**The Graph ADT and Edge Class**

Java doesn’t provide a Graph ADT

To make our own, we need

* Create a graph with the specified number of vertices
  + number of vertices cannot be changed afterwards
  + vertices are labeled from 0 to |V| - 1
* Iterate through all of the vertices in the graph
* Iterate through the vertices that are adjacent to a specified vertex
* Determine whether an edge exists between two vertices
* Determine the weight of an edge between two vertices
* Insert an edge into the graph

**Representing Vertices and Edges**

Vertices

* We can represent the vertices by integers (int variable, not with any label) from 0 up to, but not including, |V|
  + |V| means the cardinality of V, or the number of vertices in the set V

Edges

* Define the class Edge that will contain the
  + source vertex
  + destination vertex
  + weight (unweighted edges use the default value of 1.0)
* Edges are directed
* Undirected graph will have two Edge objects for each edge: one in each direction

Text, letter

Description automatically generated

Edge Class:

*Table

Description automatically generated*

*equals and hashCode should use the same data fields!*

If insert in Graph ADT is public, Edge cannot be inner class.

**Implementing the Graph ADT**

Many of the original publications of graph algorithms and their implementations did not use an object-oriented approach or even abstract data types

Two representations of graphs are most common:

1. Edges are represented by an array of lists called adjacency lists, where each list stores the vertices adjacent to a particular vertex
2. Edges are represented by a 2D array, called an adjacency matrix, with |V| rows and |V| columns

You can implement vertices and edges as sets but implementing with arrays has faster access.

Actually you do not need to represent the vertex set since it is simply from 0 to n-1 (n: # of vertices).

Adjacency List

Representation of a graph uses an array of lists - one list for each vertex

The vertices are in no particular order

Elements of linked lists are edges

Directed graph example:

Diagram

Description automatically generated

Undirected graph example:

Diagram

Description automatically generated

Adjacency Matrix

Use a 2D ( (n-1)x(n-1) )array to represent the graph

ith row and jth column represents the edge from i to j

For an unweighted graph, the entries can be boolean or int values:

* true or 1, edge exists
* false or 0, no edge

If the graph is weighted we have weight of the edge from i to j at [i][j]

* no edge situation can be represented as a special value (such as infinity 🡪 Double.POSITIVE\_INFINITY)

Integer values have benefits over boolean values for some graph algorithms that use matrix multiplication

In an undirected graph, the matrix is symmetric (i to j and j to i both represent the same edge), half of it is unnecessary, so only the lower triangle of the matrix needs to be saved

Diagram

Description automatically generated with low confidence

**Overview of the Graph Class Hierarchy**

Diagram

Description automatically generated

We don’t need external iterator for ListGraph because List in adjacency list has iterator itself.

**Class AbstractGraph**

Graphical user interface, text, application, email

Description automatically generated

Text, letter

Description automatically generated

Text, letter

Description automatically generated



This method should be linear to number of edges

**Class ListGraph**

Graphical user interface, application, table

Description automatically generated

Number of edges for one vertex (elements in a linked list that exists at one index of the edges array) should be n-1 (n = number of vertices) at most. It is not n because we don’t have self edge.

Graphical user interface, text, application

Description automatically generated Text

Description automatically generated Text

Description automatically generated

O(n) but we can also say:

O(k) 🡪 k : # of adjacent vertices to source

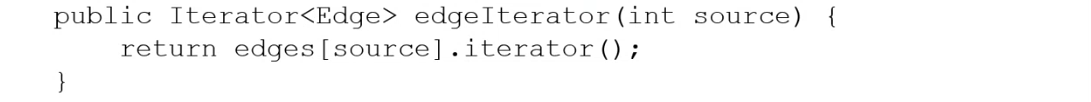
Here in isEdge method, we created an Edge without weight because equals method used in contains method of LinkedList uses equals method of the Edge which compares Edges with source and dest only.

Text, letter

Description automatically generated



This insert method is not fully correct because it may add same edge several times.

 Text, letter

Description automatically generated

O(n) but we can also say:

O(k) 🡪 k : # of adjacent vertices to source



**Class MatrixGraph**

The MatrixGraph class extends the AbstractGraph class by providing a 2D array for storing edge weights:

double[][] edges;

Upon creation of a MatrixGraph class, the constructor sets the number of rows (vertices)

It also needs its own iterator class

* This iterator performs worst than ListGraph’s iterator because we have to check all row (all items in that row can be infinity or there could be 2 edge which exist at first and last places at the row), it could be as bad as linear time so running time of this iterator’s next() is O(n)

**Comparing Implementations**

Time efficiency depends on the algorithm and the density of the graph

The density of a graph is the ratio of |E| to |V|2

* A dense graph is one in which |E| is close to, but less than |V|2 (|E| is proportional to |V|2)
* A sparse graph is one in which |E| is much less than |V|2

We can assume that |E| is

* (|V|2) for a dense graph (MatrixGraph is better)
* o(|V|2) for a sparse graph (ListGraph is better)

Dense is related to how many non-infinity values in the MatrixGraph

If there are many non-infinity values MatrixGraph is better

Many graph algorithms are of the form:

1. for each vertex u in the graph
2. for each vertex v adjacent to u
3. Do something with edge (u, v) *🡪 Assume this step is constant time*

for (int i = 0; i < n; ++i){  
 Iterator<Edge> x = graph.edgeIterator(i);  
 while (x.hasNext()){  
 //You can use x.next() here  
 Edge e = x.next();  
 }  
}

For an adjacency list:

* Step 1 is O(|V|)
* Step 2 is O(||)
  + is the number of edges that originate at vertex u
  + is at most |V| - 1

The combination of steps 1 and 2 represents examining each edge in the graph, giving (|E|)

* We “do something” number of edge (|E|) times

For an adjacency matrix:

* Step 1 is O(|V|)
* Step 2 is O(|V|)

The combination of steps 1 and 2 represents examining each edge in the graph, giving O(|V|2)

* We “do something” number of vertex (|V|) times

The adjacency list gives better performance in a sparse graph, whereas for a dense graph (|E| = |V|2) the performance is the same for both representations

Some graph algorithms are of the form:

1. for each vertex u in the vertices set
2. for each vertex v in the vertices set
3. if (u, v) is an edge
4. Do something with edge (u, v)

We call step 3, |V|2 times.

Step 4 is executed |E| times.

For an adjacency matrix:

* Step 3 tests a matrix value and is (1)
* The overall algorithm is (|V|2)

For an adjacency list:

* Step 3 searches a list and is O(||)
* So the combination of steps 2 and 3 is O(|E|)
* The overall algorithm is (|V||E|)

Her bir linked listi |V| kere search ediyoruz. Linked listi search etmek constant da olabilir, lineer de olabilir. Aranan eleman listin ortasındaymış gibi düşün:

* (x/2) \* |V| 🡪 x : size of the 1st linked list
* (y/2) \* |V| 🡪 y : size of the 2nd linked list
* …
* If you add these all up, you get 🡪 (|V|/2)(x+y+…)



So overall running time is 🡪 (|V||E|)

For a dense graph, the adjacency matrix gives better performance

For a sparse graph (|E| = (n)), the performance is the same for both representations

Thus, for time efficiency:

* if the graph is dense, the adjacency matrix representation is better
* if the graph is sparse, the adjacency list representation is better

A sparse graph will lead to a sparse matrix, or one with many POSITIVE\_INFINITY entries

These values are not included in a list representation so they have no effect on the processing time

In an adjacency matrix,

* storage is allocated for all vertex combinations (or at least half of them)
* the storage required is proportional to |V|2
* for a sparse graph, there is a lot of wasted space

In an adjacency list,

* each edge is represented by a reference to an Edge object which contains data about the source, destination, and weight
* there is also a reference to the next edge in the list
* this is four times as much information as is stored in a matrix representation (which stores only the weight)

The break-even point in terms of storage efficiency occurs when approximately 25% of the adjacency matrix is filled with meaningful data