Introducción a la Programación

Tema 15. Algoritmos de Vuelta Atrás y Voraces

1. Introducción: Algoritmos de Vuelta Atrás y Programación Dinámica

Los algoritmos de **Vuelta Atrás**, que vamos a ver en este capítulo, pueden verse implementaciones eficientes de casos particulares de **Programación Dinámica**. Cada variante de la Programación Dinámica da lugar a una variante de los algoritmos de vuelta atrás como veremos en adelante.

En el capítulo anterior hemos visto la **Programación Dinámica** como una técnica que generaliza a la conocida como **Divide y Vencerás** (con o sin memoria). Vimos diversas variantes de la programación dinámica: con y sin memoria, aleatoria, con filtro, búsqueda de todas las soluciones, búsqueda sólo de una.

En la *Programación Dinámica* a cada problema, dentro del contexto del conjunto de problemas considerado podía identificarse por un conjunto de valores para las propiedades individuales consideradas. Asociado a cada problema X aparecen un conjunto de alternativas, A_X que depende del problema. Tal como vimos el esquema de la *Programación Dinámica* es de la forma:

$$f(X) = \begin{cases} s_b, & si\ es\ un\ caso\ base \\ cA_{a \in A_X}(c(A, a, f(X_1^a), f(X_2^a), \dots, f(X_k^a))), & si\ es\ un\ caso\ recursivo \end{cases}$$

Todos los problemas $X, X_1^a, \dots, X_k^a$ son problemas del conjunto de problemas considerado. Cada subproblema debe ser de un tamaño menor que el problema original. Esta relación de tamaño nos permite definir una relación de orden total entre los problemas del conjunto. En el sentido de que $Y \prec_n X$ si el tamaño de Y es menor que el tamaño de X. Con esta definición previa se debe cumplir que $X_i^a \prec_n X$ para cada uno de los subproblemas.

El conjunto de problemas más las relaciones entre problemas y subproblemas definen implícitamente, como vimos, un grafo dirigido. Cada problema se asocia a un vértice y hay un camino entre los problemas X y X_i^a si hay una alternativa $a \in A_X$ y un entero i que represente un posible subproblema. Cada camino entre los vértices X, X_i^a la representamos de la forma $X \succ_{a,i} X_i^a$.

En el caso particular de **Reducción**, es decir cuando el número de subproblemas para una alternativa dada es igual a uno ($k = 1$) el esquema para este caso particular es:

$$f(X) = \begin{cases} s_b, & si\ es\ un\ caso\ base \\ cA_{a \in A_X}(c(X, a, f(X^a))), & si\ es\ un\ caso\ recursivo \end{cases}$$

En este caso el problema X se reduce al problema X^a (dónde podemos eliminar el subíndice al solo haber un subproblema) después de tomar la alternativa $a \in A_X$. De nuevo, como en el caso general, el conjunto de problemas más las relaciones $\succ_a$ (ahora está omitido el número del subproblema porque siempre hay 1) definen un grafo dirigido implícito. Cada vértice es un problema del conjunto de problemas y existe una arista de arista de X_1 a X_2 si $X_1 \succ_a X_2$ para algún $a \in A_{X_1}$. Ahora en el grafo (a diferencia de en el caso general de la Programación Dinámica) todos los vértices están asociados a problemas y las aristas etiquetadas con alternativas. Las hojas de este grafo son casos base o problemas que no tienen solución. Como vimos, resolver un problema, desde esta perspectiva, es encontrar un camino, si existe, desde el vértice que representa el problema hasta una hoja que sea un caso base. Encontrar todas las soluciones al problema es encontrar todos los caminos desde el vértice que representa el problema que acaban en casos base. Conocidas todas las soluciones la mejor, con respecto a una propiedad dada de la solución, es aquella que tiene mayor (o menor) valor de esa propiedad entre todas las soluciones encontradas.

Los **caminos en el grafo** serán secuencias de alternativas que representaremos por $\{a_1, a_2, \dots, a_r\}$. Partiendo de un problema dado X_0 y siguiendo la secuencia de alternativas $\{a_1, a_2, \dots, a_r\}$ alcanzaremos un problema que representaremos por X_r. Desde la perspectiva anterior una solución para el problema X_0 puede representarse como una secuencia de alternativas $\{a_1, a_2, \dots, a_r\}$ con $a_1 \in A_{X_0}$ y X_r un caso base. Alternativamente una secuencia de alternativas (un camino en el grafo) puede acabar en un problema que no tenga solución.

En la *Programación Dinámica* los casos base se tratan explícitamente. Por lo tanto en esa técnica no es necesario definir el conjunto de alternativas para esos problemas. En algunos casos es más sencillo permitir también reducciones de los casos base con solución a un nuevo problema que llamaremos **problema final**. Siguiendo esta posibilidad completamos el conjunto de problemas con un problema ficticio adicional que denominaremos el *problema final* y representaremos por Θ. Este problema final tendrá un tratamiento especial. Con este problema final tenemos una forma alternativa de caracterizar las soluciones de un problema. Ahora las hojas del grafo son el problema Θ y problemas que no tienen solución. Las soluciones vienen definidas por caminos desde el problema original hasta el *problema final* Θ. La solución al problema original la podemos representar por $s = a_1 + a_2 + \cdots + a_r$. Donde a_r es la alternativa que hemos tomado en el caso base para alcanzar el problema fina. Las soluciones del problema están asociadas a secuencias de alternativas, caminos, que conducen desde el estado inicial al estado final. Lo caminos que conducen a un problema sin alternativas pero no final no son soluciones. Más generalmente podemos considerar que los casos base no son especiales y permitir, si resulta más sencillo, que todos los casos base, con y sin solución, puedan transitar al estado final. En este caso puede ocurrir que no haya solución asociada a un

camino desde el problema inicial al problema final. Pero si tenemos en cuenta los casos base, y sólo permitimos a los que tengan solución poder transitar al estado final, entonces cada camino desde el estado inicial al final define una solución.

2. Algoritmos de Vuelta Atrás

Los algoritmos de **Vuelta Atrás** (y sus variantes) son algoritmos especialmente diseñados para resolver de forma eficiente *Problemas de Programación Dinámica en el caso particular de Reducción y sin Memoria*. Son, por lo tanto, adecuados para resolver problemas que se pueden concebir en términos de grafos abstractos (grafos implícitos como en los ejemplos de arriba) cuyos vértices representan problemas y cuyas aristas representan relaciones de reducción de un problema a otro de tamaño más pequeño cuando escogemos una de las alternativas posibles. Los algoritmos de *Vuelta Atrás* son, en su versión inglesa, son los algoritmos de *Backtracking*.

Los algoritmos de *Vuelta Atrás* hacen una exploración en profundidad del grafo. Comienzan por un vértice, que representa el problema, y van explorando en profundidad el grafo anotando la secuencia de alternativas que se han tomado. Se llaman de *Vuelta Atrás* porque, siguiendo el esquema de exploración en profundidad, cuando llegan a un vértice hoja (con solución o sin ella) vuelven hacia atrás, al vértice padre, para seguir buscando la primera u otra solución. Cada solución la podemos caracterizar por la secuencia de alternativas desde el vértice que representa el problema hasta el problema final.

Los algoritmos de *Vuelta Atrás* tienen distintas variantes que reproducen las mismas ideas que vimos en la *Programación Dinámica*, y que estudiaremos con más detalle abajo:

- *Básico*: Es el esquema de partida. Es adecuado para buscar todas las soluciones, o solo una de ellas. En este último caso el algoritmo para cuando encuentra la primera solución. Esta variante es adecuada, también, para problemas que buscan una solución solamente o la existencia o no de solución.
- *Ramifica y Poda*: El algoritmo filtra (poda) el conjunto de alternativas disponible usando información de las decisiones tomadas y cotas del valor esperado. Es la versión correspondiente de las mismas ideas vistas en la *Programación Dinámica con Filtro*. Es adecuada para resolver problemas de optimización.
- *Aleatoria*: Escoge, sin volver atrás, una de las alternativas (un número fijado de veces) de forma aleatoria y después continúa normalmente como un algoritmo de *Vuelta Atrás*.

Relacionados con los algoritmos de vuelta atrás están unos algoritmos iterativos denominados los *Algoritmos Voraces* que estudiaremos con más detalle en el capítulo siguiente y haremos la introducción en este. Si en un algoritmo de *Vuelta Atrás* escogemos con una política fija una y sólo una de las alternativas posibles y eliminamos la posibilidad de volver atrás tenemos un *Algoritmo Voraz*. Esto algoritmos recorren, sin vuelta atrás, un camino que va desde el

problema original hasta una de las hojas del grafo de problemas. Se pueden especificar escogiendo en cada estado, mediante una función, una de las alternativas disponibles. Al seguir de forma un solo camino los Algoritmos Voraces son muy rápidos. Pero hay que tener en cuenta algunas diferencias importantes con los de Vuelta Atrás:

- Los algoritmos *Voraces* son siempre mucho más rápidos que los de *Vuelta Atrás*
- Los algoritmos *Voraces* puede que no encuentren la solución aunque exista. Los de *Vuelta Atrás* siempre la encuentran si existe.
- Si estamos resolviendo un problema de optimización los algoritmos *Voraces* puede que encuentren una solución pero que no sea la óptima. Los de *Vuelta Atrás* siempre la encuentran la óptima si existe.
- Si podemos demostrar que un algoritmo *Voraz* encuentra la solución (o la solución óptima si es lo que buscamos) entonces es preferible al algoritmo *Voraz* al de *Vuelta Atrás* correspondiente. Pero es demostración hay que hacerla específicamente para algoritmos *Voraces* concretos.

2.1 Concepto de Estado

En la implementación de los algoritmos de *Vuelta Atrás* se dispone de un objeto global que vaya guardando el problema actual, el conjunto de alternativas escogidas y posiblemente otra información derivada.

Los algoritmos de *Vuelta Atrás*, parten de un vértice, el problema a resolver, van eligiendo alternativas y alcanzado nuevos vértices. Llamaremos vértice inicial al que tiene asociado el problema a resolver y vértice actual el alcanzado por el algoritmo en un momento dado. El **estado**, por tanto, guarda el vértice actual y la secuencia de alternativas que tiene como primer elemento la tomada en es el vértice inicial y como último elemento la escogida para llegar al vértice actual. La secuencia de alternativas puede ser vacía si nos encontramos en el vértice inicial. Tal como hemos visto arriba una secuencia de alternativas define una **solución** cuando alcanzamos el problema final.

Un *estado*, por lo tanto, debe ser diseñado para contener la información sobre el problema actual y la secuencia de alternativas elegidas para llegar hasta él desde el problema inicial. Necesita, además los métodos necesarios para añadir (y eliminar en su caso) alternativas. El estado inicial debe tener la información sobre el problema inicial y una secuencia vacía de alternativas escogidas.

Con estas ideas podemos diseñar un tipo genérico, *EstadoBT*, que depende de los parámetros *A,* tipo de las alternativas, y *S,* tipo de las soluciones del problema a resolver, que nos sirva para construir objetos que contengan un estado. Igualmente diseñamos el tipo *ProblemaBT* que contendrá las propiedades del problema a resolver y a partir de ellas crear el estado inicial.

```
public interface ProblemaBT<S, A, E> {
	EstadoBT<S,A> getEstadoInicial();
```

```
}

public interface EstadoBT<S, A> {
	void avanza(A a);
	void retrocede(A a);
	int size();
	boolean isFinal();
	Iterable<A> getAlternativas();
	S getSolucion();
}
```

Hagamos algunas precisiones sobre los métodos:

- El método *avanza(A a)* añade una alternativa al estado y lo actualiza para que pueda informar sobre el problema actual.
- El método *retrocede(A a)* elimina la última alternativa del estado y lo actualiza. La alternativa a eliminar toma el valor *a*.
- Las propiedades *Inicial* y *Final* informan si el problema actual es inicial o si es final respectivamente.
- El método *getAlternativas* nos devuelve, como en el caso de la Programación Dinámica las alternativas posibles en ese estado.
- El método *getSolucion* nos devuelve la solución asociada a ese estado. Este método debe llamarse sólo si el estado es final.
- Teniendo en cuenta que un estado representa un problema del conjunto de problemas el método *size* nos da su tamaño.

Podemos usar la técnica equivalente a la *Programación Dinámica con Filtro* que llamaremos *Ramifica y Poda*. En esta técnica necesitamos que el estado proporcione más información. Para ello diseñamos el tipo *EstadoBTF* que depende de un parámetro más, el parámetro E, que representa el tipo de la propiedad a optimizar.

```
public interface EstadoBTF<S, A, T extends Comparable<? super T>>
		extends EstadoBT<S, A> {
	T getObjetivo();
	T getObjetivoEstimado(A a);
}
```

Los métodos tienen el mismo objetivo que en la *Programación Dinámica con Filtro.*

2.2 Diseño, corrección y complejidad

El diseño de los algoritmos de *Vuelta Atrás* puede partir, en general, de un diseño similar al que hemos usado en *Programación Dinámica*. Si el problema resultante es de *Reducción* y *sin uso de Memoria* entonces tenemos un candidato adecuado para ser resuelto por estas técnicas. Alternativamente podemos partir de un grafo implícito definido por un conjunto de vértices que representan problemas (estados o situaciones) conectados por aristas dirigidas

que son las alternativas que tenemos para reducir un problema a otro o pasar de un estado a otro.

La siguiente tarea es diseñar el *estado* concreto para resolver el problema. Esto implica, fundamentalmente definir las propiedades adecuadas del estado que permitan que éste cumpla su cometido: pueda indicarnos el problema actual en cada momento, mantener la secuencia de alternativas escogidas para llegar al problema actual. Igualmente hay que implementar las transiciones adecuadamente. Es decir los métodos *avanza* y *retrocede*.

Para ello podemos seguir las pautas que usamos en la *Programación Dinámica* para pasar de un problema a un subproblema. En efecto el método *avanza(A a)* transforma un estado que representa un problema en otro que representa el subproblema al que se reduce cuando se toma la alternativa *a*. El método *retrocede(A a)* hace justamente lo contrario.

La corrección de este tipo de algoritmos puede seguir los mismos pasos que en el caso de la Programación Dinámica: conjunto de problemas, subproblemas y transiciones entre ellos, conjunto de alternativas, etc.

Los algoritmos de vuelta atrás suelen usarse si memoria. Esto es adecuado cuando no se comparten los subproblemas (es decir los caminos desde el problema original hasta los casos base no se cortan). En caso que se compartan los subproblemas se podrían diseñar algoritmos de vuelta atrás con memoria. La complejidad es equivalente a la del algoritmo de Programación Dinámica equivalente (con o sin memoria según el caso).

3. Detalles de implementación

3.1 Algoritmo Genérico de Vuelta Atrás

Con los tipos anteriores podemos escribir de los algoritmos genéricos de cada uno de los tipos anteriores

El Algoritmo genérico de *Backtracking* (Algoritmo de Vuelta Atrás) es de la forma:

```
public class AlgoritmoBT<S, A, T extends Comparable<? super T>>
       extends AbstractAlgoritmo {

       public static enum Tipo{Max,Min,Otro};

       public static Tipo tipo;
       public S solucion = null;
       public Set<S> soluciones =  Sets.newHashSet();
       public static int numeroDeSoluciones = 1;
       public static boolean isRandomize = false;
       public static Integer sizeRef = 10;
       public static  boolean conFiltro = false;
       public static boolean soloLaPrimeraSolucion = true;

       private ProblemaBT<S,A> problema;
```

```
   private EstadoBT<S,A> estado;
   private EstadoBTF<S,A,T> estadoF;
   private boolean exito = false;
   private T mejorValor;

   public AlgoritmoBT(ProblemaBT<S,A> p){
   problema = p;
   mejorValor = null;
}

@SuppressWarnings("unchecked")
   public void ejecuta() {
   metricas.setTiempoDeEjecucionInicial();
          do {
                estado = problema.getEstadoInicial();
                if(conFiltro) estadoF = (EstadoBTF<S,A,T>) estado;
                bt();
                } while(isRandomize &&
                      soluciones.size()<numeroDeSoluciones);
          metricas.setTiempoDeEjecucionFinal();
   }

private Iterable<A> filtraRandomize(EstadoBT<S,A> p,
   Iterable<A> alternativas){
          Iterable<A> alt;
          if(isRandomize && p.size()>sizeRef &&
                !Iterables.isEmpty(alternativas)){
                alt = Iterables2.unitaryRandomIterable(alternativas);
          }else{
                alt = alternativas;
          }
          return alt;
   }

private void actualizaSoluciones(){
   T objetivo = estadoF.getObjetivo();
   if(conFiltro){
          if(mejorValor== null ||
                AlgoritmoBT.tipo==Tipo.Max &&
                      objetivo.compareTo(mejorValor) > 0 ||
                AlgoritmoBT.tipo==Tipo.Min &&
                      objetivo.compareTo(mejorValor) < 0) {
                solucion = estado.getSolucion();
                soluciones.add(solucion);
                mejorValor = objetivo;
          }
   }else{
          solucion = estado.getSolucion();
                soluciones.add(solucion);
   }
}

private void bt() {
          metricas.addLLamadaRecursiva();
   if(estado.isFinal()){
          actualizaSoluciones();
          if(soloLaPrimeraSolucion  && solucion!=null) exito = true;
          if(!soloLaPrimeraSolucion  &&
                soluciones.size()>=numeroDeSoluciones) exito = true;
```

```
        } else {
                        for(A a: filtraRandomize(estado,estado.getAlternativas())){
                                        if(conFiltro && !pasaFiltro(a)) { continue;}
                                        estado.avanza(a);
                                        bt();
                                        estado.retrocede(a);
                                        if (exito) break;
                        }
        }
    }

     private boolean pasaFiltro(A a){
        boolean r = true;
        if(conFiltro){
               r= mejorValor==null ||
               AlgoritmoBT.tipo==Tipo.Max &&
                      estadoF.getObjetivoEstimado(a).compareTo(mejorValor) > 0
               || AlgoritmoBT.tipo==Tipo.Min &&
                      estadoF.getObjetivoEstimado(a).compareTo(mejorValor) < 0;
               }
               return r;
     }

        public Set<S> getSolucionesFiltradas(Predicate<S> p) {
               return Sets.newHashSet(Iterables.filter(soluciones,p));
        }

}
```

Como podemos comprobar hay un conjunto de variables públicas adecuadas para configurar el algoritmo

- *solucion:* La solución o la mejor solución, en su caso, encontrada
- *soluciones:* Conjunto de soluciones encotradas (todas o las más cercanas a al mejor para poder obtener las mejores)
- *numeroDeSoluciones:* Número de soluciones que vamos buscando, ya en el método aleatorio o cuando buscamos más de una.
- *isRandomize:* Verdadero si se usa el método aleatorio
- *sizeRef:* Tamaño de referencia para el método aleatorio
- *conFiltro:* Verdadero si se usa el método con Filtro.
- *soloLaPrimeraSolucion:* Verdadero si solo se quiere la primera solución encontrada o simplemente la existencia de solución. Falso si se quieren todas, la mejor o las mejores.

El método *filtraRandomize* es el mismo que en el caso de la *Programación Dinámica Aleatoria*.

El algoritmo, como podemos ver, dado un estado genera las alternativas disponibles, las filtra si estamos en el modo con filtro, las añade al estado y se llama recursivamente en el nuevo estado. Cuando vuelve de la llamada recursiva coloca el estado en la situación previa e intenta la siguiente alternativa.

Cuando alcanza un estado final actualiza la solución, el conjunto de soluciones y posteriormente decide indicar que hay que parar (colocando éxito a true).

Si del conjunto de alternativas escoge mediante una función sólo una de ellas y cuando alcanzamos el problema final o uno sin alternativas paramos el algoritmo tendríamos la implementación de un *Algoritmo Voraz*. Pero aunque es posible hacerlo así (veremos algunos ejemplos al final del capítulo) en general los Algoritmos voraces se presentan en su versión iterativa tal como veremos en el capítulo siguiente.

4. Algoritmos voraces

Los primeros, los algoritmos *Voraces*, ya los hemos visto por encima en el capítulo anterior. Guardan relación con los *Algoritmos de Vuelta Atrás* y con los de *Programación Dinámica*. Partiendo de un *Algoritmos de Vuelta Atrás* podemos diseñar otro *Voraz* indicando para cada conjunto de alternativas A_X cual de ellas elegir. En definitiva definiendo una función $h(X)$ que a partir de un problema X elige una de las alternativas del conjunto A_X. La función $h(X)$ tiene como dominio el conjunto de problemas donde A_X no es vacío. Estos algoritmos, los *Voraces*, parten de un vértice X, el problema a resolver, eligen la alternativa $h(X)$ y al tomarla reducen el problema a otro X^a. Desde el nuevo vértice comienzan de nuevo eligiendo la alternativa $h(X^a)$ hasta encontrar un problema cuyo conjunto de alternativas sea vacío. En ese momento comprueban, partir de la secuencia de alternativas escogidas, si han encontrado una solución.

Un *Algoritmo Voraz* parte de un conjunto de problemas (representados mediante un conjunto de estados) y mediante una función (que llamaremos $next$ en adelante) va decidiendo sin vuelta atrás el siguiente problema. El *Algoritmo Voraz* parte de un problema inicial (un estado inicial) y va escogiendo de forma irrevocable el siguiente hasta que llega a uno que cumple un criterio especificado.

La función $next(X, a)$ parte de un problema X y obtiene otro Y dada una alternativa que se ha escogido previamente mendiante $h(X)$.

El esquema de los algoritmos voraces es por tanto:

```
E e = p.getEstadoInicial();
A a;
while (!e.condicionDeParada()) {
       a = e.getAlternativa()
       e = e.next (a);
       }
       return e;
}
```

Donde *E* es el tipo de estado, *A* el tipo de las alternativas y p un problema a resolver.

En general los *Algoritmos Voraces* los podemos pensar sin partir de los de *Vuelta Atrás*. En ese caso partimos de un problema y sus soluciones posibles (posiblemente cada una de ellas con un valor calculado por las función objetivo). Representamos cada solución posible mediante una instancia del estado. Igual que antes para cada instancia del estado *e* se definen un conjunto de alternativas A_e. Mediante la función $h(e)$ (dónde *e* es una instancia del estado) se escoge una de las alternativas de A_e. Dado un estado una estrategia *Voraz* se define mediante la función $next(e, a)$ que, para cada estado y alternativa, define el siguiente.

Por al propiedades que hemos explicado vemos que los *Algoritmos Voraces* pueden implementarse como algoritmos iterativos.

Con los ejemplos anteriores ya podemos concluir algunas diferencias importantes con los Algoritmos de *Vuelta Atrás* (e implícitamente con los de *Programación Dinámica de Reducción*):

- Los algoritmos *Voraces* son siempre mucho más rápidos que los de *Vuelta Atrás*
- Los algoritmos *Voraces* puede que no encuentren la solución aunque exista. Los de *Vuelta Atrás* siempre la encuentran si existe.
- Si estamos resolviendo un problema de optimización los algoritmos *Voraces* puede que encuentren una solución pero que no sea la óptima. Los de *Vuelta Atrás* siempre la encuentran la óptima si existe.
- Si podemos demostrar que un algoritmo *Voraz* encuentra la solución (o la solución óptima si es lo que buscamos) entonces es preferible al algoritmo de *Vuelta Atrás* correspondiente. Pero es demostración hay que hacerla específicamente para algoritmos *Voraces* concretos.

5. Detalles de implementación del problema

Veamos los detalles de implementación del problema de las reinas ya visto en el capítulo anterior.

5.1 Problema de las Reinas

El problema de las Reinas ya lo hemos modelado en el tema anterior. Como vimos allí el problema se enuncia así:

Colocar Nr reinas en un tablero de ajedrez $Nr \times Nr$ de manera tal que ninguna de ellas amenace a ninguna de las demás. Una reina amenaza a los cuadrados de la misma fila, de la misma columna y de las mismas diagonales. Las filas y columnas toman valores en $(0, Nr - 1)$

Cada reina la representamos por $(f, c, dp, ds), c \in [0, Nr - 1]$. Siendo c_i, f_i, dp_i, ds_i la columna, la fila, la diagonal principal y la diagonal secundaria de la reina *i*. La columna y la fila son números enteros en *[0,Nr).* La diagonal principal y la diagonal secundaria son

representadas por números enteros que las identifican de manera única. Siendo estos enteros calculados como $dp_i = f_i - c_i$, $ds_i = f_i + c_i$. El tipo *Reina* tiene las propiedades individuales anteriores y una propiedad compartida: *Nr* (entero no negativo, consultable) que representa el número de reinas.

La solución buscada es de tipo *List<Reina>.*

Las propiedades compartidas del conjunto de problemas son: *Nr*, consultable.

Generalizamos el problema planteando colocar *Nr-C* reinas en las columnas *[C,Nr-1]* suponiendo ya ocupadas un conjunto de filas, un conjunto de diagonales principales y un conjunto de diagonales secundarias. El problema generalizado tiene las propiedades individuales: *C (*entero, consultable*), Fo (*conjunto de enteros, consultable), *Dp* (conjunto de enteros, consultable) y *Ds* (conjunto de enteros, consultable). En este contexto dos problemas son iguales si tienen iguales sus propiedades individuales: *C, Fo, Dp, Ds.*

Las alternativas disponibles podemos representarlas con números enteros que representarán la fila donde se ubicará la siguiente reina. La siguiente reina se colocará en una de las filas disponibles de la de la columna *C*. Las alternativas disponibles para la fila donde colocar la reina son:

$$A_{c,fo,dp,ds} = \{f \in [0, Nr) \mid f \notin fo, f - c \notin dp, f + c \notin ds\}$$

El objetivo del problema es encontrar una solución que tenga un número de reinas igual a *Nr* y ninguna reina amenace a otra. Esto es equivalente a decir que los cardinales de los conjuntos *fo*, *dp*, *ds* (propiedades de la solución) sean iguales a *Nr*.

Todas las ideas anteriores están resumidas en la Ficha siguiente:

Problema de las Reinas	
Técnica: Backtraking	
Tamaño: Nr-C	
Propiedades Compartidas	*Nr, entero*
Propiedades del estado	*C, entero no negativo* *Fo, conjunto de enteros* *Dp, conjunto de enteros, derivada* *Ds, conjunto de enteros, derivada*
Solución: List<Reina>	
Objetivo: Encontrar $s^{r,fo,dp,s}$ *tal que* $\|r\| = \|fo\| = \|dp\| = \|ds\| = Nr$	
Inicial: C = 0, Fo = {}, Dp={}, Ds={}	
Final: N = Nr	
Alternativas: $A = \{f \in [0, Nr) \mid f \notin fo,\ f - c \notin dp,\ f + c \notin ds\}$	
$avanza(f) = [C + 1, Fo + f]$	
$retrocede(f) = [C - 1, Fo - f]$	

El método *avanza(..)* se obtiene del correspondiente subproblema visto en el capítulo anterior. El método *retrocede(..)* es el inverso.

La implementación del estado es:

```
public class EstadoReinas implements EstadoBT<List<Reina>, Integer> {

	private int columna;
	private Set<Integer> filasOcupadas;
	private Set<Integer> diagonalesPrincipalesOcupadas;
	private Set<Integer> diagonalesSecundariasOcupadas;
	private List<Integer> camino;

	private EstadoReinas(int columna) {
		super();
		this.columna = columna;
		this.filasOcupadas = Sets.newHashSet();
		this.diagonalesPrincipalesOcupadas = Sets.newHashSet();
		this.diagonalesSecundariasOcupadas = Sets.newHashSet();
		this.camino = Lists.newArrayList();
	}

	public static EstadoReinas create(int columna) {
		return new EstadoReinas(columna);
	}

	...

	@Override
	public boolean isFinal() {
		return getColumna() == ProblemaReinas.numeroDeReinas;
	}

	@Override
	public boolean isInicial() {
		return getColumna()== 0;
	}

	@Override
	public List<Reina> getSolucion() {
		List<Reina> lis = Lists.newArrayList();
		int c = 0;
		for(Integer f: camino){
			Reina r = Reina.create(f, c);
			lis.add(r);
			c++;
		}
		return lis;
	}

	@Override
	public Iterable<Integer> getAlternativas() {
		Iterable<Integer> pf = Iterables2.fromArithmeticSequence(
		  0,ProblemaReinas.numeroDeReinas,1);
		Iterable<Integer> pf2 = Iterables.filter(pf,
	    new Predicate<Integer>(){
			@Override
			public boolean apply(Integer f) {
				Reina r = Reina.get(f, getColumna());
				return !getFilasOcupadas().contains(f) &&
				    !getDiagonalesPrincipalesOcupadas().
                contains(r.getDiagonalPrincipal()) &&
```

```
                                !getDiagonalesSecundariasOcupadas().
                            contains(r.getDiagonalSecundaria());
                        }});
                return pf2;
        }

        @Override
        public void avanza(Integer f) {
                Reina r = Reina.get(f, columna);
                filasOcupadas.add(r.getFila());
                diagonalesPrincipalesOcupadas.add(r.getDiagonalPrincipal());
                diagonalesSecundariasOcupadas.add(r.getDiagonalSecundaria());
                camino.add(f);
                columna = columna+1;
        }

        @Override
        public void retrocede(Integer f) {
                columna = columna-1;
                Reina r = Reina.get(f, columna);
                filasOcupadas.remove(r.getFila());
                diagonalesPrincipalesOcupadas.remove(r.getDiagonalPrincipal());
                diagonalesSecundariasOcupadas.remove(r.getDiagonalSecundaria());
                camino.remove(f);
        }
...
}
```

En el código anterior se han usado métodos de la clase *Iterables2* para generar secuencias de enteros dentro de un rango. También se han usado la clase *Iterables* y los tipos *Predicate* de *Guava*.

El test lo podemos hacer con la clase siguiente:

```
public class TestReinas {

   public static void main(String[] args) {
       AlgoritmoBT.numeroDeSoluciones = 14;
       AlgoritmoBT.isRandomize = true;
       AlgoritmoBT.soloLaPrimeraSolucion = false;
       AlgoritmoBT.sizeRef = 10;
       ProblemaReinas.numeroDeReinas = 34;
       ProblemaBT<List<Reina>,Integer> p = ProblemaReinas.create();
       AlgoritmoBT<List<Reina>,Integer,?> a = Algoritmos.createBT(p);
       a.ejecuta();
       System.out.println(a.soluciones.size());
       System.out.println(Metricas.getMetricas().getTiempoDeEjecucion());
       for(List<Reina> lis : a.soluciones){
              System.out.println(lis);
       }
   }
}
```

Según los valores que demos a las variables de configuración de *AlgoritmoBT* podemos obtener la primera solución por un método aleatorio o no, las primeras soluciones, etc.

6. Algunos problemas resueltos

Como en el caso de la *Programación Dinámica* podemos resumir en una ficha los principales aspectos de como un problema va a ser resuelto mediante un algoritmo. Para ello veamos las fichas para resolver algunos de los problemas vistos en el capítulo anterior mediante alguno de los algoritmos que acabamos de ver.

Para cada uno de ellos ofrecemos una ficha que resume las principales características del mismo pero referimos a los tipos y detalles vistos en el capítulo anterior. Podemos ver en cada caso las similitudes con la solución mediante *Programación Dinámica*. También ofrecemos la ficha para problemas que no hemos visto previamente con detalle.

6.1 Problema de la Mochila

El problema de la Mochila, como ya vimos, se enuncia así:

Tenemos uno conjunto de objetos con un determinado peso y valor que nos gustaría guardar en una mochila. La mochila tiene capacidad para llevar un peso máximo y para cada tipo de objeto hay especificado un límite máximo de unidades dentro de la mochila. Se pide encontrar la configuración de objetos que sin sobrepasar el peso de la mochila ni el número de unidades especificadas para cada uno de ellos, maximiza la suma de los valores de los objetos almacenados.

Las alternativas disponibles podemos representarlas con números enteros. El tipo *SolucionMochila* es un subtipo de *MultiSet<ObjetosEnMochila>* con las propiedades adicionales *ValorTotal* (entero, consultable) de los objetos almacenados en la mochila y *PesoTotal* (entero, consultable) de los objetos en la mochila. Una solución la representamos por $s^{om,v,p}$. El objetivo del problema es encontrar una solución cuya propiedad *Peso* sea menor o igual a la *Capacidad* de la mochila y tenga el mayor valor posible para la propiedad *Valor*.

El *estado,* además de extender el tipo *EstadoBTF<SolucionMochila,Integer,Integer>,* tendrá las propiedades *C* (capacidad actual de la mochila), *J* (entero que indica que en el problema actual podemos usar los objetos ubicados en las posiciones que van de cero a J) y *V* (valor acumulado al escoger con las alternativas ya escogidas).

Para poder usar la técnica de *Ramifica y Poda* añadimos a los problemas una propiedad compartida *ValorMaximo* (entero, consultable) que representaremos como *VM*.

Como vemos ahora debemos especificar si un problema (o lo que es equivalente un estado dado) es inicial o final y los métodos *avanza(…)* y *retrocede(…)*. El conjunto de alternativas es el

mismo que en el caso de la *Programación Dinámica con Filtro.* Aquí no son relevantes los operadores *c* y *cA*.

Todas las ideas anteriores están resumidas en la Ficha siguiente:

Problema de la Mochila	
Técnica: Ramifica y Poda	
Tamaño: J	
Propiedades Compartidas	*OD, List <ObjetoMochila>, ordenada de menor a mayor por el valor unitario.* *CI, entero, Capacidad inicial de la mochila* *VM, entero, valor inicial = $-\infty$*
Propiedades del Estado	*C, entero no negativo* *J, entero en [0,ObjetosDisponibles.Size)* *V,entero no negativo*
Solución: SolucionMochila	
Inicial: J = |OD|-1, C = CI	
Final: J < 0	
Objetivo: Encontrar $s^{o,v,p}$ tal que $p \leq c$ y v tenga el mayor valor posible	
Alternativas: $A = \{a: k..0\},\ k = \min\left(\frac{c}{v_j}, m_j\right),\ j > 0$ $A = \{k\}, \quad k = \min\left(\frac{c}{v_j}, m_j\right), j = 0$	
Función de cota: $cta(c, j, a) = av_j + ct(c - ap_j, j - 1)$	
$add(a) = [c - ap_j, j - 1, v + av_j]$	
$remove(a) = [c + ap_{j+1}, j + 1, v - av_{j+1})\ ]$	

6.2 Problema del Cambio de Monedas

Dada una *Cantidad* y un sistema monetario se pide devolver dicha cantidad con el menor número de monedas posibles. Un sistema monetario está definido por un conjunto *M* de monedas diferentes. Cada moneda la representamos por $m_i, i \in [0, r-1]$ y sea v_i el valor de cada una de ellas. La solución buscada es un multiconjunto de las monedas disponibles.

Cada moneda tiene la propiedad *Valor* (entero, consultable). Como notación para representar la moneda que ocupa la posición i en M usaremos la notación m_i^v. Y v_i para indicar el valor unitario de la moneda ubicada en la posición *i*.

El tipo *SolucionMoneda* es un subtipo de *Multiset<Moneda>* con las propiedades adicionales *Valor* (entero, consultable) de las monedas incluidas en la solución, *NúmeroDeMonedas* (entero, consultable) incluidas en la solución.

Las propiedades compartidas del conjunto de problemas son: *Monedas, List<Moneda>* ordenada y sin repetición, consultable. Monedas disponibles en el sistema monetario. La lista se mantiene ordenada de menor a mayor valor.

Las propiedades del estado son: *C*, *Cantidad (*entero, consultable*), J (*entero en [0,|*Monedas*|), consultable).

Las alternativas disponibles podemos representarlas con números enteros. El número de unidades que podemos usar de la moneda m_j si la cantidad de dinero a cambiar es c. Las alternativas disponibles están en el conjunto $A = \{0,1,2, \dots, r\}$ con $r = \frac{c}{v_j}$.

El objetivo del problema es encontrar una solución cuya propiedad *Valor* sea igual a la *Cantidad* y tenga el menor valor posible para la propiedad *NumeroDeMonedas*.

Todas las ideas anteriores están resumidas en la Ficha siguiente:

Cambio de Monedas	
Técnica: Backtraking	
Tamaño: J	
Propiedades Compartidas	*M, List <Moneda>, ordenada* *CI, entero, cantidad inicial*
Propiedades del Estado	*C, entero no negativo* *J, entero en [0,|M|)*
Solución: SolucionMoneda	
Inicial: J = |M|-1, C = CI	
Final: J < 0	
$A_{c,j} = \{a: r..0\}, \quad r = \frac{c}{v_j}, \quad j > 0$ $A_{c,j} = \{r\}, \ r = \frac{c}{v_j}, \quad c = rv_j, \quad j = 0$ $A_{c,j} = \{\}, \ r = \frac{c}{v_j}, \quad c \neq rv_j, \quad j = 0$ $A_{c,j} = \{\}, \quad j < 0$	
$add(a) = [c - av_j, j - 1]$	
$remove(a) = [c + av_{j+1}, j + 1]$	
Complejidad	

En el cálculo del conjunto de alternativas aparecen expresiones como $r = \frac{c}{v_j}, c = rv_j, j = 0$. Con ella queremos expresar que para *j=0* si *c* es divisible por v_j entonces el conjunto de alternativas es *{r}* y si no es divisible el conjunto vacío. Esto lo hacemos para que sólo los casos base con solución tengan una transición al problema final que hemos decidido que es un problema imaginario con *j<0*.

6.3 Problema de Asignación

Enunciado del problema de asignación:

Dados *n* agentes y *n* tareas, con un coste *C(i,J)* si el agente *i* ejecuta la tarea *j,* buscar una asignación de tareas a personas (donde cada tarea es asignada a una persona y cada persona tiene una sola tarea) para el coste total sea mínimo.

La solución mediante ramifica y poda podemos resumirla en:

Problema de la Asignación	
Técnica: *Algoritmo RyP*	
Propiedades Compartidas	*C: Table< Integer, Integer, Integer>*, Coste si un agente ejecuta una tarea *N: Integer*, número de agentes y de tareas *MC: Integer*, coste de la mejor solución
Propiedades del Estado	*AS: BiMap<Integer,Integer>,* asignación agente/tarea *I : Integer*, siguiente agente *CA: Integer,* coste acumulado
Solución: *SolucionAsignación*	
Objetivo: encontrar la solución que tenga el menor coste	
Alternativas: $A(as,i,ca)=\{j:0,\dots N-1 \mid j \notin ta\}$ **Función de Cota** $cta(as,i,j)=c(i,j)+\sum_{k=i+1}^{N}\min\{k:0..N-1 \mid k \notin ta\ ;c(i,k)\}$ $ta = values(as)$	
Estado Inicial: ([], {}, 0, 0)	
Estado Final*: i>=N*	
`Avanza (j):` `(as,i,ca)->(as+[i,j],i + 1,ca+c(i,j))`	
`Retrocede(j):` `(as,i,ca)->(as-[i-1],i - 1,ca-c(i-1,j))`	

Si entre las alternativas posibles elegimos una tenemos un algoritmo Voraz

Problema de la Asignación	
Técnica: *Algoritmo Voraz*	
Propiedades Compartidas	*C:Table< Integer, Integer, Integer>*, Coste de una tarea ejecutada por un agente *N: Integer*, número de agentes y de tareas
Propiedades del Estado	*AS: BiMap<Integer,Integer>,* asignación agente/tarea *I : Integer*, siguiente agente
Solución: *SolucionAsignación*	
Objetivo: encontrar $\mathbf{s}^{\mathbf{l,c}}$ tal que **c** tenga el menor valor	
Alternativas: $A(as,i)=\{j:0,\dots,N-1 \mid j \notin ta\}$	
Elección: $\mathrm{h}(A(as,i))=\underset{j\in A}{\operatorname{argmin}}\,(\mathrm{C(i,\,j)})$	
Estado Inicial*: ([], {}, 0)*	
Estado Final*: i >=N*	
`Avanza(j):` `(as,i)->(as+[i,j],i + 1)`	

6.4 Tareas y Procesadores

Se necesita realizar N tareas independientes en una máquina multiprocesador, con M procesadores pudiendo trabajar en paralelo (supóngase N > M). Sea *d(i)* el tiempo de ejecución

de la i-ésima tarea en cualquier procesador. El problema consiste en determinar en qué procesador hay que ejecutar cada uno de los trabajos, de forma que el tiempo final de la ejecución de todos los trabajos (tiempo de ejecución del procesador más cargado) sea mínimo. Supóngase que no hay restricciones acerca de cuándo puede comenzar la ejecución de cada trabajo.

Las alternativa *a* designa el procesador a que se asigna la tarea *j*. El primer lugar una propiedad compartida que guarda el mejor tiempo obtenido en una solución anterior: *MD, Integer.* En segundo lugar el conjunto de alternativas tiene dos filtros adicionales. El primero (que denominaremos *F(a)*) elimina una alternativa *a2* si ya existe otra *a1* que cumpla de *tp(a1) = tp(a2).* El segundo elimina alternativas cuya cota para el valor de la propiedad sea mayor o igual al máximo valor obtenido previamente.

Tareas y Procesadores	
Técnica: Algoritmo Ramifica y Poda con Filtro	
Propiedades Compartidas	*TD, List<Tarea> ordenada por duración de mayor a menor* *M, Integer, número de procesadores disponibles.* *MD, Integer, Tiempo de la mejor solución*
Propiedades del Estado	*TA, Map<Integer,Integer>, tareas asignadas (tarea,procesador)* *TP, Map< Integer, Integer>, Derivada, tiempo por procesador* *J, Integer, tarea actual.* *VA, Integer, Valor Acumulado, Derivada, Igual al tiempo ocupado del procesado más ocupado.*
Solución: SolucionTareas	
Objetivo: encontrar la solución tal que tiempo de ejecución sea mínimo	
Alternativas: $A(ta, tp, j) = \{a \in 0..M-1 \mid \forall a_1, a_2 : tp(a_1) \neq tp(a_2)\}$ *Ordenadas por el procesador menos cargado.*	
Función de Cota: *cta(ta,tp,j,a) =* $tp(a) + d(j)$	
Estado Inicial: $(\emptyset, \emptyset, 0)$	
Estado final: $j = \lvert TD \rvert$	
Avanza(a): $(ta, j) \rightarrow (ta + (j, a),\ j + 1)$ *Retrocede(a):* $(ta, j) \rightarrow (ta - (j-1), j-1)$	

De nuevo si escogemos sólo una de las alternativas posibles tenemos un algoritmo Voraz

Tareas y Procesadores	
Técnica: Algoritmo Voraz	
Propiedades Compartidas	*TD, List<Tarea> ordenada por duración de mayor a menor* *M, Integer, número de procesadores disponibles.*
Propiedades del Estado	*TA, Map<Integer,Integer>, tareas asignadas* *TP, Map< Integer, Integer>, Derivada, tiempo por procesador* *J, Integer, tarea actual.*
Solución: SolucionTareas	
Objetivo: encontrar la solución tal que tiempo de ejecución sea mínimo	
Alternativas: $A(ta, j) = \{a \in 0..M-1\}$	
Elección: h(ta, tp, j) = a, tal que tp(a) es menor o a todas las restantes alternativas.	
Estado Inicial: $(\emptyset, \emptyset, 0)$	
Estado final: $j = \|TD\|$	
Avanza(a): $(ta, j) \rightarrow (ta + (j + a),\ j + 1)$	

El algoritmo Voraz no usa el método *retrocede* pero lo hemos incluido para usarlo posteriormente en versión ramifica y poda. El algoritmo de Ramifica y poda añade elementos al anterior.

7. Algunos Algoritmos Voraces Conocidos

Veamos ahora algunos algoritmos voraces conocidos.

7.1 Problema del Camino Mínimo

El problema de los Camino Mínimo se enuncia así:

Dado un grafo con peso y dos vértices del mismo encontrar el camino mínimo desde un vértice al otro.

Vamos a buscar ahora un algoritmo voraz para resolver el problema: el conocido como algoritmo de ***Dijsktra.***

Como paso previo diseñaremos un iterable, que partiendo de un vértice origen, va recorriendo los vértices del grafo escogiendo como siguiente vértice el más cercano al origen a través de caminos formados por vértices ya encontrados.

En el estado mantendremos los vértices previamente encontrados, sus caminos hacia el origen y sus distancias. Los caminos hacia el origen forman un árbol. Esta información la guardamos en un agregado de tuplas de la forma (V, E, D, C) dónde *V* es un vértice, *E* la arista que indica su camino más corto hacia el origen, *D (*de tipo *Double)* su distancia hasta el origen siguiendo la arista anterior y *C* (de tipo *Boolean*) si el camino representado ya el el mínimo posible.

Por simplicidad si t es una tupla de la forma $T = (V, E, D, C)$ representaremos respectivamente por *t.v*, *t.e, t.d y t.c* la primera, la segunda, la tercera componente y la cuarta componente. El agregado de tuplas anterior lo mantendremos organizado mediante un *Map<V,T>* y un montón de *Fibonacci<T>* ordenado según la componente *t.d*.

Por *m(v)* representamos la imagen de *v* en el *Map m*, por *M+(v,t)* añadir ese par al map y por $v \in m$ si ese vértice pertenece al dominio del *Map*.

Un mónton de Fibonacci *f* es una cola de prioridad reordenable y tiene las operaciones

- *min(f): Devuelve la tupla con peso mínimo.*
- *F-t:* Elimina el par del montón.
- *F+t:* Añade la tupla al mónton.
- *F/(t,t´):* Cambia las componentes de la tupla t para convertirla en la $t´$ manteniendo el vértice *v*, haciéndo decrecer la distancia a *d´*, cambiando la arista a *e´* y reordena el montón.

Igualmentre dada un arista *e* representaremos por $w(e)$ el peso de la arista y por $ov(e, a)$ el otro vértice de la arista *e* si uno de ellos es *a*.

Todas las ideas anteriores están resumidas en la Ficha siguiente:

Iterable Siguiente Vértice Más Cercano	
Técnica: Voraz	
Propiedades Compartidas	*g, Graph <V,E>* *vo, Vértice Origen*
Propiedades del Estado	*m, Map de V sobre* $T = (V, E, D, C)$ *f, Montón de Fibonacci formado por tuplas de tipo (V,E,D,C) ordenadas según la componente d.* *r = min(f) o null si f está vacío* *va = r.v, Vértice actual* *A, Conjunto de Aristas que salen de a*
Invariante	*(v,e,d,c) ∈ f ↔* $v \in m \ \ \& \ \ v.c = false$ *Para cada v si c = true entonces d es la mínima distancia al origen y e la arista que inicia el camino hacia él.* *Si c = false entonces d es la mínima distancia al origen encontrada hasta ahora y e la arista que inicia el camino hacia él.*
Inicial: m={vo,(vo,null,0.,false)}, F={(vo,null,0.,false)}	
Final: f={}	
Alternativa: *va*	
$next(va)$ = Actualiza las *tuplas t* en $m \ y \ f$ asociadas a todos los vértices vecinos al actual *va* con los	

caminos mínimos al origen desde los mismos. Los vértices vecinos a *va* son los que se alcanzan según las aristas en A. Pone a *true* la componente c de la *tupla* asociada a *va* en *m* y elimina de f la tupla asociada a *va*.

El algoritmo, como se ve arriba, mantiene un agregado de tuplas de la forma $T = (V, E, D, C)$ indexadas mediante la componente *v* en un *Map*. Para cada *v* en el dominio del *Map* tenemos, en la tupla asociada, la distancia hacia el origen, la arista que señala el camino hacia el origen y si el camino mínimo es ya el mínimo posible.

Las tuplas se mantienen ordenadas en un montón de Fibonacci según la distancia al origen.

Partimos de un estado inicial que contiene sólo el vértice origen *vo.* En un estado cualquiera seguimos explorando las aristas salientes del vértice actual. Para cada arista *e* y cada vértice opuesto al actual *v* actualizamos el estado.

Para ello se actualiza el camino al origen desde el vértice *v.* Si *v* no había sido encontrado previamente ($v \notin m$) o el camino a través de a es más corto que el camino previamente encontrado ($v \in m, m(v).d \geq da$) entonces el camino hasta el origen empieza por la arista *e* y su distancia es $da = m(a).d + w(e)$. En los restantes caso el camino previamente encontrado permanece (distancia a través de *a* más larga o vértice que ya había encontrado el camino mínimo).

El algoritmo anterior puede implementarse como un iterador. El siguiente vértice a ser devuelto es el vértice actual. Hay vértice siguiente siempre que *f* (el montón de Fibonacci) no esté vacío. En cada estado los vértices que ya hayan ocupado previamente la posición actual ya tienen calculado el camino mínimo. Este camino se puede reconstruir a partir del Map *m*. En efecto cada vértice mantiene (en la tupla asociada en *m*) la arista que define su camino mínmio al origen. Siguiendo esa arista encontramos el vértice opuesto que nuevamente tiene su arista asociada, etc. El camino indicado podemos mostrarlo como un valor de del tipo *GraphPath<V,E>,* proporcionado por *jGraphT* y ya hemos visto en capítulos anteriores.

El *montón de Fibonacci* es una estructura de datos es muy eficiente para implementar colas de prioridad modificables. Es decir conjuntos de objetos ordenados según el valor de una propiedad y dotados de operaciones de mínimo, eliminación de mínimo, inserción, borrado, y modificación (a la baja) del valor de la propiedad. Los detalles pueden encontrarse en la literatura. Para insertar o eliminar objetos de tipo *T* en un montón de Fibonacci tenemos que hacerlo a través de un *FibonacciHeapNode.*

```
class FibonacciHeapNode<T> {
       T getData();
       double getKey();
       String toString();
}
```

Los objetos en el montón están ordenados según el valor de la propiedad *Key*. El constructor *FibonacciHeapNode(T data).*

Las operaciones y propiedades ofrecidas por el montón de Fibonacci son:

```
class FibonacciHeap<T>{
       void decreaseKey(FibonacciHeapNode<T> x, double key);
       void delete(FibonacciHeapNode<T> x);
       void insert(FibonacciHeapNode<T> node, double key);
       boolean isEmpty();
       FibonacciHeapNode<T> min();
       FibonacciHeapNode<T> removeMin();
       static FibonacciHeap<T> unión(FibonacciHeap<T> h1, FibonacciHeap<T> h2);
}
```

Con el constructor *FibonacciHeap().* Los nombres de los métodos describen suficientemente la funcionalidad ofrecida.

La implementación en *jGrapht* de este algoritmo está dado en al clase:

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

7.2 Algoritmo de Dijsktra.

Como hemos comentado anteriormente el problema de los Camino Mínimo se enuncia así:

Dado un grafo con peso y dos vértices del mismo encontrar el camino mínimo desde un vértice al otro.

El algoritmo de ***Dijsktra*** resuelve de forma eficiente este problema. El algoritmo de ***Dijsktra*** usa el *Iterable del Siguiente Vértice más Cercano* visto previamente. Parte de un vértice inicial y va recorriendo los vértices según el iterable anterior hasta que encuentra el vértice final. Posteriormente reconstruye el camino mínimo según se ha explicado.

El problema de los Camino Mínimo se puede generalizar a este otro

Dado un grafo con peso y encontrar los caminos mínimos de un vértice inicial a los vértices de un conjunto dado.

El nuevo problema se resuelve cun una generalización del algoritmo de ***Dijsktra***. De nuevo partimos de un vértice inicial y vamos recorriendo los vértices según el iterable anterior hasta que encontrar todos los vértices del conjunto proporcionado. Posteriormente reconstruye el camino mínimo para cada vértice.

La implementación en *jGrapht* de este algoritmo está dado en al clase

- *DijkstraShortestPath(Graph<V,E> graph, V startVertex, V endVertex)*

7.3 Algoritmos A*

Lo algoritmos ***A**** tienen también como objetivo encontrar el camino mínimo entre dos vértices de un grafo y son una generalización del algoritmo de ***Dijsktra.*** Usan el mismo esquema que el algoritmo de ***Dijsktra*** pero basándose en una generalización del ***Iterable Siguiente Vértice***

Más Cercano visto previamente. A este nuevo iterable lo denomiaremos ***Iterable Siguiente Vértice Más Cercano Generalizado.*** Solamente comentaremos los aspectos que lo diferencian del anterior.

Ahora el coste de un camino L del vértice inicial al actual vendrá dado por

$$G(a) = \sum_{v \in L, e \in L} w(v) + w(e) + w(v, e_e, e_s)$$

Donde:

- *w(v):* peso del vértice
- *w(e):* peso de la arista
- *w(v, ee, es):* peso de un vértice relativo a la arista que entra ee y la arista que sale del mismo.
- L un camino del vértice inicial *o* hasta el actual *a*.

Junto a la anterior usaremos una estimación $H(a)$ del coste del camino desde vértice actual *a* hasta el final. El coste calculado de un camino desde el vértice inicial hasta el final que pasa por el vértice actual *a* será

$$F(a) = G(a) + H(a)$$

Asumimos que si $H^*(a)$ es el coste real del camino más corto desde el vértice actual al vértice final entonces $H(a)$ cumple la condición $H(a) \leq H^*(a)$. Es decir asumimos que el algoritmo *A** es **admisible**.

Asumimos también que $H(x)$ cumple $H(x) \leq d(x, y) + H(y)$ para todo vértice *x* e *y* conectados por una arista *e*. Por *d(x,y)* representamos del camino de *x* a *y* (según el *G* anterior). Es decir $d(x, y) = w(x, e_e, e_s) + w(e) + w(y)$. Diremos que $H(a)$ es **monótona o consistente**.

Ahora aumentamos la información mantenida en las tupas para que sean de la forma $(V, E, D1, D2, C)$ dónde *V* es un vértice, *E* la arista que indica su camino hacia el origen, *D1 (*de tipo *Double)* su distancia hasta el origen siguiendo la arista anterior, *D2 (*de tipo *Double)* peso del camino que pasa por el vértice actual (D2 más heurística hasta el vértice final y *C* (de tipo *Boolean*) si el camino representado ya es el mínimo posible.

Es decir $D1 = G(a), D2 = F(a)$.

Por simplicidad si t es una tupla de la forma $T = (V, E, D1, D2, C)$ representaremos respectivamente por *t.v*, *t.e, t.d1, t.d2 y t.*c las sucesivas componentes. El agregado de tuplas anterior lo mantendremos organizado mediante un *Map<V,T>* y un montón de *Fibonacci<T>* ordenado según la componente *t.d2*.

El iterable *Siguiente Vecino más Cercano Generalizado* es esencialmente el mismo que el iterable Siguiente Vecino más Cercano visto anteriormente. La única diferencia es que las tuplas son ahora de la forma $(V, E, D1, D2, C)$ explicada anteriormente y el montón de Fibonaccci está ordenado, ahora por la componente *d2*.

Como en el algoritmo de ***Dijsktra*** los algoritmos **A*** usan el *Iterable del Siguiente Vértice más Cercano Generalizado* visto previamente. Parten de un vértice inicial y van recorriendo los vértices según el iterable hasta que encuentra el vértice final. Posteriormente reconstruye el camino mínimo según se ha explicado con la información contenida en las tuplas indexadas en el estado.

7.4 Bosque de Recubrimiento Mínimo: Algoritmo de Kruskal

El problema del recubrimiento mínimo se enuncia así:

Dado un grafo no dirigido y cuyas aristas tienen pesos entonces un **bosque de recubrimiento mínimo**, de ese grafo, es un subconjunto de sus aristas que forman un bosque, incluyen a todos su vértices y el peso total de todas las aristas en el bosque es el mínimo posible.

Si el grafo es conexo entonces el bosque de recubrimiento mínimo tiene una sola componente que se denomina **árbol de recubrimiento mínimo**. Si el grafo no es conexo entonces existe un bosque de recubrimiento mínimo y un árbol de recubrimiento mínimo para uno de las componentes conexas que forman dicho grafo no conexo.

El **algoritmo de Kruskal** busca un bosque de recubrimiento mínimo para un grafo no dirigido conexo o no. Este es un algoritmo *Voraz.* Por los tanto para especificarlo tenemos que indicar las propiedades compartidas, las propiedades individuales representadas en el estado, el estado inicial, la elección de la alternativa y la transición de un estado al siguiente.

Para implementar el algoritmo es conveniente usar una estructura de datos que implementa eficientemente una secuencia de conjuntos. Es la estructura que llamaremos *UnionFind.*

```
class UnionFind<T> {
       void addElement(T e);
       T find(T e);
       void union(T e1, T e2);
}
```

Sus métodos son:

- *void addElement(T e):* Añade el elemento e en un conjunto nuevo
- *T find(T e):* Devuelve el representante del conjunto donde está e.
- *void union(T e1, T e2):* Une en un solo conjunto los conjuntos en los que están e1 y e2.

Las propiedades compartidas son el grafo de partida y la lista de aristas de este grafo ordenadas por peso de menor a mayor. Las propiedades del estado son un dato (*U*) de tipo *UnionFind*, un valor de un entero (*J*) y un conjunto de aristas (SE).

Los detalles están en la Ficha siguiente:

Algoritmo de Kruskal	
Técnica: Algoritmo Voraz	
Propiedades Compartidas	*g, Graph <E,V>* *la, List<E>, aristas del grafo ordenadas por peso* *n = |la|*
Propiedades del Estado	*u, UnionFind<V>* *j, entero en [0,n-1]* *se, conjunto de aristas*
Invariante: *U* contiene todos los vértices del grafo agrupados en conjuntos. Los vértices en cada conjunto más las aristas del subconjunto de aristas en *SE* que los unen forman un bosque. En cada paso se escoge la arista en la posición *j* y se aumenta *j*. Se escoge, pues, la arista de menor peso.	
Solución: Set<E> Cuando el algoritmo termina *SE* contiene las aristas que definen el recubrimiento mínimo. El número de componentes conexas y los vértices adscritos a cada componente conexa pueden obtenerse a partir de *U.*	
Objetivo: Encontrar un subgrafo de g que sea un bosque y el peso de sus aristas sea mínimo. El grafo puede ser no conexo.	
Estado Inicial: (g^v, 0, ∅)	
Estado Final: j = n	
*Alternativa: a = L*a arista en la posición *j*	
next(a): Si la arista conecta dos conjuntos distintos estos se funden en uno y la arista se añade a *SE*. Si la arista une dos vértices del mismo conjunto, *U* y *SE* no se modifican. Se aumenta *j*.	

Este algoritmo *Voraz* mantiene el invariante: *U* contiene todos los vértices del grafo agrupados en conjuntos. Los vértices en cada conjunto más las aristas del subconjunto de aristas en *SE* que los unen forman un bosque. En cada paso se escoge la arista en posición *j* y se aumenta *j*. Se escoge, pues, la arista de menor peso. Si la arista conecta dos conjuntos distintos estos se funden en uno y la arista se añade a *SE*. Si la arista une dos vértices del mismo conjunto, *U* y *SE* no se modifican. Cuando el algoritmo termina *SE* contiene las aristas que definen el recubrimiento mínimo. El número de componentes conexas y los vértices adscritos a cada componente conexa pueden obtenerse a partir de *U.*

7.5 Árbol de Recubrimiento Mínimo: Algoritmo de Prim

El **algoritmo de Prim** busca un árbol de recubrimiento mínimo para un grafo no dirigido y conexo.

El árbol de recubrimiento mínimo podemos describirlo como un conjunto de aristas.

El algoritmo de Prim es esencialmente el mismo del iterador *Siguiente Vértice más Cercano* que se usa en el algoritmo de ***Dijsktra.*** El iterable acaba cuando recorre todos los vértices de

una componente conexa. Las aristas que definen el recubrimiento son las que aparecen en las *tuplas* asociadas en el *Map* a los diferentes vértices del grafo.

8. Otros algoritmos Voraces sobre grafos: recorrido en anchura, en profundidad y topológico

Junto con el iterable del siguiente vértice más cercano y su generalización (vistos más arriba) podemos definir otros maneras de recorrer los vértices de un grafo. Como anteriormente cada recorrido proporciona una secuencia de vértices y un árbol de recubrimiento del grafo definido por un subconjunto de las aristas del mismo.

El árbol obtenido recubre toda el subgrafo alcanzable desde el vértice origen. En el caso de grafos no dirigidos la zona alcanzable es la componente conexa donde se incluye el vértice de partida.

Vamos a ver los tres recorridos más conocidos (y sus usos): recorrido en profundidad, en anchura y orden topológico.

8.1 Recorrido en profundidad

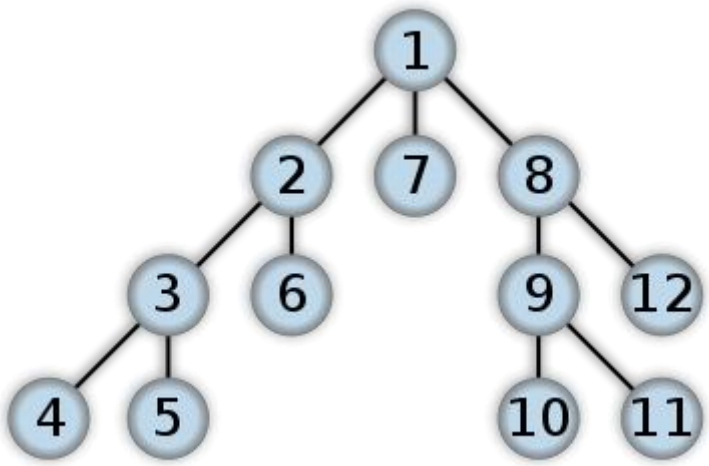
La búsqueda en profundidad tiene varias variantes según se visite un vértice antes, después o entre sus hijos. Las diferentes posibilidades son:

- *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>* de *jGrapht* 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.

Iterable en Preorden	
Técnica: Voraz	
Propiedades Compartidas	*g, Graph <V,E>* *vo, Vértice Origen*
Propiedades del Estado	*m, Map de V sobre E* *p, pila de V* *a = primero(p) o null si p está vacío. Es el vértice actual* *A, Conjunto de Aristas que salen de a*
Invariante: Los vértices en el *Map* ya han sido encontrados previamente y sus imágenes mantienen las aristas del árbol de recubrimiento que se va construyendo. Estas aristas son aquellas a través de las cuales se encuentra un vértice la primera vez. La pila mantiene los vértices que han sido encontrados pero no visitados.	
Inicial: m={vo,null}, p={vo}	
Final: p={}	
Alternativa: a	
$next(a)$: Se saca el vértice *a* (el actual) de la pila y a partir de él se recorren las aristas del conjunto A. Es lo que denominamos visitar el vértice. Para cada vértice *v* opuesto al actual *a, s*i v ha sido encontrado previamente ($v \in dom(m)$) desechamos la arista, si v no ha sido encontrado previamente añadimos la arista al árbol de recubrimiento (añadiendo el par (*v,e*) al Map) y el vértice v a la pila.	

El algoritmo mantiene el siguiente invariante: los vértices en el *Map* ya han sido encontrados previamente y sus imágenes mantienen las aristas del árbol de recubrimiento que se va

construyendo. Estas aristas son aquellas a través de las cuales se encuentra un vértice la primera vez. La pila mantiene los vértices que han sido encontrados pero sus vecinos no han sido todos visitados.

En el recorrido en profundidad en *preorden* se saca el vértice actual de la pila y a partir de él se recorren las aristas salientes (o simplemente incidentes si el grafo es no dirigido). En cada una de ellas se busca el vértice *v* opuesto al actual (proporcionado por la función *ov(e,a)*). Si *v* ha sido encontrado previamente desechamos la arista. Si *v* no ha sido encontrado previamente añadimos la arista al árbol de recubrimiento (añadiendo el par *(v,e)* al Map) y el vértice *v* a la pila.

El iterable en preorden va proporcionando los sucesivos vértices actuales.

El árbol de recubrimiento implícito en el recorrido viene definido por el conjunto de aristas asociadas a los vértices en el Map.

El anterior es el recorrido en *preorden*. El recorrido en *postorden* es similar al anterior pero vértice actual sólo se saca de la pila cuando todos sus vértices adyacentes han sido visitados. Para conseguirlo el *Map m* es ahora de la forma *Map de V sobre (E,C).* Dónde C es una variable de tipo *boolean* que es true si el vértice V y todos sus vecinos han sido visitados. Cuando se encuentra un vértice *v* nuevo (que no pertenece al dominio del *Map*) según la arista *e* se añade al *Map (v,(e,false)).* En este recorrido si el vértice actual tiene asociada una *tupla* con c = false no se saca de la pila pero si se apilan y posteriormente se camia c = true. Si el vértice actual tiene asociada una *tupla* con c = true se saca de la pila.

El iterable en *postorden* va proporcionando solamente los sucesivos vértices actuales cuando se sacan de la pila.

Vemos que en el recorrido en *postorden* anterior cada vértice puede estar dos veces como vértice actual: una vez donde todos los vecinos no hay sido visitados todavía que llamamos de **previsita** y otro posterior donde todos los vecinos han sido visitados que denominamos **postvisita**. Nos referiremos más adelante a estos dos momentos.

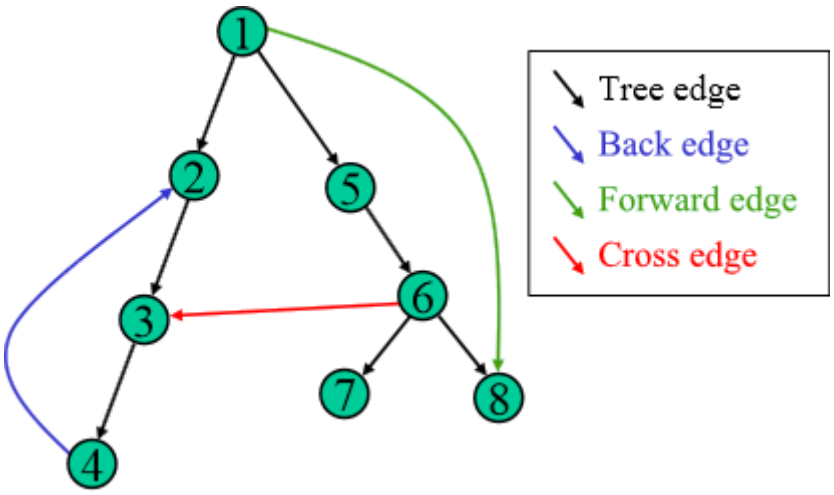
Tenemos, entonces, los siguientes tipos de recorridos en profundidad.

- *Preorden*: Es el orden en el que se visitan los padres antes que cada uno de los hijos (hay varios posibles dependiendo del árbol de recubrimiento definido).
- *Postorden*: Es un orden definido en el que primero se visitan los hijos y posteriormente el padre (hay varios posibles dependiendo del árbol de recubrimiento definido).
- *Postorden Inverso*: Es el orden inverso del *postorden* anterior (como antes hay varios posibles dependiendo del árbol de recubrimiento definido).

8.2 Versión recursiva del recorrido en profundidad: Clasificación del tipo de aristas de un grafo

Veamos ahora una versión recursiva del recorrido en profundidad y sus usos para la clasificación de las aristas en diferentes tipos.

Al recorrer el grafo aparecen varios tipos de aristas que pueden ser clasificadas por el algoritmo: **tree edges (aristas del árbol de recubrimiento)**, que van de un vértice a uno de sus hijos en el árbol de recubrimiento definido, **forward edges** que van desde un vértice a alguno de sus descendientes no inmediatos, **back edges**, que señalan de un vértice a uno de sus antecesores, y **cross edges**, que no son ninguno de los tipos anteriores. La clasificación de las aristas depende del tipo de recorrido y del árbol de recubrimiento concreto.



En el caso de que el grafo sea no dirigido sólo hay *tree* o *back edges*. En el caso del recorrido en anchura solo encontramos *tree edges*, *back edges*, y *cross edges* y no hay *forward edges*. El recorrido en anchura define un árbol de recubrimiento con la propiedad de que el camino desde la raíz a un vértice dado tiene el mínimo número de aristas.

Por comodidad definimos dos funciones adicionales sobre los vértices: *d(v) (discovery time) y f(v) (finish time)*. La primera nos da, para cada vértice, el momento en que es el vértice actual en el recorrido en preorden o en la *previsita* en el recorrido en postorden. La segunda el momento en que un vértice y todos sus descendientes han sido visitados. Es decir hasta el tiempo de la *postvisita* en el recorrido en postorden.

Estas funciones toman valores enteros. Para ir asignando esos valores disponemos de una variable global, *time*, inicializada a cero y que incrementa su valor cada vez que un vértice está en el tiempo *d(v)* o *f(v).* Estas funciones cumplen siempre *1 <= d(v) < f(v) <= 2|V|*. Dónde *|V|* es el número de vértices. Estas funciones son importantes para algunos algoritmos posteriores.

Por facilidad de exposición posterior asignamos tres colores (tres estados posibles) a los vértices según se van procesando: *blanco, gris y negro*. Estos posibles colores para cada vértice en un tiempo dado *time* del algoritmo de recorrido cumplen:

- v es *blanco* si *time < d(v).* Es decir un vértice es blanco en todo tiempo anterior al momento de ser el vértice actual en el recorrido en preorden o en la previsita en el recorrido en postorden. Al principio todos lo vértices son blancos.
- v es *gris* si *d(v) <= time < f(v).* Es decir desde el momento de ser el vértice actual hasta el momento en el cual él y todos sus descendientes han sido visitados. Es decir hasta el tiempo de la postvisita en el recorrido en postorden.
- v es *negro* si *f(v) <= time*. Es decir después de que él y todos sus descendientes han sido visitados.

Con estas ideas podemos caracterizar el tipo de arista que parte del vértice actual. Si *v* es el vértice actual y *e(v,w)* es una arista que se explora por primera vez entonces esta arista puede ser clasificada en función del color de *w*:

- *Tree edge* – si *w* es *blanco*
- *Back edge* – si *w* es *gris*
- *Forward o cross* - si *w* es *negro* y:
  - *Forward edge* – si *w* es *negro* y *d(v) < d(w)* (*w* fue vértice actual después que v)
  - *Cross edge* – si w es negro y *d(v) > d(w)* (*v* fue vértice actual después que *w*)

El esquema de recorrido en postorden anterior puede ser fácilmente ampliado para calcular las funciones $d(v), f(v), t(e)$. Siendo la última el tipo de la arista e. Aquí por complementariedad presentaremos la versión recursiva del recorrido en profundidad y mediante él calcularemos las funciones anteriores.

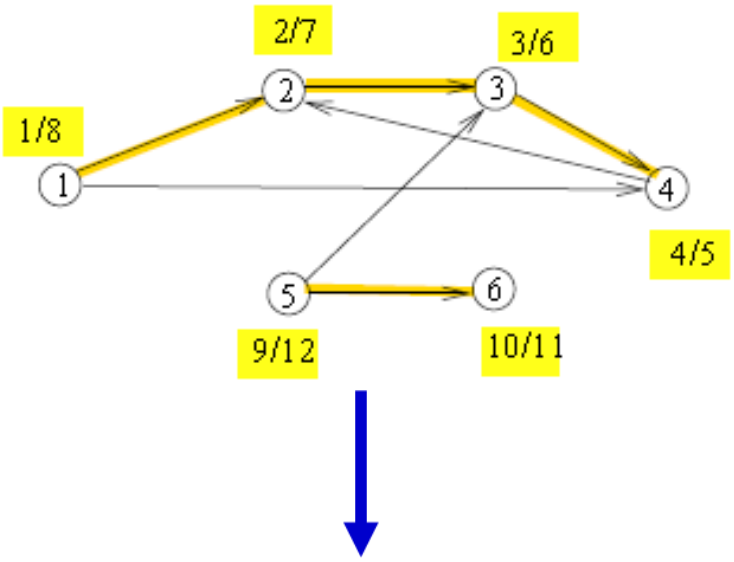
El esquema de recursivo del recorrido en profundidad, la clasificación de las aristas y el preorden y postorden obtenidos se muestra a continuación. El algoritmo usa varias variables globales de tipo función (*color, d, f)* que mantienen el color de cada vértice, el tipo de descubrimiento del vértice y el tiempo de finalización. Otra función global (*tipo*) que cada arista devuelve su tipo. Por último el algoritmo devuelve el conjunto de aristas que definen el árbol de recubrimiento en la variable global *edgeSet*. También calcula las funciones *(va, lc)* que devuelven el vértice anterior y el camino hasta la raíz de la componente correspondiente. Esto nos puede permitir preguntar cada vértice a que componente pertenece. Las funciones *previsit* y *postvisit* generan las secuencias en *preorden*, *postorden* y *postorden inverso* y los devuelven en las listas globales *prev*, *posv* y *invpos. Previsit* añade el vértice *v* al final de la lista *prev*. *Postvisit* añade el vértice *v* al final de *posv* y al principio *de invpos*. El esquema es:

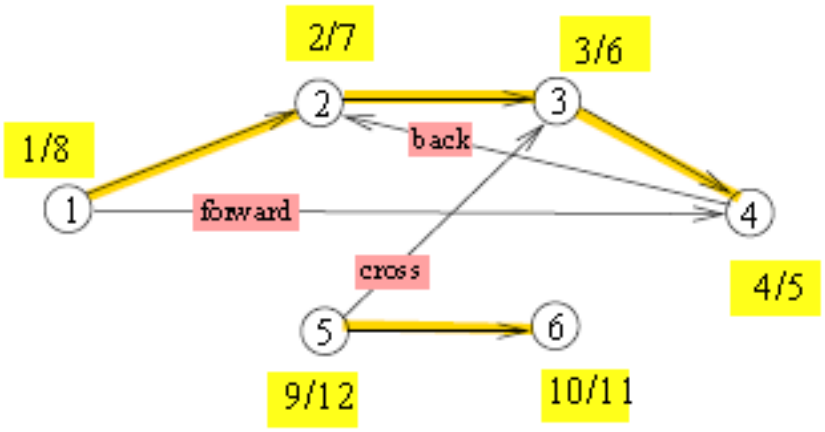
```
dfsAll(G){
    hacer que todos los vértices sean blancos;
    hacer que edgeSet esté vacío;
    inicializar va(v) a null y lc(v) a cero;
    time = 0;
    for( cada v ∈ V)
         if (v es blanco)  dfs(v);
}
```

```
dfs(v,G){
      color[v] = "gris";
      d[v] = ++time;
      previsit(v);
      for (cada w adjacente a v){
           if(w es blanco){
                    tipo(v, w) = "treeEdge";
                    añadir la arista (v, w) a edgeSet;
                    va(w) = v;
                    lc(w) = lc(v)+|(v,w)|;
           } else if(w es gris) {
                    tipo(v, w) = "backEdge";
           } else if(d[v] < d[w]) {
                    tipo(v, w) = "forwardEdge";    // w es negro
           } else {
                    tipo(v, w) = "crossEdge";      // w es negro
           }
       }
      f[v] = ++time;
      postvisit(v);
      color[v] = "black";
}
```

El algortimo, tal como está diseñado, obtiene un bosque de recubrimiento del grafo completo, la clasificación de todas las aristas del grafo y las funciones *d(v) y f(v)* para cada vértice. Si la función color la implementamos como un *Map* entonces no es necesario el color blanco que vendría definido por los vértices que no están en su dominio. Casos particulares del algoritmo anterior son visitar los vértices en preorden o en postorden sin hacer ningún cálculo adicional. A esto casos particulares los llamaremos: *dfsPreorder(v,G)* y *dfsPostorder(v,G)* cuyas versiones iterativas vimos más arriba.

En el ejemplo siguiente se muestran los valores de *d(v)* y *f(v)* obtenidos por el algoritmo para cada vértice, el tipo de arista y el bosque de recubrimiento para un grafo dirigido concreto.





A partir del gráfico es sencillo obtener los valores de *va(v)* y *lc(v).* Aplicando el algoritmo podemos obtener, también el preorden, postorden y postorden inverso. Como hemos dicho estos órdenes dependen del recorrido en particular y en concreto del bosque de recubrimiento construido. En este caso concreto el preorden viene dado por: 1, 2, 3, 4, 5, 6. El postorden por 4, 3, 2, 1, 6, 5. Y el postorden inverso por 5, 6, 1, 2, 3, 5, 4.

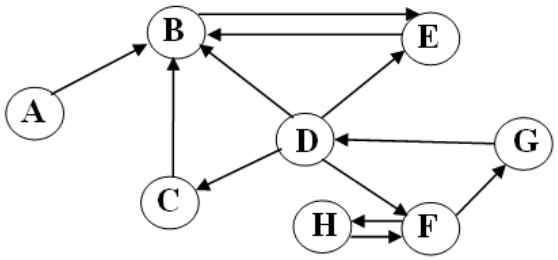
8.3 Aplicaciones del recorrido en profundidad: Componentes fuertemente conexas

El algoritmo anterior se puede usar dentro que otro que es adecuado para el cálculo de las componentes fuertemente conexas de un grafo dirigido. Recordamos que una componente fuertemente conexa de un grafo dirigido *G = (V,E)* es un conjunto maximal de vértices de *G* tales que cada par de vértices *u* y *v* de la misma son alcanzables mutuamente. Es decir existe un camino de *u* a *v* y otro de *v* a *u*. El esquema del algoritmo es:

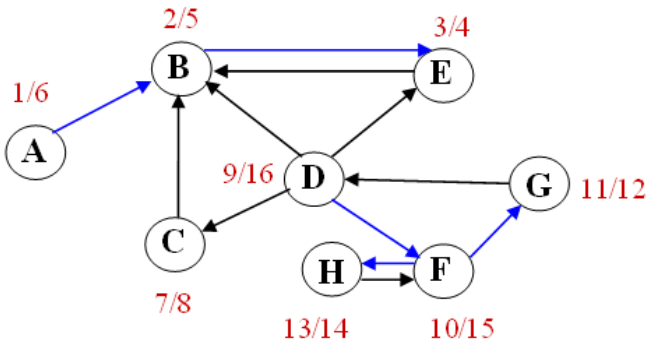
```
dfsSCC(G) {
      nc = 1;
      dfsAll(G); //
      GT = G^T;
      Inicializar cada vértice de GT a no visitado;
      S = Lista de vértices de G en Postorden inverso obtenida anteriormente;
      for(para cada v en S){
              if(v es no visitado){
                    marcar v como visitado;
                    dfsPreOrder(v, GT) //poner a visitado cada vértice alcanzado
                    nc = nc+1;
 // los vértices visitados desde v forman una componentente fuertemente conexa
              }
      }
}
```

Donde G^T es el grafo traspuesto a G. Es decir con las aristas invertidas.

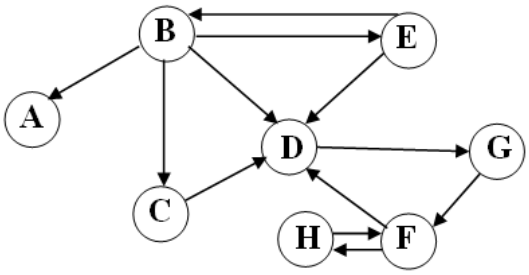
Ejemplo del algoritmo anterior:



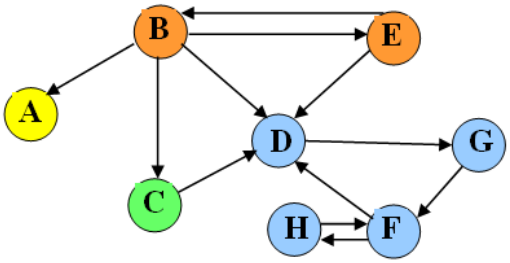
Calculando *d(v)* y *f(v)* para cada vértice:



El postorden inverso es: *D, F, H, G, C, A, B, E*. Y el grafo traspuesto de G abajo.



Ahora el algoritmo empieza por D (el primer vértice en el postorden inverso) y todos los vértices que se alcanzan pertenecen a la misma componente fuertemente conexa (SCC) que D.



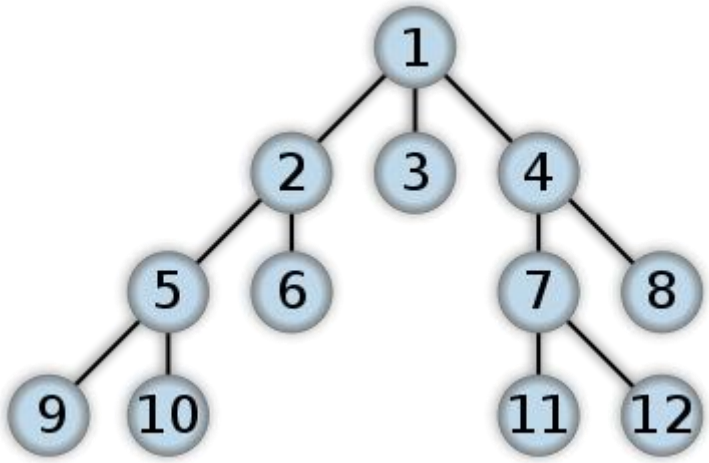
Las componentes fuertemente conexas son SCC(D) = {D, F, G, H}, SCC(C) = {C}, SCC(A) = {A}, SCC(B) = {B, E}

8.4 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)*

Iterable en Anchura	
Técnica: Voraz	
Propiedades Compartidas	*g, Graph <V,E>* *vo, Vértice Origen*
Propiedades del Estado	*m, Map de V sobre E* *p, cola de V* *a = primero(p) o null si p está vacío* *A, Conjunto de Aristas que salen de a*

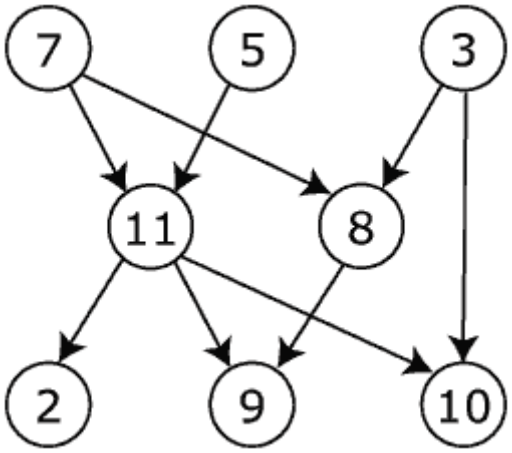
Invariante: Los vértices en el *Map* ya han sido encontrados previamente y sus imágenes mantienen las aristas del árbol de recubrimiento que se va construyendo. Estas aristas son aquellas a través de las cuales se encuentra un vértice la primera vez. La cola mantiene los vértices que han sido encontrados pero no visitados.
Inicial: m={vo,null}, p={vo}
Final: p={}
Alternativa: a
$next(a)$: Se saca el vértice *a* (el actual) de la cola y a partir de él se recorren las aristas del conjunto A. Es lo que denominamos visitar el vértice. Para cada vértice *v* opuesto al actual *a, si* v ha sido encontrado previamente ($v \in dom(m)$) desechamos la arista, si v no ha sido encontrado previamente añadimos la arista al árbol de recubrimiento (añadiendo el par (*v,e*) al Map) y el vértice v a la cola.

El invariante del estado y el esquema de funcionamiento es similar al recorrido de profundidad en preorden pero ahora se usa una cola en vez de una pila.

Como allí el árbol de recubrimiento se mantiene en el conjunto de aristas asociadas a cada vértice en el Map.

8.5 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.

Iterable Topológico	
Técnica: Voraz	
Propiedades Compartidas	*g, Graph <V,E>*
Propiedades del Estado	*sm, Vértices Maximales* *m, Map de V sobre E* *p, pila de V* *a = primero(p) o es un elemento cualquiera de sm si p está vacío o null si p y sm están vacíos.* *A, Conjunto de Aristas que salen de a*
Invariante: Los vértices en el *Map* ya han sido encontrados previamente y sus imágenes mantienen las aristas del árbol de recubrimiento que se va construyendo. Estas aristas son aquellas a través de las cuales se encuentra un vértice la primera vez. La pila mantiene los vértices que han sido encontrados pero no visitados.	
Inicial: m={vo,null}, p={vo}	
Final: sm={} , p={}	
Alternativa: a	
$next(a)$: Se saca el vértice *a* (el actual) de la pila y a partir de él se recorren las aristas del conjunto A. Es lo que denominamos visitar el vértice. Para cada vértice *v* opuesto al actual *a, s*i v ha sido encontrado previamente ($v \in dom(m)$) desechamos la arista, si v no ha sido encontrado previamente añadimos la arista al árbol de recubrimiento (añadiendo el par (*v,e*) al Map) y el vértice v a la pila.	

El orden topológico sigue un recorrido similar al recorrido en profundidad. Parte de los vértices maximales (conjunto de vértices sin aristas de entrada que incluimos en *sm*) y sigue un recorrido en profundidad. Cuando la pila se agota toma otro de los vértices maximales.

9. Problemas Propuestos

9.1 Problema de Aeropuertos

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. Se dispone de una tabla de *tiempos[1..N,1..N,0..23]* de valores enteros, cuyos elementos *tiempos[i,j,h]* contiene el tiempo que se tardaría desde el aeropuerto i al j en vuelo directo estando en el aeropuerto i a las h horas (el vuelo no sale necesariamente a las h horas). Diseñar un algoritmo de vuelta atrás que obtenga el trayecto total de tiempo mínimo entre los aeropuertos origen y destino si nos encontramos en el aeropuerto origen a una hora h dada.

NOTAS:

- No hay diferencia horaria entre los aeropuertos.
- Los retardos en los transbordos ya están incluidos en los datos.
- 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.

9.2 Problema del Laberinto

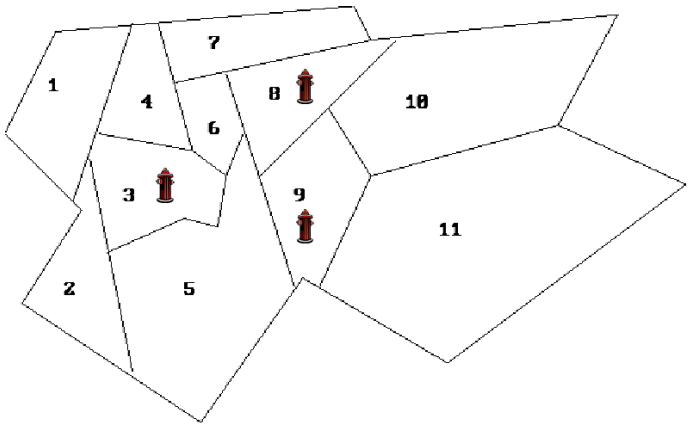
Se tiene un laberinto bidimensional, representado por un objeto de tipo laberinto (recordar ejercicio de prácticas) en el que cada casilla guarda un entero, de forma que en cada casilla puede haber un obstáculo (valor –1), un objeto de valor $v > 0$, o no haber nada (valor 0). La entrada al laberinto se produce por la casilla (1,1) (esquina superior izquierda), y la salida por la casilla (M, N). Para atravesar el laberinto, los únicos movimientos posibles son realizar un paso hacia la derecha o hacia abajo en la matriz, sin pasar dos veces por la misma casilla y sin pasar por los obstáculos.

0	-1	-1	0	0	0	0	1
0	1	0	5	5	-1	0	0
0	3	-1	0	0	0	0	0
-1	0	0	0	-1	0	-1	-1
0	2	-1	1	2	1	0	-1
4	0	5	-1	-1	0	0	0

a) Codificar un algoritmo basado en ***vuelta atrás***, para obtener una de las posibles soluciones óptimas (que maximicen el valor total obtenido). ¿Cómo modificaría el algoritmo si se quieren obtener todas las soluciones óptimas?
b) Teniendo en cuenta que el valor máximo de un objeto es VMAX, ¿cómo cambiaría el algoritmo para minimizar el número de soluciones exploradas? Utilícese como estrategia de búsqueda la selección de la casilla vecina con mayor valor en primer lugar.

9.3 Problema de 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, como se ilustra en el siguiente plano de la misma:



En cada barrio sólo puede existir una estación de bomberos como máximo. Cada estación es capaz de atender los incendios que se produzcan en la zona comprendida por su barrio y los barrios colindantes. Por motivos económicos se desea minimizar el número de estaciones de bomberos que existen en la ciudad sin dejar ningún barrio sin atender en caso de incendio. Por ejemplo, en el plano anterior una de las posibles soluciones sería asignar una estación de bombero a los barrios 3, 8 y 9.

Se Pide:

a) Implementar un algoritmo que resuelva el problema utilizando el esquema vuelta atrás.
b) Definir una función de cota y modifique el apartado anterior para que el vuelta atrás pode el árbol de expansión utilizando dicha función.

Nota: Se dispone de la matriz colindantes: *Array de [1..N, 1..N]* de Boolean que contiene verdadero en la celda *(i, j)* si los barrios *i* y *j* son colindantes y falso en caso contrario.

9.4 Ejercicio 1

Implemente un recorrido que reciba como entrada un grafo etiquetado y dirigido, un vértice de origen y un conjunto de etiquetas. El recorrido sólo podrá viajar a través de aquellas aristas cuyas etiquetas estén contenidas en el conjunto de entrada.

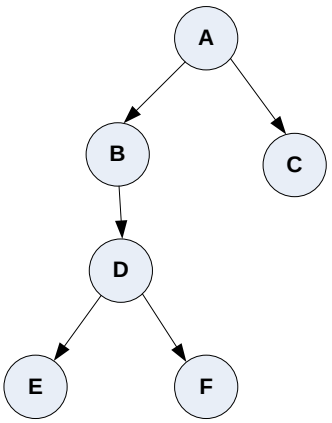
9.5 Ejercicio 2

Implemente un recorrido que reciba como entrada un grafo etiquetado y dirigido, un vértice de origen, un vértice de destino y un conjunto de etiquetas. El recorrido debe comenzar por el

vértice indicado como origen y terminará bien cuando se alance el vértice de destino, o bien cuando ya no queden más vértices por recorrer. Además, el recorrido sólo podrá viajar a través de aquellas aristas cuyas etiquetas estén contenidas en el conjunto de entrada.

9.6 Ejercicio 3

Implemente un recorrido que reciba como entrada un grafo dirigido y un vértice origen y muestre por pantalla todas las posibles rutas. Una ruta será una secuencia de vértices comenzando desde el vértice origen hasta un vértice que no tenga aristas de salida. Asuma que el grafo no tiene ningún bucle. A continuación tiene un ejemplo.



Tomando como nodo inicial el nodo B el resultado sería:
1: B, D, E
2: B, D, F

Tomando como nodo inicial el nodo A, el resultado sería:
1: A, B, D, E
2: A, B, D, F
3: A, C

Tomando como nodo inicial el nodo C, el resultado sería:
1: C

9.7 Ejercicio 4

Implemente un recorrido que reciba como entrada un grafo dirigido y construya dos conjuntos a partir de los vértices del grafo. El criterio para añadir un vértice a uno u otro conjunto es el siguiente: todos los vértices de destino del vértice a añadir deben pertenecer a un mismo conjunto, con lo que dicho vértice se añadirá al otro conjunto (es decir, si un vértice está en el conjunto A, todos sus vértices destino deben estar en el conjunto B y viceversa). Ningún vértice puede estar en más de dos conjuntos o en ninguno.

Además, el recorrido debe indicar si no es posible construir ambos conjuntos a partir del grafo de entrada.