Introducción a la Programación

Tema 13. Colecciones II: Grafos

1. Grafos en el Mundo Real

Existen múltiples y diversos ejemplos de grafos en el mundo real. El problema de los puentes de Königsberg, también llamado más específicamente problema de los siete puentes de Königsberg, que se muestra en la Fig. 1, es un célebre problema matemático resuelto por Leonhard Euler en 1736 y cuya resolución dio origen a la teoría de grafos. Su nombre se debe a Königsberg, el antiguo nombre que recibía la ciudad rusa de Kaliningrado. Esta ciudad es atravesada por el río Pregolya, el cual se bifurca para rodear con sus brazos a la isla Kneiphof, dividiendo el terreno en cuatro regiones distintas, las que entonces estaban unidas mediante siete puentes. El problema fue formulado en el siglo XVIII y consistía en encontrar un recorrido para cruzar a pie toda la ciudad, pasando sólo una vez por cada uno de los puentes, y regresando al mismo punto de inicio.

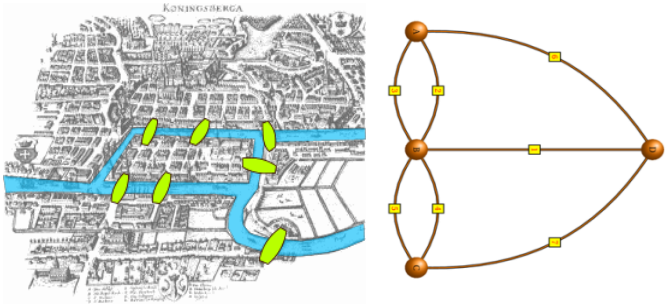


Fig. 1 Ejemplo de los puentes de Königsberg

Kirchhoff, en 1847, con el fin de estudiar el cálculo de la intensidad y la diferencia de potencial de cada elemento de una red eléctrica, entre los cuales se encuentran: resistencias, bobinas, condensadores, fuente de tensión, etc., estudió los grafos conexos con el objetivo de desarrollar un método efectivo para el análisis de estas redes eléctricas.

Los grafos también se usan en el mundo real para modelar el comportamiento. Se denomina máquina de estados a un modelo de comportamiento de un sistema con entradas y salidas (Fig. 2), en donde las salidas dependen no sólo de las señales de entradas actuales sino también de las anteriores. Las máquinas de estados se definen como un conjunto de estados que sirve de intermediario en esta relación de entradas y salidas, haciendo que el historial de señales de entrada determine, para cada instante, un estado para la máquina, de forma tal que la salida depende únicamente del estado y las entradas actuales.

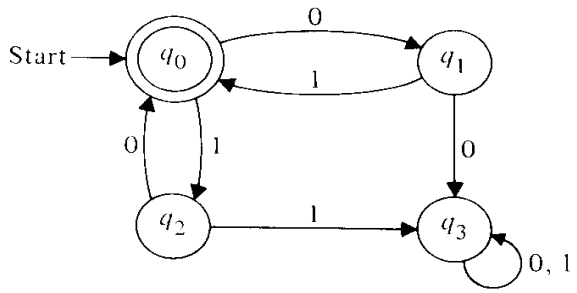


Fig. 2 Ejemplo de diagrama de estados

Otra utilización de los grafos es definir la dependencia entre diferentes entidades. Por ejemplo, tal y como se muestra en la Fig. 3 los diagramas UML son un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema. La mayoría de ellos son grafos.

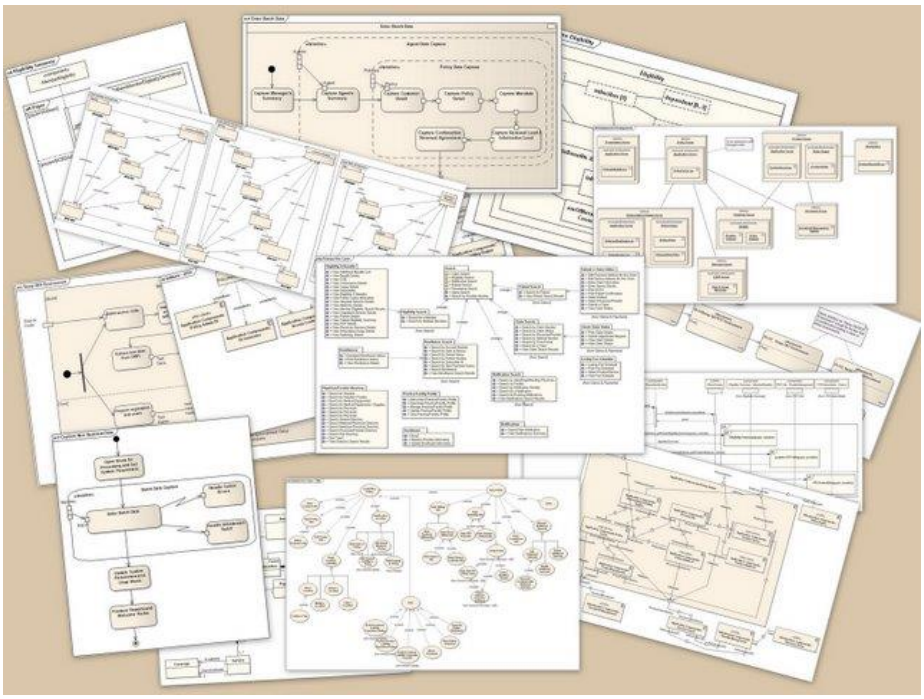
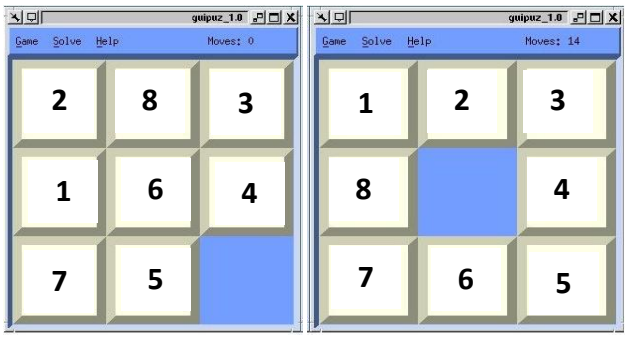


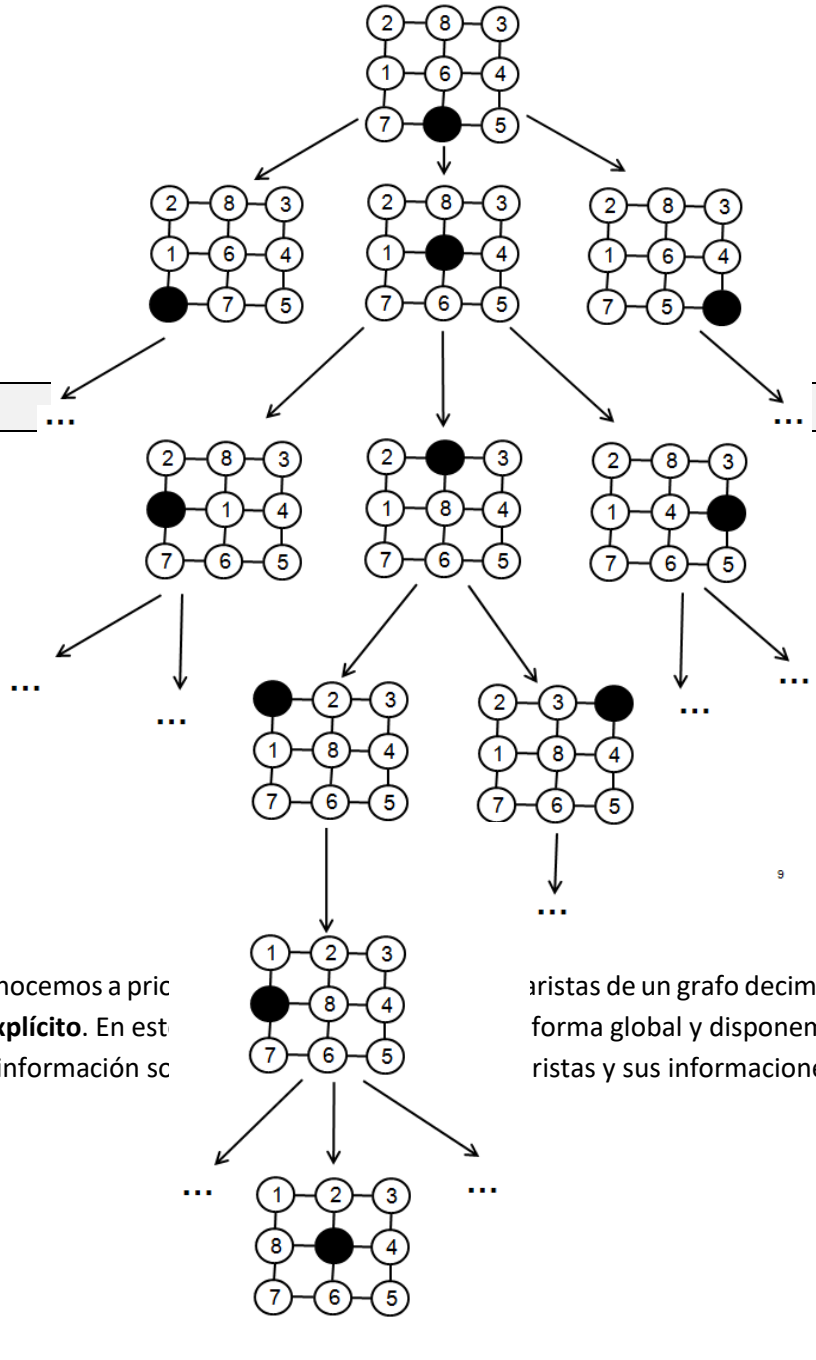
Fig. 3 Diagramas UML

Las técnicas de solución de problemas en Inteligencia Artificial, en general, incorporan el uso de grafos. Por ejemplo todo proceso de búsqueda puede ser visualizado como el recorrido por un

grafo en el que cada nodo representa un estado y cada rama representa las relaciones entre los estados cuyos nodos conecta.

El N-puzle es un problema que se puede modelar con grafos para resolverlo. Como se observa en la Fig. 4 se trata de un puzle que consiste en un marco que contiene piezas cuadradas y numeradas en orden aleatorio, con una ficha que falta. El objetivo del puzle es colocar los cuadros en orden (ver la figura), haciendo movimientos deslizantes que utilizan el espacio vacío. El n-puzle es un problema clásico para el modelado de algoritmos heurísticos.





2. Grafos

El tipo **gra** …mados aristas o arcos, q… conjunto. Es posible qu… …értice consigo mismo. Ta… …nación propia. Denomina…

Las aristas … …imeras tienen una orient… …das no tienen orientació… tienen aristas dirigidas s… …irigidas **grafos mixtos**.

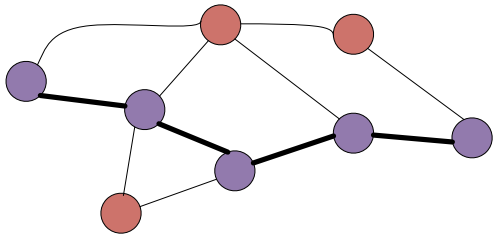
Cuando conocemos a pric… …aristas de un grafo decimos que tenemos un **grafo explícito**. En est… forma global y disponemos en memoria de toda la información sc… …ristas y sus informaciones asociadas.

Hay otros grafos, que llamaremos **grafos implícitos**, de los que no conocemos a priori todos sus vértices y aristas. Este tipo de grafos están descritos por las propiedades que cumplen los vértices y por una función que dado un vértice nos calcula sus vecinos y las aristas que lo conectan a ellos. El grafo estará constituido por el conjunto de vértices alcanzables desde el vértice dado. Los vértices del grafo se irán descubriendo a medida que el grafo se vaya recorriendo.

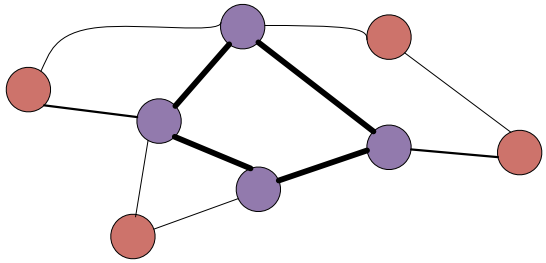
2.1 Términos de un grafo

Veamos, en primer lugar, un conjunto de términos sobre grafos:

- **Camino**: Secuencia de vértices donde cada uno está conectado con el siguiente por una arista.

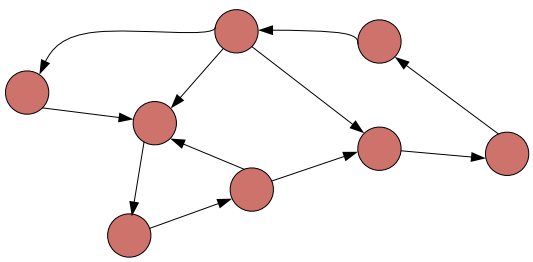


- **Longitud de un Camino**: Se llama longitud del camino al número de aristas que contiene.
- **Camino Cerrado**: Camino donde el vértice primero coincide con el último.



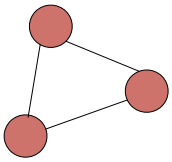
- **Camino simple:** No tiene vértices repetidos a excepción del primero que puede ser igual al último.
- **Bucle:** Arista que une vértice consigo mismo.

- **Ciclo:** Camino cerrado donde no hay vértices repetidos salvo el primero que coincide con el último.
- **Grafos Dirigidos**: Si todas sus aristas son dirigidas.

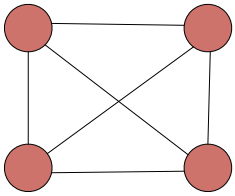


- **Grafos No Dirigidos**: Si todas sus aristas son no dirigidas.

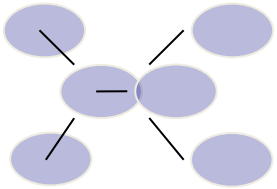
- **Grafos Mixtos**: Puede haber aristas dirigidas y no dirigidas
- **Grafos simples**: Desde un vértice v1 hasta v2 hay como máximo una arista que puede ser dirigida o no dirigida. No pueden contener bucles.



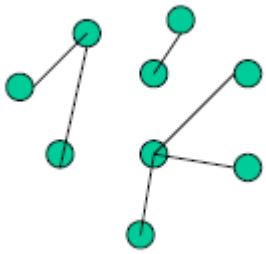
- **Multigrafos**: Son grafos donde puede haber más de una arista entre dos vértices.
- **Pseudografos**: Son grafos donde puede haber más de una arista entre dos vértices y bucles.
- **Grafo Completo**: Grafo simple en el que hay un arista entre cada par de vértices.



- **Grafo Acíclico:** Grafo que no tiene ciclos**.**
- **Grafo Conexo:** Grafo no dirigido en el que hay al menos un camino entre cada dos nodos.
- **Grafo Débilmente Conexo:** Grafo dirigido en el cual al remplazar cada arista dirigida por una no dirigida resulta un grafo no dirigido conexo.
- **Grafo Fuertemente Conexo:** Grafo dirigido dónde para cada par de vértices *u, v* existe un camino dirigido desde *u* a *v* y otro de *v* hasta *u*.
- **Árbol Libre:** Es un grafo simple conexo y sin ciclos.



- **Bosque:** Es un grafo sin ciclos. Es, también, un conjunto de árboles libres.



- **Subgrafo:** Un grafo g1 es subgrafo de otro g2 si los vértices y aristas de g1 están incluidos en los vértices y aristas de g2.
- **Componentes Conexas:** Subgrafos maximales de un grafo no dirigido formados por los vértices entre los que hay al menos un camino, más las aristas que los conectan.
- Una **componente fuertemente conexa** de un grafo dirigido es un subgrafo maximal fuertemente conexo de un grafo dirigido.

- Una **componente débilmente conexa** es un subgrafo maximal débilmente conexo de un grafo dirigido.

Para todos estos tipos nos basaremos en *JGraphT*, http://jgrapht.org/

2.2 Tipo Graph

Como para otros tipos de agregados de datos hablaremos de grafos explícitos y grafos implícitos o grafos virtuales. Los agregados explícitos se construyen agregando y/o eliminando elementos del mismo. En los agregados virtuales solo se describen predicados que indican si un elemento pertenece al agregado o no.

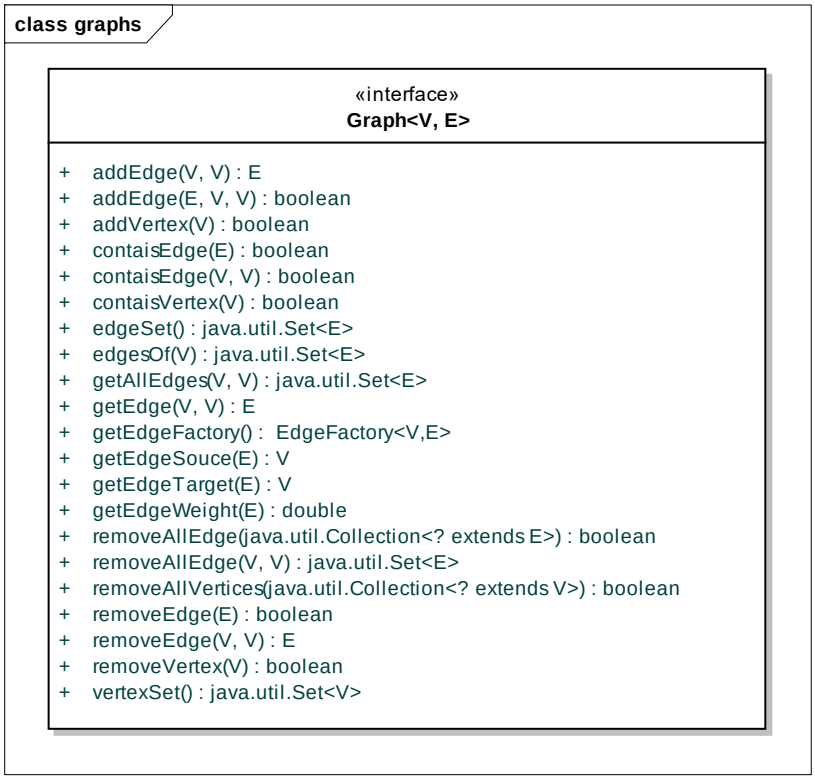
Para modelar un **grafo explícito** (**Graph**) necesitamos conocer su conjunto de vértices y su conjunto de aristas. Asumimos que cada vértice es de tipo V y cada arista de tipo E. El grafo entonces es de tipo ***Graph<V,E>.*** Usaremos la interfaz y las herramientas proporcionadas por **jGrapht** que pueden ser conseguidas en:

http://jgrapht.org/

El API que ofrece puede consultarse en:

http://jgrapht.org/javadoc/

Una interfaz que abarca los diferentes tipos de grafos es:



Interfaz de Graph

Los invariantes para todos los tipos de grafos son:

- Los extremos de una arista deben estar incluidos en el conjunto de vértices.
- Todos los vértices del grafo deben ser distintos entre sí en el sentido definido por *equals* para el tipo *V*.
- Todas las aristas del grafo deben ser distintas entre sí en el sentido definido por *equals* para el tipo *E*.

La funcionalidad ofrecida es la que se muestra en la siguiente tabla:

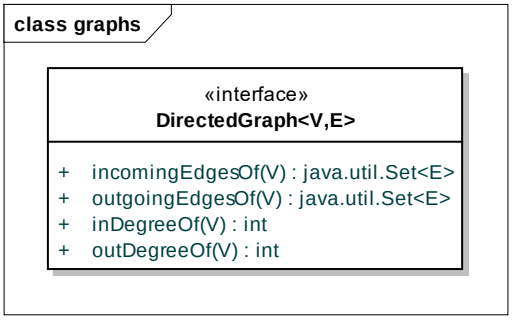
Graph<V,E>	
Invariante: Los citados anteriormente.	
addEdge(V sourceVertex, V targetVertex)	Crea una arista desde v1 hasta v2 y la añade. Devuelve la arista es creada si ha podido ser añadida o null en otro caso. La arista es creada por la factoria de aristas del grafo.
addEdge(V sv, V tv, E e)	Añade la arista e desde sv hasta tv. Devuelve verdadero si el grafo ha cambiado.
addVertex(V v)	Añade el vértice v. Devuelve verdadero si el grafo ha cambiado.
containsEdge(E e)	Devuelve verdadero si la arista e está contenida en el grafo
containsEdge(V sv, V tv)	Devuelve verdadero si existe alguna arista desde sv a tv
containsVertex(V v)	Devuelve verdadero si el vértice v está contenido en el grafo
edgeSet()	Devuelve una vista del conjunto de aristas del grafo

edgesOf(V v)	Devuelve el conjunto de aristas del grafo que tocan el vértice vertex
getAllEdges(V sv, V tv)	Devuelve el conjunto de aristas del grafo que conectan ambos vértices si existen o null si alguno no existe
getEdge(V sv, V tv)	Una de las aristas que van desde sv hasta tv si existe o null
getEdgeFactory()	Devuelve la factoria que sirve para crear las aristas del grafo.
getEdgeSource(E e)	Devuelve el vértice origen de la arista
getEdgeTarget(E e)	Devuelve el vértice destino de la arista
gedEdgeWeight(E e)	Devuelve el peso asignado a la arista
removeAllEdges(Collection< ? extends E> edges)	Elimina todas las aristas que se encuentran en la colección de entrada. Devuelve verdadero si el grafo ha cambiado
removeAllEdges(V sv, V tv)	Elimina todas las aristas que conectan el vértice sv al tv. Devuelve las antiguas aristas que conectaban sv a tv.
removeAllEdges(V sv, V tv)	Elimina todas los vertices que se encuentran el conjunto de entrada
removeEdge(E e)	Elimina la arista pasada como entrada
removeAllVertices(Collection<? extends V> vertices)	Elimina todos los vértices que se encuentran en la colección de entrada y las aristas incidentes. Devuelve verdadero si el grafo ha cambiado
removeVertex(V v)	Elimina el vértice v. Devuelve verdadero si el grafo ha cambiado. Elimina también todas las aristas incidentes en ese vértice
vertexSet()	Devuelve una vista del conjunto de vértices del grafo

Propiedades específicas de Graph

2.3 Tipo DirectedGraph

Los grafos dirigidos son aquellos en los que todas las aristas están dirigidas. Hereda todos los métodos de la interfaz *Graph* y ofrece cuatro nuevos métodos. En la siguiente figura se muestra su interfaz.



Interfaz de DirectedGraph

En la siguiente tabla se describen los métodos que añade.

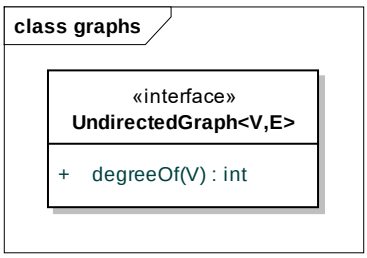
DirectedGraph<V,E> extends **Graph<V,E>**	
incomingEdgesOf (V v)	Devuelve el conjunto de aristas que llegan al vértice v.
inDegreeOf(V v)	Devuelve el número de aristas que llegan al vértice v.

outDegreeOf(V v)	Devuelve el número de aristas que salen del vértice v.
outgoingEdgesOf (V v)	Devuelve el conjunto de aristas que salen del vértice v.

Propiedades específicas de DirectedGraph respecto a Graph

2.4 Tipo UndirectedGraph

El tipo UndirectedGraph es un subtipo de *Graph* que solo contiene aristas no dirigidas. Añade un único método a los que hereda de Graph.



Interfaz de DirectedGraph

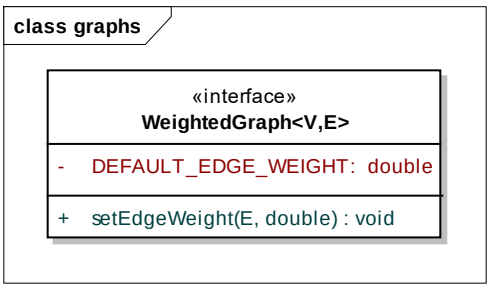
En la siguiente tabla se describen los métodos que añade.

UndirectedGraph<V,E> extends **Graph<V,E>**	
degreeOf (V v)	Devuelve el grado del vértice v. Es decier el número de aristas que lo tocan

Tabla 34. Propiedades específicas de UndirectedGraph respecto a Graph

2.5 Tipo WeightedGraph

El tipo WeightedGraph es el grafo que representa el grafo ponderado. Se caracteriza por tener un peso asociado a cada una de las aristas del grafo. En la figura siguiente se muestra su interfaz.



Interfaz de *WeightedGraph*

Esta interfaz añade un atributo y un método a los que ya hereda de *Graph* con la semántica que se presenta en la tabla 35.

WeightedGraph<V,E> extends **Graph<V,E>**	
DEFAULT_EDGE_WEIGHT	Recoge el valor que por defecto se asignará como peso a las aristas
setEdgeWeight(E e, double peso)	Permite actualizar el peso de la arista e con el valor del peso que se le pasa como entrada.

Tabla 35. Propiedades específicas de *WeigthedGraph* respecto a *Graph*

2.6 Grafos que pueden ser escuchados: ListenableGraph<V,E>

En muchos casos es muy importante disponer de un mecanismo que nos permita informar a un objeto de la actividad que está habiendo en un grafo. Posiblemente para mostrarlo gráficamente o para llevar a cabo algún cálculo adicional. Este mecanismo podemos implementarlo mediante el patrón de diseño observador-observable. Es decir uno o varios observadores (**listeners**) seguirán la actividad de un objeto observado (**listenable**). El objeto observable emitirá eventos según su actividad. Los objetos observadores recibirán los eventos y harán lo que corresponda según el evento recibido. Cuando un observador tiene interés en seguir la una actividad se añade como oyente al objeto observado.

Los grafos que pueden ser observados ofrecen el tipo *ListenableGraph*. Los objetos que están interesados en observar un grafo ofrecen el tipo *VertexSetListener<V,E>* (para oir cuando se añade o elimina un vértice) o *GraphListener<V,E>* (que además de oir cuando se añade o elimina un vértice también oye cuando se añade o elimina una arista).

ListenableGraph<V,E>	
void addGraphListener(GraphListener<V,E> l)	Añade el oyente especificado
void addVertexSetListener(VertexSetListener<V> l)	Añade un oyente especificado
void removeGraphListener(GraphListener<V,E> l)	Elimina el oyente especificado
void removeVertexSetListener(VertexSetListener<V> l)	Elimina el oyente especificado

VertexSetListener<V,E>	
void vertexAdded(GraphVertexChangeEvent<V> e)	Se ha añadido un vértice
void vertexRemoved(GraphVertexChangeEvent<V> e)	Se ha eliminado un vértice

GraphListener<V,E> extends VertexSetListener<V,E>	
void edgeAdded(GraphEdgeChangeEvent<V,E> e)	Se ha añadido un arista
void edgeRemoved(GraphEdgeChangeEvent<V,E> e)	Se ha eliminado una arista

Los eventos emitidos un grafo escuchable están relacionados con el momento de añadir o eliminar un vértice o una arista.

GraphEdgeChangeEvent<V,E>	
E getEdge()	Arista que ha sido añadida o eliminada

GraphVertexChangeEvent<V>	
V getVertex()	Vértice que ha sido añadido o eliminado

2.7 Otros subtipos de Graph

Junto con los subtipos vistos anteriormente tenemos, también, los subtipos:

- *SimpleGraph*: Desde un vértice v1 hasta v2 hay como máximo una arista. No pueden contener bucles.
- *MultiGraph*: No puede contener bucles
- *PseudoGraph*: No añade invariante.

Los tipos anteriores pueden tener vistas no modificables.

A partir de los tipos anteriores podemos tener muchos otros tipos combinándolos. Por ejemplo: Grafo Escuchable Simple Dirigido y con Peso. Para muchas de esas combinaciones hay previstas clases en *jGraphT* para crear objetos del tipo correspondiente. Estas clases implementan la interfaz del tipo correspondiente y añaden los invariantes necesarios para los sub-tipos.

Además de las combinaciones previstas podemos implementar otras que mantengan otros invariantes. Por ejemplo un Bosque. A este subtipo de *Graph<V,E>* lo denominaremos *Forest<V,E>* y se propone implementarlo como ejercicio.

2.8 Subgrafos, Unión de Grafos y Cambio de tipos

Un Grafo como hemos visto arriba es un cojunto de vértices y aristas que describiremos como *g =(v,e).* Un grafo *g1 = (v1,e1*) es un subgrafo de otro *g2(v2,e2)* si *v1* está incluido en *v2* y *e1* en *e2.* Dado un grafo podemos obtener un subgrafo del mismo tipo escogiendo un subconjunto de vértices y aristas. Asumimos que el tipo *G* cumple *G extends Graph<V,E>.*

Subgrafo: Constructores	
Subgraph(G base, Set<V> vs)	Devuelve una vista del grafo base que incluye sólo los vértices en vs. Tampoco incluye las aristas de los vértices no incluidos
Subgraph(G base, Set<V> vs, Set<E> es)	Devuelve una vista del grafo base que incluye sólo los vértices en vs y las aristas en es. Tampoco incluye las aristas de los vértices no incluidos

La unión de un grafo *g1 = (v1,e1)* y otro *g2 = (v2,e2)* es un nuevo grafo cuyos vértices son la unión de *v1* y *v2* y sus aristas la unión de *e1* y *e2*. Dados dos grafos podemos obtener el grafo unión mediante los constructores:

GraphUnion: Constructores

GraphUnion(G g1, G g2)	Devuelve una vista no modificable de la unión de g1 y g2
GraphUnion(G g1, G g2, WeightCombiner op)	Devuelve una vista no modificable de la unión de g1 y g2. El peso de las aristas se decide por el valor de op.

El tipo *WeightCombiner* define las constantes de combinación: FIRST, MAX, MIN, SECOND, SUM.

Es posible construir vistas no dirigidas a partir de grafos dirigidos o grafos con peso a partir de otros sin peso.

Constructores de Cambio de tipo	
AsUndirectedGraph(DirectedGraph<V,E> g)	Devuelve una vista no dirigida del grafo
AsWeightedGraph(Graph<V,E> g, Map<E,Double> weightMap)	Devuelve una vista con pesos del grafo

2.9 Factorías de Vértices, Aristas y Grafos

Es adecuado crear vértices, aristas y grafos mediante factorías. Para ello dispondremos de los siguientes tipos.

VertexFactory<V>	
V createVertex()	Crea un vértice

EdgeFactory<V,E>	
E createEdge(V sv, V tv)	Crea una arista sin información adicional entre los vértices sv, a tv.

El requisito que deben respetar estas factorías es que cada vértice y arista debe tener una identidad única. Es decir no puede haber dos vértices o dos aristas que sean iguales.

Las factorías anteriores las extendemos para crear vértices o aristas a partir de su representación en forma de cadenas de caracteres o de arrays de cadenas de caracteres.

Las clases *Graphs* y *Graphs2* contienen un conjunto de métodos estáticos adecuados para crear grafos desde ficheros de texto y desde otros tipos de grafos.

Graphs<V,E>	
static <V,E> UndirectedGraph<V,E> undirectedGraph(Graph<V,E> g)	Crea una vista no dirigida de g
V getOppositeVertex(Graph<V,E> g, E e, V v)	El vértice de la arista opuesto al dado
List<V> neighborListOf(Graph<V,E> g, V vertex)	Vecinos de un vétice

List<V> predecessorListOf(DirectedGraph<V,E> g, V vertex)	Predecesores de un vértice
List<V> successorListOf(DirectedGraph<V,E> g, V vertex)	Sucesores de un vértice

La factoría anterior tiene más métodos que pueden consultarse en la documentación de *JGraphT.*

2.10 Otros tipos relacionados con los grafos

Un camino en el grafo es representado por el tipo **GraphPath.**

GraphPath<V,E>	
List<E> getEdgeList()	Aristas del camino
V getEndVertex()	Vértice final
Graph<V,E> getGraph()	Grafo al que pertenece el camino
V getStartVertex()	Vértice inicial
double getWeight()	Peso total de las aristas del camino

3. Formatos de ficheros para representar grafos y visualización de los mismos

Hay diferentes tipos de formatos para representar grafos. Aquí veremos un par de estos formatos más conocidos: el formato *gdf* y el formato *dot*. Ambos formatos tienen propósitos diferentes. El primero tiene como objetivo representar la información asociada a los datos del grafo. Es decir las propiedades de los vértices y las aristas, el conjunto de vértices el conjunto de aristas y sus conexiones. El segundo está más orientado a representar la información que se necesita para representar gráficamente un grafo. Es decir posibles colores, figuras geométricas asociadas a los vértices, estilo de las aristas, etc. Veamos los detalles de cada uno de ellos.

Los archivos gdf tienen dos secciones claramente diferenciadas: la sección de vértices encabezada por la línea que comienza en *nodedef>* y la sección de aristas que comienza en *edgedef>.* Ambas secciones se componen de un conjunto de líneas cada una de las cuales describe las propiedades (separadas por comas) de un vértice o de una arista. Los nombres de las propiedades vienen en la cabecera de la sección. Aunque se puede explicitar un tipo para cada una de las propiedades usaremos el tipo por defecto (hilera de caracteres) para todas las propiedades.

Para cada vértice es completamente necesario indicar un identificador que debe ser único para cada vértice (es la primera propiedad) y puede tener tantas propiedades adicionales como consideremos. En este caso hemos añadido la propiedad *Label*.

Para cada arista hay que especificar los identificadores del vértice origen y el destino (son las dos primeras propiedades). En este caso hemos añadido la propiedad *edgeweight*. Si es

necesario podemos añadir propiedades que identifiquen de forma única a cada arista, si hubiera varias entre un mismo par de vértices.

```
nodedef>name,label
s1,Sevilla
s2,Córdoba
s3,Cádiz
s4,Málaga
edgedef>node1,node2,edgeweight
s1,s2,180.
s1,s3,120.
s2,s4,140.
s4,s1,210.
```

Los ficheros con la estructura anterior le damos extensión *gdf*. A partir de un fichero de ese tipo podemos construir un objeto de tipo *Graph<V,E>* o alguno de sus subtipos.

Para simplicidad en la asignatura usaremos una variante del formato *gdf*. Lo llamaremos estilo *lsi.* El fichero se estructura en la sección de vértices y aristas. Ambas secciones están encabezadas por *#VERTEX#* y *#EDGE#.* El primer campo de cada vértice es un identificador y debe ser único. El ejemplo anterior escrito en *lsi* es:

```
#VERTEX#
s1,Sevilla
s2,Córdoba
s3,Cádiz
s4,Málaga
#EDGE#
s1,s2,180.
s1,s3,120.
s2,s4,140.
s4,s1,210.
```

Para poder diseñar un algoritmo general que permita construir grafos de distintos tipos a partir de la información anterior diseñamos dos factorías de vértices y aristas:

StringVertexFactory<V>	
V createVertex(String[] s);	Crea un vértice a partir de s con valores para las propiedades dados en s.

StringEdgeFactory<V,E>	
E createEdge(V sv, V tv, String[] s)	Crea una arista entre los vértices *sv*, *tv* con valores para las propiedades dados en s.

En ambos casos el parámetro s (de tipo *String[]*) representa el conjunto de propiedades representadas en una línea. Las factorías extienden las por *jGrapht*. La clase *GraphsReader<V,E>* tiene un método *newGraph* para llevar a cabo esta tarea. El método toma el nombre del fichero, las factorías de vértices y aristas y un grafo. A este grafo se añaden los vértices y aristas que se deducen de la información en el fichero y se devuelve.

GraphsReader<V,E>	

`static <V,E> Graph<V, E> newGraph(`
`String file,`
`StringVertexFactory<V>,`
`StringEdgeFactory<V,E>,`
`Graph<V,E> graph)`
	Crea un grafo a partir de un fichero con la información de vértices y aristas.

La representación visual de un grafo ya construido es un tema mucho más complicado pero de gran interés. Ahora el fichero de partida debe contener información gráfica específica para indicar el tipo de vista del grafo que queremos. Para ello usaremos un fichero con extensión *gv (o dot)*.

```
graph andalucia {
  size="100,100";
  Sevilla [style = filled, color = blue];
  Cordoba -- Sevilla [label= "100+2", color= red, style= bold];
  Granada -- Cordoba [label= 10, color= red, style= bold];
  Sevilla -- Huelva [label= 15, color= red, style= bold];
  Sevilla -- Cadiz [label= 30];
  Almeria -- Granada [label= 25, color= red, style= bold];
  Granada -- Jaen[label= 32];
  Almeria --  Malaga [label= 51, color= red, style= bold];
}
```

Como vemos el fichero se compone de una cabecera y un conjunto de líneas. Cada línea describe un vértice con sus propiedades entre [] o una arista con sus propiedades también entre [] e indicando los vértices que conecta. El símbolo – representa una arista no dirigida. El símbolo -> (que no aparece en el texto) representa una arista dirigida.

Las propiedades y sus valores se representan por pares *nombre = valor*. Hay muchas propiedades predefinidas que tienen una semántica ya fijada.

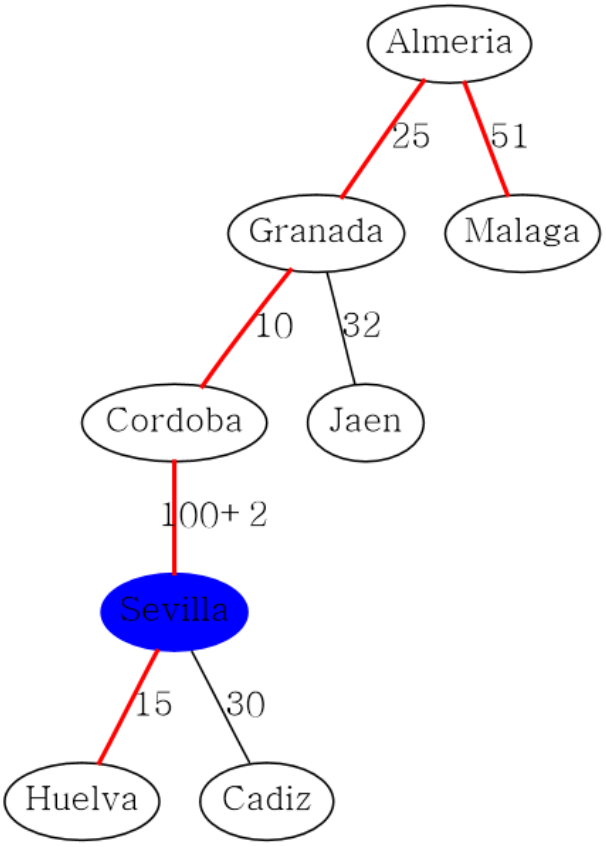
Para visualizar el gráfico con los detalles indicados en el fichero de extensión *gv* es necesario aplicar algunos algoritmos complejos para calcular la disposición (*layout)* más adecuada de los vértices y aristas. Podemos usar la herramienta *ZGRViewer* (visualizador de *Graphviz - Graph Visualization Software*) que lee ficheros con esa extensión y visualiza el grafo correspondiente siendo posible ver el resultado como una imagen. La documentación puede encontrarse en:
http://www.graphviz.org/Download.php

Los grafos construidos mediante jgrapht pueden ser exportados a un fichero con estructura *dot* (o *gv*) y visualizados con la herramienta anterior. Esto puede hacerse con la clase *DOTExporter* de jgrapht. Esta clase tiene un constructor de la forma:

public DOTExporter(VertexNameProvider<V> vertexIDProvider,
VertexNameProvider<V> vertexLabelProvider,
EdgeNameProvider<E> edgeLabelProvider,
ComponentAttributeProvider<V> vertexAttributeProvider,
ComponentAttributeProvider<E> edgeAttributeProvider)

Donde:

- `vertexIDProvider` – Para generar identificadores de los vértices.
- `vertexLabelProvider` – Para generar etiquetas de los vértices
- `edgeLabelProvider` – Para generar etiquetas de las aristas
- `vertexAttributeProvider` – Para generar otros atributos para los vértices
- `edgeAttributeProvider` – Para generar otros atributos para las aristas



El gráfico anterior es el resultado de procesar el fichero anterior con la herramienta comentada.

4. Grafos Virtuales

Añadimos la etiqueta *virtual* a un agregado de datos cuando no podemos o no queremos tener los elementos del agregado completamente en memoria y solamente disponemos de una descripción de algunas propiedades de los mismos. Así podemos tener secuencias numéricas virtuales, conjuntos y listas virtuales, etc. Un grafo virtual los definimos de forma *intensiva* o por *comprensión*. Es decir dando las propiedades de sus vértices y sus aristas. Frente a esta forma tenemos la forma *extensiva* que consiste en enumerar el conjunto de los vértices y el de las aristas. Cuando construimos un grafo (u otro agregado de datos) añadiendo vértices y aristas estamos usando la forma extensiva.

Cuando estudiamos los iterables ya vimos algunos ejemplos. En la clase *Iterables2* el método *from(Integer a, Integer b, Integer c)* genera los números enteros de *a* hasta *b* en pasos de *c* en *c.* Es un ejemplo de un iterable virtual: no necesitamos tener en la memoria todos los enteros que vamos a generar porque tenemos un algoritmo para generar cada entero a partir del anterior.

Otro ejemplo sería el *conjunto virtual* de los enteros pares comprendidos en el intervalo $[a, b)$. Un conjunto virtual, finito o no, puede describirse mediante un tipo T (una clase, interface o *enum* si usamos *Java*) y un predicado $p(x)$ sobre los elementos de ese tipo y lo podemos escribir:

$$s = \{x{:}\,T \mid p(x)\}$$

Otra variante es

$$s = \{x{:}\,T \mid p(x) \blacksquare f(x)\}$$

Que representa el conjunto de valores de $f(x)$ con un dominio representado por los valores de x que cumplen $p(x)$.

Para un conjunto virtual es fácil implementar las operaciones de más usuales de unión, intersección, pertenencia, etc., sin necesidad de construir una estructura de datos que contenga todos los elementos del conjunto en memoria.

Un multiconjunto de elementos de un tipo T se puede describir un conjunto más una función mediante una función $f{:}\,T \to int$ que devolverá la multiplicidad de ese elemento en el multiconjunto. O de forma más compacta sólo con una función como la anterior que devuelve un entero mayor de cero indicando el número de unidades de ese elemento en el multiconjunto o ni no pertenece 0. Lo indicamos de la forma:

$$B = \{x{:}\,T \mid f(x)\}$$

Una lista virtual secuencial de elementos de un tipo T se puede describir por un primer elemento $x_0{:}\,T$ y una función $s{:}\,T \to T$ que llamaremos elemento siguiente. La lista así descrita es infinita. Las listas infinitas indexadas se definen por mediante una función $f{:}\,[0, \dots) \to T$. Donde por $[0, \dots)$ queremos representar los enteros mayores o iguales a cero. De forma compacta las listas infinitas, secuenciales indexadas las podemos representar por:

$$L = (x_0{:}\,T, s)$$
$$L = (f)$$

Si queremos que sea finita hay que añadir un entero n para indicar que estamos interesados sólo en los n primeros elementos, un predicado $b(x)$ que indicará si un elemento es el último u otro predicado $d(x)$ que será verdadero para todos los elementos de la lista pero falso para el siguiente del último.

Las listas finitas indexadas de elementos de un tipo T podemos describirlas mediante un intervalo de enteros $[0, n)$ y una función $f{:}\,[0, n) \to T$.

Las diferentes versiones finitas las podemos escribir como:

$$L = (x_0 : T, s, n)$$
$$L = (x_0 : T, s, b)$$
$$L = (x_0 : T, s, d)$$
$$L = (n, f)$$

Un *grafo virtual* es algo similar. Cuando no podamos (por su tamaño) o no queramos (por su complejidad) tener en memoria todos los vértices y aristas de un grafo pero sí disponemos de un algoritmo para calcular los vecinos de un vértice y las aristas que lo conectan a él tenemos un *grafo virtual.* Un grafo virtual puede definirse mediante un *conjunto virtual de vértices* de tipo V y *para cada vértice un conjunto finito de vecinos*. El conjunto de vértices los definimos de la forma $\{v : V | q(v)\}$ dónde $q(v)$ es un predicado que indica si el vértice v es un valor válido para pertenecer al grafo. Las aristas vienen definidas implícitamente por dos vértices vecinos. El conjunto de vecinos de un vértice puede describirse de muchas maneras.

Un grafo virtual puede ser de los mismos subtipos de un grafo: *dirigido, no dirigido, simple, multígrafo, pseudografo*, etc. Como en el grafo, exigiremos que los vértices y las aristas sean únicos y para crearlos dispondremos de una factoría de vértices y otra de aristas tal como hemos visto más arriba.

Una forma común, pero no siempre posible, para describir las aristas de un grafo virtual es indicar un conjunto de m de alternativas $A = \{a_0, a_1, \dots, a_{m-1}\}$, una función $s : V \times A \to V$ que a partir de un vértice y una alternativa nos da el vértice vecino alcanzado y un *bipredicado* $p(V, A)$ que establece el dominio de la función s indicando si una alternativa es aplicable o no (es decir si existe el vecino obtenido y es válido). Un grafo virtual lo podemos representarlo entonces por:

$$G = (V, q, s, p)$$

Dados dos vértices v, v' pertenecientes al grafo, en un grafo virtual simple no dirigido, se debe cumplir que si $v' = s(v, a)$ entonces debe existir un a' tal que $v = s(v', a')$ con $a \in A,\ a' \in A,$ $p(v, a),\ p(v', a')$. Es decir por cada acción posible que permita pasar de v a v' debe existir otra a' que lo haga de v' hacia v. Diremos que a' es la acción inversa de a y viceversa. Con esas ideas podemos definir el conjunto de vecinos de un vértice $N(v)$, el conjunto de aristas incidentes en un vértice $E(v)$ y el *bipredicado* $r(v, v')$ indicando si hay una arista entre ambos vértices como:

$$N(v) = \{a : A | p(v, a) \blacksquare s(v, a)\}$$
$$E(v) = \{e : (v, v'), a : A | p(v, a) \wedge v' = s(v, a)\}$$
$$r(v, v') = (v, v') \in E(v)$$

Con las ideas anteriores podemos recorrer un grafo virtual no dirigido partiendo de un vértice inicial. Hemos de tener en cuenta que sólo alcanzaremos los vértices de la componente conexa del vértice proporcionado. Si el grafo no es conexo habrá que proporcionar un vértice inicial por cada componente conexa.

Para un grafo virtual dirigido, junto a los anteriores, existen los conceptos de vecinos salientes $Out(v)$ y entrantes $In(v)$ y aristas entrantes y salientes $EOut(v),\ EIn(v)$. La definición para el grafo dirigido de todos ellos es:

$$N(v) = In(v) \cup Out(v)$$
$$E(v) = EIn(v) \cup EOut(v)$$
$$r(v, v') = (v, v') \in E(v)$$
$$Out(v) = \{a: A | p(v, a) \blacksquare s(v, a)\}$$
$$In(v) = \{v': V, a: A | q(v') \wedge p(v', a) \wedge v = s(v', a) \blacksquare v'\}$$
$$EOut(v) = \{e: (v, v') | v' \in Out(v)\}$$
$$EIn(v) = \{e: (v', v) | v' \in In(v)\}$$

La definición anterior permite hacer un *recorrido hacia adelante* del grafo dejando una definición implícita de los vecinos entrantes. Si necesitamos una definición explícita de los vecinos entrantes tenemos que ampliar la definición anterior con un nuevo predicado p' y una segunda función t tal que $t: V \times A \to V$. La función t nos permite calcular los vecinos entrantes de un vértice a partir de las acciones disponibles para llegar. Con esta función t' resulta:

$$In(v) = \{a: A | p'(v, a) \blacksquare t(v, a)\}$$

Sin esa función el conjunto de vecinos entrantes no es implementable directamente. En el caso de grafos virtuales no dirigidos las funciones s, t son la misma y un vecino entrante es a su vez saliente pero usando en cada caso una alternativa que puede ser diferente como hemos visto arriba.

En un grafo virtual no estamos interesados en añadir ni eliminar aristas o vértices. Aunque si nos interesa decidir si un vértice o una arista pertenece al grafo o no. Igualmente puede interesarnos saber si hay una arista entre dos vértices dados u obtener todas las que haya entre dos de ellos.

Las operaciones de la factoría *Graphs* son aplicables a este tipo siempre que lo permitan los métodos correspondientes del grafo virtual.

Se proporcionan dos implementaciones de grafos virtuales *UndirectedSimpleVirtualGraph* y *DirectedSimpleVirtualGraph* con los detalles que se ven abajo. En el constructor se le pueden proporcionar varios vértices del grafo de forma explícita.

```
public interface SimpleEdge<V> {
       double getEdgeWeight()
       V getSource();
       V getTarget();
}

public class SimpleVirtualGraph<V extends VirtualVertex<V,E>,
                                E extends SimpleEdge<V>>
              implements Graph<V, E> {
              ...
}

public class UndirectedSimpleVirtualGraph<V extends VirtualVertex<V,E>,
                                          E extends SimpleEdge<V>>
              extends SimpleVirtualGraph<V,E>
              implements UndirectedGraph<V,E>{
       UndirectedSimpleVirtualGraph(EdgeFactory<V,E> edgeFactory) ...
       UndirectedSimpleVirtualGraph(EdgeFactory<V,E> edgeFactory, V[] vs) ...
       ...
}
```

```
public class DirectedSimpleVirtualGraph<V extends VirtualDirectedVertex<V,E>,
                                        E extends SimpleEdge<V>>
             extends SimpleVirtualGraph<V, E>
             implements DirectedGraph<V, E> {

       DirectedSimpleVirtualGraph(EdgeFactory<V,E> edgeFactory) ...
       DirectedSimpleVirtualGraph(EdgeFactory<V,E> edgeFactory, V[] vs) ...
       ...
}
```

Para instanciar las clases anteriores los tipos que ocupan los vértices y las aristas deben ser subtipos de *VirtualVertex* (vértice de u grafo no dirigido) o *VirtualDirectedVertex* (vértice de un grafo dirigido.

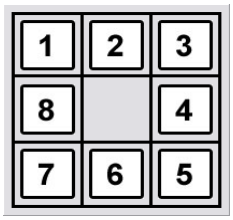
```
public interface VirtualVertex<V extends VirtualVertex<V,E>, E> {
       boolean isValid();
       Set<V> getNeighborListOf();
       Set<E> edgesOf();
       boolean isNeighbor(V e);
}

public interface VirtualDirectedVertex<V extends
                        VirtualDirectedVertex<V,E>,E>
            extends VirtualVertex<V,E> {
       Set<V> getPredecessorListOfListOf();
       Set<E> incomingEdgesOf();
       Set<V> getSuccessorListOfListOf();
       Set<E> outgoingEdgesOf();
}
```

En definitiva en un grafo virtual la información que necesitamos son los vecinos de cada vértice o alternativamente las aristas incidentes en un vértice. Una implementación de la clase `VirtualVertex<V,E>` para el caso concreto de disponer de un conjunto finito de alternativas (o acciones) para pasar de un vértice a otro vecino puede usarse la clase `AlternativeVirtualVertex<A,V>` que usa aristas del tipo `AlternativeSimpleEdge<>.`

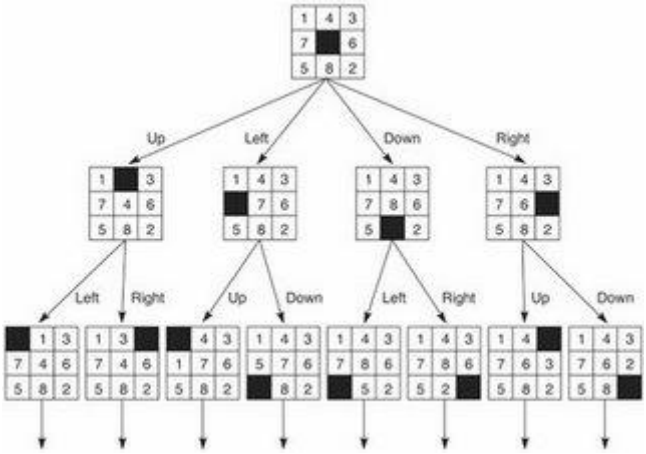
Veamos un ejemplo para concretar todo lo anterior.

Un ejemplo de grafo virtual nos lo da el problema de resolver un puzle. El juego del 8-puzzle usa un tablero con 9 casillas, las cuales van enumeradas del 1 al 8 más una casilla vacía que podemos etiquetar con 0. Los movimientos posibles del puzle consisten en intercambiar la casilla vacía con alguno de sus vecinos mediante movimientos horizontales, verticales, hacia la izquierda o derecha. El problema consiste en dada una configuración inicial llegar a una configuración final (meta) mediante los movimientos permitidos.

El problema puede ser modelado mediante un grafo virtual simple y no dirigido. Los vértices del grafo representan cada una de las posibles distribuciones de los 8 números y una casilla vacía en un tablero de 3 filas y tres columnas. Un vértice en particular viene representando en el gráfico:



Las aristas representan los posibles movimientos: *Arriba, Izquierda, Abajo, Derecha.* Algunos vértices y aristas vienen representados en el gráfico siguiente.



Como podemos ver el grafo virtual es simple y no dirigido. Resolver el problema propuesto consiste en partir de un vértice inicial dado y encontrar un camino de longitud mínima hasta otro vértice final también dado.

Dejamos como ejercicio todos estos detalles de implementación. Podemos ver que el número de vértices de este grafo es *9! = 362880.* La forma de verlo es considerar que cada vértice puede ser representado por una lista de nueve números (los números del tablero puestos en filas consecutivamente ejemplo). El número de estas listas es igual a las permutaciones de los nueve números.

El número de vértices (*362880)* nos aconseja considerar el grafo virtualmente en vez de construirlo completamente en memoria que en muchos casos será inviable.

```
public class EstadoPuzzle implements
       VirtualVertex<EstadoPuzzle,DefaultSimpleEdge<EstadoPuzzle>> {

    public static EstadoPuzzle create(Integer... d) {
             return new EstadoPuzzle(d);
    }

       private Integer[][] datos;
       private int i0;
       private int j0;
       public static int numFilas = 3;

       private EstadoPuzzle(Integer... d) {
        ...
       }

       public EstadoPuzzle getVecino(int f, int c){
```

```
            if(f<0 || f>=EstadoPuzzle.numFilas || c<0 ||
                  c>=EstadoPuzzle.numFilas)
                        throw new IllegalArgumentException();
            Integer[][] dd =
                  new Integer[EstadoPuzzle.numFilas][EstadoPuzzle.numFilas];
            for (int i = 0; i < EstadoPuzzle.numFilas; i++) {
                  for (int j = 0; j < EstadoPuzzle.numFilas; j++) {
                        dd[i][j] = datos[i][j];
                  }
            }
            dd[f][c] = datos[this.i0][this.j0];
            dd[this.i0][j0] = datos[f][c];
            return new EstadoPuzzle(dd,f,c);
      }

      @Override
      public Set<EstadoPuzzle> getNeighborListOf() {
            List<ParInteger> ls = InitialAggregate.list(
                  ParInteger.create(1,0),
                  ParInteger.create(0,1),
                  ParInteger.create(-1,0),
                  ParInteger.create(0,-1));
            return ls.stream()
                  .<ParInteger> map((ParInteger x)->
                              ParInteger.create(x.p1+this.i0,
                                                x.p2+this.j0))
                  .filter((ParInteger x)-> x.p1>=0 &&
                                           x.p1<EstadoPuzzle.numFilas &&
                                           x.p2>=0 &&
                                           x.p2<EstadoPuzzle.numFilas)
                  .<EstadoPuzzle>map((ParInteger x)->
                                           this.getVecino(x.p1,x.p2))
                   .collect(Collectors.<EstadoPuzzle>toSet());
      }

      public Set<DefaultSimpleEdge<EstadoPuzzle>> edgesOf(){
            return this.getNeighborListOf().stream()
               .<DefaultSimpleEdge<EstadoPuzzle>>map(x->
                              DefaultSimpleEdge.<EstadoPuzzle>getFactoria()
                                    .createEdge(this,x))
               .collect(Collectors.<DefaultSimpleEdge<EstadoPuzzle>>toSet());
      }
...
}
```

Aunque no incluido en el código anterior podemos considerar si un valor del tipo `EstadoPuzzle` es válido

5. Recubrimientos, componentes conexas y ciclos en grafos

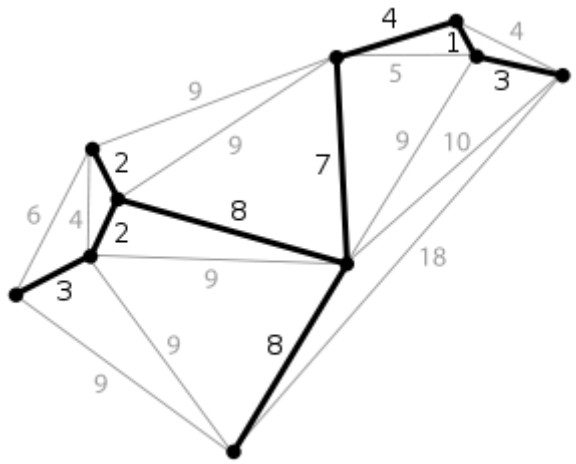
5.1 Recubrimiento mínimo: Algoritmo de Kruskal

Un bosque de recubrimiento mínimo de un grafo es subgrafo que sea un bosque, incluya todos los vértices del grafo y minimice la suma total de los pesos de las aristas escogidas.

El algoritmo de *Kruskal* busca un **bosque de recubrimiento mínimo** de un grafo. Su complejidad es *O(E log E).* Donde E es el número de aristas.

El algoritmo devuelve el conjunto de aristas que definen el bosque.

KruskalMinimumSpanningTree(Graph<V,E> graph)	
Set<E> getMinimumSpanningTreeEdgeSet()	Conjunto de aristas del árbol de recubrimiento mínimo
double getMinimumSpanningTreeTotalWeight()	Coste de las aristas del árbol de recubrimiento mínimo



5.2 Recubrimiento mínimo: Algoritmo de Prim

El algoritmo de *Prim* calcula también un árbol de recubrimiento mínimo para grafos conexos.

PrimMinimumSpanningTree(Graph<V,E> graph)	
Set<E> getMinimumSpanningTreeEdgeSet()	Conjunto de aristas del árbol de recubrimiento mínimo
double getMinimumSpanningTreeTotalWeight()	Coste de las aristas del árbol de recubrimiento mínimo

Aplicación del recubrimiento mínimo

La aplicación usual es un problema de diseño de una red telefónica. Suponga que tiene varias ciudades que desea conectar mediante líneas telefónicas. La compañía telefónica cobra diferentes cantidades de dinero para conectar diferentes pares de ciudades. ¿Qué ciudades conectar para que todas estén conectadas entre sí al mínimo precio?

5.3 Recubrimiento de vértices

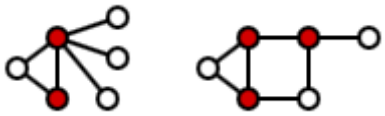
El problema del recubrimiento mínimo anterior consistía en escoger un subconjunto de aristas cuyos extremos incluyeran todos los vértices y cuya suma de pesos fuera mínima. Podríamos llamarlo recubrimiento mínimo de aristas. De forma similar el problema llamado recubrimiento de vértices consiste en escoger el mínimo número de vértices que tocan todas las aristas del grafo.

La clase *VertexCovers* implementa un algoritmo para resolver el problema.

VertexCovers()
static <V,E> Set<V> find2ApproximationCover(Graph<V,E> g)
static <V,E> Set<V> findGreedyCover(UndirectedGraph<V,E> g)

Ambos métodos implementan algoritmos de aproximación para resolver el problema. Devuelven el subconjunto de vértices buscado o un superconjunto aproximado del mismo.

Los vértices llenos son el recubrimiento de vértices mínimo del grafo.



Aplicación

Si tenemos un grafo cuyas aristas representan carreteras, los vértices representan cruces entre ellas, pretendemos colocar cámaras de seguridad en algunos cruces de tal manera que puedan ver todas las carreteras que acceden al cruce. Si pretendemos colocar el mínimo número de cámaras podemos obtener la solución mediante recubrimiento de vértices.

5.4 Componentes conexas

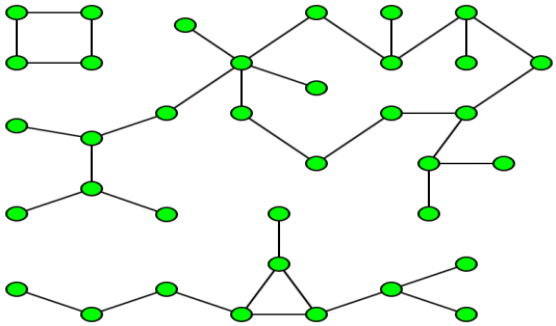
Un grafo no dirigido es conexo si entre cada dos vértices existe un camino del grafo. Una **componente conexa de un grafo no dirigido** es un subgrafo maximal conexo. Un grafo conexo tiene una sola componente conexa.

Un **grafo dirigido es débilmente conexo** (weakly connected) si al reemplazar cada arista dirigida por una no dirigida resulta un grafo conexo. Un grafo dirigido es fuertemente conexo si para cada par de vértices *u, v* existe un camino dirigido de *u* a *v* o de *v* a *u*. Una **componente fuertemente conexa** de un grafo dirigido es un subgrafo maximal fuertemente conexo de un grafo dirigido. Una **componente débilmente conexa** es un subgrafo maximal débilmente conexo de un grafo dirigido.

La clase **ConnectivityInspector** calcula componentes conexas de un grafo no dirigido y componentes débilmente conexas de un grafo dirigido. El tipo es, también, una implementación del tipo **GraphListener.**

ConnectivityInspector(DirectedGraph<V,E> g)	
ConnectivityInspector(UndirectedGraph<V,E> g)	
Set<V> connectedSetOf(V vertex)	Conjunto de vértices conectados a vertex
List<Set<V>> connectedSets()	Lista de componentes conexas
boolean isGraphConnected()	Si es conexo
boolean pathExists(V sourceVertex, V targetVertex)	Si existe un camino desde un vértice al otro

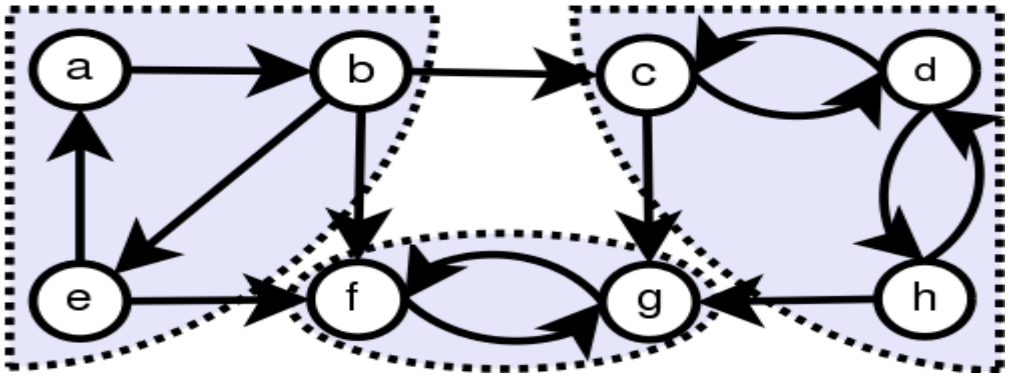
Componentes conexas de un grafo:



El tipo **StrongConnectivityInspector** calcula las componentes fuertemente conexas de un grafo dirigido.

StrongConnectivityInspector(DirectedGraph<V,E> directedGraph)	
List<Set<V>> stronglyConnectedSets()	Lista de componentes fuertemente conexas
boolean isStronglyConnected()	Si es fuertemente conexo
List<DirectedSubgraph<V,E>> stronglyConnectedSubgraphs()	Lista de subgrafos fuertemente conexos

Componentes fuertemente conexas.



5.5 Ciclos en grafos

Calcula si hay ciclos en un grafo

CycleDetector (DirectedGraph<V,E> directedGraph)	
Set<V> findCycles()	Conjunto de vértices en alguno de los ciclos
boolean detectCycles()	Decide si el grafo tiene ciclos
boolean detectCyclesContainingVertex(V v)	Decide si el grafo tiene ciclos

6. Recorridos sobre grafos

Los grafos son agregados de datos y, como todos los agregados, pueden ofrecer diferentes maneras de recorrer los elementos del mismo. En concreto diferentes maneras de recorrer los vértices de un grafo. En Java cada forma concreta de recorrer los elementos de un agregado se consigue implementado un objeto de tipo Iterable. Cada recorrido puede considerarse, también, como un problema de búsqueda de un vértice con propiedades específicas entre los vértices de un grafo. Por eso hablaremos indistintamente como recorridos o búsqueda. De entre ellos los más conocidos son: recorrido en anchura, recorrido en profundidad, orden topológico y siguiente vértice más cercano. Veamos cada uno de ellos y la forma de obtenerlos.

Aunque los recorridos pueden ser extendidos para recorrer todas las componentes conexas de un grafo en los que sigue consideraremos sólo el recorrido de una componente conexa. Cada recorrido comienza en un nodo inicial (o en un conjunto de vértices iniciales) y es exhaustivo en el sentido que visita todos los vértices de la componente conexa dada (o de todas las componentes conexas del grafo si el recorrido se ha extendido para ello).

Cada recorrido define un árbol de recubrimiento (o un bosque si el recorrido se ha extendido a todas las componentes conexas del grafo). Las aristas del árbol de recubrimiento definen caminos, con propiedades específicas para cada recorrido, desde cada vértice al vértice inicial (vértices iniciales).

Cada recorrido implementa el tipo *Iterator<V>* y puede informar a *listeners* sobre los siguientes eventos:

- Ha comenzado el recorrido de una nueva componente conexa
- Ha terminado el recorrido de una componente conexa
- Se ha atravesado una arista dada
- Se ha visitado un vértice dado
- Se ha cerrado un vértice dado. Cuándo se cierra un vértice es algo específico de cada recorrido.

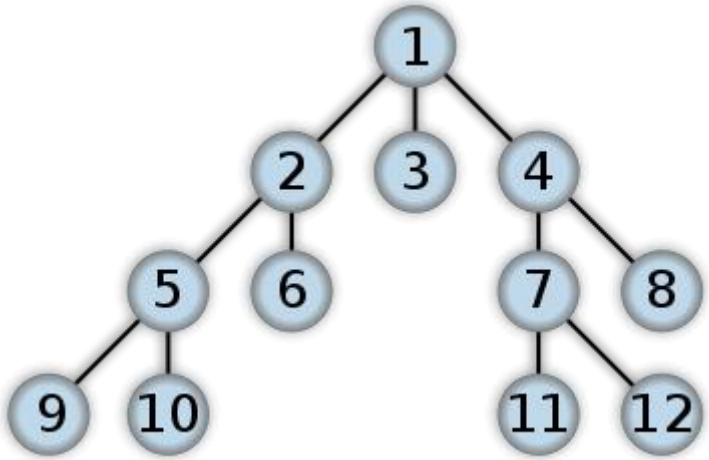
Los *listeners* deben implementar *TraversalListener<V,E>.*

6.1 Recorrido en anchura

La búsqueda en anchura visita cada nodo, posteriormente sus hijos, luego los hijos de estos, etc. Es, por lo tanto un recorrido por niveles o si se quiere por distancia al nodo origen (asociando peso 1 a cada arista). Primero los que están a distancia cero, luego los que están a distancia 1, etc.

Este recorrido se puede aplicar a grafos dirigidos o no dirigidos.

En el gráfico se muestra un recorrido en anchura. Dependiendo del orden en que se recorran los hijos de un nodo hay varios recorridos que cumplen las restricciones anteriores dado un grafo y un vértice inicial.



El recorrido en anchura define un árbol de recubrimiento donde las aristas escogidas definen caminos mínimos desde cada vértice al inicial (asociando peso 1 a cada arista). La distancia al vértice inicial es el nivel de cada vértice.

Cuando desde un vértice se alcanza otro por primera vez a través de una arista ésta se incluye en el árbol de recubrimiento.

La clase *BreadthFirstIterator<V,E>* implementa un iterador que hace el recorrido en anchura. Sus constructores son:

- *BreadthFirstIterator(Graph<V,E> g):* Vértice inicial arbitrario
- *BreadthFirstIterator(Graph<V,E> g, V startVertex)*

6.2 Usos del recorrido en Anchura

La búsqueda en anchura define un árbol de recubrimiento sobre sobre le grafo y cada uno de los vértices pueden ser etiquetados, como en el caso del recorrido en profundidad, por un

número entero que indica la posición en la que es visitado. Este árbol de recubrimiento no es único porque depende de la forma de añadir los vecinos del vértice actual al conjunto SA.

El recorrido en anchura tiene la propiedad que el camino de la raíz a un vértice en el árbol de recubrimiento tiene el mínimo número de aristas posible.

El recorrido en anchura y el orden definido a partir del mismo tienen diferentes aplicaciones:

- Cálculo de las componentes conexas de un grafo no dirigido. Una componente conexa está formada por todos los vértices alcanzables desde uno dado.
- Cálculo de caminos de un vértice a los demás. Esencialmente estos caminos vienen definidos por el árbol de recubrimiento asociado al recorrido.
- Encontrar el camino más corto entre un vértice y todos los demás de su componente conexa. Los caminos definidos por el árbol de recubrimiento son los caminos mínimos (en número de aristas) desde el vértice origen a los demás.

6.3 Recorrido según el siguiente más cercano

Es un tipo de recorrido que generaliza la búsqueda en anchura. Ahora el siguiente vértice es el más cercano al inicial según la suma de los pesos de las aristas del camino hasta él.

Este recorrido se puede aplicar a grafos dirigidos o no dirigidos.

La clase *ClosestFirstIterator<V,E>* implementa un iterador que hace el recorrido en según el siguiente más cercano. Sus constructores son:

- *ClosestFirstIterator(Graph<V,E> g)*
- *ClosestFirstIterator(Graph<V,E> g, startVertex)*

Este iterador va cambiando el árbol de recubrimiento a medida que va avanzando. Al final el árbol de recubrimiento construido define los caminos mínimos desde cada vértice al vértice inicial. Esa información puede ser consultada mediante los métodos de la clase anterior:

- *double getShortestPathLength(V vertex):* Longitud del camino mínimo hasta el vértice inicial.
- *public E getSpanningTreeEdge(V vertex):* Arista que define el camino mínimo hasta el vértice inicial.

Junto a los anteriores dispondremos de una variante de *ClosestFirstIteratorG<V,E>* con un funcionamiento y métodos similares a *ClosestFirstIteratorG<V,E>* pero donde la longitud del camino se define como en los algoritmos *A** que veremos más abajo.

El recorrido según el siguiente más cercano es la base de del Algoritmo de *Dijkstra* y de los *Algoritmos A*.*

6.4 Recorrido en profundidad

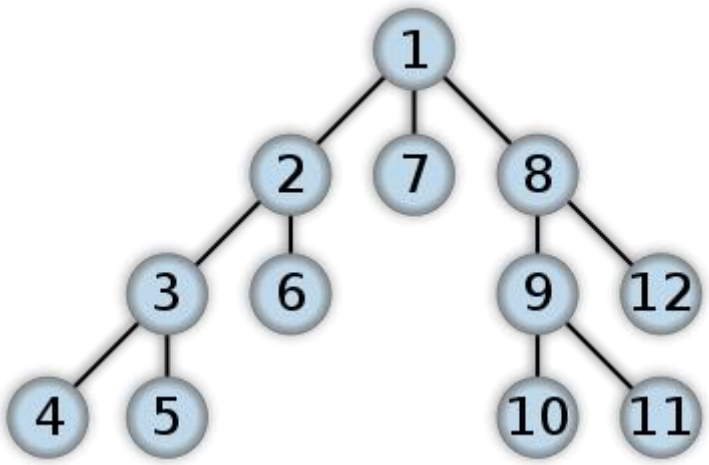
La búsqueda en profundidad tiene varias variantes según se visite un vértice antes, después o entre sus hijos. Las diferentes posibilidades son:

- *Preorden*: Cada vértice se vista antes que cada uno de sus hijos
- *Postorden*: Cada vértice se visita después que todos sus hijos
- *Inorden*: Si el número de hijos de cada vértice en el árbol de recubrimiento es dos como máximo entonces cada vértice se visita después de sus hijos izquierdos y antes de sus hijos derechos.

En el recorrido en preorden se visita un nodo, luego su primer hijo, luego el primer hijo de este, etc. Es, por lo tanto un recorrido, como su nombre indica, que avanza en profundidad.

Este recorrido se puede aplicar a grafos dirigidos o no dirigidos.

En el gráfico se muestra un recorrido en preorden. Dependiendo del orden en que se recorran los hijos de un nodo hay varios recorridos que cumplen las restricciones anteriores dado un grafo y un vértice inicial.



La clase *DepthFirstIterator<V,E>* implementa un iterador que hace el recorrido en preorden. Sus constructores son:

- *DepthFirstIterator(Graph<V,E> g)*
- *DepthFirstIterator(Graph<V,E> g, V startVertex)*

Cuando desde un vértice se alcanza otro por primera vez a través de una arista ésta se incluye en el árbol de recubrimiento.

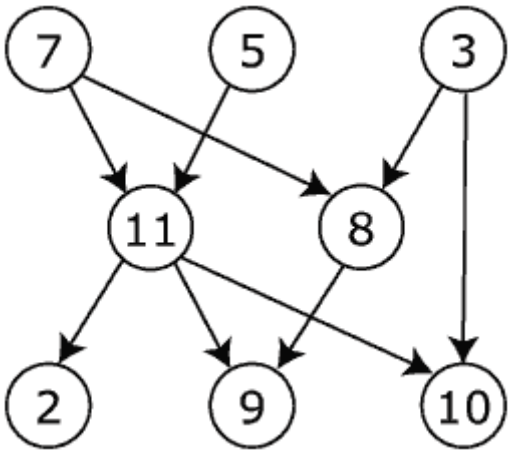
6.5 Usos de del recorrido en profundidad

La búsqueda en profundidad y los órdenes definidos a partir del mismo tienen diferentes aplicaciones:

- Obtención de un bosque de recubrimiento de un grafo dirigido o no.
- Cálculo de las componentes conexas de un grafo no dirigido: Para hacerlo recorremos todos los vértices del grafo y mantenemos un conjunto de vértices ya clasificados. Para cada vértice no clasificado añadimos a un conjunto los vértices alcanzables desde él y los incluimos entre los ya clasificados. Escogemos el siguiente vértice aún no clasificado para formar la segunda componente conexa y así sucesivamente hasta que clasifiquemos todos los vértices.
- Cálculo de las componentes fuertemente conexas de un grafo dirigido. Usando el algoritmo visto más arriba.
- Cálculo de caminos de un vértice a los demás en un grafo dirigido o no dirigido o decidir que no existe camino. Esencialmente estos caminos (o no existencia de caminos) vienen definidos por el árbol de recubrimiento asociado al recorrido que empieza en el vértice dado.
- Comprobar si un grafo no dirigido es conexo. No es conexo si existe más de una componente conexa.
- Comprobar si un grafo dirigido es conexo. No es conexo si existe más de una componente fuertemente conexa. Si un grafo dirigido es conexo existen caminos entre cada par de vértices.
- Ordenación topológica de un grafo dirigido: la ordenación topológica de los vértices de un grafo no dirigido viene dada por el Postorden Inverso.
- Cálculo de las componentes débilmente conexas de un grafo dirigido *G*. Es calcular las componentes conexas del grafo *G'* obtenido a partir de *G* haciendo todas las aristas no dirigidas.

6.6 Orden topológico

Es un tipo de recorrido que se aplica a grafos dirigidos. En este recorrido cada vértice va después que los vértices que le anteceden en el grafo dirigido.



Con las restricciones anteriores hay varios recorridos posibles. Algunos son:

- 7, 5, 3, 11, 8, 2, 9, 10
- 3, 5, 7, 8, 11, 2, 9, 10
- 3, 7, 8, 5, 11, 10, 2, 9
- 5, 7, 3, 8, 11, 10, 9, 2
- 7, 5, 11, 3, 10, 8, 9, 2
- 7, 5, 11, 2, 3, 8, 9, 10

La clase *TopologicalOrderIterator<V,E>* implementa un iterador que hace el recorrido en orden topológico. Sus constructores son:

- *TopologicalOrderIterator(DirectedGraph<V,E> dg)*
- *TopologicalOrderIterator(DirectedGraph<V,E> dg, Queue<V> queue):* La cola, y su posible implementación, permiten elegir uno de los posibles recorridos.

Una aplicación del orden topológico es en la programación de una secuencia de tareas que tienen precedencias entre ellas. Las tareas están representados por vértices, y hay un arco (o arista) desde x a y si la tarea x debe completarse antes que la tarea y comience.

7. Camino mínimo

El problema de camino mínimo se puede enunciar del siguiente modo:

> Dado un grafo y dos vértices encontrar el camino (secuencia de aristas) con longitud mínima entre ambos vértices.

7.1 Algoritmo de Dijkstra

El algoritmo de *Dijkstra* encuentra el camino más corto entre dos vértices de un grafo. Los pesos deben ser no negativos. La implementación que se ofrece tiene una complejidad de $O(V^2)$.

Si e_i es el peso de una arista dada entonces para el uso de este algoritmo definimos la longitud del camino *C* formado por una secuencia de aristas como

$$\sum_{i \in C} e_i$$

La información sobre los pesos de las aristas está siempre disponible en método disponible en el tipo *Graph<V,E>:*

- *double getEdgeWeight(E e)*

La clase *DijkstraShortestPath<V,E>* implementa el algoritmo. Sus constructores y métodos relevantes son:

DijkstraShortestPath(Graph<V,E> graph, V startVertex, V endVertex)	
GraphPath<E,V> getPath()	Camino mínimo
List<E> getPathEdgeList()	Aristas del camino mínimo
double getPathLength()	Longitud del camino mínimo

Este es el algoritmo más utilizado para encontrar el camino más corto entre dos punto de un grafo (aunque sea virtual), por ejemplo, en grafos que representen mapas, nodos de una red, etc.

7.2 Algoritmo de Bellman-Ford

El algoritmo de *Bellman-Ford* encuentra el camino mínimo (entendido, como antes, como suma de los pesos de las aristas del camino) entre dos vértices. El algoritmo acepta pesos negativos en las aristas. Su complejidad es *O(VE)*. Donde V es el número de vértices y E el número de aristas.

La clase *BellmanFordShortestPath<V,E>* implementa el algoritmo. Sus métodos más importantes son:

BellmanFordShortestPath(Graph<V,E> graph, V startVertex)	
List<E> getPathEdgeList(V endVertex)	Camino mínimo desde el startVertex hasta endVertex
double getCost(V endVertex)	Coste del camino mínimo desde el startVertex hasta endVertex

7.3 Algoritmo de Floyd–Warshall

El algoritmo de **Floyd–Warshall** encuentra los caminos más cortos (definidos como anteriormente) entre cada dos vértices de un grafo (permite pesos positivos y negativos). La complejidad del algoritmo es $O(V^3)$.

La clase *FloydWarshallShortestPaths<V,E>* implementa el algoritmo. Sus métodos más importantes son:

FloydWarshallShortestPaths(Graph<V,E> graph)	
GraphPath<V,E> getShortestPath(V a, V b)	Camino mínimo entre a y b
List<GraphPath<V,E>> getShortestPaths(V v)	Caminos mínimos desde v al resto de vértices
double shortestDistance(V a, V b)	Longitud del camino mínimo de a hasta b
int getShortestPathsCount()	Número de caminos mínimos
double getDiameter()	Longitud del camino mínimo más largo
Graph<V,E> getGraph()	Grafo de partida

7.4 Algoritmos A*

Los algoritmos A* son una generalización del algoritmo de *Dijkstra* donde la definición de longitud del camino es más general. Ahora podemos asignar pesos no sólo a las aristas. También a los vértices, etc.

Sea, como antes, e_i el peso de una arista, v_i el peso asociado a un vértice, h_{ij} el costo estimado del camino del vértice *i* al *j* y w_{ijk} el peso asociado al vértice *i* asumiendo que se ha llegado a él mediante la arista *j* y se sale de él mediante la arista *k*.

La longitud del camino *C* es entonces

$$|C| = \sum_{i \in C} e_i + \sum_{i \in C} v_i + \sum_{i \in C} w_{ijk}$$

Los algoritmos *A** tratan con una idea más general que llamaremos **ruta** que pasa por un vértice dado. Una ruta viene definida por un camino que va desde el vértice inicial a un vértice intermedio (vértice actual) y una estimación del costo del camino hasta el vértice final. Si la ruta está definida por el vértice inicial V_i al vértice final V_f y pasando por un vértice actual V_a entonces la longitud de la misma es la suma del coste ya conocido del camino *C* que va de V_i a V_a más el coste estimado para llegar al final. El coste de esa ruta es:

$$|R| = \sum_{i \in C} e_i + \sum_{i \in C} v_i + \sum_{i \in C} w_{ijk} + h_{af}$$

Donde C es el camino que va de V_i a V_a. Si V_a es el vértice final entonces la ruta coincide con el camino y h_{af} es cero.

Para aplicar los algoritmos *A** incorporamos la información anterior al grafo. Un grafo con esa información define el tipo *AStarGraph<V, E>.*

```
public interface AStarGraph<V, E> extends Graph<V,E> {
	double getVertexWeight(V vertex);
	double getVertexWeight(V vertex, E edgeIn, E edgeOut);
	double getWeightToEnd(V startVertex, V endVertex,
	Function<V,Double> goalDistance, Set<V> goalSet);
}
```

Lo métodos devuelven la siguiente información:

- *double getVertexWeight(E edge):* Peso asignado a la arista
- *double getVertexWeight(V vertex):* Peso asignado al vértice
- *double getVertexWeight(V vertex, E edgeIn, E edgeOut):* Peso asignado a la transición de dos aristas en un vértice.
- *double getWeightToEnd(V actual, V endVertex, Function<V,Double> goalDistance, Set<V> goalSet):* Distancia desde el vértice actual al final, a un objetivo especificado en *goalDistance*, al conjunto de vértices *goalSet* o una combinación de las mismas.

Los algoritmos *A** funcionan como el de *Dijkstra* pero en cada momento mantienen, en el árbol de recubrimiento generado, la ruta mínima que va del vértice inicial al final a través del vértice actual.

En estos algoritmos el criterio de parada es cuando se encuentra el vértice final. Se pueden generalizar para que el criterio de parada sea cuando se encuentra un vértice que cumple un predicado (cuando la función *goalDistance* devuelve cero) o cuando se alcanza alguno de los vértices de un conjunto. En el primer caso nos devolverá el camino mínimo del vértice inicial al final. En el segundo caso devuelve el camino mínimo al vértice encontrado y en el tercero nos devolverá el camino a vértice del conjunto más cercano al origen.

La clase *AStarAlgorithm<V,E>* implementa el algoritmo *A** comentado. El algoritmo no solamente busca el camino mínimo de un vértice a un destino. También puede buscar el camino mínimo a un conjunto de vértices o desde un vértice hasta el primero que cumple un predicado.

Si queremos maximizar el peso total del camino podemos conseguirlo asignado pesos negativos a las diferentes aristas, vértices, distancias y transiciones del grafo.

Sus constructores y métodos relevantes son:

AStarAlgorithm(AStarGraph<V,E> graph, V startVertex, V endVertex) ***AStarAlgorithm(AStarGraph<V,E> graph, V startVertex, V endVertex, Predicate<V> goal)*** ***AStarAlgorithm(AStarGraph<V,E> graph, V startVertex, V endVertex, Set<V> goalSet)***	
GraphPath<V,E> getPath()	Camino mínimo
List<E> getPathEdgeList()	Aristas del camino mínimo
double getPathLength()	Longitud del camino mínimo
AStarIterator<V, E> getIterator()	El iterador usado por el algoritmo

La lista de vértices puede ser obtenida con el método de la clase *Graphs*.

- *static <V,E> List<V> getPathVertexList(GraphPath<V,E> path)*

Los métodos nos devuelven el camino mínimo calculado, una lista de las aristas que lo forman, su longitud y el iterador usado por el algoritmo. El iterador nos proporciona, a su vez, información sobre el camino desde los vértices alcanzados hasta el vértice inicial y tiene los métodos:

AStarIterator (AStarGraph<V,E> graph, V startVertex, V endVertex) ***AStarIterator(AStarGraph<V,E> graph, V startVertex, V endVertex, Predicate<V> goal)*** ***AStarIterator(AStarGraph<V,E> graph, V startVertex, V endVertex, Set<V> goalSet)***	
E getSpanningTreeEdge(V vertex)	Arista que conduce de vertex al origen o *null* si el vértice no ha sido alcanzado o el vértice origen
double getShortestPathLength(V vertex)	Longitud del camino que pasa por vertex
boolean hasNext()	Si hay más vértices
V next()	Siguiente vértice

Los algoritmos *A** son adecuados para buscar un camino mínimo entre un vértice inicial y un vértice objetivo o entre un vértice inicial y otro que cumpla un predicado dado. Un algoritmo de búsqueda de este tipo es admisible si garantiza encontrar el camino a la solución con el mínimo costo.

Parte de un nodo origen e intenta dirigirse al destino por el camino más corto, para eso, visita a su vecino que dé mejor estimación para la solución buscada. En la literatura este tipo algoritmos tienen asociadas dos funciones: *g(n)* y *h(n)* siendo n el vértice actual. La primera función, *g(n)*, define el costo del camino desde el vértice inicial al vértice actual (el valor |C| definido anteriormente o alguna generalización del mismo para un camino C que va desde el vértice inicial la actual). La segunda, *h(n)*, es una función heurística que estima el costo del vértice actual al final. Asociada a la ruta hay un coste que denominaremos *f(n).* Tenemos entonces:

f (n) = g(n) + h(n)

- *g(n) =* costo de llegar desde el nodo inicial al nodo actual n. Representa el coste real del camino recorrido para llegar a dicho nodo, *n.*
- *h(n) =* costo adicional para llegar desde el nodo actual al estado objetivo. Representa el valor heurístico del nodo a evaluar desde el actual, n, hasta el final. Es la estimación de lo que queda por recorrer.

El algoritmo, como hemos comentado, sigue el orden del siguiente vecino más cercano en el sentido que al ruta que pasa por él es la más corta. En caso de que dos vértices definan rutas igual de largas según la función f(n) entonces se prefiere el vértice con menor h(n).

Los algoritmos *A** tienen asociado el concepto de **admisibilidad**. Si asumimos que *h*(n)* es el coste real del camino más corto desde el vértice actual al vértice final entonces y *h(n*) cumple la condición $h(n) \leq h^*(n)$ decimos que el algoritmo A* es admisible.

Los algoritmos *A** que cumplen esta condición, es decir *h(n)* nunca sobreestima *h*(n),* son denominados algoritmos **admisibles.** Si el algoritmo no es admisible no podemos garantizar que encuentre la solución óptima.

Si *h(x)* satisface la condición adicional $h(x) \leq d(x,y) + h(y)$, para todo *x* e *y* conectados por la arista de longitud *d(x,y)*, la llamaremos **monótona o consistente**. Si *h(n)* es consistente entonces el algoritmo se puede implementar de tal forma que cada vértice sea procesado una sola vez. Los vértices ya procesados constituyen el conjunto de vértices cerrados o visitados.

Si en los algoritmos *A** hacemos *h(n) =0* y las aristas tienen pesos mayor o igual a cero la heurística es consistente. Este caso particular es el *Algoritmo de Dijkstra.*

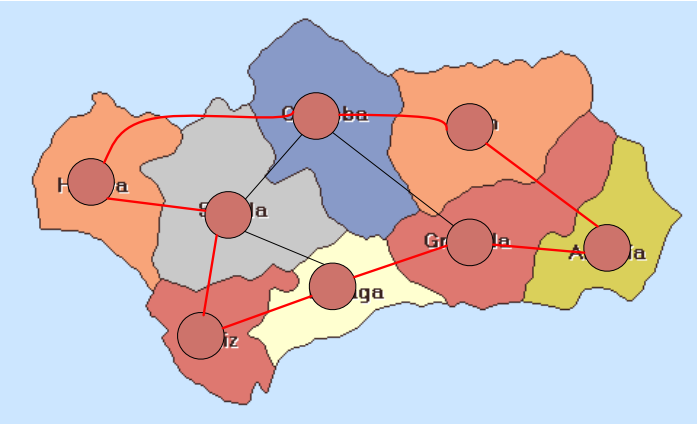
Estos algoritmos son muy adecuados para resolver problemas de caminos mínimos en grafos virtuales.

7.5 Problema del viajante

Es un problema muy conocido en teoría de grafos.

Se denomina **Problema del Viajante** a encontrar un camino cerrado en un grafo que pase por todos los vértices del grafo una sola vez y tenga la longitud mínima. El algoritmo **HamiltonianCycle()** permite calcular una aproximación de la solución del problema. La implementación concreta requiere que el grafo sea completo y se verifique la desigualdad triangular. Es decir para tres vértices *x, y, z* entonces *d(x,y)+d(y,z) > d(x,z).*

HamiltonianCycle()
List<V> getApproximateOptimalForCompleteGraph(SimpleWeightedGraph<V,E> g)



La forma de obtener un grafo completo que sea una vista de otro que no lo es se deja como ejercicio.

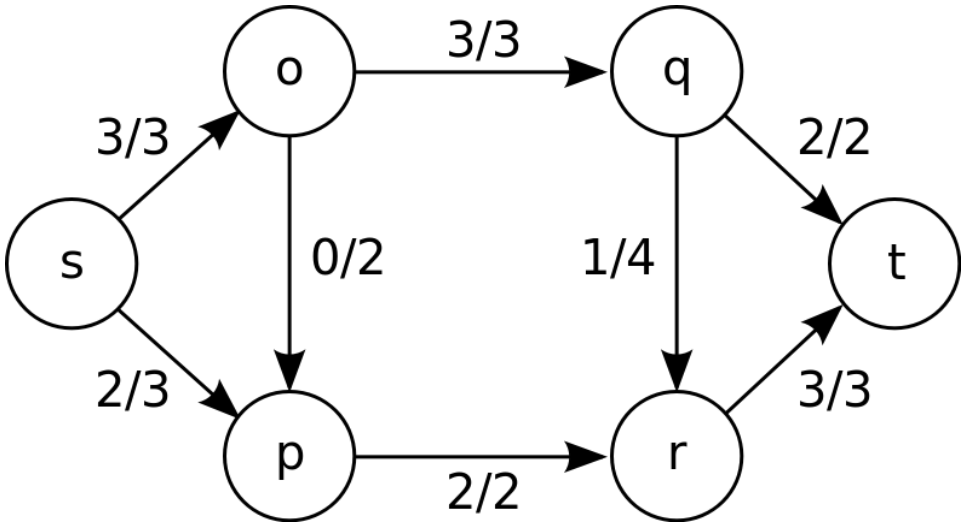
8. Redes de flujo

Una **red de flujo** es un grafo dirigido donde cada arista tiene una capacidad (que debe ser no negativa) y recibe un flujo que debe ser menor o igual a esa capacidad. Un flujo sobre una red de flujo debe satisfacer la siguiente restricción: la cantidad de flujo que llega a un vértice debe ser igual al que sale del mismo excepto cuando es un vértice fuente o un vértice sumidero. Las restricciones del flujo en cada vértice son denominadas restricciones de *Kirchorff*. Los de producen flujo se llaman fuentes y los que consumen sumideros.

8.1 Flujo máximo

El problema del **Flujo Máximo** calcula el máximo flujo que puede fluir por cada arista desde una fuente hasta un sumidero con las restricciones en las capacidades máximas en las aristas. Los datos de partida son los pesos de las aristas del grafo dirigido que representan la capacidad máxima de las mismas. El algoritmo de **EdmondsKarp** resuelve el problema. La complejidad de este algoritmo tiene la cota superior *VE^2.*

EdmondsKarpMaximumFlow(DirectedGraph<V,E> network)	
void calculateMaximumFlow(V source, V sink)	Calcula el flujo máximo desde *source* a *sink*
V getCurrentSink()	Sumidero actual
V getCurrentSource()	Fuente actual
Map<E,Double> getMaximumFlow()	Flujo máximo por cada una de las aristas
double getMaximumFlowValue()	Flujo máximo

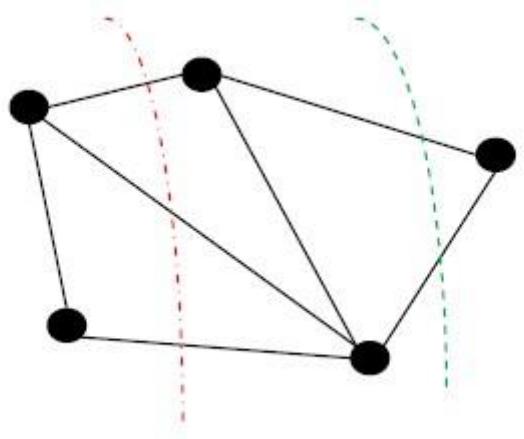


Por ejemplo, suponga que se tienen distintas máquinas conectadas por distintas tecnologías de redes con distintos anchos de banda. Este algoritmo permite calcular cuál sería el máximo ancho de banda posible entre dos equipos, teniendo en cuenta el ancho de banda de todos los posibles nodos intermedios.

8.2 Corte mínimo

Un corte en una red de flujo es una partición de sus vértices en dos conjuntos disjuntos tal que una de ellas contenga el vértice fuente y la otra el vértice sumidero. El peso del corte es la suma de los pesos de las asistas que van del primer conjunto al segundo.

El **problema del corte mínimo** es encontrar el corte con el peso mínimo en una red de flujo.



La clase **StoerWagnerMinimumCut<V,E>** es una implementación del algoritmo que busca el corte mínimo**.** Esta implementación tiene complejidad $|V||E|log|E|$.

StoerWagnerMinimumCut (DirectedGraph<V,E> network)	
Set<V> minCut()	Conjunto de vértices de un lado del corte mínimo
double minCutWeight()	Peso del corte mínimo
double vertexWeight(V v)	Suma de los pesos de las aristas que entran en *v*

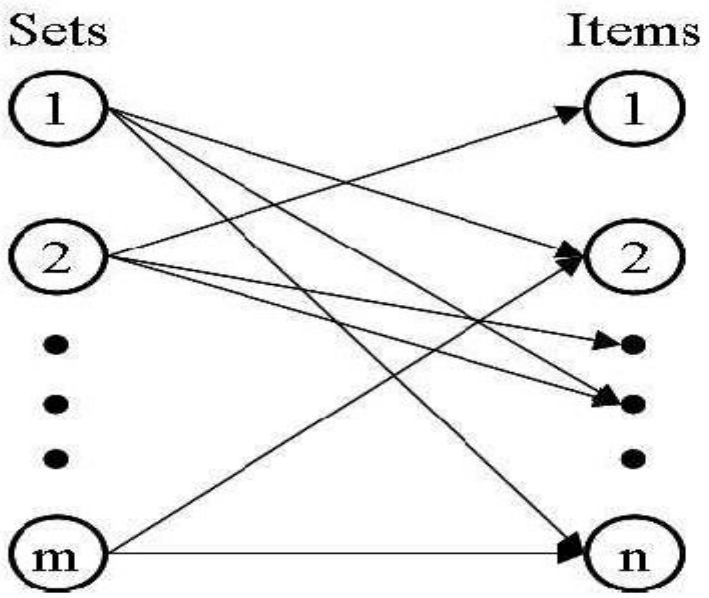
Algunas cuestiones sobre los cortes en una red de flujo:

- Un corte en una red de flujo R representada por un grafo $G = (V, E)$ lo designaremos como (S, T). Dónde S y T son los dos subconjuntos de vértices que definen el corte y $S \cup T = V$, $S \cap T = \emptyset$. Dónde V son todos los vértices del grafo.
- La capacidad de un corte la designaremos por $c(S, T)$. La capacidad de un corte es la suma de las capacidades de las aristas que salen de vértices en S y entran en vértices en T.
- De entre todos los cortes hay uno con capacidad mínima que designaremos por (S', T').
- Designaremos por $f(R), \max(R)$ un el flujo posible en la red de flujo R y el máximo flujo posible.
- Designaremos por $f(S, T)$ el flujo por el corte correspondiente
- Siendo R una red de flujo y (S, T) un corte se cumple $f(R) = f(S, T)$ para cualquier flujo y también para el flujo máximo.
- Siendo R una red de flujo y (S', T') su corte mínimo se cumple $max(R) = f(S', T')$.

8.3 Aplicaciones del corte mínimo: El problema de la selección

El problema de la selección se puede enunciar de la siguiente forma:

Dado un conjunto $L = \{1, \dots, m\}$ de proyectos tal que escoger uno de ellos supone un beneficio b_i y un conjunto $R = \{1, \dots, n\}$ de herramientas tales que escoger una de ellas significa un coste c_j . El problema tiene la restricción adicional que escoger un proyecto de L nos obliga a escoger también un subconjunto de herramientas de R. Esa restricción la podemos representar mediante un grafo bipartito dirigido de la forma (L, R vienen representados por Items, Sets):



El problema es encontrar un subconjunto $A \subseteq L$ que maximice el beneficio total

$$\max_A \left(\sum_{i \in A} b_i - \sum_{j \in \Gamma(A)} c_j \right)$$

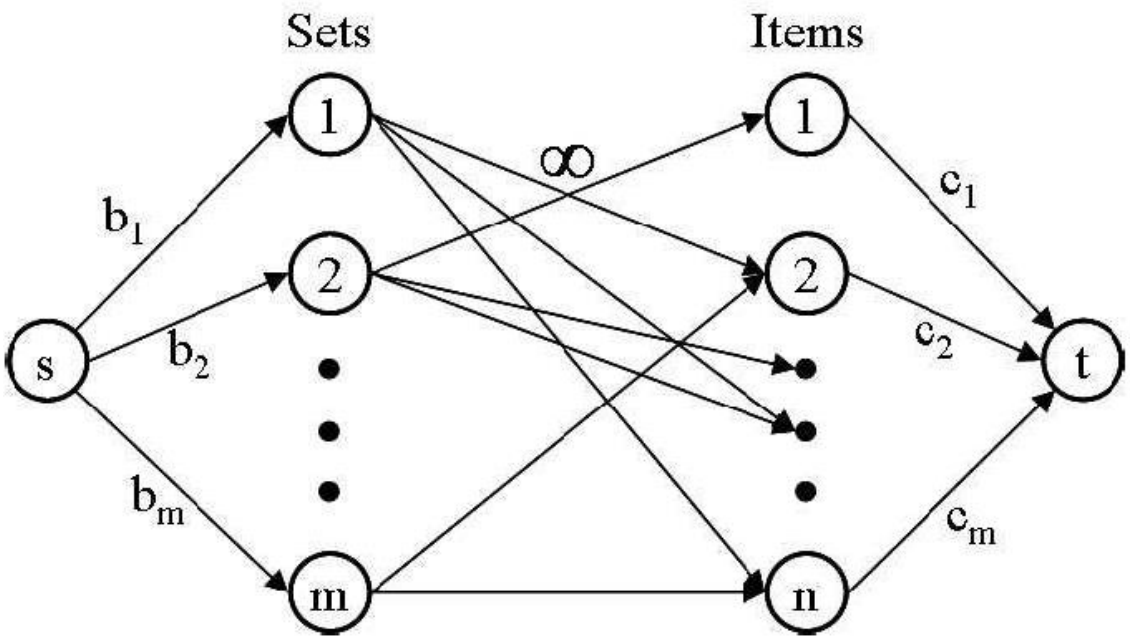
Dónde $\Gamma(A)$ es el conjunto de herramientas requeridas por los proyectos en A. De lo anterior vemos que:

$$\max_A \left(\sum_{i \in A} b_i - \sum_{j \in \Gamma(A)} c_j \right) = \max_A \left(\sum_{i=1}^{m} b_i - \sum_{i \notin A} b_i - \sum_{j \in \Gamma(A)} c_j \right) = \max_A \left(C - \sum_{i \notin A} b_i - \sum_{j \in \Gamma(A)} c_j \right)$$

Y por lo tanto el conjunto que maximiza la expresión anterior es el mismo que minimiza

$$\min_A \left(\sum_{i \notin A} b_i + \sum_{j \in \Gamma(A)} c_j \right)$$

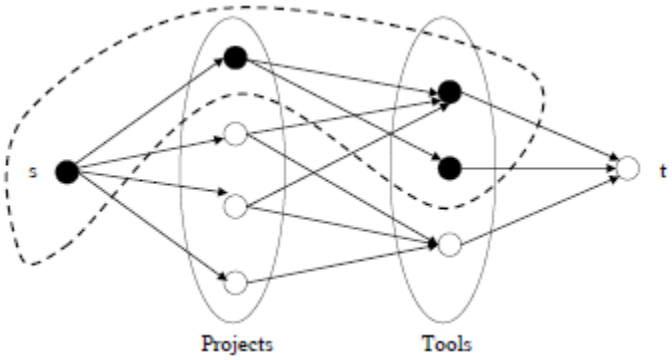
El problema puede ser reducido a un problema de corte mínimo añadiendo los vértices s, t y las restricciones en la forma:



En la red anterior un corte (S, T) de capacidad finita, como el que se muestra en la siguiente figura, no puede incluir aristas de capacidad infinita. Cada corte de capacidad finita define de manera única un subconjunto de proyectos A y de herramientas B de la forma $A = S \cap L$, $\Gamma(A) \subset B = S \cap R$.

Los cortes de capacidad mínima cumplirán, además, $\Gamma(A) = B$ y por lo tanto su capacidad será, por lo visto arriba

$$c(S,T) = \sum_{i \notin A} b_i + \sum_{j \in B} c_j$$



Por lo tanto la solución del problema es encontrar el corte mínimo en la red anterior y posteriormente los conjuntos A, B como la intersección del conjunto S definido por el corte mínimo con los conjuntos L, R.

8.4 Problema de red de flujo

El **problema de red de flujo** es una generalización del problema del de flujo máximo visto anteriormente. En este problema junto al flujo máximo permitido en cada arista se puede establecer un flujo mínimo en cada una de ellas. Además se fija un coste unitario para transportar el flujo por cada arista. Se generaliza el concepto de fuente y sumidero para permitir varias fuentes y varios sumideros. Un vértice puede ser, entonces, fuente, sumidero o intermedio. En los vértices fuentes y sumideros se pueden añadir restricciones sobre el flujo máximo o mínimo producido o consumido y un coste unitario. En los vértices intermedios se puede añadir la restricciones sobre le flujo que los cruza (igual a la suma del flujo de las aristas entrantes) y un coste unitario. Las restricciones del problema son una generalización de las leyes de *Kirchoff* vistas arriba. Como antes a cada arista se asocia un flujo. A los vértices fuentes un flujo naciente y a los sumideros un flujo consumido. Los vértices intermedios tienen asociado un flujo que circula por ellos igual a la suma de los flujos de las aristas entrantes. Asumimos para cada vértice intermedio la suma de los flujos entrantes menos los salientes es igual a cero. En los vértices fuente la suma de los flujos entrantes más el flujo producido en el vértice menos los salientes es igual a cero. Y en los vértices sumideros la suma de los flujos entrantes menos los salientes y menos el flujo producido en el vértice es igual a cero. La función a optimizar es la suma de los flujos sobre cada arista por su precio unitario más el flujo producido o consumido en los vértices respectivos por su coste unitario. El problema tiene como objetivo minimizar esa función objetivo. De la misma forma se puede definir un problema que maximice ese coste.

8.5 El tipo FlowGraph<V,E> y su transformación en un problema de Programación Lineal

Partimos de un grafo dirigido que tiene que verificar las siguientes propiedades:

- Un vértice fuente no tiene aristas de entrada
- Un vértice sumidero no tiene aristas de salida

A partir de un grafo de flujo se puede construir un Problema de Programación Lineal. Este problema, posteriormente puede resolverse mediante el algoritmo del Simplex o, si añadimos restricciones tales que algunas variables tomen valores enteros, mediante un algoritmo de Programación Lineal Entera.

Las ideas para hacer esa transformación son:

- Asociar una variable a cada arista para representar el flujo que pasa por la arista
- Asociar una variable a cada vértice fuente para representar el flujo que se crea
- Asociar a cada vértice sumidero una variable para representar el flujo que se consume
- El número de variables obtenido es igual al número de aristas más el número de vértices fuente, más el número de vértices sumidero.
- Generar las restricciones de flujo asociadas a cada vértice (leyes de *Kircchoff*):
    - Si el vértice es intermedio entonces el flujo de entrada es igual al flujo de salida
    - Si el vértice es fuente entonces el flujo creado es igual al flujo de salida.
    - Si el vértice es sumidero el flujo de entrada es igual al flujo consumido.
- Generar las restricciones asociadas a aristas, flujo en fuentes y sumideros y flujo que pasa por los vértices intermedios

    - El flujo que pasa por cada arista puede tener una cota inferior y una cota superior
    - Los flujos creados y consumidos en los vértices fuente y sumideros pueden tener una cota inferior y otra superior
    - Los flujos que pasan por los vértices intermedios, igual al flujo entrante a los mismos, puede tener una cota inferior y otra superior.
- Generar la función objetivo como la suma de los siguientes costes:
    - Coste asociado al flujo por cada arista
    - Coste asociado al flujo que se crea el los vértices fuente
    - Coste asociado al flujo que se consume en los vértices sumidero
    - Coste asociado al flujo que pasa por un vértice intermedio. Es igual al flujo de entrada en este tipo de vértices por su coste unitario.
- Maximizar o minimizar la función objetivo según se prefiera

Denominamos, en general Problema de *Programación Lineal* al que puede ser especificado mediante:

- Un conjunto de variables que toman valores en el domino de los número reales
- Un función objetivo que pretende minimizar o un expresión lineal de la forma

$$\min/\max \sum_{i=0}^{n-1} x_i v_i$$

- Un conjunto de restricciones lineal sobre las variables del problema de la forma

$$\sum_{i=0}^{n-1} x_i w_i \leq c$$

- Si existen restricciones adicionales sobre los tipos de las variables (variables binarias, enteras, etc). Entonces el problema se denomina de Programación Lineal Entera.

Este tipo de problema puede ser resuelto de forma muy eficiente por el conocido algoritmo del *Simplex*.

De forma compacta el problema de Programación Lineal asociado al grafo de flujo se puede escribir de la siguiente forma. Asumamos que V es el conjunto de vértices, F el conjunto de vértices fuente, S el conjunto de vértices sumidero, $I = V - F \cup S$ el conjunto de vértices intermedios, E el conjunto de aristas. Asumimos un número entero asociado a cada vértice y arista. El costo unitario asociado al paso del flujo por la arista j lo representamos por $w_j, j \in E$. El costo unitario asociado a la producción o consumo o paso de una unidad por el vértices $i \in V$ lo representaremos por w_i, a las cotas superiores e inferiores en vértices y aristas los representamos respectivamente por $b_i^u, b_i^d, b_j^u, b_j^d$, el flujo creado en vértices fuente o consumido en vértices sumidero x_i y el flujo por una arista y_j. Las ecuaciones resultantes son:

$$\min \sum_{i \in V} w_i x_i + \sum_{j \in E} w_j y_j$$

$$x_i = \sum_{k \in Out(i)} y_k, \quad i \in F$$

$$\sum_{k \in In(i)} y_k = x_i, \quad i \in S$$

$$\sum_{k \in In(i)} y_k = \sum_{k \in Out(i)} y_k, \quad i \in I = V - F \cup S$$

$$b_i^d \leq x_i \leq b_i^u, \qquad i \in F \cup S$$

$$b_j^d \leq y_j \leq b_j^u, \qquad j \in E$$

$$b_i^u \leq \sum_{k \in In(i)} y_k \leq b_i^u, \qquad i \in I = V - F \cup S$$

La información necesaria para definir problema de redes de flujo podemos incluirla en un fichero y a partir de él construir un grafo con la información adecuada. Diseñamos para ello el tipo **FlowGraph<V,E>** que tiene las siguientes propiedades**:**

La información anterior podemos añadirla a la información disponible en el grafo. Para ellom definimos el tipo *FlowGraph*:

```
public interface FlowGraph<V, E> extends Graph<V,E> {
	double getMinEdgeWeight(E edge);
	double getUnitEdgeWeight(E edge);
	boolean isSource(V vertex);
	boolean isSink (V vertex);
	double getMaxVertexWeight(V vertex);
	double getMinVertexWeight(V vertex);
	double getUnitVertexWeight(V vertex);
}
```

El peso de la aristas *(getEdgeWeight(e))* contiene la cota máxima del flujo sobre un arista.

La clase **FlowAlgorithm<V,E>** se ha diseñado para convertir un grafo que representa una red de flujo en un problema de Programación Lineal.

Esta clase calcula asocia un entero único a cada vértice y arista. A partir de ahí define una única variable asociada a cada vértice y arista. Posteriormente calcula las restricciones asociadas para construir el Problema de *Programación Lineal*. Este problema se podría completar con restricciones adicionales no definidas en el grafo u otras sobre el domino de las variables para construir un *Problema de Programación Lineal Entera*. Esto puede ser necesario si queremos que la solución buscada al problema de flujo esté en el dominio de los enteros.

Algunos métodos de esta clase son:

FlowAlgorithm(FlowGraph<V,E > network).

(FlowGraph<V,E> network, boolean min) True si se trata de minimizar.

FlowAlgoritm create(FlowGraph grafo)	Crea el algoritmo
FlowAlgoritm create(FlowGraph grafo, boolean min)	Crea el algoritmo
ProblemaPL getProblemaLP()	Construye el problema equivalente
BiMap<Integer,VertexOrEdge> getInverseIndex()	Map entero vértice/arista
String toStringIndex()	Indices
Map<E,Double> getEdgeFlow(AlgoritmoPL a)	Flujo por cada una de las aristas
Map<V,Double> getSourceFlow(AlgoritmoPL a)	Flujo en los vértices fuente
Map<V,Double> getSinkFlow(AlgoritmoPL a)	Flujo en los vértices sumidero
Integer getNumItems()	Número de vértices más número de aristas

Veamos como ejemplo el grafo que se especifica en el siguiente fichero.

```
#VERTEX#
A,0
B,0
C,0,2.,14.,2.
D,0
S,1,0.,inf,1.
T,2,0.,inf,-1.
#EDGE#
A,C,0.,10.,1.
C,A,0.,4.,1.
A,B,1.,12.,1.
D,B,3.,7.,1.
B,C,2.,9.,1.
C,D,2.,14.,1.
S,A,0.,16.,1.
S,C,0.,13.,1.
B,T,0.,20.,1.
D,T,0.,4.,1.
```

Cada vértice tiene la siguiente información:

- *id,tipo,min,max,coste*

Que representa un identificador único para e vértice, el tipo de vértice (0, intermedio, 1, fuente, 2, sumidero).

Si el vértice es intermedio entonces puede indicarse solamente

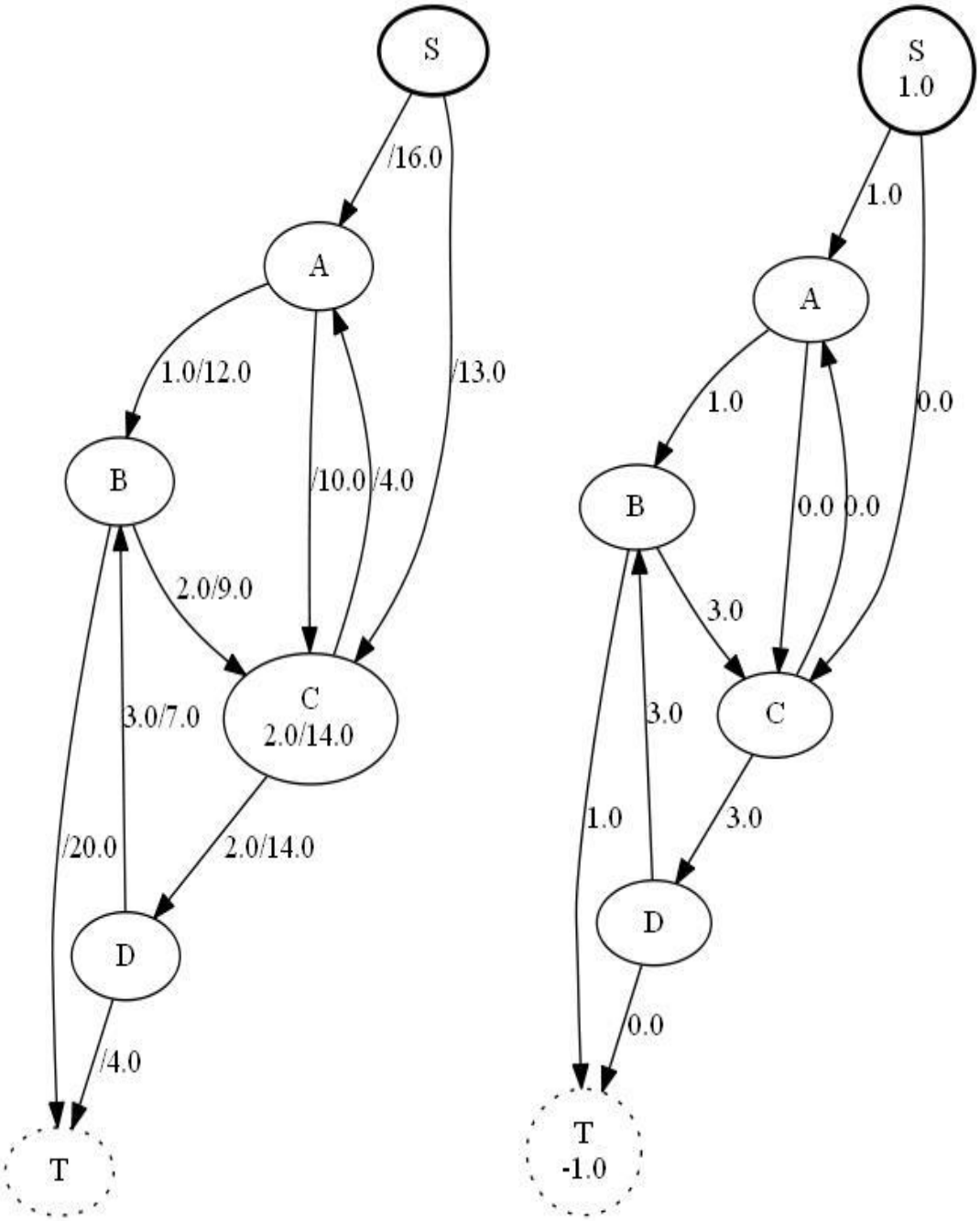
- *id,0*

Asumiendo que significa

- *id,tipo,0,inf,0*

Infinito ha sido representado en le fichero por la cadena *inf*.

Los grafos del problema y de la solución son:



La numeración asignada a las aristas, los vértices fuente y sumidero y las restricciones asociadas son:

```
min:  +3.0 x0 +x1 +x2 +x3 +3.0 x4 +x5 +x6 +3.0 x7 +x8 +x9 +x10 -x11;

 +x0 <= 10.0;
 +x1 <= 4.0;
 +x2 <= 12.0;
 +x2 >= 1.0;
```

```
 +x3 <= 7.0;
 +x3 >= 3.0;
 +x4 <= 9.0;
 +x4 >= 2.0;
 +x5 <= 14.0;
 +x5 >= 2.0;
 +x6 <= 16.0;
 +x7 <= 13.0;
 +x8 <= 20.0;
 +x9 <= 4.0;
 -x0 +x1 -x2 +x6 = 0.0;
 +x2 +x3 -x4 -x8 = 0.0;
 +x0 -x1 +x4 -x5 +x7 = 0.0;
 +x0 +x4 +x7 <= 14.0;
 +x0 +x4 +x7 >= 2.0;
 -x3 +x5 -x9 = 0.0;
 -x6 -x7 +x10 = 0.0;
 +x8 +x9 -x11 = 0.0;

Indices
6,S--A
7,S--C
0,A--C
8,B--T
1,C--A
3,D--B
5,C--D
11,T
2,A--B
4,B--C
9,D--T
10,S
```

Si estamos interesados en definir directamente un problema de Programación Lineal usaremos el tipo IProblemaPL y la clase ProblemaPL que lo implementa con los métodos:

```
public interface IProblemaPL {

	public TipoDeOptimizacion getTipo();
	public LinearObjectiveFunction getObjectiveFunction();
	public void setObjectiveFunction(LinearObjectiveFunction
			objectiveFunction);
	public void setObjectiveFunction(double[] coefficients, double
			constantTerm);
	public Collection<LinearConstraint> getConstraints();
	public void addConstraint(LinearConstraint lc);
	public void addConstraint(double[] coefficients, Relationship
		relationship, double value);
	public void addSosConstraint(List<Integer> ls, Integer nv);
	public Integer getGetNumOfVariables();
	public void setTipoDeVariable(int e, TipoDeVariable tipo);
	public void setTipoDeTodasLasVariables(TipoDeVariable tipo);
	public void setVariableLibre(int e);
	public void setVariableSemicontinua(int e);
	public List<Integer> getVariablesEnteras();
	public List<Integer> getVariablesBinarias();
	public void setNombre(Integer e, String s);
	public String getNombre(Integer e);
```

```
        public List<Integer> getVariablesLibres();
        public List<Integer> getVariablesSemicontinuas();
        public String toStringConstraints();
        public void toStringConstraints(String fichero);
}
```

La descripción más detallada de los métodos puede encontrarse en clase ProblemaPL.

Una vez construido el problema mediante le tipo anterior disponemos de dos algoritmos para buscar las soluciones:

- AlgoritmoPL. Que implementa el método del Simplex reutilizando las librerías de Apache.
- AlgoritmoPLI. Que implementa un algoritmo de Programación Lineal Entera reutilizando las librerías de LpSolve.
- Algoritmos. La factoría anterior dispone de los métodos siguientes:

```
        public static AlgoritmoPL createPL(ProblemaPL p) {
                return AlgoritmoPL.create(p);
        }
        public static AlgoritmoPL createPL(ProblemaPL p, String fichero) {
                return AlgoritmoPL.create(p);
        }
        public static AlgoritmoPLI createPLI(String fichero) {
                return AlgoritmoPLI.create(fichero);
        }
        public static AlgoritmoPLI createPLI(ProblemaPL p,  String fichero) {
                return AlgoritmoPLI.create(fichero);
        }
```

Con el primero a partir de un objeto del tipo *ProblemaPL* se instancia un algoritmo del Simplex para resolverlo. Se ignoran todas las restricciones que no sean específicas de Programación Lineal. Por ejemplo la declaración de variables enteras o binarias. Con el parámetro fichero se indica el nombre de un fichero para guardar las restricciones generadas.

El tercer método toma un fichero de entrada escrito en el formato adecuado para ser procesado por LpSolve e instancia un algoritmo para resolverlo. El formato de este tipo de fichero puede verse en formato LpSolve.

Alternativamente podemos resolver el problema de Programación Lineal a partir de un *ProblemaPL* con todas sus restricciones.

Las restricciones asociadas a un problema *ProblemaPL* pueden generarse y guardarse en un fichero mediante el método:

```
public void toStringConstraints(String fichero);
```

En capítulos posteriores veremos más detalles de *Programación Lineal* y *Programación Lineal Entera.*

8.6 Problemas relacionados con una red de flujo

El problema del **flujo máximo** es un problema particular que pretende maximizar el flujo desde un vértice o un conjunto de vértices a un conjunto de destinos. El problema se modela dando a cada arista un flujo mínimo de cero, máximo el dado por la arista y coste cero. El vértice origen es un vértice fuente sin cota superior y coste cero. Cada vértice destino lo modelamos como un vértice consumidor sin cota superior y coste -1. Al ser los vértices destinos consumidores tratamos de obtener el máximo flujo posible y para ello escogemos minimizar la función objetivo asociada al problema. Una vez encontrado el flujo máximo por cada arista podemos determinar el corte mínimo. Como podemos recordar el corte mínimo dos partía el conjunto de vértices en dos (S, T). El conjunto S está formado por el conjunto de vértices alcanzables desde el vértice fuente a través de aristas que no están saturadas. Es decir aquellas en la que el flujo es menor que la capacidad. El conjunto T es el complementario y el corte las aristas que van de S a T.

El **problema del corte mínimo** se puede resolver como un problema de flujo máximo anterior y posteriormente encontrar un corte cuyas aristas van llenas. Es decir su flujo es igual al máximo permitido.

El problema de los caminos disjuntos en aristas trata de encontrar los caminos, que sin compartir ninguna arista, van de un origen a un destino o aun un conjunto de destinos. Se modela como un problema de flujo máximo con un flujo máximo en cada arista de uno. El número de caminos viene dado por el flujo máximo. Cada camino por el conjunto de aristas que tienen flujo uno. Se puede generalizar para varios vértices origen tomados de un conjunto y para varios vértices destino tomados de otro conjunto.

El **problema de los caminos disjuntos en vértices** trata de encontrar los caminos, que sin compartir ningún vértice (excepto los propios origen y destino), que van de un origen a un destino o un conjunto de destinos. Se modela como un problema de flujo máximo donde, además de añadir a cada arista una cota superior de uno, añadimos a cada nodo intermedio (distinto al origen y destino) una restricción sobre el flujo que lo cruza con cota superior uno, cota inferior cero y coste cero. El número de caminos viene dado por el flujo máximo. Cada camino, como antes, viene dado por el conjunto de aristas que tienen flujo uno.

El **problema de la conectividad de redes** trata de buscar el mínimo número de aristas tales que eliminadas desconectan un vértice de un destino. Equivale al problema de corte mínimo con pesos de aristas igual a uno.

El **problema de transporte** es otro caso particular. Se trata de enviar un flujo desde un conjunto de vértices productores a otro de vértices consumidores con un coste unitario asociado al transporte desde un vértice a otro. En este problema los costes y beneficios unitarios de los vértices son cero. Este problema se puede generalizar considerando vértices intermedios que no producen ni consumen flujo.

El **problema de la asignación** trata de asignar *n* personas a *m* tareas y *n>=m*. Se supone que el coste de realizar la tarea *j* por la persona *i* es *c(i,j).* Se trata de decidir qué persona hace cada

tarea para minimizar la suma de los costes respectivos. El problema se modela mediante una red de flujo cuyos vértices son las personas y las tareas. Una arista dirigida desde cada persona a cada tarea con coste el coste de la asignación correspondiente. Máximo flujo igual a uno. Las personas las modelamos como vértices productores con máximo flujo igual a uno, y mínimo igual a cero y coste cero. Las tareas las modelamos como vértices consumidores con máximo y mínimo flujo igual a uno y coste cero. La solución viene dada por las aristas con flujo uno.

El **problema de camino mínimo** (un caso particular del problema del camino mínimo que se resuelve por el algoritmo de *Dijkstra* visto arriba) trata de busca el camino de longitud mínimo desde un vértice origen a un vértice destino en ese tipo de grafos. Puede ser modelado mediante una red de flujo. Se trata de asignar a cada arista un cota superior uno, inferior cero y un coste igual a la longitud de la arista. El vértice origen lo modelamos como un vértice productor con coste cero y capacidad de producción mínima y máxima uno. El vértice destino lo modelamos como un vértice consumidor con coste cero y capacidad de consumo mínima y máxima uno. La solución del problema (el camino mínimo viene dado por la secuencia de aristas que tengan flujo uno).

Otra variante es encontrar el camino mínimo desde un vértice origen a cada uno de *n* destinos. El problema se modela como antes. Cada nodo destino vértice consumidor con coste cero, capacidad de consumo máxima y mínima de uno. El nodo origen productor con coste cero y capacidad de producción mínima y máxima igual a *n*.

9. Coloreado de Grafos

Dado un grafo se define su **número cromático** como el mínimo número de colores necesario para dar un color a cada vértice de tal forma que dos vértices vecinos tengan colores distintos.

La clase **CromaticNumber** hace una implementación de este algoritmo**.**

CromaticNumber()
static <V,E> int findGreedyChromaticNumber(UndirectedGraph<V,E> g)
static <V,E> Map<Integer,Set<V>> findGreedyColoredGroups(UndirectedGraph<V,E> g)

Los dos métodos implementan algoritmos de aproximación para resolver el problema. El primero devuelve el número cromático (número de colores distintos). El segundo devuelve para cada posible color el conjunto de vértices que lo tienen asignado. Cada posible color viene representado por un entero *0, 1, …*

9.1 Aplicaciones del coloreado de grafos

Scheduling

Problemas donde se trata de programar la ejecución de un conjunto de tareas en distintas franjas horarias. Las tareas pueden ser ejecutadas en cualquier orden pero hay pares de tareas que tienen recursos compartidos. Estos problemas se pueden representar mediante un grafo

no dirigido con un vértice por cada tarea y una arista entre aquellos vértices que tengan recursos compartidos. El problema se puede resolver mediante coloreado de grafos. El número cromático nos dará el número mínimo de franjas horarias necesarias y cada grupo de vértice con el mismo color representa tareas que se pueden ejecutar en la misma franja horaria.

Sudoku

El Sudoku es un problema conocido. Un ejemplo se muestra abajo. En un Sudoku las diferentes casillas no en la misma fila, en la misma columna, o en la misma caja deben tener distinto valor asociado. Cada casilla debe tener un valor del 1 al 9.

Para ver la conexión entre Sudoku y coloreado de grafos, primero describiremos un grafo asociado al Sudoku. A cada casilla se le asocia un vértice. El grafo tiene 81 vértices. Cuando dos casillas no pueden tener el mismo valor (ya sea porque están en la misma fila, en la misma columna, o en la misma caja) añadimos una arista entre los vértices asociados.

Asignamos arbitrariamente un número (del 1 al 9) a cada color. Coloreamos el grafo anterior partiendo de la base de que ya algunos vértices tienen un color. Los vértices con el mismo color pueden tener el mismo número. Los vértices con distinto color deben tener distinto valor asociado. Si dentro de los vértices con el mismo color ya hay uno con color previamente definido los demás toman el mismo color. El grafo se colorea con 9 colores diferentes.

Un ejemplo de Sudoku sería:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

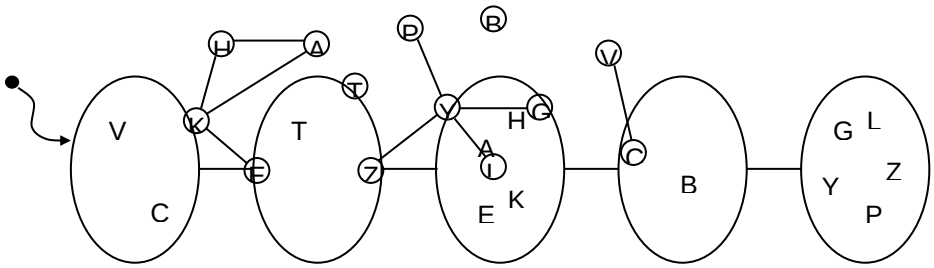
Problema de la Reinas

Con las mismas ideas se podría resolver el problema de las 8 reinas. El problema consistente en colocar 8 reinas en un tablero de ajedrez sin que ninguna de ellas amenace a otra. A partir del problema definimos un grafo cuyos vértices son cada una de las casillas. Añadiremos una arista entre dos vértices si ubicadas dos reinas en las correspondientes casillas se amenazan entre sí.

Coloreamos el grafo anterior. Si escogemos una casilla en cada fila pero todas del mismo color obtendremos la solución del problema.

10. Ejercicios resueltos

1: Desarrolle un método que, dado un grafo explícito no dirigido devuelva una lista de conjuntos de vértices donde cada elemento de la lista sea el conjunto de todos los vértices de una componente conexa distinta (una componente conexa es un conjunto maximal de vértices alcanzables); entre todos los conjuntos de la lista deben estar todos los vértices del grafo. Por ejemplo, para el grafo



La lista podría ser (el orden es indiferente):

```
public <V, E> List<Set<V>> conjuntos(UndirectedGraph<V, E> g) {

        ConnectivityInspector<V, E> ci =
             new ConnectivityInspector<V, E>(g);

        return ci.connectedSets();
}
```

2: En un juego, existen una serie de planetas. Cada planeta produce un recurso (por ejemplo R) y consume una serie de recursos (por ejemplo C01 y C02).

- Escriba un método que, admita como parámetro un grafo en el que los vértices representan a los planetas, y un recurso R. El método devolverá un conjunto con aquellos planetas que consumen el recurso R.
- Responda a la siguiente pregunta. ¿qué estructura auxiliar se podría utilizar para conocer en todo momento los planetas que consumen un recurso sin necesidad de buscarlos en el grafo?
- Los planetas están conectados mediante agujeros de gusano (aristas). Cada agujero de gusano tiene su propio coste que se ha de pagar para viajar por él. Escriba un método para el que dado un grafo G, y un planeta P, devuelva el camino con menor coste hasta un planeta (cualquiera) que consuma el recurso producido por P.
- Escriba un último método que reciba como parámetro un grafo y un planeta de origen P, dos planetas cuales quiera P01 y P02 y un coste C. el método deberá devolver cierto si al añadir un agujero de gusano entre P01 y P02 con un coste de viaje de C, el coste del camino con el coste mínimo a un planeta que consuma el recurso que produce P es más pequeño que el coste del menor coste sin que exista ese agujero de gusano y falso en caso contrario.

Ojo, después de la ejecución del método, el grafo debe ser igual a como lo era antes de la ejecución del método.

Interfaz Planeta:

Recurso getRecursoProducido()
List<Recurso> getRecursosConsumidos()
boolean equals(Object o)

Por simplicidad, se ha utilizado la clase String para modelar los recursos. En el primer método se utiliza un filtro, definido dentro del propio método, para encontrar los planetas que consumen un recurso determinado.
Una buena estructura auxiliar sería utilizar un Multimap en el que, para cada recurso (clave) tuviéramos almacenado la colección de planetas que consumen dicho recurso. En esta solución se incluye una Function que permite crear dicho Multimap a partir de un grafo.
El tercer método aplica el algoritmo de Dijsktra para encontrar los caminos qu hay desde el planeta origen hasta todos los planetas que consumen ese recurso. Después se queda con el camino más corto encontrado. Como ejercicio adicional, pruebe a resolver este problema utilizando el algoritmo de Floyd-Warshall en vez de Dijsktra.
En el último método se utiliza el algoritmo de Dijsktra para calcular el camino más corto, después, se añade una nueva arista al grafo y s e vuelve a aplicar el algoritmo de Dijsktra para ver si el camino ahora es más barato. Por último se elimina la arista añadida para el grafo quede igual que al principio.

```
	public Set<Planeta> planetsConsumig(String r) {

		class FilterByConsumedResource implements Predicate<Planeta> {
			String r;

			public FilterByConsumedResource(String r) {
				this.r = r;
			}

			@Override
			public boolean apply(Planeta arg0) {
				return arg0.getRecursosConsumidos().contains(r);
			}
		}

		return Sets.filter(wg.vertexSet() , new
				FilterByConsumedResource(r));
	}

	class Set2Multimap implements Function<Set<Planeta>, Multimap<String,
				Planeta>> {

		@Override
		public Multimap<String, Planeta> apply(Set<Planeta> arg0) {
			Multimap<String, Planeta> mm = HashMultimap.create();
			for (Planeta p: arg0) {
				for (String r: p.getRecursosConsumidos()) {
					mm.put(r, p);
				}
			}

			return mm;
		}
	}

	public Double
minimumCostToConsumingPlanet(SimpleWeightedGraph<Planeta,
				DefaultWeightedEdge> wg ,Planeta p) {
		Set<Planeta> pConsuming =
			this.planetsConsumig(p.getRecursoProducido());
		assert pConsuming .size()> 0;

		Double min = Double.MAX_VALUE;
		DijkstraShortestPath<Planeta, DefaultWeightedEdge> aDisjktra;
		Double l;

		for (Planeta pDes: pConsuming) {
```

```
                aDisjktra =
                        new DijkstraShortestPath<Planeta,
                                DefaultWeightedEdge>(wg, p, pDes);
                        l =  aDisjktra.getPathLength();
                        assert l < Double.MAX_VALUE;

                        if (l < min) min = l;
        }


        return min;
}

public Boolean decreaseCost(SimpleWeightedGraph<Planeta,
                        DefaultWeightedEdge> wg ,
                Planeta pOrg, Planeta pA, Planeta pB, double cost) {
        Double cOrg = this.minimngCostToconsumingPlanet(wg, pOrg);
        assert cOrg != -1;
        assert cOrg < Double.MAX_VALUE;

        wg.setEdgeWeight(wg.addEdge(pA, pB), cost);
        Double cNext = this.minimngCostToconsumingPlanet(wg, pOrg);

        wg.removeEdge(pA, pB);


        return cNext < cOrg;
}
```

3: La empresa de mensajería PE sólo tiene sede en algunas ciudades, sin embargo, sirve paquetes a todo el territorio.

Diseñe e implemente una solución (puede ser un único método, dos, una clase, dos, etc.) para que, dado un grafo en el que los vértices son ciudades y el peso de las aristas es el coste de viajar de una ciudad a otra, y un conjunto que contenga las ciudades dónde PE tiene sede, devuelva un Map dónde la clave será cada una de las ciudades del grafo y el valor será la ciudad más cercana dónde PE tenga una sede.

La ciudad más cercana a una ciudad C dónde PE tenga sede, será la propia ciudad C.

En esta solución se ha utilizado una clase que calcula el Map una única vez, la primera vez que se necesita. Para ello se utiliza el algoritmo de Floyd-Warshall ya implementado en JGraphT.

```
class MapShortestCities {

        Map<String, String> m;
        SimpleWeightedGraph<String, DefaultWeightedEdge> wg;

        public MapShortestCities(SimpleWeightedGraph<String,
                        DefaultWeightedEdge> wg) {
                this.wg = wg;
                m = null;
        }


        public Map<String, String> getMap(Set<String> cities) {
                if (cities.isEmpty())
                        return m;
                if (m == null)
                        calculateMap(cities);
```

```
                return m;
        }

        private void calculateMap(Set<String> cities) {
                m = new HashMap<String, String>();
                String sCity = null;
                Double cost;

                FloydWarshallShortestPaths<String, DefaultWeightedEdge>
                      fwAlg =
                       new FloydWarshallShortestPaths<String,
                         DefaultWeightedEdge>(wg);

                for(String city: wg.vertexSet()) {
                       if (cities.contains(city))
                              m.put(city, city);
                       else {
                              cost = Double.MAX_VALUE;
                              for (String myCity: cities) {
                                     if (fwAlg.getShortestPath(myCity,
                                       city).getWeight()<cost) {
                                            cost =
                                fwAlg.getShortestPath(myCity, city).getWeight();
                                            sCity = myCity;
                                     }
                              }
                              m.put(city, sCity);
                       } // else
                } // for
        }
} // Class
```

4: Implemente una función que convierta un grafo sin peso en un grafo exactamente igual pero en el que todas las aristas tengan un peso de 1.0.

A partir de la transformación anterior, implemente un método que, dado un grafo, un vértice origen, un vértice destino y un número de aristas, indique si existe o no un camino que vaya desde el origen al destino pasando como máximo por el número de aristas indicadas.

Con esta Function es posible aplicar algunos algoritmos, como Disjktra, hasta una distancia determinada. Así podríamos resolver problemas del tipo: encontrar el camino más corto de A a B siempre que dicho camino no recorra más de N vértice o N aristas.

```
public class G2WG<V> implements Function<Graph<V,DefaultEdge>,
WeightedGraph<V,DefaultWeightedEdge>>
{
        @Override
        public WeightedGraph<V, DefaultWeightedEdge> apply(
                Graph<V, DefaultEdge> arg0) {

                WeightedGraph<V, DefaultWeightedEdge> wg =
                       new SimpleWeightedGraph<V,
                         DefaultWeightedEdge>(DefaultWeightedEdge.class);

                Graphs.addAllVertices(wg, arg0.vertexSet());
                for (DefaultEdge de: arg0.edgeSet()) {
                       wg.setEdgeWeight(
                         wg.addEdge(arg0.getEdgeSource(de),
                             arg0.getEdgeTarget(de)), 1.0);
                }

                return wg;
```

```
        }

    }

    public <V> boolean isPathWithSize(Graph<V, DefaultWeightedEdge> wg, V
                start, V end, double rad) {

        DijkstraShortestPath<V, DefaultWeightedEdge> aDisjktra =
                new DijkstraShortestPath<V, DefaultWeightedEdge>(wg,
                    start, end, rad);

        return aDisjktra.getPathLength() <= rad;
    }
```

11. Ejercicios propuestos

1: Una red social imaginaria y pequeña, se representa con un grafo en el que los vértices modelan a los miembros de las redes sociales y las aristas (sin etiqueta ni peso) modelan la amistad entre dos miembros. Desarrolle:

a) Un método que devuelva un conjunto con aquellos miembros que no tengan ningún amigo.

b) Un método que, dados dos miembros, devuelva la lista más corta de amigos que hay desde un miembro a otro miembro.

2: En el fichero "topología.txt" se almacena la topología de una red de comunicaciones. Los vértices (etiquetados con objetos de clase String), representan máquinas que se pueden comunicar, y las aristas (etiquetas con valores double), representan el año de banda disponible entre dos máquinas. Desarrolle un método que, dado dos máquinas A y B, encuentre el camino por el que se puede obtener el mayor ancho de banda.

3: Existe una clase CV con los siguientes atributos: {a: String, b: Integer, c: Boolean} y existe una segunda clase CE con los siguientes atributos: {s: String, i: Integer}. También existe un fichero G.txt que contiene la información necesaria para representar un grafo en el que las etiquetas de los vértices sean objetos CV y las etiquetas e las aristas sean objetos CE.

Implemente un método, y todos los elementos auxiliares necesarios, que permita cargar dicho grafo desde el fichero.

4: En un tablero de ajedrez vacío se pintan varias casillas de color rojo y se coloca una reina sobre una de ellas. El objetivo planteado es conseguir que a reina visite todas las casillas rojas moviendo el menos número de casillas posibles.

5: Una empresa de autobuses tiene andenes en algunas ciudades de un país y, el mapa de carreteras de dicho país, está modelado como un grafo en el que los vértices están etiquetados con el nombre de la ciudad y las aristas con la distancia entre dos ciudades.

Desarrolle un método que permita calcular todas las rutas posibles junto con su distancia.

6: Un juego consiste en establecer una red vías entre distintas ciudades de un mapa. Los jugadores compran vías y las colocan en el mapa, para conectar distintas ciudades (toda nueva vía que un jugador ponga tiene que estar conectada a la red de de vías de dicho jugador).

Además, el juego también permite vender vías obsoletas. Suponiendo que la red de vías de un jugador está expresada como un grafo (las vías serían las aristas del grafo), utilice el algoritmo de Kruskall para determinar qué vías (aristas del grafo) se podrían vender de tal manera que todas las ciudades que ya estuvieran conectadas por vías siguieran estando conectadas.

7: En un juego similar al del ejercicio anterior, un jugador puede establecer una ruta entre dos ciudades, esto es, construir un conjunto de vías. El coste de la ruta depende de las parcelas que cruce la vía, así, al existir montañas, o ríos o pueblos, el coste de la vía de dicha ruta aumentará.

El mapa del terreno se organiza en vértices y aristas. Los vértice representa una parcela y las aristas representan las posibles direcciones que puede tomar el trazado de la vía, en otras palabras, a qué otra parcela se puede ir para seguir construyendo. Además, una arista está etiquetada con el valor de construir un tramo de vía que vaya de una parcela a otra.

Utilice el algoritmo de Dysktra para encontrar el trazado más barato entre dos vértices dados.

8: Resuelva el problema de las Torres de Hanoi mediante grafos virtuales. El juego consiste en tres varillas verticales. En una de las varillas se apila un número indeterminado de discos (elaborados de madera) que determinará la complejidad de la solución, por regla general se consideran ocho discos. Los discos se apilan sobre una varilla en tamaño decreciente. No hay dos discos iguales, y todos ellos están apilados de mayor a menor radio en una de las varillas, quedando las otras dos varillas vacantes. El juego consiste en pasar todos los discos de la varilla ocupada (es decir la que posee la torre) a una de las otras varillas vacantes. Para realizar este objetivo, es necesario seguir tres simples reglas:

a) Sólo se puede mover un disco cada vez.

b) Un disco de mayor tamaño no puede descansar sobre uno más pequeño que él mismo.

c) Sólo puedes desplazar el disco que se encuentre arriba en cada varilla.

9: Una red de aeropuertos desea mejorar su servicio al cliente mediante terminales de información. Dado un aeropuerto origen y un aeropuerto destino, el terminal desea ofrecer la información sobre los vuelos que hacen la conexión y que minimizan el tiempo del trayecto total. Por un lado, se tiene la información de los vuelos, o lo que es lo mismo, lo que tarda un avión en ir de un aeropuerto origen a uno destino. Además, también se puede calcular cuánto tarda la escala en un aeropuerto concreto ya que se tiene la hora del vuelo de llegada y la hora del vuelo de salida.

Diseñar el problema mediante grafos virtuales y resolverlo mediante algoritmos de recorridos sobre grafos vistos en teoría, ¿cuál cree que es el más adecuado?

NOTAS:

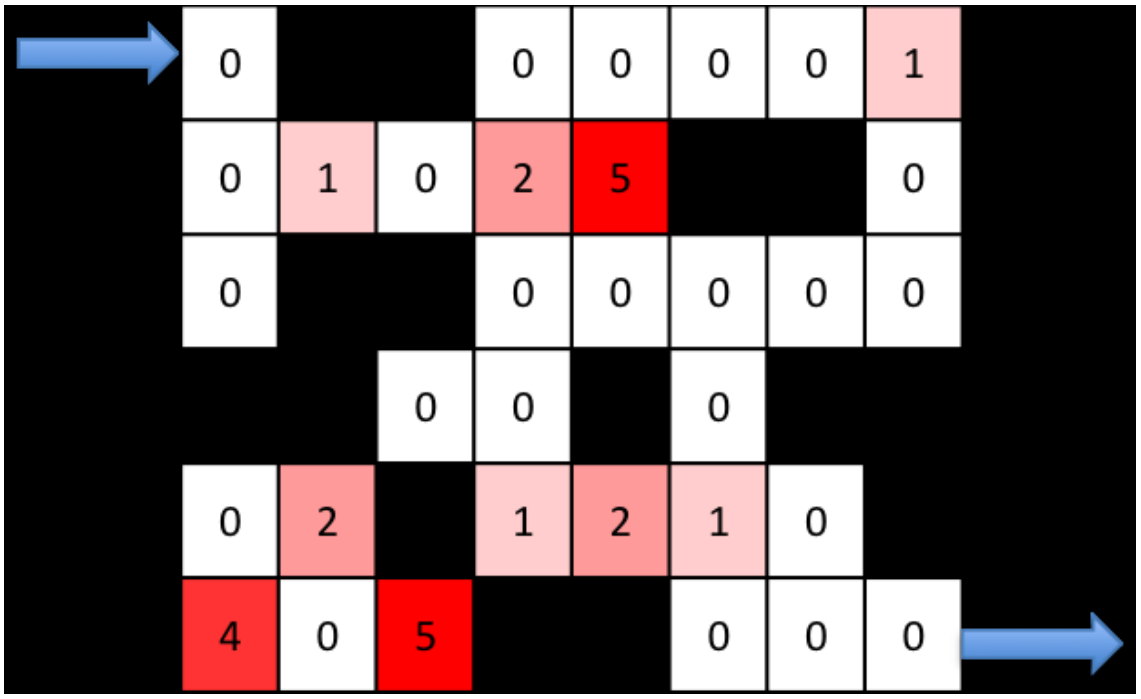
- No hay diferencia horaria entre los aeropuertos.
- No se deben tener en cuenta situaciones imprevisibles como retardos en vuelos.
- Todos los días el horario de vuelos en cada aeropuerto es el mismo.

10: Tenemos un laberinto formado por casillas con las siguientes características:

• Existen casillas infranqueables (las modelaremos con un entero de valor -1).

• Las demás casillas pueden atravesarse pero la dificultad para hacerlo no es siempre la misma. Para modelarlo usaremos un entero que variará entre 0 y 10. Las más sencillas de atravesar tienen coste 0 y las más costosas tendrán valor 10.

Dado un fichero con la información de cada casilla se creará el laberinto que lo modela y se resolverá utilizando un grafo virtual. Partimos siempre de la casilla superior izquierda y finalizamos por la casilla inferior derecha. Los movimientos permitidos son en vertical y horizontal, nunca en diagonal.



Un ejemplo de dicho fichero es el siguiente:

0, -1, -1, 0, 0, 0, 0, 1
0, 1, 0, 2, 5, -1, -1, 0
0, -1, -1, 0, 0, 0, 0, 0
-1, -1, 0, 0, -1, 0, -1, -1
0, 2, -1, 1, 2, 1, 0, -1
4, 0, 5, -1, -1, 0, 0, 0.

Se desea *solucionar* el problema utilizando la ruta con menor dificultad (puede pasar por más casillas pero la suma de los pesos de las casillas atravesadas será mínima).

11: Un plan de estudios está estructurado para que los alumnos no puedan matricularse libremente de las asignaturas que deseen. Existen asignaturas que deben haberse cursado (y aprobado) anteriormente para que un alumno se pueda matricular de una asignatura dada. Así, los prerrequisitos de una asignatura pueden ser 0 o más asignaturas. Por ejemplo, si los prerrequisitos de la asignatura A3 son las asignaturas A2 y A1, todo alumno debe haber aprobado estas dos asignaturas antes de matricularse de A3. En el siguiente ejemplo se muestra el grafo para el caso de un plan de estudios de cinco asignaturas:

Aquí, las asignaturas A1, A2 y A4 no tienen prerrequisitos. Los prerrequisitos de A3 son A1 y A2 y los de A5 son A3 y A4. En este caso, si un alumno tiene aprobada la asignatura A1, podrá cursar las asignaturas A2 y A4, y si tiene aprobadas A1 y A2, podrá cursar las asignaturas A3 y A4.

Implemente un método que, dado el grafo que representa el plan de estudios y la lista de asignaturas que un alumno tiene aprobadas, devuelva una lista con las asignaturas que puede cursar el próximo año.

List<Asignatura> asignaturasPosibles(
Graph<Asignatura, DefaultEdge> planEstudios, List<Asignatura> aprobadas)

Todas las asignaturas de la lista de entrada aprobadas son correctas (están en el grafo y no se han producido incoherencias como que aparezcan como aprobadas sólo A1 y A5).

12 El problema de cruzar el rio

Tenemos 3 misioneros y 3 caníbales y un bote para cruzar el rio. El bote tiene capacidad para 2 personas a lo sumo. Se trata que los 6 individuos crucen el rio de forma que en ningún momento haya más caníbales que misioneros en cualquiera de los dos márgenes del rio. ¿Cuál es la sucesión de viajes del bote para transportar las seis personas de una orilla a otra?. Representar el problema como un grafo virtual.

13 El problema del caballo en el juego de ajedrez

Consideremos un tablero de ajedrez y un caballo. Se pregunta si es posible que el caballo parta de una casilla y visite todos los otros 63 casilleros una solo vez volviendo al punto inicial.

14 **Problema de horarios**.

Se deben impartir varias asignaturas y no se pueden programar a la misma hora dos asignaturas que tienen alumnos matriculados en ambas. En este caso los vértices representan asignaturas y una arista entre dos vértices significa que hay alumnos matriculados en las asignaturas que representan los vértices.

¿Se puede hacer un horario de n horas sin incompatibilidades?

15 **Problema de las Estaciones de Bomberos**

El Gabinete Técnico de Protección contra Incendios del ayuntamiento de una ciudad decide revisar la localización de las estaciones de bomberos que controla. La ciudad está dividida en barrios. En cada barrio sólo puede existir una estación de bomberos. Cada estación es capaz de atender los incendios que se produzcan en la zona comprendida por su barrio y los barrios colindantes.