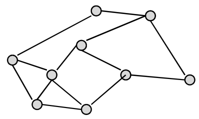


CSIS 3103

Graphs

Graphs

Tree nodes have only one parent
Graphs do not have this limitation
Graphs algorithms are used in

- large communication networks
- software that makes the Internet function

Graphs describe

- roads maps
- airline routes
- course prerequisites

Graph Terminology

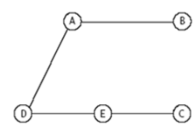
Graph: A data structure that consists of a set of *vertices* (nodes) and a set of *edges* (relations)

- Edges represent paths or connections between the vertices
- The set of vertices and the set of edges must both be finite

Visual Representation of Graphs

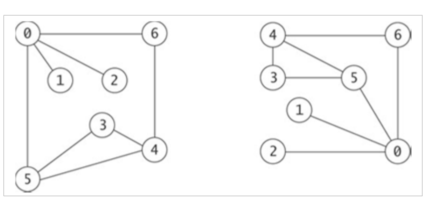
Vertices: points or labeled circles
Edges: lines joining the vertices

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



Visual Representation of Graphs

The layout of the vertices and the labeling are not relevant



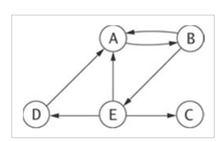
V = {0, 1, 2, 3, 4, 5, 6}
E = {{0, 1}, {0, 2}, {0, 5}, {0, 6}, {3, 5}, {3, 4}, {4, 5}, {4, 6}}

Directed and Undirected Graphs

Undirected graph: The edges have no direction

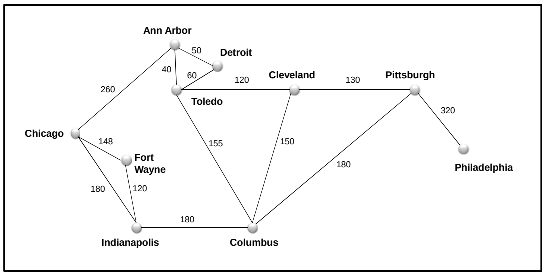
Directed graph (digraph): Each edge is directed from one vertex to another (or the same) vertex

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



Weighted Graphs

A graph in which each edge carries a value



More Terminology

Adjacent vertices: Two vertices in a graph that are connected by an edge

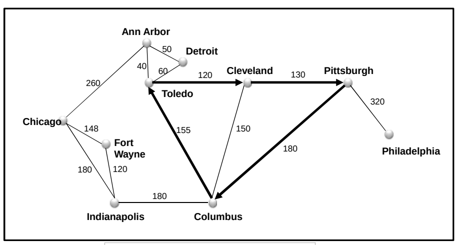
Path: A sequence of vertices where each successive vertex is adjacent to its predecessor

Cycle: A path from a node back to itself

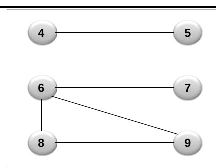
Acyclic graph: A graph with no cycles

Connected graph: A graph in which every vertex is reachable from every other vertex

Cycle
Connected graph



Unconnected graph



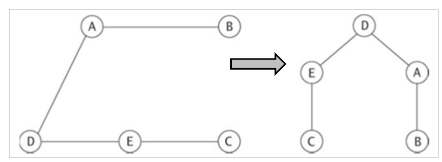
Graphs and Trees

A tree is a special case of a graph

Any graph that is

- connected
- contains no cycles

can be viewed as a tree by making one of the vertices the root



The Graph ADT

A graph ADT needs to support

- Creating a graph with a specified number of vertices
- Iterating through all of the vertices in the graph
- Iterating through the vertices that are adjacent to a specified vertex
- Determining whether an edge exists between two vertices
- Finding the weight of an edge between two vertices
- Inserting an edge into the graph

The Java API does not provide a Graph ADT

The Edge Class (for weighted digraphs)

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.

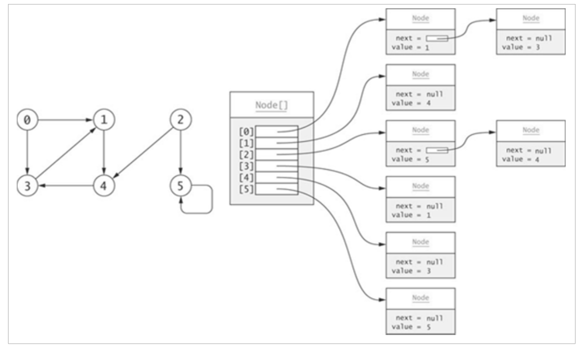
Implementing the Graph ADT

The most common representations of graphs:

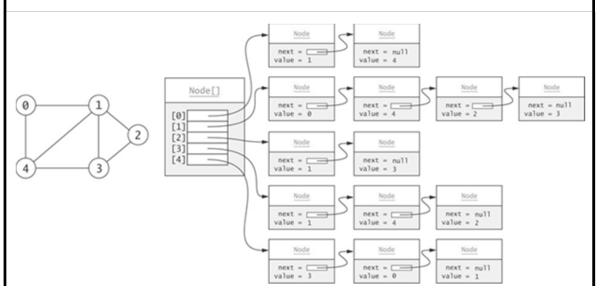
Adjacency lists: Edges are represented by an array of lists where each list stores the vertices adjacent to a particular vertex

Adjacency matrix: Edges are represented by a two dimensional array

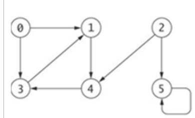
Adjacency List



Adjacency List

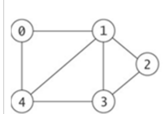


Adjacency Matrix



Column

Row	[0]	[1]	[2]	[3]	[4]	[5]
[0]		1.0		1.0		
[1]					1.0	
[2]					1.0	1.0
[3]		1.0				
[4]				1.0		
[5]						1.0

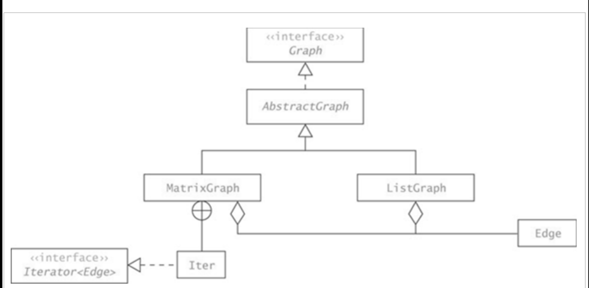


Column

Row	[0]	[1]	[2]	[3]	[4]
[0]		1.0			1.0
[1]	1.0		1.0	1.0	1.0
[2]		1.0		1.0	
[3]		1.0	1.0		1.0
[4]	1.0	1.0		1.0	

(Unweighted graph entries can be boolean values)

The Graph Class Hierarchy



Class AbstractGraph

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.

The ListGraph Class

Data Field	Attribute
private List<Edge>[] edges	An array of Lists 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 Traversals

- Most graph algorithms involve visiting each vertex in a systematic order
- The most common traversal algorithms
  - Breadth first search
  - Depth first search

Breadth-First Search

Start at a vertex and visit it,
then visit all vertices that are adjacent to it,
then visit vertices with path length 2 from it,
path length 3, etc.

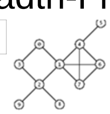
- Must visit all nodes for which the shortest path from the start node is length k before visiting any node for which the shortest path from the start node is length $k + 1$

Algorithm for Breadth-First Search

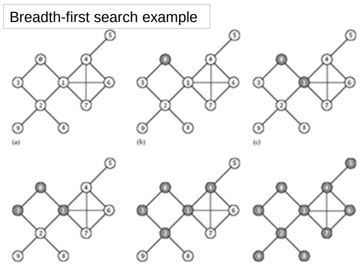
Algorithm for Breadth-First Search

1. Take an arbitrary start vertex, mark it identified (color it light blue), and place it in a queue.
2. **while** the queue is not empty
3. Take a vertex, *u*, out of the queue and visit *u*.
4. **for** all vertices, *v*, adjacent to this vertex, *u*
5. **if** *v* has not been identified or visited
6. Mark it identified (color it light blue).
7. Insert vertex *v* into the queue.
8. We are now finished visiting *u* (color it dark blue).

Breadth-First Search



Breadth-first search example



Breadth-First Search

Trace of Breadth-First Search of Graph in Figure 12.15

Vertex Being Visited	Queue Contents After Visit	Visit Sequence
0	1 3	0
1	3 2 4 6 7	0 1
3	2 4 6 7	0 1 3
2	4 6 7 8 9	0 1 3 2
4	6 7 8 9 5	0 1 3 2 4
6	7 8 9 5	0 1 3 2 4 6
7	8 9 5	0 1 3 2 4 6 7
8	9 5	0 1 3 2 4 6 7 8
9	5	0 1 3 2 4 6 7 8 9
5	empty	0 1 3 2 4 6 7 8 9 5

Depth-First Search

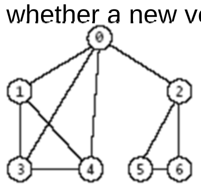
Start at a vertex and visit it,

choose one adjacent vertex to visit,

then choose a vertex adjacent to that vertex…,
…and so on until you can go no further;

then back up and see whether a new vertex can

FIGURE 12.18
Graph to Be Traversed Depth First



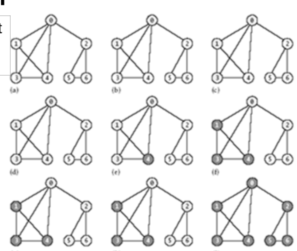
Algorithm for Depth-First Search

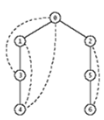
Algorithm for Depth-First Search

1. Mark the current vertex, *u*, visited (color it light blue), and enter it in the discovery order list
2. **for** each vertex, *v*, adjacent to the current vertex, *u*
3. **if** *v* has not been visited
4. Set parent of *v* to *u*.
5. Recursively apply this algorithm starting at *v*.
6. Mark *u* finished (color it dark blue) and enter *u* into the finish order list.

Depth-First Search

Depth-first search example





Depth-First Search Tree of Figure 12.18

Depth-First Search

Trace of Depth-First Search of Figure 12.19

Operation	Adjacent Vertices	Discovery (Visit) Order	Finish Order
Visit 0	1, 2, 3, 4	0	
Visit 1	0, 3, 4	0, 1	
Visit 3	0, 1, 4	0, 1, 3	
Visit 4	0, 1, 3	0, 1, 3, 4	
Finish 4			4
Finish 3			4, 3
Finish 1			4, 3, 1
Visit 2	0, 5, 6	0, 1, 3, 4, 2	
Visit 5	2, 6	0, 1, 3, 4, 2, 5	
Visit 6	2, 5	0, 1, 3, 4, 2, 5, 6	
Finish 6			4, 3, 1, 6
Finish 5			4, 3, 1, 6, 5
Finish 2			4, 3, 1, 6, 5, 2
Finish 0			4, 3, 1, 6, 5, 2, 0