Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

Slides are prepared from

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

The Graph ADT

- A graph is a collection of vertices and edges
- The abstraction as a combination of three data types: ***Vertex***, ***Edge***, and ***Graph***
- A ***Vertex*** is *a lightweight object* that stores an arbitrary element provided by the user (e.g., an airport code)
- We assume the element can be retrieved with the **getElement()** method
- An ***Edge*** also stores an associated object (e.g., a flight number, travel distance, cost), which is returned by its **getElement()** method
- The primary abstraction for a graph is the Graph ADT
- We presume that a graph can be either undirected or directed, with the designation declared upon construction

The Graph ADT

- The Graph ADT includes the following methods:

numVertices( ):	Returns the number of vertices of the graph.
vertices( ):	Returns an iteration of all the vertices of the graph.
numEdges( ):	Returns the number of edges of the graph.
edges( ):	Returns an iteration of all the edges of the graph.
getEdge(u, v):	Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).
endVertices(e):	Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.

The Graph ADT

- The Graph ADT includes the following methods:

opposite(v, e):	For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.
outDegree(v):	Returns the number of outgoing edges from vertex v.
inDegree(v):	Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).
outgoingEdges(v):	Returns an iteration of all outgoing edges from vertex v.
incomingEdges(v):	Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).

The Graph ADT

- The Graph ADT includes the following methods:

insertVertex(x):	Creates and returns a new Vertex storing element x.
insertEdge(u, v, x):	Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.
removeVertex(v):	Removes vertex v and all its incident edges from the graph.
removeEdge(e):	Removes edge e from the graph.

Data Structures for Graphs

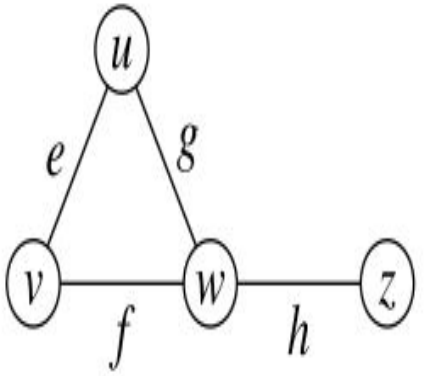
- There are four data structures for representing a graph
- In each representation, we maintain a collection to store the vertices of a graph
- The four representations differ greatly in the way they organize the edges
  - In an **edge list**, we maintain an unordered list of all edges. This minimally suffices, but there is no efficient way to locate a particular edge (u,v), or the set of all edges incident to a vertex v
  - In an **adjacency list**, we additionally maintain, for each vertex, a separate list containing those edges that are incident to the vertex. This organization allows us to more efficiently find all edges incident to a given vertex
  - An **adjacency map** is similar to an adjacency list, but the secondary container of all edges incident to a vertex is organized as a map, rather than as a list, with the adjacent vertex serving as a key. This allows more efficient access to a specific edge (u,v), for example, in O(1) expected time with hashing
  - An **adjacency matrix** provides worst-case O(1) access to a specific edge (u,v) by maintaining an n ×n matrix, for a graph with n vertices. Each slot is dedicated to storing a reference to the edge (u,v) for a particular pair of vertices u and v; if no such edge exists, the slot will store null

A summary of the performance of edge structures

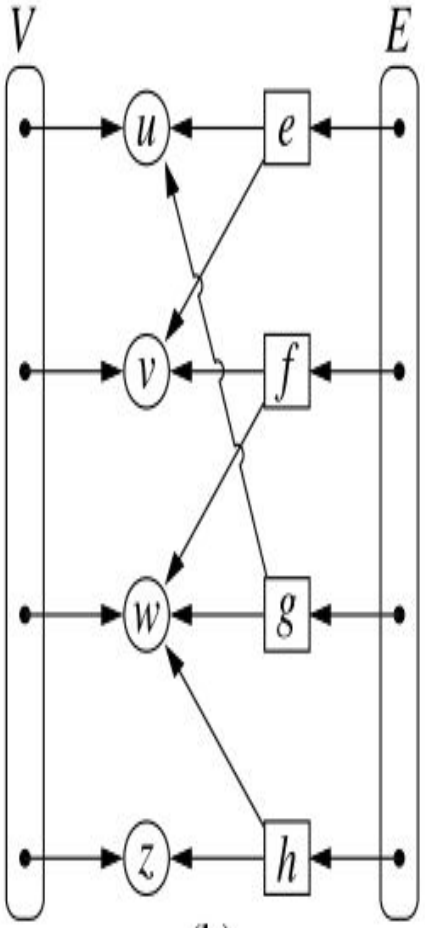
Method	Edge List	Adj. List	Adj. Map	Adj. Matrix
numVertices( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
numEdges( )	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices( )	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges( )	$O(m)$	$O(m)$	$O(m)$	$O(m)$
getEdge(u, v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
outDegree(v) inDegree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
outgoingEdges(v) incomingEdges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insertVertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
removeVertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insertEdge(u, v, x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
removeEdge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

Edge List Structure

- The edge list structure is possibly the simplest, though not the most efficient, representation of a graph ***G***
- All vertex objects are stored in an unordered list ***V***, and all edge objects are stored in an unordered list ***E***
- Collections V and E are represented with doubly linked lists using **LinkedPositionalList** class



(a)



(b)

Edge List Structure

Vertex Objects

- The vertex object for a vertex ***v*** storing element ***x*** has instance variables for:
  - A reference to element ***x***, to support the ***getElement()*** method
  - A reference to the position of the vertex instance in the list ***V***, thereby allowing v to be efficiently removed from ***V*** if it were removed from the graph

Edge Objects

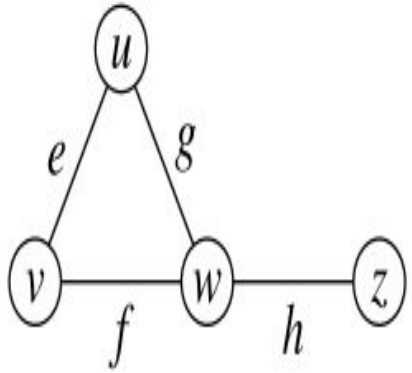
- The edge object for an edge ***e*** storing element ***x*** has instance variables for:
  - A reference to element ***x,*** to support the ***getElement()*** method
  - References to the vertex objects associated with the endpoint vertices of ***e***. These will allow for constant-time support for methods ***endVertices(e)*** and ***opposite(v, e)***.
  - A reference to the position of the edge instance in list ***E***, thereby allowing ***e*** to be efficiently removed from ***E*** if it were removed from the graph

Performance of the Edge List Structure

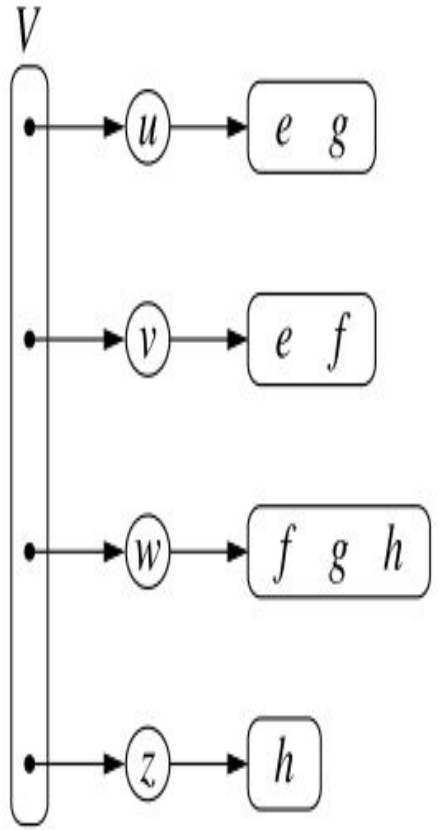
- Space usage is ***O(n+m)*** for representing a graph with ***n*** vertices and ***m*** edges
- Running time:
  - **query methods**: quering the respective list ***V*** or ***E***, the ***numVertices*** and ***numEdges*** methods run in ***O(1)*** time
  - **iterate methods**: iteratinf through the appropriate list, the methods ***vertices*** and ***edges*** run respectively in ***O(n)*** and ***O(m)*** time
  - the ***O(m)*** running times of methods ***getEdge(u, v), outDegree(v)***, and ***outgoingEdges(v)*** (and corresponding methods ***inDegree*** and ***incomingEdges***)
  - **Update methods**: add a new vertex ***insertVertex(x)***, a new edge ***insertEdge(u, v, x)***, or to remover an edge ***removeEdge(e)*** from the graph in ***O(1)*** time. The ***removeVertex(v)*** method has a running time of ***O(m)***

Adjacency List Structure

- The adjacency list structure for a graph adds extra information to the edge list structure that supports direct access to the incident edges (and thus to the adjacent vertices) of each vertex
- Specifically, for each vertex v, we maintain a collection I(v), called the incidence collection of v, whose entries are edges incident to v
- In the case of a directed graph, outgoing and incoming edges can be respectively stored in two separate collections, $I_{out}(v)$ and $I_{in}(v)$
- Traditionally, the incidence collection I(v) for a vertex v is a list, which is why we call this way of representing a graph the adjacency list structure



(a)



(b)

Performance of the Adjacency List Structure

Method	Running Time
numVertices( ), numEdges( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
getEdge(u, v)	$O(\min(\deg(u), \deg(v)))$
outDegree(v), inDegree(v)	$O(1)$
outgoingEdges(v), incomingEdges(v)	$O(\deg(v))$
insertVertex(x), insertEdge(u, v, x)	$O(1)$
removeEdge(e)	$O(1)$
removeVertex(v)	$O(\deg(v))$