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 Risaro	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:

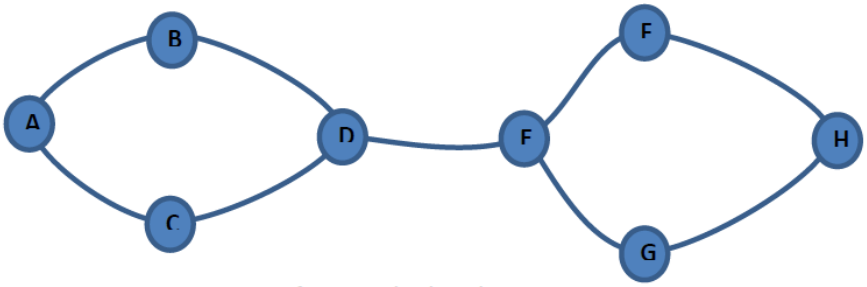
<vértice>: <vértice adyacente 1>, <vértice adyacente 2>

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>

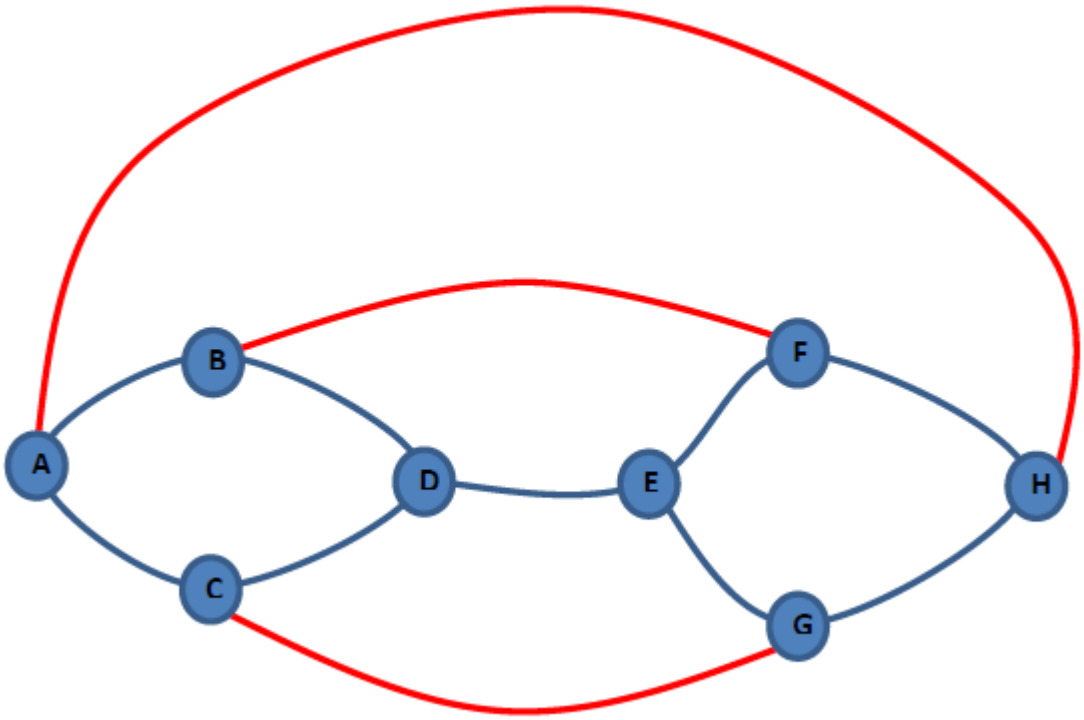
Ejemplo:



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):

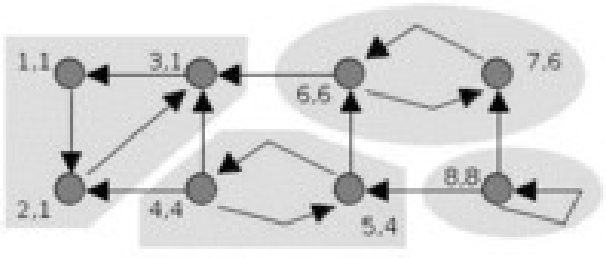
```
Leer el archivo de entrada
Cargar el grafo
Ubicar los ciclos (Utilizando un algoritmo basado en DFS para
obtenerlos)
Si la robustez pedida es compatible con el grafo entonces
     cicloUno = obtenerCiclo(1)
     cicloDos = obtenerCiclo(2)
      Mientras haya vertices en ciclos y robustez <> robPedida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

Análisis de Orden

Hemos implementado una **lista doblemente enlazada**, esto garantiza que la **inserción** de un elemento se hace en tiempo constante. Esta lista cuenta con una estructura auxiliar (HashMap) utilizada para llevar el registro de los elementos de la lista, esta estructura garantizaun costo constante para las operaciones **obtener**, **contiene** e **insertar**. Utilizando esta colección como base para todas nuestras estructuras podemos mantener los tiempos acotados solo a los costos propios del algoritmo, sin incurrir en un overhead por el uso de la colección.

El **grafo** se implementó utilizando listas de adyacencia, dado que nuestras listas realizan todas sus operaciones en tiempo constante, las operaciones estándar del grafo, **agregar arco**, **agregar vértice**, **verificar si un vértice está en el grafo**, se realizan en tiempo constante.

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 estrictamente 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) que es equivalente a **O (V).**

Algoritmo: Obtener los valores de entrada
Utilizado en: Clase CargadorDeGrafos, método cargar.
Análisis: Se lee cada linea del archivo, vértice, y por cada una los items adyacentes, aristas, por lo que el orden es **O(|V| + |A|)**.

Algoritmo: Aumentar la robustez
Utilizado en: Clase AumentadorDeRobustez, método aumentar.
Análisis: Este algoritmo itera por los ciclos del grafo, en el peor caso (grafo completamente desconectado) tendremos |V| ciclos, e itera por cada uno de los vértices de ese ciclo, en el peor caso (grafo con un solo ciclo) tendremos |V| vértices en un ciclo.

Para cada vértice de cada ciclo, conecta ese vértice con un vértice del "ciclo siguiente"(*) hasta que agrega tantos vértices como la robustez pedida, de esta forma aumentamos la robustez entre cada ciclo, al finalizar habremos aumentado la robustez total del grafo.

Dado que recorremos todos los ciclos, y para cada ciclo todos los vértices que lo componen, que un vértice esta en un único ciclo y que lo emparejamos con cada vértice del ciclo siguiente, nos encontramos con que visitamos cada vértice 2 veces por lo que el costo de esta operación es O(2|V|) = O(|V|).

(*) A los efectos de este algoritmo consideramos la lista de ciclos como una lista circular por lo que "ciclo siguiente" puede ser el ciclo que le sigue al actual o el primero en caso de estar en el último ciclo de la lista.

Teniendo en cuenta que nuestra implementación hace uso de los 3 algoritmos mencionados anteriormente de manera secuencial el costo total es la suma de sus costos es decir: **El orden total de este trabajo es O(|V| + |A| + |V| + |V|) = O(3|V| + |A|) = O(|V| + |A|).**

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_1_2013_11_4/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

Para la resolución de este trabajo práctico tuvimos que diseñar, optimizar e implementar un algoritmo que pudiera aumentar la robustez de un grafo dado.

En la primera iteración logramos un algoritmo que conseguía el objetivo pero contaba con una serie de fallas detectadas oportunamente por el corrector.

El principal problema de la primera entrega estaba relacionado con el orden de las operaciones, dado que nuestra implementación de lista requería un costo elevado para operaciones básicas como la búsqueda de un elemento y sumado al hecho de que la mayoría de los pasos dependían de dicha lista el costo total terminaba siendo muy elevado.

Otro de los problemas se daba al no considerar correctamente el grado de los vértices del grafo al momento de elegir cuales unir.

Para la segunda entrega atacamos, principalmente, las operaciones de la lista buscando optimizarlas lo más posible ya que todas las otras estructuras de datos y algoritmos recaen en ella y cualquier costo extra sobre la estructura de datos base puede hacer crecer exponencialmente el costo de las operaciones más simples.

En este sentido decidimos utilizar una estructura auxiliar dentro de la lista que nos permitiera rápido acceso a un elemento dado, para ello utilizamos un mapa, en particular un HashMap, cada elemento insertado en la lista es insertado también en el mapa, dado que la implementación utilizada garantiza un tiempo constante para todas las operaciones, sin importar el tamaño del mapa, no incurrimos en una penalidad por el uso de esta estructura auxiliar. Luego de implementado este cambio modificamos las operaciones de búsqueda para que hagan uso de esta nueva estructura, evitando recorrer la lista entera para determinar si un ítem se encuentra o no en ella.

Una vez finalizada esta tarea procedimos a mejorar el algoritmo que detecta los ciclos en el grafo. En un principio utilizamos una versión de Tarjan bastante modificada, luego de revisar el funcionamiento del programa decidimos ir a una versión más estándar, aunque mantuvimos algunas pequeñas diferencias para simplificar su implementación en nuestro contexto.

Luego nos abocamos a solucionar algunos pequeños problemas en el algoritmo que aumenta la robustez del grafo, en particular, no estábamos teniendo en cuenta el grado de los vértices, esto hacía que el algoritmos enlazara dos vértices de dos ciclos distintos sin priorizar los de menor grado, llegando a soluciones que no eran correctas.

Esta experiencia, con sus errores, nos hizo ver la importancia de implementar unas buenas estructuras de datos ya que una pequeña optimización en ellas, el cambio fue sencillo y de baja complejidad, nos llevó a mejorar sustancialmente la performance general de la solución.

Código Fuente

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;

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 {

		System.out.println("Teoría de algoritmos - TP 1");
		System.out.println("Autores:");
		System.out.println("Alejo Vinjoy - 83.989");
		System.out.println("Santiago Nicolas Risaro Sesar - 84.623");

		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));
      }

      /**
       * Método principal del TP, se encarga de unir todas las piezas.
       *
       * 1° Carga el grafo: O(|V| + |A|).
       * 2° Busca los ciclos: O(|V|).
       * 3° Aumenta la robustez del grafo O(|V|).
       * 4° Imprime los resultados O(k), siendo k la cantidad de aristas encontradas.
       *
       * El orden total de este trabajo es O(|V| + |A| + |V| + |V| + k) =
       * O(3|V| + |A|) = O(|V| + |A|)
       *
       * @param robustezDeseada
       * @param archivo
       * @throws IOException
       */
      @SuppressWarnings({ "unchecked" })
      public void ejecutar(int robustezDeseada, BufferedReader archivo) throws
IOException {

             cargador.cargar(archivo);

             grafo.encontrarCiclos(grafo.getVertices());
             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);

                }


        }

}
```

Grafo.java

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

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;

        /**
         * Inicializa el grafo y las estructuras de datos auxiliares por el usadas.
         *
         * El orden de esta operación es O(1).
         */
        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 vértice al grafo solo si no se encuentra previamente agregado.
	 *
	 * Tanto las operaciones de agregar como de verificar si un elemento está
contenido o no
	 * en la lista son constantes por lo que el costo de esta operación es O(1).
	 *
	 * @see ListaEnlazada#contiene(Object)
	 * @see ListaEnlazada#agregar(Object)
	 *
	 * @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 vértices (no es grafo dirigido)
	 * Si los vértices no pertenecen al grafo los agrega.
	 *
	 * Las operaciones para obtener y agregar un vértice son O(1), este método
	 * solo hace uso de esas operaciones por lo que el orden de este método es O(1).
	 *
	 * @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;
		}

		if (!inicio.getAdyacentes().contiene(fin)) {
			inicio.getAdyacentes().agregar(fin);
		}

		if (!fin.getAdyacentes().contiene(inicio)) {
			fin.getAdyacentes().agregar(inicio);
		}
```

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

	}

	/**
	 * Recupera una referencia a un vértice dado.
	 *
	 * @see ListaEnlazada#obtener(Object)
	 *
	 */
	private Vertice<T> obtener(Vertice<T> buscado) {

		return this.vertices.obtener(buscado);
	}

	/**
	 * Verifica si un vértice está o no en el grafo.
	 *
	 * @see ListaEnlazada#contiene(Object)
	 */
	public boolean contieneVertice(Vertice<T> verticeBuscado) {

		return vertices.contiene(verticeBuscado);
	}

	/**
	 * Recorrido en profundidad del grafo.
	 * Su costo es O(|V|) pues recorre todos los vértices.
	 */
	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);
				getRecorridoDFS().agregar(vert);
				recorridoDFS(vert.getAdyacentes());
			}
		}
		return;
	}

	/**
	 * Recorrido en ancho
	 * Su costo es O(|V|) pues recorre todos los vértices.
	 *
	 */
	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);
				getRecorridoBFS().agregar(vert);
			}
```

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

             }

             return;
      }

      public void encontrarCiclos(ListaEnlazada<Vertice<T>> vertices){
             encontrarCiclosWrapper(vertices);
             cargarCiclos();
      }


      /**
       * Encuentra los ciclos en el grafo.
       * Utilizado para determinar sobre que vertices crear los arcos para
       * la robustez.
       *
       * El método para encontrar ciclos en los grafos se basa en el algoritmo de
             Tarjan (aunque no es estrictamente 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 nodo
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.

             En esta implementación el algoritmo se llama, como máximo 2 veces para
cada
             vértice por lo que el orden sería O (2|V|) que es equivalente a O(|V|).
       */
      private void encontrarCiclosWrapper(ListaEnlazada<Vertice<T>> vertices) {

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

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

                    if (!vert.isVisitado()) {
                          vert.setVisitado(true);
                          vert.setIndex(index);
                          vert.setLowLink(index);
                          index++;
                          visitados.add(vert);

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

                          while (iterAdyacente.hasNext()) {
```

```
                                Vertice<T> vertAdyacente = iterAdyacente.next();

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

    private void cargarCiclos(){

        ListaEnlazada<Vertice<T>> noUsados = new ListaEnlazada<Vertice<T>>();

        Vertice<T> verticeAux=visitados.get(0);
        for (Vertice<T> ver : visitados) {
            if (ver.getLowLink() == verticeAux.getIndex()
                        || ver.getIndex() == verticeAux.getIndex()) {
                subset.agregar(ver);
            } else {
                verticeAux=ver;
                if (subset.tamanio() > 2) {
                    ciclosGrafo.agregar(subset);
                } else {
                    noUsados.agregar(ver);
                }
                subset=new ListaEnlazada<Vertice<T>>();
                subset.agregar(ver);
            }
        }

        if (subset.tamanio() > 2) {
            ciclosGrafo.agregar(subset);
        }

        if (noUsados.tamanio() > 0) {
            ciclosGrafo.agregar(noUsados);
        }

    }
}
```

ListaEnlazada.java

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

public class ListaEnlazada<T> {

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

    private Map<T, T> elementos;

    /**
     * Construye la lista enlazada. Inicializa los punteros necesarios para su
utilización.
     * Construye a su vez la colección auxiliar utilizada para buscar los elementos.
     *
```

```
	 * Estas dos operaciones tienen costo O(1) por lo que el costo de inicializar la
colección es O(1).
	 */
	public ListaEnlazada () {

		header.siguiente = header.anterior = header;
		elementos = new HashMap<T, T>();
	}

	/**
	 * Verifica el tamaño de la lista, costo O(1)
	 *
	 * @return true si la lista está vacía
	 */
	public Boolean vacia() {
		return tamanio == 0;
	}

	/**
	 * Obtiene el primer elemento, dado que guardamos una referencia al primer
elemento
	 * el costo de esta operación es O(1).
	 *
	 * @return primer elemento de la lista.
	 */
	public T primero() {
		return header.siguiente.elemento;
	}

	/**
	 * Obtiene el último elemento, dado que guardamos una referencia al último
elemento
	 * el costo de esta operación es O(1).
	 *
	 * @return último elemento de la lista.
	 */
	public T ultimo() {
		return header.anterior.elemento;
	}

	/**
	 * Agrega el elemento al final de la lista, es simplemente una operación entre
punteros
	 * por lo que el costo de esta operación es O(1).
	 *
	 *  Adicionalmente agrega el elemento a la estructrua auxiliar, acorde a la
documentación
	 *  de dicha estructura de datos el costo es constante.
	 *
	 *  Por lo antedicho, el costo de agregar un elemento es O(1).
	 *
	 * @param elemento a agregar en la lista
	 */
	public void agregar(T elemento) {
		agregarAntes(elemento, header);
		elementos.put(elemento, elemento);
	}

	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;
		}

		/**
		 * Comprueba si esta colección contiene o no el datoBuscado.
		 *
		 * El chequeo se hace contra la estructura auxiliar, elementos, esta colección garantiza que
		 * la operación de get se realiza en tiempo constante, por lo que esta operación es O(1).
		 *
		 * @param datoBuscado item que se quiere saber si está o no en la lista
		 * @return true si el datoBuscado está en esta colección.
		 */
		public boolean contiene(T datoBuscado) {

			return this.elementos.containsKey(datoBuscado);
		}

		/**
		 * Devuelve una referencia al dato buscado.
		 *
		 * El chequeo se hace contra la estructura auxiliar, elementos, esta colección garantiza que
		 * la operación de get se realiza en tiempo constante, por lo que esta operación es O(1).
		 *
		 * @param buscado item del que se requiere una referencia que esté en esta lista.
		 * @return una nueva referencia a buscado, que se encuentra en esta lista, null si el item no está en la colección.
		 */
		public T obtener(T buscado) {

			return elementos.get(buscado);
		}

		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;
		}

	}

}
```

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;

public class AumentadorDeRobustez {

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

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

```
	/**
	 * Método encargado de agregar aristas entre distintos vértices para aumentar la robustez del grafo.
	 *
	 * Este algoritmo itera por los ciclos del grafo, en el peor caso (grafo completamente desconectado)
	 * tendremos |V| ciclos, e itera por cada  uno de los vértices de ese ciclo, en el peor caso (grafo con un solo ciclo)
	 * tendremos |V| vértices en un ciclo.
	 *
	 * Para cada vértice de cada ciclo, conecta ese vértice con un vértice del "ciclo siguiente"(*) hasta que agrega
	 * tantos vértices como la robustez pedida, de esta forma aumentamos la robustez entre cada ciclo, al finalizar
	 * habremos aumentado la robustez total del grafo.
	 *
	 * Dado que recorremos todos los ciclos, y para cada ciclo todos los vértices que lo componen, que un vértice esta
	 * en un único ciclo y que lo emparejamos con cada vértice del ciclo siguiente, nos encontramos con que visitamos cada
	 * vértice 2 veces por lo que el costo de esta operación es O(2|V|) = O(|V|).
	 *
	 * (*) A los efectos de este algoritmo consideramos la lista de ciclos como una lista circular por lo que
	 * "ciclo siguiente" puede ser el ciclo que le sigue al actual o el primero en caso de estar en el
	 * último ciclo de la lista.
	 *
	 * @param ciclos
	 * @param robustez
	 */
	@SuppressWarnings({ "rawtypes", "unchecked" })
	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();

				int robustezAlcanzada = 0;

				while (verticesPrimerCiclo.hasNext()) {

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

					while (verticesSegundoCiclo.hasNext()) {

						Vertice verticeCicloDos = verticesSegundoCiclo.next();
```

```
                                        if (puedenEnlazarse(verticeCicloUno,
verticeCicloDos, robustez)) {

                                              grafo.agregarArco(verticeCicloUno,
verticeCicloDos);
                                              aristasAgregadas.agregar(new
Arista(verticeCicloUno, verticeCicloDos));
                                        }

                                        robustezAlcanzada++;
                                  }
                            }

                      }
               }
        }

        @SuppressWarnings({ "rawtypes", "unchecked" })
        private boolean puedenEnlazarse(Vertice verticeoUno, Vertice verticeDos, int
robustez) {

               int gradoVerticeUno = verticeoUno.getGradoVertice();
               int gradoVerticeDos = verticeDos.getGradoVertice();

               return gradoVerticeUno < robustez &&
                      gradoVerticeDos < robustez &&
                         !verticeoUno.getAdyacentes().contiene(verticeDos) &&
                         !verticeoUno.equals(verticeDos);
        }

        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;

public class CargadorDeGrafos {

	private final Grafo<String> grafo;

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

	}

	/**
	 * Método encargado de construir un grafo en base a la información obtenida de un archivo dado.
	 *
	 * Este método recorre todas las lineas del archivo (vértices) y para cada una de ellas recorre
	 * la lista de vértices adyacentes (arcos) por lo que el costo, en tiempo, de cargar un grafo es O(|V| + |A|)
	 * @param reader
	 * @throws IOException
	 */
	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);
	}
}
```