

Grafos: Implementación por Listas de Adyacencia

Abstract: Introducción general: problemas típicos de la implementación matricial y posibles ventajas de la implementación encadenada. Elementos básicos sugeridos para la implementación encadenada. Diseño de clases para la implementación encadenada: jerarquía de arcos y jerarquía de grafos. Modelos de uso y aplicación. Algoritmos Ávidos: Revisión y prueba de validez del algoritmo de Prim para la obtención del Árbol de Expansión Mínimo de un grafo no dirigido. Análisis y sugerencias de implementación con grafos representados por listas de adyacencia.

1.) Introducción.

Un grafo con n nodos podría tener un máximo de n^2 arcos (si no se admiten arcos paralelos, es decir, varios arcos que conecten el mismo par de nodos). En general decimos que un grafo es *denso* si su número de arcos m es $O(n^2)$. Y decimos que el grafo es *disperso* (o *poco denso*) si su número de arcos m es $O(n)$.

Hemos visto la forma de implementar un grafo en forma matricial y hemos puesto de relieve que esa forma de implementación resulta en general sencilla de implementar e intuir, pero es poco adaptable a necesidades eventuales de expandir el grafo si se requiere agregar vértices y arcos no previstos, además de ocupar memoria en forma ineficiente para la representación de los arcos si el grafo es poco denso, ya que la implementación matricial emplea una matriz con tantos casilleros como arcos podría tener el grafo en el peor caso y por lo tanto el espacio de memoria usado es $O(n^2)$. Esto podría ser conveniente si el grafo es denso, pero es claramente una desventaja si el grafo es disperso o muy disperso, ya que la matriz de adyacencias del grafo ocupa demasiado espacio en representar arcos que no existen. Por este motivo, en muchas aplicaciones se busca implementar grafos mediante alguna técnica que favorezca el uso eficiente de la memoria y en ese sentido hay varias maneras posibles de hacerlo usando *listas* en lugar de una matriz.

Cuando el grafo es poco denso o se requiere mayor flexibilidad para agregar vértices y/o arcos, incluyendo la posibilidad de gestionar eventuales arcos paralelos (que serían difíciles de representar en la implementación matricial), entonces se recomienda usar una implementación basada en listas ligadas, que puede asumir distintas formas y estrategias, y que se conoce en general como *implementación de grafos por listas de adyacencia*, o bien, *implementación de grafos por listas multien cadenadas*, o también (más simple...) *implementación encadenada de grafos*.

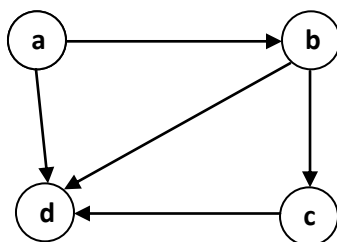
Es obvio que la forma de implementación del grafo puede afectar al rendimiento de un algoritmo dado: por caso, si en la implementación matricial se requiriese procesar todos los arcos de un grafo de n vértices, ese proceso tendría como mínimo un tiempo de $O(n^2)$ (ya que la matriz de adyacencia representa todos los posibles arcos, existan o no en el grafo). Sin embargo, en la implementación encadenada se suele tener una lista separada conteniendo sólo a los arcos que existen en el grafo, y por lo tanto, si el grafo es poco denso y el número de arcos m es $O(n)$, entonces el mismo proceso tendría un tiempo de $O(n)$ para visitar todos los arcos.

2.) Implementación de grafos por listas de adyacencia.

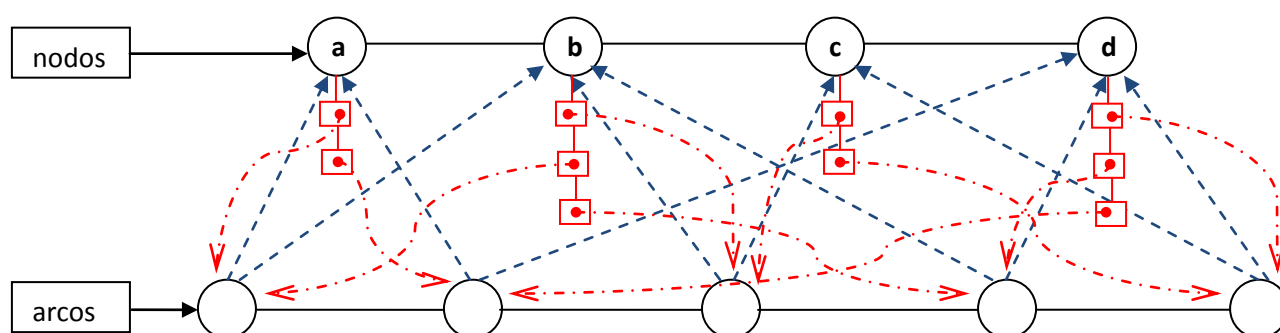
Como dijimos, se pueden pensar muchas estrategias basadas en listas ligadas para implementar un grafo. Desde aquí sugerimos un planteo que permita con cierta comodidad acceder tanto a los vértices como a los arcos del grafo, mediante recorridos de tiempo lineal. Los elementos a usar son¹:

- Una lista que contenga a todos los vértices (por ejemplo, una *LinkedList* o una *ArrayList*).
- Una segunda lista que contenga a todos los arcos.
- Sea *Arc* la clase que representa a los arcos del grafo. Cada objeto de la clase *Arc* contendrá referencias o punteros a los dos vértices unidos por ese arco, además del peso del arco, que por razones de sencillez, se supone único y entero.
- Sea *Node* la clase que representa a los vértices del grafo. Cada objeto de la clase *Node*, contendrá a su vez una lista con referencias o punteros a cada uno de los arcos que tienen a ese nodo como inicial o final (es decir: la lista de arcos de un vértice debe apuntar a todo arco que entre o salga de ese nodo si el grafo es dirigido, o a todo arco que sea incidente al vértice si el grafo es no dirigido).

A modo de ejemplo, mostramos el siguiente grafo dirigido sencillo:



Con los elementos sugeridos más arriba, el grafo de la figura anterior podría implementarse como muestra esta otra figura:



¹ La fuente esencial en la que está basada la implementación de grafos que sigue en esta sección, es el material del curso "Design and Analysis of Algorithms II" – Stanford University (a cargo de *Tim Roughgarden*, Associate Professor): <https://www.coursera.org/courses>.

Si bien la gráfica mostrada es un tanto intimidante, en la práctica simplemente requiere definir correctamente las clases *Node* y *Arc*, y usar estructuras de listas como las ya provistas por Java en el paquete *java.util* (como las citadas *LinkedList* y *ArrayList*).

Es simple ver que con esta implementación, el *espacio de memoria* a emplear para representar un grafo con n vértices y m arcos, será $O(n + m)$ que si el grafo es poco denso resulta preferible comparado con el $O(n^2)$ que insume la implementación matricial.

Uno de los problemas a resolver, es que el grafo podría ser dirigido o no dirigido y eso impacta visiblemente en la forma en que luego se comparan dos arcos para saber si son iguales o no. Si el grafo es dirigido, los arcos X e Y serán iguales sólo si sus pesos son iguales y además se cumple que:

$$X.inicio == Y.inicio \ \&\& \ X.fin == Y.fin$$

donde se asume que *inicio* y *fin* son los vértices de partida y de llegada del arco, respectivamente. Pero si el grafo es no dirigido, entonces los arcos X e Y serán iguales en la misma situación anterior, o bien si los nodos inicial y final de cada arco coinciden en forma cruzada. En otras palabras, para hacer el test de igualdad entre dos arcos en un grafo no dirigido, debe controlarse que coincidan los pesos y además sea cierta la siguiente condición general:

$$(X.inicio == Y.inicio \ \&\& \ X.fin == Y.fin) \ || \ (X.inicio == Y.fin \ \&\& \ X.fin == Y.inicio)$$

El manejo de esta doble lógica para el test de igualdad en el modelo *Grafos [implementación encadenada]* que acompaña a la ficha, se resolvió mediante una jerarquía de clases para representar a los arcos, aplicando una variante liviana del *patrón Strategy*: la clase *Arc* se define **abstracta**, y contiene las referencias a los vértices inicial y final del arco, además del peso:

```
public abstract class Arc <T> implements Comparable< Arc<T> >
{
    protected Node<T> init;
    protected Node<T> end;
    protected int weight;

    public Arc(Node <T> in, Node <T> en)
    {
        this(in, en, 0);
    }

    public Arc(Node in, Node en, int w)
    {
        if(in == null) { throw new NullPointerException( "Error:..." ); }
        if(en == null) { throw new NullPointerException( "Error:..." ); }
        init = in;
        end = en;
        weight = w;
    }

    public int compareTo(Arc<T> o)
    {
        return this.weight - o.weight;
    }

    public Node <T> getInit()
    {
        return init;
    }

    public Node <T> getEnd()
```

```

    {
        return end;
    }

    public int getWeight()
    {
        return weight;
    }

    public void setInit(Node <T> in)
    {
        if( in != null ) { init = in; }
    }

    public void setEnd(Node <T> en)
    {
        if( en != null ) { end = en; }
    }

    public void setWeight(int w)
    {
        weight = w;
    }

    public boolean isSelfLoop()
    {
        return (init.equals(end));
    }

    public int hashCode()
    {
        int hash = 7;
        hash = 53 * hash + Objects.hashCode(this.init);
        hash = 53 * hash + Objects.hashCode(this.end);
        hash = 53 * hash + this.weight;
        return hash;
    }

    public String toString()
    {
        return "("+init.getValue()+", "+end.getValue()+"[" + weight + "])";
    }
}

```

La clase *Arc* no provee ningún método abstracto, pero la idea es que las derivadas se encarguen de redefinir convenientemente el método *equals()* que se hereda desde *Object* (y del cual la clase *Arc* no realiza ninguna implementación). Las derivadas propuestas para *Arc* son dos: *UndirectedArc* y *DirectedArc*, cada una para modelar arcos no dirigidos y arcos dirigidos respectivamente. Ambas subclases son simples ya que ninguna agrega nuevos atributos, y sólo el método *equals()* requiere algo de trabajo:

```

public class UndirectedArc<T> extends Arc<T>
{
    public UndirectedArc(Node<T> in, Node<T> en)
    {
        super(in, en);
    }

    public UndirectedArc(Node in, Node en, int w)
    {
        super(in, en, w);
    }
}

```

```

public boolean equals(Object x)
{
    if(x == null) { return false; }
    if( ! (x instanceof Arc) ) { return false; }

    Arc <T> p = null;
    try
    {
        p = (Arc <T>) x;
    }
    catch(ClassCastException e)
    {
        return false;
    }

    if(this.weight != p.weight) { return false; }
    boolean p1 = p.init.equals(this.init) && p.end.equals(this.end);
    boolean p2 = p.init.equals(this.end) && p.end.equals(this.init);
    return p1 || p2;
}

}

public class DirectedArc<T> extends Arc<T>
{
    public DirectedArc(Node<T> in, Node<T> en)
    {
        super(in, en);
    }

    public DirectedArc(Node in, Node en, int w)
    {
        super(in, en, w);
    }

    public boolean equals(Object x)
    {
        if(x == null) { return false; }
        if( ! (x instanceof Arc) ) { return false; }

        Arc <T> p = null;
        try
        {
            p = (Arc <T>) x;
        }
        catch(ClassCastException e)
        {
            return false;
        }

        if(this.weight != p.weight) { return false; }
        return (p.init.equals(this.init) && p.end.equals(this.end));
    }

}

```

Note que todas estas clases usan generics para hacer control de homogeneidad (la clase parametrizada T indica la clase de los objetos que se podrán almacenar en cada vértice, y al no imponer en ella ninguna restricción de herencia o implementación, entonces cualquier objeto de cualquier clase podrá almacenarse en un vértice del grafo).

Además, observe que la clase *Arc* implementa la interface *Comparable* de forma de hacer que el método *compareTo()* compare los pesos de los arcos. Esto será particularmente útil en algoritmos que

requieran (por ejemplo) buscar el arco de menor peso (*algoritmo de Prim*, usando un heap, para el árbol de expansión mínimo) u ordenar por pesos el conjunto de arcos (*algoritmo de Kruskal* también para el árbol de expansión mínimo). Sin embargo, el método *compareTo()* así planteado no es consistente con el método *equals()*: el primero sólo controla los pesos e indicará que dos arcos son iguales si sus pesos coinciden, pero *equals()* no sólo controla los pesos sino también los vértices de partida y llegada. Por lo tanto, tenga cuidado: si requiere hacer control de igualdad estricta para (por ejemplo) validar que un arco ya exista en el grafo y evitar agregar uno igual, entonces debe usar *equals()* para ese test de igualdad. Sólo use la versión que hemos provisto aquí de *compareTo()* cuando necesite establecer una relación de orden (por menor o por mayor) entre dos arcos.

Los vértices del grafo se representan como objetos de la clase *Node*, que contiene un atributo *value* para almacenar el objeto propio de ese vértice, y una lista conteniendo referencias a los arcos incidentes a cada nodo (tanto los que entran como los que salen de ese nodo):

```
public class Node <T>
{
    private T value;
    private LinkedList < Arc <T> > arcs;

    public Node(T x)
    {
        this(x, null);
    }

    public Node(T x, LinkedList <Arc <T> > a)
    {
        if(x == null) { throw new NullPointerException("Error:..."); }

        value = x;
        if(a == null) { a = new LinkedList <> (); }
        arcs = a;
    }

    public T getValue()
    {
        return value;
    }

    public LinkedList < Arc <T> > getArcs()
    {
        return arcs;
    }

    public void setValue(T x)
    {
        if(x != null) { value = x; }
    }

    public void setArcs(LinkedList < Arc <T> > a)
    {
        if(a != null) { arcs = a; }
    }

    public boolean equals(Object x)
    {
        if(x == null) { return false; }
        if(! (x instanceof Node)) { return false; }
        return ((Node<T>) x).value.equals(this.value);
    }

    public int hashCode()
```

```
{
    return value.hashCode();
}

public String toString()
{
    return value.toString();
}
}
```

En cuanto al propio grafo, aquí también hemos recurrido a una jerarquía de clases con una variante liviana del *patrón State* para modelar con más precisión la diferencia entre grafos dirigidos y no dirigidos: ambos casos pueden tomarse como distintos estados definitivos en los que puede estar un grafo, y la diferencia sutil entre ambos estados es la clase de arcos deben crearse para armar cada grafo: los grafos dirigidos deberán crear arcos de la clase *DirectedArc* mientras que los grafos no dirigidos deberán crear arcos del tipo *UndirectedArc*. En ese sentido, la clase *Graph* es abstracta pero brinda casi todo el comportamiento que sus derivadas necesitarán, incluyendo numerosos métodos sobrecargados para agregar vértices y arcos al grafo:

```
public abstract class Graph <T> implements Cloneable
{
    protected LinkedList < Node <T> > vertices;
    protected LinkedList < Arc <T> > edges;

    // un flag para recordar si el grafo acepta o no arcos paralelos...
    protected boolean allow_parallel_arcs;

    public Graph()
    {
        this(null, null, false);
    }

    public Graph(boolean p)
    {
        this(null, null, p);
    }

    public Graph(LinkedList< Node <T> > v, LinkedList< Arc <T> > a)
    {
        this(v, a, false);
    }

    public Graph(LinkedList<Node <T>> v, LinkedList<Arc <T>> a, boolean p)
    {
        if(v == null) { v = new LinkedList<> (); }
        this.vertices = v;

        if(a == null) { a = new LinkedList<> (); }
        this.edges = a;

        this.allow_parallel_arcs = p;
    }

    public boolean add(T n)
    {
        if(n == null) { return false; }

        Node<T> nt = new Node<>(n);
        return this.add(nt);
    }
}
```

```
public boolean add(Node<T> v)
{
    if(v == null) { return false; }

    if(vertices.contains(v)) { return false; }
    return vertices.add(v);
}

public boolean add(Arc <T> a)
{
    // si el arco es null, salir con false...
    if(a == null) { return false; }

    // si el nodo inicial es null, salir con false...
    Node <T> in = a.getInit();
    if(in == null) { return false; }

    int idxin = vertices.indexOf(in);
    if(idxin == -1) { return false; }

    // si el nodo final es null, salir con false...
    Node <T> en = a.getEnd();
    if(en == null) { return false; }

    int idxen = vertices.indexOf(en);
    if(id xen == -1) { return false; }

    // acceder a las listas de arcos de los vértices inicial y final...
    Node <T> ni = vertices.get(idxin);
    LinkedList < Arc <T> > lni = ni.getArcs();

    Node <T> ne = vertices.get(id xen);
    LinkedList < Arc <T> > lne = ne.getArcs();

    if(this.allow_parallel_arcs || ! this.edges.contains(a))
    {
        lni.add(a);
        lne.add(a);
        this.edges.add(a);
        return true;
    }

    return false;
}

public boolean addArc(T n1, T n2)
{
    if(n1 == null || n2 == null) { return false; }
    return addArc(new Node<>(n1), new Node<>(n2), 0, false);
}

public boolean addArc( Node <T> in, Node <T> en )
{
    return addArc( in, en, 0, false );
}

public boolean addArc(T n1, T n2, int w)
{
    if(n1 == null || n2 == null) { return false; }
    return addArc(new Node<>(n1), new Node<>(n2), w, false);
}

public boolean addArc( Node <T> in, Node <T> en, int w )
```



```
{
    return this.addArc( in, en, w, false );
}

public boolean addArc( T n1, T n2, boolean create )
{
    if(n1 == null || n2 == null) { return false; }
    return addArc(new Node<>(n1), new Node<>(n2), 0, create);
}

public boolean addArc( Node <T> in, Node <T> en, boolean create )
{
    return this.addArc(in, en, 0, create);
}

public boolean addArc(T n1, T n2, int w, boolean create)
{
    if(n1 == null || n2 == null) { return false; }
    return addArc(new Node<>(n1), new Node<>(n2), w, create);
}

public boolean addArc(Node <T> in, Node <T> en, int w, boolean create)
{
    // si alguno de los vértices de entrada es null, salir con false...
    if( in == null || en == null ) { return false; }

    int idxin = vertices.indexOf(in);
    if( idxin == -1 && create == false) { return false; }

    int idxen = vertices.indexOf(en);
    if( idxen == -1 && create == false ) { return false; }

    if(idxin == -1) {this.add(in);}
    else {in = this.vertices.get(idxin);}

    if(id xen == -1) {this.add(en);}
    else {en = this.vertices.get(id xen);}

    Arc <T> arc = createArc(in, en, w);
    return this.add(arc);
}

public boolean allow_Parallel_Arcs()
{
    return this.allow_parallel_arcs;
}

public Object clone() throws CloneNotSupportedException
{
    Graph copy = (Graph) super.clone();

    copy.vertices = new LinkedList <> ();
    for(Node v : this.vertices)
    {
        copy.vertices.add(new Node(v.getValue()));
    }

    copy.edges = new LinkedList<> ();
    for(Arc a : this.edges)
    {
        Node <T> in = new Node(a.getInit().getValue());
        Node <T> en = new Node(a.getEnd().getValue());
        int w = a.getWeight();
    }
}
```

```

        copy.addArc(in, en, w, false);
    }

    return copy;
}

public int countEdges()
{
    return edges.size();
}

public int countNodes()
{
    return vertices.size();
}

public abstract Arc <T> createArc(Node <T> in, Node <T> en, int w);

public Arc <T> getRandomArc()
{
    int m = this.countEdges();
    if(m == 0) { return null; }
    int ri = (int)(Math.random() * m);
    return edges.get(ri);
}

public int grade(int k)
{
    if( k < 0 || k >= vertices.size() ) { return -1; }
    return vertices.get(k).getArcs().size();
}

public String toString()
{
    StringBuilder res = new StringBuilder("");
    for(int i = 0; i < vertices.size(); i++)
    {
        Node n = vertices.get(i);
        res.append("\n\t").append(n.getValue()).append("\t[ ");
        LinkedList < Arc <T> > a = n.getArcs();
        for(int j = 0; j < a.size(); j++)
        {
            Arc <T> e = a.get(j);
            T vi = e.getInit().getValue();
            T ve = e.getEnd().getValue();
            int w = e.getWeight();
            if(!ve.equals(n.getValue())) { res.append(ve); }
            else { res.append(vi); }
            res.append("[").append(w).append("] ");
        }
        res.append("]");
    }
    res.append("\n");
    return res.toString();
}
}

```

La clase contiene un atributo para la lista de vértices y otra para la lista de arcos, como se indicó más arriba. Además, la clase provee un atributo booleano *allow_parallel_arcs* que será usado como un indicador para marcar si el grafo aceptará o no arcos paralelos.

El único método abstracto que provee la clase *Graph* es ***createArc()***, que se deja para las clases derivadas y su única función es crear un arco del tipo que sea correcto para el grafo. Existen otros métodos que oportunamente serán útiles, tales como *clone()* [que crea y retorna una copia clonada del grafo], *getRandomArc()* [que selecciona y retorna aleatoriamente un arco del grafo, sin removerlo], *grade()* [que calcula y retorna el grado de un vértice, o sea, la cantidad de arcos que inciden en ese vértice] y *countEdges()* y *countNodes()* [que respectivamente retornan la cantidad de arcos y la cantidad de vértices del grafo]. El método *toString()* retorna una cadena con una representación adecuada del contenido del grafo.

Esencialmente, todos los métodos que agregan un arco al grafo terminan invocando a la siguiente sobrecarga del método *addArc()*:

```
public boolean addArc(Node <T> in, Node <T> en, int w, boolean create)
{
    if( in == null || en == null ) { return false; }

    int idxin = vertices.indexOf(in);
    if( idxin == -1 && create == false ) { return false; }

    int idxen = vertices.indexOf(en);
    if( idxen == -1 && create == false ) { return false; }

    if(idxin == -1) {this.add(in);}
    else {in = this.vertices.get(idxin);}

    if(idxen == -1) {this.add(en);}
    else {en = this.vertices.get(idxen);}

    Arc <T> arc = createArc(in, en, w);
    return this.add(arc);
}
```

Este método toma como parámetro los dos vértices a unir por el arco, el peso del arco y un flag para indicar si los vértices deben crearse en caso de no existir previamente. Al comenzar, realiza controles para verificar que los vértices de entrada no sean *null* o no estén ya ingresados al grafo. Si todo va bien, la anteúltima línea del método invoca al método *createArc()* para que se cree el arco del tipo que corresponda, y ese arco será finalmente agregado al grafo por el método *add(Arc)* que también está en la clase:

```
public boolean add(Arc <T> a)
{
    if(a == null) { return false; }
    Node <T> in = a.getInit();
    if(in == null) { return false; }

    int idxin = vertices.indexOf(in);
    if(idxin == -1) { return false; }

    Node <T> en = a.getEnd();
    if(en == null) { return false; }

    int idxen = vertices.indexOf(en);
    if(idxen == -1) { return false; }

    Node <T> ni = vertices.get(idxin);
    LinkedList < Arc <T> > lni = ni.getArcs();

    Node <T> ne = vertices.get(idxen);
    LinkedList < Arc <T> > lne = ne.getArcs();
}
```

```
        if(this.allow_parallel_arcs || ! this.edges.contains(a))
        {
            lni.add(a);
            lne.add(a);
            this.edges.add(a);
            return true;
        }

        return false;
    }
}
```

Este método toma el arco que se propone para agregar al grafo, y lo agrega si se cumplen ciertas condiciones: si el grafo permite arcos paralelos, lo agregará incluso si ya había un arco igual en el grafo, pero si el grafo no admite arcos paralelos, lo agregará sólo si ese arco no existía ya en la lista de arcos del grafo.

Las clases derivadas de *Graph* son dos: *DirectedGraph* y *UndirectedGraph*. Ambas son clases concretas y su implementación resulta directa:

```
public class DirectedGraph<T> extends Graph<T>
{
    public DirectedGraph()
    {
    }

    public DirectedGraph(boolean p)
    {
        super(p);
    }

    public DirectedGraph(LinkedList< Node<T> > v, LinkedList< Arc<T> > a)
    {
        super(v, a);
    }

    public DirectedGraph(LinkedList<Node<T>> v, LinkedList<Arc<T>> a, boolean p)
    {
        super(v, a, p);
    }

    public Arc<T> createArc(Node <T> in, Node <T> en, int w)
    {
        return new DirectedArc(in, en, w);
    }

    public String toString()
    {
        StringBuilder res = new StringBuilder("[");
        for(int i = 0; i < vertices.size(); i++)
        {
            Node n = vertices.get(i);
            res.append("\n\t").append(n.getValue()).append(":\t[ ");
            LinkedList < Arc <T> > a = n.getArcs();
            for(int j = 0; j < a.size(); j++)
            {
                Arc <T> e = a.get(j);
                T vi = e.getInit().getValue();
                T ve = e.getEnd().getValue();
                int w = e.getWeight();
                if(vi.equals(n.getValue()))
                {
                    res.append(ve);
                    res.append("[").append(w).append("] ");
                }
                //res.append(" ");
            }
        }
    }
}
```

```

        }
        res.append("]");
    }
    res.append("\n");
    return res.toString();
}

public class UndirectedGraph<T> extends Graph<T>
{
    public UndirectedGraph()
    {
    }

    public UndirectedGraph(boolean p)
    {
        super(p);
    }

    public UndirectedGraph(LinkedList< Node<T> > v, LinkedList< Arc<T> > a)
    {
        super(v, a);
    }

    public UndirectedGraph(LinkedList<Node<T>> v, LinkedList<Arc<T>> a, boolean p)
    {
        super(v, a, p);
    }

    public Arc<T> createArc(Node <T> in, Node <T> en, int w)
    {
        return new UndirectedArc(in, en, w);
    }
}

```

En ninguna de ambas clases se agrega atributo alguno, y ambas implementan el método *createArc()* de forma de retornar un arco dirigido o un arco no dirigido según sea la clase que representa al grafo. Sólo la clase *DirectedGraph* realiza algo más, como es redefinir el método *toString()* para armar la cadena de salida en forma más acorde al hecho de que los arcos del grafo son dirigidos.

Finalmente, la clase *Principal* incluida en el modelo, muestra un método *main()* para testear en forma simple las clases aquí presentadas:

```

public class Principal
{
    public static void main(String args[])
    {
        UndirectedGraph <String> ug1 = new UndirectedGraph<>();
        ug1.add("a");
        ug1.add("b");
        ug1.add("c");
        ug1.add("d");
        ug1.addArc("a", "b", 2);
        ug1.addArc("a", "b", 2); // paralelo: no debe permitirlo...
        ug1.addArc("b", "c", 3);
        ug1.addArc("b", "d", 1);
        ug1.addArc("d", "c", 4);
        System.out.println("Grafo 1 (no dirigido - sin arcos paralelos: ");
        System.out.println(ug1);
        System.out.println();

        UndirectedGraph <String> ug2 = new UndirectedGraph<>(true);
        ug2.add("a");
        ug2.add("b");
        ug2.add("c");
        ug2.add("d");
    }
}

```

```
ug2.addArc("a", "b", 2);
ug2.addArc("a", "b", 2); // paralelo: debe permitirlo...
ug2.addArc("b", "c", 3);
ug2.addArc("b", "d", 1);
ug2.addArc("d", "c", 4);
System.out.println("Grafo 2 (no dirigido - con arcos paralelos: ");
System.out.println(ug2);
System.out.println();

DirectedGraph <String> dg1 = new DirectedGraph<>();
dg1.add("a");
dg1.add("b");
dg1.add("c");
dg1.add("d");
dg1.addArc("a", "b", 2);
dg1.addArc("a", "b", 2); // paralelo: no debe permitirlo...
dg1.addArc("c", "b", 3);
dg1.addArc("b", "d", 1);
dg1.addArc("d", "c", 4);
System.out.println("Grafo 3 (dirigido - sin arcos paralelos: ");
System.out.println(dg1);
System.out.println();

DirectedGraph <String> dg2 = new DirectedGraph<>(true);
dg2.add("a");
dg2.add("b");
dg2.add("c");
dg2.add("d");
dg2.addArc("a", "b", 2);
dg2.addArc("a", "b", 2); // paralelo: debe permitirlo...
dg2.addArc("c", "b", 3);
dg2.addArc("b", "d", 1);
dg2.addArc("d", "c", 4);
System.out.println("Grafo 4 (dirigido - con arcos paralelos: ");
System.out.println(dg2);

DirectedGraph <String> dg3 = null;
try
{
    dg3 = (DirectedGraph<String>)dg2.clone();
}
catch(CloneNotSupportedException e)
{
}
System.out.println("Grafo 5 (dirigido - con arcos paralelos [clonado]: ");
System.out.println(dg3);
}
}
```

En el modelo *Grafos [implementación encadenada]* que acompaña a esta ficha, se muestra una implementación detallada de las ideas discutidas aquí. Dejamos el análisis profundo del código fuente para los alumnos, así como cualquier eventual agregado o modificación.

3.) Algoritmos Ávidos: Algoritmo de Prim - Árbol de Expansión Mínimo (Prueba de Validez)

Hemos presentado en una ficha anterior el problema del *Árbol de Expansión Mínimo (AEM)* para un *Grafo no dirigido, conexo, denso y ponderado*. El propio algoritmo se mostró implementado en un contexto de grafos representados en forma matricial, y ahora se discutirán brevemente algunas ideas para implementarlo en grafos modelados con listas de adyacencia, entre otros conceptos.

Por lo pronto, observemos que el *algoritmo de Prim* es un *algoritmo ávido*: se basa en la repetición sistemática de una regla local, con la esperanza de llegar finalmente a un resultado global correcto. En el caso del algoritmo de Prim, la regla ávida es la *propiedad de arco de cruce de peso mínimo*, que

indica que si se tiene dos subgrafos G_1 y G_2 de un grafo G conexo y ponderado, entonces el *arco de cruce con menor peso* (esto es, el arco con un vértice en G_1 y el otro en G_2 que tenga el menor peso) pertenece a un AEM para G . El algoritmo completo consiste en partir de un subconjunto G_1 que contenga a sólo un vértice x de G , tomar el arco de cruce con menor peso entre los que vinculan a x con el resto de los vértices, pasar el segundo vértice de este arco a G_1 y repetir la regla, una y otra vez, hasta que G_1 contenga a todos los vértices de G .

Sin embargo, hemos visto también que cuando se trata de algoritmos ávidos es estrictamente necesario demostrar que la aplicación de regla ávida local efectivamente permitirá llegar siempre a un resultado final correcto. En el caso del algoritmo de Prim, la demostración de su validez es relativamente simple²:

- ✓ **Prueba de Validez del Algoritmo de Prim:** queremos probar que el *Algoritmo de Prim* siempre obtiene un AEM para el grafo $G = \{V, A\}$ de partida, sabiendo que G es no dirigido, conexo y ponderado, con $V =$ el conjunto de n vértices de G y $A =$ el conjunto de m arcos de G .

Demostración:

Dados dos subgrafos G_1 y G_2 de G , en cada iteración del algoritmo de Prim se debe encontrar un arco de cruce $a(u, v)$ que permita conectar un vértice u del subgrafo G_1 con otro vértice v en el subgrafo G_2 . Estamos seguros que podremos encontrar al menos un arco de cruce $a(u, v)$ debido a que G es conexo y esto implica que siempre existirá un camino que conecte a u con v .

Por lo pronto, podemos probar rápidamente que la salida T del algoritmo de Prim es un *árbol de expansión para G* (esto es: un grafo que contiene a todos los vértices de G , sin ciclos y con todos sus vértices conectados, aunque aún no imponemos que sea *mínimo*) debido a los siguientes hechos:

- a.) Todos los vértices de T están conectados, ya que por definición los arcos y los vértices que se van agregando a T en cada pasada están conectados en G , y T contiene a todos los vértices de G ya que el propio proceso de construcción sólo finaliza cuando T llega a contener a todos los vértices de G .
- b.) El agregado de un arco $a(u, v)$ en T nunca produce un ciclo: en efecto, si T contiene a los vértices que ya se tomaron para el AEM parcial, entonces el conjunto $X = \{V - T\}$ contiene a los vértices restantes que aún no entraron al AEM parcial. Suponga que se toma el arco de cruce $a(u, v)$ para agregarlo al AEM parcial, con $u \in T$ y $v \in X$. Entonces $a(u, v)$ es el primer arco de cruce encontrado que une a u con v , pues de otro modo se habría encontrado antes a otro arco $a'(u, v)$ y v ya estaría en T . Como $a(u, v)$ es el primero que une a u con v , al agregarlo al AEM parcial estamos seguros que no había otro arco $a'(u, v)$ en el AEM parcial y por lo tanto no se produce un ciclo.

Ahora intentemos probar que el árbol de expansión T producido por el algoritmo de Prim es *mínimo en cuanto a la suma de sus pesos*. Sea T_1 un árbol de expansión mínimo (AEM) para G . Pueden ocurrir dos situaciones:

- 1.) Si $T = T_1$ entonces T es un árbol de expansión mínimo para G , ya que T_1 lo es.

² La fuente en la que se inspira la demostración que sigue, está tomada de Wikipedia (de la que recomendamos la versión en inglés): http://en.wikipedia.org/wiki/Prim%27s_algorithm. Asimismo, la idea usada para probar que el agregado de un arco al AEM parcial nunca produce un ciclo, está tomada del material del curso "Design and Analysis of Algorithms II" – Stanford University (Tim Roughgarden, Associate Professor): <https://www.coursera.org/courses>.

- 2.) Si $T \neq T1$, sea $e(u, v)$ el primer arco agregado durante la construcción de T , tal que $e(u, v)$ no está en $T1$ y sea P el conjunto de vértices conectados por los arcos ya agregados al AEM parcial antes que e . Entonces un vértice de e está en P y el otro no. Ya que $T1$ es un AEM de G , entonces hay un camino en $T1$ que une esos dos vértices. Si nos desplazamos por ese camino se debe encontrar un arco f que une a un vértice en P con uno que no está en P . Entonces en la iteración que e se agrega a P , también se podría haber agregado f , y se hubiese agregado en vez de e si el peso de f [denotado como $w(f)$] fuera menor que peso el de e [denotado como $w(e)$]. Ya que f no se agregó se concluye que $w(f) \geq w(e)$. Si ahora consideramos $T2$ como el grafo que se obtiene al remover f de $T1$ y agregar e , podemos ver que $T2$ sigue siendo conexo, tiene la misma cantidad de arcos que $T1$, y la suma total de los pesos de sus arcos no es mayor que la suma de pesos de $T1$. Por lo tanto $T2$ es también un AEM para G .

Esto quiere decir que la salida T obtenida por la aplicación repetida del proceso local de tomar cada vez el arco de cruce de menor peso, siempre será finalmente un AEM para G y se demuestra así la validez del algoritmo de Prim.

4.) Algoritmo de Prim: Detalles de Implementación³.

Cuando se presentó el modelo de Grafos implementados en forma matricial, se incluyó un método que permitía calcular un árbol de expansión mínimo de un grafo con n vértices. Dado que la matriz de adyacencias usada para representar los arcos tenía n^2 casilleros (uno por cada posible arco, exista o no) y considerando que el algoritmo de Prim esencialmente procesa los m arcos del grafo, entonces el tiempo de ejecución del algoritmo presentado (en el peor caso, en que $m = n^2$) era $O(m) = O(n^2)$.

Si el grafo se representa en forma de listas de adyacencia, entonces los m arcos quedan disponibles en una lista ligada, sin usar espacio extra para arcos que no existen, y por lo tanto, si se utiliza un heap para encontrar el arco de menor peso en tiempo logarítmico, el tiempo de ejecución para el peor caso puede reducirse a

$$O(m * \log(m)) = O(m * \log(n^2)) = O(m * 2 \log(n)) = O(m * \log(n))$$

El uso de un heap para ir recuperando en forma rápida el siguiente arco de menor peso hace más veloz el proceso, al costo de utilizar espacio extra para el propio heap. La idea es comenzar con un subconjunto $V1$ del grafo G , tal que $V1$ contenga sólo a uno de los vértices del grafo, y otro subconjunto $V2$ que contenga al resto de los vértices. Luego tomar el arco A de cruce entre $V1$ y $V2$ que tenga el menor peso, y pasar a $V1$ el vértice de A que estuviese en $V2$. Continuar aplicando esa regla ávida hasta que $V1$ contenga a todos los vértices del grafo original.

Un esquema en pseudocódigo para un algoritmo que calcule un AEM y retorne la suma de sus pesos, sería:

Entrada: un grafo no dirigido, ponderado, conexo y denso $G = \{V, A\}$ donde V es el conjunto de n vértices y A es el conjunto de m arcos $a = (x, y, w)$ (tales que $x, y \in V$ y w es el peso del arco a).

Salida: la suma de pesos s de los arcos de un árbol de expansión mínimo (AEM) para G .

Algoritmo esencial:

- Sea una lista de vértices $V1$ con un vértice cualquiera x de V : $V1 = [v]$.
- Sea $s = 0$.
- Sea una lista de arcos T vacía, para los arcos del AEM parcial: $T = []$.
- Mientras $V1 \neq V$:
 - Tome el arco $a(u, v, w)$ **de menor peso w** , con $(u \in V1)$ y $(v \notin V1)$.
 - Agregue el arco $a(u, v, w)$ a la lista T .
 - Sume w a s .
 - Agregue el vértice v al conjunto $V1$.
- Retorne la suma de pesos s de los arcos en T .

Está claro que el paso crítico en este algoritmo esencial, es la **recuperación del arco $a(u, v, w)$ con menor peso** entre todos los arcos de cruce. Si la búsqueda del arco de menor peso se hace en forma ingenua, comparando contra todos los arcos por fuerza bruta, entonces el algoritmo tendrá un tiempo de ejecución $O(m * n)$. Si el grafo no es muy grande un algoritmo de fuerza bruta podría obtener el resultado buscado en un tiempo aceptable. Pero como se sugirió, una mejor implementación consiste en usar un *heap* para almacenar los arcos de cruce, de forma que ese *heap* se ordene de acuerdo a los pesos de los arcos y obtener así el arco de menor peso en tiempo logarítmico en cada iteración del ciclo. Usando correctamente un *heap*, el tiempo de ejecución total del *algoritmo de Prim* baja (como vimos) a $O(m * \log(n))$ [aunque al costo de usar memoria extra para el *heap*].

Observe que dado que los pesos de los arcos pueden ser negativos, entonces la suma de pesos del AEM podría ser cualquier número entero *negativo, cero o positivo*.

Se deja para los alumnos la tarea de programar estas ideas e implementar una versión rápida del *algoritmo de Prim*.

³ La fuente en la que está basado el algoritmo que sigue, es el material del curso "*Design and Analysis of Algorithms II*" – Stanford University (a cargo de *Tim Roughgarden*, Associate Professor): <https://www.coursera.org/courses>.