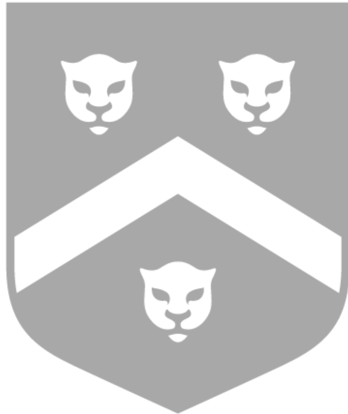


Graphs

Professor Frank Kreimendahl

School of Computing and Data Science

Wentworth Institute of Technology

November 14, 2022

Graph ADT
Graphs
Terminology
Graph Class

Graph ADT

Graphs

- Trees describe relationships in a strict hierarchy
- Graphs describe networks with more interconnected relationships
- Graphs can represent many useful things:
  - Devices on an electronic network
  - Components on a silicon chip
  - Road maps
  - Course prerequisites
  - States and transitions in a system

Graph ADT
Terminology
Definition
Example
Visualization
Edge Features
More Terminology
Edge Features
Graph Class

Terminology

Graph ADT
Terminology
Definition
Example
Visualization
Edge Features
More Terminology
Edge Features
Graph Class

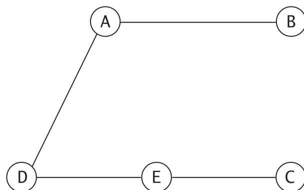
Definition

- A graph is a data structure that consists of a set of *vertices* (or nodes) and a set of *edges* (relations) between pairs of vertices
- Edges represent paths or connections between vertices
- Both the set of vertices and the set of edges must be finite
- Either set may be empty

Graph ADT
Terminology
Definition
Example
Visualization
Edge Features
More Terminology
Edge Features
Graph Class

Example

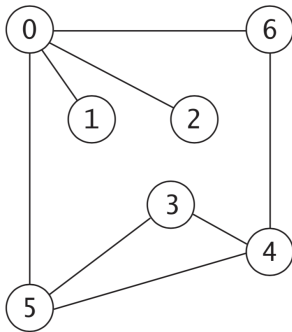
V = {A, B, C, D, E}
E = {{A, B}, {A, D}, {C, E}, {D, E}}

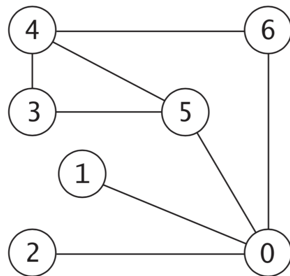


- Each edge is represented by the two vertices it connects
- If there is an edge between vertices x and y, there is a *path* from x to y and vice versa

Visualization

The physical layout of the vertices and their labeling is not relevant



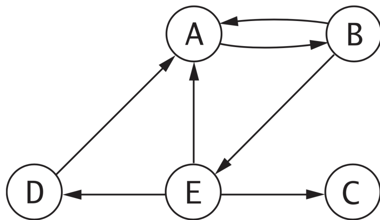


Two equivalent graphs

Graph ADT
Terminology
Definition
Example
Visualization
Edge Features
More Terminology
Edge Features
Graph Class

Edge Features

- Edges are *undirected* if they represent a transition in both directions
- Edges are *directed* if they represent a transition in only one direction
- Edges are *unweighted* if all transitions' costs are equal
- Edges are *weighted* if there are different costs associated with different transitions



Directed graph with arrowed edges

Graph ADT
Terminology
Definition
Example
Visualization
Edge Features
More Terminology
Edge Features
Graph Class

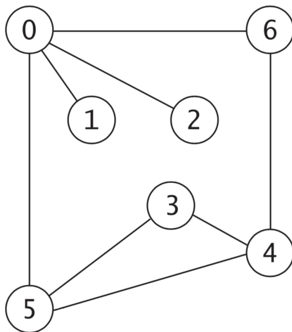
More Terminology

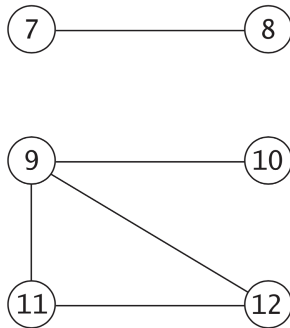
- Two vertices are *adjacent* and *neighbors* if there is an edge from one vertex to the other
- A *path* is a sequence of edges between adjacent vertices
- A *simple path* is a sequence with all unique edges and vertices (except maybe the first/last vertex)
- A *cycle* is a simple path with the same start and end vertex

Graph ADT
Terminology
Definition
Example
Visualization
Edge Features
More Terminology
Edge Features
Graph Class

Edge Features

- A graph is *connected* if there is a path from every vertex to every other vertex
- A *connected component* is a subset of vertices that are connected





An *unconnected* graph with three connected components

Graph ADT

Terminology

Graph Class

Requirements

Graph Class

Vertices and Edges

Representations

Adjacency List

AbstractGraph

ListGraph

Graph Algorithms

Graph Class

Graph ADT
Terminology
Graph Class
Requirements
Graph Class
Vertices and Edges
Representations
Adjacency List
AbstractGraph
ListGraph
Graph Algorithms

Requirements

- Java does not provide a Graph data structure
- Desired operations:
  - Create a graph with a specific number of vertices
  - Iterate through all vertices
  - Iterate through all neighbors of a vertex
  - Insert an edge
  - Iterate through all edges
  - Check if an edge exists
  - Determine edge weight

Graph ADT
Terminology
Graph Class
Requirements
Graph Class
Vertices and Edges
Representations
Adjacency List
AbstractGraph
ListGraph
Graph Algorithms

Graph Class

Data Field	Attribute
private int dest	The destination vertex for an edge.
private int source	The source vertex for an edge.
private double weight	The weight.
Constructor	**Purpose**
public Edge(int source, int dest)	Constructs an Edge from source to dest. Sets the weight to 1.0.
public Edge(int source, int dest, double w)	Constructs an Edge from source to dest. Sets the weight to w.
Method	**Behavior**
public boolean equals(Object o)	Compares two edges for equality. Edges are equal if their source and destination vertices are the same. The weight is not considered.
public int getDest()	Returns the destination vertex.
public int getSource()	Returns the source vertex.
public double getWeight()	Returns the weight.
public int hashCode()	Returns the hash code for an edge. The hash code depends only on the source and destination.
public String toString()	Returns a string representation of the edge.

Graph ADT
Terminology
Graph Class
Requirements
Graph Class
Vertices and Edges
Representations
Adjacency List
AbstractGraph
ListGraph
Graph Algorithms

Vertices and Edges

- Vertex representations:
  - Each vertex is represented by an integer, starting at 0
- Edge class:
  - Requires source vertex
  - Requires destination vertex
  - Requires weight
  - Edges are directed so we will use two edges to represent a single edge in an undirected graph

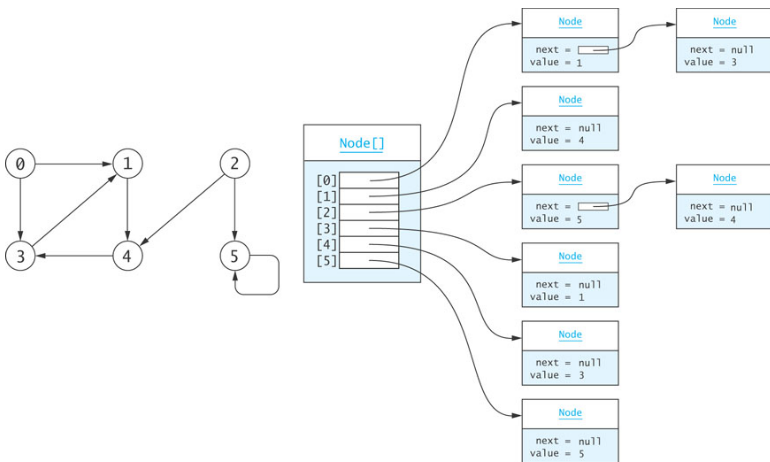
Graph ADT
Terminology
Graph Class
Requirements
Graph Class
Vertices and Edges
Representations
Adjacency List
AbstractGraph
ListGraph
Graph Algorithms

Representations

- There are two common graph representations
- Depending on the intended use, one or the other is more efficient
- Adjacency list:
  - Uses an array of lists
  - Each element represents a vertex, and each entry in its list represents adjacent vertices
- Adjacency matrix:
  - Uses a square two-dimensional array
  - Each element records whether there is a connection from the vertex in that row to the vertex in that column
  - Elements can hold weights as well

Adjacency List

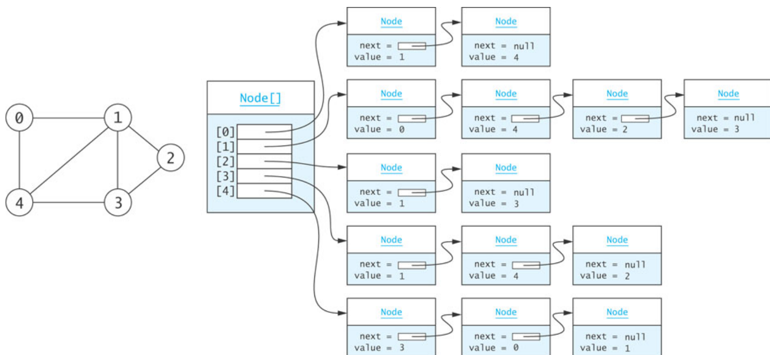
Graph ADT
Terminology
Graph Class
Requirements
Graph Class
Vertices and Edges
Representations
Adjacency List
AbstractGraph
ListGraph
Graph Algorithms



Array of linked lists that hold adjancent nodes

Graph ADT
Terminology
Graph Class
Requirements
Graph Class
Vertices and Edges
Representations
Adjacency List
AbstractGraph
ListGraph
Graph Algorithms

Adjacency List



Undirected graph with edges going both ways

Graph ADT
Terminology
Graph Class
Requirements
Graph Class
Vertices and Edges
Representations
Adjacency List
AbstractGraph
ListGraph
Graph Algorithms

AbstractGraph Class

Data Field	Attribute
`private boolean directed`	**`true`** if this is a directed graph.
`private int numV`	The number of vertices.
Constructor	**Purpose**
`public AbstractGraph(int numV, boolean directed)`	Constructs an empty graph with the specified number of vertices and with the specified `directed` flag. If `directed` is **`true`**, this is a directed graph.
Method	**Behavior**
`public int getNumV()`	Gets the number of vertices.
`public boolean isDirected()`	Returns **`true`** if the graph is a directed graph.
`public void loadEdgesFromFile(Scanner scan)`	Loads edges from a data file.
`public static Graph createGraph (Scanner scan, boolean isDirected, String type)`	Factory method to create a graph and load the data from an input file.

Graph ADT
Terminology
Graph Class
Requirements
Graph Class
Vertices and Edges
Representations
Adjacency List
AbstractGraph
ListGraph
Graph Algorithms

ListGraph Class

Data Field	Attribute
`private List<Edge>[] edges`	An array of `List`s to contain the edges that originate with each vertex.
Constructor	**Purpose**
`public ListGraph(int numV, boolean directed)`	Constructs a graph with the specified number of vertices and directionality.
Method	**Behavior**
`public Iterator<Edge> edgeIterator(int source)`	Returns an iterator to the edges that originate from a given vertex.
`public Edge getEdge(int source, int dest)`	Gets the edge between two vertices.
`public void insert(Edge e)`	Inserts a new edge into the graph.
`public boolean isEdge(int source, int dest)`	Determines whether an edge exists from vertex `source` to `dest`.

Graph ADT
Terminology
Graph Class
Requirements
Graph Class
Vertices and Edges
Representations
Adjacency List
AbstractGraph
ListGraph
Graph Algorithms

Data fields

```
import java.util.*;

/** A ListGraph is an extension of the
    AbstractGraph abstract class that uses an array
    of lists to represent the edges. */
public class ListGraph extends AbstractGraph {
  // Data Field
  /** An array of Lists to contain the edges that
      originate with each vertex. */
  private List<Edge>[] edges;
. . .
}
```

Graph ADT
Terminology
Graph Class
Requirements
Graph Class
Vertices and Edges
Representations
Adjacency List
AbstractGraph
ListGraph
Graph Algorithms

Constructor

```
/** Construct a graph with the specified number of
    vertices and directionality.
    @param numV The number of vertices
    @param directed The directionality flag
*/
public ListGraph(int numV, boolean directed) {
  super(numV, directed);
  edges = new List[numV];
  for (int i = 0; i < numV; i++) {
    edges[i] = new LinkedList<Edge>();
  }
}
```

Graph ADT
Terminology
Graph Class
Requirements
Graph Class
Vertices and Edges
Representations
Adjacency List
AbstractGraph
ListGraph
Graph Algorithms

isEdge

```
/** Determine whether an edge exists.
    @param source The source vertex
    @param dest The destination vertex
    @return true if there is a (src, dst) edge
*/
public boolean isEdge(int src, int dst) {
  return edges[src].contains(new Edge(src, dst));
}
```

Graph ADT
Terminology
Graph Class
Requirements
Graph Class
Vertices and Edges
Representations
Adjacency List
AbstractGraph
ListGraph
Graph Algorithms

insert and edgeIterator

```
/** Insert a new edge into the graph.
    @param edge The new edge
*/
public void insert(Edge edge) {
  edges[edge.getSource()].add(edge);
  if (!isDirected()) {
  edges[edge.getDest()].add(new Edge(edge.getDest(),
                              edge.getSource(),
                              edge.getWeight()));
  }
}
```

```
public Iterator<Edge> edgeIterator(int source) {
  return edges[source].iterator();
}
```

Graph ADT
Terminology
Graph Class
Requirements
Graph Class
Vertices and Edges
Representations
Adjacency List
AbstractGraph
ListGraph
Graph Algorithms

getEdge

```
/** Get the edge between two vertices. If an
    edge does not exist, an Edge with a weight
    of Double.POSITIVE_INFINITY is returned.
    @param source The source
    @param dest The destination
    @return the edge between these two vertices
*/
public Edge getEdge(int source, int dest) {
  Edge target =
    new Edge(source, dest, Double.POSITIVE_INFINITY);
  for (Edge edge : edges[source]) {
    if (edge.equals(target))
      return edge; // Desired edge found, return it.
  }
  return target; // Desired edge not found.
}
```

Graph ADT
Terminology
Graph Class
Requirements
Graph Class
Vertices and Edges
Representations
Adjacency List
AbstractGraph
ListGraph
Graph Algorithms

Graph Algorithms

- Graphs support a variety of analysis to solve many problems
- Many algorithms follow common forms, though the specifics vary

ANALYZEGRAPH(G)

1: **for** vertex u in G **do**
2:     **for** each vertex v adjacent to u **do**
3:         Do something with vertex v or edge (u,v)