Nombre y Apellido: ..... N°Legajo: .....

Segundo Parcial de Estructuras de Datos y Algoritmos

Segundo Cuatrimestre de 2011 – 15/11/2011

Ejercicio 1	*Ejercicio 2*	*Ejercicio 3*	*Nota*

Condición Mínima de Aprobación: Tener por lo menos dos ejercicios con B-

Ejercicio 1

Se cuenta con la siguiente implementación parcial de un grafo no dirigido:

```
public class Graph<V> {

	private HashMap<V, Node> nodes = new HashMap<V, Node>();
	private List<Node> nodeList = new ArrayList<Node>();

	public void addVertex(V vertex) {
		if (!nodes.containsKey(vertex)) {
			Node node = new Node(vertex);
			nodes.put(vertex, node);
			nodeList.add(node);
		}
	}

	public void addArc(V v, V w) {
		Node origin = nodes.get(v);
		Node dest = nodes.get(w);
		if (origin != null && dest != null && !origin.equals(dest)) {
			for (Arc arc : origin.adj) {
				if (arc.neighbor.info.equals(w)) {
					return;
				}
			}
			origin.adj.add(new Arc(dest));
			dest.adj.add(new Arc(origin));
		}
	}

	private class Node {
		V info;
		boolean visited = false;
		int tag = 0;
		List<Arc> adj = new ArrayList<Arc>();

		public Node(V info) {
			this.info = info;
		}
		public int hashCode() {
			return info.hashCode();
		}
		public boolean equals(Object obj) {
			if (obj == null || (obj.getClass() != getClass())) {
				return false;
			}
			return info.equals(((Node)obj).info);
		}
	}
	private class Arc {
		Node neighbor;

		public Arc(Node neighbor) {
			this.neighbor = neighbor;
		}
	}
}
```

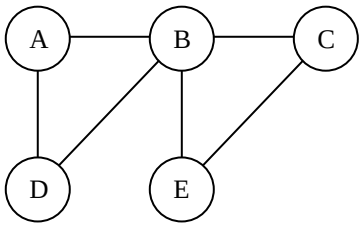
Un ciclo euleriano en un grafo es un ciclo que cubre todas las aristas del grafo, pasando una única vez por cada una.

El siguiente algoritmo encuentra un ciclo euleriano en un grafo **conexo**:

- Para que exista un ciclo euleriano, todos los nodos deben tener grado par.
- Tomar un vértice *v*, y construir cualquier camino hasta volver a *v*, sin repetir aristas (el grado par de los nodos garantiza que esto sea posible). El ciclo formado puede no cubrir todas las aristas del grafo.
- Mientras exista un vértice *v* que pertenezca al ciclo actual y que tenga aristas que no forman parte del ciclo, comenzar otro ciclo desde *v* siguiendo aristas no utilizadas hasta volver a *v*, y unir este ciclo con el que ya se tiene formado.

Agregar a la implementación de grafos un método que encuentre un ciclo euleriano en un grafo conexo, utilizando el algoritmo anterior. Debe retornar una lista con los nodos que conforman el ciclo, o *null* si el grafo no admite un ciclo euleriano.

Ejemplo: para el siguiente grafo, una posible salida del algoritmo es la lista {C, E, B, D, A, B, C}.



Ejercicio 2

Se desea usar un grafo para implementar una red, donde cada nodo representa un router, y las aristas distancias. Cuando un router recibe un paquete cuyo destino es otro router DEST, debe enviarlo por el camino más corto; para lo cual tiene que tener asociado el router GATEWAY, que es el vecino por el cual comienza el camino más corto hacia DEST. Se pide:

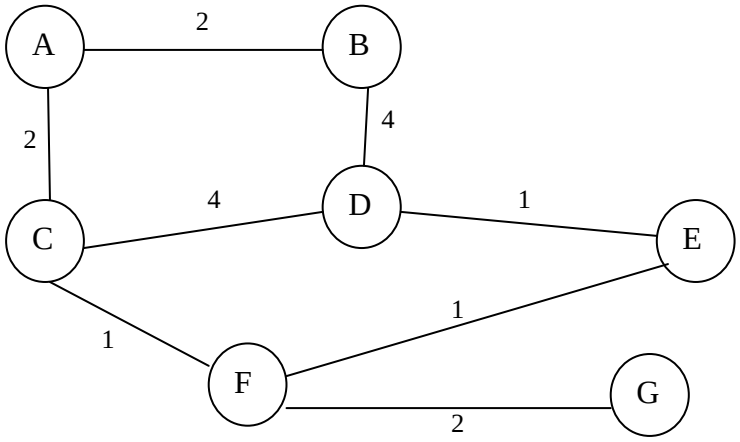
a) Modificar la implementación de grafos del ejercicio anterior para que admita pesos en las aristas (valores enteros positivos).
b) Agregar el siguiente método:

```
public HashSet<V, V> route(V nodeOrigin);
```

que arme la tabla de ruteo para un nodo determinado.

Ejemplo:

para el siguiente grafo, y nodo A debe armar la siguiente tabla



A	null
B	B
C	C
D	C
E	C
F	C
G	C

Ejercicio 3

El Boggle es un juego de tablero, formado por 16 dados con letras en sus caras, que son mezclados y ubicados en una grilla de 4 x 4, como se muestra a continuación:

U	N	D	S
M	H	O	R
T	A	L	A
E	R	I	N

El objetivo del juego es encontrar la mayor cantidad posible de palabras formadas por letras adyacentes, comenzando desde cualquier posición y recorriendo los dados en forma horizontal y vertical.

Las palabras deben tener como mínimo 4 letras. Las letras que forman cada palabra deben estar ubicadas en dados adyacentes (horizontal o vertical). Cada dado puede ser utilizado una única vez en cada palabra.

Implementar un algoritmo que dado un tablero y un diccionario de palabras posibles retorne todas las palabras que se pueden formar:

```
public static Set<String> solve(char[][] board, Trie dictionary)
```

Ejemplo:

Si se tiene un diccionario formado por el siguiente conjunto de palabras:
{CASA, HOLA, MUNDO, DOLAR, PERRO, HARINA, RANA}

Entonces el algoritmo retornaría el conjunto:
{HOLA, MUNDO, DOLAR, HARINA}

Para representar al diccionario se utiliza la siguiente implementación parcial de un trie:

```
public class Trie {

	private Trie[] children = new Trie['Z' - 'A' + 1];
	private boolean end;

	public void add(String word) {
		word = word.toUpperCase();
		add(word.toCharArray(), 0);
	}

	private void add(char[] word, int from) {
		if (from == word.length) {
			end = true;
		} else {
			int index = word[from] - 'A';
			if (children[index] == null) {
				children[index] = new Trie();
			}
			children[index].add(word, from + 1);
		}
	}

	public Trie getChild(char c) {
		return children[c - 'A'];
	}

	public boolean isEnd() {
		return end;
	}
}
```