**COMP232 – Data Structures & Problem Solving**

**Spring 2018**

**Homework #9 - Solutions**

**Graphs**



1. Consider the directed graph shown to the right:

a. Draw the adjacency matrix representation for the graph. Set each entry in the matric to either 1, to indicate the edge exists, or 0 to indicate the edge does not exist.



b. Draw the adjacency list representation for the graph.



2. Implement the getNeighbors method in the CS232DirectedAdjacencyMatrixGraph class. The No2Tests class contains tests that you can use to check your implementation of this functionality.

**public** ArrayList<Integer> getNeighbors(**int** v) {

checkVertex(v);

ArrayList<Integer> neighbors = **new** ArrayList<Integer>();

/\*

\* The neighbors of a vertex are the indices of all of the

\* non-null entries in the vertex's row of the adjacency

\* matrix, so put those indies in to the ArrayList.

\*/

**for** (**int** nv = 0; nv < numVertices(); nv++) {

**if** (edges[v][nv] != **null**) {

neighbors.add(nv);

}

}

**return** neighbors;

}

3. Implement the following methods in the CS232DirectedAdjacencyListGraph class:

a. addEdge and getEdge. The No3aTests class contains tests that you can use to check your implementation of this functionality.

Both addEdge and getEdge make use of the getEdgeObject helper method shown below. The implementations of addEdge and getEdge are on the following page.

/\*

\* Helper method that gets the Edge object associated with an edge.

\*/

**private** Edge<E> getEdgeObject(**int** v1, **int** v2) {

// Use an iterator to search the list of edges for v1

LinkedList<Edge<E>> list = edgeLists[v1];

Iterator<Edge<E>> it = list.iterator();

**while** (it.hasNext()) {

Edge<E> edge = it.next();

**if** (edge.equals(**new** Edge<E>(v1, v2, **null**))) {

// found it.

**return** edge;

}

}

// Edge is not in the graph.

**return** **null**;

}

**public** **void** addEdge(**int** v1, **int** v2, E value) {

checkVertices(v1, v2);

**if** (v1 == v2) {

**throw** **new** IllegalArgumentException(

"Self-edges are not allowed: v1 cannot equal v2.");

}

**if** (value == **null**) {

**throw** **new** IllegalArgumentException("Edge value cannot be null.");

}

Edge<E> edge = getEdgeObject(v1, v2);

**if** (edge != **null**) {

// edge is already in the graph, just replace its value.

edge.edgeObject = value;

} **else** {

// edge not in the graph.

// Add the edge to the list for v1.

edgeLists[v1].add(**new** Edge<E>(v1, v2, value));

numEdges++;

}

}  
  
 **public** E getEdge(**int** v1, **int** v2) {

checkVertices(v1, v2);

Edge<E> edge = getEdgeObject(v1, v2);

**if** (edge != **null**) {

// edge is in the graph.

**return** edge.edgeObject;

} **else** {

// edge not in the graph.

**return** **null**;

}

}

b. removeEdge. The No3bTests class contains tests that you can use to check your implementation of this functionality. Note: No3bTests also repeats all of the No3aTests to ensure that the new functionality does not break any of the old functionality.

Note that removeEdge relies on the equals method defined in the Edge class.

**public** E removeEdge(**int** v1, **int** v2) {

checkVertices(v1, v2);

// Use an iterator to search the list of edges for v1

LinkedList<Edge<E>> list = edgeLists[v1];

Iterator<Edge<E>> it = list.iterator();

**while** (it.hasNext()) {

Edge<E> edge = it.next();

**if** (edge.equals(**new** Edge<E>(v1, v2, **null**))) {

// found it, so use the iterator to remove it.

it.remove();

numEdges--;

**return** edge.edgeObject;

}

}

// edge not in the list, so also not in the graph.

**return** **null**;

}

c. getNeighbors. The No3cTests class contains tests that you can use to check your implementation of this functionality.

**public** ArrayList<Integer> getNeighbors(**int** v) {

checkVertex(v);

ArrayList<Integer> neighbors = **new** ArrayList<Integer>();

// Use an iterator to get all of the edges departing v

LinkedList<Edge<E>> list = edgeLists[v];

Iterator<Edge<E>> it = list.iterator();

**while** (it.hasNext()) {

neighbors.add(it.next().endVertex);

}

**return** neighbors;

}

4. Implement the inDegree method in each of the following classes:

a. CS232DirectedAdjacencyMatrixGraph. The No4aTests class contains tests that you can use to check your implementation of this functionality.

**public** **int** inDegree(**int** v) {

checkVertex(v);

**int** in = 0;

/\*

\* Non-null entries in each row, nv, of the adjacency matrix indicate

\* the edges leaving vertex nv and going to the vertex corresponding to

\* the column. So every non-null entry in column v of the adj-matrix

\* indicates an edge coming into vertex v from vertex nv. So the

\* in degree can be computed by count up the non-null entries in column v

\* of the adjacency matrix.

\*/

**for** (**int** nv = 0; nv < numVertices(); nv++) {

**if** (edges[nv][v] != **null**) {

in++;

}

}

**return** in;

}

b. CS232DirectedAdjacencyListGraph. The No4bTests class contains tests that you can use to check your implementation of this functionality.

**public** **int** inDegree(**int** v) {

checkVertex(v);

**int** in = 0;

/\*

\* For every vertex, vi, go through the list of edges leaving vi looking

\* for an edge going to vertex v...

\*/

**for** (**int** vi = 0; vi < numVertices(); vi++) {

LinkedList<Edge<E>> edgesLeavingVi = edgeLists[vi];

Iterator<Edge<E>> it = edgesLeavingVi.iterator();

**boolean** foundEdgeToV = **false**;

**while** (it.hasNext() && !foundEdgeToV) {

**if** (it.next().endVertex == v) {

in++;

/\*

\* Can only have a single edge from vi to v so once we find

\* one, there is no need to go though the rest of the edges

\* leaving vi.

\*/

foundEdgeToV = **true**;

}

}

}

**return** in;

}

c. Give the asymptotic bounds for each of these implementations.

For the implementations given:

a. Θ(|V|) – The adjacency matrix is a |V|x|V| matrix and we examine every row, nv, in column v to determine if an edge exists from nv to v.

b. Ω(|V|), O(|V|+|E|) – The adjacency list has |V| linked lists and each list is searched to see if it contains an edge from vertex vi to vertex v. If the edge from vi to v is the first one in each list, or each list contains O(1) edges (e.g. empty) we get the lower bound of Ω(|V|). If however, we have to search every element in every list (e.g. there are no edges to v) then we get the upper bound of O(|V|+|E|).

Note: If finding the in-degree is a very common operation in our application we might make a time-space tradeoff allowing us to get the in degree of a vertex in Θ(1) time in both implementations. How might we do that?

5. Create a class named CS232UndirectedAdjacencyMatrixGraph and complete its implementation. Complete this class with as little code as possible – it should require very little code. Hint: Use inheritance, override the necessary methods and use the super-class methods. The No5Tests class contains tests that you can use to check your implementation of this functionality.

**public** **class** CS232UndirectedAdjacencyMatrixGraph<V, E> **extends**

CS232DirectedAdjacencyMatrixGraph<V, E> {

**public** CS232UndirectedAdjacencyMatrixGraph(**int** numVertices) {

/\*

\* Just use the super class constructor to create the matrix.

\*/

**super**(numVertices);

}

**public** **void** addEdge(**int** v1, **int** v2, E value) {

/\*

\* Add the edge from v1 to v2 using the super-class method. Note that

\* this adjusts the size if necessary.

\*/

**super**.addEdge(v1, v2, value);

/\*

\* Now add the edge from v2 to v1. Because we always add the edge in

\* both directions that makes this equivalent to an undirected graph.

\*/

edges[v2][v1] = value;

}

**public** E removeEdge(**int** v1, **int** v2) {

/\*

\* Remove the edge from v1 to v2 using the super-class method. Note that

\* this adjusts the size if necessary.

\*/

E tmp = **super**.removeEdge(v1, v2);

/\*

\* Remove the corresponding edge from v2 to v1.

\*/

edges[v2][v1] = **null**;

**return** tmp;

}

}

6. Consider the Graph ADT implementation for undirected graphs in problem #5. This implementation trades off simple code in favor of wasting space.

a. Explain why there is wasted space in this implementation.

Space is wasted because both the edge from v1 to v2 and the edge from v2 to v1 is stored. Because the graph is undirected it would be sufficient to store just one of these edges.

b. Explain how the wasted space could be eliminated.

We could use a ragged (triangular) array with an array of i edge (E) objects for vertex i (e.g. 0 for v0, 1 for v1, 15 for v15, etc). Then when working with an edge (v1,v2) we always look in the row for the larger of v1, v2. That is, the edge (3, 5) will be stored in the column 3 of row 5, as will be edge (5,3) since it is the same edge. Note: this eliminates more than ½ of the space used by the adjacency matrix.

c. Describe how this would complicate the code.

This complicates the code in several ways:

* Getting, adding and removing edges now requires that the larger vertex be found and used as the row index.
* Getting the neighbors of a vertex v, now requires traversing both row v and column v (for all rows greater than v). Traversing the row gives all edges to vertices with indices smaller than v. Traversing the column will give all edges to vertices with indices larger than v. This complication similarly applies to computing the in degree or out degree of a vertex as they are simply the number of neighbors in an undirected graph.

7. For the directed graph from question #1:

a. List the vertices in the order in which they would be printed by a DFS starting at vertex #1 (assume adjacent vertices are considered in numeric order).

*1 2 3 5 4 9 6 7 8*

NOTE: The DFS method contains a for loop that ensures that all vertices are visited by invoking the helper method on each separate component of the graph.

b. List the vertices in the order in which they would be printed in a BFS starting at vertex #1 (assume adjacent vertices are considered in numeric order).

*1 2 4 3 9 5 6 7 8*

NOTE: The DFS method contains a for loop that ensures that all vertices are visited by invoking the helper method on each separate component of the graph.

8. Complete the BFS (breadth-first search) method in the CS232GraphAlgorithms class. The No8Tests class contains tests that you can use to check your implementation of this functionality.

/\*

\* Do a breadth-first search/traversal of the graph printing out the object

\* associated with each vertex as it is visited.

\*/

**public** **static** **void** BFS(CS232Graph<?, ?> g) {

// mark all vertices as unvisited.

**for** (**int** v = 0; v < g.numVertices(); v++) {

g.setVertexMark(v, CS232Graph.*UNVISITED*);

}

/\*

\* Go through all of the vertices and do a BFS starting at any that are

\* unvisited. This this is necessary to ensure that all connected

\* components are searched. This seems at first to be inefficient.

\* However, note that when a component is searched by BFSComponent all

\* of the vertices in that component will be marked as visited. Thus,

\* BFSComponent will be invoked on each component only once.

\*/

**for** (**int** v = 0; v < g.numVertices(); v++) {

**if** (g.getVertexMark(v) == CS232Graph.*UNVISITED*) {

*BFSComponent*(g, v);

}

}

}

/\*

\* Helper method for BFS that does a BFS of the connected component

\* containing v.

\*/

**private** **static** **void** BFSComponent(CS232Graph<?, ?> g, **int** v) {

Queue<Integer> q = **new** LinkedList<Integer>();

/\*

\* Put the first vertex into the queue and mark it as VISITED. NOTE: We

\* mark vertices as visited when they go into the queue. This is a bit

\* premature, but we know that they will be visited on removal and this

\* ensure that each vertex is inserted into the queue only once.

\*/

g.setVertexMark(v, CS232Graph.*VISITED*);

q.add(v);

**while** (!q.isEmpty()) {

// Get the vertex at the head of the queue.

**int** curV = q.poll();

// Do what needs to be done at vertex.

System.*out*.println(g.getVertexObject(curV));

// Add each unvisited neighbor of v to the queue.

ArrayList<Integer> neighbors = g.getNeighbors(curV);

**for** (**int** n : neighbors) {

**if** (g.getVertexMark(n) == CS232Graph.*UNVISITED*) {

g.setVertexMark(n, CS232Graph.*VISITED*);

q.add(n);

}

}

}

}

9. Some applications in which a graph data structure may be used will typically yield low-density graphs while others may yield high-density graphs. Discuss how the expected density of the graphs in a given application might affect whether you use an adjacency matrix or an adjacency list representation.

There is a time/space tradeoff between the adjacency matrix and adjacency list implementations of the Graph ADT. With an adjacency matrix space is allocated for all possible edges, but each edge can be accessed very quickly via array accesses. If a graph is sparse then much of the space allocated for the edges will be wasted space. With an adjacency list space is only allocated for those edges that actually exist. This can result in a significant space savings for sparse graphs. However, because the edges do not appear at a fixed index in the lists, accessing a particular edge is slower because it requires a traversal of the list.

10. Discuss how Java abstract classes are similar to and different than interfaces and when they are useful.

A java abstract class is similar to an interface in that it can define, without implementing, specific methods, including name, return type and parameter list, that must be implemented by any class that extends the abstract class. However, unlike an interface, an abstract class may also provide fields, constructors and method implementations for some methods. Any fields, constructors or methods implemented by the abstract class are inherited by any class that extends the abstract class, just like any other time inheritance is used.

Abstract classes are most useful when it is anticipated that all sub-classes require some common functionality (i.e. the implementation will would be the same in all sub-classes), but also have some methods that must have class-specific implementations. By writing the code for the common functionality in the abstract class, all sub-classes can inherit that implementation rather than having to re-implement it if an interface was used. By declaring the methods that have class-specific implementations abstract (i.e. no implementation is given), the abstract class guarantees that all sub-classes will have some implementation of the methods, in the same way that interfaces do, which allows for polymorphic behavior.

The CS232AbstractGraph class is an prototypical example of an abstract class. It provides implementations of the functionality that is the same in both the CS232AdjacencyMatrixGraph and CS232AdjacencyListGraph classes, but leaves abstract the methods that are unique to each implementation. The concrete sub-classes (CS232AdjacencyMatrixGraph and CS232A`djacencyListGraph) then provide implementations for those methods that rely on the specific representation being used.