Teoría de algoritmos 1 – 75.29

Trabajo Práctico Nº: 1

Integrantes:		
Padrón	**Nombre y Apellido**	**Email**
83.989	Alejo Vinjoy	avinjoy@gmail.com
84.623	Santiago Nicolás Risaro Sesar	nrisaro@gmail.com

Para uso de la cátedra
Primera entrega
Corrector
Observaciones
Segunda entrega
Corrector
Observaciones

Universidad de Buenos AiresFacultad de IngenieríaTeoría de Algoritmos (75.29)

Contenido

Enunciado

75.29 Teoría de Algoritmos I

Trabajo Práctico Nº 1

Robustez en grafos

Fecha de entrega: 16 de octubre de 2013

Definición: Sea el grado de robustez de un grafo la cantidad mínima de aristas que es necesario remover del grafo para que el mismo sea no conexo.

Desarrollar un algoritmo que dado un grafo y un grado de robustez, enumere todas las aristas que sería necesario agregar al grafo en cuestión, para que el mismo alcance el grado de robustez especificado. Dicho algoritmo debe ser lo más óptimo posible.

Implementar el algoritmo en una aplicación que tome como entrada dos parámetros: el grado de robustez y el nombre del archivo con la definición del grafo.
El archivo con la definición del grafo debe respetar el siguiente formato:

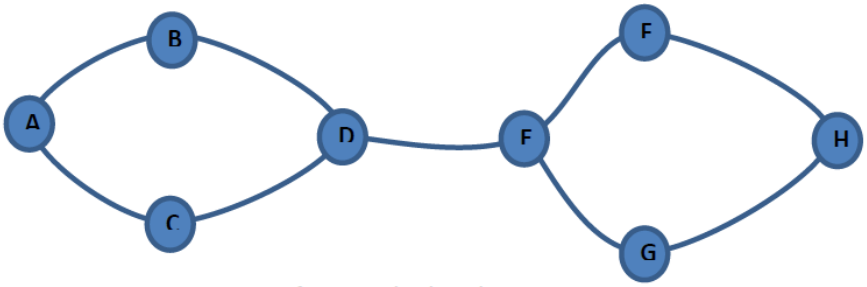
<vértice>: <vértice adyacente 1>, <vértice adyacente 2>

Ejemplo de invocación de la aplicación:
tdatp1 3 grafo1.txt

En cuanto a la salida de la aplicación, la misma consistirá en listar por pantalla las aristas a agregar
para alcanzar el grado de robustez requerido. El formato de salida es el siguiente:

<arista #>: <vértice origen>, <vértice destino>

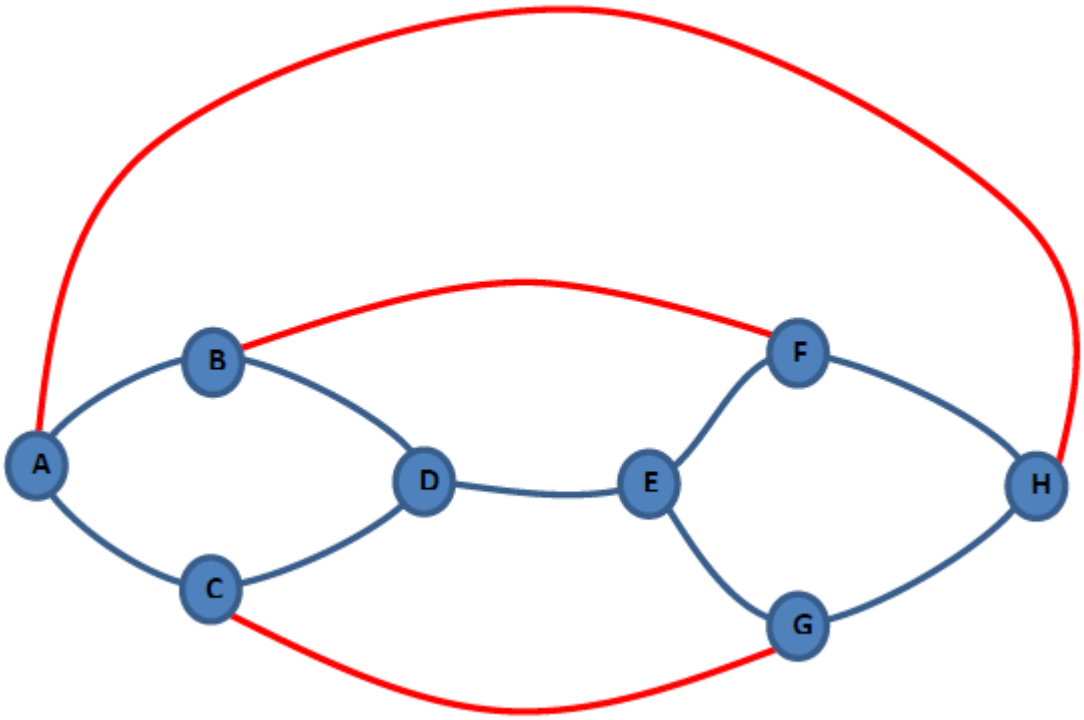
Ejemplo:



Grafo 1: grado de robustez 1

***grafo1*.txt**
A: B, C
B: A, D
C: A, D
D: B, C, E
E: D, F, G
F: E, H
G: E, H
H: F, G

Grafo 1 modificado: grado de robustez 3



Grafo 1 modificado: grado de robustez 3

Salida
arista1: A, H
arista2: B, F
arista3: C, G

Estrategia de Resolución

La estrategia de resolución gira en torno a dos grandes partes o módulos:
Módulo deCarga: Aquí se realiza la carga desde el archivo de entrada y se crea el grafo que se va a trabajar.
Clases Principales Involucradas: CargadorDeGrafos, Arista, Grafo, ListaEnlazada.

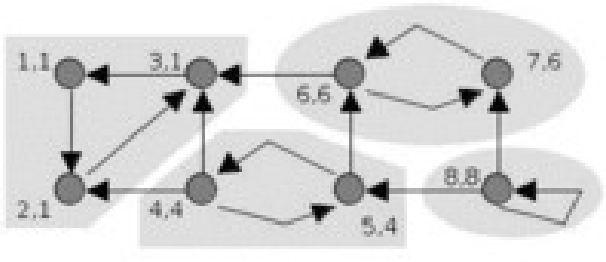
Módulo de resolución: Realiza todas las iteraciones por el grafo, encuentra los ciclos y aumenta si corresponde la robustez.
Clases Principales Involucradas: Grafo, ListaEnlazada, Vector, AumentadorDeRobustez.

El algoritmo utilizado para resolver este trabajo práctico es el siguiente (pseudocódigo):

```
Leer el archivo de entrada
Cargar el grafo
Ubicar los ciclos (Utilizando un algoritmo basado en DFS para
obtenerlos)
Si la robustez pedida es compatible con el grafo entonces
     cicloUno = obtenerCiclo(1)
     cicloDos = obtenerCiclo(2)
      Mientras haya vertices en ciclos y robustez <> robPedida
           Crear arcos entre vértices de los ciclos
     Fin Mientras
     Robustez++
Fin Si
```

Análisis de Orden

Algoritmo: Obtener los ciclos de un grafo (Algoritmo de Tarjan)
Utilizado en: Clase Grafo, método encontrarCiclos(Vertice).
Análisis: El método para encontrar ciclos en los grafos se basa en el algoritmo de Tarjan (aunque no es el mismo) (http://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm)
La idea básica de este algoritmo es la siguiente: Se parte de algún node del grafo y se realiza un recorrido DFS, marcándolo con un índice y para mantener una identificación de la rama desde donde ha venido se utiliza un indicador llamado lowLink. Como cualquier recorrido DFS, cada nodo se visita sólo una vez , descartando la revisita del nodo ya explorado. Cada vez que se encuentre un ciclo cerrado se guardará en una vector los nodos visitados, cada uno de ellos tendrá su índice de visita y un lowLink que indica desde que rama se llamó.
Gráficamente para un digrafo / multigrafo:



En esta implementación el algoritmo se llama, como máximo 2 veces para cada vértice por lo que el orden sería O (2V), pero al final nos encontramos con la parte que carga los diferentes ciclos encontrados la cual itera utilizando dos ciclos por todos los vétices, siendo su peor caso **O (V^2).**

Algoritmo: Obtener los valores de entrada
Utilizado en: Clase cargadorDeGrafos, método cargar.
Análisis: Aquí básicamente se abre el archivo, y por cada línea que tengo se realiza la carga del vértice y sus adyacente, lo cual, nos puede llevar como peor caso **O (L x V),** donde L es la cantidad de líneas a cargar y V la cantidad de vértices adyacentes por líneas.

Algoritmo: Inserción en la lista
Utilizado en: Clase ListaDeAdyacencia, método agregar.
Análisis: Por tratarse de una lista doblemente enlazada cada ítem de la lista conoce su predecesor y su sucesor y la lista conoce tanto el primer elemento como el último, insertar un elemento al final de la lista es simplemente actualizar referncias para insertarlo entre el primer y último elemento, por ende la complejidad es **O (1).**

Algoritmo: Inserción Vértice en Grafo
Utilizado en: Clase Grafo, método agregar vértice.
Análisis: La inserción en la lista es **O(1)**, pero debemos chequear que no se encuentre ya insertado el vértice, para ello debemos recorrer toda la lista, por ende la complejidad es **O (n).**

Algoritmo: Inserción Arco en Grafo
Utilizado en: Clase Grafo, método agregar arco.
Análisis: Para mantener correctamente las referencias entre los objetos, antes de trazar un arco debemos buscar las instancias de los vértices afectados **O(n)** para cada vértice, y luego insertamos la arista en **O(1)**, por ende la complejidad es **O (n).**

Indicaciones para la ejecución

Se asume que en la PC donde va a ejecutarse el programa se encuentra instalada y correctamente configurada una versión del JRE. Este trabajo práctico fue desarrollado utilizando la versión 1.6 de java y testeado con el JRE oficial, provisto por Oracle. Si bien no fue probado con otra configuración, el trabajo debería ejecutarse sin problemas con máquinas virtuales versión 1.7 y/o no oficiales. Debido a los cambios en la API entre la versión 1.5 y 1.6 de java no hay garantías de que pueda ejecutarse correctamente con versiones anteriores a 1.6.

Para la compilación se utilizó Maven 3.

Se asume que existe una carpeta $HOME/tda y que en ella se han descargado los archivos a utilizar.

Ejecución

1. Posicionarse en el directorio donde se encuentra el archivo tdatp1.jar

   cd $HOME/tda

2. Ejecutar el programa

   java -jar tdatp1.jar <ROBUSTEZ> [<ARCHIVO>]

<ROBUSTEZ> indica la robustez a la que se quiere llevar el grafo.

<ARCHIVO> es un parámetro opcional con la ruta del archivo que contiene la definición del grafo. Si no se provee este parámetro el programa buscará un archivo llamado “grafo.txt” en la carpeta $HOME/tda

Compilación

1. Posicionarse en la carpeta donde se descargará el código fuente

   cd $HOME/tda

2. Descargar una copia del código fuente

   svn checkout http://teoria-de-algoritmos-2013-2.googlecode.com/svn/tags/Entrega_1_2013_10_16/TeoriaDeAlgoritmos

3. Poscionarse en la carpeta recién descargada

   cd TeoriaDeAlgoritmos

4. Ejecutar maven

   mvn clean package

5. El ejecutable se encuentra en la carpeta target

Conclusiones

Luego de implementar los algoritmos presentes en este informe que se discutieron en clase, podemos ver que el la práctica influye mucho la implementación utilizada contra la teoría. A futuro para una mejor performance, se podría crear el grafo con un vector de listas de adyacencia en vez de listas de listas de adyacencia. Esto redundará en una menor complejidad al insertar un arco en el grafo.
Hemos puesto en práctica el backtracking utilizando un algoritmo basado en DFS para lograr encontrar los ciclos en los grafos.

Código Fuente

El código fuente completo puede ser accedido en http://code.google.com/p/teoria-de-algoritmos-2013-2/source/browse/#svn%2Ftags%2FEntrega_1_2013_10_16%2FTeoriaDeAlgoritmos

Arista.java

```
package ar.fi.uba.tda.colecciones;

public class Arista {

	private Vertice<?> origen;
	private Vertice<?> destino;


	public Arista(Vertice<?> verticeCicloUno, Vertice<?> verticeCicloDos) {
		this.origen = verticeCicloUno;
		this.destino = verticeCicloDos;
	}

	public Vertice<?> getOrigen() {
		return origen;
	}

	public Vertice<?> getDestino() {
		return destino;
	}

	@Override
	public String toString() {
		return origen.getContenido() + ", " + destino.getContenido();
	}
}
```

Robustez.java

```
package ar.fi.uba.tda;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.Iterator;

import ar.fi.uba.tda.colecciones.Arista;
import ar.fi.uba.tda.colecciones.Grafo;
import ar.fi.uba.tda.colecciones.ListaEnlazada;
import ar.fi.uba.tda.colecciones.Vertice;
import ar.fi.uba.tda.util.AumentadorDeRobustez;
import ar.fi.uba.tda.util.CargadorDeGrafos;

public class Robustez {

	private final CargadorDeGrafos cargador;
	private final AumentadorDeRobustez aumentador;

	@SuppressWarnings("rawtypes")
	private final Grafo grafo;
```

```
	public Robustez(Grafo<String> grafo, CargadorDeGrafos cargador,
AumentadorDeRobustez aumentador) {
		this.grafo = grafo;
		this.cargador = cargador;
		this.aumentador = aumentador;
	}

	@SuppressWarnings({ "unchecked", "rawtypes" })
	public static void main(String[] args) throws IOException {

		if (args.length > 0) {

			int robustezDeseada = Integer.valueOf(args[0]);
			BufferedReader archivo = leerArchivo(args);

			Grafo grafo = new Grafo();
			CargadorDeGrafos cargador = new CargadorDeGrafos(grafo);
			AumentadorDeRobustez aumentador = new AumentadorDeRobustez(grafo);

			new Robustez(grafo, cargador,
aumentador).ejecutar(robustezDeseada, archivo);

		} else {

			System.err.println("Se debe ingresar el grado de robustez y el
nombre de archivo");
			System.err.println("O solo el grado de robustez");
			System.err.println("i.e.: java -jar Robustez.jar 3");
		}
	}

	private static BufferedReader leerArchivo(String[] args) throws
FileNotFoundException {

		String rutaArchivo = "grafo.txt";

		if (args.length > 1) {
			rutaArchivo = args[1];
		}

		return new BufferedReader(new FileReader(rutaArchivo));
	}

	@SuppressWarnings({ "unchecked", "rawtypes" })
	public void ejecutar(int robustezDeseada, BufferedReader archivo) throws
IOException {

		cargador.cargar(archivo);

		grafo.encontrarCiclos((Vertice) grafo.getVertices().primero());
		aumentador.aumentar(grafo.getCiclosGrafo(), robustezDeseada);

		ListaEnlazada<Arista> aristas = aumentador.getAristasAgregadas();
		int numeroArista = 0;

		Iterator<Arista> itAristas = aristas.iterador();

		while (itAristas.hasNext()) {

			numeroArista++;
			Arista arista = itAristas.next();
```

```
                        System.out.println("Arista " + numeroArista + ": " + arista);

                }

                //System.out.println("FIN");

        }

}
```

Grafo.java

```
package ar.fi.uba.tda.colecciones;

import java.util.Iterator;
import java.util.Vector;

public class Grafo<T> {

        private ListaEnlazada<Vertice<T>> vertices;
        private ListaEnlazada<Vertice<T>> recorridoDFS;
        private ListaEnlazada<Vertice<T>> recorridoBFS;
        private ListaEnlazada<ListaEnlazada<Vertice<T>>> ciclosGrafo;
        ListaEnlazada<Vertice<T>> subset = new ListaEnlazada<Vertice<T>>();
        private Vector<Vertice<T>> visitados;
        private Long index;

        public Grafo(ListaEnlazada<Vertice<T>> vertices) {
                super();
                this.vertices = vertices;
        }

        public Grafo() {
                this.vertices = new ListaEnlazada<Vertice<T>>();
                this.recorridoDFS = new ListaEnlazada<Vertice<T>>();
                this.recorridoBFS = new ListaEnlazada<Vertice<T>>();
                this.ciclosGrafo = new ListaEnlazada<ListaEnlazada<Vertice<T>>>();
                this.visitados = new Vector<Vertice<T>>();
                this.index = 0L;
        }

        public ListaEnlazada<Vertice<T>> getVertices() {
                return vertices;
        }

        public void setVertices(ListaEnlazada<Vertice<T>> vertices) {
                this.vertices = vertices;
        }

        public Integer getCantidadDeNodosGrafo() {
                return this.vertices.tamanio();
        }

        public ListaEnlazada<Vertice<T>> getRecorridoDFS() {
                return recorridoDFS;
        }

        public void setRecorridoDFS(ListaEnlazada<Vertice<T>> recorridoDFS) {
                this.recorridoDFS = recorridoDFS;
        }

        public ListaEnlazada<Vertice<T>> getRecorridoBFS() {
                return recorridoBFS;
```

```
	}

	public void setRecorridoBFS(ListaEnlazada<Vertice<T>> recorridoBFS) {
		this.recorridoBFS = recorridoBFS;
	}

	public ListaEnlazada<ListaEnlazada<Vertice<T>>> getCiclosGrafo() {
		return ciclosGrafo;
	}

	public void setCiclosGrafo(
				ListaEnlazada<ListaEnlazada<Vertice<T>>> ciclosGrafo) {
		this.ciclosGrafo = ciclosGrafo;
	}

	/**
	 * Agrega un vï¿½rtice al grafo
	 *
	 * @param vert
	 */
	public void agregarVertice(Vertice<T> vert) {
		if (vert != null && vert.getContenido() != null
					&& !this.contieneVertice(vert)) {
				this.vertices.agregar(vert);
			}

	}

	/**
	 * Crea un arco entre 2 vï¿½rtices (no es grafo dirigido)
	 *
	 * @param inicio
	 * @param fin
	 */
	public void agregarArco(Vertice<T> inicio, Vertice<T> fin) {
		Vertice<T> inicioEnGrafo = this.obtener(inicio);

		if (inicioEnGrafo != null) {
			inicio = inicioEnGrafo;
		}

		Vertice<T> finEnGrafo = this.obtener(fin);

		if (finEnGrafo != null) {
			fin = finEnGrafo;
		}

		inicio.getAdyacentes().agregar(fin);
		fin.getAdyacentes().agregar(inicio);

		this.agregarVertice(inicio);
		this.agregarVertice(fin);

	}

	private Vertice<T> obtener(Vertice<T> buscado) {

		return this.vertices.obtener(buscado);
	}

	public boolean contieneVertice(Vertice<T> verticeBuscado) {

		return vertices.contiene(verticeBuscado);
```

```
	}

	/**
	 * Recorrido en profundidad
	 */
	public void recorridoDFS(ListaEnlazada<Vertice<T>> vertices) {

		Iterator<Vertice<T>> iterador = vertices.iterador();

		while (iterador.hasNext()) {
			Vertice<T> vert = iterador.next();
			if (!vert.isVisitado()) {
				vert.setVisitado(true);
				// System.out.println(vert);
				getRecorridoDFS().agregar(vert);
				recorridoDFS(vert.getAdyacentes());
			}
		}
		return;
	}

	/**
	 * Recorrido en ancho
	 */
	public void recorridoBFS(ListaEnlazada<Vertice<T>> vertices) {

		Iterator<Vertice<T>> iterador = vertices.iterador();

		while (iterador.hasNext()) {
			Vertice<T> vert = iterador.next();

			if (!vert.isVisitado()) {
				vert.setVisitado(true);
				// System.out.println(vert);
				getRecorridoBFS().agregar(vert);
			}

			Iterator<Vertice<T>> iteVertice = vert.getAdyacentes().iterador();
			while (iteVertice.hasNext()) {
				Vertice<T> vertAdy = iteVertice.next();
				if (!vertAdy.isVisitado()) {
					vertAdy.setVisitado(true);
					// System.out.println(vertAdy);
					getRecorridoBFS().agregar(vertAdy);
				}
			}

		}

		return;
	}

	/**
	 * Encuentra los ciclos en el grafo.
	 * Utilizado para determinar sobre que vertices crear los arcos para
	 * la robustez.
	 */

	public void encontrarCiclos(Vertice<T> vert) {

		if (!vert.isVisitado()) {
			vert.setVisitado(true);
			vert.setIndex(index);
```

```
                    vert.setLowLink(index);
                    index++;
                    visitados.add(vert);
                    // System.out.println(vert);

                    Iterator<Vertice<T>> iterAdyacente = vert.getAdyacentes()
                                .iterador();

                    while (iterAdyacente.hasNext()) {
                          Vertice<T> vertAdyacente = iterAdyacente.next();

                          if (!vertAdyacente.isVisitado()) {
                                encontrarCiclos(vertAdyacente);
                                vert.setLowLink(Math.min(vert.getLowLink(),
                                            vertAdyacente.getLowLink()));
                          } else if (visitados.contains(vertAdyacente)) {
                                vert.setLowLink(Math.min(vert.getLowLink(),
                                            vertAdyacente.getIndex()));
                          }
                    }
             }

             if (vert.getLowLink() == vert.getIndex()) { // Es el primero volviendo de
la recursion
                    int count =0;
                    Vertice <T> verticeAux=vert;
                    while (count < visitados.size()) {
                          for (Vertice<T> ver : visitados) {
                                if (ver.getLowLink() == verticeAux.getIndex()
                                            || ver.getIndex() ==
verticeAux.getIndex()) {
                                      subset.agregar(ver);
                                } else {
                                      verticeAux = ver;
                                      ciclosGrafo.agregar(subset);
                                      subset=new ListaEnlazada<Vertice<T>>();
                                      subset.agregar(ver);
                                }
                                count++;
                          }
                          ciclosGrafo.agregar(subset);

                    }
                    return;
             }
       }

}
```

ListaEnlazada.java

```
package ar.fi.uba.tda.colecciones;

import java.util.Iterator;
import java.util.ListIterator;

public class ListaEnlazada<T> {

       private Elemento header = new Elemento(null, null, null);
       private Integer tamanio = 0;

       public ListaEnlazada () {
```

```
		header.siguiente = header.anterior = header;
	}

	public Boolean vacia() {
		return tamanio == 0;
	}

	public T primero() {
		return header.siguiente.elemento;
	}

	public T ultimo() {
		return header.anterior.elemento;
	}

	public void agregar(T elemento) {
		agregarAntes(elemento, header);
	}

	private void agregarAntes(T elemento, Elemento siguiente) {

		Elemento nuevo = new Elemento(elemento, siguiente, siguiente.anterior);
		nuevo.anterior.siguiente = nuevo;
		nuevo.siguiente.anterior = nuevo;
		tamanio++;
	}

	public Integer tamanio() {
		return tamanio;
	}

	public ListIterator<T> iterador() {
		return new IteradorListaEnlazada();
	}

	private class IteradorListaEnlazada implements ListIterator<T> {

		private int siguiente;
		private Elemento siguienteElemento;

		public IteradorListaEnlazada () {

			siguiente = 0;
			siguienteElemento = header;
		}

		@Override
		public boolean hasNext() {
			return siguiente != tamanio;
		}

		@Override
		public T next() {

			T aRetornar = null;

			if (siguiente < tamanio) {

				aRetornar = siguienteElemento.siguiente.elemento;
				siguienteElemento = siguienteElemento.siguiente;
				siguiente++;
			}
```

```
			return aRetornar;
		}

		@Override
		public boolean hasPrevious() {
			return siguiente != 0;
		}

		@Override
		public T previous() {

			T aRetornar = null;

			if (siguiente > 0) {

				aRetornar = siguienteElemento.anterior.elemento;
				siguienteElemento = siguienteElemento.anterior;
				siguiente--;
			}

			return aRetornar;
		}

		@Override
		public void remove() {

		}

		@Override
		public int nextIndex() {
			return 0;
		}

		@Override
		public int previousIndex() {
			return 0;
		}

		@Override
		public void set(T e) {
		}

		@Override
		public void add(T e) {
		}

	}

	private class Elemento {

		T elemento;
		Elemento anterior;
		Elemento siguiente;

		public Elemento(T elemento, Elemento siguiente, Elemento anterior) {

			this.elemento = elemento;
			this.anterior = anterior;
			this.siguiente = siguiente;
		}

	}
```

```
	public boolean contiene(T datoBuscado) {

		return obtener(datoBuscado) != null;
	}

	public T obtener(T buscado) {

		T encontrado = null;
		Iterator<T> iterador = this.iterador();

		while (encontrado == null && iterador.hasNext()) {

			T item = iterador.next();

			if (item.equals(buscado)) {
				encontrado = item;
			}
		}

		return encontrado;
	}

}
```

Vertice.java

```
package ar.fi.uba.tda.colecciones;

public class Vertice<T> {

	private T contenido;
	private boolean visitado;
	private ListaEnlazada<Vertice<T>> adyacentes;
	private Long index;
	private Long lowLink;


	public Vertice() {
		super();
	}

	public Vertice(T contenido){
		super();
		this.contenido=contenido;
		this.adyacentes=new ListaEnlazada<Vertice<T>>();
	}

	public T getContenido() {
		return contenido;
	}
	public void setContenido(T contenido) {
		this.contenido = contenido;
	}
	public ListaEnlazada<Vertice<T>> getAdyacentes() {
		return adyacentes;
	}
	public void setAdyacentes(ListaEnlazada<Vertice<T>> adyacentes) {
		this.adyacentes = adyacentes;
	}

	public Long getIndex() {
		return index;
	}
	}
```

```
	public void setIndex(Long index) {
		this.index = index;
	}

	public Long getLowLink() {
		return lowLink;
	}

	public void setLowLink(Long lowLink) {
		this.lowLink = lowLink;
	}

	public boolean isVisitado() {
		return visitado;
	}

	public void setVisitado(boolean vistado) {
		this.visitado = vistado;
	}

	public Integer getGradoVertice(){
		return this.adyacentes.tamanio();
	}

	@Override
	public int hashCode() {
		final int prime = 31;
		int result = 1;
		result = prime * result
				+ ((contenido == null) ? 0 : contenido.hashCode());
		return result;
	}

	@Override
	@SuppressWarnings("unchecked")
	public boolean equals(Object obj) {
		if (this == obj) {
			return true;
		}
		if (obj == null) {
			return false;
		}
		if (!(obj instanceof Vertice)) {
			return false;
		}
		Vertice<T> other = (Vertice<T>) obj;
		if (contenido == null) {
			if (other.contenido != null) {
				return false;
			}
		} else if (!contenido.equals(other.contenido)) {
			return false;
		}
		return true;
	}

	@Override
	public String toString() {
		return "Vertice: "+contenido.toString()+" (i: "+this.getIndex()+" , l:
"+this.getLowLink()+")";
	}

}
```

AumentadorRobustez.java

```
package ar.fi.uba.tda.util;

import java.util.Iterator;
import java.util.ListIterator;

import ar.fi.uba.tda.colecciones.Arista;
import ar.fi.uba.tda.colecciones.Grafo;
import ar.fi.uba.tda.colecciones.ListaEnlazada;
import ar.fi.uba.tda.colecciones.Vertice;

public class AumentadorDeRobustez {

	private final Grafo<?> grafo;
	private ListaEnlazada<Arista> aristasAgregadas = new ListaEnlazada<Arista>();

	public AumentadorDeRobustez(Grafo<?> grafo) {
		this.grafo = grafo;
	}

	@SuppressWarnings({ "unchecked", "rawtypes" })
	public void aumentar(ListaEnlazada<ListaEnlazada<Vertice>> ciclos, int robustez)
{

		if (laRobustezEsCompatibleConElGrafo(robustez)) {

			ListIterator<ListaEnlazada<Vertice>> listaDeCiclos =
ciclos.iterador();

			while (listaDeCiclos.hasNext()) {

				ListaEnlazada<Vertice> ciclo = listaDeCiclos.next();

				ListaEnlazada<Vertice> siguienteCiclo =
obtenerSiguienteCiclo(ciclos, listaDeCiclos);

				Iterator<Vertice> verticesPrimerCiclo = ciclo.iterador();
				Iterator<Vertice> verticesSegundoCiclo =
siguienteCiclo.iterador();

				int robustezAlcanzada = 0;

				while (robustezAlcanzada < robustez &&
verticesPrimerCiclo.hasNext() && verticesSegundoCiclo.hasNext()) {

					Vertice verticeCicloUno = verticesPrimerCiclo.next();
					Vertice verticeCicloDos =
verticesSegundoCiclo.next();

					if (!
verticeCicloUno.getAdyacentes().contiene(verticeCicloDos) && !
verticeCicloUno.equals(verticeCicloDos)) {
						aristasAgregadas.agregar(new
Arista(verticeCicloUno, verticeCicloDos));
					}

					robustezAlcanzada++;
				}

			}
		}
	}
```

```
	private boolean laRobustezEsCompatibleConElGrafo(int robustez) {
		return grafo.getCantidadDeNodosGrafo() > robustez;
	}

	@SuppressWarnings("rawtypes")
	private ListaEnlazada<Vertice>
obtenerSiguienteCiclo(ListaEnlazada<ListaEnlazada<Vertice>> ciclos,

		ListIterator<ListaEnlazada<Vertice>> listaDeCiclos) {

		ListaEnlazada<Vertice> siguienteCiclo;

		if (listaDeCiclos.hasNext()) {
			siguienteCiclo = listaDeCiclos.next();

			corregirIterador(listaDeCiclos);
		} else {
			siguienteCiclo = ciclos.primero();
		}

		return siguienteCiclo;
	}

	@SuppressWarnings("rawtypes")
	private void corregirIterador(
			ListIterator<ListaEnlazada<Vertice>> listaDeCiclos) {
		if (listaDeCiclos.hasNext()) {
			listaDeCiclos.previous();
		}
	}

	public ListaEnlazada<Arista> getAristasAgregadas() {
		return aristasAgregadas;
	}

}
```

CargadorDeGrafos.java

```
package ar.fi.uba.tda.util;

import java.io.BufferedReader;
import java.io.IOException;

import ar.fi.uba.tda.colecciones.Grafo;
import ar.fi.uba.tda.colecciones.Vertice;

public class CargadorDeGrafos {

	private final Grafo<String> grafo;

	public CargadorDeGrafos(Grafo<String> grafo) {
		this.grafo = grafo;

	}

	public void cargar(BufferedReader reader) throws IOException {

		String linea = reader.readLine();

		while(linea != null) {

			String[] vertices = linea.split(":");
```

```
                    String vertice = vertices[0].trim();

                    if (vertices.length > 1) {

                          String[] adyacentes = vertices[1].split(",");

                          for (int i = 0; i < adyacentes.length; i++) {

                                agregarArista(vertice, adyacentes[i].trim());
                          }

                    } else {

                          grafo.agregarVertice(new Vertice<String>(vertice));
                    }

                    linea = reader.readLine();
             }
       }

       public void agregarArista(String verticeInicial, String verticeFinal) {

             Vertice<String> inicio = new Vertice<String>(verticeInicial);

             Vertice<String> fin = new Vertice<String>(verticeFinal);

             grafo.agregarArco(inicio, fin);

       }

}
```

RobustezTest.java

```
package ar.fi.uba.tda;

import static org.mockito.Matchers.anyString;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.never;
import static org.mockito.Mockito.verify;
import static org.mockito.Mockito.when;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.PrintStream;

import org.junit.Before;
import org.junit.Test;

import ar.fi.uba.tda.colecciones.Arista;
import ar.fi.uba.tda.colecciones.Grafo;
import ar.fi.uba.tda.colecciones.ListaEnlazada;
import ar.fi.uba.tda.colecciones.Vertice;
import ar.fi.uba.tda.util.AumentadorDeRobustez;
import ar.fi.uba.tda.util.CargadorDeGrafos;

public class RobustezTest {

       private PrintStream err;

       @Before
       public void setup() {
```

```
		err = mock(PrintStream.class);
		System.setErr(err);
	}

	@Test
	public void elProgramaTomaComoEntradaLaRobustezYElNombreDeArchivo() throws
IOException {

		Robustez.main(new String[]{"3", "src/test/resources/grafo.txt"});

		verify(err, never()).println(anyString());
	}

	@Test
	public void elProgramaTomaComoEntradaLaRobustezYTomaUnNombreDeArchivoDefault()
throws IOException {

		Robustez.main(new String[]{"3"});

		verify(err, never()).println(anyString());
	}

	@Test
	public void siNoSeProveenDatosInformaDelError() throws IOException {

		Robustez.main(new String[]{});

		verify(err).println("Se debe ingresar el grado de robustez y el nombre de
archivo");
		verify(err).println("O solo el grado de robustez");
		verify(err).println("i.e.: java -jar Robustez.jar 3");
	}

	@Test
	@SuppressWarnings({ "unchecked", "rawtypes" })
	public void elProgramaCargaElGrafoBuscaCiclosYAumentaLaRobustez() throws
IOException {

		Grafo grafo = mock(Grafo.class);
		CargadorDeGrafos cargador = mock(CargadorDeGrafos.class);
		AumentadorDeRobustez aumentador = mock(AumentadorDeRobustez.class);
		BufferedReader reader = mock(BufferedReader.class);
		ListaEnlazada<Vertice<String>> listaVertices = mock(ListaEnlazada.class);
		Vertice<String> vertice = mock(Vertice.class);

		ListaEnlazada<ListaEnlazada<Vertice>> listaCiclos =
mock(ListaEnlazada.class);

		when(grafo.getVertices()).thenReturn(listaVertices);
		when(listaVertices.primero()).thenReturn(vertice);
		when(grafo.getCiclosGrafo()).thenReturn(listaCiclos);
		when(aumentador.getAristasAgregadas()).thenReturn(new
ListaEnlazada<Arista>());

		Robustez robustez = new Robustez(grafo, cargador, aumentador);

		robustez.ejecutar(3, reader);

		verify(cargador).cargar(reader);
		verify(grafo).encontrarCiclos(vertice);
		verify(grafo).getCiclosGrafo();
		verify(aumentador).aumentar(listaCiclos, 3);
```

```
	}

	@Test
	public void grafoQueNoSePuedeAumentarYPidoRobustez3() throws IOException {

		Robustez.main(new String[]{"3",
"src/test/resources/grafoImposibleDeAumentar.txt"});
		verify(err, never()).println(anyString());
	}

	@Test
	public void grafoGrandeRobustez3() throws IOException {

		Robustez.main(new String[]{"3", "src/test/resources/grafoGrande.txt"});
		verify(err, never()).println(anyString());
	}

}
```

GrafoTest.java

```
package ar.fi.uba.tda.colecciones.test;

import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.is;

import org.junit.Before;
import org.junit.Test;

import ar.fi.uba.tda.colecciones.Grafo;
import ar.fi.uba.tda.colecciones.Vertice;

public class GrafoTest {

	private Grafo<String> grafo;
	private Vertice<String> verticeUno;
	private Vertice<String> verticeDos;
	private Vertice <String> verticeA = new Vertice<String>("A");
	private Vertice <String> verticeB = new Vertice<String>("B");
	private Vertice <String> verticeC = new Vertice<String>("C");
	private Vertice <String> verticeD = new Vertice<String>("D");
	private Vertice <String> verticeE = new Vertice<String>("E");
	private Vertice <String> verticeF = new Vertice<String>("F");
	private Vertice <String> verticeG = new Vertice<String>("G");
	private Vertice <String> verticeH = new Vertice<String>("H");

	@Before
	public void setup() {

		grafo = new Grafo<String>();

		verticeUno = new Vertice<String>("VerticeUno");
		verticeDos = new Vertice<String>("VerticeDos");

	}

	@Test
	public void grafoVacioTest() {
		assertThat("El grafo estï¿½ vacï¿½o", grafo.getCantidadDeNodosGrafo(),
is(0));
	}

	@Test
	public void agregoUnVerticeTest() {
```

```
		grafo.agregarVertice(verticeUno);
		assertThat("Agrego un vertice", grafo.getCantidadDeNodosGrafo(), is(1));
	}

	//TODO:Tenemos que ver si la lista acepta repetidos
	@Test
	public void agregoUnArcoTest() {

		grafo.agregarArco(verticeUno, verticeDos);
		assertThat("Hay dos vertices", grafo.getCantidadDeNodosGrafo(), is(2));
		assertThat("Hay un arco entre Vertice Uno y Dos",
grafo.getVertices().primero().getAdyacentes().primero(),
is(grafo.getVertices().ultimo()));
	}

	@Test
	public void agregoUnVerticeTestYLoMarco() {

		verticeUno.setVisitado(true);
		grafo.agregarVertice(verticeUno);
		assertThat("Agrego un vertice", grafo.getCantidadDeNodosGrafo(), is(1));
		assertThat("Esta marcado?", grafo.getVertices().primero().isVisitado(),
is(true));

	}

	@Test
	public void alAgregarUnArcoSoloAgregaLosVerticesSiCorresponde() {

		grafo.agregarVertice(verticeUno);

		grafo.agregarArco(verticeUno, verticeDos);
		assertThat("Hay dos vertices", grafo.getCantidadDeNodosGrafo(), is(2));
		assertThat("Hay un arco entre Vertice Uno y Dos",
grafo.getVertices().primero().getAdyacentes().primero(),
is(grafo.getVertices().ultimo()));

	}

	@Test
	public void
alAgregarUnArcoSoloAgregaLosVerticesSiCorrespondeAunqueNoSeanLaMismaInstancia() {

		Vertice<String> verticeDosBis = new Vertice<String>("VerticeDos");

		grafo.agregarVertice(verticeUno);
		grafo.agregarVertice(verticeDosBis);

		grafo.agregarArco(verticeUno, verticeDos);
		assertThat("Hay dos vertices", grafo.getCantidadDeNodosGrafo(), is(2));
		assertThat("Hay un arco entre Vertice Uno y Dos",
grafo.getVertices().primero().getAdyacentes().primero(),
is(grafo.getVertices().ultimo()));

	}

	@Test
	public void noAgregaVerticesRepetidos() {

		Vertice<String> verticeDosBis = new Vertice<String>("VerticeDos");

		grafo.agregarVertice(verticeDos);
```

```
		grafo.agregarVertice(verticeDosBis);

		assertThat("Hay dos vertices", grafo.getCantidadDeNodosGrafo(), is(1));

	}

	@Test
	public void agregarGrafoEnunciadoYDFSTest() {
		crearGrafoEnunciado();
		grafo.recorridoDFS(this.grafo.getVertices());
		assertThat("Hay 8 vertices en el recorrido",
grafo.getRecorridoDFS().tamanio(), is(8));

	}

	@Test
	public void agregarGrafoEnunciadoYBFSTest() {
		crearGrafoEnunciado();
		grafo.recorridoBFS(this.grafo.getVertices());
		assertThat("Hay 8 vertices en el recorridos",
grafo.getRecorridoBFS().tamanio(), is(8));
	}

	@Test
	public void encontrarCiclosTest() {
		crearGrafoEnunciado();
		grafo.encontrarCiclos(this.grafo.getVertices().primero());
		assertThat("Hay 8 vertices", grafo.getCantidadDeNodosGrafo(), is(8));
		assertThat("Hay 2 ciclos en el grafo", grafo.getCiclosGrafo().tamanio(),
is(2));
		assertThat("El primer ciclo es de 4 ",
grafo.getCiclosGrafo().primero().tamanio(), is(4));
		assertThat("El segundo ciclo es de 4 ",
grafo.getCiclosGrafo().ultimo().tamanio(), is(4));
	}

	@Test
	public void encontrarCiclosEnGrafoAciclicoTest() {
		crearGrafoAciclico();
		grafo.encontrarCiclos(this.grafo.getVertices().primero());
		assertThat("Hay 8 vertices", grafo.getCantidadDeNodosGrafo(), is(7));
		assertThat("Hay 4 ciclos en el grafo", grafo.getCiclosGrafo().tamanio(),
is((grafo.getCantidadDeNodosGrafo()+1)/2));
	}

	private void crearGrafoEnunciado() {
		verticeA = new Vertice<String>("A");
		verticeB = new Vertice<String>("B");
		verticeC = new Vertice<String>("C");
		verticeD = new Vertice<String>("D");
		verticeE = new Vertice<String>("E");
		verticeF = new Vertice<String>("F");
		verticeG = new Vertice<String>("G");
		verticeH = new Vertice<String>("H");

		grafo.agregarArco(verticeA, verticeB);
		grafo.agregarArco(verticeA, verticeC);
		grafo.agregarArco(verticeB, verticeD);
		grafo.agregarArco(verticeC, verticeD);
		grafo.agregarArco(verticeD, verticeE);
		grafo.agregarArco(verticeE, verticeF);
		grafo.agregarArco(verticeE, verticeG);
		grafo.agregarArco(verticeF, verticeH);
```

```
		grafo.agregarArco(verticeG, verticeH);
	}

	private void crearGrafoAciclico() {
		verticeA = new Vertice<String>("A");
		verticeB = new Vertice<String>("B");
		verticeC = new Vertice<String>("C");
		verticeD = new Vertice<String>("D");
		verticeE = new Vertice<String>("E");
		verticeF = new Vertice<String>("F");
		verticeG = new Vertice<String>("G");

		grafo.agregarArco(verticeA, verticeB);
		grafo.agregarArco(verticeB, verticeC);
		grafo.agregarArco(verticeC, verticeD);
		grafo.agregarArco(verticeD, verticeE);
		grafo.agregarArco(verticeE, verticeF);
		grafo.agregarArco(verticeF, verticeG);
	}


}
```

ListaEnlazadaTest.java

```
package ar.fi.uba.tda.colecciones.test;

import static org.hamcrest.MatcherAssert.*;
import static org.hamcrest.Matchers.*;

import java.util.Iterator;
import java.util.ListIterator;

import org.junit.Before;
import org.junit.Test;

import ar.fi.uba.tda.colecciones.ListaEnlazada;

public class ListaEnlazadaTest {

	private ListaEnlazada<Object> lista;

	@Before
	public void setup() {

		lista = new ListaEnlazada<Object>();
	}

	@Test
	public void listaVacia() {


		assertThat("la lista no estÃ¡ vacÃa", lista.vacia(), is(true));
	}

	@Test
	public void enUnaListaVaciaElPrimerElementoEsNulo() {

		assertThat("el primer elemento", lista.primero(), nullValue());
	}

	@Test
	public void enUnaListaVaciaElUltimoElementoEsNulo() {
```

```
		assertThat("el primer elemento", lista.ultimo(), nullValue());
	}

	@Test
	public void enUnaListaVaciaElTamanioEsCero() {

		assertThat("el tamaÃ±o", lista.tamanio(), is(0));
	}

	@Test
	public void siAgregoUnElementoEsElPrimero() {

		Object elemento = new Object();
		lista.agregar(elemento);

		Object primero = lista.primero();

		assertThat("el primer elemento", primero, notNullValue());
		assertThat("el primer elemento", primero, is(elemento));

		assertThat("el tamaÃ±o", lista.tamanio(), is(1));

		assertThat("lista vacia", lista.vacia(), is(false));
	}

	@Test
	public void siAgregoUnElementoEsElUltimo() {

		Object elemento = new Object();
		lista.agregar(elemento);

		assertThat("el primer elemento", lista.ultimo(), notNullValue());
		assertThat("el primer elemento", lista.ultimo(), is(elemento));
	}

	@Test
	public void siAgregoDosElementosElTamanioEsDos() {

		Object primero = new Object();
		Object segundo = new Object();

		lista.agregar(primero);
		lista.agregar(segundo);

		assertThat("el tamaÃ±o de la lista", lista.tamanio(), is(2));
	}

	@Test
	public void siAgregoDosElementosElSegundoEsElUltimo() {

		Object primero = new Object();
		Object segundo = new Object();

		lista.agregar(primero);
		lista.agregar(segundo);

		assertThat("el primer elemento", lista.primero(), is(primero));
		assertThat("el ultimo elemento", lista.ultimo(), is(segundo));
	}

	@Test
	public void siLaListaTieneElementosElIteradorTieneSiguiente() {
```

```
		Object primero = new Object();

		lista.agregar(primero);

		Iterator<Object> iterador = lista.iterador();

		assertThat("la lista tiene elementos", iterador.hasNext(), is(true));

	}

	@Test
	public void siLaListaNoTieneElementosElIteradorNoTieneSiguiente() {

		Iterator<Object> iterador = lista.iterador();

		assertThat("la lista tiene elementos", iterador.hasNext(), is(false));

	}

	@Test
	public void puedoRecorrerLaListaEnOrden() {

		Object primero = new Object();
		Object segundo = new Object();

		lista.agregar(primero);
		lista.agregar(segundo);

		Iterator<Object> iterador = lista.iterador();

		assertThat("el primer elemento", iterador.next(), is(primero));
		assertThat("el segundo elemento", iterador.next(), is(segundo));

	}

	@Test
	public void despuesDeSacarElUltimoElementoYaNoPuedoIterar() {

		Object primero = new Object();
		Object segundo = new Object();

		lista.agregar(primero);
		lista.agregar(segundo);

		Iterator<Object> iterador = lista.iterador();

		assertThat("el primer elemento", iterador.next(), is(primero));
		assertThat("el segundo elemento", iterador.next(), is(segundo));

		assertThat("la lista tiene elementos", iterador.hasNext(), is(false));

	}

	@Test
	public void puedoRecorrerLaListaEnOrdenInverso() {

		Object primero = new Object();
		Object segundo = new Object();

		lista.agregar(primero);
		lista.agregar(segundo);

		ListIterator<Object> iterador = lista.iterador();
```

```
		assertThat("el primer elemento", iterador.next(), is(primero));
		assertThat("el segundo elemento", iterador.next(), is(segundo));

		assertThat("el primer elemento", iterador.previous(), is(primero));

		assertThat("la lista tiene previo", iterador.hasPrevious(), is(true));
	}

	@Test
	public void buscaUnElementoEnLaLista() {

		lista.agregar("Hola");

		boolean encontrado = lista.contiene("Hola");

		assertThat("el elemento", encontrado, is(true));
	}

	@Test
	public void siElElementoNoEstaDaBuscarloFalse() {

		lista.agregar("Hola");

		boolean encontrado = lista.contiene("Chau");

		assertThat("el elemento", encontrado, is(false));
	}

	@Test
	public void obtieneUnElementoEnLaLista() {

		lista.agregar("Hola");

		Object encontrado = lista.obtener("Hola");

		assertThat("el elemento", encontrado, is((Object)"Hola"));
	}

	@Test
	public void siElElementoNoEstaDaObteneroDevuelveNull() {

		lista.agregar("Hola");

		Object encontrado = lista.obtener("Chau");

		assertThat("el elemento", encontrado, nullValue());
	}
}
```

AumentadorDeRobustezTest.java

```
package ar.fi.uba.tda.util.test;

import org.hamcrest.MatcherAssert;
import org.hamcrest.Matchers;
import org.junit.Test;

import ar.fi.uba.tda.colecciones.Arista;
import ar.fi.uba.tda.colecciones.Grafo;
import ar.fi.uba.tda.colecciones.ListaEnlazada;
import ar.fi.uba.tda.colecciones.Vertice;
import ar.fi.uba.tda.util.AumentadorDeRobustez;
```

```
public class AumentadorDeRobustezTest {

	@Test
	@SuppressWarnings({ "rawtypes" })
	public void siRecibeUnNodoYRobustez1NoAgregaAristas() {

		Grafo<String> grafo = new Grafo<String>();

		AumentadorDeRobustez aumentador = new AumentadorDeRobustez(grafo);

		ListaEnlazada<ListaEnlazada<Vertice>> ciclos = new
ListaEnlazada<ListaEnlazada<Vertice>>();

		ListaEnlazada<Vertice> primerCiclo = new ListaEnlazada<Vertice>();

		primerCiclo.agregar(new Vertice<String>());
		ciclos.agregar(primerCiclo);
		int robustez = 1;

		aumentador.aumentar(ciclos, robustez);

		ListaEnlazada<Arista> aristasAgregadas =
aumentador.getAristasAgregadas();

		MatcherAssert.assertThat("la cantidad de aristas agregadas",
aristasAgregadas.tamanio(), Matchers.is(0));
	}

	@Test
	@SuppressWarnings({ "unchecked", "rawtypes" })
	public void siRecibeDosNodosYRobustez1AgregaUnaArista() {

		Grafo<String> grafo = new Grafo<String>();
		Vertice<String> verticeA = new Vertice<String>("A");
		Vertice<String> verticeB = new Vertice<String>("B");

		grafo.agregarVertice(verticeA);
		grafo.agregarVertice(verticeB);

		AumentadorDeRobustez aumentador = new AumentadorDeRobustez(grafo);

		ListaEnlazada<ListaEnlazada<Vertice>> ciclos = new
ListaEnlazada<ListaEnlazada<Vertice>>();

		ListaEnlazada<Vertice> primerCiclo = new ListaEnlazada<Vertice>();
		primerCiclo.agregar(verticeA);

		ListaEnlazada<Vertice> segundoCiclo = new ListaEnlazada<Vertice>();
		segundoCiclo.agregar(verticeB);


		ciclos.agregar(primerCiclo);
		ciclos.agregar(segundoCiclo);

		int robustez = 1;

		aumentador.aumentar(ciclos, robustez);

		ListaEnlazada<Arista> aristasAgregadas =
aumentador.getAristasAgregadas();

		MatcherAssert.assertThat("la cantidad de aristas agregadas",
aristasAgregadas.tamanio(), Matchers.is(1));
```

```
			MatcherAssert.assertThat("el origen de la arista", (Vertice<String>)
aristasAgregadas.primero().getOrigen(), Matchers.is(verticeA));
			MatcherAssert.assertThat("el destino de la arista", (Vertice<String>)
aristasAgregadas.primero().getDestino(), Matchers.is(verticeB));
	}

	@Test
	@SuppressWarnings({ "rawtypes" })
	public void siRecibeDosNodosYRobustez2NoAgregaArista() {

			Grafo<String> grafo = new Grafo<String>();
			Vertice<String> verticeA = new Vertice<String>("A");
			Vertice<String> verticeB = new Vertice<String>("B");

			grafo.agregarVertice(verticeA);
			grafo.agregarVertice(verticeB);

			AumentadorDeRobustez aumentador = new AumentadorDeRobustez(grafo);

			ListaEnlazada<ListaEnlazada<Vertice>> ciclos = new
ListaEnlazada<ListaEnlazada<Vertice>>();

			ListaEnlazada<Vertice> primerCiclo = new ListaEnlazada<Vertice>();
			primerCiclo.agregar(verticeA);

			ListaEnlazada<Vertice> segundoCiclo = new ListaEnlazada<Vertice>();
			segundoCiclo.agregar(verticeB);


			ciclos.agregar(primerCiclo);
			ciclos.agregar(segundoCiclo);

			int robustez = 2;

			aumentador.aumentar(ciclos, robustez);

			ListaEnlazada<Arista> aristasAgregadas =
aumentador.getAristasAgregadas();

			MatcherAssert.assertThat("la cantidad de aristas agregadas",
aristasAgregadas.tamanio(), Matchers.is(0));
	}

	@Test
	@SuppressWarnings({ "rawtypes" })
	public void siRecibeDosNodosYaEnlazadosYRobustez1NoAgregaArista() {

			Grafo<String> grafo = new Grafo<String>();
			Vertice<String> verticeA = new Vertice<String>("A");
			Vertice<String> verticeB = new Vertice<String>("B");

			grafo.agregarVertice(verticeA);
			grafo.agregarVertice(verticeB);

			grafo.agregarArco(verticeA, verticeB);

			AumentadorDeRobustez aumentador = new AumentadorDeRobustez(grafo);

			ListaEnlazada<ListaEnlazada<Vertice>> ciclos = new
ListaEnlazada<ListaEnlazada<Vertice>>();

			ListaEnlazada<Vertice> primerCiclo = new ListaEnlazada<Vertice>();
			primerCiclo.agregar(verticeA);
```

```
			ListaEnlazada<Vertice> segundoCiclo = new ListaEnlazada<Vertice>();
			segundoCiclo.agregar(verticeB);


			ciclos.agregar(primerCiclo);
			ciclos.agregar(segundoCiclo);

			int robustez = 1;

			aumentador.aumentar(ciclos, robustez);

			ListaEnlazada<Arista> aristasAgregadas =
aumentador.getAristasAgregadas();

			MatcherAssert.assertThat("la cantidad de aristas agregadas",
aristasAgregadas.tamanio(), Matchers.is(0));
	}

	@Test
	@SuppressWarnings({ "rawtypes", "unchecked" })
	public void siRecibeDosGruposDeDosNodosYRobustez2AgregaDosAristas() {

			Grafo<String> grafo = new Grafo<String>();
			Vertice<String> verticeA = new Vertice<String>("A");
			Vertice<String> verticeB = new Vertice<String>("B");
			Vertice<String> verticeC = new Vertice<String>("C");
			Vertice<String> verticeD = new Vertice<String>("D");

			grafo.agregarVertice(verticeA);
			grafo.agregarVertice(verticeB);
			grafo.agregarVertice(verticeC);
			grafo.agregarVertice(verticeD);

			grafo.agregarArco(verticeA, verticeB);
			grafo.agregarArco(verticeC, verticeD);

			AumentadorDeRobustez aumentador = new AumentadorDeRobustez(grafo);

			ListaEnlazada<ListaEnlazada<Vertice>> ciclos = new
ListaEnlazada<ListaEnlazada<Vertice>>();

			ListaEnlazada<Vertice> primerCiclo = new ListaEnlazada<Vertice>();
			primerCiclo.agregar(verticeA);
			primerCiclo.agregar(verticeB);

			ListaEnlazada<Vertice> segundoCiclo = new ListaEnlazada<Vertice>();
			segundoCiclo.agregar(verticeC);
			segundoCiclo.agregar(verticeD);

			ciclos.agregar(primerCiclo);
			ciclos.agregar(segundoCiclo);

			int robustez = 2;

			aumentador.aumentar(ciclos, robustez);

			ListaEnlazada<Arista> aristasAgregadas =
aumentador.getAristasAgregadas();

			MatcherAssert.assertThat("la cantidad de aristas agregadas",
aristasAgregadas.tamanio(), Matchers.is(2));
```

```
			MatcherAssert.assertThat("el origen de la primera arista",
(Vertice<String>) aristasAgregadas.primero().getOrigen(), Matchers.is(verticeA));
			MatcherAssert.assertThat("el destino de la primera arista",
(Vertice<String>) aristasAgregadas.primero().getDestino(), Matchers.is(verticeC));

			MatcherAssert.assertThat("el origen de la segudna arista",
(Vertice<String>) aristasAgregadas.ultimo().getOrigen(), Matchers.is(verticeB));
			MatcherAssert.assertThat("el destino de la segunda arista",
(Vertice<String>) aristasAgregadas.ultimo().getDestino(), Matchers.is(verticeD));
	}

	@Test
	@SuppressWarnings({ "unchecked", "rawtypes" })
	public void siRecibeDosGruposDeDosNodosYRobustez1AgregaUnaArista() {

			Grafo<String> grafo = new Grafo<String>();
			Vertice<String> verticeA = new Vertice<String>("A");
			Vertice<String> verticeB = new Vertice<String>("B");
			Vertice<String> verticeC = new Vertice<String>("C");
			Vertice<String> verticeD = new Vertice<String>("D");

			grafo.agregarVertice(verticeA);
			grafo.agregarVertice(verticeB);
			grafo.agregarVertice(verticeC);
			grafo.agregarVertice(verticeD);

			grafo.agregarArco(verticeA, verticeB);
			grafo.agregarArco(verticeC, verticeD);

			AumentadorDeRobustez aumentador = new AumentadorDeRobustez(grafo);

			ListaEnlazada<ListaEnlazada<Vertice>> ciclos = new
ListaEnlazada<ListaEnlazada<Vertice>>();

			ListaEnlazada<Vertice> primerCiclo = new ListaEnlazada<Vertice>();
			primerCiclo.agregar(verticeA);
			primerCiclo.agregar(verticeB);

			ListaEnlazada<Vertice> segundoCiclo = new ListaEnlazada<Vertice>();
			segundoCiclo.agregar(verticeC);
			segundoCiclo.agregar(verticeD);

			ciclos.agregar(primerCiclo);
			ciclos.agregar(segundoCiclo);

			int robustez = 1;

			aumentador.aumentar(ciclos, robustez);

			ListaEnlazada<Arista> aristasAgregadas =
aumentador.getAristasAgregadas();

			MatcherAssert.assertThat("la cantidad de aristas agregadas",
aristasAgregadas.tamanio(), Matchers.is(1));
			MatcherAssert.assertThat("el origen de la primera arista",
(Vertice<String>) aristasAgregadas.primero().getOrigen(), Matchers.is(verticeA));
			MatcherAssert.assertThat("el destino de la primera arista",
(Vertice<String>) aristasAgregadas.primero().getDestino(), Matchers.is(verticeC));

	}

	@Test
	@SuppressWarnings({ "unchecked", "rawtypes" })
```

```
	public void siRecibeTresGruposDeDosNodosYRobustez1AgregaDosAristas() {

		Grafo<String> grafo = new Grafo<String>();
		Vertice<String> verticeA = new Vertice<String>("A");
		Vertice<String> verticeB = new Vertice<String>("B");
		Vertice<String> verticeC = new Vertice<String>("C");
		Vertice<String> verticeD = new Vertice<String>("D");
		Vertice<String> verticeE = new Vertice<String>("E");
		Vertice<String> verticeF = new Vertice<String>("F");

		grafo.agregarVertice(verticeA);
		grafo.agregarVertice(verticeB);
		grafo.agregarVertice(verticeC);
		grafo.agregarVertice(verticeD);
		grafo.agregarVertice(verticeE);
		grafo.agregarVertice(verticeF);

		grafo.agregarArco(verticeA, verticeB);
		grafo.agregarArco(verticeC, verticeD);
		grafo.agregarArco(verticeE, verticeF);

		AumentadorDeRobustez aumentador = new AumentadorDeRobustez(grafo);

		ListaEnlazada<ListaEnlazada<Vertice>> ciclos = new
ListaEnlazada<ListaEnlazada<Vertice>>();

		ListaEnlazada<Vertice> primerCiclo = new ListaEnlazada<Vertice>();
		primerCiclo.agregar(verticeA);
		primerCiclo.agregar(verticeB);

		ListaEnlazada<Vertice> segundoCiclo = new ListaEnlazada<Vertice>();
		segundoCiclo.agregar(verticeC);
		segundoCiclo.agregar(verticeD);

		ListaEnlazada<Vertice> tercerCiclo = new ListaEnlazada<Vertice>();
		tercerCiclo.agregar(verticeE);
		tercerCiclo.agregar(verticeF);

		ciclos.agregar(primerCiclo);
		ciclos.agregar(segundoCiclo);
		ciclos.agregar(tercerCiclo);

		int robustez = 1;

		aumentador.aumentar(ciclos, robustez);

		ListaEnlazada<Arista> aristasAgregadas =
aumentador.getAristasAgregadas();

		MatcherAssert.assertThat("la cantidad de aristas agregadas",
aristasAgregadas.tamanio(), Matchers.is(2));
		MatcherAssert.assertThat("el origen de la primera arista",
(Vertice<String>) aristasAgregadas.primero().getOrigen(), Matchers.is(verticeA));
		MatcherAssert.assertThat("el destino de la primera arista",
(Vertice<String>) aristasAgregadas.primero().getDestino(), Matchers.is(verticeC));

		MatcherAssert.assertThat("el origen de la segudna arista",
(Vertice<String>) aristasAgregadas.ultimo().getOrigen(), Matchers.is(verticeC));
		MatcherAssert.assertThat("el destino de la segunda arista",
(Vertice<String>) aristasAgregadas.ultimo().getDestino(), Matchers.is(verticeE));

	}
```

```
	@Test
	@SuppressWarnings({ "rawtypes" })
	public void noAgregaAristaQueUnaUnVerticeConSiMismo() {

		Grafo<String> grafo = new Grafo<String>();
		Vertice<String> verticeA = new Vertice<String>("A");
		Vertice<String> verticeB = new Vertice<String>("B");

		grafo.agregarVertice(verticeA);
		grafo.agregarVertice(verticeB);

		AumentadorDeRobustez aumentador = new AumentadorDeRobustez(grafo);

		ListaEnlazada<ListaEnlazada<Vertice>> ciclos = new
ListaEnlazada<ListaEnlazada<Vertice>>();

		ListaEnlazada<Vertice> primerCiclo = new ListaEnlazada<Vertice>();
		primerCiclo.agregar(verticeA);

		ciclos.agregar(primerCiclo);

		int robustez = 1;

		aumentador.aumentar(ciclos, robustez);

		ListaEnlazada<Arista> aristasAgregadas =
aumentador.getAristasAgregadas();

		MatcherAssert.assertThat("la cantidad de aristas agregadas",
aristasAgregadas.tamanio(), Matchers.is(0));
	}
}
```

CargadorDeGrafosTest.java

```
package ar.fi.uba.tda.util.test;

import static org.hamcrest.MatcherAssert.*;
import static org.hamcrest.Matchers.*;

import java.io.BufferedReader;
import java.io.IOException;

import org.junit.Before;
import org.junit.Test;
import static org.mockito.Mockito.*;

import ar.fi.uba.tda.colecciones.Grafo;
import ar.fi.uba.tda.colecciones.ListaEnlazada;
import ar.fi.uba.tda.colecciones.Vertice;
import ar.fi.uba.tda.util.CargadorDeGrafos;

public class CargadorDeGrafosTest {

	private Grafo<String> grafo;
	private CargadorDeGrafos cargador;

	@Before
	public void setup() {

		grafo = new Grafo<String>();
		cargador = new CargadorDeGrafos(grafo);
	}
```

```
	@Test
	public void agregaUnaAristaUniendoDosVerticesDados() {

		cargador.agregarArista("vertice1", "vertice2");

		assertThat("la cantidad de vertices", grafo.getCantidadDeNodosGrafo(),
is(2));
		assertThat("el vertice uno y el dos",
grafo.getVertices().primero().getAdyacentes().primero(),
is(grafo.getVertices().ultimo()));
	}

	@Test
	public void noAgregaVerticesRepetidos() {

		cargador.agregarArista("vertice1", "vertice2");
		cargador.agregarArista("vertice1", "vertice3");

		assertThat("la cantidad de vertices", grafo.getCantidadDeNodosGrafo(),
is(3));
		assertThat("el vertice uno y el dos",
grafo.getVertices().primero().getAdyacentes().primero(), is(new
Vertice<String>("vertice2")));
	}

	@Test
	public void cargaDesdeUnArchivoUnVerticeConUnAdyacente() throws IOException {

		BufferedReader reader = mock(BufferedReader.class);
		when(reader.readLine()).thenReturn("A:B", (String)null);

		cargador.cargar(reader);

		Vertice<String> verticeA = new Vertice<String>("A");
		Vertice<String> verticeB = new Vertice<String>("B");

		assertThat("la cantidad de vertices", grafo.getCantidadDeNodosGrafo(),
is(2));

		assertThat("el primer vertice", grafo.getVertices().primero(),
is(verticeA));
		assertThat("el segundo vertice", grafo.getVertices().ultimo(),
is(verticeB));

		assertThat("el vertice A y el B",
grafo.getVertices().primero().getAdyacentes().primero(), is(verticeB));
	}

	@Test
	public void cargaDesdeUnArchivoUnVerticeConDosAdyacentes() throws IOException {

		BufferedReader reader = mock(BufferedReader.class);
		when(reader.readLine()).thenReturn("A:B,C", (String)null);

		cargador.cargar(reader);

		Vertice<String> verticeA = new Vertice<String>("A");
		Vertice<String> verticeB = new Vertice<String>("B");
		Vertice<String> verticeC = new Vertice<String>("C");

		assertThat("la cantidad de vertices", grafo.getCantidadDeNodosGrafo(),
is(3));
```

```
			assertThat("el primer vertice", grafo.getVertices().primero(),
is(verticeA));
			assertThat("el tercer vertice", grafo.getVertices().ultimo(),
is(verticeC));

			assertThat("el vertice A y el B",
grafo.getVertices().primero().getAdyacentes().primero(), is(verticeB));
			assertThat("el vertice A y el C",
grafo.getVertices().primero().getAdyacentes().ultimo(), is(verticeC));
		}

		@Test
		public void cargaDesdeUnArchivoDosVerticeConDosAdyacentes() throws IOException {

			BufferedReader reader = mock(BufferedReader.class);
			when(reader.readLine()).thenReturn("A:B,C", "B:D,E", (String)null);

			cargador.cargar(reader);

			Vertice<String> verticeA = new Vertice<String>("A");
			Vertice<String> verticeB = new Vertice<String>("B");
			Vertice<String> verticeC = new Vertice<String>("C");
			Vertice<String> verticeD = new Vertice<String>("D");
			Vertice<String> verticeE = new Vertice<String>("E");

			assertThat("la cantidad de vertices", grafo.getCantidadDeNodosGrafo(),
is(5));

			ListaEnlazada<Vertice<String>> vertices = grafo.getVertices();

			Vertice<String> verticeAEnElGrafo = vertices.obtener(verticeA);
			Vertice<String> verticeBEnElGrafo = vertices.obtener(verticeB);
			Vertice<String> verticeCEnElGrafo = vertices.obtener(verticeC);
			Vertice<String> verticeDEnElGrafo = vertices.obtener(verticeD);
			Vertice<String> verticeEEnElGrafo = vertices.obtener(verticeE);

			assertThat("el vertice A", verticeAEnElGrafo, is(verticeA));
			assertThat("arista entre A y B",
verticeAEnElGrafo.getAdyacentes().primero(), is(verticeB));
			assertThat("arista entre A y C",
verticeAEnElGrafo.getAdyacentes().ultimo(), is(verticeC));

			assertThat("el vertice B", verticeBEnElGrafo, is(verticeB));
			assertThat("arista entre B y A",
verticeBEnElGrafo.getAdyacentes().primero(), is(verticeA));
			assertThat("arista entre B y E",
verticeBEnElGrafo.getAdyacentes().ultimo(), is(verticeE));

			assertThat("el vertice C", verticeCEnElGrafo, is(verticeC));
			assertThat("arista entre C y A",
verticeCEnElGrafo.getAdyacentes().primero(), is(verticeA));

			assertThat("el vertice D", verticeDEnElGrafo, is(verticeD));
			assertThat("arista entre D y B",
verticeDEnElGrafo.getAdyacentes().primero(), is(verticeB));

			assertThat("el vertice E", verticeEEnElGrafo, is(verticeE));
			assertThat("arista entre E y B",
verticeEEnElGrafo.getAdyacentes().primero(), is(verticeB));

		}

		@Test
```

```
	public void cargaDesdeUnArchivoUnVerticeSinAdyacentes() throws IOException {

		BufferedReader reader = mock(BufferedReader.class);
		when(reader.readLine()).thenReturn("A", (String)null);

		cargador.cargar(reader);

		Vertice<String> verticeA = new Vertice<String>("A");

		assertThat("la cantidad de vertices", grafo.getCantidadDeNodosGrafo(),
is(1));

		assertThat("el primer vertice", grafo.getVertices().primero(),
is(verticeA));

	}
}
```