)	Edge(10, 4)	Edge(10, 8)	Edge(10, 11)		
Houston	neighbors[11]	Edge(11, 8)	Edge(11, 9)	Edge(11, 10)			

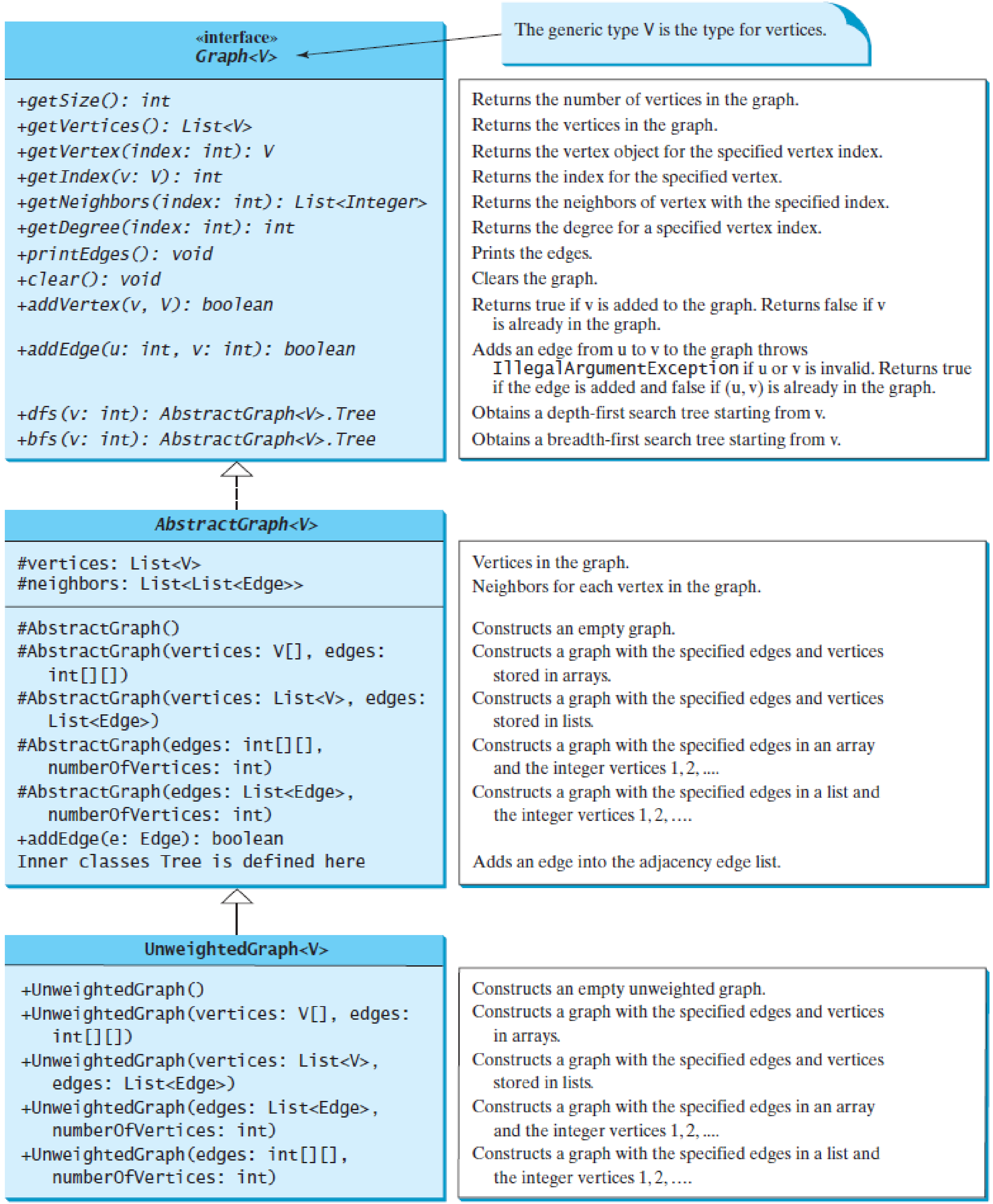
Representing Adjacency Edge List Using ArrayList

```
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, ...

```java
public interface Graph<V> {
  /** 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);
  /** 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();
}
```

```java
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();

  /** 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;
    }
  }
```

```java
@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;
  }
}

/** 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());
}
```

```
/** 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]);
  }
}
@Override /** Add an edge to the graph */
public boolean addEdge(int u, int v) {
  return addEdge(new Edge(u, v));
}
/** Create adjacency lists for each vertex */
private void createAdjacencyLists(List<Edge> edges,int numberOfVertices){
  for (Edge edge: edges) {
    addEdge(edge.u, edge.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;
  }
```

```java
/** 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);
}

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

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

@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 (out)degree for a specified vertex */
public int getDegree(int u) {
  return neighbors.get(u).size();
}
```

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

}
```

```
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:

AbstractGraph<V>.Tree	
`-root: int`	The root of the tree.
`-parent: int[]`	The parents of the vertices.
`-searchOrder: List<Integer>`	The orders for traversing the vertices.
`+Tree(root: int, parent: int[], searchOrder: List<Integer>)`	Constructs a tree with the specified root, parent, and `searchOrder`.
`+getRoot(): int`	Returns the root of the tree.
`+getSearchOrder(): List<Integer>`	Returns the order of vertices searched.
`+getParent(index: int): int`	Returns the parent for the specified vertex index.
`+getNumberOfVerticesFound(): int`	Returns the number of vertices searched.
`+getPath(index: int): List<V>`	Returns a list of vertices from the specified vertex index to the root.
`+printPath(index: int): void`	Displays a path from the root to the specified vertex.
`+printTree(): void`	Displays tree with the root and all edges.

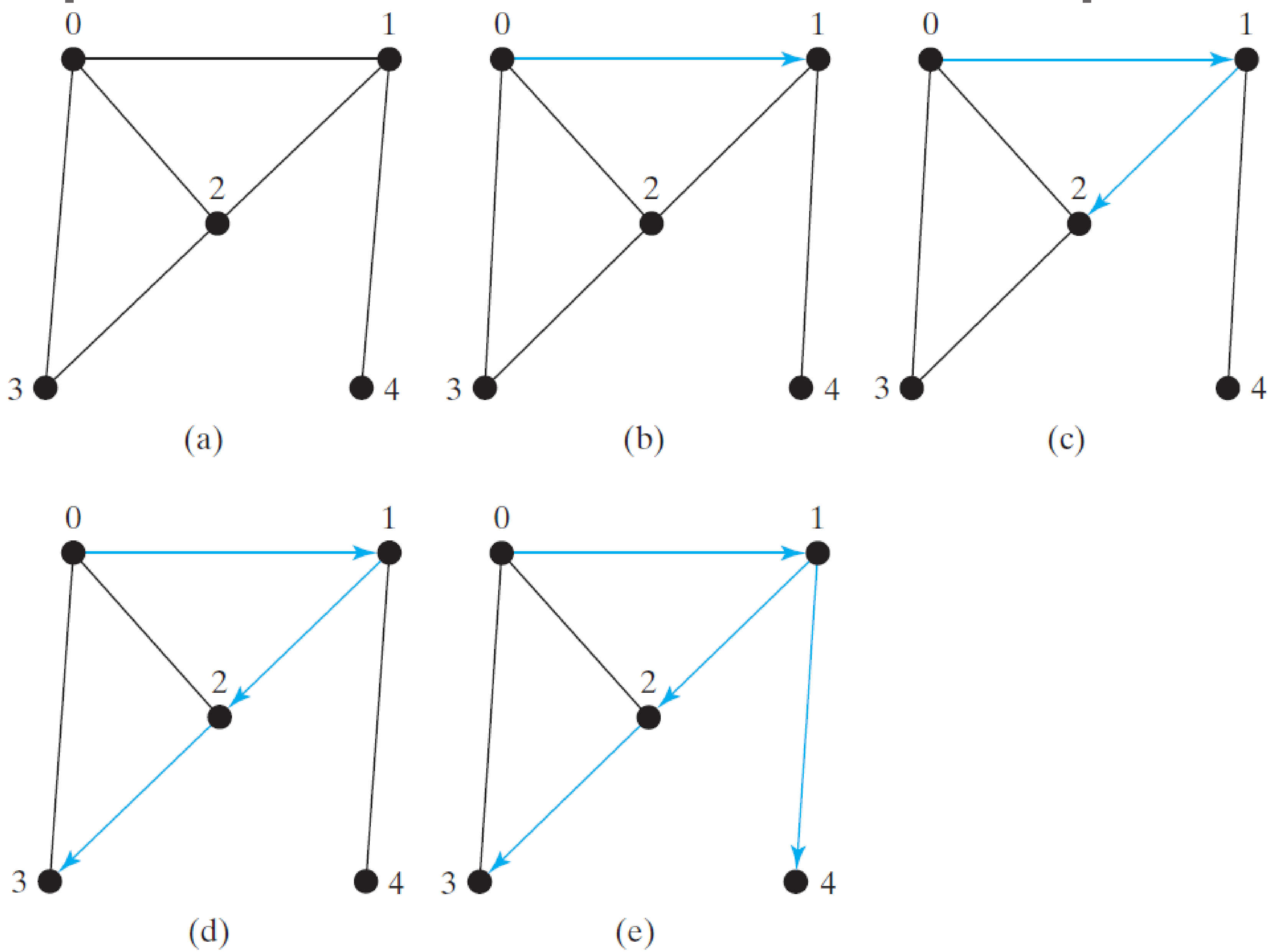
Depth-First Search

- 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

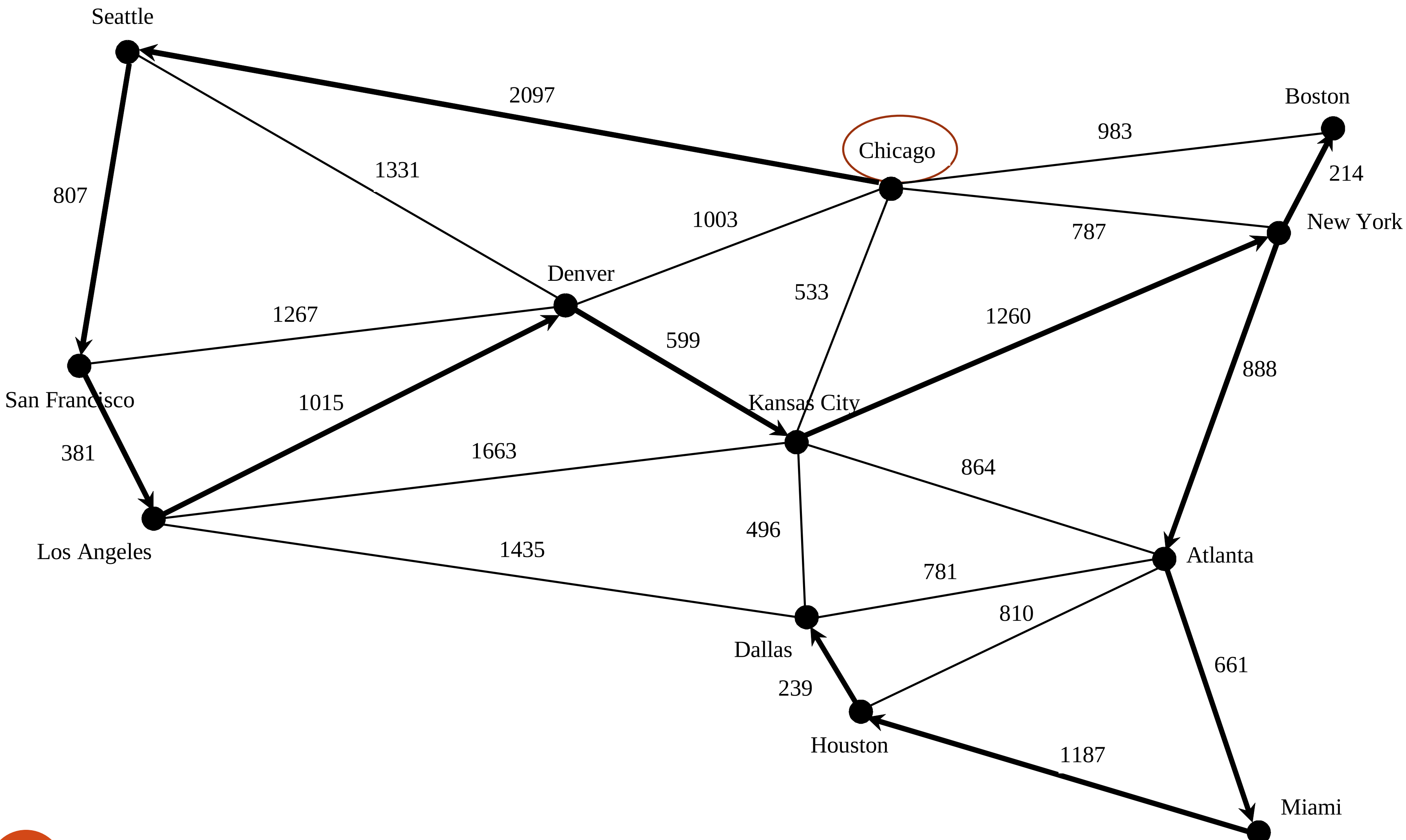
```
Input: G = (V, E) and a starting vertex v
Output: a DFS tree rooted at v
Tree dfs(vertex v) {
  visit v;
  for each neighbor w of v
    if (w has not been visited) {
      set v as the parent for w;
      dfs(w);
    }
}
```

- 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



Depth-First Search Example



```java
@Override /** Obtain a DFS tree starting from vertex v */
public Tree dfs(int v) {
  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()];
  List<Integer> searchOrder = new ArrayList();
  // Recursively search
  dfs(v, parent, isVisited, searchOrder);
  // Return the search tree
  return new Tree(v, parent, searchOrder);
}

/** Recursive method for DFS search */
private void dfs(int u, int[] parent, boolean[] isVisited,
     List<Integer> searchOrder) {
  // 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, isVisited, searchOrder); // Recursive search
    }
}
```

```java
// 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];
  }
```

```java
/** 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) + " ");
}

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

```java
  /** 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();
  }
}
```

```java
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("the 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
 the parent of Seattle is Chicago
 the parent of San Francisco is Seattle
 the parent of Los Angeles is San Francisco
 the parent of Denver is Los Angeles
 the parent of Kansas City is Denver
 the parent of New York is Kansas City
 the parent of Boston is New York
 the parent of Atlanta is New York
 the parent of Miami is Atlanta
 the parent of Houston is Miami
 the parent of Dallas is Houston
```

Applications of the DFS

- Detecting whether a graph is **<u>connected</u>**
  - 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)
  - Search the graph starting from one of the 2 vertexes
  - Check if the second vertex is reached by DFS

Applications of the DFS

- Finding all connected components:
  - A ***connected component*** is a maximal connected subgraph in which every pair of vertices are connected by a path
    - Label all vertexes as unreached
    - Repeat until no vertex is unreached
      - Start from any unreached vertex and compute DFS (marking all reached vertexes, including the first vertex, as reached) -> this DFS is one connected component

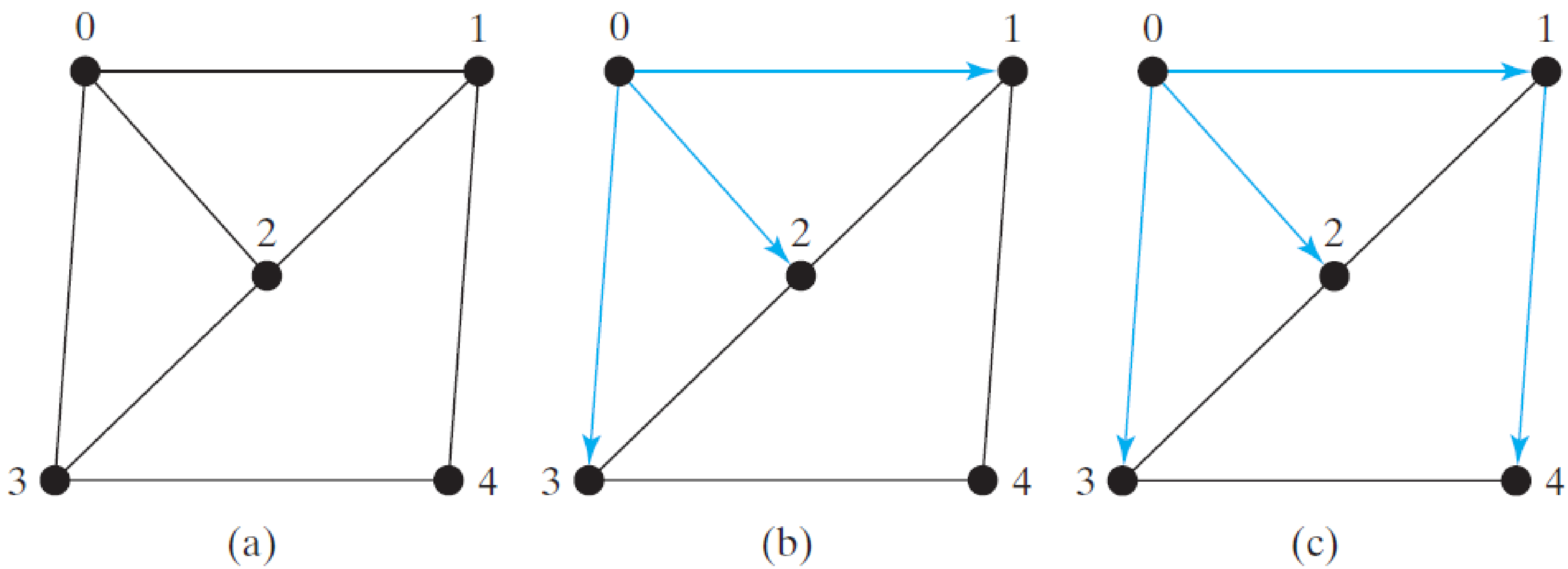
Breadth-First Search

- 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 (e.g., prints) 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

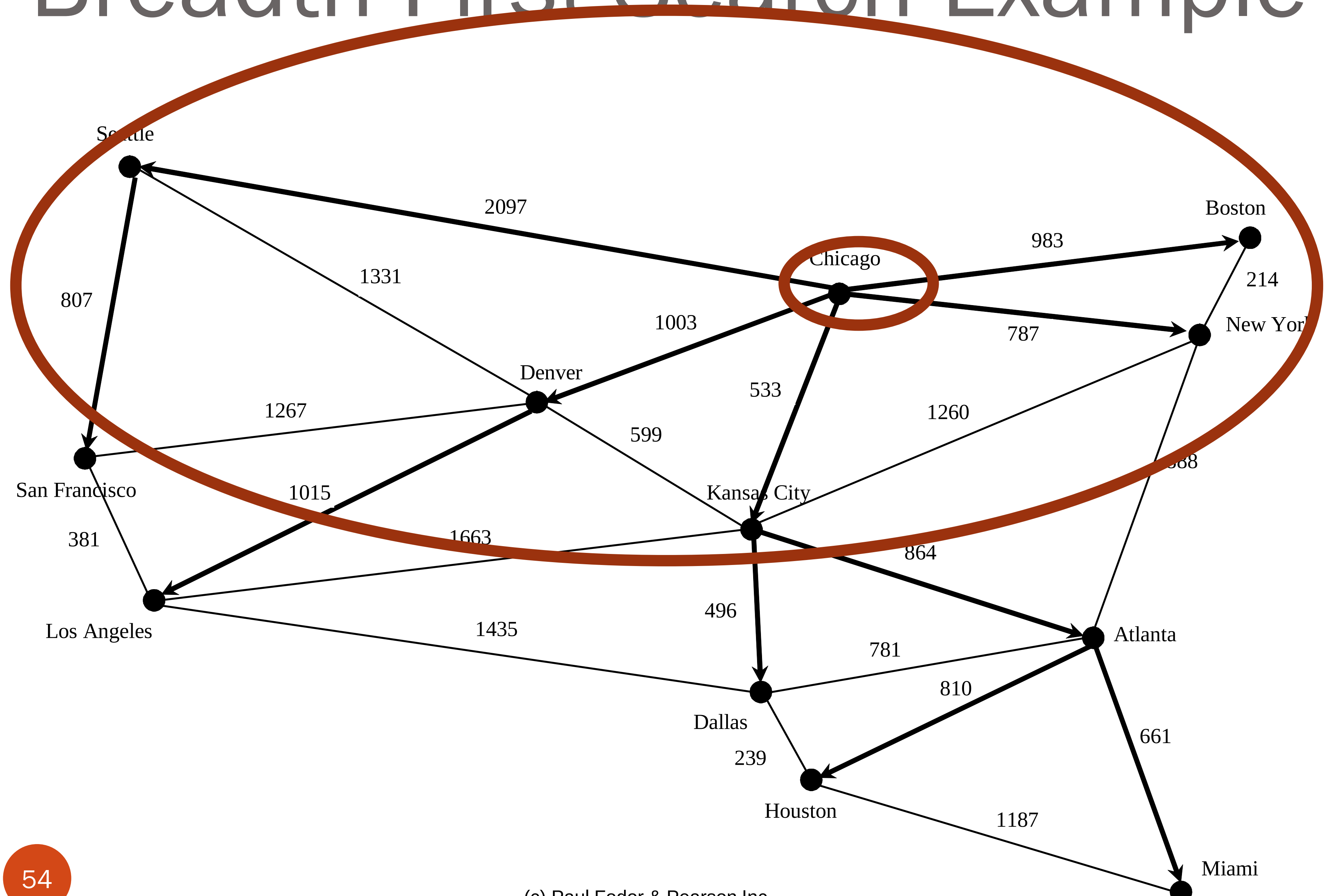


Queue: 0 isVisited[0] = true

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

Queue: 2 3 4 isVisited[4] = true

Breadth-First Search Example



```java
@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);
}
```

```java
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
    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("the 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
the parent of Seattle is Chicago
the parent of San Francisco is Seattle
the parent of Los Angeles is Denver
the parent of Denver is Chicago
the parent of Kansas City is Chicago
the parent of Boston is Chicago
the parent of New York is Chicago
the parent of Atlanta is Kansas City
the parent of Miami is Atlanta
the parent of Dallas is Kansas City
the 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)
  - check is the size of the spanning tree is the same with the number of vertices
- Detecting whether there is a path between two vertices
  - Compute the BFS from the first vertex and check if the second vertex is reached
- 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
- Detect 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, while another is the union of the even number stratas