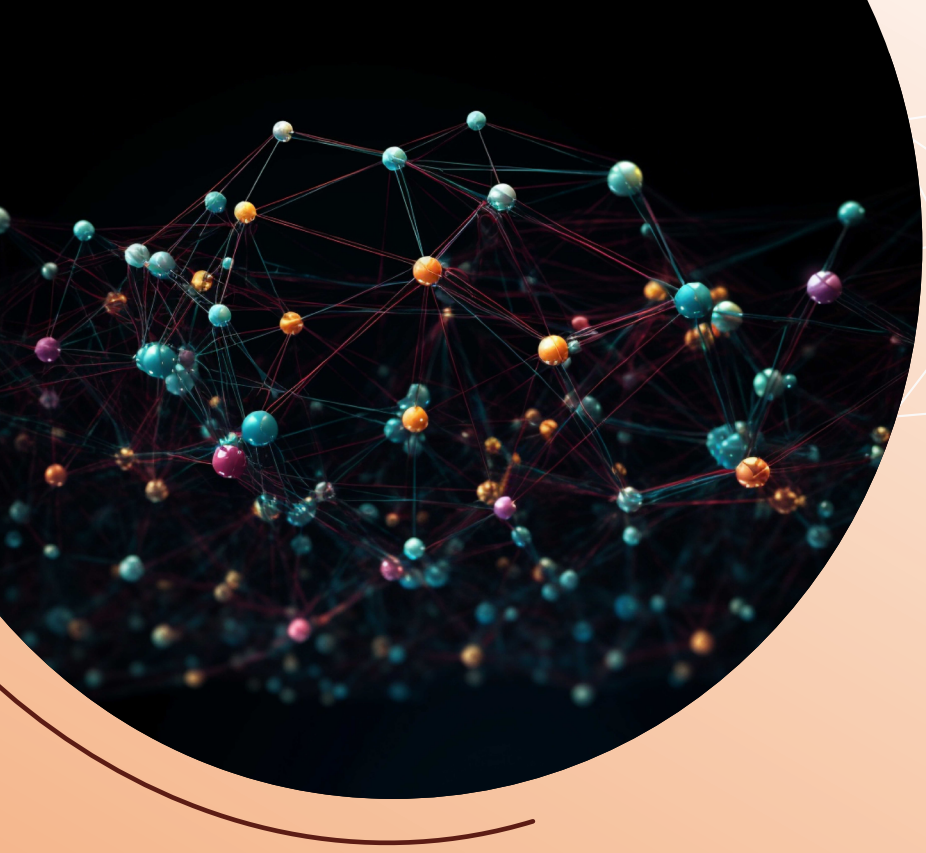


PROBLEMAS, MODELOS, GRAFOS Y ALGORITMOS

Miguel Toro Bonilla

Editorial Universidad de Sevilla

Problemas, modelos, grafos y algoritmos

Problemas, modelos, grafos y algoritmos // Miguel Toro Bonilla

Editorial Universidad de Sevilla

Colección: Manuales de Informática del
Instituto de Ingeniería Informática

Problemas, modelos, grafos y algoritmos // Miguel Toro Bonilla

Miguel Toro Bonilla

Problemas, modelos, grafos y algoritmos

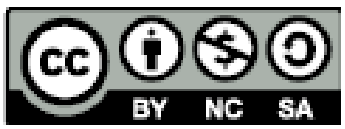
Editorial Universidad de Sevilla

Sevilla 2023

Problemas, modelos, grafos y algoritmos // Miguel Toro Bonilla

Colección: Manuales de Informática del
Instituto de Ingeniería Informática

Núm.: 4



Editorial Universidad de Sevilla 2023
c/ Porvenir, 27 - 41013 Sevilla.
Tlfs.: 954 487 447; 954 487 451; Fax: 954 487 443
Correo electrónico: eus4@us.es
Web: https://editorial.us.es

DOI: https://dx.doi.org/10.12795/9788447225057

Maquetación: Miguel Toro

Diseño de cubierta y edición electrónica:
referencias.maquetacion@gmail.com

Índice

Objetivo y agradecimientos

Este libro está pensado como material de la asignatura Análisis y Diseño de Datos y Algoritmos, que constituye la continuación natural de Fundamentos de Programación. Asumimos, por tanto, que el lector conoce previamente esos contenidos. Están a su disposición, no obstante, en esta misma colección, dos obras de consulta: *Fundamentos de programación: Python* y *Fundamentos de programación: Java*. Estos conceptos los vamos a concretar en el lenguaje de programación Java. En *Algoritmos y tipos de datos en C*, también accesible en esta serie de manuales, se verán estos conceptos en el lenguaje C.

Para abordar el diseño de algoritmos, es necesario tener asimilados los elementos de la programación en algún lenguaje. Conocer Java y sus peculiaridades se hace indispensable en el seguimiento del contenido. Una vez vistas las técnicas sobre diseño de algoritmos iterativos y recursivos, contenidas en *Análisis y diseño de algoritmos y tipos de datos*, abordamos ahora un conjunto de técnicas algorítmicas de uso general: Programación Lineal Entera, Algoritmos Genéticos, Algoritmos A*, Backtracking, Programación Dinámica y otras.

Buscamos de forma prioritaria resolver problemas de optimización aplicando las diversas técnicas algorítmicas y comprender el funcionamiento de las mismas. Con esta finalidad, necesitamos diseñar modelos que, mediante restricciones, expliquen los problemas de una manera lo menos ambigua posible.

Para describir de forma compacta los modelos y los algoritmos, se usa una notación específica que tiene una traducción directa a Java. Al final, se incluye una sección con la notación empleada. Asimismo, se incluyen ejemplos, resueltos muchos de ellos a partir del empleo de las técnicas descritas.

Este texto tiene su origen en la experiencia de varios años de enseñanza de la asignatura Análisis y Diseño de Datos y Algoritmos en la Universidad de Sevilla. El material procede de versiones anteriores que han sido transformadas hasta alcanzar la forma actual. Es mucha la deuda contraída con los profesores Carmelo del Valle, Irene Barba, Rafael Ceballos, Andrés Jiménez, Francisco Ferrer, Fernando de la Rosa y otros muchos, que han detectado errores y sugeridos cambios. A todos ellos les quiero agradecer sus esfuerzos y dedicación. Los defectos que contiene esta obra son responsabilidad única del autor. Vaya un especial agradecimiento a Carmelo del Valle, que ha revisado concienzudamente el texto y sugerido muchas modificaciones útiles.

En https://github.com/migueltoro/adda_v3 puede encontrarse el código de los ejemplos, dividido en proyectos a los que iremos haciendo referencia a lo largo del texto.

En https://github.com/migueltoro/adda_v* se podrán encontrar próximas versiones.

Miguel Toro

Sevilla, septiembre de 2021

Problemas y modelos

En este volumen vamos a abordar problemas que se pueden modelar mediante un conjunto de variables que toman valores en un dominio, una función objetivo y un conjunto de restricciones. El planteamiento general de estos problemas será de la forma:

$$\begin{gathered}\max/\min_{x} f(x) \\ c_i(x), \;\; i = 0, \ldots, n-1 \\ x \in \Omega\end{gathered}$$

Las variables del problema son x que toman valores en un dominio Ω. Normalmente la variable x será una lista de n valores $[x_0, x_1, \ldots, x_{n-1}]$. Dominios posibles son los enteros, reales, etc. A la función $f(x)$ la llamaremos *función objetivo* y de manera general produce un número real que hay que maximizar o minimizar. Las *restricciones* $c_i(x)$ son predicados sobre las variables del problema que restringen sus posibles valores. Ejemplos de restricciones son $g_i(x) \leq 0$, $g_i(x) = 0$, $g_i(x) \geq 0$ donde asumimos que $g_i(x)$ produce un resultado numérico (entero o real según los casos). Si las $g_i(x)$ son polinomios de primer orden en las variables decimos que son *restricciones lineales*. Si, adicionalmente, la función objetivo $f(x)$ también es un polinomio de primer orden en las variables decimos que el *modelo es lineal*. Las restricciones de la forma $a_i \leq x_i \leq b_i$, tomando valores x_i en un dominio numérico las llamamos *restricciones de rango*.

Junto con las restricciones lineales usaremos otros dominios y otras restricciones y función objetivo más generales: no lineales y usando operadores sobre tipos de datos más complejos que veremos más adelante.

Un catálogo de restricciones y la notación asociada puede encontrarse en la sección de Notación y catálogo de restricciones de uso general.

Algunos problemas y sus modelos

Un modelo de un problema viene definido por un conjunto de variables x tomando valores en dominios Ω, un conjunto de restricciones y una función objetivo. Un modelo especifica las soluciones de un problema, pero no encuentra la solución. Definido un modelo debemos encontrar un algoritmo para encontrar la solución o soluciones del problema que verifican el modelo. Veamos algunos ejemplos.

Problema de la mochila

El *Problema de la Mochila* parte de una lista de objetos L de tamaño n. Cada objeto ob_i de la lista es una tupla de la forma $ob_i = (w_i, v_i, m_i)$ donde w_i, v_i, m_i son, respectivamente, su peso, su valor unitario y el número de unidades disponibles. La mochila tiene una capacidad C. El problema busca ubicar en la mochila el máximo número de unidades de cada objeto que quepan en la mochila, teniendo en cuenta las disponibles, para que el valor de estos sea máximo. Se trata del problema de la mochila sin fraccionamiento, y con disponibilidad de múltiples elementos de un mismo tipo.

El problema que abordamos es el de la mochila sin fraccionamiento, y con disponibilidad de múltiples elementos de un mismo tipo.

Si x_i es el número de unidades del objeto i en la mochila el problema puede modelarse como

$$\max \sum_{i=0}^{n-1} x_i v_i$$
$$\sum_{i=0}^{n-1} x_i w_i \leq C$$
$$x_i \leq m_i, \quad i \in [0, n-1]$$
$$int \quad x_i, \quad i \in [0, n-1]$$

Por defecto, asumimos en estos modelos, salvo que se indique lo contrario, que las variables toman valores mayores o iguales a cero.

La primera expresión muestra el objetivo a maximizar: es la función objetivo; la segunda, las restricciones de la capacidad de la mochila; la tercera, las debidas al número máximo de unidades disponibles de cada objeto. Son restricciones de rango. El último enunciado indica que las variables x_i toman valores en los enteros. Sin este enunciado las variables tomarían valores reales.

En este problema la función objetivo y las restricciones son lineales con respecto a las variables. Habrá muchos otros problemas en los que esto no será así.

El modelo anterior involucra variables de tipo entero y operaciones lineales sobre ese tipo de datos. Modelos similares se pueden construir con variables en dominios binarios, la variable solo toma los valores 0 y 1, o reales. Con estos dominios, entero, binario y real, se pueden modelar muchos problemas relevantes.

Problema de la asignación

En este problema tenemos una lista de agentes L y una lista de tareas T ambas del mismo tamaño n. Sea c_{ij} el coste de que el agente i realice la tarea j. Se pretende asignar a cada agente una tarea y sólo una de tal forma que se ejecuten todas las tareas con el coste mínimo.

En un primer modelo del problema escogemos las variables binarias x_{ij} que toman valor 1 si el agente i ejecuta la tarea j y cero si no la ejecuta.

$$\min \sum_{i=0,j=0}^{n-1,n-1} x_{ij}\, c_{ij}$$

$$\sum_{j=0}^{n-1} x_{ij} = 1, \quad i \in [0, n-1]$$

$$\sum_{i=0}^{n-1} x_{ij} = 1, \quad j \in [0, n-1]$$

$$bin\ x_{ij}, \quad i \in [0, n-1],\ j \in [0, n-1]$$

Las restricciones primera y segunda indican que cada agente solo puede tener asignada una tarea y cada tarea un solo agente.

Este es un modelo lineal.

Otra posibilidad es modelar el problema con las variables enteras x_i, donde x_i es la tarea asignada al agente i:

$$\min \sum_{i=0}^{n-1} c(i, x_i)$$

$$AD_{i=0}^{n-1} x_i$$

$$x_i < n, \quad i \in [0, n-1]$$

$$int\ x_i, \quad i \in [0, n-1]$$

Lo que indicamos con $AD_{i=0}^{n-1} x_i$ es que los elementos de la lista $x = [x_0, x_1, \ldots, x_{n-1}]$ deben ser todos diferentes. En la notación que estamos usando una secuencia de variables $x_i, i \in [0, n)$ la consideraremos, alternativamente, como la lista x cuando sea conveniente.

Una tercera posibilidad es:

$$min \sum_{i=0}^{n-1} c(i, x_i)$$

$$P_{i=0}^{n-1}(x_i, i)$$

$$int\ x_i, \quad i \in [0, n-1]$$

Lo que indicamos con $P_{i=0}^{n-1}(x_i, i)$ es que la lista $x = [x_0, x_1, \ldots, x_{n-1}]$ es una permutación de $[0,1, \ldots, n-1]$.

Aquí han aparecido dos nuevos tipos de restricciones $P_{i=0}^{n-1}(x_i, i)$ y $AD_{i=0}^{n-1}x_i$ que no son lineales.

Un catálago más completo de restricciones puede encontrarse en la sección Notación y catálogo de restricciones de uso general.

Problema del camino mínimo

Hay muchos más dominios interesantes. Uno de ellos es la búsqueda de caminos mínimos en un grafo. Un grafo es un conjunto de vértices y conectados por un conjunto de aristas. Para ver detalles de los grafos puede consultarse el libro *Análisis y diseño de algoritmos y tipos de datos* también en esta colección.

Por ejemplo si tomamos un grafo g con n vértices de tipo entero. Es decir un grafo de tipo *Graph<Integer,E>.* El grafo tiene una función de peso asociada a cada una de sus aristas, $w_g(v1, v2)$, y queremos modelar el problema del camino mínimo del vértice s hasta el t asumiendo esos vértices son distintos.

Escogemos las variables $x_i, i \in [0, r)$ de tipo entero donde x_i indica el vértice del camino en la posición i y r el número de vértices del camino.

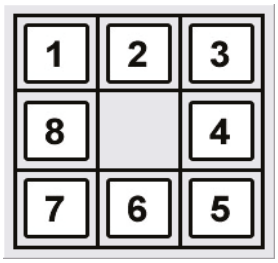
$$min \sum_{i=0}^{r-2} w_g(x_i, x_{i+1})$$
$$x_0 = s$$
$$x_{r-1} = t$$
$$OP_{i=0|g}^{r-1}x_i$$
$$x_i < n, i \in [0, n)$$
$$2 \leq r < n$$
$$int\ r$$
$$int\ x_i, i \in [0, r-1]$$

El modelo indica que buscamos un camino abierto en un grafo g desde el vértice s hasta el t tal que la suma de los pesos de sus aristas sea mínima. El camino vendrá dado por la lista $x = [x_0, x_1, \dots, x_{r-1}]$ de vértices que cumplen la restricción $OP_{i=0|g}^{r-1}x_i$. Es decir que forman un camino simple y abierto en el grafo g. Éste modelo ya involucra, como vemos, tipos de datos y restricciones que no son lineales. Además, el número de x_i es r que

es otra variable. Un modelo no nos da la solución de un problema, solo nos indica las restricciones que debe cumplir la solución.

Problema del 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, marcada con cero, con alguno de sus vecinos mediante movimientos horizontales, hacia la izquierda o derecha, o verticales, hacia arriba o hacia abajo. El problema consiste en dada una configuración inicial llegar a una configuración final (meta) mediante los movimientos permitidos y en el mínimo número de pasos posible. Una configuración concreta del puzle sería:



Un tipo de modelos que puede ser adecuado en éste y otros muchos casos es representar el problema mediante un *grafo implícito o virtual*. Es éste un tipo de grafo donde los vértices y aristas se definen mediante conjuntos definidos por comprensión, es decir mediante un tipo y un predicado que define los valores válidos. Un grafo virtual g de tipo *Graph<V,E>* lo podemos definir mediante dos conjuntos: el conjunto de sus vértices sv y el conjunto se de sus aristas. Así $g = (sv, se)$. Cada uno de estos conjuntos tendrán asociado un tipo de datos y un predicado:

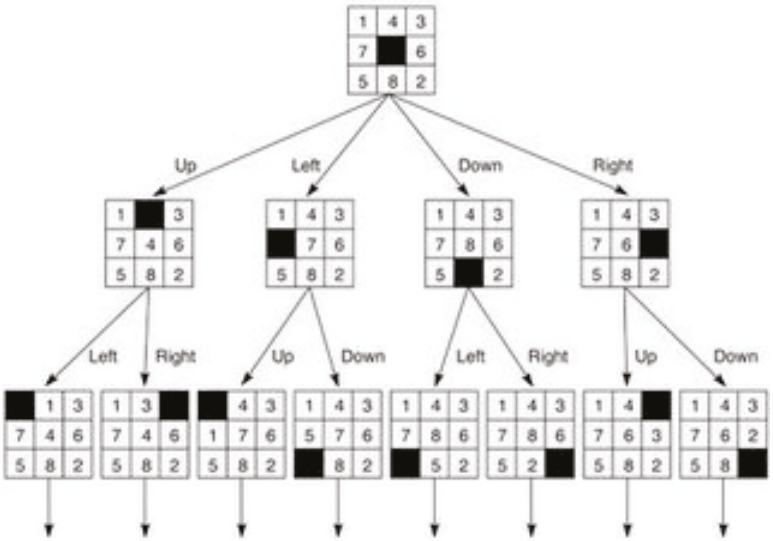
$$sv = \{v{:}\, V \mid isValid(v)\}$$

$$se = \{v1{:}\, sv, v2{:}\, sv \mid isNeigbord(v1, v2)\}$$

El tipo V de los vértices debe ser diseñado específicamente para cada problema. Pero no todos los valores de V serán válidos. Los vértices del grafo serán los valores válidos del topo V. El predicado *isValid(v)* indica si *v* es un vértice válido. Por otra parte *isNeighbor(v1,v2)* será verdadero si la arista *(v1,v2)*, que une los vértices válidos *v1, v2,* existe.

Veamos el tipo de los vértices para un grafo virtual que modele el poblema del puzle.

Hay cuatro posibles acciones o movimientos: *Arriba (Up), Izquierda (Left), Abajo (Down), Derecha (Right).* Estas acciones serán posibles en algunas configuraciones y en otras no. Abajo se incluyen algunas configuraciones tras el movimiento correspondiente.



Cada configuración posible será uno de los vértices de nuestro grafo virtual. Diseñamos el tipo Vp para representar todas las diferentes configuraciones. Este tipo será el tipo de los vértices del grafo.

Los valores del tipo Vp serán todas las configuraciones posibles del puzle. Podemos designar cada casilla por una tupla (f, c) que indique su fila y columna. Sean $p = (f, c)$ los valores de esa tupla que representamos por el tipo *IntPair* y $P = \{(f, c): f \in [0, n), c \in [0, n)\}$ el conjunto de todas las casillas. Este tipo se puede implementar como un *record* de Java.

Para modelar los tipos seguiremos las ideas de la programación orientada a objetos. Desde este punto de vista un tipo se define como un conjunto de propiedades de tipos conocidos y un conjunto de restricciones sobre los valores de esas propiedades, Un tipo tendrá, además, métodos de factoría, parsing, etc. Para ver más detalles sobre el diseño de tipos veanse los libros de *Fundamentos de Programación* en esta misma colección.

Las propiedades del tipo *IntPair* podrían ser *first(),* la primera componente del par, y *second(),* la segunda componente del mismo..

Las propiedades de *Vp*

- Integer f: Fila
- Integer c: Columna

Y las propiedades de *Vp*

- *Integer n:* Número de filas
- $IntPair\ bp()$: Una tupla *(f,c)* con la posición de la casilla negra
- $Integer\ get(p),\ p \in P$*:* Valor entero contenido en la casilla p definida por la fila f y la columna c.

Un vértice es válido si tiene todos los valores de sus casillas distintos y comprendidos entre 0 y n^2.

- $Boolean\ isValid() \equiv AD_{p \in P}\ get(p),\ \forall_{p \in P}\ 0 \leq get(p) < n^2$

Dos vértices son vecinos si teniendo sus casillas negras a distancia de Manhattan 1 se puede obtener uno a partir del otro intercambiando las respectivas casillas negras y permaneciendo las demás iguales. Asumamos que el método $swap(p1)$ produce una nueva configuración tras intercambiar la casilla negra por la que está en la posición $p1$. Entonces tenemos:

- $Boolean\ isNeighbor(Vp\ v) \equiv\ mh(this.bp(), v.bp()) = 1\ \wedge$
  $this.swap(v.bp) = v$

Como en Java, *this* representa el objeto actual que estamos diseñando. Por otra parte la distancia de Manhattan se puede definir como :

$$mh((f1, c1), (f2, c2)) = |f1 - f2| + |c1 - c2|$$

A partir del tipo *Vp* y los predicados *isValid* y *isNeighbor* podemos definir por comprensión el grafo g de tipo *Graph<Vp,DefaultEdge>.*

$$g = (sv, se)$$
$$sv = \{v{:}\ Vp \mid v.isValid()\}$$
$$se = \{v1{:}\ sv, v2{:}\ sv \mid v1.isNeighbor(v2)\}$$

A partir del grafo g nuestro problema es un problema de camino mínimo en ese grafo. Con esos tipos ya podemos diseñar un modelo:

$$\min r$$
$$x_0 = v_0$$
$$x_{r-1} = v_1$$
$$OP_{i=0|g}^{r-1} x_i$$
$$int\ r$$
$$Vp\ x_i, i \in [0, r-1]$$

El modelo define un grafo virtual (por comprensión) y especifica que debemos buscar el camino (simple y abierto) con extremos v_0, v_1 y el mínimo número de vértices. Esto es equivalente a buscar los caminos mínimos donde los pesos en las aristas es 1, ya que se busca en menor número de movimientos.

La variables son x_i cuyo tipo Vp es el que hemos definido previamente para representar las diferentes configuraciones del puzle. El número de variables es r que a su vez es variable. El camino vendrá dado por la lista $x = [x_0, x_1, \dots, x_{r-1}]$ de vértices que cumplen la restricción $OP_{i=0|g}^{r-1} x_i$.

Concretaremos todas estas ideas en el apartado de Grafos Virtuales.

Transformación de modelos

Junto con las restricciones anteriores es conveniente recordar algunas propiedades que nos permitirán transformar unos modelos en otros. Estas transformaciones serán importantes para las técnicas que veremos más adelante.

Transformación de problemas de maximización a problemas de minimización:

$$\min_{x \in \Omega} f(x) \equiv \max_{x \in \Omega} -f(x)$$

Un problema *multiobjetivo* puede ser convertido en otro *uniobjetivo* combinando los diferentes objetivos:

$$\max_{x\in\Omega} f_0(x), \dots, \max_{x\in\Omega} f_{n-1}(x) \equiv \max_{x\in\Omega} \sum_{i=0}^{n-1} \omega_i f_i(x)$$

Donde el valor relativo de las ω_i indicará la prioridad del objetivo correspondiente.

La forma que hemos adoptado para reducir un problema multiobjetivo a otro monobjetivo es solo una de las posibles y en cualquier caso no es la más general. No es el objetivo de esta obra profundizar en las técnicas optimización multiobjetivo. Se remite al lector interesado a la bibliografía incluida en la sección de Bibliografía.

Las restricciones de un problema pueden ser introducidas en la función objetivo. La idea general es convertir una restricción como $c(x)$ en una función numérica $dc(x)$ que llamaremos *función de distancia*, que cumpla $dc(x) = 0$ cuando se cumpla la restricción y $dc(x) \geq 0$ cuando no se cumpla. Es decir:

$$r = dc(x) = \begin{cases} r = 0, & c(x) \\ r > 0, & !\, c(x) \end{cases}$$

Dada una función de distancia a una restricción podemos incluir la restricción en la función objetivo combinando la función objetivo con una penalización.

$$\begin{matrix} \max f(x) \\ c(x) \\ x \in \Omega \end{matrix} \equiv \begin{cases} \max f(x) - k * dc(x) \\ x \in \Omega \end{cases}$$

$$\begin{matrix} \min f(x) \\ c(x) \\ x \in \Omega \end{matrix} \equiv \begin{cases} \min f(x) + k * dc(x) \\ x \in \Omega \end{cases}$$

Escogiendo una constante k suficientemente grande.

Finalmente tendremos que distinguir si la solución obtenida es válida o no. Una solución será válida si las distancias a las restricciones del problema son todas cero.

Un catálogo más completo de distancias a restricciones puede encontrarse en la sección Notación y catálogo de restricciones de uso general.

Programación Lineal Entera

La *Programación Lineal Entera* es una técnica adecuada para encontrar soluciones de un modelo lineal. Actualmente hay buenas herramientas informáticas para resolver problemas de Programación Lineal Entera. Aquí usaremos *Gurobi* como herramienta de referencia.

Un modelo lineal, tal como hemos dicho anteriormente, se compone de variables que toman valores enteros, binarios o reales, un conjunto de restricciones lineales sobre esas variables y una función objetivo lineal. Se permitirán algunas restricciones adicionales que explicaremos más adelante.

Al problema anterior lo denominaremos *Problema de Programación Lineal Entera*. Si las variables son todas reales sin más el problema lo llamaremos *Problema de Programación Lineal.* La *Programación Lineal Entera* tiene un mayor poder expresivo que la *Programación Lineal* pero su complejidad computacional es más alta que polinomial. Si relajamos las variables de tipo entero y binario para considerarlas simplemente reales obtenemos, como hemos dicho, un Problema de Programación Lineal. Al problema resultante se le llama *relajación lineal* del primero y su solución es una aproximación del original. Los problemas de Programación Lineal se resuelven mediante el algortimo del *Simplex* que suponemos conocido. Detalles sobre este algoritmo se pueden encontrar en la bibliografía del final.

Para resolver estos problemas se usa una mezcla de relajación lineal más algoritmos de vuelta atrás que veremos más adelante. El problema relajado linealmente puede resolverse muy eficientemente mediante el algoritmo del *Simplex,* pero con él sólo obtendremos una aproximación a la solución del problema de Programación Lineal Entera. Para resolver este primero se encuentra, mediante el algoritmo del Simplex, la solución de la relajación lineal que es aproximada y posteriormente, mediante vuelta atrás, el valor óptimo entero o binario que respete las restricciones adicionales. Los algoritmos que resuelven los problemas de Programación Lineal Entera tienen complejidades superiores a las polinomiales en el caso peor.

En esta sección estamos interesados en modelos lineales de problemas *y* asumiendo que disponemos de los algoritmos necesarios para encontrar la solución. Estos algoritmos los proporciona *Gurobi*. Para concretar, un Modelo de un Problema de *Programación Lineal Entera* se compone de:

- Un conjunto de variables reales. Una variable por defecto es real
- Un conjunto de variables que toman valores enteros. Se declaran como *int*
- Un conjunto de variables que toman valores binarios cero o uno. Se declaran como *bin*.
- Una función objetivo compuesta como una combinación lineal de las variables
- Un conjunto de restricciones (mayor, menor, igual, mayor o igual, menor o igual) entre combinaciones lineales de variables.

Adicionalmente se pueden añadir otro tipo de variables:

- *Variables Libres*: Las variables que se usan en los modelos de *Programación Lineal Entera* son por defecto positivas. Es decir, hay una restricción adicional por defecto que asegura que cada variable es mayor o igual a cero. Las variables libres son aquellas que no tienen esa restricción. Se declaran indicando explícitamente que su cota inferior y superior es muy grande
- *Semicontinuas*: Son variables que pueden tomar el valor cero o estar incluido en un rango.

La Programación Lineal Entera ha ido progresando y añadiendo otros operadores y restricciones para llegar a ser lo que denominaremos como Programación Lineal Entera Extendida (PLEE). Una de estas restricciones es:

- Restricciones del tipo $b = 1 \rightarrow c(x)$, llamada *restricción con indicador* o *variables indicadoras*. Aquí b es una variable binaria y $c(x)$ es una restricción. La restricción indica que si $b = 1$, entonces la restricción $c(x)$ debe ser verdadera, pero si $b = 0$ la restricción $c(x)$ no tiene que cumplirse obligatoriamente.
- Veremos más de PLEE en secciones posteriores.

Un lenguaje específico de dominio

Como herramienta informática para resolver los problemas de programación lineal entera usaremos *Gurob*i y para facilitar su uso hemos diseñado un lenguaje específico de dominio para escribir modelos de programación lineal entera extendida. A partir del modelo escrito en ese lenguaje que llamaremos *Modelo LSI* podemos obtener un fichero en formato *LP* que puede ser resuelto en *Gurobi*. El *Modelo LSI* es muy compacto, usa listas por comprensión, variables indicadoras, algunas restricciones adicionales como *allDifferents*, restricciones combinadas con *or* y llamadas a funciones externas escritas en *Java*. La sintaxis del lenguaje puede verse en el fichero *PLIModel.g4.html* dentro del proyecto *Solve* del repositorio. Veamos un ejemplo concreto, su modelo y el correspondiente fichero en formato *LSI*.

Problema de la Asignación

En este problema tenemos una lista de agentes L y una lista de tareas T ambas del mismo tamaño n. El coste de que el agente i realice la tarea j sea c_{ij}. Se pretende asignar a cada agente una tarea y sólo una de tal forma que se ejecuten todas las tareas con el coste mínimo.

Asumimos las variables binarias x_{ij} que toman el valor 1 si el agente i ejecuta la tarea j y cero si no la ejecuta. El problema puede ser modelado de la forma:

$$\min \sum_{i=0,j=0}^{n-1,n-1} x_{ij}\, c_{ij}$$

$$\sum_{j=0}^{n-1} x_{ij} = 1, \quad i \in [0, n-1]$$

$$\sum_{i=0}^{n-1} x_{ij} = 1, \quad j \in [0, n-1]$$

$$bin\, x_{ij}, \quad i \in [0, n-1],\ j \in [0, n-1]$$

Ahora las variables son binarias, toman valores cero y uno y tenemos un *Problema de Programación Lineal Entera*. El primer conjunto de restricciones indica que cada agente i tiene que realizar una tarea y sólo una. El segundo conjunto de restricciones indica que cada la tarea j tiene que ser realizada por un agente y sólo uno. Hay un total de $2n$ restricciones.

El fichero *asignación LSI* concreta el modelo anterior en formato *LSI* para ser resuelto:

```
head section

Integer getN()
Integer getM()
Double costes(Integer i, Integer j)
Integer n = getN()
Integer m = getM()

goal section

min  sum(costes(i,j) x[i,j], i in 0 .. n, j in 0 .. m)

constraints section

sum(x[i,j], i in 0 .. n) = 1, j in 0 .. n
sum(x[i,j], j in 0 .. n) = 1, i in 0 .. n

bin

x[i,j], i in 0 .. n, j in 0 .. n
```

Como se puede ver el lenguaje consta de varias secciones: head section, goal section, constrainst section, bin.

La sección *head section* contiene las declaraciones e inicializaciones de variables y de funciones cuyo cuerpo se diseñará en *Java*. Estas variables y funciones pueden usarse dentro del modelo.

La sección *goal section* contiene la declaración de la función objetivo que puede ser *min* o *max* seguida de una expresión. Hay si la función objetivo es una constante independiente de las variables del problema entonces estamos buscando solamente una soluicón que cumple las restricciones.

Las expresiones se construyen con los operadores aritméticos usuales, más el operador sumatorio cuyos índices toman valor en un rango y pueden ser filtrados por un predicado. Un predicado es a su vez una expresión de tipo *boolean*. Los operadores permitidos se indican en la sintaxis del lenguaje.

La sección *constraints section* contiene la declaración de las restricciones del problema. Las restricciones permitidas son:

- Igual
- Menor o igual
- Mayor o igual
- Variable indicadora

Estas son las restricciones básicas. Además de estas hay disponibles otras restricciones más complejas que comentaremos más adelante.

Debemos tener en cuenta que el lenguaje *LSI*, como el formato LP, no distingue entre desigualdades estrictas y no estrictas, por lo que, por ejemplo, < y <= son equivalentes y por lo tanto no se recomienda usar el operador <.

Finalmente están las secciones que declaran cotas para las variables, *bounds section,* la declaración de variables binarias, *bin*, enteras, *int*, libres *free* y semicontinuas *semi-continuous*. Implícitamente todas las variables del modelo son mayores o iguales a cero. Una *variable libre*, *free*, es aquella que no tiene esta restrcción y por lo tanto puede tomar valores negativos. Una *variable semicontinua, semi-continuous,* es aquella cuyos valores deben estar en un intervalo o ser cero. Veremos más adelante otros detalles sobre las variables libres y semicontinuas.

Las variables pueden estar indexadas en la forma *x[i,j].*

Están disponibles listas por comprensión para expresar sumatorios, restricciones, y declaración de variables. Por ejemplo, las expresiones siguientes se escribirían como se indica.

$$\sum_{i=0,j=0}^{n} c_{ij}x_{ij}$$

```
sum(costes(i,j) x[i,j], i in 0 .. n, j in 0 .. m)
```

$$\sum_{i=0,j=0|j>i}^{n} c_{ij}x_{ij}$$

```
sum(costes(i,j) x[i,j], i in 0 .. n, j in 0 .. m | j > i)
```

En la *head section* se declaran un conjunto de variables, funciones y predicados que pueden ser usadas dentro del modelo. El código para esas funciones, que deben ser *static*, se hace en Java y se incluye en una clase que debe ser indicada a la hora de llamar a la herramienta. El código en Java es de la forma:

```
AuxGrammar.generate(TareasPLI.class,"models/tareas.lsi",
       "ficheros/tareas.lp");
Locale.setDefault(new Locale("en", "US"));
GurobiSolution solution =
       GurobiLp.gurobi("ficheros/tareas.lp");
System.out.println(solution.toString((s, d) -> d > 0));
```

La primera línea traduce el fichero de nuestro modelo *tareas.lsi* a un fichero en formato *LP,* el fichero *tareas.lp. Gurobi* lee este fichero y calcula la solución en un valor del tipo *GurobiSolution.* Este tiene como propiedades públicas: *objVal,* de tipo Double, que guarda el valor de la fujcón objetivo y *values,* de tipo *Map<String,Double>,* que asocia a cada variable su valor en la solución. El método *String toString(BiPredicate<String,Double> pd)* nos permite imprimir las variables y valores que cumplen el bipredicado.

Podemos entonces filtrar las soluciones con valor mayor que cero o de alguna otra forma. La variables x[1,2], etc. en el lenguaje LSI se traducen

como *x_1_2* en el formato *LP*. Además se generan otras variables cuyos nombres incluyen $. Detalles sobre el formato LP se pueden encontrar en la documentación de Gurobi que se incluye el la bibliografía al final del libro.

La traducción del lenguaje LSI al lenguaje LP está implementada usando *Antlr4.* Una herramienta que nos facilita el trabajo de construir reconocedores sintácticos y la gestión de los árboles de sintaxis abstracta relacionados. La documentación sobre *Antlr4* se incluye en la bibliografía.

Algunas técnicas útiles en los modelos de Programación Lineal Entera

Las variables binarias son de un uso generalizado en los modelos lineales. Es conveniente conocer la implementación lineal de algunas relaciones lógicas entre este tipo de variables. La justificación se puede encontrar viendo los valores binarios que satisfacen esas restricciones.

Por otra parte es interesante conocer un conjunto de métodos útiles para expresar de forma lineal algunas de las restricciones que podemos ver en la sección de Notación y Catálogo de Restricciones. Estos mecanismos implicarán incluir nuevas variables y en algunos casos incluirlas en al función objetivo.

And

La restricción *x1 = 1 and x2 = 1,* es equivalente a:

$$x1 = 1 \; and \; x2 = 1 \equiv \begin{cases} \mathrm{x1} + \mathrm{x2} = 2 \\ bin \; x1, x2 \end{cases}$$

Or

La restricción *x1 = 1 or x2 = 1,* es equivalente a:

$$x1 = 1 \; or \; x2 = 1 \equiv \begin{cases} \mathrm{x1} + \mathrm{x2} \geq 1 \\ bin \; x1, x2 \end{cases}$$

Or Exclusivo (no ambos)

La restricción *x1 = 1 xor x2 = 1*, es equivalente a:

$$x1 = 1\ xor\ x2 = 1) \equiv \begin{cases} \text{x1} + \text{x2} = 1 \\ bin\ x1, x2 \end{cases}$$

Escoge una de las dos o ninguna, pero no las dos a la vez

La restricción $x + y \leq 1$, $bin\ x, y$ o lo que es lo mismo $x \leq 1 - y$ o también $y \leq 1 - x$ escoge una de las dos o ninguna, pero no las dos a la vez. En efecto si miramos la última versión podemos comprobar que los pares de valores posibles son: (0,0), (1,0), (0,1) pero no (1,1)

Como máximo n

$$\sum_{i=0}^{r} x_i \leq n$$

Implicación entre variables binarias

La restricción *If-then, if x1 = 1, then x2 = 1* es equivalente a:

$$if\ x1 = 1\ then\ x2 = 1 \equiv \begin{cases} \text{x1} \leq \text{x2} \\ bin\ x1, x2 \end{cases}$$

La restricción $bin\ x, y,\ x - y \leq 0$, o lo que es lo mismo $bin\ x, y,\ x \leq y$ impone una relación de implicación entre ambas. Observando la última versión podemos comprobar que los pares de valores posibles son: (0,0), (0,1), (1,1) pero no (1,0). Es decir, si $x = 1$ entonces $y = 1$

Doble implicación

La restricción $bin\ x, y,\ x - y = 0$, o lo que es lo mismo $bin\ x, y,\ x = y$, hace que si $x = 1$ entonces $y = 1$ o si $x = 0$ entonces $y = 0$. Es decir, ambas son verdaderas o falsas a la vez y por lo tanto hay una relación de doble implicación entre ambas. Podemos comprobar que los pares de valores posibles son: (0,0), (1,1) pero no (1,0), (0,1).

Resultante And

La restricción *y = (x1 = 1 and x2 = 1)* es equivalente a:

$$y = (x1 = 1\ and\ x2 = 1) \equiv \begin{cases} y \leq x1 \\ y \leq x2 \\ y \geq x1 + x2 - 1 \\ bin\ y, x1, x2 \end{cases}$$

En el lenguaje *LSI* se puede expresar como:

```
y[] = AND (x[1], x[2])
```

La restricción AND espera una lista de variables binarias.

Resultante Or

La restricción *y = (x1 = 1 or x2 = 1)* es equivalente a:

$$y = (x1 = 1\ or\ x2 = 1) \equiv \begin{cases} y \geq x1 \\ y \geq x2 \\ y \leq x1 + x2 \\ bin\ y, x1, x2 \end{cases}$$

En el lenguaje *LSI* se puede expresar como:

```
y[] = OR (x[1], x[2])
```

La restricción OR espera una lista de variables binarias.

Resultante or Exclusivo (no ambos) o desigualdad entre binarias

La restricción *y = (x1 = 1 xor x2 = 1)* es equivalente a:

$$y = (x1 = 1\ xor\ x2 = 1) \equiv \begin{cases} y \geq x1 - x2 \\ y \geq x2 - x1 \\ y \leq x1 + x2 \\ y \leq 2 - x1 - x2 \\ bin\ y, x1, x2 \end{cases}$$

Con esta restricción podemos expresar la restricción $x1 != x2$, $bin\ y, x1, x2$

Resultante Max

La restricción $y = \underset{i=0,n-1}{max}\ x_i$ es equivalente a:

$$y = \underset{i=0,n-1}{max}\ x_i \equiv \begin{cases} min\ y \\ y \geq x_0 \\ \dots \\ y \geq x_{n-1} \\ int\ y, x_0, \dots, x_{n-1} \end{cases}$$

Esta equivalencia implica que la variable y debe entrar en la función objetivo. En este caso el problema puede convertirse en multiobjetivo que debemos transformar en monobjetivo por las técnicas comentadas en la sección previa de transformación de modelos.

En el lenguaje *LSI* se puede expresar como:

```
y[] = MAX (x[0], x[1], x[2],…, x[n-1]))
```

La restricción MAX espera una lista de variables o valores.

Resultante Min:

La restricción $y = \underset{i=0,n-1}{min}\ x_i$, es equivalente a:

$$y = \underset{i=0,n-1}{min}\ x_i \equiv \begin{cases} max\ y \\ y \leq x_0 \\ \dots \\ y \leq x_{n-1} \\ int\ y, x_0, \dots, x_{n-1} \end{cases}$$

Esta equivalencia implica que la variable y debe entrar en la función objetivo. En este caso el problema puede convertirse en multiobjetivo que debemos transformar en monobjetivo por las técnicas comentadas en la sección previa de transformación de modelos.

En el lenguaje LSI se puede expresar como:

```
y[] = MIN (x[0], x[1], x[2],…, x[n-1]))
```

La restricción MIN espera una lista de variables o valores.

Valor absoluto

La restricción *valor absoluto* se expresa en el lenguaje *LSI* como

```
r = ABS (a)
```

Esta restricción tiene el equivalente

$$ABS(a) = MAX(a, -a)$$

Como vimos la restricción MAX implica que una variable nueva debe entrar en la función objetivo.

La herramienta *Gurobi* ya tiene implementada esta restricción

Valor de una variable en un conjunto

Se trata de modelar que una variable entera toma un valor de entre un conjunto dado de valores $\{v_0, \dots, v_{n-1}\}$.

$$x \in \{v_0, \dots, v_{n-1}\} \equiv x = \sum_{i=0}^{n-1} y_i v_i, \sum_{i=0}^{n-1} y_i = 1, bin\ y_i, i \in [0, m)$$

Donde y_i son variables binarias.

Si los valores pueden ser generados por una función $f(i)$ y hacemos $f(i) = v_i$ representaremos la restricción como $I_{i=0}^{n-1}(x, f(i))$.

En el lenguaje LSI las listas de valores se puede expresar por enumeración o por comprensión. Si tenemos el conjunto de valores posibles {0, 2, 3} la restricción se expresa en el lenguaje LSI se expresa como:

```
x[] in 0, 2, 3
```

Si los valores están indicados por compresión de la forma $L_{i=0}^{3-1} i$, el conjunto de valores es 0, 1, 2, y al restricción se escribe como:

```
x[] in v in 0 .. 3
```

En las listas por comprensión en rango de los valores del índice es cerrado por la izquierda y abierto por la derecha. Es uso de esta restricción supone introducir las *n* variables adicionales y_i.

Permutaciones

La restricción toma dos listas. Las listas pueden estar expresadas por comprensión o simplemente por enumeración. El tamaño de la lista de la izquierda debe ser menor o igual la de la derecha. La denominamos de forma compacta $P_{i=0,j=0}^{n-1,m-1}(f(i),g(j))$.

En el ejemplo siguiente la lista de la izquierda está expresada mediante comprensión y la de la derecha mediante enumeración.

```
permutation(x[i], i in 0 .. 3 ; 0, 2, 3)
```

La primera lista expresada por compresión es *x[i], i in 0 .. 3* y la segunda expresada por enumeración es *0, 2, 3*. Ambas listas están separadas por un punto y coma.

La restricción expresa que las variables de la lista izquierda son una permutación de los valores de la lista de la derecha si las listas tienen el mismo tamaño y de un prefijo si la de la izquierda tiene un tamaño menor. Este problema se modela internamente como un problema de asignación donde puede haber más tareas (segunda lista) que agentes (primera lista).

$$x_i = \sum_{j=0}^{m-1} r_{ij}\, y_j\,, \qquad i \in [0,n)$$

$$P_{i=0,j=0}^{n-1,m-1}(x_i,y_j) \equiv \quad \sum_{j=0}^{m-1} r_{ij} = 1, \quad i \in [0,n)$$

$$\sum_{i=0}^{n-1} r_{ij} \leq 1, \quad j \in [0,m)$$

$$int\ x_i, bin\ r_{ij}, i \in [0,n), j \in [0,m)$$

Es uso de esta restricción supone introducir las $n * m$ variables adicionales r_{ij} que los valores y_j sean constantes.

Programación Lineal Entera Extendida

Las herramientas informáticas actuales ofrecen un conjunto de restricciones que no son estrictamente lineales, pero pueden reducirse a ellas con algunas transformaciones adicionales y algunos dominios de variables nuevos como variables semicontinuas y variables libres.

Estas restricciones y dominios adicionales se ofrecen en Gurobi y están incluidas en el lenguaje *LSI.*

Variables libres

Por defecto en Programación lineal entera las variables son mayores o iguales a cero. Una variable libre es la que no tiene esta restricción. Esto se expresa indicando que la variable tiene cotas inferiores bajas y superiores altas

Se declaran en el lenguaje *LSI* en la sección *free.*

Una variable libre se puede sustituir por la diferencia de dos variables con cota inferior cero.

$$free\ x \equiv\ y1 - y2$$

Restricción con indicador

Son restricciones del tipo $b = 1 \rightarrow c(x)$, llamada *restricción con indicador* o *variables indicadoras*. Aquí b es una variable binaria y $c(x)$ es una restricción. La restricción con indicador indica que si $b = 1$, entonces $c(x)$ debe cumplirse pero si $b = 0$ la restricción $c(x)$ no tiene porqué verificarse.

Se pueden expresar en el lenguaje *LSI* como

```
x[3]=1 -> y[5] - y[7] <= 0
```

Una restricción con indicador se puede implementar con la llamada técnica del *big M.*

$$x = 1 \rightarrow a\ y\ \leq b \equiv \begin{cases} a\ y \leq b\ +\ M(1 - x) \\ bin\ x \end{cases}$$

Siendo *M* una constante suficientemente grande. Efectivamente si $x = 1$ la restricción se reduce a $a\ y \leq b$ y si $x = 0$ entonces a $a\ y \leq b\ +\ M$ que simpre se cumple si M es suficientemente grande. El valor de M es dependiente de la restricción. Hay una amplia literatura sobre este tema pero no profundizaremos más en él aquí.

Variables semicontinuas

Son variables cuyo valor puede ser cero o estar en un intervalo que no contiene el cero. Es decir:

$$x = 0\ \vee a \leq x \leq b$$

Se declaran en el lenguaje LSI en la sección *semi-continuous.*

Pueden implementarse usando variables indicadoras

$$x = 0\ \vee a \leq x \leq b \equiv \begin{cases} r = 1 \rightarrow x = 0 \\ r = 0 \rightarrow a \leq x,\ \ x \leq b \\ bin\ r \end{cases}$$

Desigualdad

Modelo de la restricción *a != b*. siendo ambas enteras

La restricción $a\ \neq b$ es equivalente a

$$r\ =\ |a - b|, \qquad r \geq 1, int\ a, b, r$$

En el lenguaje *LSI* se expresa como

```
x1 != x2
```

Si las variables fueran binarias la restricción se puede expresar como un operador *xor*. Entonces *x1 != x2,* es equivalente a:

$$x1! = x2 \equiv \begin{cases} x1 + x2 = 1 \\ bin\ x \end{cases}$$

Usando las variables indicadoras:

$$x1\ ! = x2 \equiv \begin{cases} r = 1 \rightarrow x1 - x2 \geq 1 \\ r = 0 \rightarrow x2 - x1 \geq 1 \\ bin\ r, int\ x1, x2 \end{cases}$$

Otra alternativa es

$$x1 \; != x2 \equiv |x1 - x2| \geq 1$$

AllDiferent

Modelo de la restricción *allDifferent* y otras relacionadas

Es una restricción que podemos reducir a un conjunto de restricciones *or.*

$$AD_{i=0}^{n-1}(x_i) \equiv \; x_i \neq x_j \, , \qquad i = 0..m-1, j = i+1 \ldots m-1$$

En el lenguaje lsi se expresa como

```
allDifferent(x0,x2,…)
```

La restricción toma una lista de variables e impone que sus valores sean diferentes.

Con las variables indicadoras anteriores podemos implementar la restricción *allDifferent* en la forma:

$$AD_{i=0}^{n-1}(x_i) \equiv \begin{cases} r_{ij} = 1 \rightarrow x_i - x_j \geq 1, \;\; i,j \in [0,n), j > i \\ r_{ij} = 0 \rightarrow x_j - x_i \geq 1, \;\; i,j \in [0,n), j > i \\ bin \, r_{ij}, \;\; i,j \in [0,n), j > i \end{cases}$$

Otra alternativa es

$$AD_{i=0}^{n-1}(x_i) \equiv \; |x_i - x_j| \geq 1, \qquad i = 0..n, j = i+1 \ldots n$$

Restricciones combinadas con el operador or e implica

Sean las m restricciones como:

$$c_0(x)$$
$$\ldots$$
$$c_{m-1}(x)$$

Entonces podemos combinarlas con el operador *or* e incluso indicar que queremos que se satisfagan algunas de ellas y liberar el resto. Esto lo podemos modelar con las siguientes restricciones ampliadas:

$$y_0 = 1 \rightarrow c_0(x)$$
$$\dots$$
$$y_{m-1} = 1 \rightarrow c_{m-1}(x)$$
$$y_0 + \cdots + y_{m-1} \geq k$$
$$bin\ \ y_i, \quad i: [0, m-1]$$

La restricción $y_0 = 1 \rightarrow c_0(x)$ se llama restricción con variable indicadora como hemos dicho arriba. La variable indicadora es y_0, que debe ser binaria. La restricción indica que si $y_0 = 1$ entonces $c_0(x)$ tiene que ser verdadera. Con la última restricción indicamos que se deben satisfacer al menos k de las $c_i(x)$ indicadas. En el lenguaje *LSI* podemos escribir:

```
or(>=,k,c1|c2|…)
```

Con anterior queremos decir que al menos se cumplirán al menos k de las restricciones que se indican. Puede haber otro operador relacional como =, >,<, >=.

Podemos, además, establecer relaciones lógicas entre las restricciones que se satisfacen. La restricción adicional $y_r \leq y_s$, como vimos más arriba, hace que si $y_r = 1$ entonces $y_s = 1$. Es decir que si se cumple $c_r(x)$ también se debe cumplir $c_s(x)$ y por lo tanto hay una relación de implicación entre ambas. Si, por otra parte, añadimos $y_0 + \cdots + y_{m-1} = 1$ entonces indicamos que se cumpla solo una de ellas.

La restricción de implicación entre restricciones la podemos expresar en el lenguaje *lsi* como

```
c1 => c2
```

Esto es equivalente a:

$$y_1 = 1 \rightarrow c_1(x)$$
$$y_2 = 1 \rightarrow c_2(x)$$
$$y_1 - y_2 < 0$$
$$bin\ \ y_i, \quad i: [1,3]$$

Restricciones no lineales

Se pueden expresar en el lenguaje *LSI* como una función lineal a trozos

```
y[] = PWL (x[]) : (0,1) (2,5) (5,13)
```

La secuencia de pares define una función lineal a trozos. Si el valor de *x* está entre dos valores especificados se interpola el valor de *y,* y si el valor de x está fuera del rango especificado se extrapola según la pendiente del último para de pares.

La restricción expresa una ligadura entre las variables *x* e *y* dada por la función lineal a trozos defina por la secuencia de pares.

Complejidad de los problemas de Programación Lineal Entera

La complejidad de los problemas de *Programación Lineal* y los de *Programación Lineal Entera* es muy diferente.

Los problemas de *Programación Lineal* (PL) suelen resolverse por el algoritmo del *Simplex* que, aunque no sea teóricamente polinómico en el caso peor, en la práctica se comporta como polinómico en la mayoría de los casos de interés.

Si a las soluciones de un problema lineal les exigimos que sean enteras, nos encontramos con un problema de *Programación Lineal Entera* (PLE). Este problema es *NP-duro*, concepto que no abordaremos aquí, que indica que su complejidad es más alta que la polinomial y por lo tanto hace imposible la obtención de soluciones exactas para problemas de tamaños grandes. Los algoritmos que se utilizan para resolver este problema suelen comenzar relajando las restricciones del problema para convertirlo en un Problema de Programación Lineal y posteriormente utilizar técnicas de Ramifica y Poda para encontrar la solución entera exacta. Más detalles sobre la complejidad de la Programación Lineal Entera se pueden encontrar en la bibliografía del final del libro.

Como una aproximación a la complejidad se sabe que esta es proporcional al número de huecos ocupados en una matriz que tenga columnas que sean las variables y filas las restricciones. Un hueco ocupado indicaría que

una determinara variable está en una restricción. No es fácil estimar este número en general, pero podemos ver que crecerá al ritmo de número de variables y de restricciones y por lo tanto podemos tener una idea de la complejidad estimando el número de variables y el de restricciones y si podemos ser más precisos el número de variables en restricciones (lo que hemos llamado el número de huecos ocupados).

Algunos ejemplos

Veamos aquí un catálogo de problemas que pueden ser resueltos mediante modelos de Programación Lineal Entera. Al final veremos otros para podemos obtener un modelo, pero no lineal y que nos invita a resolverlos con otras técnicas.

Inversión de capital

El problema se puede enunciar de la siguiente forma: Supongamos que deseamos invertir una cantidad total T. Tenemos identificadas n oportunidades de inversión. Cada oportunidad de inversión requiere de una cantidad c_i y se espera un beneficio v_i con $i \in [0, n)$. ¿Qué inversiones debemos realizar con el fin de maximizar beneficio total?

El problema anterior, de una forma abstracta, es similar al Problema de la Mochila visto anteriormente. Ahora vamos a considerar algunas restricciones adicionales que podríamos querer añadir. Por ejemplo, consideremos limitaciones del tipo siguiente:

1. Sólo podemos hacer 2 inversiones como máximo.
2. Si se hace la inversión r la s también se debe hacer.
3. Si se hace la inversión u la v no se puede hacer y viceversa.

Asumiendo que las variables binarias x_i toman valor 1 si se realiza la inversión i y cero en caso contrario podemos modelar el problema como:

$$\max \sum_{i=0}^{n-1} x_i v_i$$
$$\sum_{i=0}^{n-1} x_i c_i \leq T$$
$$\sum_{i=0}^{n-1} x_i \leq 2$$
$$x_r \leq x_s$$
$$x_u + x_v \leq 1$$
$$bin \quad i, \quad i \in [0, n)$$

La restricción $x_r - x_s \leq 0$, o $x_r \leq x_s$, asumiendo que las variables toman valores binarios, tiene como soluciones posibles para (x_r, x_s) los pares $\{(0,0), (0,1), (1,1)\}$ y por lo tanto si $x_r = 1$ implica que $x_s = 1$.

La restricción $x_u + x_v \leq 1$, asumiendo que las variables toman valores binarios, tiene como soluciones posibles para (x_r, x_s) los pares $\{(0,0), (0,1), (1,0)\}$ y por lo tanto se escoge una de las dos inversiones o ninguna, pero no las dos a la vez.

Problema de las estaciones de bomberos

Para ilustrar este modelo, consideremos el siguiente problema de localización: Una ciudad está considerando la ubicación de sus estaciones de bomberos. La ciudad se compone de n barrios. Cada barrio es vecino de otros barrios y la relación de vecindad se puede representar mediante un grafo no dirigido cuyos vértices representan los barrios y existe una arista entre dos barrios si son vecinos. Una estación de bomberos se puede colocar en cualquier barrio y es capaz de gestionar los incendios, tanto para ese barrio y como para los vecinos. El objetivo es minimizar el número de estaciones de bomberos.

Si $N(i)$ es el conjunto de vértices vecinos del vértice *i* incluido él mismo. y escogemos las variables binarias x_i, $i \in [0, n)$, que tomarán el valor 1 si el vértice v_i es escogido, el problema de las estaciones de bomberos puede ser formulado como:

$$min \sum_{i=0}^{n-1} x_i$$
$$(\sum_{j:N(i)} x_j) \geq 1, \quad i \in [0,n)$$
$$bin\ x_i, \quad i \in [0,n)$$

El modelo LSI será:

```
head section

Integer getN()
Boolean esVecino(Integer i, Integer j)
Integer n = getN()

goal section

min  sum(x[i], i in 0 .. n)

constraints section

sum(x[j], j in 0 .. n | esVecino(i,j)) >= 1, i in 0 .. n

bin

x[i], i in 0 .. n
```

El predicado *esVecino(i,j)* se define como:

$$esVecino(i,j) \equiv j \in N(i)$$

Problema de las reinas

El problema consiste en colocar n reinas en un tablero de ajedrez $n \times n$ de tal manera que ninguna de ellas amenace a ninguna de las demás. Una reina amenaza a las casillas de la misma fila, de la misma columna y de las mismas diagonales. Las filas y columnas toman valores en $0..n-1$. Escogiendo las variables binarias x_{ij} que toman valor 1 si ubicamos una reina en la casilla (i,j) el problema puede ser formulado como:

$$\sum_{i=0}^{n-1} x_{ij} = 1, \qquad j = 0, \dots, n-1$$

$$\sum_{j=0}^{n-1} x_{ij} = 1, \qquad i = 0, \dots, n-1$$

$$\sum_{(i,j)|j-i=dp} x_{ij} \leq 1, \qquad dp = dp1, \dots, dp2$$

$$\sum_{(i,j)|j+i=ds} x_{ij} \leq 1, \qquad ds = ds1, \dots, ds2$$

$$bin\ x_{ij}, \qquad i,j = 0, \dots, n-1$$

El índice *dp* representa las distintas diagonales paralelas a la diagonal principal y *ds* las paralelas a diagonal segundaria: las ecuaciones respectivas son $y = x + dp, y = -x + ds$. Los valores mínimos y máximos de *dp, ds* son por lo tanto: $dp1 = -n + 1,\ dp2 = n - 1,\ ds1 = 0,\ ds2 = 2n - 2$.

En este problema no se trata de encontrar un mínimo. Se trata de encontrar una solución y puede haber muchas. En este caso se podría obviar la función objetivo pero por el diseño del lenguaje *LSI* es necesario poner un valor en la función objetivo que puede ser constante o variable.

El modelo *LSI* para este problema es:

```
head section

Integer n = 150

goal section

min  x[0,0]

constraints section

sum(x[i,j], i in 0 .. n) = 1, j in 0 .. n
sum(x[i,j], j in 0 .. n) = 1, i in 0 .. n
sum(x[i,j], i in 0 .. n, j in 0 .. n | j-i = k) <= 1, k in -n+1
.. n
sum(x[i,j], i in 0 .. n, j in 0 .. n  | j+i = k) <= 1, k in 0
.. 2*n-1

bin

x[i,j], i in 0 .. n, j in 0 .. n
```

El problema puede ser modelado alternativamente usando las variables enteras $x_i, \quad i: [0, n-1]$ y las restricciones:

$$\begin{gathered} \mathrm{AD}_{i=0}^{n-1} x_i \\ \mathrm{AD}_{i=0}^{n-1} [\![(x]\!]_i - i) \\ \mathrm{AD}_{i=0}^{n-1} [\![(x]\!]_i + i) \\ x_i < n, \quad i: [0, n-1] \\ int \ \ x_i, \quad i: [0, n-1] \end{gathered}$$

Las variables x_i representan la ordenada de la casilla donde hay una reina con abscisa *i*. Es decir en la columna *i* hay una reina en la fila x_i. Las filas ocupadas serán los valores de las x_i, las diagonales principales ocupadas serán los valores $x_i - i$ y las diagonales secundarias ocupadas serán los valores $x_i + i$.

El modelo *LSI* es:

```
head section

Integer n = 20

goal section

min  sum(x[i], i in 0 .. n)

constraints section

allDifferent(x[i], i in 0 .. n)
allDifferent(x[i] + i, i in 0 .. n)
allDifferent(x[i] - i, i in 0 .. n)

bounds section

x[i] <= n - 1, i in 0 .. n

int

x[i], i in 0 .. n
```

Si comparamos los tiempos de ejecución para ambos modelos podemos comprobar que es más eficiente con el modelo que usa variables binarias. La razón es que la restricción *allDifferent* aunque es muy compacta se despliega en un número importante de nuevas variables y restricciones or

tal como se puede comprobar en la definición. Vemos que esta restricción, aunque compacta, no es eficiente es la Programación Lineal Entera.

Hay otra forma posible usando variables enteras *x[i]* que representen como antes las ordenadas de las casillas donde hay una reina y la restricción *permutation.* El modelo es:

$$P_{i=0}^{n-1}(x_i, i)$$
$$AD_{i=0}^{n-1}(x_i - i)$$
$$AD_{i=0}^{n-1}(x_i + i)$$
$$int\ \ x_i, \quad i: [0, n-1]$$

Es un modelo muy adecuado para ser implementado mediante algoritmos genéticos por ejemplo, aunque un modelo lineal extendido sería:

```
head section

Integer n = 150

goal section

min  x[0]

constraints section

permutation(x[i], i, i in 0 .. n)
allDifferent(x[i] + i, i in 0 .. n; k, k in -n+1 .. n)
allDifferent(x[i] - i, i in 0 .. n; k, k in 0 .. 2*n-1)


int
x[i], i in 0 .. n
```

Las tres restricciones indican:

- La primera que los valores de las ordenadas son diferentes y toman los valores entre todas las posibles filas
- La segunda que las diagonales principales donde se ubican las reinas son diferentes
- La segunda que las diagonales secundarias donde se ubican las reinas son diferentes

Redes de flujo

Una *red de flujo* es un grafo dirigido donde cada arista tiene una capacidad (que debe ser no negativa) y por ella pasa 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 vértices que producen flujo se llaman fuentes y los que consumen sumideros.

Redes de flujo generalizadas

Aquí vamos a considerar *redes de flujo generalizadas*. En estas redes, además de las propiedades de una red de flujo, cada vértice y arista puede tener asociado un límite inferior, uno superior y un coste unitario del flujo que pasa por él. La red de flujo generalizada tiene, además, una función objetivo que es la suma de los costes del flujo que circula por las aristas, por los vértices y el consumido o producido en ellos. Esta función objetivo podemos maximizarla o minimizarla. Podemos añadir la restricción adicional de que el flujo que circula por vértices y aristas tenga valor entero. Una *red de flujo simple* no tiene cotas asociadas a los vértices, cota inferior cero en las aristas.

Llamaremos *vértices fuente* a aquellos en los que se crea flujo y no tienen aristas entrantes y *vértices sumidero* a aquellos en los que se consume flujo y no tienen aristas salientes.

A partir de un grafo de flujo se puede construir un *Problema de Programación Lineal* (o Programación Lineal Entera).

Las ideas para hacer esa transformación son:

- Partir de una vista del grafo de partida cuyos vértices sean enteros.
- Sean las variables *y[i,j]* asociadas al flujo que pasa por la arista *(i,j)*
- Si fuera necesario sea *x[i]* el flujo que pasa por el vértice *i.* En los vértices intermedios y en los vértices sumidero este flujo es igual al flujo de las aristas entrantes, y al flujo de las aristas salientes en el caso de los vértices fuentes.

- Existen restricciones de flujo asociadas a cada vértice intermedio (leyes de *Kirchoff*): lo que entra es igual a lo que sale.
- Puede haber restricciones asociadas al flujo en las aristas, al flujo en fuentes y sumideros y al flujo que pasa por los vértices intermedios. En cada caso puede haber una cota superior y otra inferior.
- Puede haber un coste unitario asociado al flujo en las aristas, en vértices fuentes y sumideros y al flujo que pasa por los vértices intermedios.
- Maximizar o minimizar la función objetivo que se establezca.

Esa información se puede incluir en un modelo *LSI*.

```
head section

Integer getN()
Boolean containsEdge(Integer i, Integer j)
Double capacidadMax(Integer i, Integer j)
Integer n = getN()

goal section

max sum(y[0,j], j in 0 .. n | containsEdge(0,j))

constraints section

sum(y[j,i], j in 0 .. n | containsEdge(j,i))
     - sum(y[i,j], j in 0 .. n | containsEdge(i,j)) = 0, i in
             1 .. n-1

bounds section

y[i,j] <= capacidadMax(i,j), i in 0 .. n, j in 0 .. n |
containsEdge(i,j)
```

Donde en el ejemplo anterior se ha modelado un problema de flujo máximo siendo *0* el vértice fuente y *n-1* el vértice sumidero y solo teniendo cotas superiores para las aristas. Las variables *y[i,j]* representan el flujo que pasa por la arista *(i,j),* la restricción formula las leyes de *Kirchoff* y la función objetivo maximiza el flujo saliente del vértice fuente. Si hay más información se declararán las correspondientes funciones en la sección de cabecera y se ampliará la sección de cotas (*bounds section*).

También podemos ver que usamos una vista de un grafo cuyos vértices son enteros. Esta vista se implementa mediante la clase *IntegerVertexGraphView<V,E>*. Esta clase implementa *Graph<Integer,SimpleEdge<Integer>>* tomando un *Graph<V,E>* de punto de partida y está disponible en el repositorio.

Las vistas sobre grafos se pueden ver con más detalle en el libro de esta misma colección *Análisis y Diseño de Algoritmos y Tipos de Datos.*

Datos de una red de flujo

Los datos para una red de flujo son:

- Un grafo dirigido
- Las cotas sobre los flujos en aristas y vértices
- Los costes unitarios de los flujos en aristas y vértices

Esta información se puede dar de muchas maneras. Una sería dar un grafo y usar el peso de la arista como capacidad máxima de la misma y el resto de información con diversos *Map<E,Double>* o *Map<V,Double>.*

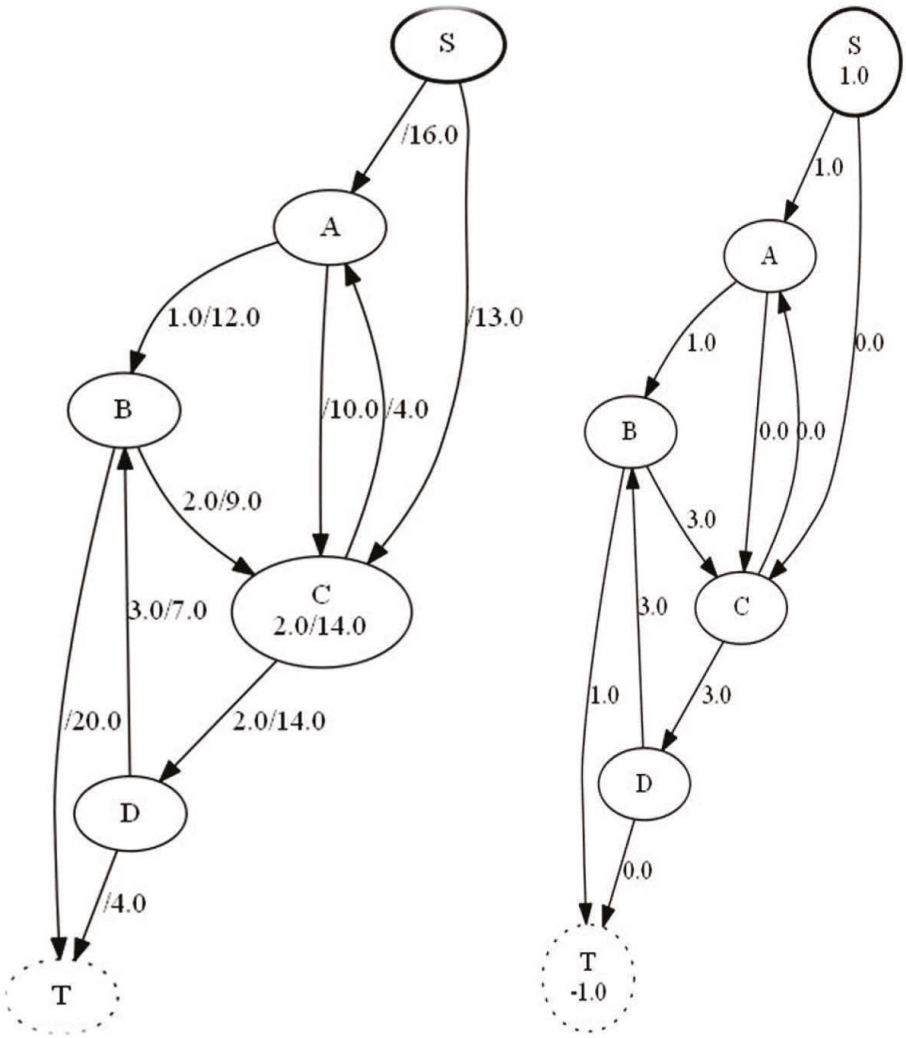
Una forma compacta de proporcionar toda la información necesaria es diseñar un grafo cuyos vértices y aristas guarden toda la información que necesitemos. Sea el tipo de estos vértices *FlowVertex*, el tipo de las aristas *FlowEdge* y el tipo del grafo *FlowGraph*. Este tipo grafo podemos construirlo a partir de un fichero en formato *LSI*. Los detalles de la lectura de grafos a partir de ficheros pueden verse en el libro de esta misma colección *Análisis y Diseño de Algoritmos y Tipos de Datos*.

Veamos como ejemplo la red de flujo que se especifica en el siguiente fichero en formato LSI.

```
#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,cost.* Donde *id* es un identificador único para el vértice, el tipo de vértice (0, intermedio, 1, fuente, 2, sumidero). Si el vértice es intermedio podemos indicar simplemente *id,0* que asume los valores por defecto *id,0,0,inf,0*. Infinito ha sido representado en el fichero por la cadena *inf*. El identificador podría ser un número entero.

Cada arista tiene la siguiente información: *id1, id2, min, max, cost* que representa una arista de los vértices *id1* a *id2* con flujo mínimo *min*, flujo máximo *max* y coste unitario *cost*. Si se exporta el grafo del ejemplo anterior y las soluciones con los métodos *DOTExporter* de *jgrapht* obtenemos:



En la figura de la izquierda se muestra los datos de una red de flujo. El vértice S es un vértice fuente, el T un vértice sumidero y el resto vértices intermedios. En la cada arista se muestran los límites inferior y superior del flujo aceptado en esa arista. En los vértices intermedios se muestran los límites inferior y superior de del flujo que puede circular por ellos.

En la figura de la derecha se muestra la solución de la red de flujo. Es decir el flujo por cada arista, el flujo que se produce en el vértice fuente y el que se consume en el vértice sumidero.

Problemas relacionados con grafos y redes de flujo

Hay un conjunto de problemas relacionados con las redes de flujo que pueden modelarse con modelos lineales.

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 indicado para 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. Para obtener el máximo flujo posible escogemos maximizar la función objetivo asociada al problema.

La clase *EdmondsKarpMFImpl* de *JGrapht* implementa también el algoritmo de *Edmonds_Karp que* resuelve el problema de flujo máximo en una red de flujo simple.

El problema del *corte mínimo* busca encontrar una partición U, V de los vértices de la red tal que U incluye a los vértices fuente, V a los sumideros, y las aristas que van de U a V, que formar un corte, minimizan la suma de sus capacidades máximas. Si designamos por b_j^u la cota supeirior del flujo en la arista j, el problema el corte mínimo puede ser formulado como uno de Programación Lineal Entera en la forma:

$$\min \sum_{j \in E} b_j^u y_j$$
$$x_i - x_j + y_{ij} \geq 0, \quad (i,j) \in E$$
$$x_i = 0, \quad i \in F$$
$$x_i = 1, \quad i \in S$$
$$bin\ x_i, \quad i \in V$$
$$bin\ y_{ij}, \quad (i,j) \in E$$

La variable binaria x_i será 0 si pertenece al conjunto U y 1 si pertenece a V. Por otra parte y_{ij} será 1 si pertenece al corte mínimo y 0 si no pertenece. La restricción $x_i - x_j + y_{ij} \geq 0$ obliga a que $y_{ij} = 1$ si está en el corte $(x_i = 0, x_j = 1)$ y no impone ninguna obligación si no está en el corte $(x_i = 0, x_j = 0$ o $x_i = 1, x_j = 1)$. La función objetivo busca el mínimo de la suma de las capacidades máximas de las aristas que están en el corte $(y_{ij} = 1)$ y escoge $y_{ij} = 0$ para las aristas que no están en el corte. El algoritmo de *Edmonds_Karp* resuelve también el problema del corte mínimo en una red de flujo simple.

Elección de proyectos

En un problema de elección de proyectos, hay n proyectos y m herramientas necesarias. Cada proyecto i produce ingresos $beneficio(i)$ y cada herramienta j cuesta $coste(j)$ si se usa. Cada proyecto requiere varias herramientas y cada herramienta puede ser compartida por varios proyectos. El problema es determinar qué proyectos y máquinas deben seleccionarse y comprarse respectivamente, de modo que se maximice el beneficio. Sea $necesita(i,j)$ verdadero si el proyecto i necesita la herramienta j.

```
head section
…
goal section

max sum(beneficio(i) x[i], i in 0 .. n) - sum(coste(j) y[j], j
      in 0 .. m)

constraints section

x[i] - y[j] <= 0, i in 0 .. n, j in 0 .. m | necesita(i,j)

bin

x[i], i in 0 .. n
y[j], j in 0 .. m
```

Caminos disjuntos en aristas en grafos dirigidos

El problema de los *caminos disjuntos en aristas en grafos dirigidos* trata de encontrar los caminos, que, sin compartir ninguna arista, van de un origen a un destino o a un conjunto de destinos. Se modela como un problema de flujo con flujos 0 o 1 para cada arista y haciendo el origen una fuente y los destinos sumideros. Es decir que asignamos una variable binaria a cada arista. El número de caminos viene dado por el flujo máximo. Cada camino por el conjunto de aristas consecutivas con flujo uno. Se puede generalizar para varios vértices origen tomados de un conjunto y para varios vértices destino tomados de otro conjunto. El modelo escrito en formato *LSI* es:

```
head section

Integer getN()
Boolean containsEdge(Integer i, Integer j)
Integer n = getN()

goal section

max sum(x[0,j], j in 0 .. n | containsEdge(0,j))

constraints section

sum(x[j,i], j in 0 .. n | containsEdge(j,i))
    - sum(x[i,j], j in 0 .. n |
              containsEdge(i,j)) = 0, i in 1 .. n-1

bin

x[i,j], i in 0 .. n, j in 0 .. n | containsEdge(i,j)
```

Caminos disjuntos en vértices en grafos dirigidos

El problema de los *caminos disjuntos en vértices en grafos dirigidos* trata de encontrar los caminos, que sin compartir ningún vértice (excepto los propios origen y destino), van de un origen a un destino o un conjunto de destinos. Se modela como un problema de flujo máximo donde, además de asignar una variable binaria a cada arista restringimos el flujo por cada vértice a 1 como máximo. El número de caminos viene dado por el flujo máximo. Cada camino, como antes, viene dado por el conjunto de aristas consecutivas que tienen flujo uno. La solución en formato *LSI* puede encontrarse en el repositorio.

Conectividad de redes

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.

Problema del transporte

El problema del *transporte* es otro caso particular de red de flujo. Se trata de minimizar el coste del abastecimiento a una serie de puntos de demanda a partir de un grupo de puntos de oferta —posiblemente de distinto número—, teniendo en cuenta los distintos precios de envío de

cada punto de oferta a cada punto de demanda. Los puntos de oferta y de demanda son los vértices y sumideros de la red de flujo. En este problema los costes y beneficios unitarios de los vértices fuente y sumidero son cero. El problema se puede generalizar considerando vértices intermedios encargados de almacenar los productos que se transportan. Estos vértices intermedios no producen ni consumen flujo, pero pueden tener restricciones al flujo que pasa por ellos y coste de almacenamiento en los mismos.

Problema de la asignación

El problema de la *asignación* trata de asignar *n* personas a *m* tareas y como ya hemos comentado. 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 y cuyo flujo puede ser cero o uno. Las personas las modelamos como vértices productores con flujo igual a uno y coste cero. Las tareas las modelamos como vértices consumidores con flujo igual a uno. La solución viene dada por las aristas con flujo uno.

Problema del camino mínimo en grafos dirigidos sin ciclos

El problema de *camino mínimo grafos dirigidos sin ciclos* trata de buscar el camino de longitud mínima desde un vértice origen a un vértice destino en ese tipo de grafos. Este problema se puede resolver mediante programación lineal entera y también por el algoritmo de *Dijkstra*.

Puede ser modelado mediante una red de flujo. Se trata de asignar a cada arista una variable binaria y un coste igual a la longitud de la arista. El vértice origen lo modelamos como un vértice fuente con coste cero y capacidad de producción igual a uno. El vértice destino lo modelamos como un vértice consumidor con coste cero y capacidad de consumo igual a uno. El camino mínimo solución del problema viene dado por la secuencia de aristas que tengan flujo uno.

Aunque es posible resolver el problema del camino mínimo de esta manera, se usa, generalmente, el algoritmo de Dijkstra.

En los grafos con ciclos el problema se concreta en:

- Una variable binaria por arista, 1 si pasa flujo, 0 si no pasa.
- Un flujo de salida del vértice origen de 1.
- Un flujo de entrada en el vértice destino de 1.
- Lo que entra en cada vértice intermedio es igual a lo que sale. Son las denominadas Leyes de *Kirchoff* .
- Función objetivo igual a la suma de los pesos de las aristas por sus variables asociadas.

Camino mínimo en grafos no dirigidos con ciclos

El problema se enuncia igual que antes pero ahora el grafo es no dirigido y puede tener ciclos. Los grafos de las redes de flujo son dirigidos. En ese caso a cada arista asociamos una variable binaria por arista. Si el grafo es no dirigido para poder usar los planteamientos de las redes de flujos a cada arista (i, j) del grafo asociamos dos variables binarias: $y(i, j)$, $y(j, i)$ que tendrán valor 0 o 1 según haya flujo por esa arista en la dirección correspondiente.

La única fuente es v_0, el vértice origen y el único sumidero v_1, el vértice destino. De los vértices origen y destino sale y llega un solo camino. En el resto de las ciudades se cumplen las ecuaciones de *Kirchhoff*.

En los grafos con ciclos el problema tiene varias partes como antes:

- Dos variables binarias por arista. 1 si pasa flujo. 0 si no pasa en la dirección correspondiente.
- El el vértice oringen un flujo de salida de 1 y de entrada cero.
- En el vértice destino un flujo de entrada de 1 y de salida 0.
- Leyes de *Kirchoff* para el resto de los vértices: lo que entra es igual a lo que sale.
- Función objetivo igual a la suma de los pesos de las aristas por sus variables asociadas.
- Unas variables x_i que indican el vértice que se alcanza en el paso i.
- Adicionalmente una restricción para impedir bucles cerrados. Siendo v_0 el vértice de partida esta última restricción es de la forma:

$$x_i - x_j + n\, y_{ij} \leq n - 1, i \in [0, n-1], j \in [0, n-1]$$

$$x_i \leq n-1, i \in [0, n-1]$$
$$x_0 = v_0$$

Las restricciones adicionales están diseñadas para impedir que los caminos de k pasos $x_0, \ldots, x_k$ puedan ser bucles cerrados.

Con esas restricciones no hay caminos cerrados porque si sumamos todas las desigualdades para esa posible ruta cerrada de k pasos se elimina cada valor de x_i con la del paso anterior :

$$x_0 - x_1 + n\, y_{0\,1} \leq n-1$$
$$x_1 - x_2 + n\, y_{1\,2} \leq n-1$$
$$\ldots$$
$$x_{k-1} - x_k + n\, y_{k-1\,k} \leq n-1$$

Al sumar, si el camino es cerrado, $x_0 = x_k$, $y_{ij} = 1$ y cada x_i se anula con la del paso anterior, por lo que obtenemos $k\, n \leq k\, (n-1)$. Lo cual es una contradicción si el camino es cerrado .

Por otra parte si $y_{ij} = 0$, la arista no está en el camino. Por lo tanto, la restricción se reduce a:

$$x_i - x_j \leq n-1$$

*Q*ue se verifica ya que $x_i \leq n-1$.

Todo lo anterior da lugar al modelo para el cálculo del camino mínimo:

```
head section

Integer getN()
Double edgeCost(Integer i, Integer j)
Boolean containsEdge(Integer i, Integer j)
Integer n = getN()
Integer origin()
Integer target()
Integer v0 = origin()
Integer v1 = target()

goal section

min sum(edgeCost(i,j) y[i,j], i in 0 .. n, j in 0 .. n |
containsEdge(i,j))

constraints section

sum(y[v0,j], j in 0 .. n | containsEdge(v0,j)) = 1
sum(y[i,v1], i in 0 .. n | containsEdge(i,v1)) = 1
sum(y[j,i], j in 0 .. n | containsEdge(j,i))
  - sum(y[i,j], j in 0 .. n | containsEdge(i,j)) = 0,
         i in 1 .. n | i != origin() && i != v1
x[i] - x[j] + n y[i,j] <= n - 1, i in 0 .. n, j in 0 .. n |
containsEdge(i,j)
x[v0] = 0

bounds section

x[i] <= n-1, i in 0 .. n

bin

y[i,j], i in 0 .. n, j in 0 .. n | containsEdge(i,j)

int

x[i], i in 0 .. n
```

Problema del viajante en grafos no dirigidos

A diferencia del problema anterior, ahora el camino debe pasar por todos los vértices y debe ser cerrado.

En los grafos con ciclos el problema tiene varias partes como antes:

- Dos variables binarias por arista. 1 si pasa flujo. 0 si no pasa
- Leyes de *Kirchoff* para todos vértices

- Adicionalmente una restricción para impedir bucles cerrados de tamaño menor que n pero permitir el de tamaño n que incluye todos los vértices. Esta última restricción es como antes pero ahora dejando fuera el vértice 0, por el que se permite que pase un camino cerrado de longitud n:

$$x_i - x_j + n\, y_{ij} \leq n-1, i \in [1, n-1], j \in [1, n-1]$$
$$x_i \leq n-1, i \in [0, n-1]$$
$$x_0 = 0$$

```
head section

Integer getN()
Double edgeCost(Integer i, Integer j)
Boolean containsEdge(Integer i, Integer j)
Integer n = getN()

goal section

min sum(edgeCost(i,j) y[i,j], i in 0 .. n, j in 0 .. n |
containsEdge(i,j))

constraints section

sum(y[i,j], i in 0 .. n | containsEdge(i,j)) = 1, j in 0 .. n
sum(y[i,j], j in 0 .. n | containsEdge(i,j)) = 1, i in 0 .. n
x[i] - x[j] + n y[i,j] <= n-1, i in 1 .. n, j in 1 .. n |
containsEdge(i,j)
x[0] = 0

bounds section

x[i] <= n-1, i in 0 .. n

bin

y[i,j], i in 0 .. n, j in 0 .. n | containsEdge(i,j)

int

x[i], i in 0 .. n
```

Algoritmos genéticos

Los algoritmos genéticos son adecuados para resolver problemas de optimización. Estos problemas pueden escribirse de la forma:

$$max/min \ \ f(x)$$
$$c_i(x), \ \ i = 0, \dots, n-1$$
$$x \in \Omega$$

Donde x es un vector de variables y Ω un dominio dónde se escogen los valores de las variables. Las $c_i(x)$ son las restricciones del problema. Una restricción es un predicado sobre las variables del problema. Los valores que cumplen las restricciones los llamaremos soluciones válidas o espacio de soluciones del problema. Podemos considerar que la función objetivo $f(x)$ es una expresión que devuelve valores reales.

Los algoritmos genéticos (GA) son algoritmos iterativos generales de optimización combinatoria. Son algoritmos probabilísticos cuyo diseño está inspirado en los mecanismos evolutivos que se encuentran en la naturaleza. Para emular la selección natural los algoritmos genéticos usan un conjunto de cromosomas que llamaremos población. Esta población, como en la selección natural, evoluciona para que vayan superviviendo los mejores hasta encontrar el o los mejores. Un cromosoma es un objeto que tiene un estado interno con dos propiedades visibles: valor y fitness. El valor es uno de los posibles valores del espacio V de valores del cromosoma. La función de fitness $ft(v)$ es una propiedad que mide el

grado de aptitud del cromosoma. Tendremos disponible, además, una función $s(v)$ que para cada valor del cromosoma nos proporciona una solución x del espacio de soluciones del problema. Esta solución obtenida del cromosoma puede ser válida o no. La población del algoritmo genético evoluciona hasta encontrar el máximo de la función de fitness para uno de los valores del espacio de valores del cromosoma.

Cada cromosoma tiene asociado un espacio de valores. Algunos cromosomas que usaremos aquí tienen los siguientes espacios de valores:

- Listas de ceros y unos de tamaño n. Lo llamaremos *cromosoma binario*.
- Listas de enteros de tamaño n en rangos especificados: $a_i..b_i,\ i \in [0,n)$. Lo llamaremos *cromosoma de rango*.
- Listas de números reales de tamaño n en rangos especificados: $a_i..b_i$, $i \in [0,n)$. Lo llamaremos *cromosoma real*.
- Permutaciones, posiblemente con repetición, de la secuencia de enteros $[0,n)$. Lo llamaremos *cromosoma permutación*.

En un caso general puede ser necesario diseñar nuevos cromosomas con otros espacios de valores asociados. Veremos algunos otros más adelante.

Cada cromosoma tiene un estado interno, su *genotipo*, y su representación externa, su *fenotipo*. El estado interno puede ser de muchos tipos, pero los más frecuentes son: listas de enteros y listas de números reales. A partir de ese estado interno se obtienen los valores del cromosoma que veremos externamente.

Para emular la selección natural los algoritmos genéticos usan un conjunto de cromosomas que llamamos *población*. Una población va evolucionando a través de tres operaciones: *mutación, cruce y selección*.

- Dado un cromosoma podemos obtener otro aplicándole un *operador de mutación* que consiste en modificar aleatoriamente una parte de su estado interno. Un posible operador de mutación modifica una casilla concreta del estado interno del cromosoma. Hay muchos otros operadores de mutación.
- Dados dos cromosomas podemos obtener otro aplicando a ambos un *operador de cruce* que combina aleatoriamente los estados internos de ambos cromosomas para obtener el nuevo. Un operador posible

operador de cruce divide las listas de ambos cromosomas por una posición dada y obtener dos nuevos cromosomas: el primero con la primera parte del primero y la segunda del segundo y el segundo cromosoma con la primera parte del segundo y la segunda del primero. Hay muchos otros operadores de cruce.

- A una población de cromosomas le podemos aplicar un *operador de selección* para escoger un subconjunto de la misma. Una política de selección muy usada es la denominada *selección por torneo*. Consiste en seleccionar sin reemplazamiento un grupo de n individuos al azar y de entre ellos escoger el mejor par. El tamaño del grupo se denomina *aridad del torneo*. Una *aridad* alta implica que los individuos peores casi nunca son escogidos.

Con los anteriores elementos un algoritmo genético evoluciona partiendo de una *población inicial* escogida aleatoriamente y de un *tamaño* fijado. A cada nueva población se le llama *generación*. A esta población se le aplican operadores de selección, cruce y mutación siguiendo el esquema:

1. Se escogen los mejores individuos de una población según una *tasa de elitismo* y se pasan a la siguiente generación sin modificación.
2. Se repiten los siguientes pasos hasta que la nueva población alcanza el tamaño prefijado.
    a. Con la *política de selección* elegida se escogen dos cromosomas para pasar a la siguiente generación. En un porcentaje establecido dado por la *tasa de cruce* se aplica el *operador de cruce* y en ese caso los hijos sustituyen a los padres. Si no hay cruce los cromosomas escogidos pasan sin modificación a la siguiente generación.
    b. A cada individuo de los dos anteriores se le aplica en un porcentaje establecido por la *tasa de mutación* el *operador de mutación* fijado.

El algoritmo termina cuando se cumpla la *condición de finalización* establecida sobre la población. Los criterios de parada más frecuentes son:

- Número de generaciones
- Tiempo transcurrido
- Que existan en la población un número de cromosomas con las condiciones especificadas.

Para resolver un problema mediante algoritmos genéticos (o técnicas aleatorias en general) tenemos que escoger un tipo de cromosoma que genere un espacio de valores V adecuado. Claramente el espacio de valores debe incluir el espacio de soluciones del problema. Es decir $\{x: \Omega | c(x)\} \subseteq \{s(v), v: V\}\}$. El primero es el conjunto de soluciones válidas del problema y el segundo el conjunto de soluciones del problema que puedan ser obtenidos mediante los valores proporcionados por los cromosomas definidos.

Pueden existir distintos tipos de cromosomas adecuados. En principio, la mejor alternativa es aquella en la que el conjunto de solucione obtenidas a partir del *espacio de valores* del cromosoma es de un tamaño cercano al espacio de soluciones del problema. Es decir, el conjunto de *soluciones inválidas* obtenidas a partir del conjunto de valores es lo más pequeño posible.

Para resolver un problema mediante algoritmos genéticos debe tener un modelo en la forma:

$$\begin{array}{c} max \;\; ft(v) \\ v \in V \end{array}$$

Siendo V el espacio de valores del c romosoma elegido y $ft(v)$ la función de fitness. Esto implica que las restricciones deben estar incluidas en la función objetivo. Esto se puede hacer usando *funciones de distancia* asociadas a cada restricción. Esto implica incluir en la función de fitness la minimización de las distancias asociadas a cada una de las restricciones.

Asegurar que los valores proporcionados por los cromosomas son válidos puede ser difícil en la mayoría de los casos, e innecesario por otro lado. Pero si esto no es posible siempre podemos incluir en la función de fitness la minimización de las distancias asociadas a cada una de las restricciones. El problema se convierte en multiobjetivo que posteriormente debemos transformar en monoobjetivo.

Disponiendo de una función de distancia asociada a cada restricción podemos hacer la transformación:

$$\begin{matrix} max\ f(x) \\ c(x) \\ x \in \Omega \end{matrix} \equiv \begin{matrix} max\ f(x) - k * dc(x) \\ x \in \Omega \end{matrix}$$

$$\begin{matrix} min\ f(x) \\ c(x) \\ x \in \Omega \end{matrix} \equiv \begin{matrix} max\ -f(x) - k * dc(x) \\ x \in \Omega \end{matrix}$$

Escogiendo una constante k suficientemente grande.

En la sección Notación y Catálogo de Restricciones se incluye una colección de distancias a las restricciones más usuales.

Se defiende aquí hacer uso de un pequeño conjunto de cromosomas predefinidos junto con sus operadores genéticos, que puedan ser usados para resolver una amplia mayoría de problemas.

Un catálogo de cromosomas

Hay una amplia gama de problemas combinatorios que pueden ser abordados partiendo de un catálogo de cromosomas cuyos operadores de cruce y mutación son conocidos y pueden ser reutilizados. Veamos algunos tipos de cromosomas adecuados para ser usados en los Algoritmos Genéticos. Todos ellos se pueden implementar basándose en dos cromosomas aportados por las librerías de Apache: *BinaryChrosomose* y *RandomKey<T>.* Referencias a las librerías de Apache se incluyen al final del libro.

Para el diseño e implementación de los cromosomas que usaremos partimos del tipo genérico *Chromosome<E>:*

```
public interface Chromosome<E> {
   E decode();
   double fitness();
}
```

- *E decode():* Función de decodificación que proporciona los valores del cromosoma
- *double fitness():* Función de fitness
- *E:* Tipo de los valores del cromosoma

Cada cromosoma tiene, además de la vista *Chromosome<E>,* otra vista con datos adicionales para construir el cromosoma que la representaremos por el tipo *ChromosomeData<E,S>.*

```
public interface ChromosomeData<E,S> {
	Integer size();
	ChromosomeType type();
	Double fitnessFunction(E value);
	S solucion(E value);
}
```

Donde los métodos tienen el significado:

- *size()*: Es el tamaño de la lista del cromosoma y
- *type()*: El tipo del cromosoma
- *fitnessFunction(E value)*: El cálculo del valor del fitness a partir de los valores
- *solucion(E value):* Solución asociada al cromosoma.

Los tipos disponibles en el repositorio son:

```
public enum ChromosomeType{Binary,Range,Real,InSet,SubList,
	Permutation, PermutationSubList,Blocks, Expression}
```

Cada tipo de cromosomas tiene valores de un tipo concreto. Los datos de los diferentes tipos de cromosomas son:

ValuesInRangeChromosome<E,S>

Representa una lista de valores de tipo *E* que están en un rango. Dispondremos de varios subtipos: *BinaryChromosome, RangeChromosome y DoubleChromosome*. Los detalles específicos para cada uno de estos cromosomas se indicarán en una implementación del tipo *ValuesInRangeProblemAG<E,S>.* Dónde *E* es el tipo de los elementos del cromosoma y S el tipo de la solución del problema.

```
interface ValuesInRangeChromosome<E,S> extends
       ChromosomeData<List<E>,S> {}
     E max(Integer i);
     E min(Integer i);
}
```

Restricciones:

- *d = decode();*
- *n = decode.size();*
- *min(i)<= d[i] < max(i), i:0..n-1*;

Binary

Este cromosoma es un caso particular del anterior donde los valores son enteros de mínimo 0 y máximo 1. Es un cromosoma básico a partir del cual se pueden construir otros más complejos.

Usos:

Este cromosoma es adecuado para modelar un amplio abanico de situaciones. Puede ser considerado el cromosoma básico a partir del cual construir otros. Con él podemos modelar todos los problemas de *Programación Lineal Entera* que tengan variables binarias.

Range

Es un subtipo de *ValuesInRangeChromosome<Integer,S>* cuyos valores son listas de enteros en los rangos especificados.

Usos:

Es un cromosoma adecuado para resolver problemas cuya solución es un *Multiset* formado con elementos de un conjunto dado u otros problemas en los que aparecen variables enteras en un rango.

Real

Es un subtipo de *ValuesInRangeChromosome<Double,S>.* Los valores de estos cromosomas son listas de números reales en rangos especificados.

Este cromosoma es adecuado para modelar funciones de varias variables reales de varias variables de las que se quiere obtener el máximo o el mínimo en un dominio especificado mediante rangos.

Cromosoma con secuencia normal

Los cromosoma con secuencia normal, SeqNomalChromosome, tienen valores que son listas de *Integer* que se pueden construir a partir de permutaciones de una *secuencia normal*. Esta secuencia, de tamaño size(), se puede proporcionar como dato del problema.

Por defecto se proporciona una secuencia normal formada por los enteros en $[0, n)$ donde cada entero i se repite $m(i)$ veces.

```
public interface SeqNormalData<S> extends
              ChromosomeData<List<Integer>,S> {
       Integer itemsNumber();
       default Integer maxMultiplicity(int index){ return 1; }
       default List<Integer> normalSequence() { … }
       default Integer size() { return
              normalSequence().size();}
       }
}
```

Restricciones y notación:

- *n = itemsNumber();*
- *s = normalSequence().size();*
- *d = decode();*
- *r = d.size();*
- *n<=r<=s;*
- *m(i) = maxMultiplicity (i);*

Cada cromosoma de este tipo tiene asociada la *secuencia normal*. La longitud de la secuencia normal es $\sum_{i=}^{n-1} m_i$.

Dispondremos de varios subtipos:

- Permutation
- PermutationSubList.
- Sublist

Permutation

Es un subtipo de *SeqChromosome* cuyos valores son listas de enteros que son permutaciones de la *secuencia normal*. Es un cromosoma adecuado para resolver problemas cuya solución es una permutación de un multiconjunto dado de objetos.

PermutationSubList

Es un subtipo de *SeqChromosome* cuyos valores son listas de enteros que son permutaciones de subconjuntos de la secuencia normal. Es un cromosoma adecuado para resolver problemas cuya solución es una permutación de un subconjunto de un multiconjunto dado de objetos.

SubList

Es un subtipo de *SeqChromosome* cuyos valores son listas de enteros que son subconjuntos sin permutar de la secuencia normal.

Otros cromosomas disponibles

ValuesInSetData<S>

Es un cromosoma cuyos valores son listas de enteros que cuyos valores en cada casilla están en un conjunto especificado

```
public interface ValuesInSetData<S> extends
             ChromosomeData<List<Integer>,S> {
      List<Integer> values(Integer i);
}
```

BlocksData<S>

Es cromosoma cuyos valores son listas de enteros divididas en bloques. Dentro de cada bloque aparecerán permutaciones de los valores especificados previamente para ese bloque. Cada bloque viene especificado por los límites de ese bloque y los valores que pueden aparecer en ese bloque.

```
public interface BlocksData<S> extends
              ChromosomeData<List<Integer>,S> {
       List<Integer> blocksLimits();
       List<Integer> initialValues();
}
```

ExpressionData

Es un cromosoma cuyos valores son expresiones construidas a partir de un conjunto dado de operadores. Para concretar los detalles del cromosoma debemos establecer el número de variables y de constantes máximo, el rango y tipo de las constantes y el operador de combinación de las expresiones parciales.

Para flexibilizar el tipo del cromosoma podemos especificar la longitud de la cabeza del cromosoma y el número de genes. Ejemplos de uso pueden verse en el repositorio. Para más detalles puede consultarse en la literatura sobre *Gene Expression Programming*. Al final del libro se incluyen referencias sobre el tema.

El tipo ExpressionData es:

```
public interface ExpressionData extends
              ChromosomeData<Exp,Exp>{
       Integer headLength();
       Integer numGenes();
       Integer numVariables();
       Integer numConstants();
       Integer maxValueConstant();
       Type constType();
       List<Operator> operators();
       Operator.Nary nAryOperator();
       default Integer maxArity() {
              return operators().stream()
                     .mapToInt(x ->x.id().arity())
                     .max().getAsInt();
       }
       default Integer tailLength() {
              return headLength() * (maxArity() - 1) + 1;
       }
       default Integer numItemsPorGen() {
              return headLength() + tailLength();
       }
       default Integer size() {
              return numItemsPorGen()*numGenes() +
                     numConstants();
       }
}
```

Ejemplos

Problema de la mochila

Se parte de L, una lista de objetos de tamaño n, donde cada objeto ob_i es de la forma $ob_i = (w_i, v_i, m_i)$ que representan, respectivamente, su peso unitario, su valor unitario y el número máximo de unidades posibles. La mochila tiene una capacidad C. El problema busca ubicar en la mochila el máximo valor posible de los objetos colocados en la mochila siempre que no superen la capacidad.

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

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

$$x_i \leq m_i, \quad i \in [0, n-1]$$

$$int \quad x_i, \quad i \in [0, n-1]$$

Escogiendo un cromosoma *Range*

$$ft = \sum_{i=0}^{n-1} d[i] v_i - K \;\; dle(\sum_{i=0}^{n-1} d[i]\, w_i - C)$$

Donde d es el vector de *decode,* que representa el valor devuelto por el cromosoma, y *dle* la función de distancia a la restricción menor o igual.

Problema de la asignación

En este problema tenemos una lista de agentes L y una lista de tareas T ambas del mismo tamaño n. El coste de que el agente i realice la tarea j sea c_{ij}. Se pretende asignar a cada agente una tarea y sólo una de tal forma que se ejecuten todas las tareas con el coste mínimo.

Una primera versión del problema es hacer una implementación de la solución propuesta más arriba mediante la técnica de la *Programación Lineal Entera*. En la solución asumimos las variables binarias x_{ij} toman

valor 1 si el agente i ejecuta la tarea j y cero si no la ejecuta. Decimos que hemos codificado el problema mediante las variables binarias x_{ij}. La solución en *Programación Lineal Entera* fue:

$$\min \sum_{i=0,j=0}^{n-1,n-1} x_{ij}\, c_{ij}$$

$$\sum_{j=0}^{n-1} x_{ij} = 1, \quad i \in [0, n-1]$$

$$\sum_{i=0}^{n-1} x_{ij} = 1, \quad j \in [0, n-1]$$

$$bin\, x_{ij}, \quad i \in [0, n-1],\ j \in [0, n-1]$$

Este modelo nos llevaría a usar un cromosoma binario de tamaño n^2 y donde la variable x_{ij} estaría en la casilla $d[n * i + j]$ de valor proporcionado por *decode()*. La función de fitness debe contener, además de la función objetivo, el resto de las restricciones.

$$f = -\sum_{i=0,j=0}^{n-1,n-1} x_{ij}\, c_{ij} - k_1 \sum_{i=0}^{n-1} deq(\sum_{j=0}^{n-1} x_{ij} - 1) - k_2 \sum_{j=0}^{n-1} deq(\sum_{i=0}^{n-1} x_{ij} - 1)$$

Donde *deq* es la distancia a la restricción de igualdad.

Otra posibilidad es modelar el problema con las variables enteras x_i. Donde x_i es la tarea asignada al agente i:

$$min \sum_{i=0}^{n-1} c(i, x_i)$$

$$P_{i=0}^{n-1}(x_i, i)$$

$$int\, x_i, \quad i \in [0, n-1]$$

Lo que indicamos es que la lista $x = [x_0, x_1, \ldots, x_{n-1}]$ es una permutación de $[0,1,\ldots,n-1]$.

Este modelo nos sugiere escoger un cromosoma de permutación de tamaño n. Su vector decodificado será una lista de enteros permutación de la secuencia normal que en este caso es $d = [0,1,2,\dots,n-1]$. Si asumimos que $d[i]$ es la tarea asignada al agente i la función de fitness se puede escribir como:

$$ft(d) = -\sum_{i=0}^{n-1} c(i, d[i])$$

Dado que el cromosoma escogido ya se tiene en cuenta las restricciones del modelo.

¿Cuál de las dos formas es mejor? Claramente la primera si disponemos de un buen cromosoma de permutación. Veamos algunos criterios para escoger un tipo de cromosoma frente a otro.

En primer lugar la función de fitness es más sencilla porque los valores del cromosoma son válidos y por lo tanto no tenemos que incluir distancias a restricciones.

Pero fundamentalmente porque el porcentaje de soluciones no válidas (que no cumplen las restricciones) es muy alto en el segundo caso. En efecto en el segundo caso el número total variables es n^2 y de valores posibles del vector *decode* es 2^{n^2}. Dadas las $2n$ restricciones de igualdad el número de variables libres es $n^2 - 2n$ y el total de valores válidos, que cumplen las restricciones, es 2^{n^2-2n}. El cociente nos da la probabilidad de encontrar un valor válido, no el óptimo, al azar y es:

$$\frac{2^{n^2-2n}}{2^{n^2}} = \frac{2^{n^2}}{2^{2n}2^{n^2}} = \frac{1}{2^{2n}}$$

La probabilidad de encontrar una solución no válida es por tanto:

$$1 - \frac{1}{2^{2n}} = \frac{2^{2n} - 1}{2^{2n}}$$

La probabilidad de encontrar una solución no válida es muy alta. Este valor nos da una medida de la calidad del cromosoma.

En el segundo caso todos los valores del cromosoma son válidos.

Problema de las estaciones de bomberos

Una ciudad se compone de n barrios. Cada barrio es vecino de otros barrios y la relación de vecindad se puede representar mediante un grafo no dirigido cuyos vértices representan los barrios y existe una arista entre dos barrios si son vecinos. Queremos ubicar una estación de bomberos en algunos barrios con la restricción que en cada barrio o en uno de sus vecinos haya una estación de bomberos. El objetivo es minimizar el número de estaciones de bomberos. Si las variables binarias x_j indican si un barrio tendrá estación de bomberos o no y $N(j)$ el conjunto de barrios vecinos al j incluido el mismo. El problema de Programación Lineal Entera era:

$$\min \sum_{j=0}^{n-1} x_j$$
$$(\sum_{i:N(j)} x_i) \geq 1, \quad j \in [0, n)$$
$$bin\ x_j, \quad j \in [0, n)$$

Este modelo nos llevaría a un cromosoma binario de tamaño n, con n restricciones y función de fitness:

$$ft = -\sum_{j=0}^{n-1} x_j - K \sum_{j=0}^{n-1} dge\,(\sum_{i:N(j)} x_i - 1)$$

Un modelo alternativo sería:

$$min \sum_{j=0}^{m-1} x_j$$
$$|\cup_{j=0|x_j=1}^{n-1} V(j)| = n$$
$$bin\ x_j, \quad j \in [0, n)$$

Donde $V(j)$ es un conjunto que incluye los barrios vecinos a j incluyendo él mismo, Ahora la única restricción indica que la unión de todos los conjuntos $V(j)$ es igual al total de los barrios. El modelo no es lineal, pero tiene menos restricciones. La función de fitness queda en la forma:

$$f = -\sum_{j=0}^{n-1} x_j - K\ deq(|\cup_{j=0|p}^{n-1} V(j)| - n)$$

Problema de los anuncios

Un canal de televisión quiere obtener el máximo rendimiento (en euros) de la secuencia de anuncios que aparecerá en la cadena después de las campanadas de fin de año. La secuencia de anuncios del año durará T segundos como máximo. Hay una lista L, de tamaño n, de anuncios que se ofertan para ser emitidos. Cada anuncio $a(i)$ tiene un tiempo de duración $t(i)$ y está dispuesto a pagar un precio $p(i) = \frac{b\ t(i)}{p(i)} + c$. Dónde b, c son constantes y $p(i)$ es la posición en la que se emitirá el anuncio, comenzando en 1, si llega a ser emitido. Se quiere emitir el subconjunto de L cuyo tiempo total de emisión sea menor o igual que T y maximice el precio total de los anuncios.

Un modelo del problema donde x_i indica el anuncio que se emitirá en la posición i es:

$$max \sum_{i=1}^{s} (\frac{b\ t(x_i)}{i} + c)$$
$$s \leq n$$
$$\sum_{i=1}^{s} t(x_i) \leq T$$
$$P_{i=1}^{n}(x_i, i)$$
$$int\ x_i,\ \ i \in [1, n]$$
$$int\ s$$

Haciendo que la variable x_i esté en la casilla $d[i-1]$ podemos escoger un cromosoma de permutación con la función de fitness se escribe como:

$$ft(d) = \sum_{i=0}^{s-1} (\frac{b\ t(d[i])}{i+1} + c)$$

Donde s es el primer valor dónde la expresión siguiente se hace positiva.

$$\sum_{i=0}^{s} t(d[i]) - T$$

Como se puede comprobar el cálculo del fitness y la condición sobre s se pueden implementar en un solo bucle que recorre el vector d.

Alternativamente se podría usar un cromosoma *PermutationSublist* cuyos valores serán los índices de los anuncios escogidos. Se deja esta idea como ejercicio.

Algoritmos de Simulated Annealing

De forma similar a lo algoritmos genéticos para resolver un problema es posible resolver problemas mediante *Simulated Annealing.* Como en los algoritmos genéticos el modelo debe tener la forma:

$$\begin{array}{c} min\ \ f(v) \\ v \in \mathrm{V} \end{array}$$

Representemos por e el estado del algoritmo, por $m(e)$ el resultado de aplicarle un operador de mutación y por $e' = m(e)$.

Sean f, f' los valores de la función objetivo para los estados e, e' y $\Delta = f' - f$ el incremento. Entonces para un incremento dado se acepta el nuevo estado con probabilidad

$$pa(\Delta) = \begin{cases} 1, & \Delta \leq 0 \\ e * e^{-\Delta/T}, & \Delta > 0 \end{cases}$$

Es decir, se acepta con total seguridad si el incremento es negativo (estamos minimizando) y con la probabilidad indicada si es positivo. La probabilidad depende de un concepto llamado temperatura. La temperatura, que intenta emular la temperatura de un sistema cuando se va enfriando, toma un valor inicial y posteriormente va disminuyendo hasta acercarse a cero. Una cuestión clave es la estrategia de enfriamiento.

Es decir, el mecanismo de disminución de la temperatura. Se han propuesto varias alternativas. Aquí, en un primer momento, escogeremos

$$T = T_0\,\alpha^i, \qquad 0 < \alpha < 1$$

Dónde α es un parámetro en el intervalo [0,1), i el número de iteración y T_0 la temperatura inicial. La teemperatura T va decreciendo desde el valor T_0 con un ritmo marcado por el parámetro α. Con esta elección del enfriamiento la probabilidad es:

$$pa(\Delta, i) = \begin{cases} 1, & \Delta \leq 0 \\ e^{1-\Delta/T_0\,\alpha^i}, & \Delta > 0 \end{cases}$$

Será necesario establecer n el número de iteraciones. El algoritmo comienza a dar pasos desde $i = 0$ y en las primeras iteraciones acepta incrementos positivos con probabilidad $e^{1-\Delta/T_0\,\alpha^i}$. En el paso i acepta incrementos positivos con probabilidad $e^{1-\Delta/T_0\alpha^i}$. Para ajustar el algoritmo debemos escoger los parámetros anteriores. Buscamos el valor de la ratio Δ/T_0 y del parámetro α en función de p_0, p_f, n. Para conseguir que la función $pa(\Delta, i)$ sea continua en $\Delta = 0$ hacemos $\Delta/T_0 = 1$. Por lo tanto $T_0 = \Delta/T_0$. La probabilidad de aceptación al principio será muy alta y al final p_f (en el paso n, 0.01 por ejemplo). A partir de lo anterior vemos que debe cumplirse

$$e^{1-\alpha^{-n}} = p_f$$

Despejando en la primera ecuación tenemos

$$\alpha = \sqrt[n]{\frac{1}{1 - \ln p_f}}$$

Si $p_f = 0.01,\ n \approx 300$ tenemos $\alpha \approx \sqrt[300]{0.178} \approx 0.994$.

O alternativamente

Si $p_f = 0.01, n \approx 200$ tenemos $\alpha \approx \sqrt[200]{0.178} \approx 0.991$.

En muchos casos es más interesante fijar α y calcular n para que se cumpla la condición anterior: $n = \frac{\ln 0.178}{\ln \alpha}$. Para $\alpha = 0.95$ tenemos $n = 34$.

Escogidos esos parámetros el tiempo de ejecución del algoritmo es proporcional a n. Junto a la anterior expresión para la evolución de la temperatura hay muchas otras posibles. Una de ellas, también bastante común es

$$T = \frac{T_0}{1 + \ln(1 + \alpha i)}$$

Donde i es el número de la iteración y T_0 la temperatura inicial. Como antes habrá que calcular los valores adecuados de T_0, α, n.

Haciendo de nuevo $\Delta/T_0 = 1$ ahora tenemos

$$e^{-ln(1+\alpha n)} = 1/(1 + \alpha n) = p_f, n = \frac{1 - p_f}{\alpha p_f} \approx \frac{99}{\alpha}$$

Operadores de mutación

Para implementar el algoritmo debemos disponer de un objeto que implemente:

```
public interface StateSa {
	Double fitness();
	StateSa mutate();
	StateSa random();
	StateSa copy();
}
```

Una primera posibilidad es usar los cromosomas disponibles para los algoritmos genéticos, escoger un cromosoma cuyo espacio de valores sea adecuado para el problema a resolver, usar los operadores de mutación para ese cromosoma, la función de decodificación y la función de fitness definida (cambiada de signo puesto que en los algoritmos genéticos maximizamos). En el repositorio hay un adaptador para concretar esta posibilidad: *StateSaChromosome.*

Con la estrategia anterior en *Simulated Annealing* podríamos usar los mismos cromosomas, pero usando sólo los operadores de mutación y descartando los de cruce y selección. También podemos diseñar nuevos tipos para cada caso concreto que implementen *StateSa.*

El algoritmo y ejemplos pueden verse en el repositorio.

Problemas y grafos

En este capítulo vamos a ver varias técnicas generales para resolver problemas basándose en grafos: algoritmos *A**, *Backtracking*, *Programación Dinámica*, *Algoritmos Voraces (Greedy)*, *Búsqueda Local (Local Search)* y *SimulatedAnnealing.*

En todas ellas partimos de un modelo de un problema. Es decir, unas variables que toman valores en un dominio, una función objetivo y unas restricciones.

$$\max_{x}/\min f(x)$$
$$c(x)$$
$$x \in \Omega$$

Para aplicar estas técnicas debemos, en primer lugar, imaginar un conjunto de problemas o un conjunto de estados y relaciones entre ellos que forman un *grafo virtual*. El conjunto de problemas puede ser evidente o debemos imaginarlo generalizando el problema, es decir, añadiendo propiedades. Los problemas serán los vértices del grafo y las relaciones entre ellos las aristas que contendrán información relevante. Los grafos que veremos primero tendrán aristas que conectarán un problema con otro. Más adelante veremos problemas que tienen asociados *hipergrafos* que son grafos donde las aristas conectan un vértice (un problema) con varios de ellos.

Grafos virtuales

Introducción: espacio de estados y grafos virtuales

Un *grafo virtual o implícito* es un tipo de grafo donde los vértices y aristas se definen mediante conjuntos definidos por comprensión. Es decir, mediante un tipo y un predicado que define los valores válidos. Un grafo virtual *g* de tipo *Graph<V,E>* lo podemos definir como $g = (sv, se)$. Donde sv, se son el conjunto de sus vértices y el de sus aristas. Cada uno de estos conjuntos tendrán asociado un tipo de datos y un predicado:

$$sv = \{v{:}\, V \mid valid(v)\}$$

$$se = \{v1{:}\, sv, v2{:}\, sv \mid neighbor(v1, v2)\}$$

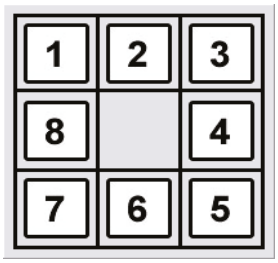
El predicado *valid(v)* indica si *v* es un vértice válido del tipo V y *neighbor(v1, v2)* si la arista *(v1, v2)* que une los vértices válidos *v1, v2* es válida. Las aristas, en general, tendrán asociadas acciones adecuadas para pasar de uno a otro de los vértices que conectan.

Usualmente los grafos virtuales que usaremos serán grafos dirigidos.

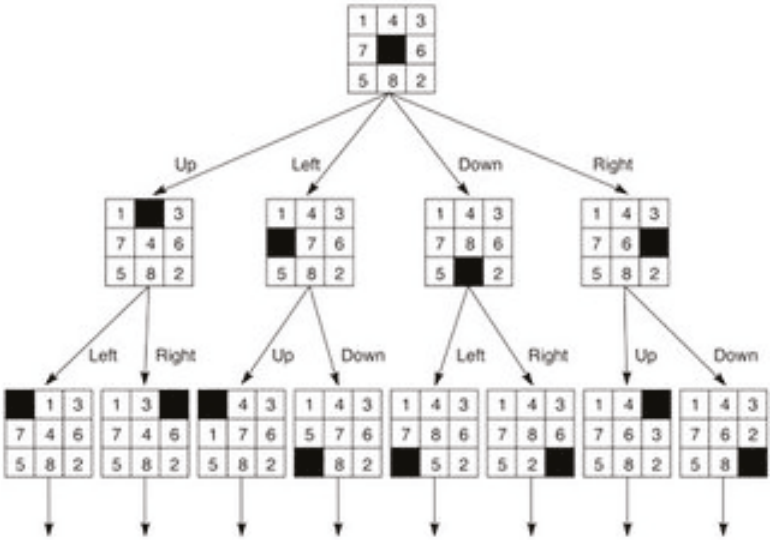
Problema del puzle

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, hacia la izquierda o derecha, o verticales, hacia arriba o hacia abajo. El problema consiste en dada una configuración inicial llegar a una configuración final (meta) mediante los movimientos permitidos y en el mínimo número de pasos posible.

El problema puede ser modelado mediante un grafo virtual del tipo anterior. Los vértices del grafo, que concretaremos en la clase *Vp* (*VerticePuzle)* que representan cada una de las posibles distribuciones de los *9* números, incluyendo la casilla vacía como 0, en un tablero de 3 filas y tres columnas. Un vértice en particular viene representando en el gráfico:



Hay cuatro posibles acciones o movimientos: *Arriba (Up), Izquierda (Left), Abajo (Down), Derecha (Right).* Las aristas serán del tipo *Ep* (*AristaPuzzle)*. Algunos vértices y aristas vienen representados en el gráfico siguiente.



Los valores del tipo *Vp* serán todas las configuraciones posibles del puzle. Podemos designar cada casilla por una tupla (f, c) que indique su fila y columna. Sean $p = (f, c)$ los valores de esa tupla que representamos por el tipo *IntPair* y $P = \{(f, c): f \in [0, n), c \in [0, n)\}$ el conjunto de todas las casillas. Este tipo se puede implementar como un *record* de Java.

Para modelar los tipos seguiremos las ideas de la programación orientada a objetos. Desde este punto de vista un tipo se define como un conjunto de propiedades de tipos conocidos, algunas de las cuales se concretan en métodos, y un conjunto de restricciones sobre los valores de esas propiedades. Un tipo tendrá, además, métodos de factoría, parsing, etc. para ver más detalles sobre el diseño de tipos veanse los libros de *Fundamentos de Programación* en esta misma colección.

Las propiedades del tipo *IntPair* podrían ser

Propiedades de *IntPair*

- Integer f: Fila
- Integer c: Columna

Y las propiedades de *Vp*

- *Integer n:* Número de filas
- $IntPair\ bp()$: Una tupla *(f,c)* con la posición de la casilla negra
- $Integer\ get(p),\ p \in P$*:* Valor entero contenido en la casilla p definida por la fila f y la columna c.

Un vértice es válido si tiene todos los valores de sus casillas distintos y comprendidos entre 0 y $n^2 - 1$.

- $Boolean\ isValid() \equiv AD_{p:P}\ get(p),\ \forall_{p:P}\ 0 \leq get(p) < n^2$

En los grafos virtuales cada arista suele tiene asociada un acción $a(v1, v2)$ y un peso $w(v1, v2)$. En el ejemplo del *Puzle* $w(v1, v2) = 1$ y las acciones posibles son las del conjunto $Ap = \{Up, Left, Down, Right\}$. Solo un subconjunto $Ap(v)$ de Ap nos dará las acciones permitidas en un vértice concreto v.

El método *Vp neighbor(A a)* nos indica el vecino que se alcanza si tomamos la acción a. Para definirlo asociemos a cada acción una dirección de movimiento. Sean estas direcciones los pares de enteros de la lista:

$$d = [(-1,0), (0,-1), (1,0), (0,1)]$$

En estos pares hemos asumido que la casilla (0,0) es la superior izquierda y que las filas aumentan hacia abajo. Con estos elementos el método que nos da el vecino al tomar una acción lo definimos como:

$$v.neighbor(a) = v.swap(v.bp(), v.bp + d[a])$$

Asumamos que el método $swap(p1, p2)$ produce una nueva configuración tras intercambiar las casillas $p1, p2$ y el operador $+$ suma los dos pares correspondientes. Para que el vecino obtenido sea válido not todas las

acciones son posibles. Podemos definir el conjunto $Ap(v)$ de acciones posibles en v de la forma:

$$Ap(v) = \{a: Ap |\ v.neighbor(a).isValid()\}$$

Dos vértices v y $v1$ son vecinos si existe una acción válida en v que nos lleve hasta *v1*. Así el predicado *isNeighbor* podemo definirlo como:

$$v.isNeighbor(Vp\ v1) \equiv \exists_{a:Ap(v)} v.neighbor(a) = v1\ .$$

A partir del tipo *Vp* y los predicados *isValid*, *isNeighbor* podemos definir por comprensión el grafo *g* de tipo *Graph<Vp,DefaultEdge>.*

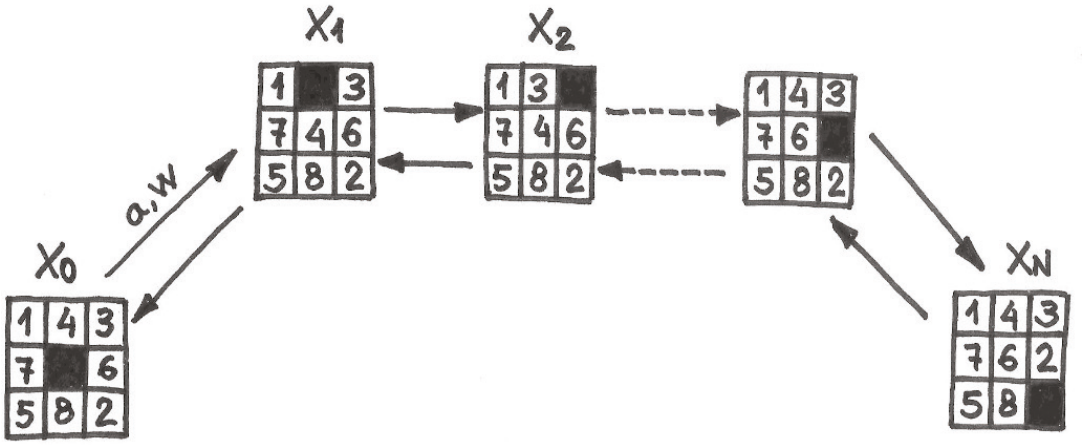
$$g = (sv, se)$$
$$sv = \{v: Vp \mid v.isValid()\}$$
$$se = \{v1: sv, v2: sv \mid v1.isNeighbor(v2)\}$$

A partir del grafo *g* nuestro problema es un problema de camino mínimo en ese grafo. Con esos tipos ya podemos diseñar un modelo:

$$\min r$$
$$x_0 = v_0$$
$$x_{r-1} = v_1$$
$$OP_{i=0|g}^{r-1} x_i$$
$$int\ r$$
$$Vp\ x_i, i \in [0, r-1]$$

El modelo define un grafo virtual (por comprensión) y especifica que debemos buscar el camino (simple y abierto) con extremos v_0, v_1 y el mínimo número de vértices. La variables son x_i son del tipo *Vp* que hemos definido previamente para representar las diferentes configuraciones del puzle. El número de variables es r que a su vez es variable. El camino vendrá dado por la lista $x = [x_0, x_1, \dots, x_{r-1}]$ de vértices que cumplen la restricción $OP_{i=0|g}^{r-1} x_i$.

Este sería una imagen de un camino en el grafo:



Para implementar un grafo virtual instanciamos los tipos *V, E, A* (vértices, aristas y alternativas). En este caso escogemos *V = Vp, E = SimpleEdge<Vp>, A = Ap.* Donde el tipo *SimpleEdge<Vp>* tiene como propiedades sus vértices extremos, su peso y la acción asociada.

El tipo V de los vértices extenderá *VirtualVertex<V,E,A>*:

```
interface VirtualVertex<V,E,A>{
       Public Boolean isValid();
       Public List<A> actions();
       Public V neighbor(A a);
       public E edge(A a);
       …
}
```

Donde *isValid()* representa el predicado comentado arriba, *actions()* proporciona los valores del conjunto $A(v)$ ordenados en una lista, *neighbor(A a)* la función comentada y *edge(A a)* la arista $(this, this.neighbor(a))$.

El tipo *E* de las aristas implementa el interface *SimpleEdgeAction<V,A>.*

```
interface SimpleEdgeAction<V, A> extends SimpleEdge<V>{
       public A action();
}
```

```
interface SimpleEdge<V> {
	V source();
	V target();
	Double weight();
}
```

Para disponer de un grafo virtual, debemos implementar, por lo tanto, el tipo de los vértices, que debe ser un subtipo de *VirtualVertex<V,E,A>* donde *V* es el tipo del vértice que estamos implementando, *E* el tipo de las aristas y *A* el tipo de las acciones.

A, el tipo de las acciones, puede escogerse entre un tipo ya conocido *(Integer, Boolean, …)* o diseñar un tipo nuevo. En el caso del Puzle el tipo *Ap* es un tipo enumerado.

Implementados los tipos anteriores la clase *SimpleVirtualGraph* implementa un grafo virtual que será del tipo *Egraph<V,E>* que extiende el tipo *Graph<V,E>*.

El tipo *EGraph<V, E>* es un grafo extendido que extiende el tipo *Graph<V,E>* con la información sobre el vértice inicial y los vértices finales, etc.

```
public interface EGraph<V, E> extends Graph<V,E> {
	V startVertex();
	Predicate<V> goal();
	…
}
```

Para crear una instancia del grafo extendido disponemos de un builder que podemos usar en la forma:

```
EGraph<VertexPuzzle, EdgePuzzle> graph =
	EGraph.virtual(start,x->x.equals(end))
		.edgeWeight(x->x.weight())
		.endVertex(end)
		.heuristic((v1,p,v2)->0.)
		.build();
```

Con el mecanismo del builder podemos concretar todas las propiedades del grafo virtual extendido. Las propiedades no especificadas tendrán un valor por defecto.

Builder es un patrón de diseño creacional que nos permite construir objetos con muchas propiedades de una manera flexible indicando sólo las propiedades relevantes y manteniendo valores por defecto para las no relevantes. El builder ofrece un método para indicar el valor de cada propiedad relevante y finalmente construye un objeto del tipo especificado. El código puede encontrarse en el repositorio. Si no usaramos el builder necesitaríamos un método de factoría con muchos parámetros o muchos metodos de factoría sobrecargados lo que hace ilegible el código para construir el grafo virtual extendido.

Una vez concretado el grafo extendido nuestro objetivo será encontrar caminos desde el vértice inicial a uno de los vértices finales. Estos caminos definirán las soluciones que estamos buscando para un problema. El camino de peso mínimo, o máximo según el caso, definirá la solución óptima.

Los caminos son secuencias de vértices y aristas que se representan por el tipo *GraphPath<V,E>* que tiene las propiedades:

```
interface GraphPath<V,E>
       List<E> getEdgeList()
       V getEndVertex()
       Graph<V,E> getGraph()
       Int getLength()
       V getStartVertex()
       List<V> getVertexList()
       double getWeight()
};
```

Veamos la concreción del problema del Puzzle anterior. Sea el tipo para los vértices *VertexPuzzle*, el tipo para las aristas *EdgePuzzle* y el tipo para las acciones *ActionPuzzle.*

El tipo *ActionPuzzle* es un tipo enumerado con un método que asocia una dirección a cada acción.

```
public enum ActionPuzzle {
	Up,Left,Down,Right;

	public static IntPair direction(ActionPuzzle a) {
		return switch(a) {
		case Up -> IntPair.of(-1,0);
		case Down -> IntPair.of(1,0);
		case Left -> IntPair.of(0,-1);
		case Right -> IntPair.of(0,1);
		};
	}
}
```

En este caso cada acción, como se definió en el modelo, tiene un nombre y una dirección de movimiento de la casilla negra asociada.

El tipo de las aristas es fácil de implementar

```
record EdgePuzzle(VertexPuzzle source, VertexPuzzle target,
	ActionPuzzle action, Double weight) implements
		ActionSimpleEdge<VertexPuzzle,ActionPuzzle> {
	public static EdgePuzzle of(VertexPuzzle v1,
		VertexPuzzle v2,
			ActionPuzzle a) {
		return new EdgePuzzle(v1, v2, a, 1.);
	}
}
```

El grueso de la programación está en la implementación del *VertexPuzzle*, el tipo del vértice. Como vemos el tipo tiene dos propiedades básicas: *blackPosition*, la posición de la casilla negra y *datos*, los datos en las casillas. A partir de ella debemos implementar el resto de las propiedades del vértice: *actions(), neighbor(a), edge(a) y isValid().*

```
record VertexPuzzle(IntPair blackPosition,Integer[][] datos)
        implements ActionVirtualVertex<VertexPuzzle,
              EdgePuzzle, ActionPuzzle>{
       @Override
       public VertexPuzzle neighbor(ActionPuzzle a)
              return this.swap(this.blackPosition()
              .add(ActionPuzzle.direction(a)));
       }
       @Override
       public List<ActionPuzzle> actions() {
              return Stream.of(ActionPuzzle.values())
              .filter(a->VertexPuzzle.validPosition(
                     this.blackPosition()
                           .add(ActionPuzzle.direction(a))))
                                  .collect(Collectors.toList());
       }
       @Override
       public EdgePuzzle edge(ActionPuzzle a) {
              return  EdgePuzzle.of(this, this.neighbor(a), a);
       }
       @Override
       public Boolean isValid() {
              return validData(this.datos()) &&
                     getDato(blackPosition()) == 0;
       }
```

```
       private static Boolean validData(Integer[][] datos) {
              Set<Integer> s = Arrays.stream(datos)
              .flatMap(f->Arrays.stream(f))
              .filter(e->VertexPuzzle.validDato(e))
              .collect(Collectors.toSet());
              return s.size()== n*n;
       }
```

```
       private static boolean validDato(Integer d) {
              return 0<=d && d < VertexPuzzle.numFilas*
                     VertexPuzzle.numFilas;
       }
       …
}
```

La propiedad *actions()* nos da las acciones disponibles en ese vértice. La propiedad *neighbor(a)* obtiene el nuevo vértice tras aplicar la acción *a* que se supone está disponible en el sentido de que el nuevo vértice es válido. El vecino se obtiene calculando la nueva posición de la casilla negra e intercambiando el contenido de esta casilla con el de la antigua casilla

negra. El método *isValid()* indica si el valor es un valor válido del tipo y el método *edge(a)* nos da la arista asociada a la aplicación de la acción *a*.

De modelos a grafos extendidos

Veamos ahora problemas cuya solución está relacionada con el camino mínimo, entendido como camino de peso mínimo, en un grafo. En el caso del *Puzle* la solución del problema es una lista de vértices que forman un camino mínimo entre el vértice de partida y el de llegada. En otros casos la relación entre la solución y el camino es más compleja.

En general partimos de un modelo de un problema y pretendemos buscar un grafo extendido cuyos caminos entre el vértice inicial y alguno de los finales estén asociados a las soluciones del problema definidas por el modelo.

El primer paso es imaginar las propiedades del tipo V de los vértices del grafo y el predicado que define los valores válidos de ese tipo. Esta tarea, que llamaremos *generalización*, es similar a la que llevamos a cabo cuando realizamos un diseño recursivo. En aquel caso buscábamos un conjunto de problemas, ahora un conjunto de vértices. En ambos casos se trata de imaginar propiedades de un tipo. En este caso los vértices van a ser estados de un camino que progresa hasta el vértice final. El modelo nos va a sugerir valores que deben cumplir ciertas restricciones y por lo tanto debemos incluirlos como propiedades de los vértices. Estas propiedades serán útiles para definir los vértices válidos.

El primer paso es establecer las propiedades de los vértices a partir de las variables del modelo y sus restricciones. Los valores que toman las variables del modelo dan una pista sobre el tipo y los valores de las acciones. El peso de las aristas vendrá definido por la función objetivo. No olvidemos que el peso de un camino, la suma del peso de sus aristas, debe ser el definido por la función objetivo.

Junto a lo anterior tenemos los siguientes elementos:

Vértice Inicial (startVertex): El vértice del grafo desde donde partimos. Está asociado al problema inicial.

Casos Base: Son vértices de los cuales ya sabemos la solución o sabemos que no existe. Estos casos bases es conveniente caracterizarlos: saber si

tienen solución y, si la tienen, cuál es su valor. Los casos base se convertirán en vértices con un único vecino que es un vértice final.

Vértices Finales (goal): El vértice o vértices del grafo donde queremos llegar. Puede ser un problema o un conjunto de vértices definidos por un predicado. Puede ser un problema existente o uno que imaginamos para este fin. Establecemos que si los casos base tienen solución tendrán un único vecino que será un *vértice final*. La arista hacia él tendrá un peso igual al valor de la función objetivo de la solución del caso base y la acción que corresponda. Si el caso base no tiene solución no tiene vecinos.

La decisión de escoger un conjunto de vértices finales en los cuales los valores de la función objetivo están fijados y añadir una única arista desde los casos base hasta loe vértices finales simplifica, creemos, el esquema global.

Restricción sobre el vértice final (goalHasSolution): Un predicado que indica si el vértice final indicado tiene solución o no. En general es posible hacer que los vértices finales sean todos válidos filtrando las transiciones en los vértices previos a ellos, pero en algunos casos esto puede complicar el código. Por defecto esta restricción devuelve *true.*

Solución de un problema final (goalSolution): Una función que indica el valor de la función objetivo asociada a un vértice final si ese valor existe. Un vértice final tiene solución si la restricción *goalHasSolution* es *true* en ese vértice y si no tiene el valor de la solución será null. Si existe solución en los vértices finales establecemos entonces que los valores de la función objetivo en esos vértices finales será 0, en los grafos con caminos de tipo *Sum* y el valor de la propiedad escogida del último vértice en los de tipo *Last.*

Peso de una arista: Las funciones objetivo que son de la forma $\sum_{i:C} w_i$, donde C es un camino, son adecuadas para asociar el peso w_i a una arista. En general ese peso dependerá de las propiedades de la arista: vértice fuente, vértice destino y acción asociada. Para funciones objetivo de otro tipo puede que no sea posible asociar un peso a la arista. Este caso lo veremos más adelante. Por defecto asumimos que el peso de una arista es $w_i = 1$.

Peso del camino: Un camino en un grafo es una variable de tipo *GraphPath<V,E>.* En muchos casos el peso del camino es la suma de los

pesos de las aristas. Esta definición del peso del camino puede deducirse generalmente de la función objetivo del modelo del problema. Veremos ejemplos más adelante. Pero hay muchos casos en los que el peso del camino se debe definir de otra forma dependiendo del problema en cuestión. Estos casos los veremos más adelante.

Camino Solución: Las soluciones que buscamos en este tipo de problemas son caminos en el grafo diseñado que van desde el vértice inicial a uno final que cumpla la restricción establecida. Cada camino tiene un peso. La solución óptima es el camino de peso mínimo (o máximo).

Solución: En la mayoría de los problemas queremos que la solución sea más específica que un camino. Es decir, queremos que sea de un tipo *S* dado. Para ello, debemos implementar un método que transforme un *GraphPath<V,E>* al tipo *S* indicado.

La mayoría de esta información se incorpora en el grafo extendido, el tipo *Egraph<V,E>* que veremos más adelante. Algunas de sus propiedades son:

```
public interface EGraph<V, E> extends Graph<V,E> {
       V startVertex();
       Predicate<V> goal();
       V endVertex();
       Predicate<V> goalHasSolution();
       PathType pathType();
       …
}
```

Donde *startVertex* es el vértice inicial, *endVertex* un vértice final, *goal* el predicado que especifica los vértices finales, *goalHasSolution* el predicado que indica si el vértice final tiene solución o no, *edgeWeight* el peso de la arista y *pathType* el tipo del camino (esta propiedad indica la forma de calcular el peso del camino).

La clase *SimpleVirtualGraph* implementa un grafo extendido.

Con los detalles anteriores el grafo asociado al problema del *puzle* visto arriba es:

```
VertexPuzzle start = VertexPuzzle.of(1,2,3,4,5,0,6,7,8);
VertexPuzzle end = VertexPuzzle.of(1,2,3,4,6,5,8,7,0);
```

```
EGraph<VertexPuzzle, EdgePuzzle> graph =
EGraph.virtual(start,x->x.equals(end),PathType.Sum, Type.Min)
       .edgeWeight(x->x.weight())
       .endVertex(e2)
       .heuristic(HeuristicaPuzzle::heuristica)
       .build();
```

Un grafo no virtual puede ser también convertido en un grafo extendido como veremos más adelante.

Concretado el grafo extendido, real o virtual, debemos usar un algoritmo para obtener el camino mínimo. ¿Qué algoritmo usar? La opción más simple es el algoritmo de *Dijsktra* que se ofrece en Jgrapht.

```
GraphAlg.dijsktra(graph,e1,e2,…);
```

Los algoritmos *Voraces, A*, Backtraking, Programación Dinámica* los veremos en las secciones siguientes y ya están disponibles en el respositorio.

Problema de la mochila

Veamos un ejemplo para ilustrar lo anterior.

El problema de la *mochila,* ya resuelto anteriormente con otras técnicas, parte de una lista de objetos *Lo* de tamaño n. A su vez cada objeto ob_i de la lista es de la forma $ob_i = (w_i, v_i, m_i)$ dónde w_i, v_i, m_i son, respectivamente su peso, su valor unitario y el número máximo de unidades permitidas. Además, la mochila tiene una capacidad C. El problema busca ubicar en la mochila el máximo número unidades de cada objeto que quepan en la mochila para que el valor de estos sea máximo. Si x_i es el número de unidades del objeto i en la mochila el problema puede enunciarse como un problema de Programación Lineal de la forma:

$$\max \sum_{i=0}^{n-1} x_i v_i$$
$$\sum_{i=0}^{n-1} x_i w_i \leq C$$
$$x_i \leq m_i, \quad i \in [0, n-1]$$
$$int \quad x_i, \quad i \in [0, n-1]$$

La cuestión ahora es: ¿podemos buscar un grafo tal que la solución del problema sea un camino mínimo en el mismo? Para responder a esa pregunta, seguimos los mismos pasos que antes:

Vértices. Para encontrar los vértices debemos generalizar el problema y diseñar un tipo adecuado. Generalizamos el problema para obtener un conjunto de problemas que formen los vértices del grafo. La generalización la conseguimos introduciendo dos variables nuevas: i, r. El problema representado por estas variables es solucionar el problema de una mochila de capacidad *r* y objetos desde *i* hasta el final de la lista.

Llamemos a este tipo *MochilaVertex* que hereda de *VirtualVertex<MochilaVertex,MochilaEdge,Integer>* con las propiedades:

Propiedades:

i: Integer. Un índice en la lista de objetos.
r: Integer. La capacidad de la mochila.
t: Integer. Derivada *t = n-i.*
n: Integer. Número de objetos, compartida.
C: Integer. Capacidad Inicial, compartida.

Descripción: El problema generalizado pretende colocar en una mochila de capacidad *r* diversas unidades de objetos que están desde el índice *i* hasta el final de la lista.

Igualdad: Dos problemas son iguales si tiene las mismas *i, r.* Estas propiedades identifican el problema que representaremos por (i, r)

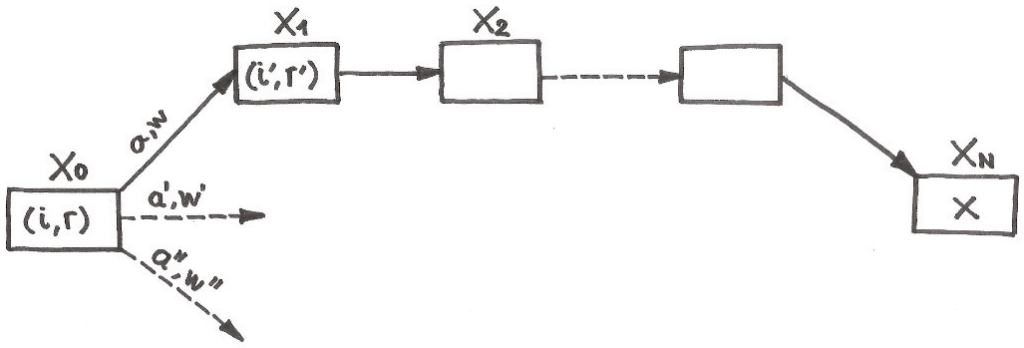
Vértice Válido: $v \rightarrow 0 \leq v.i \leq n, \ v.r \geq 0$

Acciones: Partiendo del problema (i, r) escogemos como acciones el número de unidades que podemos tomar del objeto ob_i. Las acciones

posibles son entonces los enteros $0..m_i$. Pero puede que todas no sean válidas. Asumimos que cuando cogemos a unidades enteras partiendo de un problema (i, r) pasamos a plantearnos el problema $(i+1, r - a * w_i)$. Para que la alternativa sea válida el vértice alcanzado debe ser válido. Debe cumplirse $i + 1 \leq n, r - a * w_i \geq 0$. Las soluciones para la segunda restricción nos llevan a $a \leq \frac{r}{w_i}$ y por otra parte $0 \leq a \leq m_i$. De lo que concluimos que la lista de acciones disponibles (método *actions()*) debe devolver una lista con los enteros $0..\min(m_i, \frac{r}{w_i})$

1. *Vecino (método V neighbor(A a)):* Asumiendo que la acción es válida el vecino de (i, r) siguiendo a será $(i+1, r - a * w_i)$. Observamos que el tamaño del problema destino *(t-1)* es más pequeño que el del problema fuente *(t).*
   Los casos base, vistos más abajo, harán más específicos sus vecinos. El vértice final no tiene vecinos.
2. *Arista al vecino:* Diseñamos el tipo *MochilaEdge*, que hereda de *ActionSimpleEdge<MochilaVertex,Integer>.* Junto a los vértices fuente y destino y la acción, debemos calcular el peso. El peso de la arista es el valor asociado a la acción tomada. La suma de esos valores en un camino solución es la función objetivo. Escogemos el peso $a * v_i$ que al sumarse a lo largo del camino nos da la función objetivo.

Con las consideraciones anteriores ya tenemos el grafo que buscábamos.



El vértice inicial es $(0, C)$ y el final cumple el predicado $v \rightarrow v.i() = n$.

Casos base son problemas de tamaño pequeño cuya solución es conocida. Algunos casos base son:

- $(n-1, r)$: Su único vecino es un problema final con acción $a = \min(m_{n-1}, \frac{r}{w_{n-1}})$ y el peso de la arista: $a * v_{n-1}$
- $(i, 0)$: Su único vecino es el problema final $(n, 0)$ con acción $a = 0$ y el peso de la arista 0.

Algunos detalles de la implementación de los tipos *MochilaVertex Y MochilaEdge* son:

```
record MochilaVertex(Integer index, Integer capacidadRestante)
        implements VirtualVertex<MochilaVertex, MochilaEdge,
             Integer> {
      public static Predicate<MochilaVertex> goal() {
             return v->v.index == MochilaVertex.n;
      }
```

```
      public Integer greedyAction() {
        returnMath.min(this.capacidadRestante/
                    DatosMochila.getPeso(index),
             DatosMochila.getNumMaxDeUnidades(index));
      }
```

```
      @Override
      public List<Integer> actions() {
             if(this.index == n) return new ArrayList<>();
             Integer nu = greedyAction();
             if(this.index == n-1)
                    return new ArrayList<>(nu);
             List<Integer> alternativas =
             IntStream.rangeClosed(0,nu)
                          .boxed()
                          .collect(Collectors.toList());
             Collections.reverse(alternativas);
             return alternativas;
      }
```

```
        @Override
        public MochilaVertex neighbor(Integer a) {
                MochilaVertex r;
                Integer cr = capacidadRestante - a *
                        DatosMochila.getPeso(index);
                if (this.index == MochilaVertex.n - 1 ||
                                this.capacidadRestante == 0.)
                        r = MochilaVertex.of(MochilaVertex.n,0);
                else r = MochilaVertex.of(index + 1, cr);
                return r;
        }
```

```
        @Override
        public MochilaEdge edge(Integer a) {
                MochilaVertex v = this.neighbor(a);
                return MochilaEdge.of(this,v,a);
        }
        …
}
```

```
record MochilaEdge(MochilaVertex source, MochilaVertex target,
                Integer action, Double weight)
        implements SimpleEdgeAction<MochilaVertex,Integer> {

        public static MochilaEdge of(MochilaVertex v1,
                MochilaVertex v2, Integer a) {
                Double w = a*DatosMochila
                        .getValor(v1.index()).doubleValue();
                return new MochilaEdge(v1,v2, a, w);
        }

}
```

A partir de aquí tenemos disponible el grafo virtual de la forma:

```
EGraph<MochilaVertex, MochilaEdge> graph =
        SimpleVirtualGraph.sum(e1,MochilaVertex.goal(),
                x->x.weight());
```

Podemos observar que todos los vértices finales, todos los que cumplen la condición *v.i ==n*, tienen solución por lo que la restricción *goalHasSolution()* es true.

Caminos y grafos extendidos

El tipo *EGraphPath<V,E>,* que llamaremos *camino extendido,* es una extensión de *GraphPath<V,E>* que incorpora las propiedades del camino, la forma de calcular su peso, el valor de la función objetivo asumiendo conocido este valor para el vértice vecino, el valor de la función objetivo para los problemas finales, y otros métodos que pueden simplificar el uso de los caminos.

Los caminos que hemos visto hasta ahora tienen asociado un peso que es la suma del peso de las aristas. También hemos comentado que el peso de las aristas del grafo que asociamos a un problema está relacionado con la función objetivo del modelo. En los ejemplos que hemos visto la función objetivo es un sumatorio que debe ser maximizado o minimizado. Ha sido el caso del problema del puzle y del problema de la mochila. En ese caso el peso de las aristas está relacionado con cada uno de los sumandos de la función objetivo. Minimizar o maximizar la función objetivo es equivalente a encontrar un camino de peso mínimo o máximo.

Pero la función objetivo puede ser más compleja y por eso tenemos que extender la noción de peso del camino. La primera posibilidad es que en la función objetivo aparezcan también pesos asociados a los vértices o incluso al paso por los vértices.

Sea w_i el peso de la arista i, z_i el peso asociado al vértice i, u_{ijk} el peso asociado al paso por el vértice i asumiendo que se ha llegado a él mediante la arista j y se sale de él mediante la arista k.

Este peso para un camino aparece en problemas de tráfico aéreo. Imaginemos que tenemos un conjunto de vuelos, cada vuelo tiene un origen, un destino y un tiempo de vuelo y queremos calcular el vuelo más corto desde un origen a un destino posiblemente con varias escalas. Claramente el problema se puede modelar mediante un grafo y la función objetivo, el tiempo total, es la suma del tiempo de los vuelos más las esperas en los aeropuertos para conectar un vuelo con otro. Estas esperas son los u_{ijk} anteriores,

Si representamos por C_v los índices de los vértices de un camino, por C_E los índices de sus aristas, por C_p el conjunto de tripletas ijk dónde i es un

vértice del camino, j la arista el camino que llega al vértice i, y k la arista el camino que sale del vértice i del camino y por $|C|$ su peso entonces tenemos.

$$|C| = \sum_{j \in C_E} w_j + \sum_{i \in C_V} z_i + \sum_{ijk \in C_P} u_{ijk}$$

Cada tripleta ijk representa un vértice del camino junto con las aristas del camino entrante y saliente a ese vértice. Si z_i, u_{ijk} son cero entonces el peso del camino es solo la suma de los pesos de las aristas.

Estos caminos cuyo peso se calcula con el sumatorio anterior los llamamos caminos de tipo *Sum*.

Esos pesos en los vértices, el peso del paso por los vértices, el vértice inicial, etc. deben ser incorporados al grafo. Un grafo extendido, *EGraph<V,E>,* es, como hemos dicho, una extensión de *Graph<V,E>* con toda esa información.

Pero aún hay otras formas de definir el peso de un camino. Una segunda posibilidad es cuando el peso del camino se define como el peso asociado a su último vértice. Esta posibilidad aparece cuando la función objetivo no es un simple sumatorio. Estos caminos cuyo peso se asocia a una propiedad del último vértice los llamamos caminos de tipo *Last*. Los caminos de este tipo *Last* pueden ser reducidos a los de tipo *Sum* usando como peso de las aristas la diferencia entre la propiedad que define el peso del camino entre los vértices origen y destino. Por lo tanto, elegir un camino de tipo *Last* es simplemente por conveniencia en algunos problemas concretos.

Aunque puede haber otras formas de definir el peso de un camino aquí vamos a considerar dos: cuando el peso es la suma de pesos asociados a aristas, vértices y paso por los vértices que llamaremos camino tipo *Sum*, y cuando el peso del camino viene dado por una propiedad del último vértice que llamaremos camino tipo *Last*. El tipo *PathType* recoge esas dos posibilidades: *Sum* y *Last*.

Según la forma concreta de calcular el peso del camino tendremos dos tipos de caminos: *Sum* y *Last.* En los primeros el peso del camino es la

suma de los pesos de las aristas más otros pesos adicionales en algunos y el del segundo el peso del camino es el valor de una propiedad del último vértice. Según el tipo de sus caminos los grafos extendidos también serán de tipo *Sum* o tipo *Last*.

Veamos las diversas propiedades, su significado y las peculiaridades según el tipo del camino.

```
enum PathType{Sum,Last}
interface EGraphPath<V,E> extends GraphPath<V,E> {
	E lastEdge();
	EGraphPath<V, E> add(E edge);
	Double add(E edge,V vertexActual,
		Double accumulateValue,E lastEdge);
```

lastEdge(): La última arista
add(E e): Un nuevo camino tras añadir la arista
add(E edge, V vertexActual, Double accumulateValue, E lastEdge): El peso que tendría el camino tras añadir la arista *edge,* asumiendo que el último vértice es *vertexActual,* el peso del camino hasta el último vértice es *accumulateValue* y la última arista del camino es *lastEdge*.

```
EGraphPath<V, E> remove();
```

remove(): Un nuevo camino tras eliminar la última arista y el último vértice.

```
EGraphPath<V, E> copy();
```

copy(): Una copia

```
Double estimatedWeightToEnd(V vertexActual,Double weight,
	Predicate<V> goal, V end,
	TriFunction<V,Predicate<V>,V,Double> heuristic);
```

Estimación del peso del camino desde el vértice inicial hasta el vértice final pasando por el vértice actual, usando la heurística y asumiendo que el peso el camino hasta el vértice actual es *weight.* Este valor se calcula de forma diferente para los caminos de tipo *Sum* y los de tipo *Last*.

En el caso de caminos de tipo *Sum* la estimación del peso del camino se obtiene sumando el peso hasta el vértice actual, *weight,* más la heurística, que estima el peso del vértice actual hasta el vértice final, y posiblemente el peso del paso por el vértice actual.

En el caso de caminos de tipo *Last* el peso de un camino se determina por el valor de una propiedad del último vértice. En este caso la heurística es una estimación de la propiedad escogida del último vértice. Por lo tanto la estimación del peso del camino actual es igual a la heurística.

```
Double boundedValue (V vertexActual,Double weight,
       Predicate<V> goal, V end, E edge,
       TriFunction<V,Predicate<V>,V,Double> heuristic);
```

El valor devuelto por el método *boundedValue* es una cota para el peso del camino desde el vértice inicial que pasa por la arista *edge,* usa la heurística y asume que el peso del camino hasta el vértice actual es *weight.* A diferencia del método *estimatedWeightToEnd* anterior el método *boundedValue* asume que tomamos la arista *edge* tras el vértice actual. Esto nos permitirá filtrar aristas no intersantes en los algortimos que buscan caminos óptimos.

Esta cota se calcula de forma diferente para los caminos de tipo *Sum* que los de tipo Last. En el primer caso es la suma de *weight,* el peso hasta el vértice actual*,* el peso de la arista *edge*, la heurística desde el vértice alcanzado siguiendo edge hasta el final y posiblemente el peso del paso por el vértice actual.

En el segundo caso es igual a la heurística desde el vértice alcanzado siguiendo *edge* hasta el final.

```
       EGraphPath<V, E> concat(GraphPath<V,E> path);
       GraphPath<V, E> reverse();
       PathType type();
}
```

concat(GraphPath<V,E> Pat): El resultado de concatenar el nuevo camino
reverse(): El camino inverso
type(): El tipo de camino

Los caminos extendidos son una extensión del tipo *GraphPath<V,E>.* Las propiedades de este tipo son:

- *List<V> getEdgeList():* Lista de aristas del camino
- *List<E> getVertexList():* Lista de vértices del camino
- *V getStartVertex():* Primer vértice
- *V getEndVertex():* Último vértice
- *Double getWeight()*: Peso del camino
- *Integer getLength():* Longitud del camino (número de aristas)
- *Graph<V,E> getGraph():* Grafo al que pertenece el camino

La información necesaria para los caminos extendidos la incorporamos en el grafo virtual extendido, el tipo *Egraph<V,E>,* algunas de cuyas propiedades son*:*

- *goal():* El predicado que cumplen los vértices finales.
- *goalHasSolution():* Predicado sobre los vértices finales que indica si tiene solución o no. Valor por defecto true.
- *goalSolutionValue(V va):* El valor de la función objetivo para un vértice final *va*. En general para los caminos de tipo *Sum* este valor es cero y para los de tipo *Last* el valor de la propiedad relevante del vértice.
- *fromNeighborSolutionValue(Double w, V va, E e, E le*): El valor de la función objetivo en el vértice actual *va*, asumiendo que el valor de la función objetivo para el vecino según la arista *e* es *w* y la arista anterior es *le*. Para caminos de tipo Sum este valor es *add(w,va,e,le)* y para caminos tipo Last es *w*.
- *getVertexWeight(V vertex):* El peso asociado a un vértice. Su valor por defecto es null.
- *getVertexPassWeight(V vertex, E edgeIn, E edgeOut).* El peso asociado al paso por un vértice con *edgeIn* como arista de llegada y edgeOut como arista de salida. Su valor por defecto es null.
- *edgesListOf(V v):* La lista de aristas salientes de un vértice.
- *oppositeVertex(E edge, V v):* El vértice opuesto a uno dado en una determinada arista.
- *startVertex():* El vértice inicial.

- *endVertex():* Uno de los vértices finales. Su valor por defecto es null.
- *pathType():* El tipo de los caminos del grafo*: Sum o Last.*
- *type():* El tipo del problema: *Max, Min, One, All*. Estos valores indican, respectivamente, si buscamos el máximo de la función objetico o el mínimo o alternativamente solo una solución o todas las soluciones. Algunos algoritmos como A* y PDR solo admiten losdos primeros valores. El algoritmo BT admite todas las posibilidades.
- *Function<V, E> greedyEdge():* La estrategia voraz que asociaremos a vértice del grafo.
- *TriFunction<V, Predicate<V>, V, Double> heuristic:* La heurística que asociaremos a cada vértice del grafo.

```
public interface EGraph<V, E> extends Graph<V,E> {

	double getVertexPassWeight(V vertex, E edgeIn,
		E edgeOut);
	double getVertexWeight(V vertex);
	List<E> edgesListOf(V v);
	EGraphPath<V, E> initialPath();
	V oppositeVertex(E edge, V v);
	V startVertex();
	V endVertex();
	Predicate<V> goal();
	Predicate<V> goalHasSolution();
	Double goalSolutionValue(V vertexActual);
	Double fromNeighbodSolutionValue(Double weight,
		V vertexActual, E edge, E lastEdge);
	Double add(E edge, V vertexActual, Double weight,
		E lastEdge);
	Double estimatedWeightToEnd(V vertexActual,
		Double weight, Predicate<V> goal, V end,
		TriFunction<V,Predicate<V>,V,Double> heuristic);
	Double boundedValue (V vertexActual, Double weight,
		Predicate<V> goal, V end, E edge,
		TriFunction<V,Predicate<V>,V,Double> heuristic);
	PathType pathType();
	Function<V, E> greedyEdge();
	TriFunction<V, Predicate<V>, V, Double> heuristic();
	public static enum Type{Min,Max,All,One}
	Type type();
```

El grafo extendido también incluye propiedades ligadas con sus caminos: *add, estimatedWeightToEnd, boundedValue.* Esta propiedades como

hemos explicado arriba calculan, respectivamente, el peso que tendría el camino tras añadir la arista *edge* asumiendo que el peso del camino hasta el vértice actual es *weight* y la arista anterior *lastEgde*, estimación del peso del camino desde el vértice inicial hasta el vértice final usando la heurística y asumiendo que el peso el camino hasta el vértice actual es *weigh y u*na cota para el peso del camino desde el vértice inicial que pasa por la arista edge*,* usa la heurística *heuristic* y asume que el peso del camino hasta el vértice actual es *weight.*

Un grafo extendido se puede obtener a partir de un grafo virtual mediante la clase *SimpleVirtualGraph<V,E>* o a partir de un grafo ordinario a partir de la clase *EgraphI<V,E>.* Algunos de los métodos de factoría se muestran abajo. Como podemos ver, instanciar un grafo extendido puede tener muchos parámetros. Por ello en el repositorio se ha diseñado un *builder* para esta tarea. El *builder* tiene dos instancias: una para grafos virtuales y otras para grafos extendidos obtenidos a partir de un grafo real.

```
public interface EGraphBuilder<V, E> {
	EGraphBuilder<V, E> edgeWeight(
		Function<E, Double> edgeWeight);
	EGraphBuilder<V, E> vertexWeight(
		Function<V, Double> vertexWeight);
	EGraphBuilder<V, E> vertexPassWeight(
		TriFunction<V, E, E, Double> vertexPassWeight);
	EGraphBuilder<V, E> startVertex(V startVertex);
	EGraphBuilder<V, E> goal(Predicate<V> goal);
	EGraphBuilder<V, E> endVertex(V endVertex);
	EGraphBuilder<V, E> goalHasSolution(
		Predicate<V> goalHasSolution);
	EGraphBuilder<V, E> pathType(PathType pathType);
	EGraphBuilder<V, E> greedyEdge(
	Function<V, E> greedyEdge);
	EGraphBuilder<V, E> heuristic(
	     TriFunction<V, Predicate<V>, V, Double> heuristic);
	EGraphBuilder<V, E> type(Type type);
	EGraph<V, E> build();
}
```

Un *builder* es un patrón de diseño que permite construir objetos complejos paso a paso y en particular instanciar un objeto que puede tener muchos parámetros con valores por defecto. Los detalles pueden verse en el repositorio.

Un ejemplo de uso es:

```
EGraph<ColorVertex, ColorEdge> graph =
EGraph.virtual(e1,ColorVertex.goal(),PathType.Last,Type.Min)
       .vertexWeight(v->v.nc().doubleValue())
       .heuristic((v1,p,v2)->(double) v1.nc())
       .build();
```

Tareas y procesadores

El problema se formula de la siguiente manera: Dado una lista de n tareas con duraciones d_i y un conjunto de m procesadores buscar la asignación de tareas a procesadores tal que el tiempo final de ejecución sea mínimo.

Este problema puede ser modelado mediante Programación Lineal Entera pero aquí vamos a proponer otro modelo para buscar un grafo asociado al problema. Este modelo es:

$$\min \max_{j:0..n-1} \Sigma_{i=0|x_i=j}^{n-1} d_i$$
$$0 \leq x_i < m, \quad i \in [0, n-1]$$
$$int\ x_i, \quad i \in [0, n-1]$$

La variable x_i indica el procesador al que se asigna la tarea i. Como vemos la función objetivo no es un simple sumatorio. Para buscar un grafo lo generalizamos:

TareasYProcesadoresVertex:

Propiedades:

- i: Integer, basica
- cargas: List<Double> de tamaño m, cargas de los procesadores, básica
- cm: Double, derivada, carga del procesador más cargado

Invariante:

- $cm = \max_{j:0..m-1} cargas[j]$

Problema final:

1. p.i ==n

Acciones:

$A_i = \{0..m-1\}$

Vecino

$neighbord(a) = (i+1, cargas + (a, d_i))$

Con la notación $cargas + (a, d_i)$ indicamos que el par (a, d_i) actualiza la casilla a de la lista $cargas$ añadiendole la cantidad d_i.

Peso de la arista: Debido a la forma de la función objetivo elegimos caminos de tipos *Last* por facilidad. Si hubiéramos elegido caminos de tipo Sum el peso de una arista *e* sería *e.target.cm - e.origen.cm.*

Peso del camino: Debido a la forma de la función objetivo elegimos caminos de tipos *Last* y podemos asignar un peso a un camino completo que sean una función del último vértice del camino. En concreto la propiedad *cm* del último vértice. Efectivamente tal como hemos diseñado el grafo tras cada arista una tarea ha sido asignada a un procesador. A lo largo del camino las sucesivas asignaciones de tareas quedan acumuladas en los procesadores. Lo que queremos minimizar es justamente la carga del procesador más cargado al final del camino.

Como hemos comentado antes podríamos haber elegido que el camino fuera de tipo *Sum* con el peso indicado más arriba.

Un grafo extendido *Egraph<V,E>* es una extensión de *Graph<V,E>* que contiene, además de los vértices y las aristas información sobre las propiedades de los caminos del grafo, el vértice inicial, los posibles vértices finales, si un vértice final tiene solución o no, la forma de calcular el peso del camino, el camino inicial formado solo por el vértice inicial, etc. Ya presentamos arriba una versión reducida. La versión completa es:

```
interface EGraph<V, E> extends Graph<V,E> {

	double getVertexPassWeight(V vertex, E edgeIn,
		E edgeOut);
	double getVertexWeight(V vertex);
	List<E> edgesListOf(V v);
	EGraphPath<V, E> initialPath();
	V startVertex();
	Predicate<V> goal();
	V endVertex();
	Predicate<V> goalHasSolution();
	PathType pathType();
}
```

En la información de un grafo extendido aparece la información:

- *startVertex:* El vértice de inicio de los caminos en este grafo extendido
- *edgeWeight:* El peso de las aristas. Su valor por defecto es por el método *getEdgeWeight(edge)* del grafo base.
- *vertexWeight:* El peso de los vértices. Su valor por defecto 0.
- *vertexPassWeight:* El peso de paso por los vértices. Su valor por defecto 0.
- *type:* El tipo de los caminos de este grafo. Según la forma de calcular el peso de un camino éste tiene diferentes tipos que estudiamos antes. Puede ser *Sum* o *Last.*
- *goal():* El predicado que cumplen los vértices finales
- *endVertex():* Un vértice final
- *goalHasSolution():* Un predicado que indica si un vértice final tiene solución o no. Por defecto es *v->true.*
- *edgesListOf(V v):* Una lista con los vértices vecinos a *v*.
- *initialPath():* Un camino formado solo por el vértice inicial y del tipo establecido.

Un grafo extendido se puede obtener a partir de un grafo virtual o añadiendo la información necesaria a un grafo real. En cada caso la información no aportada toma el valor por defecto.

Los métodos de factoría para obtener un *EGraphBuilder<V, E>* a partir del cula construir un *EGraph<V, E>* se encuentran en el tipo *EGgraph*.

```
EGraphBuilder<V, E> virtual(V startVertex,
       Predicate<V> goal,PathType pathType,Type type){…}

EGraphBuilder<V, E> ofGraph(G graph,V startVertex,
       Predicate<V> goal,PathType pathType,Type type){…}
```

El método *virtual* nos servirá para construir un *Egraph virtual* y el método ofGraph para construir un *Egraph* a partir de un objeto de tipo *Graph<V,E>*. Teniendo un *EGraphBuilder<V, E>* podemos ir añadiendo los valores de las propiedades relevantes y finalmente construir el grafo extendido. Las propiedades no indicadas en los parámetros tomaran valores por defecto fijados.

Un ejemplo de añadir información a un grafo real lo podemos encontrar en el ejemplo de los vuelos que se encuentra en el repositorio.

Este ejemplo modela problemas de tráfico aéreo. Imaginemos que tenemos un conjunto de vuelos, cada vuelo tiene un origen, un destino y un tiempo de vuelo y queremos calcular el vuelo más corto desde un origen a un destino posiblemente con varias escalas. Claramente el problema se puede modelar mediante un grafo con vértices que son aeropuertos y aristas que son vuelos de un aeropuerto a otro. Cada vuelo tiene como propiedades el tiempo de salida del aeropuerto origen y el tiempo de llegada al aeropuerto destino. En el problema la función objetivo, el tiempo total para ir de un aeropuerto a otro posiblemente pasando por algunos aeropuertos intermedios, es la suma del tiempo de los vuelos más las esperas en los aeropuertos intermedios para conectar un vuelo con otro.

```
DirectedWeightedMultigraph<String,Vuelo> graph =
       leeGrafo("ficheros/vuelos.txt");

String end = "Malaga";

EGraph<String, Vuelo> g =
       EGraph.ofGraph(graph,"Sevilla",
              v->v.equals(end),PathType.Sum,Type.Min)
       .edgeWeight(Vuelo::getDuracion)
       .vertexPassWeight(
              (v,e1,e2)-> Vuelo.getVertexPassWeight(v,e1,e2))
       .build();
```

En el ejemplo anterior se lee de un fichero un grafo *graph,* dirigido con pesos con vértices de tipo *String* y aristas de tipo *Vuelo*. Para construir el grafo exendido a partir del *DirectedWeightedMultigraph<String,Vuelo>* se indican cuáles son los pesos de las aristas con el método *edgeWeight* y los pesos de los pasos por los vértices con *vertexPassWeight.* El primero toma como parámetro una función tal que para cada arista nos da su peso. El segundo, un *TriFunction* que para cada vértice, una arista de entrada y otra de salida no da el peso de paso por el vértice.

Las propiedades no indicadas en los parámetros tomaran valores por defecto.

Heurísticas y Funciones de cota

En los grafos que estamos usando para representar problemas existe siempre un *vértice inicial*, *startVertex*, de donde parten todos los caminos que se extienden hasta un último vértice al que llamamos el *vértice final*. Los *vértices finales* los definimos, en general, mediante aquellos que cumplen un predicado sobre vértices, *goal*. Un vértice final es un vértice que cumpla ese predicado. En la mayoría de los casos los vértices finales suelen ser vértices creados con ese fin que puede que no representen ningún problema real.

En cada momento del camino estaremos en el *vértice actual*. Llamamos *camino actual* a uno que partiendo del vértice inicial pasa por el vértice actual y continúa hasta un vértice el final.

Una heurística es una estimación del peso del camino desde el vértice actual hasta un vértice final y es una función del tipo siguiente o un método con el mismo tipo funcional:

```
TriFunction<V,Predicate<V>,V,Double> heuristic
```

El primer parámetro es el vértice actual, el segundo el objetivo, un predicado que define los vértices finales y el tercero un vértice final. El tercero es vértice final en concreto, aunque podría ser *null* en cuyo caso el vértice final concreto no es relevante para el cálculo de la heurística.

A la cantidad $h(a)$, la longitud estimada del camino desde v_a, el vértice actual, hasta v_t, un vértice final, se le llama *heurística*. Esta es una medida que va a ser muy relevante, como veremos, en los algoritmos *A*, Backtraking y Programación Dinámica de Reducción* que estudiaremos más abajo.

Junto con la anterior definimos $g(a)$ y $f(a)$. La primera $g(a)$ es el peso del camino desde el vértice inicial hasta el actual v_a. La segunda $f(a)$ es una estimación del peso del camino desde el vértice inicial al final que pasa por el vértice actual.

En el caso de caminos de tipo *Sum* tenemos que $f(a) = g(a) + h(a)$ y para caminos de tipo *Last* $f(a) = h(a)$.

Una heurística tiene que cumplir las siguientes propiedades:

- La heurística tiene que ser *admisible*. Si asumimos que $h^*(a)$ es el peso real del camino más corto desde v_a hasta v_f y $h(a)$ el estimado entonces debe cumplirse la condición $h(a) \leq h^*(a)$. Es decir, la distancia estimada hasta el final debe ser una cota inferior del valor real.
- La heurística es *monótona o consistente* si $h(a) \leq w(a,b) + h(b)$ para todo par de vértices v_a, v_b conectados por una arista de peso $w(a,b)$. Si una heurística es monótona se cumple $f(a) \leq f(b)$ siendo a, b dos vértices consecutivos del camino y el problema es de minimización. En problemas de maximización se cumple $f(a) \geq f(b)$. La exigencia de que una heurística sea monótona es más fuerte que simplemente ser admisible. El general será suficiente que la heurística sea admisible, pero si es monótona es una garantía de que es admisible.
- Si el problema fuera de maximización entonces, dado que cambiamos el signo en los pesos de las aristas, la heurística *admisible* debe cumplir $h(a) \geq h^*(a)$ y una relación similar para que la heurística sea monótona.
- En general tendremos un conjunto de vértices finales. Todos los que cumplan un determinado predicado. Para caminos de tipo *Sum* la heurística debe ser cero, $h(t) = 0$, en todos los vértices finales y en los caminos de tipo *Last* igual a la propiedad asociada

al último vértice del camino que conduce desde el vértice inicial a uno final. Es decir $h(t) = g(t)$.

Veremos ejemplos de heurísticas más adelante. Su uso es muy importante para guiar a los algoritmos de camino mínimo a encontrar eficientemente el camino óptimo. A la suma del peso del camino hasta el vértice actual más la heurística lo llamamos *peso estimado del camino*. Es lo que hemos llamado $f(a)$.

Hay dos formas de calcular las heurísticas: mediante un cálculo de las diferencias entre el vértice actual y el final o mediante algoritmos voraces de los cuales que veremos ejemplos más abajo.

Siguiendo la primera de las posibilidades en el caso del *puzle* una heurística adecuada sería *numDiferentes(v1,v2)* que mide el número de elementos diferentes en las casillas. Puesto que es un problema de minimización y podemos ver que este número es menor que el número óptimo de pasos para llevar el puzle de *v1* a *v2* la heurística es admisible.

Hemos de tener en cuenta que el cálculo de la heurística es diferente si el camino es de tipo *Sum* o de tipo *Last*. En los caminos de tipo *Sum* se trata de estimar la suma de los pesos de las aristas, y otros pesos que puedan aparecer, del camino del vértice actual al final. En los caminos de tipo *Last* se trata de estimar la propiedad indicada del vértice final.

Las *heurísticas* se usan, entre otras cosas, para calcular las *funciones de cota*. Estas funciones dependen de *a*, una de las acciones a tomar, y acotan el valor de la función objetivo de las soluciones que encontraremos si escogemos esta alternativa para llegar a un vértice final partiendo del vértice actual.

Las funciones de cota para caminos de tipos *Sum* se pueden calcular a partir de una heurística de la forma:

```
Double cota(V v1, V vf, A a) {
       E edge = v1.edge(a);
       V v = v1.neighbor(a);
       return edge.weight + heuristica(v,vf);
                     //pueden aparecer otros pesos
}
```

Las funciones de cota para caminos de tipos Last se pueden calcular a partir de una heurística de la forma:

```
Double cota(V v1, V vf, A a) {
       E edge = v1.edge(a);
       V v = v1.neighbor(a);
       return heuristica(v,vf);
}
```

Las *funciones de cota* nos permiten descartar alternativas si cumplen

$$ct(v1, a) \geq f$$

Donde f es un valor ya conocido de la función objetivo para una solución previamente encontrada. Si se cumple lo anterior la alternativa a puede ser descartada porque todas las soluciones encontradas siguiendo a serán peores o iguales a la ya encontrada cuyo valor de la función objetivo era f. O dicho de otra manera solo seguimos buscando por aquellas alternativas en que se cumpla $ct(v1, a) < f$.

Esto es así porque si f_a es el valor óptimo de la función objetivo que se alcanza siguiendo la alternativa a entonces $ct(v1, a) = w_a + h(v2) \leq f_a$, por la propiedad exigida para la heurística. Por lo tanto $f_a \geq ct(v1, a) > f$ y el camino que empieza por a puede ser descartado porque el valor de la función objetivo que encontraremos será mayor.

De forma similar para que las *funciones de cotas* en un *problema de maximización* permiten descartar una alternativa si:

$$ct(v1, a) \leq f$$

Pesos de caminos, soluciones voraces y heurísticas

Como hemos comentado las soluciones de los problemas asociados a grafos son definido por caminos en el grafo que van desde el vértice inicial a uno de los vértices finales.

Para cada problema definido por un grafo hay un conjunto de soluciones y por lo tanto un conjunto de caminos. De todos ellos uno es el que define la solución óptima y su peso el peso óptimo que llamaremos w^*. Las soluciones voraces producen soluciones que pueden no ser óptimas.

Llamaremos peso voraz w^g al peso de una solución voraz. Ambos pesos cumplen $w^g \geq w^*$ si el problema es de minimización y $w^g \leq w^*$ si s de maximización.

Junto al peso voraz disponemos del peso heurístico que llamaremos $w^h = h(s)$. Este peso es igual a la heurística desde el vértice inicial s a uno de los finales t. Si la heurística es admisible la relación entre los tres pesos es $w^g \geq w^* \geq w^h$ si el problema es de minimización y $w^g \leq w^* \leq w^h$ si es de maximización. Estas relaciones nos permiten hacer un test a las soluciones voraces y las heurísticas. Si se cumple $w^g = w^h$ podemos asegurar que la solución voraz es la solución óptima. Por otra parte la distancia entre w^g, w^h nos da una idea de la bondad de la solución voraz y la heurística. Mejores en la medida que están más cerca.

El valor voraz no sirve para establecer cotas a los valores de la función objetivo. Estas cotas nos van a permitir filtrar alternativas que no conducirán a soluciones mejores como veremos.

Junto a los pesos anteriores usaremos una estimación para un camino que parte el vértice inicial se encuentra en el vértice actual a y debe continuar hasta un vértice final. A la estimación de este peso lo llamaremos w^a. Anteriormente hemos llamado a este peso $f(a) = w^a$. En los caminos de tipo *Sum* este peso es igual a $w^a = w_1^a + w_2^a$. Siendo w_1^a el peso del camino ya recorrido desde el vértices inicial hasta a y $w_2^a = h(a)$ el valor de la heurística desde el vértice a hasta el final. En el caso de caminos tipo *Last* $w^a = h(a)$. La relación de estos pesos es $w^g \geq w^* \geq w^a$ si el problema es de minimización y $w^g \leq w^* \leq w^a$ si es de maximización. Los pesos w^a sirven para ordenar los caminos que han llegado a diversos vértices intermedios pero que continuarán hasta el final. Veremos su importancia más adelante.

Por último, tenemos estimaciones para los pesos de los caminos que han llegado hasta el vértice actual a, toman la alternativa r para alcanzar el vértice b y luego continúan hasta el final t. Estos pesos los llamaremos $w^{a,r}$. Estos pesos cumplen también $w^g \geq w^* \geq w^{a,r}$ si el problema es de minimización y $w^g \leq w^* \leq w^{a,r}$ si es de maximización. Los pesos $w^{a,r}$ sirven para filtrar alternativas que conducirían a soluciones peores que las ya encontradas. Veremos su importancia más adelante.

En los caminos de tipo *Sum* los pesos anteriores este peso es igual a $w^{a,r} = w_1^a + ct(a,r)$. Siendo w_1^a el peso del camino ya recorrido desde el vértices inicial hasta a y $ct(a,r)$ el valor de la función de cota del camino que parte del vértice a toma la alternativa hasta r y continua hasta el final t. En el caso de caminos tipo *Last* $w^{a,r} = ct(a,r) = h(b)$.

La relación entre las heurísticas y las funciones de cota la hemos visto anteriormente. En definitiva en los caminos de tipo Sum $ct(a,r) = w(a,b) + h(b)$. Siendo $w(a,b)$ el peso de la arista de a a b posiblemente con pesos adicionales como hemos visto arriba. En el caso de caminos tipo *Last* $ct(a,r) = h(b)$.

Acciones, simetrías y equivalencia de vértices

Los grafos que diseñamos tienen sus vértices definidos de forma implícita en el sentido de que no estamos interesados en enumerarlos. Los definimos cono el conjunto de valores de un tipo, el tipo del vértice, que cumplen una propiedad. Decimos que son vértices válidos si cumplen la propiedad dada. De la misma forma las acciones válidas son el conjunto de valores de un tipo, el tipo de la acción, que llevan de un vértice válido a otro también válido. Cada par vértice-acción válida define una arista del grafo. El tipo del vértice, la propiedad que deben cumplir los vértices, el tipo de la arista y la propiedad que cumplen las aristas válidas se deducen del modelo que hemos hecho para un problema.

Lo anterior permite restringir los caminos posibles que llevan del vértice inicial a uno de los finales para que coincida con las soluciones del problema.

En muchos problemas se presentan, además, simetrías que darán lugar a soluciones distintas pero equivalentes. Estas soluciones equivalentes tienen el mismo valor de la función objetivo y algunas restricciones adicionales. Si solo estamos interesados en una de las soluciones simétricas el grafo puede ser podado. Esto se concretará en que las acciones disponibles solo escogerán una las posibilidades simétricas.

Para concretar esto asumimos que se puede definir una relación de equivalencia entre soluciones en la forma de un predicado $eqs(s1, s2)$. En un grafo virtual podemos asociar a cada vértice una solución parcial por

lo que la relación de equivalencia anterior, si existe, se puede extender a los vértices en la forma $eqs(v1, v2)$. La idea es filtrar el conjunto de alternativas para quedarnos solo con las que conduzcan a vértices no equivalentes:

$$A_v^f = \{a: A_v \mid V_{a1,a2 \mid a1 \neq a2} \, !\, (eqs(vcn(v, a1), vcn(v, a2)))\}$$

Veamos como ejemplo el problema de las tareas y los procesadores visto anteriormente. Hemos dicho que las acciones válidas eran el conjunto de los m procesadores disponibles.

$$A_i = \{0..m-1\}$$

Aquí podemos ver una simetría en el sentido que dos procesadores que estén igualmente cargados son indistinguibles. Podemos restringir las acciones para escoger solo uno de los procesadores igualmente cargados. Según esa simetría podemos definir que dos vértices son equivalentes si el número de procesadores con la misma carga es el mismo en ambos vértices. Es decir, agrupamos los procesadores según la carga que tienen. Dos vértices son equivalentes desde este punto de vista si tienen el mismo número de grupos formados según cada grupo tiene los mismos elementos.

Para filtrar las acciones las agrupamos según la carga del procesador asociado a la acción. Dos acciones en el mismo grupo tienen asociado dos procesadores con la misma carga. Finalmente escogemos solo una acción de cada grupo descartando las demás.

El código para las acciones quedaría entonces como:

```
List<Integer> actions() {
	if(this.index == DatosTyP.n) return List2.of();

	Map<Double, List<Integer>> s =
		IntStream.range(0, DatosTyP.m).boxed()
		.collect(Collectors
			.groupingBy(p -> this.cargas().get(p)));

	return  s.values().stream()
		.map(ls -> ls.get(0))
		.collect(Collectors.toList());
}
```

En definitiva, formamos grupos con los procesadores con la misma carga y tomamos de cada grupo uno de los procesadores como alternativa válida.

Heurísticas

Como vemos, los grafos virtuales nos permiten asociar un grafo a un problema de tal manera que los caminos del vértice inicial hasta los finales definan las soluciones del mismo.

Este grafo puede ser de un tamaño muy grande por lo que es conveniente filtrar al máximo las aristas para reducir en lo posible el número de caminos válidos. Esto es lo que hacemos defiendo las acciones válidas, y por lo tanto las aristas válidas en un vértice. También hacemos esto usando heurísticas y funciones de cota tal como hemos explicado.

En algunos casos es posible reordenar los vértices de alguna forma que permita encontrar antes la solución óptima. Las heurísticas son muy dependientes de cada problema, aunque se pueden dar indicaciones generales.

No entraremos aquí en mucho detalle de este tema. Solo indicar que una heurística muy general nos sugiere acceder antes los vértices que están más restringidos. Es decir, a los que tienen menos aristas asociadas.

Veremos algunos ejemplos, entre ellos el problema del sudoku. Es este caso los vértices del grafo virtual van a ser las diferentes configuraciones de las casillas a medida que lo rellenamos.

Las casillas que nos quedan sin rellenar tendrán diferentes restricciones según los valores ya asignados a las casillas rellenas. La metaheurística anterior nos indica que ordenemos los vértices a visitar de mayor a menor número de restricciones asociadas. Esto implica escoger la siguiente casilla a rellenar. Los detalles pueden verse en el ejemplo del Sudoku más adelante.

Diseño de un grafo virtual

Para diseñar un grafo virtual partimos siempre de un modelo del problema. Es decir, explicitamos las variables y su tipo, la función objetivo y las restricciones que deben satisfacer las variables. A partir del modelo

diseñamos el grafo incluyendo toda la información relevante que será usada posteriormente por los algoritmos que buscarán el camino óptimo.

El primer paso es escoger los tipos V de los vértices, E de las aristas y A de las acciones.

El grafo debe tener unos vértices de un tipo V cuyas propiedades obtendremos mediante generalización a partir de las variables del modelo. Es muy importante especificar el conjunto de valores válidos de los vértices.

Las aristas, de tipo E, deben tener asociado un peso del tal forma, que el peso del camino represente los valores de la función objetivo.

Para el diseño las acciones partimos del tipo A cuyos valores no dan las posibles acciones y posteriormente filtramos esos valores para el vértice alcanzado tras tomar la acción sea válido. La concreción del método *actions()* del vértice sigue esas ideas. A partir de lo anterior podemos diseñar el método *neighbor(a)* que encuentra el vecino de un vértice tras tomar una acción válida.

El orden de las acciones es muy relevante para la eficiencia de los algoritmos posteriores. La recomendación general es que la primera acción sea la voraz para ese vértice y el resto ordenadas por su cercanía a la voraz.

El orden en que se recorren los vértices también es muy relevante por la misma razón. La combinación del orden de los vértices más el orden de las aristas debe dar lugar a que la primera arista sea la voraz. El orden de los vértices es dependiente del tipo de problema, pero en los problemas donde hay que recorrer un conjunto de elementos: objetos a introducir en la mochila, casillas de un sudoku, etc. En estos casos tenemos que recorrer esos elementos según un orden. Este orden es muy importante. Esta ordenación puede hacer solamente una vez al principio o en cada paso del algoritmo justamente al final del método *neighbor(a)*. El Sudoku es un caso particular donde se puede aplicar esta última estrategia.

Las simetrías del problema pueden usarse como filtros adicionales a la hora de calcular las acciones posibles en el método *actions().*

Es muy conveniente encontrar los casos base. Como para ellos sabemos su solución, o sabemos que no tiene, incluimos esos casos en el método

actions(). Si el problema que es un caso base no tiene solución el conjunto de acciones estará vacío. Pero si tiene una o varias soluciones entonces escogemos sólo una de ellas: la óptima. El único vecino de un caso base con solución será un problema final.

El grafo debe ser completado con un vértice inicial, un predicado para los vértices finales, el tipo de los caminos (Sum o Last), el tipo de los problemas (Min, Max, One, All), la política voraz y la heurística. Estas propiedades tienen valores por defecto caso de no explicitarse.

En casos concretos puede hacer falta añadir información adicional como el peso de los vértices, las aristas o el paso por los vértices.

Como la variabilidad del número de parámetros del grafo puede ser grande usamos un *builder* para construirlo.

Problemas e Hypergrafos

En los casos más generales un vértice tiene varios vecinos tras tomar una alternativa. En este caso los grafos que se forman son *hipergrafos.* En nuestro caso las aristas de esos grafos (*hiperedges*) tiene un vértice fuente y varios destinos. Diseñamos *hipergrafos virtuales* que, de forma similar a los grafos virtuales, se concretan implementado el tipo de sus vértices, que deben extender *HyperVertex<V,E,A>,* y el tipo de sus aristas, que ahora deben extender *HyperEdge<V,A>.* Ahora el peso de las aristas vendrá definido como una función del peso de los vértices.

Estos tipos tienen los métodos:

```
interface HyperEdge<V extends HyperVertex<V, E, A, S>,
              E extends HyperEdge<V,E,A,S>, A, S>{
       V source();
       A action();
       Double weight(List<Double> solutions);
       S solution(List<S> solutions);
       E me();
       default List<V> targets() {
              return this.source().neighbors(this.action());
       }
       …
}
```

```
interface HyperVertex<V extends HyperVertex<V, E, A, S>,
              E extends HyperEdge<V,E,A,S>, A, S>{
       Boolean isValid();
       List<A> actions();
       List<V> neighbors(A a);
       E edge(A a);
       Boolean isBaseCase();
       Double baseCaseSolutionWeight();
       public S baseCaseSolution();
       public V me();
       …
}
```

Para estos problemas seguimos usando los conceptos de *problema inicial* y *caso base,* pero no el de problema final. Aquí no es tan relevante el concepto de vértices finales porque las soluciones buscadas son árboles cuyos vértices sin hijos son casos base y cuya raíz e el problema inicial. En los problemas que modelamos con grafos las soluciones son caminos que van del vértice inicial a uno final y los casos base se convierten en vértices con un solo hijo o ninguno.

Como hemos dicho en los problemas modelados con hipergrafos las soluciones son árboles cuyas hojas son casos base y cuya raíz es el problema a resolver. Esto cambia complemente el enfoque del problema. Las soluciones del problema son árboles con peso que modelamos mediante el tipo *GraphTree<V,E,A>.* Este tipo es un tipo recursivo con dos subtipos: *Gtb<V,E,A>, Gtr<V,E,A>* que representan, respectivamente, los árboles sin hijos y con hijos.

```
public sealed interface GraphTree
       <V extends HyperVertex<V, E, A, ?>,
       E extends HyperEdge<V, E, A,?>, A>
              permits Gtb<V,E,A>,Gtr<V,E,A>{

       V vertex() {
       A action();
       public default Double weight() {…}
       public default S solution() {…}
       public default List<GraphTree2<V, E, A, S>> children()
       {…}
       …
}
```

```
public static record Gtb<V extends HyperVertex2<V, E, A, ?>,
       E extends HyperEdge2<V, E, A, ?>, A>(V vertex)
              implements GraphTree2<V, E, A> {…}

public record Gtr<V extends HyperVertex2<V, E, A, ?>,
       E extends HyperEdge2<V, E, A, ?>, A>
       (V vertex, A action, List<GraphTree2<V,E,A>> children)
       implements GraphTree2<V, E, A> {…}
```

Vemos que un *GraphTree<V, E,A,S>* tiene un vértice y un peso y en el caso que tenga hijos además una acción y una lista de árboles hijos.

A partir de la información especificada para cada que concrete los tipos *HyperVertex* e *HyperEdge* podemos implementar un algoritmo genérico para resolver el problema que dividimos en varios pasos: encontrar la alternativa óptima para cada vértice, encontrar el árbol óptimo cuya raíz es el problema principal y por último encontrar la solución a partir del árbol óptimo. Los detalles de cada uno de slo pasos los veremos más abajo en la sección de *Implementación de la Programación Dinámica*.

Problema De Floyd

Como ejemplo de estas ideas veamos el problema de encotrar los caminos mínimos en un grafo *g* de tipo *Graph<V,E>* mediante el *Algoritmo de Floyd* que llamaremos el *Problema de Floyd.* Este problema define un *hipergrafo hg* con los datos del grafo de entrada y sobre él calcula el árbol mínimo.

Asumimos que cada vértice del grafo está indexado mediante un entero comprendido en $[0, n-1]$ dónde n es el número de vértices. Esto lo podemos conseguir obteniendo una vista adecuada con *IntegerVertexGraphView*. Queremos encontrar el *Camino Mínimo* entre dos ciudades dadas que representaremos por i, j. Este problema se puede resolver de muchas otras formas como hemos visto. Aquí vamos a resolver el problema generalizándolo a este otro: encontrar el *Camino Mínimo* de i a j usando como camino intermedio ciudades cuyos índices estén en el conjunto $[k, n)$. Con este planteamiento cada problema lo podemos representar por (i, j, k). Para que sea válido i, j toman valores en $[0, n-1]$ y k en $[0, n]$. El valor de $k = n$ indica que el camino intermedio no contiene ninguna ciudad. Podemos definir el predicado $isValid(i, j, k) \equiv i \in [0, n-1] \wedge j \in [0, n-1] \wedge k \in [0, n]$.

Ahora podemos modelar el problema mediante un *hipergrafo* cuyos vértices son los problemas generalizados (i, j, k). Las acciones (alternativas) posibles definirán las *hiperaristas*. Para cada problema tenemos dos alternativas $\{true, false\}$. La primera alternativa representa pasar por la ciudad k. La segunda no pasar.

Un vértice (i, j, k) será un caso base, que denotaremos por $b(i, j, k)$, si $k = n$, que indica que no pasamos por ninguna ciudad intermedia. El predicado b es, pues, $b(v, j, k) \equiv i = n$.

A partir de aquí podemos definir los vecinos del vértice (i, j, k) cuando se toma la alternativa a:

$$neighbors(i, j, k, a) = \begin{cases} [(i, j, k+1)], & a = false \\ [(i, k, k+1), (k, j, k+1], & a = true \end{cases}$$

Las acciones posibles son $A = [true, false]$.

Como podemos ver hemos obtenido un *hipergrafo* que llamaremos *hg.* Debemos hacer notar que es distinto al grafo g dado al principio. El hipergrafo *hg* definido es virtual en el sentido que sus vértices están bien definidos pero no tienen que estar en memoria. El conjunto de vértices el grafo está formado por las tripletas (i, j, k) tales que se cumple el predicado $isValid(i, j, k)$. Por otra parte a partir de un vértice se pueden alcanzar otros al tomar una acción determinada según indica la función $neighbors(a)$

Podemos asignar pesos a los vértices y las aristas del hipergrado definido. Como cada vértice tiene asociado un problema podemos asignarle un peso que el valor objetivo de la solución del problema. A cada arista le podemos asignar el peso de la solución obtenida si tomamos la alternativa asocida a la arista. Sean $ws(i, j, k)$ y $ws(i, j, k, a)$ los pesos del vértice (i, j, k) y de la solución si tomamos la alternativa a. Entre estos pesos existen relaciones recursivas.

Si $ws(i, j, k)$ es el peso óptimo de la función objetivo en el vértice $v = (i, j, k)$ y $ws(i, j, k, a)$ el peso de la arista que parte de v tomando la alternativa a entonces tenemos para este problema:

$$ws(i,j,k) = \begin{cases} s_b(i,j,k), & i = n \\ \min\limits_{a \in A} ws(i,j,k,a), & i \neq n \end{cases}$$

$$ws(i,j,k,a) = \begin{cases} ws(i,j,k+1), & a = false \\ ws(i,k,k+1) + ws(k,j,k+1), & a = true \end{cases}$$

Hay vértices cuya solución es conocida y por lo tanto su peso. Son los llamados casos base. Asumiendo que $w_g(i,j)$ es el peso de la arista (i,j) en el grafo dado g, los casos base y los pesos de sus soluciones, cuando las hay, son:

$$ws_b(i,j,k) = \begin{cases} \bot, & (i,j) \notin g, k = n \\ w_g(i,j), & (i,j) \in g, k = n \end{cases}$$

La arista óptima para un vértice (i,j,k), que representaremos por $ope(i,j,k)$, que no sea un caso base, es $ope(i,j,k) = \underset{a \in A}{argmin}\, w(i,j,k,a)$.

El arbol óptimo $opt(i,j,k)$ para un vértice (i,j,k) podemos definirlo de forma recursiva como:

$$s(i,j,k) = \begin{cases} [i,j], & b(v,j,k) \\ s(i,j,k+1), & !\, b(v,j,k) \wedge !\, ope(i,j,k) \\ s(i,k,k+1) + s(k,j,k+1), & !\, b(v,j,k) \wedge ope(i,j,k) \end{cases}$$

Donde $[i,j]$ es el formado por camino los vértices i,j y $s(i,k,k+1) + s(k,j,k+1)$ la concatenación de los caminos respectivos.

El arbol óptimo $opt(i,j,k)$ para un vértice (i,j,k) se define de forma similar como

$$opt(i,j,k) = \begin{cases} tb(i,j,k), & b(v,j,k) \\ tr(i,j,k,ope(i,j,k)), & !\, b(v,j,k) \end{cases}$$

Donde tb construye un árbol con un vértice que es caso base y tr un arbol definido por un vértice y una de sus acciones posibles.

Los tipos *FloydVertex* y *FloydEgde* implementan estas ideas. El tipo de las acciones es Boolean. El peso de las aristas se calcula a partir del peso de los vértices vecinos *(targets()).*

```
record FloydVertex(Integer i,Integer j,Integer k) implements
       HyperVertex<FloydVertex,FloydEdge,Boolean,
       GraphWalk<Integer,SimpleEdge<Integer>> >{
       public static FloydVertex of(
              Integer i,Integer j,Integer k) {
              return new FloydVertex2(i,j,k);
       }
       @Override
       public List<Boolean> actions() {
              if(this.isBaseCase()) return List.of();
              return List.of(false,true);
       }
       @Override
       public List<FloydVertex> neighbors(Boolean a) {
              List<FloydVertex> r=null;
              if(!a) r = List.of(FloydVertex2.of(i,j,k+1));
              else r = List.of(FloydVertex2.of(i, k, k+1),
                     FloydVertex2.of(k, j, k+1));
              return r;
       }
```

```
@Override
public FloydEdge edge(Boolean a) {
       return FloydEdge2.of(this,this.neighbors(a), a);
}
```

```
@Override
public Boolean isBaseCase() {
       return this.i.equals(this.j) ||  k == n;
}
```

```
@Override
public Double baseCaseSolutionWeight() {
       Double r = null;
       if(this.i.equals(this.j)) r = 0.;
       else if(k ==n && FloydVertex2.graph
              .containsEdge(this.i, this.j))
              r = FloydVertex2.graph.getEdge(i, j).weight();
       else if(k ==n && !FloydVertex2.graph
              .containsEdge(this.i, this.j))
              r = null;
       return r;
}
```

```
record FloydEdge(FloydVertex source,
       List<FloydVertex> targets,Boolean action) implements
              HyperEdge<FloydVertex,FloydEdge,Boolean,
              GraphWalk<Integer,SimpleEdge<Integer>> >{

       @Override
       public Double solutionWeight(List<Double> solutions) {
              Double weight = null;
              if (!action()) weight = solutions.get(0);
              else weight = solutions.get(0) +
                           solutions.get(1);
              return weight;
       }
}
```


Algoritmos voraces

Introducción

Anteriormente hemos visto el camino para obtener un grafo a partir de un modelo de un problema. La tarea ahora es encontrar el camino mínimo que va del vértice inicial al final. Los primeros algoritmos que estudiaremos son los algoritmos voraces, luego veremos los algoritmos *A*, Backtraking, Programación Dinámica de Reducción* y otros.

Un algoritmo voraz, *greedy* en inglés, es un algoritmo iterativo que partiendo del vértice inicial toma en cada vértice una sola de las alternativas posibles definida por una determinada política, que representaremos por el método *getAlternativa(e)* que depende del vértice, y continúa con el vértice vecino hasta que se cumple una condición de parada. La condición de parada en la mayoría de los casos es que se haya alcanzado un vértice final o no haya alternativas (aristas) disponibles.

El esquema sería:

```
V v = verticeInicial();
A a;
while (!condicionDeParada(v)) {
       a = getAlternativa(a)
       e = next(a);
}
return r(e);
```

Los algoritmos voraces puede que no encuentren ninguna solución al problema o si encuentran alguna que no sea la óptima.

A la alternativa que define la política de un algoritmo voraz la denominamos *alternativa voraz*, al camino recorrido, si encuentra el vértice final, *camino voraz* y a la solución asociada, si existe, *solución voraz* y el valor asociado de la función objetivo *valor voraz*. A la alternativa voraz hay asociada una *arista voraz* que es otra forma de definir la política del algoritmo voraz.

El valor óptimo de la función objetivo los llamaremos *valor óptimo*.

En un problema de minimización el valor voraz es una cota superior del valor óptimo. Este valor puede ser usado como valor inicial de la función objetivo en otras técnicas que veremos más adelante.

En un problema de maximización el valor voraz es una cota inferior del valor óptimo.

En el caso del problema de la mochila un algoritmo voraz podría ser:

- Ordenar la lista de objetos según su valor unitario. Es decir según la propiedad derivada v_i/w_i.
- Tomar en el grafo del problema que vimos la mayor de las alternativas enteras disponibles. Es decir, coger el máximo de unidades posibles del objeto *i.* Esta es la acción voraz que tiene asociada la arista voraz.
- Continuar el algoritmo hasta el problema final.
- El algoritmo devolverá el peso del camino recorrido si iniciamos una variable con valor cero y en cada paso la vamos actualizando con el peso de la arista si el camino es de tipo *Sum*. En el caso de caminos de tipo *Last* el último vértice nos dará el peso del camino.
- Si queremos obtener el camino, y no solamente el peso asociado a la solución voraz, entonces iniciamos un camino vacío y le vamos añadiendo las sucesivas aristas encontradas.

En este caso de la Mochila la arista voraz es:

```
MochilaEdge greedyEdge() {
       Integer a = Math.min(this.capacidadRestante/
              DatosMochila.getPeso(index),
              DatosMochila.getNumMaxDeUnidades(index));
       return MochilaEdge.of(this,this.neighbor(a), a);
}
```

Con esta arista el algoritmo para calcular el valor voraz es:

```
Double voraz(MochilaVertex v, MochilaVertex lastVertex) {
       Integer lastIndex = lastVertex.index();
       Double r = 0.;
       while (v.capacidadRestante()> 0 &&
                     v.index() < lastIndex) {
              Integer a = v.greedyEdge().action();
              r = r + a * DatosMochila.getValor(v.index());
              v = v.neighbor(a);
       }
       return r;
}
```

Dado un grafo extendido y una función que nos proporcione la arista voraz se puede programar de forma genérica la búsqueda del valor voraz o la solución voraz.

El algoritmo voraz se implementa como un iterador sobre los vértices del grafo. Este iterador tiene un estado que es un vértice del grafo y la arista de llegada a este vértice.

```
public class GreedyOnGraph<V,E> implements Iterator<Gog<V,E>>,
       Iterable<Gog<V,E>> {
       public static <V,E> GreedyOnGraph<V,E> of(
              EGraph<V,E> graph,Function<V,E> nextEdge) {
              return new GreedyOnGraph<V,E>(graph, nextEdge);
       }
       public static record Gog<V,E>(V vertex, E edge) {
              public static  <V,E> Gog<V,E> of(V vertex,
                            E edge){
                     return new Gog<V,E>(vertex,edge);
              }
       }
       private V state;
       private E edge;
       private EGraph<V,E> graph;
       private Function<V,E> nextEdge;
       …
```

El iterador continuará mientras el estado tenga alternativas disponibles y no sea un estado final.

```
@Override
public boolean hasNext() {
  return state != null && !this.graph.edgesOf(state).isEmpty()
       && !this.graph.goal().test(state);
}
```

El estado siguiente se consigue a través de la arista proporcionada por la función *nextEdge* y a partir de esta buscando el vértice opuesto al estado actual.

```
@Override
public Gog<V,E> next() {
       V old = state;
       edge= this.nextEdge.apply(state);
       if(edge !=null) this.state =
              graph.oppositeVertex(edge, old);
       else this.state = null;
       return Gog.of(old, edge);
}
```

Cualquier iterador lo podemos transformar en un stream con los métodos de la clase Stream2 que se encuentra en el respositorio. El iterador obtenido es un iterador de pares vértice-arista (el tipo *Gog<V,E>*).

```
Stream<Gog<V,E>> streamPair() {
       return Stream2.of(this);
}
```

Más detalles sobre la conversión de iteradores en stream y viceversa pueden encontrarse en el libro *Análisis y Diseño de Algoritmos y Tipos de Datos* también en esta colección.

A partir del stream de pares anterior podemos obener stream de vértices y aristas.

```
Stream<V> stream() {
       return streamPair().map(p->p.vertex());
}
Stream<E> streamEdges() {
       return streamPair().map(p->p.edge()).filter(e->e!=null);
}
```

A partir de los streams obenidos del iterador podemos conseguir el camino voraz partiendo del camino que contiene sólo el vértice inicial y añadiendo cada una de las aristas encontradas.

```
GraphPath<V,E> path(){
	EGraphPath<V,E> path = this.graph.initialPath();
	this.streamEdges().forEach(e->path.add(e));
	return path;
}
```

Como hemos comentado los algoritmos voraces puede que no encuentren ninguna solución. El camino que devuelven será solución si su último vértice es un vértice final que tiene solución. Es decir que cumpla el predicado *goalHasSolution()*.

```
public Boolean isSolution(GraphPath<V,E> gp) {
	V last = gp.getEndVertex();
	return graph.goal().test(last) &&
		graph.goalHasSolution().test(last);
}
```

En algunos casos puede ser útil disponer de caminos voraces que se obtengan con una política aleatoria: elegir aleatoriamente una arista de entre las opciones disponibles

```
<V,E> GreedyOnGraph<V,E> random(EGraph<V,E> graph) {
	Function<V,E> nextEdge =
		v -> graph.edgesListOf(v).isEmpty() ? null:

	List2.randomUnitary(graph.edgesListOf(v)).get(0);
		return new GreedyOnGraph<V,E>(graph,nextEdge);
}
```

La clase List2 se encuentra en el repositorio y ofrece algunos métodos de utilidad. En particular el método *randomUnitary* devuelve una lista con un solo elemento escogido aleatoriamente de otra lista de entrada.

Para el caso de la Mochila el uso de este algoritmo para obtener un camino voraz es:

```
SimpleVirtualGraph.endVertexG = v2;
EGraph<MochilaVertex,MochilaEdge> graph =
       SimpleVirtualGraph.sum(v1,MochilaVertex.goal(),
              x->x.weight());

GreedyOnGraph<MochilaVertex, MochilaEdge> gs =
       GreedyOnGraph.of(graph,MochilaVertex::greedyEdge);

GraphPath<MochilaVertex, MochilaEdge> gp = gs.path().get();
```

Los algoritmos voraces en general se pueden implementar sin partir de un grafo extendido. En ese caso los vértices se sustituyen por estados de tipo *E* y debemos contar con un operador unario que nos proporcione la política y un predicado que indique la condición de parada. El uso de estos algoritmos es de la forma:

```
Greedy<E> of(E start,UnaryOperator<E> next,Predicate<E> goal);
```

Su implementación se hace como un iterador que luego podremos transformar en un stream.

```
public class Greedy<E> implements  Iterator<E>, Iterable<E> {

       public static <E> Greedy<E> of(E start,
              UnaryOperator<E> next, Predicate<E> goal) {
              return new Greedy<E>(start, next, goal);
       }

       private E state;
       private UnaryOperator<E> next;
       private Predicate<E> goal;
       …
```

```
       @Override
       public boolean hasNext() {
              return !this.goal.test(state);
       }
       @Override
       public E next() {
              E old = state;
              state = this.next.apply(state);
              return old;
       }
```

Los algoritmos voraces pueden ser usados en muchos casos para calcular heurísticas. Para ello, debemos relajar el problema, eliminando algunas restricciones. Este nuevo problema relajado necesitará posiblemente un nuevo tipo de vértices, estados, y un conjunto de alternativas diferentes, que posibemente no respeten las restricciones del problema original. En el caso de la mochila con la política de escoger el entero $min(m_i, r/w_i)$, donde r es la capacidad restante entera, w_i también entera, obtenemos un algoritmo voraz. El peso del camino voraz obtenido nos vale como valor inicial de la función objetivo.

Busquemos ahora un algoritmo voraz para obtener una heurística. Escogemos un estado (i, r), con r real y como política el número real $\min(m_i, r/w_i)$. La división r/w_i ahora es real. Con el algoritmo voraz asociado obtenemos una heurística admisible. Esto significa escoger de cada objeto, ordenados éstos como hemos dicho, el máximo de unidades y fracciones de unidades. Podemos ver que esto nos va a dar un valor mayor o igual que el óptimo. Es una heurística admisible para un problema de maximización. Veamos cómo hacerlo.

```
record Md(Integer index, Double cr) {
       public static Md of(Integer index, Double cr) {
              return new Md(index,cr);
       }
       public Double heuristicAction() {
              return Math.min(cr()/
                     DatosMochila.getPeso(index()),
                     DatosMochila.getNumMaxDeUnidades(index()));
       }
       …
```

Implementamos un tipo que formado por un par entero-real que llamaremos *Md*. Este va a ser el estado del algoritmo voraz con métodos que calculen la acción heurística, el vértice siguiente según esa acción y el peso de la arista correspondiente.

```
        public Md next() {
                Double a = heuristicAction();
                return new Md(index()+1, cr()
                        - a * DatosMochila.getPeso(index()));
        }
        public Double weight() {
                if(this.index >= DatosMochila.n) return 0.;
                return heuristicAction()
                        *DatosMochila.getValor(index());
        }
}
```

Con esos elementos podemos implementar un algoritmo voraz que calcule la heurística.

```
public static Double hu(Md v1, Predicate<Md> goal) {
        Greedy<Md> r = Greedy.of(v1,v->v.next(),goal);
        return r.stream().mapToDouble(v->v.weight()).sum();
}
```

Anteriormente hemos comentado que una heurística debe tener prototipo determinado por lo que debemos ajustar el algoritmo *hu* anterior al prototipo exigido.

```
Double heuristic(MochilaVertex v1,
        Predicate<MochilaVertex> goal, MochilaVertex v2) {
        return hu(Md.of(v1.index(),
                (double)v1.capacidadRestante()),
                v->v.index()==v2.index()|| v.cr()==0.);
}
```

A la acción que define la política de un algoritmo voraz cumpliendo las restricciones del problema la llamamos *acción voraz*. Estas definen un algoritmo voraz que encuentra si existe una *solución voraz*. Son adecuadas para buscar un valor inicial de la función objetivo. Para estos algoritmos los estados son los vértices del grafo asociado al problema.

A la acción, que sin cumplir las restricciones del problema, define la política de un algoritmo voraz capaz de calcular una heurística admisible la llamamos *acción heurística*. Los algoritmos voraces asociados son adecuados para buscar una heurística. Para estos algoritmos hay que definir un estado adecuado.

Como hemos dicho los *valores voraces* pueden implementarse de forma genérica tomando como datos un grafo extendido (*graph*) y una política para elegir la acción (*nextEdge*).

Las *heurísticas* también pueden implementarse en muchos casos mediante un algoritmo voraz definido por un estado, un estado inicial, una función que da el estado siguiente y un criterio de parada.

Un primer criterio importante a tener en cuenta es que en los problemas de minimización se debe cumplir $h \leq v$ siendo h el valor de una heurística admisible y v un valor voraz. Si $h = v$ entonces la solución voraz es también una solución óptima.

Si el problema es de maximización entonces se debe cumplir $h \geq v$ siendo h el valor de una heurística admisible y v un valor voraz. Si $h = v$ entonces la solución voraz es también una solución óptima.

Orden de las aristas en un grafo virtual y solución voraz

Los grafos virtuales que diseñamos a partir de los modelos tienen vértices y aristas. Los vértices deben ser valores válidos del tipo correspondiente. Las aristas tienen asociada una de las acciones posibles en el vértice origen. Una acción es posible si asegura que el vértice destino es también válido.

Además, en muchos casos es conveniente dotar a las aristas de un orden para ser recorridas. Este orden está relacionado con la solución voraz en el sentido que la primera de las acciones debe ser la acción voraz.

En el caso del problema de la mochila hemos visto que un algoritmo voraz sería:

- Ordenar la lista de objetos según su valor unitario. Es decir según la propiedad derivada v_i/w_i
- Tomar la mayor de las alternativas enteras disponibles. Es decir, coger el máximo de unidades posibles del objeto *i*. Esta es la acción voraz que tiene asociada la arista voraz.

A partir de esta solución voraz podemos concluir que las acciones obtenidas en el método *actions()* del grafo virtual se deben devolver ordenadas de mayor a menor asumiendo que previamente hemos ordenado los objetos según su valor unitario. Es decir según la propiedad derivada v_i/w_i.

Esta estrategia hará que el camino desde el vértice inicial al final siguiendo para cada vértice la primera arista sea la solución voraz. Esto será importante en algunos algoritmos para filtrar alternativas que no conducirán a mejores soluciones.

Algoritmo A*

Los algoritmos *A** parten de un grafo, real o virtual, un vértice inicial, un vértice final y una heurística. Buscan el camino óptimo del vértice inicial al final. El peso del camino está asociado al grafo extendido de partida.

Introducción e implementación

El algoritmo *A** utiliza una función de evaluación $f(v)$ que representa el peso del camino desde el vértice inicial hasta el final pasando por el vértice v (el peso del camino se calcula de forma diferente según sea tipo *Sum* o tipo *Last* según vimos anteriormente). En ese cálculo interviene $h(v)$ que representa la heurística desde el vértice actual v hasta el final, y $g(v)$, el peso del camino recorrido para llegar a dicho nodo v desde el nodo inicial. En el caso de tipo *Sum* $f(v) = g(v) + h(v)$ y en el caso *Last* $f(v) = h(v)$.

Como camino óptimo podemos entender que sea de peso mínimo o máximo. Asumiremos en general que es mínimo. Si es máximo lo transformaremos en mínimo por la propiedad $max\ f(x) = min - f(x)$. La heurística tendrá que ser admisible para el caso de minimizar o de maximizar.

Los algoritmos *A** son algoritmos iterativos y como ellos tienen un estado, una función que nos proporciona el siguiente estado y un criterio de

parada. Como los algoritmos iterativos se pueden implementar como un iterador y además proporcionan el camino mínimo exacto. Veamos sus diferentes elementos.

El estado del algoritmo *A** mantiene, para cada vértice encontrado, información de su distancia al origen, la arista en la que comienza el camino mínimo y si está cerrado o no. Para organizar esta información diseñamos, en primer lugar, el tipo *Data* que es un *record* de la forma:

```
record Data<V, E> (V vertex, E edge, Double distanceToOrigin,
       Boolean closed) {

       public static <V, E> Data<V, E> of(
              V vertex, E edge, Double distance) {
                 return new Data<>(vertex,edge,distance,false);
       }
       public static <V, E> Data<V, E> toTrue(Data<V, E> d) {
              return new Data<>(d.vertex, d.edge,
                     d.distanceToOrigin,true);
       }
}
```

Por simplicidad al tipo Data lo llamaremos *D*. Vemos que los valores de este tipo son una tupla de la forma $D = (V, E, W, C)$ dónde V es un vértice, E la arista que indica su camino hacia el origen, W *(*de tipo *Double)* su distancia hasta el origen siguiendo la arista anterior (es la cantidad $g(v)$ anterior) y C un booleano que indica si el vértice está cerrado o no. El concepto de abierto o cerrado los explicaremos más abajo.

Las tuplas de valores de tipo D queremos mantenerlas ordenadas en conjunto según los valores $f(v)$, el peso del camino desde el vértice inicial hasta el final pasando por el vértice v, de cada vértice. Para ello usamos un tupla llamada *Handle<Double,D>* que por simplicidad llamaremos el tipo *H*.

Por simplicidad en la exposición si *h* es una tupla de tipo H y *d* es una tupla de tipo *D* representaremos, respectivamente por *d.v, d.e, d.w, d.c*, la primera, la segunda, la tercera y cuarta componente de d. Es decir *d.v, d.e, d.w, d.c* se obtendrán mediante los métodos *vertex(), edge(), distanceToOrigin() y closed()* aplicados a un objeto de tipoD.

Igualmente con *h.t, h.d.* En el código *h.t, h.d* se obtendrá mediante los métodos *getKey()* y *getValue()* aplicados a un objeto de tipo *H*.

El tipo *H (Handle)* es mutable y tiene los métodos:

- *h.key():* Devuelve *h.t*
- *h.value():* Devuelve *h.d*
- *h.decreaseKey(t):* Cambia el valor de la componente *h.t*. El montón donde está el objeto *h* queda reordenado.

Un *Montón de Fibonacci (Fibonacci<Double,D>)* es un conjunto de valores de tipo *H* ordenados según la componente demenor a mayor *h.t* (el valor $f(v)$ de cada vértice) y con la peculiaridad de que es reordenable (podemos modificar *h.t* y el conjunto quedará ordenado de nuevo). Un *montón de Fibonacci* es por lo tanto una cola de prioridad reordenable de valores de tipo *H,* ordenados según la componente *h.t* y tiene las operaciones

- *f.deleteMin():* Saca del montón y devuelve la tupla *h* (de tipo *Handle*) con peso mínimo.
- *f.insert(t,d):* Añade al montón la tupla *(t,d)* y devuelve la correspondiente tupla de tipo *H* (*Handle*)

El estado de los algoritmos *A** contiene un *Map<V,H>* con los vértices previamente encontrados, su camino mínimo hacia el origen y su distancia mínima al origen y un montón de *Fibonacci<Double,D>*.

Los *vértices encontrados* serán las claves del *Map<V,H>.* Los *vértices abiertos* aquellos cuya información, de tipo D, asociada está en el Montón de Fibonacci. Los *vértices cerrados* el conjunto de las claves del *Map* menos los abiertos, es decir el conjunto de los vértices encontrados que ya han sido visitados y por lo tanto sacados del *Monton de Fibonacci*.

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

El estado inicial contiene el vértice inicial, el peso del camino inicial y la estimación del peso hasta el final.

$$m = \{v_0 : h_0\}, f = \{h_0\}, h_0 = (t_0, d_0), d_0 = (v_0, -, w_0, false)$$

Siendo v_0 el vértice inicial, t_0 la estimación del peso del camino hasta el final y w_0 el peso del camino formado por el vértice inicial.

La implementación del algoritmo *AStar* es de la forma:

```
public class AStar<V,E> implements Iterator<V>, Iterable<V> {

     public static <V, E> AStar<V, E> of(EGraph<V, E> graph,
             Double bestValue,GraphPath<V, E> optimalPath) {
             return new AStar<V,E>(
                    graph,bestValue,optimalPath);
     }

     public static Comparator<Double> comparator =
       Comparator.naturalOrder();

     protected EGraphPath<V,E> ePath;
     public Map<V,Handle<Double,Data<V,E>>> tree;
     public FibonacciHeap<Double,Data<V,E>> heap;
     private Double bestValue = null; //mejor valor estimado
     private GraphPath<V, E> optimalPath = null;
             //mejor camino estimado
```

Handle implementa el tipo *H* anterior, *Data* el tipo D y *FibonacciHeap* el montón de Fibonacci. El constructor crea el estado inicial del algoritmo.

```
AStar(EGraph<V, E> graph, Double bestValue,
              GraphPath<V, E> optimalPath) {
       this.graph = graph;
       AStar.comparator = AStar.type.equals(AStarType.Min)?
         Comparator.naturalOrder():Comparator.reverseOrder();
       this.tree = new HashMap<>();
       this.ePath = graph.initialPath();
       this.heap = new FibonacciHeap<>(comparator);
       Data<V,E> data =
         Data.of(startVertex,null,ePath.getWeight());
       Double d = graph.estimatedWeightToEnd(startVertex,
         data.distanceToOrigin,goal,end,heuristic);
       Handle<Double, Data<V, E>> h = this.heap.insert(d,data);
       this.tree.put(startVertex,h);
       this.bestValue = bestValue;
       this.optimalPath = optimalPath;
}
```

Si queremos recorrer todos los vértices del grafo la condición de parada es que el montón de Fibonacci esté vacío. Es el método *hasNext* del iterador.

```
public boolean hasNext() {
	return !heap.isEmpty();
}
```

Veamos la función que transforma un estado en el siguiente. Es decir, el método *next* del iterador.

En cada paso del algoritmo se saca el nodo que esté primero en *abiertos*, el montón de Fibonacci, recorre sus vecinos y los inserta en en el map y en el montón junto con las propiedades calculadas: distancia al origen, arista hacia el origen y distancia estimada al final.

```
@Override
public V next() {
	Handle<Double, Data<V, E>> hActual = heap.deleteMin();
	Data<V, E> dActual = hActual.getValue();
	V vertexActual = dActual.vertex;
	Double actualDistance = dActual.distanceToOrigin;
	E edgeToOrigen = dActual.edge;
	if(forget(actualDistance,  vertexActual)) return null;
	for (E backEdge : graph.edgesListOf(vertexActual)) {
	   V v = Graphs.getOppositeVertex(
	             graph,backEdge,vertexActual);
	   Double newDistanceToOrigin = graph.add(v,
	             actualDistance,backEdge,edgeToOrigen);
	   Double newDistanceToEnd= graph.estimatedWeightToEnd(
	       v,newDistanceToOrigin, goal, end, heuristic);
	   if (!tree.containsKey(v)) {
	       Data<V,E> dv=Data.of(v,backEdge,
	             newDistanceToOrigin);
	       Handle<Double, Data<V, E>> hv =
	             heap.insert(newDistanceToEnd, dv);
	       tree.put(v, hv);
	       … // sigue abajo
```

Si alguno de sus vecinos ya había sido encontrado antes, estará en el map y allí estará el camino mínimo al origen. En este caso buscamos su distancia al origen según el camino anteriormente encontrado y, si la nueva distancia es menor, actualizamos el map con el nuevo camino

mínimo al origen y el montón con la nueva distancia estimada hasta el final.

```
            } else if (comparator.compare(newDistanceToOrigin,
                  tree.get(v).getValue().distanceToOrigin())<0){
               Data<V, E> dv = Data.of(v, backEdge,
                     newDistanceToOrigin);
               Handle<Double, Data<V, E>> hv = tree.get(v);
               hv.setValue(dv);
               hv.decreaseKey(newDistanceToEnd);
            }
         }
         return vertexActual;
}
```

El algoritmo descarta la expansión de los vértices para los que podemos estimar que no son útiles para encontrar un camino óptimo. Esto se consigue con la llamada al método forget.

```
Boolean forget(Double actualDistance, V v) {
         Double w = graph.estimatedWeightToEnd(v,actualDistance);
         Boolean r = false;
         r = this.bestValue != null &&
               comparator.compare(w,this.bestValue) >= 0;
         if(r) this.tree.remove(v);
         return r;
}
```

Como vemos el algoritmo A*, con lo explicado hasta ahora, parará cuando quede vacío el Montón de Fibonacci. En ese caso recorrerá todos los vértices del grafo en el orden establecido: se visita antes cuyo peso estimado del camino desde el vértice inicial a un final pasando por él sea menor.

Pero así sin más el algoritmo sería muy lento. Tal como explicaremos abajo, y esta es la gran ventaja del algortimo A*, podemos parar cuando encontremos el primer vértice final que tenga solución (que cumpla el método *goalHasSolution()*).

Para implementar esa idea obtenemos un stream a partir del iterador:

```
public Stream<V> stream() {
         return Stream2.of(this);
}
```

Y, posteriormente, encontramos el primer vértice que cumpla el requisito pedido: vértice final que tenga solución.

```
Optional<V> last = this.stream()
       .filter(v -> v != null)
       .filter(graph.goal().and(graph.goalHasSolution()))
       .findFirst();
```

Posteriormente habrá que recontruir el camino mínimo desde el vértice encontrado hasta el vértice inicial siguiendo las aristas guardadas en la componente d.e asociada a cada vértice en el *Map<V,H>.*

La arista hacia el origen en el camino mínimo desde cada vértice la obtenemos:

```
public E getEdgeToOrigin(V v) {
       return tree.get(v).getValue().edge;
}
```

Un breve resumen de esa implementación es:

```
…
V v = endVertex;
if (!tree.containsKey(v)) return Optional.empty();
Handle<Double, Data<V, E>> hav = this.tree.get(v);
Data<V, E> dav = hav.getValue();
Double weight = dav.distanceToOrigin;
E edge = dav.edge;
List<E> edges = new ArrayList<>();
while (edge != null) {
edges.add(edge);
v = Graphs.getOppositeVertex(graph, edge, v);
edge = this.getEdgeToOrigin(v);
}
Collections.reverse(edges);
…
```

El algoritmo nos devuelve un *GraphPath<V,E>* que posteriormente habrá que transformar en la solución buscada.

Los detalles del código pueden verse en el repositorio donde se ha usado un montón de Fibonacci de la librería *org.jheaps.tree.FibonacciHeap.* Para

obtener una instancia del algoritmo podemos hacerlo con los métodos de factoría siguientes:

```
AStar<V, E> of(EGraph<V, E> graph) {
              return new AStar<V, E>(graph,null,null);
}
AStar<V, E> of(EGraph<V, E> graph,
       Double bestValue,GraphPath<V, E> optimalPath) {
       return new AStar<V, E>(graph,bestValue,optimalPath);
}
```

En muchos casos puede ser interesante visualizar los vértices y aristas que han sido alcanzados por el algoritmo. Estos vértices y aristas forman un grafo que puede ser obtenido por el método *graph()* anterior del tipo *AStar*.

```
public SimpleDirectedGraph<V,E> graph(){
       SimpleDirectedGraph<V,E> g =
                     Graphs2.simpleDirectedGraph();
              for(V v:tree.keySet()) {
                     g.addVertex(v);
              }for(V v:tree.keySet()) {
                     E e = tree.get(v).getValue().edge();
                     if (e != null) {
                            V source = graph.getEdgeSource(e);
                            V target = graph.getEdgeTarget(e);
                            g.addEdge(source, target, e);
                     }
              }
              return g;
}
```

En muchos casos podemos combinar el algoritmo A* con un algoritmo que no proporcione una solución voraz. Para ello podemos usar el siguiente método de factoría:

```
AStar<V, E> of(EGraph<V, E> graph) {
       GreedyOnGraph<V, E> ga = GreedyOnGraph.of(graph);
       Optional<GraphPath<V, E>> gp = ga.solutionPath();
       if(gp.isPresent()) return Astar.of(graph,gp.get().
              getWeight(),gp.get());
       else return Astar.of(graph,null,null);
}
```

Corrección del algoritmo y filtrado de vértices

El algoritmo mantiene un invariante:

- Los vértices encontrados y todavía no visitados, los abiertos, se mantienen ordenados en el montón según el peso del camino estimado hasta el final que pasa por ellos.
- Los vértices encontrados tienen asociado en el map el camino mínimo hasta el origen pasando por los vértices encontrados.

En cada paso del algoritmo, el nodo con el valor $f(v)$ más bajo se elimina del montón, los valores f, g de sus vecinos se actualizan y si estos vecinos si no estaban se agregan al montón.

Cuando *A** termina su búsqueda, ha encontrado una ruta desde el principio hasta un vértice final cuyo coste real es menor o igual que el costo estimado de cualquier camino desde el inicio hasta el final a través de cualquier vértice abierto. Cuando la heurística es admisible, el algoritmo A * puede ignorar los nodos abiertos con seguridad porque no es posible que conduzcan a una solución de coste más bajo que la que ya obtenida.

Veamos la justificación de lo anterior.

Supongamos que el algoritmo *A** usa una heurística admisible, y que *p* corresponde a una solución óptima. Es decir, es *p* es un vértice final y *g(p)* es minimal. Supongamos también que *m* es un vértice correspondiente a una solución subóptima. Esto es, *m* es final y *g(m) > g(p).* Justificaremos que *m* nunca será seleccionado del montón y por lo tanto lo anterior no podrá ocurrir.

Ha de existir un vértice *n* en el montón (vértices abiertos) que pertenece al camino óptimo que acaba en *p*. Como *h* es admisible y tanto *n* como *p* están en el mismo camino a la solución, entonces:

$$f(n) = g(n) + h(n) \leq g(n) + h^*(n) = f^*(n) = f^*(p)$$

Si el algoritmo *A** selecciona *m* antes que *n*, entonces *f(m) ≤ f(n)* y por tanto $f(m) \leq f(n) \leq f^*(p)$. Como m y p son finales $h(m) = h(p) = 0$ por lo

que $f(m) = g(m), f^*(p) = g(p)$. Combiando lo anterior concluimos que $g(m) \leq g(p)$.

Pero habíamos supuesto que $g(m) > g(p)$, que está en contradicción con lo anterior. Luego el algoritmo A^* nunca selecciona del montón un vértice correspondiente a una solución subóptima. En otras palabras, si un vértice final es seleccionado del montón (y por tanto devuelto como solución) entonces es una solución óptima.

El algoritmo es una combinación entre búsquedas del tipo primero en anchura con primero en profundidad: mientras que $h(v)$ tiende a seguir primero en profundidad, $g(v)$ tiende a seguir primero en anchura. De este modo, se cambia de camino de búsqueda dependiendo de los valores de $h(v)$, $g(v)$. Es un algoritmo que va haciendo una búsqueda primero el que tiene mejor valor de $f(v)$.

Como todo algoritmo de búsqueda en amplitud, A* es un algoritmo completo: en caso de existir una solución, siempre dará con ella. Si se cumple $g(v) = 0$, nos encontramos ante una búsqueda voraz. Si se cumple $h(v) = 0$, A* pasa a ser una búsqueda en anchura no informada.

Para garantizar la optimimalidad del algoritmo, la función $h(v)$ debe ser una heurística admisible. De no cumplirse dicha condición, el algoritmo pasa a denominarse simplemente A, y, a pesar de seguir siendo completo, no se asegura que el resultado obtenido sea el camino de coste mínimo.

Como se ha mencionado anteriormente $h(n)$ es una estimación de la distancia al nodo objetivo, un vértice final. Por lo tanto

- Si $h(v)$ hace una estimación perfecta, A* converge inmediatamente al objetivo.
- Si $h(v) = 0$, la función $g(n)$ controla la búsqueda siguiendo por el siguiente más cercano al origen.
- Si $h(v) = 0$, $g(v) = 0$ la búsqueda será aleatoria.
- Si $h(v) = 0$ la búsqueda será primero en anchura.

Las propiedades anteriores son para caminos de tipo *Sum*. Existen unas equivalentes para caminos de tipo *Last*.

Las propiedades anteriores permiten filtrar (olvidar) algunos vértices cerrados si conocemos una solución voraz y por lo tanto un primer valor de la función objetivo. Sean *GraphPath<V, E> optimalPath* y *Double bestValue* una solución voraz y el valor de la función objetivo en esa solución. Sea v un vértice cerrado. Sabemos que *g(v)* es el peso del camino más corto al origen desde ese vértice ya ha sido obtenido cuando el vértice se saca del montón y pasa a cerrado. Por otra parte, para caminos Sum, *f(v)=g(v)+h(v)* y como *h(v)* es admisible se cumple *f(v) = g(v)+h(v)* $\leq f^*(v) = g(v^*)$. Por otra parte *bestValue* $\geq f^*(v)$ siendo $f^*(v) = g(v^*)$ el valor óptimo de la función objetivo y v^* un vértice final que es el extremo del camino óptimo. Por lo tanto podemos concluir que si *f(v)*$\geq$ *bestValue.* El vértice v puede ser olvidado porque siguiendo a partir de él obtendremos soluciones peores que la voraz. Olvidado quiere decir que no recorremos sus vecinos y lo eliminamos de las claves del map que guarda los vértices ya encontrados.

Una posibilidad es añadir una línea en el código de AStar antes de recorrer a sus vecinos:

```
if(forget(actualDistance,  vertexActual)) return null;
```

El método forget es de la forma:

```
Boolean forget(Double actualDistance, V v) {
       Double w = ePath.estimatedWeightToEnd(actualDistance,v,
              graph.goal(), graph.endVertex(), heuristic);
       Boolean r = false;
       r = this.bestValue != null &&
              comparator.compare(w,this.bestValue) >= 0;
       if(r) this.tree.remove(v);
       return r;
}
```

Este diseño permite además de filtrar partes del grafo disminuir la memoria que está siendo usado.

Uso de los algoritmos A*

Veamos cómo usarlos en el problema de la Mochila ya visto anteriormente.

En primer lugar, creamos los vértices inicial y final y un *EGraph<MochilaVertex, MochilaEdge>.* Es decir, creamos un grafo extendido tal como hemos explicado antes. Con el método siguiente creamos el grafo a partir de un grafo virtual definido por los tipos *MochilaVertex, MochilaEdge*:

```
DatosMochila.capacidadInicial = 78;
MochilaVertex e1 = MochilaVertex.of(78);
MochilaVertex e2 = MochilaVertex.lastVertex();
EGraph<MochilaVertex, MochilaEdge> graph =
EGraph.virtual(e1,MochilaVertex.goal(),PathType.Sum, Type.Max)
       .greedyEdge(MochilaVertex::greedyEdge)
       .heuristic(MochilaHeuristic::heuristic)
       .build();
```

El grafo usa una función para definir los pesos de las aristas que en este caso depende del peso definido en *MochilaEdge*. El grafo tiene caminos de tipo *Sum*.

Hemos de hacer notar que pasamos como parámetro la heurística con el método que hemos diseñado, *MochilaHeuristic::heuristic*.

```
public static Double heuristic(MochilaVertex v1,
        Predicate<MochilaVertex> goal, MochilaVertex v2) {
             return heuristic(v1,v2);
}
```

Obtenemos una instancia del algoritmo *A**:

```
AStar<MochilaVertex, MochilaEdge> ms =
       GraphAlg.aStar(graph,bv,gp);
```

Finalmente pedimos al algoritmo que busque el camino mínimo y nos devuelve un camino a partir del cual calculamos la solución. Poniéndolo todo junto tenemos:

```
DatosMochila.iniDatos("ficheros/objetosMochila.txt");
DatosMochila.capacidadInicial = 78;
MochilaVertex e1 = MochilaVertex.of(78.);
MochilaVertex e2 = MochilaVertex.lastVertex();
```

```
EGraph<MochilaVertex, MochilaEdge> graph =
EGraph.virtual(e1,MochilaVertex.goal(),PathType.Sum, Type.Max)
       .greedyEdge(MochilaVertex::greedyEdge)
       .heuristic(MochilaHeuristic::heuristic)
       .build();

GreedyOnGraph<MochilaVertex, MochilaEdge> rr =
       GreedyOnGraph.of(graph);

GraphPath<MochilaVertex, MochilaEdge> gp = rr.path();

AStar<MochilaVertex, MochilaEdge> ms =
       AStar.of(graph,gp.getWeight(),gp);

GraphPath<MochilaVertex,MochilaEdge>> path=ms.search().get();

List<MochilaEdge> edges = path.getEdgeList();
SolucionMochila s = MochilaVertex.getSolucion(edges);
System.out.println(s);
```

El algoritmo de *Dijsktra* es un caso particular del *A** cuando la heurística es 0 para todos lo vértices en problemas cuyo camino es de tipo *Sum*.

Problema de tareas y procesadores

Como un segundo ejemplo veamos el problema de las tareas y los procesadores. El problema se formula de la siguiente manera: dada una lista de n tareas con duraciones d_i y un conjunto de m procesadores buscar la asignación de tareas a procesadores tal que el tiempo total de ejecución sea mínimo.

Este problema ya lo resolvimos mediante programación lineal entera y algoritmos genéticos. Ahora queremos resolverlo mediante algoritmos *A**. Un modelo adecuado es:

$$\min \max_{j:0..n-1} \Sigma_{i=0|x_i=j}^{n-1} d_i$$
$$0 \leq x_i < m, \quad i \in [0, n-1]$$
$$int\ x_i, \quad i \in [0, n-1]$$

En este segundo modelo la variable x_i indica el procesador al que se asigna la i.

Como ya vimos en este problema podemos considerar caminos de tipo *Last* con el peso de los caminos definido por la propiedad *cm* del último vértice.

Generalizamos el problema con las propiedades *i, cargas* que ya vimos.

Solución Voraz: Escoger la acción $a = nMin$. La solución encontrada nos proporcionará una cota superior de la función objetivo óptima y puede servirnos como su valor inicial.

Heurística: Propiedad *cm*: carga del procesador más cargado. Esta es una heurística válida porque los siguientes vértices tendrán esta propiedad igual o mayor.

Con los datos anteriores implementamos la clase de los vértices:

```
record TyPVertex(Integer index, List<Double> cargas)
          implements ActionVirtualVertex<TyPVertex,
             ActionSimpleEdge<TyPVertex,Integer>, Integer>{
       …
}
```

Las aristas son de clase ActionSimpleEdge<TyPVertex,Integer>.

Finalmente instanciamos el algoritmo A* y lo ejecutamos.

```
TyPVertex.datos("ficheros/tareas.txt",5);
TyPVertex e1 = TyPVertex.first();
TyPVertex e2 = TyPVertex.last();
Predicate<TyPVertex> goal = v->v.getIndex()==TyPVertex.n;

EGraph<TyPVertex,SimpleEdgeAction<TyPVertex,Integer>> graph =
    SimpleVirtualGraph.last(e1,v->v.goal(),v->v.maxCarga());

AStar<TyPVertex,SimpleEdgeAction<TyPVertex, Integer>> ms =
      GraphAlg.aStar(graph, Heuristica::heuristic,
            AStarType.Min)

GraphPath<TyPVertex,SimpleEdgeAction<TyPVertex,Integer>> path
      = ms.search().get();

SolucionTyP s = TyPVertex.getSolucion(path);
===
public class Heuristica { Double heuristic(TyPVertex v1,
      Predicate<TyPVertex> p, TyPVertex v2) {
            return v1.maxCarga();
      }
}
```

Implementación directa del algoritmo A*

Es también conveniente aprender a implementar este algoritmo para un problema concreto sin la necesidad de usar un grafo virtual, aunque siempre recomendamos hacer previamente un modelo del problema mediante un grafo virtual. Esto nos ayudará a encontrar el tipo de los problemas, las acciones para pasar a otros, y el peso de los caminos.

El primer paso es diseñar el tipo de datos que concretará la tupla *D* que hesmos visto anteriormente en la implementación genérica del algoritmo. Lo más simple es mediante un record:

```
record AStarMochila(MochilaProblem vertex,Integer a,
            MochilaProblem lastVertex,
            Double distanceToOrigin) {
      AStarMochila of(MochilaProblem vertex, Integer a,
                  MochilaProblem lastVertex,
                  Double distanceToOrigin) {
      return new AStarMochila(vertex, a,
            lastVertex,distanceToOrigin);
      }
}
```

El tipo MochilaProblem se diseña de forma similar al vértice del grafo comentado más arriba, pero sin estar obligado a implementar ninguna interfaz.

```
record MochilaProblem(Integer index, Integer
       capacidadRestante) {
       public MochilaProblem vecino(Integer a) { … }
       public List<Integer> acciones() { … }
```

El algoritmo AstarMochila que implementa el método search() que devuelve el número de unidades de cada uno de los objetos sigue los pasos del algoritmo genérico visto arriba. Pero no vamos a implementar un iterador. Simplemente es un algoritmo iterativo con un while y un estado. El estado es:

```
public class MochilaAStar {
public MochilaProblem start;
public Map<MochilaProblem,Handle<Double,AStarMochila>> tree;
public FibonacciHeap<Double,AStarMochila> heap;
public Boolean goal;
```

Map<MochilaProblem,Handle<Double,AStarMochila>>: Un *Map* que asocia a cada vértice una tupla con un valor real y otra tupla de tipo *AstarMochila.* Guarda los datos de los vértices ya visitados.

FibonacciHeap<Double,AStarMochila>: Un montón de Fibonacci de tuplas, se tipo AStarMochila, ordenadas según la primera componente.

Las operaciones con el montón devuelve tuplas del tipo *Handle<Double, AStarMochila>* con propiedades *getKey(), getValue()* y operaciones *setValue(V value)*, *decreaseKey(K newKey).*

```
public List<Integer> search() {
       MochilaProblem vertexActual = null;
       while (!heap.isEmpty() && !goal) {
              Handle<Double, AStarMochila> ha
                     = heap.deleteMin();
              AStarMochila dataActual = ha.getValue();
              vertexActual = dataActual.vertex();
              …
```

Como se ha descrito arriba, el algoritmo comienza con un montón de Fibonacci que contiene únicamente el vértice inicial y continua hasta que está vacío o se ha alcanzado un vértice final. Los valores en el montón de

Fibonacci son tuplas de tipo *AStarMochila* ordenadas por la distancia estimada, según la heurística, del camino completo del vértice inicial al final. En cada paso se saca del montón la tupla que tenga asociada la mínima distancia. De aquí se obtiene el vértice actual.

```
            for (Integer a : vertexActual.acciones()) {
                  MochilaProblem v = vertexActual.vecino(a);
                  Double newDistance =
                        dataActual.distanceToOrigin() -
                              a*DatosMochila.valor(
                                    vertexActual.index());
                  Double newDistanceToEnd = newDistance -
                        Heuristica.heuristica(v);
```

Para cada arista saliente del vértice actual, se busca el vértice opuesto al actual y si comprueba si la distancia del vértice opuesto al origen es menor a través del vértice actual. Si lo es actualiza la distancia mínima del vértice opuesto al origen y la arista que comienza el camino hacia él. Si el vértice opuesto no tenía todavía una distancia al origen se actualiza con la nueva distancia a través del vértice actual.

```
                  if (!tree.containsKey(v)) {
                        AStarMochila ac =
                              AStarMochila.of(v, a,
                              vertexActual, newDistance);
                        Handle<Double, AStarMochila> nh
                              = heap.insert(
                                    newDistanceToEnd,ac);
                        tree.put(v,nh);
```

```
                  }else if (newDistance < tree.get(v)
                        .getValue().distanceToOrigin()) {
                        AStarMochila ac
                              = AStarMochila.of(v, a,
                              vertexActual, newDistance);
                        Handle<Double, AStarMochila> hv
                              = tree.get(v);
                        hv.setValue(ac);
                        hv.decreaseKey(newDistanceToEnd);
                  }
            }
            this.goal = vertexActual.index() ==
                        DatosMochila.n;
      }
      return acciones(vertexActual);
}
```

```
List<Integer> acciones(MochilaProblem v) {
	List<Integer> ls = new ArrayList<>();
	Integer a = this.tree.get(v).getValue().a();
	while (a != null) {
		ls.add(a);
		v = this.tree.get(v).getValue().lastVertex();
		a = this.tree.get(v).getValue().a();
	}
	Collections.reverse(ls);
	return ls;
}
```

Los caminos mínimos hasta el origen se mantienen en el *Map*. A partir de él, se pueden obtener la lista de unidades de cada objeto con el método acciones.

Esquemas recursivos: Backtracking

Introducción e implementación

Los algoritmos de *backtracking* necesitan un grafo y una definición del peso asociado a un camino. Ese peso del camino puede tener una definición muy general como ya hemos visto. Los vértices deben tener asociado un tamaño y es necesario que los vecinos de un vértice tengan un tamaño menor. Se necesitan, además, un vértice inicial, un vértice final y/o la definición de casos base, y es importante disponer de una heurística, aunque no es necesario.

Los algoritmos de *backtracking* son adecuados para buscar un camino mínimo o máximo, también para contar el número de soluciones y para encontrarlas todas o un número dado de ellas si las hay.

El algoritmo de *backtracking* va recorriendo el grafo en profundidad de forma recursiva, desde el vértice inicial. En cada vértice *v*, el algoritmo comprueba si v es un vértice final. Si no es un vértice final enumera recursivamente todos los sus vecinos. Los vértices vecinos que no satisfagan un criterio de filtro son descartados.

Por lo tanto, el algoritmo recorre solo una parte del grafo. El costo total del algoritmo es el número de vértices visitados multiplicado por el costo de obtener y procesar cada vértice.

La implementación puede hacerse de forma genérica. Veamos aquí las ideas generales de la implementación.

En primer lugar diseñamos un el tipo mutable *State<V,E>* del cual vamos a disponer de un solo objeto que llevará el control del algoritmo. Este objeto mantiene el vértice actual y el camino hasta él desde el vértice inicial. Este objeto se actualiza cuando pasamos a un vértice vecino o volvemos al vértice anterior.

```
interface State<V, E> {
	void forward(E edge);
	void back(E edge);
	Double getAccumulateValue();
	EGraphPath<V, E> getPath();
	EGraph<V, E> getGraph();
	V getActualVertex();
}
```

- *V getActualVertex():* El vértice actual
- E*GraphPath<V, E> path*: El camino desde el vértice inicial al vértice actual
- *Double accumulateValue()*: El valor acumulado de la función objetivo: coste de *path*.
- *void forward(E edge):* Actualiza el estado según la arista *edge*. Es decir, actualiza el vértice actual según la arista *edge*, el peso y el camino hacia el origen.
- *void back(E edge):* Actualiza según la arista *edge* para volver al estado anterior. Es decir, actualiza el vértice actual según la arista *edge*, el peso y el camino hacia el origen.

Podemos construir una instancia del *algoritmo de backtracking* con el método:

```
BT<V, E, S> of(
	EGraph<V, E> graph,
	Function<GraphPath<V, E>, S> solution,
	Double bestValue,
	GraphPath<V,E> optimalPath,
	Boolean withGraph);
```

En muchos casos nos interesa representar el grafo de los vértices y aristas alcanzados por el algoritmo. La propiedad *withGraph* será *true* o *false*

según queramos o no guardar la información necesaria para mostrar el grafo de ejecución del algoritmo. Junto con las propiedades incluidas en el método anterior el algoritmo tiene las propiedad pública *solutions* que guardará el conjunto de soluciones encontradas cuando queremos buscar más de una.

```
public Set<S> solutions;
```

Las propiedades públicas son:

- *Double bestValue*: Mejor valor encontrado de la función objetivo. Esta propiedad puede ser inicializada mediante un algoritmo voraz previo.
- *GraphPath<V,E> optimalPath:* El mejor camino encontrado hasta el momento. Esta propiedad puede ser inicializada mediante un algoritmo voraz previo.
- *Set<S> solutions*: Conjunto de soluciones encontradas. Esta propiedad es relevante en los casos de querer buscar varias soluciones.

El *algoritmo de backtracking* usa un objeto de tipo *State<V,E>.*

```
public void search() {
	initialGraph();
	State<V,E> initialState =
		StatePath.of(graph,this.goal,this.end);
	search(initialState);
}
```

```
public void search(State<V, E> state) {
	V actual = state.getActualVertex();
	if (goal.test(actual)) this.update(state);
	else {
		for (E edge : graph.edgesListOf(actual)) {
			if (this.forget(state,edge)) continue;
			state.forward(edge);
			search(state);
			addGraph(actual,edge);
			state.back(edge);
		}
	}
}
```

La implementación del tipo *State<V,E>* y de los métodos *initialGraph* y *addGraph*, la veremos más adelante. Ahora nos detendremos en los

métodos *update(state)* y *forget(state,edge).* El primero actualiza las soluciones encontradas y el mejor valor, el segundo olvida la parte del grafo que se alcanza vía la arista *edge*.

El método *forget(state, edge)* filtra, olvida, las soluciones que podemos garantizar que no son mejores que las que ya tenemos. Para ello usa la función de cota calculada a partir de la heurística. Los casos *Min, Max* quedan fusionados usando el comparator.

```
Boolean forget(State<V,E> state, E edge) {
       Boolean r = false;
       Double w = state.geGraph().boundedValue(
          state.getActualVertex(),state.getAccumulateValue(),
          edge,graph.goal(),graph.endVertex(), heuristic);
       if(this.bestValue != null) r =
             comparator.compare(w,this.bestValue) >= 0;
       return r;
}
```

Si se alcanza el vértice final actualiza el conjunto de soluciones y la mejor solución con *update(state).* Debemos recordar que las soluciones son caminos desde el vértice inicial al vértice final. Cada vez que encontramos el vértice final encontramos una solución que podemos acumular en un conjunto de soluciones y actualizar la mejor de ellas.

```
void update(State<V, E> state) {
       if (graph.goalHasSolution()
             .test(state.getActualVertex())) {
       if (this.type == Type.All || this.type == Type.One){
             S s = solution.apply(state.getPath());
             this.solutions.add(s);
       } else if (this.type == Type.Min ||
             this.type == Type.Max) {
         if (this.bestValue == null || this.comparator.compare(
             state.getAccumulateValue(),this.bestValue) < 0) {
             this.bestValue = state.getAccumulateValue();
             this.optimalPath = state.getPath();
         }
       }
       }
}
```

La implementación de *State<V,E>* tiene una *List<E> edges* mantener el camino hacia el origen de cada vértice, un valor acumulado de la función

objetivo hasta el vértice actual y el vértice actual. El estado va actualizando estas propiedades con los métodos *back* y *forward*.

```
class StatePath<V,E> implements State<V, E> {
	private V actualVertex;
	private EGraphPath<V, E> path;
	private EGraph<V,E> graph;
	private List<E> edges;
	private List<Double> weights;
	private Double accumulateValue;
```

El método *forward* da un paso adelante actualizando el camino y su peso.

```
@Override
public void forward(E edge) {
	E lastEdge = edges.isEmpty()?null:List2.last(edges);
	this.accumulateValue = this.getGraph().add(
	  this.actualVertex,this.accumulateValue,edge,lastEdge);
	this.actualVertex = Graphs.getOppositeVertex(graph,edge,
		this.actualVertex);
	this.edges.add(edge);
	this.weights.add(this.accumulateValue);
}
```

El método *back* da un paso atrás actualizando el camino y su peso.

```
@Override
public void back(E edge) {
	this.actualVertex = Graphs.getOppositeVertex(
		graph,edge,this.actualVertex);
	this.edges.remove(this.edges.size()-1);
	this.weights.remove(this.weights.size()-1);
	this.accumulateValue = !this.weights.isEmpty()?
		List2.last(this.weights):
		graph.initialPath().getWeight();
}
```

El método getPath obtiene el camino a partir de la lista de aristas.

```
@Override
public EGraphPath<V, E> getPath() {
	EGraphPath<V,E> ePath = graph.initialPath();
	for(E e:this.edges) {
		ePath.add(e);
	}
	return ePath;
}
```

Una implementación alternativa del estado sería incluir la lista de vértices.

En muchos casos podemos combinar el algoritmo BT con un algoritmo que nos proporcione una solución voraz. Para ello podemos usar el siguiente método de factoría:

```
BT<V, E, S> of(EGraph<V, E> graph,
              Function<GraphPath<V, E>, S> solution) {
       GreedyOnGraph<V, E> ga = GreedyOnGraph.of(graph);
       Optional<GraphPath<V, E>> gp = ga.solutionPath();
       if(gp.isPresent()) return BT.of(graph,
              solution,gp.get().getWeight(),gp.get(),false);
       else return BT.of(graph,
              solution, null, null, false);
}
```

Uso del algoritmo de backtracking

Un uso frecuente de este tipo de algoritmos es para encontrar la mejor solución, alguna solución o varias soluciones de un problema.

Resolvemos de nuevo el problema de la mochila, pero ahora mediante Bactracking. El modelado del grafo ya lo hemos hecho anteriormente. También la implementación de los tipos *MochilaVertex* y *MochilaEdge* e igualmente una heurística adecuada para el problema. Con estos elementos comunes con loa algoritmos Voraces y A* el planteamiento mediante backtracking es:

```
DatosMochila.iniDatos("ficheros/objetosMochila.txt");
MochilaVertex.capacidadInicial = 78;
MochilaVertex e1 = MochilaVertex.initialVertex();
MochilaVertex e2 = MochilaVertex.lastVertex();

SimpleVirtualGraph.endVertexG = e2;
EGraph<MochilaVertex, MochilaEdge> graph =
       SimpleVirtualGraph.sum(
              e1,MochilaVertex.goal(),x->x.weight());

GreedyOnGraph<MochilaVertex, MochilaEdge> rr =
       GreedyOnGraph.of(graph,MochilaVertex::greedyEdge);

GraphPath<MochilaVertex, MochilaEdge> path = rr.search();
```

```
BT<MochilaVertex, MochilaEdge,SolucionMochila> ms =
       BT.of(graph,
       MochilaHeuristic::heuristic,
       MochilaVertex::getSolucion,
       BTType.Max);

ms.bestValue = path.getWeight();
ms.optimalPath = path;

ms.search();
SolucionMochila s = ms.getSolution().get();
System.out.println(s);
```

Algunas cuestiones deben ser destacadas: el grafo asociado al problema tiene caminos de tipo *Sum*, todos los problemas finales tienen solución y el peso de las aristas es el peso ya establecido en el tipo *MochilaEdge*.

Se ha buscado una solución voraz con el algoritmo *GreedyOnGraph* y la política *MochilaVertex::greedyEdge* que ya hemos comentado. Obtenemos así un valor voraz *bv* y un camino voraz *path*. Estos valores podemos utilizarlos para filtrar soluciones peores. Esta solución voraz no tiene por qué existir pero en el caso de la mochila si existe.

Una instancia del algoritmo de backtracking se consigue con el método *BT.of* cuyos parámetros son: el grafo *graph*, la *heurística MochilaHeuristic::heuristic*, una *función para calcular la solución* a partir de un GraphPath *MochilaVertex::getSolucion* y la indicación que es un *problema de maximización* BTType.Max.

Para conseguir la máxima eficiencia inicializamos el mejor valor, variable *ms.bestValue*, con el valor voraz conseguido previamente. Igualmene inicializamos el mejor camino *ms.optimalPath.* El algoritmo descartará cualquier solución cuyo valor sea menor que *bestValue* y si este es el óptimo la solución del problema será la solución voraz .

Tras la ejecución del algoritmo con *ms.search()* podemos obtener la solución del problema con el método *ms.getSolution().* Este método nos da la mejor solución si el tipo del problema es *BTType.Max* o *BTType.Min* o una solución cualquiera si es *BTType.One*. Hay problemas donde nos pueden interesar todas las soluciones o algunas de ellas (*BTType.All*). En ese caso el método *ms.getSolutions()* nos da el conjunto de soluciones.

En muchos casos nos interesa representar el grafo de los vértices y aristas alcanzados por el algoritmo. Esto lo conseguimos con el método *ms.graph().* Este grafo se va construyendo con el método privado

```
void addGraph(V v, E edge) {
	if(withGraph) {
		V v2 = Graphs.getOppositeVertex(graph,edge,v);
		if(!this.outGraph.containsVertex(v))
			this.outGraph.addVertex(v);
		if(!this.outGraph.containsVertex(v2))
			this.outGraph.addVertex(v2);
		if(!this.outGraph.containsEdge(edge))
			this.outGraph.addEdge(v, v2, edge);
	}
}
```

El grafo obtenido puede ser visualizado con las técnicas vistas para todos los grafos.

Implementación directa del algoritmo de backtracking

En algunos casos puede ser interesante implementar directamente el algoritmo de backtracking para un problema particular sin recurrir a la implementación genérica. En ese caso, aunque usemos el diseño del grafo no será necesario obtener una instancia del mismo.

Para hacer esa implementación necesitamos una implementación de los problemas como hemos hecho en la implementación directa en el caso de los algoritmos A*. Sea MochilaProblem este tipo para el problema de la mochila.

Necesitamos un estado que lleve el control del algoritmo. El estado guarda el vértice actual y los detalles del camino desde el vértice inicial. Esos detalles son el valor acumulado, la lista de acciones y la lista de vértices, aunque puede escogerse otras alternativas simpre que se se guarde el camino desde del vértice inicial hasta el actual.

El estado es un tipo mutable cuyo diseño es similar en casi todos los problemas. La actualización del valor acumulado depende de cada problema y está asociado con el peso de las aristas del grafo previamente diseñado si los caminos son de tipo *Sum* o en la forma explicada en su momento para los caminos tipo *Last*.

```
class StateMochila {
	private MochilaProblem vertice;
	private Integer valorAcumulado;
	private List<Integer> acciones;
	private List<MochilaProblem> vertices;
```

```
void forward(Integer a) {
	this.acciones.add(a);
	MochilaProblem vcn = this.vertice().vecino(a);
	this.vertices.add(vcn);
	this.valorAcumulado = this.valorAcumulado() +
		a * DatosMochila.valor(this.vertice().index());
	this.vertice = vcn;
}
```

El estado tiene una propiedad para calcular la solución a partir de los detalles del camino desde el origen y dos operaciones para actualizar sus propiedades: *forward(a) y back(a)* que, respectivamente, avanzan el estado según la arista asociada a la acción *a* y retroceden según *a*.

```
void back(Integer a) {
	this.acciones.remove(this.acciones.size()-1);
	this.vertices.remove(this.vertices.size()-1);
	this.vertice=this.vertices.get(this.vertices.size()-1);
	this.valorAcumulado = this.valorAcumulado() -
		a * DatosMochila.valor(this.vertice.index());
}
```

```
SolucionMochila solucion() {
     return SolucionMochila.of(MochilaBT.start,this.acciones);
}
```

Si no tenemos una aproximación inicial a la solución y con el estado diseñado el algoritmo de bactracking es de la forma.

```
public static MochilaProblem start;
public static StateMochila estado;
public static Integer maxValue;
public static Set<SolucionMochila> soluciones;

public static void btm(Integer capacidadInicial) {
	MochilaBT.start = MochilaProblem.of(0,capacidadInicial);
	MochilaBT.estado = StateMochila.of(start);
	MochilaBT.maxValue = Integer.MIN_VALUE;
	MochilaBT.soluciones = new HashSet<>();
	btm();
}
```

Si ya conocemos una solución voraz:

```
public static void btm(Integer capacidadInicial,
              Integer maxValue,
              SolucionMochila s) {
       MochilaBT.start = MochilaProblem.of(0,capacidadInicial);
       MochilaBT.estado = StateMochila.of(start);
       MochilaBT.maxValue = maxValue;
       MochilaBT.soluciones = new HashSet<>();
       MochilaBT.soluciones.add(s);
       btm();
}
```

El algoritmo debe tener en cuenta los mecanismos de filtro: la variable *cota* es una cota de los valores del camino que pasa por el vértice actual y sigue según la alternativa a. Si esta cota es inferior a *MochilaBT.maxValue* descartamos esa parte del grafo.

Si alcanzamos un vértice final y el valor acumulado es mejor que *MochilaBT.maxValue* actualizamos el mejor valor y añadimos la solución encontrada al conjunto de soluciones.

Para que la implementación sea correcta debemos tener previamente un buen diseño del grafo que nos permita implementar *MochilaProblem* y en particular sus propiedades, el cálculo de las acciones disponibles y el vecino según una acción.

El algoritmo recursivo para este problema de la mochila es:

```
public static void btm() {
       if(MochilaBT.estado.vertice().index()==DatosMochila.n) {
              Integer value = estado.valorAcumulado();
              if(value > MochilaBT.maxValue) {
                     MochilaBT.maxValue = value;
                     MochilaBT.soluciones.add(
                            MochilaBT.estado.solucion());
              }
       } else {
              List<Integer> alternativas =
                     MochilaBT.estado.vertice().acciones();
              for(Integer a:alternativas) {
                     Double cota =
                            MochilaBT.estado.valorAcumulado()+
                                 Heuristica.cota(
                                   MochilaBT.estado.vertice(),a);
                     if(cota <= MochilaBT.maxValue) continue;
                     MochilaBT.estado.forward(a);
                     btm();
                     MochilaBT.estado.back(a);
              }
       }
}
```

Backtracking aleatorio

Para buscar algunas soluciones en problemas complejos hay una variante muy eficaz de los algoritmos de *backtracking* que denominaremos *Backtracking Aleatorio*. Muy someramente se trata de elegir aleatoriamente una de las alternativas para los problemas de tamaño mayor que un umbral y seguir todas las alternativas para los problemas de tamaño menor y repetir esta estrategia hasta encontrar el número de soluciones deseadas.

Este algoritmo es especialmente eficaz para encontrar una solución al problema de las reinas.

El algoritmo puede ser instanciado con el método *BTR.of:*

```
BTR<V, E, S> BTR.of(
       EGraph<V, E> graph,
       Function<GraphPath<V, E>, S> solution,
       BTType type,
       Function<V, Integer> size);
}
```

Problema de las reinas

El problema consiste en colocar n reinas en un tablero de ajedrez $n \times n$ de tal manera que ninguna de ellas amenace a ninguna de las demás. Una reina amenaza a las casillas de la misma fila, de la misma columna y de las mismas diagonales. Las filas y columnas toman valores en $0..n-1$.

Ya hemos abordado este problema para ser resuelto mediante programación lineal entera y algoritmos genéticos. Veamos cómo resolverlo mediante Bactracking.

Un modelo del problema usando las variables enteras $x_i, \quad i: [0, n-1]$, que indica que una reina se colocará en la casilla (i, x_i), es el siguiente:

$$P_{i=0}^{n-1}(x_i, i)$$
$$AD_{i=0}^{n-1}(x_i - i)$$
$$AD_{i=0}^{n-1}(x_i + i)$$
$$int \;\; x_i, \quad i \in [0, n-1]$$

Recordamos que $AD_{i=0}^{n-1} f(i)$ es una restricción que indica que los valores de $f(i)$ son distintos y $P_{i=0}^{n-1}(x_i, i)$ que los valores de x_i son una permutación de los valores $i \in [0, n-1]$. La razón para este modelo es que las filas ocupadas serán los valores de las x_i, las diagonales principales ocupadas serán los valores $x_i - i$ y las diagonales secundarias ocupadas serán los valores $x_i + i$ y en los tres casos deben ser diferentes.

Para abordar el problema por Bactracking buscamos un grafo cuyos vértices sean los problemas obtenidos al generalizar el problema de partida. Llamemos *ReinasVertex* al tipo de los vértices de este grafo:

ReinasVertex:

Propiedades:

- i: Integer
- fo: List<Integer>, filas ocupadas
- *dp: Set<Integer>,* diagonales principales ocupadas, derivada
- *ds: Set<Integer>,* diagonales secundarias ocupadas, derivada
- *vl: Set<Integer>,* valores libres de las filas para colocar en columna i, derivada
- *errores():* Número de errores en los problemas finales, derivada.
- *n: Integer,* compartida, número columnas igual al de filas
- *t: Integer,* derivada n-i, tamaño

Invariante:

- $dp = S_{k:0..i-1}(fo[i] - i)$
- $ds = S_{k:0..i-1}(fo[i] + i)$
- $vl = S_{k:0..i-1|p(k)}k, \quad p(k) \equiv k \notin fo \wedge (k - i) \notin dp \wedge (k + i) \notin ds$
- errores = *n-|dp|+n-|fo|+n-|ds|*

La propiedad *errores* no es necesaria pero la usamos para mostrar que la solución encontrada es adecuada y también para caracterizar los problemas finales que tienen solución.

Interpretación:

Encontrar la asignación de las reinas desde *i* hasta el final asumiendo que ya están bien colocadas, sin amenazarse entre ellas, las reinas de *0..i-1*

Igualdad

- Dos problemas son iguales si lo son i, fo

Es válido

- *i>=0, i<=n, |fo|=|dp|=|ds|*

Factoría:

- inicial(): Crea el problema (0,{}]
- goal(v) = p.i ==n

Acciones:

$A_i = a: \{0..n-1\} | a \notin fo, a - i \notin dp, a + i \notin ds\}$

Vecino

Caso general: $neighbord(a) = (i + 1, fo + a)$

Peso de la arista: Peso de la arista 1

Peso del camino: Suma de los pesos de las aristas

Solución Voraz: Escoger aleatoriamente entre las alternativas disponibles.

Heurística: 0.

Solución: Map<Integer,Integer> que recoja la fila donde se colocará la reina en cada columna.

Con estas ideas implementamos el tipo de los vértices cuyos detalles pueden verse en le repositorio:

```
public record ReinasVertex(Integer index, List<Integer> fo,
       IntegerSet dpo, IntegerSet dso) implements
       VirtualVertex<ReinasVertex,
              SimpleEdgeAction<ReinasVertex,Integer>,Integer> {

       public static Predicate<ReinasVertex> goalHasSolution(){
              return v->v.errores() == 0;
       }

      @Override
       public List<Integer> actions() {
              List<Integer> r =
                     IntStream.range(0,ReinasVertex.n).boxed()
                     .filter(e->!this.fo.contains(e) &&
                     !this.dpo.contains(e+this.index) &&
                     !this.dso.contains(e-this.index))
                     .collect(Collectors.toList());
              return r;
       }
```

```
        @Override
        public ReinasVertex neighbor(Integer a) {
                Integer index = this.index+1;
                List<Integer> fo = new ArrayList<>(this.fo);
                fo.add(a);
                IntegerSet dpo = this.dpo.addNew(a+this.index);
                IntegerSet dso = this.dso.addNew(a-this.index);
                return ReinasVertex.of(index, fo, dpo, dso);
        }
        …
}
```

Hacemos notar que en este problema no todos los problemas finales tienen solución. La tienen solo aquellos en los cuales la propiedad *errores()* sea cero. El tipo de las aristas es simplemente *SimpleEdgeAction.*

Resolver el problema mediante backtracking para obtener todas las soluciones es de la forma:

```
ReinasVertex.n = 8;
ReinasVertex e1 = ReinasVertex.first();
Predicate<ReinasVertex> goal=v -> v.index() == ReinasVertex.n;

EGraph<ReinasVertex,SimpleEdgeAction<ReinasVertex,Integer>>
graph =
EGraph.virtual(e1,ReinasVertex.goal(),PathType.Last,Type.All)
        .goalHasSolution(ReinasVertex.goalHasSolution())
        .solutionNumber(1000)
        .vertexWeight(v->v.errores().doubleValue())
        .build();

BTR<ReinasVertex,ActionSimpleEdge<ReinasVertex,
                Integer>, SolucionReinas> ms =
        BTR.of(graph,
        SolucionReinas::of,v->ReinasVertex.n-v.index(),15);

ms.search();
System.out.println(ms.getSolutions());
```

Los parámetros del método *BT.of* son:

- *graph*: Un grafo extendido
- *solution:* Una función que calcula la solución a partir del camino solución
- *type:* El tipo del algoritmo que puede ser *BTType {Min,Max,One,All},* pero para el problema de las reinas solo tiene

sentido *One, All* si no especificamos una función objetivo. Al escoger All y el número de soluciones alto (1000) obtendremos todas las soluciones.

Pero para encontrar solo una solución a este tipo de que cumpla las restricciones, aunque no sea la mejor, es más eficiente usar *Bracktracking aleatorio* de la forma:

```
ReinasVertex.n = 110;
ReinasVertex e1 = ReinasVertex.first();

EGraph<ReinasVertex,SimpleEdgeAction<ReinasVertex,
       Integer>> graph =
       EGraph.virtual(e1,ReinasVertex.goal(),PathType.Last,
              Type.All)
       .goalHasSolution(ReinasVertex.goalHasSolution())
       .solutionNumber(1)
       .vertexWeight(v->v.errores().doubleValue())
       .build();

BTR<ReinasVertex,SimpleEdgeAction<ReinasVertex,Integer>,
       SolucionReinas> ms =

       BTR.of(graph,SolucionReinas::of,
              v->ReinasVertex.n-v.index(),15);

ms.search();
System.out.println(ms.getSolution());
```

Ahora la función *v->ReinasVertex.n-v.index()* calcula el tamaño del problema y si el mayor que *threshold,* en este caso 15*,* escoge aleatoriamente una de las alternativas.

Programación dinámica

La programación dinámica es un algoritmo muy general para resolver los problemas anteriores. Veremos primero un tipo particular llamado *Programación Dinámica de Reducción*. Es el algoritmo adecuado cuando hemos modelado nuestro problema con grafos. Es decir, tras tomar una acción pasamos de un vértice a otro. La *Programación Dinámica* en general sirve para resolver problemas a los que hemos asociado un *hypergrafo*. Es decir, tras tomar una acción pasamos de un vértice a una lista de otros vértices.

Programación dinámica de reducción

Los algoritmos de *Programación Dinámica* son algoritmos recursivos con memoria. Por tanto, son convenientes sólo cuando hay solapamiento, es decir, se puede llegar a un problema/vértice desde distintos caminos. Backtracking no usaba memoria.

La Programación Dinámica general busca el árbol mínimo en un hipergrafo: cada problema tiene varios subproblemas tras tomar una acción. Un caso particular de la misma que veremos primero es *Programación Dinámica de Reducción* que busca caminos óptimos en un grafo: cada problema tiene un solo subproblema tras tomar una acción.

Implementación

Veamos los detalles del algoritmo de *Programación Dinámica de Reducción*. Por cuestiones de eficiencia buscaremos, en primer lugar, soluciones parciales modeladas por el tipo *Sp<A>* que representaremos por *sp* y son tuplas (a, w). La primera componente a indica la acción que constituye la solución y w el valor de la función objetivo en la solución. La memoria del algoritmo es de la forma *Map<P,Sp<A>>.* Donde P es el tipo que implementa el problema y p, p' problemas concretos. El tipo P tiene las propiedades de un vértice del grafo que habíamos diseñado y otros métodos que veremos abajo. El esquema recursivo del algoritmo para el caso de *Programación Dinámica de Reducción* es:

$$pdr(p) = \begin{cases} m[p], & p \in m \\ sb(p), & b(p) \\ \underset{a \in A_p}{sP}\,(p, sA(p, a, pdr(p'))), & !\,b(p), p' = v(p, a) \end{cases}$$

El algoritmo de *Programación Dinámica de Reducción* tiene un conjunto de atributos privados

```
public class DPR<V, E> {
	public EGraph<V, E> graph;
	public Double bestValue = null;
	private TriFunction<V, Predicate<V>, V, Double>
		heuristic;
	private Comparator<Sp<E>> comparatorEdges;
	private Comparator<Double> comparator;
	public Map<V, Sp<E>> solutionsTree;
	private PDType type;
	private EGraphPath<V, E> path;
	public GraphPath<V, E> optimalPath;
```

- *EGraph<V, E> graph:* El grafo extendido asociado al problema
- *Double bestValue:* El mejor valor de la función objetivo encontrado
- *GraphPath<V, E> optimalPath:* El mejor camino encontrado
- *TriFunction<V, Predicate<V>, V, Double> heuristic:* La heuristica
- *Comparator<Sp<E>> comparatorEdges:* El orden de las soluciones parciales
- *Comparator<Double> comparator:* El orden de los valores de la función objetivo

- *Map<V, Sp<E>> solutionsTree:* La memoria de soluciones parciales
- *PDType type:* El tipo que puede ser *Min, Max*.
- *EGraphPath<V, E> path:* El camino inicial

Con esos atributos el constructor de la clase inicializa los diversos órdenes, la memoria, etc. El mejor valor y el mejor camino pueden ser inicicializadas posteriormente si es necesario.

```
DPR(EGraph<V, E> g, Double bestValue,
              GraphPath<V, E> optimalPath, Boolean withGraph) {
       this.graph = g;
       this.comparatorEdges = this.type == PDType.Min?
          Comparator.naturalOrder():Comparator.reverseOrder();
       this.comparator = this.type == PDType.Min?
          Comparator.naturalOrder():Comparator.reverseOrder();
       this.solutionsTree = new HashMap<>();
       this.bestValue = bestValue;
       this.optimalPath = optimalPath;
       this.withGraph = withGraph;
}
```

El algoritmo devuelve un camino opcional de tipo *Optional<GraphPath<V, E>>.* Opcional porque puede que el problema no tenga solución. En Java el tipo *Optional<E>* es un contenedor que puede contener o no un valor distinto de *null*.

El algoritmo llama al método recursivo *search* que construye la memoria de soluciones parciales *solutionsTree* y a partir de aquí obtiene el camino solución con el método *pathFrom*.

```
Optional<GraphPath<V, E>> search() {
       iniciaGraph();
       this.solutionsTree = new HashMap<>();
       search(graph.startVertex(),0., null);
       return pathFrom(graph.startVertex());
}
```

El método recursivo separando el tratatamiento de los casos finales de los recursivos es:

```
private Sp<E> search(V actual, Double accumulateValue,
                     E edgeToOrigin) {
       Sp<E> r = null;
       if(this.solutionsTree.containsKey(actual)) {
              r = this.solutionsTree.get(actual);
       } else if (graph.goal().test(actual)) {
              if (graph.goalHasSolution().test(actual)) {
                     update(accumulateValue);
                     r = Sp.of(
                     this.graph.goalSolution(actual), null);
              } else r = null;
                     this.solutionsTree.put(actual, r);
```

Si encuentra un vértice final comprueba si tiene solución (*goalHasSolutionn.test(actual))* y si la tiene actualiza el mejor valor. Si no la tiene devuelve null. El peso de la solución de los problemas finales viene dado por la propiedad *goalSolution(actual)* del camino y depende del tipo de camino. En los caminos Sum es 0. Y en los Last el peso del vértice.

Al encontrar un vértice final que cumple la restriccción se actualiza el mejor valor con el método update

```
protected void update(Double accumulateValue) {
       if(this.bestValue == null || comparator.compare(
                     accumulateValue,this.bestValue) < 0) {
              this.bestValue = accumulateValue;
       }
}
```


En los casos recursivos se calcula la solución parcial de un vértice a partir de las soluciones parciales de los vecinos.

```
} else {
	List<Sp<E>> rs = new ArrayList<>();
	for (E edge : graph.edgesListOf(actual)) {
		if (this.forget(edge,actual,
			accumulateValue,graph.goal(),
			graph.endVertex())) continue;
		V v=Graphs.getOppositeVertex(graph,edge,actual);
		Double ac = this.graph.add(actual,
			accumulateValue,edge,edgeToOrigin);
		Sp<E> s = search(v,ac,edge);
		if (s!=null) {
			E lastEdge=this.solutionsTree.get(v).edge;
			Double spv =
				this.graph.fromNeighborSolution(v,
				s.weight,edge,lastEdge);
			Sp<E> sp = Sp.of(spv,edge);
			rs.add(sp);
		}
		addGraph(actual, edge);
	}
	if (!rs.isEmpty()) {
		r = rs.stream().filter(s->s!=null)
			.min(this.comparatorEdges).orElse(null);
		this.solutionsTree.put(actual, r);
	}
	}
	return r;
}
```

El algoritmo tiene una memoria en una variable que para cada vértice encontrado asocia un par (e,w): la arista que define el camino solución desde ese vértice y el valor de la función objetivo de la solución. Este par lo llamaremos *solución parcial*. La variable de la memoria es:

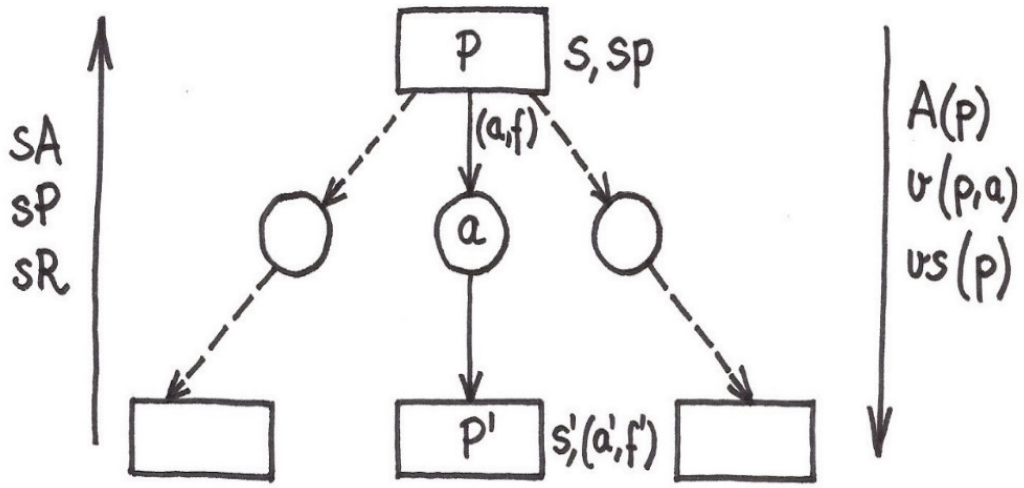
```
Map<V,Sp<E>> solutionsTree;
```

No olvidemos que en todos los problemas que modelamos mediante grafos y resolvemos por *algoritmos voraces, A*, Backtracking o Programación Dinámica de Reducción* la solución es un camino en el grafo desde el vértice inicial al final.

Recursivamente para cada vértice encuentra la solución parcial, el par (e,w). Esta solución puede ser *null* que indica que el problema

representado por ese vértice no tiene solución. Si no tiene solución entonces la respuesta es *null*.

El esquema gráfico de cálculo es:



Para cada vértice calcula la solución parcial de los vecinos y a partir de ellas acumula una lista de soluciones parciales siguiendo cada una de las alternativas. La solución parcial asociada a cada alternativa depende del tipo de camino, del peso de la arista y el vértice. El método *fromNeighborSolution* hace este cálculo. En el caso de un camino tipo Sum se cumple *s.weight = s'.weight+e.weight* (siendo e la arista que conecta con el vecino) pero esta expresión puede incluir otros pesos. En el caso de un camino tipo *Last* se cumple *s.weight = s'.weight.*

A partir de las soluciones obtenidas en los caminos hacia sus vecinos siguiendo las diferentes alternativas se obtiene la mejor que no sea *null* o *null* si todas lo son:

```
r = rs.stream().filter(s->s!=null)
       .min(this.comparatorEdges).orElse(null);
```

Pero hemos de tener en cuenta que todas las soluciones pueden haber sido filtradas. En ese caso no se guarda ninguna solución en la memoria porque

no la hemos obtenido al haber sido filtradas todas. La solución asociada a ese vértice se encontrará si se llega a él siguiendo otro camino desde la raíz.

Terminado el algoritmo disponemos de la memoria *solutionsTree* que acumula las soluciones parciales de todos los problemas encontrados. Buscando en esta memoria un vértice encontramos su solución parcial, un par (e, w), y siguiendo la arista encontramos un vecino y luego otro formado un camino hasta el vértice final. Este camino óptimo desde el vértice inicial lo calcula el método *pathFrom* usando la memoria *solutionsTree.*

El algoritmo devuelve un camino opcional porque puede que no haya solución. Si se ha proporcionado un camino voraz que es óptimo, el algoritmo filtrará todas las demás posibilidades y devolverá ese camino. A partir de ese camino óptimo se encuentra la solución del problema.

El algoritmo filtra los caminos que conducen a soluciones peores de una forma similar al backtracking. Usa el método *forget* para descartar caminos y si no los descarta actualiza el valor acumulado y sigue buscando a partir del vértice vecino:

```
private Boolean forget(E edge, V actual,
              Double accumulateValue,Predicate<V> goal,V end) {
       Boolean r = false;
       Double w = this.path.boundedValue(accumulateValue,
              actual, edge, goal, end,
                     (v1,p,v2)->this.newHeuristic(v1,p,v2));
       if(this.bestValue != null)
              r = comparator.compare(w, this.bestValue) >= 0;
       return r;
}
```

Este método usa la heurística para estimar el peso de un camino hasta el vértice final. Pero como el algoritmo usa memoria puede que ya haya calculado previamente una solución para un vértice. En ese caso se usará como heurística el valor de la función objetivo de la solución encontrada

ya que esa solución es óptima para ese vértice. Para ello se ha diseñado el método *newHeuristic* que es usado por forget.

```
private Double newHeuristic(V v1, Predicate<V> p, V v2) {
	Double r;
	if (this.solutionsTree.containsKey(v1))
		r = this.solutionsTree.get(v1).weight();
	else
		r = graph.heuristic().apply(v1, p, v2);
	return r;
}
```

Tras completado el algoritmo que calcula a solución parcial óptima para cada vértice obtenemos un *Map<V,Sep<E>>* que asocia a cada vértice su solución parcial óptima. A partir de ese map debemos reconstuir el camino óptimo para un vértice dado. La forma de hacerlo es partir de un camino inicial que sólo contiene ese vértice e ir añadiendo las aristas óptimas guardadas en el *Map* para el vértice actual y cada uno de los vértices vecinos alcanzados siguiendo la arista óptima.

```
Optional<GraphPath<V, E>> pathFrom(V vertex) {
	if(this.solutionsTree.get(vertex) == null)
		return Optional.empty();
	E edge = this.solutionsTree.get(vertex).edge;
	EGraphPath<V,E> ePath = EGraphPath.ofVertex(this.graph,
		vertex,this.graph.pathType());
	while(edge != null) {
		ePath.add(edge);
		vertex =
		    Graphs.getOppositeVertex(graph,edge,vertex);
		edge = this.solutionsTree.get(vertex).edge;
	}
	this.optimalPath = ePath;
	return Optional.of(ePath);
}
```

Como vemos siempre se devuelve un *Optional<GraphPath<V, E>>* puesto que puede que no haya solució. Esto ocurre si en el Map la solución parcial asociada al vértice es *null*.

Si hemos proporcionado al algoritmo una solución voraz, esta información quedará guardada en el atributo *optimalPath* que será *null* en otro caso. Si la solución voraz es óptima entonces todos los vértices serán filtrados y por lo tanto se devolverá el camino voraz ya guardado en *optimalPath*.

Teniendo en cuenta lo anterior el método público *Optional<GraphPath<V, E>> search()* es:

```
Public Optional<GraphPath<V, E>> search() {
       iniciaGraph();
       this.solutionsTree = new HashMap<>();
       Sp<E> r = search(graph.startVertex(),0., null);
       if(r == null && this.optimalPath !=null)
              return Optional.of(this.optimalPath);
       return pathFrom(graph.startVertex());
}
```

En muchos casos podemos combinar el algoritmo DPR con un algoritmo que nos proporcione una solución voraz. Para ello podemos usar el siguiente método de factoría:

```
DPR<V, E> ofGreedy(EGraph<V, E> graph) {
       GreedyOnGraph<V, E> ga = GreedyOnGraph.of(graph);
       Optional<GraphPath<V, E>> gp = ga.search();
       if(gp.isPresent()) return DPR.of(graph,null,
              gp.get().getWeight(),gp.get(),false);
       else return DPR.of(graph,null,null,null,false);
}
```

Uso de la Programación Dinámica de Reducción

Los algoritmos de *Programación Dinámica de Reducción* necesitan un grafo y una definición del peso asociado a un camino. Como en *Backtracking* los vértices deben tener asociado un tamaño y es necesario que los vecinos de un vértice tengan un tamaño menor. Se necesitan un vértice inicial, un vértice final o la definición de casos base, y es importante disponer de una heurística, aunque no es necesario. Se necesita, además, poder calcular la solución de un problema asumido que se conocen las soluciones de los problemas vecinos.

Los algoritmos de *Programación Dinámica de Reducción* son adecuados para buscar un camino mínimo o máximo.

Frente a los algoritmos de *Backtracking* la *Programación Dinámica de Reducción* es útil cuando hay problemas compartidos. Es decir, hay caminos que convergen en un mismo vértice intermedio. La *Programación Dinámica de Reducción* al usar memoria recuerda los cálculos previos y no

vuelve a repetirlos lo que supone un aumento de eficiencia. Si hay pocos vértices compartidos puede ser más adecuado usar *Backtracking.*

Cuando usamos *Programación Dinámica de Reducción* es muy importante definir exactamente la igualdad entre dos problemas. Esta definición indicará si los problemas están repetidos o no.

Como en el caso de los algoritmos de *Backtracking* es muy importante definir los casos base y si tienen solución o no. Como técnica general definimos opcionalmente un problema final posiblemente ficticio que tenga una solución conocida como ya vimos al diseñar los grafos asociados a los problemas.

Los casos base los tenemos en cuenta añadiendo una sola arista con el peso correspondiente desde ellos al problema final si tienen solución y declarando que no tienen vecinos si no tuvieran solución. Estas consideraciones las hacemos al diseñar el grafo y en particular al implementar el método *actions()* del vértice.

Aunque ya lo hemos visto en otros ejemplos es conveniente también en *Programación Dinámica de Reducción* partir de un modelo del problema. Las variables nos van a orientar sobre las acciones y su significado, la función objetivo sobre los pesos de las aristas y las restricciones sobre el conjunto de acciones posibles. Para las restricciones de mayor o igual ya hemos visto esto para el caso de la Mochila. En el caso de restricciones de igualdad añadimos al modelo una restricción de desigualdad, mayor o menor según los casos, para poder deducir las acciones posibles de esa restricción añadida. Esto es válido para todas las técnicas: *A**, *Backtracking y Programación Dinámica de Reducción*

Problema de las monedas

Tenemos un sistema monetario donde las conchas marinas hacen de monedas. Existen n tipos de monedas. Cada tipo i tiene un valor, v_i, y un peso, w_i. El Banco Central Etrusco *(BCE*) sólo admite cambios por valor V. Es decir, los clientes pueden dar dinero por valor total V, y a cambio se les devuelve la misma cantidad en las monedas que el cliente desee. Nuestro propósito es dar las monedas con mayor peso total, y a cambio recibir las monedas con menor peso posible.

En definitiva, debemos encontrar una combinación de monedas con valor total V, y donde el peso total sea el máximo. Y, por otro lado, también debes encontrar una combinación por valor V y donde el peso total sea el mínimo. Se supone que existe un número ilimitado de monedas de todos los tipos. Sólo se admitirá como válida la solución óptima.

Un modelo de problema es escoger variables x_i, y_i que representen el número de unidades de cada tipo elegidas para entregar y para recibir. Podemos deducir que el número máximo de unidades de la moneda i será $m_i = V/v_i$ donde suponemos valores y división enteros. Con ese dato el modelo del problema sería:

$$\max \sum_{i=0}^{n-1} x_i w_i - y_i w_i$$

$$\sum_{i=0}^{n-1} x_i v_i = V$$

$$\sum_{i=0}^{n-1} y_i v_i = V$$

$$x_i \leq m_i, \quad i \in [0, n-1]$$

$$y_i \leq m_i, \quad i \in [0, n-1]$$

$$int \quad x_i, y_i, \quad i \in [0, n-1]$$

El modelo se puede dividir en dos que resolveremos secuencialmente: uno de maximizar para calcular las variables x_i y otro exactamente igual de minimizar para calcular y_i. Una forma alternativa sería resolverlo como un único problema siendo las alternativas pares de enteros (x_i, y_j) pero esta solución sería más compleja y menos eficiente al dispararse el número de acciones posibles y complicar la elección de heurísticas y soluciones voraces.

Además, al resolver el mismo problema, primero maximizando la función objetivo y luego minimizado la misma función objetivo con las mismas restricciones, nos va a permitir ver detalles sobre las heurísticas y las soluciones voraces en ambos casos. Como dijimos más arriba las soluciones voraces cumplen las restricciones del problema y tienen como objetivo encontrar una solución que aproxime el valor óptimo y nos pueda

servir como primer valor estimado de la función objetivo tanto en *Backtracking* como en *Programación Dinámica de Reducción.* La heurística sirve para orientar a los algoritmos de A* hacia el objetivo y filtrar vecinos en *Backtracking* y *Programación Dinámica de Reducción.* La heurística tiene que ser admisible, pero, al calcularla mediante un algoritmo voraz, usualmente seguimos un camino que no cumple las restricciones.

Detallaremos los tipos y la generalización necesarios para abordar el problema mediante *Programación Dinámica de Reducción* y veremos a la vez el caso de maximización y el de minimización.

Moneda:

- Valor: Integer, v
- Peso: Integer, w
- PesoUnitario: Double, derivada, w/v
- Monedas: List<Moneda>, compartida

MonedaVertex:

Propiedades:

- i: Integer, básica
- vr: Integer, Valor restante, básica
- n: Integer, compartida
- V: Integer, compartida
- t: Integer, derivada n-i

Interpretación:

Encontrar la elección de monedas cuyo valor suma *vr* y su peso es máximo, o mínimo, teniendo en cuenta solo las monedas de *i* hasta el final

Igualdad

- Dos problemas son iguales si lo son *i, vr*

Es válido: i >= 0, i <= n, vr >= 0

Al decidir *vr >= 0* hemos añadido la restricción $vr = V - \sum_{i=0}^{n-1} x_i v_i \geq 0$ que nos va a permitir escoger un conjunto de acciones que hagan más eficiente la solución. Exigir que se cumpla la igualdad $vr = 0$ lo haremos

en el problema final y en los casos bases. Escogemos como problemas finales los que cumplen *i ==n*.

Factoría:

- *inicial():* Crea el problema (0,V)
- *goal():* Define los problemas finales p.i ==n

Problema Final *p.i ==n*

El problema final tiene solución si p.vr == 0. Este predicado define el método *goalHasSolution()*.

Sea $n_i = \frac{vr}{v_i}$

Casos base y alternativas

1. p.vr == 0, tiene solución, $A_i = \{0\}$
2. pi. == n-1, vr es divisible por v_i, tiene solución si, $A_i = \{n_i\}$
3. pi. == n-1, vr no es divisible por v_i, no tiene solución, $A_i = \{\}$

Acciones en el caso general

$A_i = \{n_i \,..\, 0\}$, *caso general de maximización o minimización*. Debido a que el valor de vr para el vecino es $vr - a * v_i \geq 0$, como ya hemos comentado y por lo tanto $a \leq \frac{vr}{v_i} = n_i$. Las acciones están ordenadas para que primero se siga el camino voraz.

Vecino

Caso general: $neighbord(a) = (i + 1, vr - a * v_i)$

Casos Base 1: $neighbord(0) = (n, 0)$

Casos Base 2: $neighbord(n_i) = (n, 0)$

Peso de la arista

Peso: $w = a * w_i$

Peso del camino:

Suma de los pesos de las aristas

Solución Voraz

- Si es de maximizar ordenar las monedas por la razón $^{p_i}/_{v_i}$ de mayor a menor y tomar la acción $n_i = \frac{vr}{v_i}$ con valor inicial *V* para *vr*. Es decir, ordenamos las monedas por su peso unitario de mayor a menor y vamos escogiendo de cada moneda la mayor cantidad de unidades posibles de las monedas más pesadas
- Si es de minimizar ordenar las monedas por la razón $^{p_i}/_{v_i}$ de menor a mayor y tomar la acción $n_i = \frac{vr}{v_i}$ con valor inicial *V* para *vr*. Es decir ordenamos las monedas por su peso unitario de menor a mayor y vamos escogiendo de cada moneda la mayor cantidad de unidades posibles de las monedas menos pesadas.

Heurística

- Si es de maximizar ordenar las monedas por la razón $^{p_i}/_{v_i}$ de mayor a menor y diseñar un algoritmo voraz con acción heurística $n_i = \frac{vr}{v_i}$ pero siendo ahora vr una cantidad real y la división real.
- Si es de minimizar ordenar las monedas por la razón $^{p_i}/_{v_i}$ de menor a mayor y tomar la acción $n_i = \frac{vr}{v_i}$ con valor inicial *V* para *vr* pero siendo ahora vr una cantidad real y la división real.

Con estas ideas implementamos el tipo de los vértices y las aristas

```
record MonedaVertex(Integer index,Integer valorRestante)
       implements   ActionVirtualVertex<MonedaVertex,
              MonedaEdge, Integer> {
       …
}
```

Los detalles del método *actions*

```
@Override
public List<Integer> actions() {
       List<Integer> r;
       if(this.index() == MonedaVertex.n)
              r = new ArrayList<>();
       else if(this.index() == MonedaVertex.n-1 &&
              this.valorRestante%Moneda.valor(this.index)==0) {
                     r = List.of(this.accionVoraz().action());
       } else if(this.index() == MonedaVertex.n-1 &&
              this.valorRestante%Moneda.valor(this.index)!=0) {
              r = new ArrayList<>();
       } else {
              Integer nue = this.valorRestante()/
                            Moneda.valor(this.index);
              r = IntStream.range(0,nue+1).boxed()
                            .collect(Collectors.toList());
              Collections.reverse(r);
       }
       return r;
}
```

El cálculo del vecino.

```
@Override
public MonedaVertex neighbor(Integer a) {
       MonedaVertex r;
       if(this.valorRestante() == 0) r = MonedaVertex.last();
       else r= MonedaVertex.of(this.index()+1,
            this.valorRestante()-a*Moneda.valor(this.index()));
       return r;
}
```

La implementación de la arista

```
record MonedaEdge(MonedaVertex source, MonedaVertex target,
              Integer action, Double weight)
       implements ActionSimpleEdge<MonedaVertex,Integer> {
    public static MonedaEdge of(MonedaVertex c1,
                  MonedaVertex c2, Integer action) {
           Double w = (double)(action*Moneda.peso(c1.index()));
       return new MonedaEdge(c1, c2, action, w);
    }
}
```

Solución:

- La solución más adecuada serían dos *Multiset<Integer>* que indicarían el número de unidades de cada moneda entregadas y recibidas
- Los dos *multiset* se obtendrían resolviendo primero un problema de maximización y luego el mismo pero de minimización

La parte de maximización es de la forma:

```
MonedaVertex.datosIniciales("ficheros/monedas3.txt", 36);
MonedaVertex e1 = MonedaVertex.first();

EGraph<MonedaVertex, MonedaEdge> graph =
 EGraph.virtual(e1,MonedaVertex.goal(),PathType.Sum,Type.Max)
       .goalHasSolution(MonedaVertex.goalHasSolution())
       .greedyEdge(MonedaVertex::aristaVoraz)
       .heuristic(MonedasHeuristica::heuristic)
       .build();

GreedySearchOnGraph<MonedaVertex, MonedaEdge> rr =
       GreedyOnGraph.of(graph);

GraphPath<MonedaVertex, MonedaEdge> path =    rr.search();
```

El método *datosIniciales*, además de leer los datos de un fichero, ordena las monedas con el criterio explicado arriba.

Podemos comprobar que en el caso de minimización la heurística es menor o igual que el valor voraz y, en el caso de maximización, la heurística es mayor o igual que el valor voraz. Si ambas son iguales la solución voraz es la óptima.

El algoritmo de Programación Dinámica de Reducción usa como inicialización los valores proporcionados por los métodos voraces si estos encuentran la solución.

```
GraphPath<MonedaVertex, MonedaEdge> path2 = rr.path();
DPR<MonedaVertex, MonedaEdge, SolucionMonedas> ms2;
if (rr.isSolution(path2)) {
       System.out.println("3 = " + SolucionMonedas.of(path2));
       ms2 = DPR.ofGreedy(graph);
}else {
       ms2 = DPR.of(graph);
}
```

```
Optional<GraphPath<MonedaVertex, MonedaEdge>> s1ms1.search();

if (s1.isPresent()) System.out.println(
      SolucionMonedas.of(s1.get()));
else System.out.println("No hay solucion");
…
```

Como puede verse tras leer los datos del problema creamos el vértice inicial y ordenamos las monedas tal como hemos comentado. Tras ello creamos un grafo virtual derivado de las definiciones de los vértices y aristas. La primera tarea es calcular una solución voraz que nos pueda servir como primera aproximación a la solución. Esto lo hacemos con el método *GreedyOnGraph.of* que tiene como parámetros:

- Un grafo

Posteriormente creamos un algoritmo de Programación Dinámica de Reducción con el método *DPR.of* que tiene como parámetros:

- Un grafo

El valor de la solución voraz se lo proporcionamos al algoritmo para que filtre las aristas del grafo que conducen a peores resultados.

La minimización puede verse en el repositorio. El esquema es similar pero ahora tras leer los datos del fichero debemos ordenar los datos de otra forma.

Implementación directa de la Programación Dinámica de Reducción

Una implementación de la solución de un problema mediante Programación Dinámica de Reducción comienza por diseñar un tipo que represente los problemas. Este tipo sigue las mismas ideas que la implementación de los vértices del grafo anterior, pero sin comprometerse a implementar ninguna interface. Veamos el mismo problema de las monedas nuevamente.

```
public record MonedaProblem(Integer index,
       Integer valorRestante) {
       …
       public List<Integer> acciones() {
              List<Integer> r;
              if(this.index() == MonedaProblem.n) r =
                     new ArrayList<>();
              else if(this.index() == MonedaProblem.n-1 &&
                           this.valorRestante%Moneda.
                                  valor(this.index) == 0) {
                     r = List.of(this.accionVoraz());
              } else if(this.index() == MonedaProblem.n-1 &&
                           this.valorRestante%Moneda.
                                  valor(this.index) != 0) {
                  r = new ArrayList<>();
              } else {
                     Integer nue = this.valorRestante()/
                           Moneda.valor(this.index);
                     r = IntStream.range(0,nue+1).boxed()
                           .collect(Collectors.toList());
                     Collections.reverse(r);
              }
              return r;
       }
```

```
       public MonedaProblem vecino(Integer a) {
              MonedaProblem r;
              if(this.valorRestante() == 0) r =
                     MonedaProblem.last();
              else r = MonedaProblem.of(this.index()+1,
                     this.valorRestante()-
                           a*Moneda.valor(this.index()));
              return r;
       }
       …
}
```

A partir del aquí el algoritmo de Programación Dinámica de Reducción. Las ideas fundamentales siguen los mismos pasos explicados anteriormente en la implementación genérica:

Necesitamos el tipo *Spm* que representa un par acción-valor de la función objetivo. Es lo que hemos llamado antes solución parcial.

```
public class MonedaPD {
	public static record Spm(Integer a,Integer weight)
		implements Comparable<Spm> {
		public static Spm of(Integer a, Integer weight) {
			return new Spm(a, weight);
		}
		@Override
		public int compareTo(Spm sp) {
			return this.weight.compareTo(sp.weight);
		}
	}
```

```
	public static Integer maxValue = Integer.MIN_VALUE;
	public static MonedaProblem start;
	public static Map<MonedaProblem,Spm> memory;

	public static SolucionMonedas pd(Integer initialValue) {
		MonedaPD.maxValue = Integer.MIN_VALUE;
		MonedaPD.start =MonedaProblem.of(0,initialValue);
		MonedaPD.memory = new HashMap<>();
		pd(start,0,memory);
		return MonedaPD.solucion();
	}
```

El tratamiento de los casos base

```
public static Spm pd(MonedaProblem vertex,
	Integer accumulateValue,
	Map<MonedaProblem,Spm> memory) {
	Spm r;
	if(memory.containsKey(vertex)) {
		r = memory.get(vertex);
	} else if(vertex.index() == MonedaProblem.n) {
		r = null;
		if (MonedaProblem.goalHasSolution().test(vertex))
{
			r = Spm.of(null, 0);
			memory.put(vertex, r);
			if (accumulateValue > MonedaPD.maxValue)
				MonedaPD.maxValue = accumulateValue;
			}
			memory.put(vertex,r);
```

Y el tratamiento de los casos recursivos

```
} else {
       List<Spm> soluciones = new ArrayList<>();
       for(Integer a:vertex.acciones()) {
              Double cota = accumulateValue +
                     MonedasHeuristica.cota(vertex,a);
              if(cota <= MonedaPD.maxValue) continue;
              Integer ac = accumulateValue+
                     a*Moneda.valor(vertex.index());
              Spm s = pd(vertex.vecino(a),ac,memory);
              if(s!=null) {
                     Spm sp = Spm.of(a,s.weight()+
                            a*Moneda.valor(vertex.index()));
                                   soluciones.add(sp);
              }
       }
       if(!soluciones.isEmpty()){
              r = soluciones.stream().filter(s->s != null)
                  .max(Comparator.naturalOrder()).orElse(null);
              memory.put(vertex,r);
       }
}
return r;
}
```

La Programación Dinámica de Reducción es una técnica recursiva y con memoria. La memoria es en este caso *Map<MonedaProblem,Spm>.* En esta memoria se irán acumulando las soluciones parciales de los diferentes problemas resueltos.

Por cada problema se pregunta en primer lugar si ya ha sido resuelto lo que significa comprobar si está en la memoria o no. Si no ha sido resuelto se pregunta si es un problema final y si lo es, si tiene solución, lo cual implica ver si cumple la restricción indicada. Si es así , se actualiza el mejor valor con el valor acumulado del camino y se pone la solución en la memoria. En los problemas cuyos caminos son de tipo Sum la solución de los problemas finales que cumplen la restricción es *(null,0.).* Y si no cumple la restricción *null*. Para los problemas cuyos caminos asociados es de tipo Last la solución de los problemas que cumplen la restricción es *(null,v).* Donde v es una propiedad del problema.

Si el problema no ha sido resuelto y no es final se resuelven los problemas vecinos para obtener una lista de soluciones parciales, *soluciones,* según cada una de las alternativas. Hay una solución parcial asociada a cada

acción que se obtiene a partir de la solución parcial del vecino y del peso asociado a esa arista. En los casos de caminos Sum, como en este caso, la relación es:

```
Spm sp = Spm.of(a,s.weight()+a*Moneda.valor(vertex.index()));
```

Si es camino fuera de tipo Last la relación sería:

```
Spm sp = Spm.of(a,s.weight);
```

El algoritmo filtra las ramas que conduzcan a peores soluciones usando la función de cota:

```
Double cota=accumulateValue+MonedasHeuristica.cota(vertex,a);
if(cota < MonedaPD.maxValue) continue;
```

El método *MonedasHeuristica.cota(vertex,a)* debe tener en cuenta si el vértice alcanzado desde vertex tomando la alternativa a ya tiene solución en la memoria. Si es así sustituimos la heurística por el peso de la solución encontrada.

A partir de la memoria podemos obtener la lista de acciones que definen el camino óptimo y a partir de ahí obtenemos la solución:

```
public static SolucionMonedas solucion(){
	List<Integer> acciones = new ArrayList<>();
	MonedaProblem v = MonedaPD.start;
	Spm s = MonedaPD.memory.get(v);
	while(s.a() != null) {
		acciones.add(s.a());
		v = v.vecino(s.a());
		s = MonedaPD.memory.get(v);
	}
	return SolucionMonedas.of(acciones);
}
```

Los detalles completos pueden verse en el repositorio.

Programación Dinámica

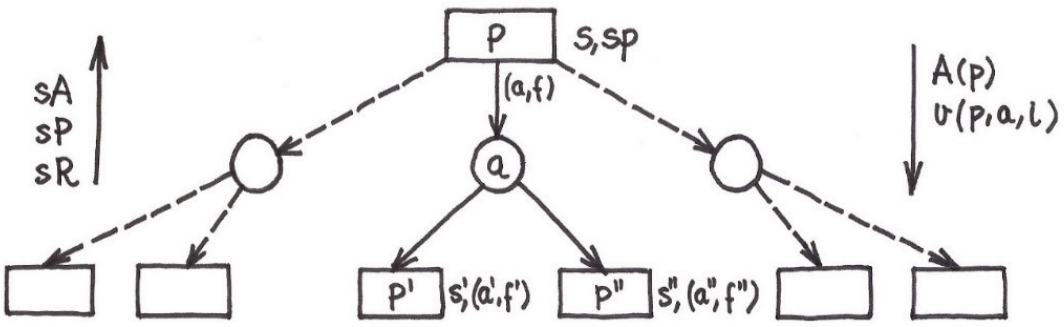
La Programación Dinámica es muy similar a la Programación Dinámica de Reducción. Es, también, un algoritmo recursivo con memoria. La diferencia ahora está en que los problemas pueden tener varios subproblemas y por lo tanto necesitamos un *hipergrafo* virtual en lugar de un grafo. Los hipergrafos que usaremos aquí son los más sencillos como ya vimos: el cambio está en que cada vértice tras tomar una acción puede tener varios vecinos.

Otra diferencia importante frente a las técnicas anteriores cuyas soluciones estaban asociadas a caminos es que ahora las soluciones están asociadas a árboles. Estos árboles tienen una raíz que es el problema por resolver, cada vértice tiene otros árboles vecinos según la acción que constituye la solución y todas las hojas son casos base.

Ya vimos anteriormente los detalles de los grafos virtuales.

Implementación de la Programación Dinámica

La *Programación Dinámica* general tiene todos los elementos que la *Programación Dinámica de Reducción* por lo que su implementación es similar. De la misma forma la memoria del algoritmo es de la forma *Map<P,Sp<A>>.* Donde *P* es el tipo que implementa el problema y *Sp<A>* un par (a, w). Pero ahora tenemos *hipergrafos* como se puede ver en el siguiente esquema. Tampoco existe vértice final. Ahora tenemos casos base y el valor de la función objetivo de su solución.



El conjunto de vértices del hipergrafo será el conjuto de valores del tipo P que cumplen el predicado $isValid(p)$. Por otra parte a partir de un vértice se pueden alcanzar otros al tomar una acción determinada según indica la función $neighbors(p, a)$. En cada vértice habrá un conjunto A_p de acciones posibles. Una acción posible es aquella que nos lleva a vecinos válidos. Con esos elementos hemos obtenido un *hipergrafo* que llamaremos *hg.* El hipergrafo *hg* definido es virtual en el sentido que sus vértices están bien definidos pero no tienen que estar en memoria.

Podemos asignar pesos a los vértices y las aristas del hipergrado definido. Como cada vértice tiene asociado un problema y podemos asignarle un peso que es el valor objetivo de la solución del problema. Cada arista que sale de un problema podemos caracterizarla por el par (p, a). A cada arista le podemos asignar el peso de la solución obtenida si tomamos la alternativa asocida a la arista. Sean $ws(p)$ y $ws(p, a)$ los pesos del vértice asociado al problema p y de la solución si tomamos la alternativa a. Entre estos pesos existen relaciones recursivas.

Hay problemas cuya solución es conocida y por lo tanto su peso. Son los llamados casos base. Sea $b(p)$ un predicado que nos indicada si un problema es un caso base. El peso de la función objetivo para un caso base será $ws_b(p)$ que puede ser null, que representaremos por $\perp$, si el problema no tiene solución.

Si $wsv(p)$ es el peso óptimo de la función objetivo en el vértice p y $wse(p, a)$ el peso de la arista que parte de p tomando la alternativa a entonces tenemos las relaciones:

$$wsv(p) = \begin{cases} ws_b(p), & b(p) \\ \min\limits_{a \in A_p} wse(p, a), & !\, b(p) \end{cases}$$

$$wse(p, a) = \begin{cases} \perp, & \perp \in wsv(neighbors(p, a)) \\ w(wsv(neighbors(p, a))), & \perp \notin wsv(neighbors(p, a)) \end{cases}$$

Siendo *w(...)* una función que combina los pesos de las soluciones de los vecinos para obtener el pseo de la solución cuando tomamos una determinada acción a.

La acción óptima para un vértice p que no sea un caso base, que representaremos por $opa(p)$, es $opa(e) = \underset{a \in A_p}{argmin}\, wse(p, a)$. Si el problema fuera de maximizar se sustituiría por $\underset{a \in A_p}{argmax}\, wse(p, a)$.

El arbol óptimo $opt(p)$ para un vértice p se define de forma similar como

$$opt(p) = \begin{cases} tb(p), & b(p) \\ tr(p, opa(p)), & !\, b(p) \end{cases}$$

Donde tb construye un árbol con un vértice que es caso base y tr un arbol definido por un vértice y una de sus acciones posibles.

La solución óptima de un problema $ops(p)$:

$$ops(p) = \begin{cases} s_b(p), & b(p) \\ s(ops(neighbors(p, opa(p)))), & !\, b(p) \end{cases}$$

Donde $s(\dots)$ combina las soluciones optimas de los vecinos.

Para abordar de forma eficiente el algoritmo explicitado diseñamos el tipo *Sp<E>* que llamaremos *solución parcial*.

```
record Sp<E>(Double weight, E edge)
       implements Comparable<Sp<E>> {…}
```

Cada solución parcial para el problema viene definida por la alternativa que define esa solución y el peso de la función objetivo para la misma. El tipo *Sp<E>* es un par formado por el peso de la solución asociada a un problema y la arista asociada a la acción correspondiente. Cada vértice tiene asociadas varias soluciones parciales: una por cada acción posible, algunas de las cuales pueden ser *null*. La solución parcial óptima para un problema p es un par que contiene $(wsv(p), (p, opa(p))$.

Los vértices del hipergrafo deben implementar el tipo:

```
interface HyperVertex<V extends HyperVertex<V, E, A, S>,
             E extends HyperEdge<V,E,A,S>, A, S>{
       Boolean isValid();
       List<A> actions();
       List<V> neighbors(A a);
       E edge(A a);
       Boolean isBaseCase();
       Double baseCaseSolutionWeight();
       public S baseCaseSolution();
       public V me();
       …
}
```

El método *actions()* calcula el conjunto A_p que lo devuelve en forma de lista para dar un orden a las acciones. El método *neighbors(a)* implementa la función $neighbors(p, a)$. El método *egde(a)* construye la arista *(p,a).* El método *isValid()* indica si el problema es válido. Los métodos *isBaseCase(), baseCaseSolutionWeight(), baseCaseSolution()* nos indican, respectivamente, si el problema es un caso base y si lo es cual es el peso de su solución si la tiene o null si no la tiene y su solución. Son las funciones y predicados representadas por $b(p), ws_b(p)$ y $s_b(p)$.

Y las aristas del hipergrafo deben implmentar el tipo:

```
interface HyperEdge<V extends HyperVertex<V, E, A, S>,
             E extends HyperEdge<V,E,A,S>, A, S>{
       V source();
       A action();
       Double weight(List<Double> solutions);
       S solution(List<S> solutions);
       E me();
       default List<V> targets() {
              return this.source().neighbors(this.action());
       }
       …
}
```

Donde los métodos *source(), action()* y *targets()* nos indican el origen, acción y destinos de la arista. El método *weight(…)* combina los pesos de los vecinos. Implementa la función *w(…).* Y el método *solution(…)* que combina las soluciones de los vecinos implementa la función *s(…).*

El esquema recursivo de la *Programación Dinámica* para el calculo del valor óptimo de la función objetivo es:

$$pd(p) = \begin{cases} m[p], & p \in m \\ ws_b(p), & b(p) \\ \min\limits_{a \in A_p} wse(p,a), & !\,b(p) \end{cases}$$

$$wse(p,a) = \begin{cases} \bot, & \bot \in wsv(neighbors(p,a)) \\ w(wsv(neighbors(p,a))), & \bot \notin wsv(neighbors(p,a)) \end{cases}$$

La alternativa óptima se puede calcular de foram similar. Ambos, alternativa óptima y valor óptimo de la función objetivo, se pueden calcular a la vez acumulándolos en un par (a, w). Este par es la solcuión parcial que hemos representado por el tipo *Sp<A>.*

El algoritmo genérico que busca la solución paracial óptima para cada vértice se puede implementar mediante un método *sp()* en el tipo *HyperVertex* y otro en el tipo *HyperEdge* que colaboran entre si. El método en el tipo *HyperVertex es:*

```
public default Sp<E> sp() {
	Sp<E> r = null;
	if (this.isSolved())
		r = this.spData();
	else {
		if (this.isBaseCase()) {
			Double br = baseCaseWeight();
			if (br != null) r = Sp.of(br,null);
		} else {
			r = this.edgesOf().stream()
			.map(e->e.sp())
			.peek(e->this.setAllSpData(e))
			.filter(s -> s != null)
			.min(this.orderData())
			.orElse(null);
	}
	this.setSpData(r);
	}
	return r;
}
default Boolean hasSolution() {
	return this.sp() != null;
}
```

El método en el tipo *HyperEdge es:*

```
public default Sp<E> sp() {
	Sp<E> r = null;
	if(this.hasSolution()) {
		Double weight =
			this.weight(this.targets().stream()
				.map(v -> v.weight()).toList());
		r = Sp.of(weight, me());
	}
	return r;
}
```

Ambos métodos colaboran para resolver el problema y acceden a la memoria compartida en la clase *Data* que veremos abajo.

```
public default Boolean isSolved() {
	return Data.<V,E>of().solved(this.me());
}
```

```
public default Double weight() {
	if(this.sp() != null)
		return this.sp().weight();
	else
		return null;
}
```


Una implementación posible es diseñar una clase clase *Data* que contenga la memoria y los elementos compartidos.

```
public class Datos<V extends HyperVertex<V, E, A, ?>,E extends
HyperEdge<V,E,A,?>,A> {

	private static Object datos = null;

	@SuppressWarnings("unchecked")
	public static <V extends HyperVertex<V, E, A, ?>,
	E extends HyperEdge2<V,E,A,?>,A> Datos<V, E, A> get() {
		if(datos == null) datos = new Datos<>();
		return (Datos<V, E, A>) datos;
	}
	private Datos() {
		super();
	}
	private Map<V,Sp<E>> memory = new HashMap<>();
	private SetMultimap<V,Sp<E>> allProblems =
		SetMultimap.create();

	public static enum DpType{Max, Min}
	public static DpType type = DpType.Min;

	public SimpleDirectedGraph<Union<V, E>, DefaultEdge>
		graph() {…}

	public static <V extends HyperVertex<V, E, ?, ?>,
		E extends HyperEdge<V,E,?,?>>
	void toDotHypergraph(SimpleDirectedGraph<Union<V,E>,
		DefaultEdge> g, String file, V initial) {…}

	public static <V extends HyperVertex<V, E, ?, ?>,
		E extends HyperEdge<V,E,?,?>>
	void toDotAndOr(SimpleDirectedGraph<Union<V,E>,
		DefaultEdge> g,String file, V initial) {…}
```

La clase *Data* tiene un método de factoría para que funcione como un *Singleton*. Es decir, para que siempre devuelva el mismo objeto. La clase, además, puede construir el grafo bipartito, sus vértices pueden ser de dos tipos diferentes, asociados a la ejecución y sus vistas gráficas como grafo And/Or o como Hipergrafo.

Para completar el algoritmo, debemos implementar el método *solution* del tipo *HyperVertex.*

```
default public S solution() {
	if (this.isBaseCase())
		return this.baseCaseSolution();
	else
		return this.sp().edge().solution();
}
```

Y en el tipo *HyperEdge*.

```
default S solution() {
	return solution(this.targets().stream()
			.map(v->v.solution()).toList());
}
```

Los métodos *weight()* en los tipos *HyperVertex* e *HyperEdge* obtienen los pesos a partir de las soluciones parciales. El tipo *HyperVertex* tiene implementado un método, *graphTree()*, para obtener el árbol óptimo asociado a un vértice.

El resto de los detalles pueden verse en el repositorio.

Usos de la Programación Dinámica

El algoritmo de *Floyd,* que ya vimos antes al presentar los hipergrafos, busca calcular el camino mínimo en un grafo g de tipo *Graph<V,E>.* Este algoritmo construye un *hipergrafo hg* con los datos del grafo de entrada y sobre él calcula el árbol mínimo. Recordemos los detalles de nuevo.

Asumimos que cada vértice del grafo está indexado mediante un entero comprendido en $[0, n-1]$ siendo n es el número de vértices. Queremos encontrar el *Camino Mínimo* entre dos ciudades dadas que representaremos por i, j. Este problema se puede resolver de muchas otras formas como hemos visto. Aquí vamos a resolver el problema generalizándolo a este otro: encontrar el *Camino Mínimo* de i a j usando como camino intermedio ciudades cuyos índices estén en el conjunto $[0, k]$. Con este planteamiento cada problema lo podemos representar por (i, j, k). Para que sea válido i, j toman valores en $[0, n-1]$ y k en $[0, n]$. El

valor de $k = n$ indica que el camino no contiene ninguna ciudad intermedia.

Ahora podemos modelar el problema mediante un *hipergrafo* cuyos vértices son los problemas generalizados (i, j, k). Las acciones (alternativas) posibles definirán las *hiperaristas*. Para cada problema tenemos dos alternativas $\{true, false\}$. La acción es de tipo Boolean. La primera alternativa representa pasar por la ciudad k. La segunda no pasar.

La forma de usar lo anterior es implementar los vértices en la clase *FloydVertex* e implementar el tipo de las aristas *FloydEdge* que ya vimos anteriormente en la sección *Problema de Floyd* más arriba.

El uso ahora es:

```
SimpleWeightedGraph<Ciudad, Carretera> graph =
        leeDatos("./ficheros/andalucia.txt");
IntegerVertexGraphView<Ciudad, Carretera> graph2 =
         IntegerVertexGraphView.of(graph);
Integer origen = graph2.getIndex(Ciudad.ofName("Sevilla"));
Integer destino = graph2.getIndex(Ciudad.ofName("Almeria"));

FloydVertex.graph = graph2;
FloydVertex.n = graph2.vertexSet().size();
FloydVertex p = FloydVertex.initial(origen,destino);

Data.type = Data.DpType.Min;

GraphPath<Integer, SimpleEdge<Integer>> gp = p.solution();

GraphTree<FloydVertex, FloydEdge, Boolean,
       GraphWalk<Integer, SimpleEdge<Integer>>> t =
       p.graphTree();

SimpleDirectedGraph<Union<FloydVertex,FloydEdge>,
       DefaultEdge> g = p.datos().graph();

Datos.toDotHypergraph(g, "ficheros/hyperGraph.gv", p);
Datos.toDotAndOr(g, "ficheros/andOrGraph.gv", p);
```

Como vemos los pasos son:

- Leemos de un fichero un grafo *graph*
- Obtenemos una vista el grafo cuyos vértices son enteros

- Obtener el primer vértice *p* del grafo virtual al partir de las ciudades origen y destino
- Obtener la solución del problema
- Obtener el árbol solución asociado al vértice que indiquemos
- Obtener las vistas del hipergrafo e ejecución como grafo *And/Or* o como hipergrafo

Implementación directa de la Programación Dinámica

Como en la versión de reducción necesitamos en primer lugar un tipo que representa las soluciones parciales

```
record Spf(Boolean a,Double weight) implements
              Comparable<Spf> {

       public static Spf of(Boolean a, Double weight) {
              return new Spf(a, weight);
       }

       @Override
       public int compareTo(Spf sp) {
              return this.weight.compareTo(sp.weight);
       }
}
```

Un tipo que representa los problemas siguiendo las ideas de los vértices del hipergrafo, pero sin comprometerse a implementar ninguna interface.

```
record FloydProblem(Integer i,Integer j,Integer k) {

       public static FloydProblem of(Integer i,Integer j) {
              return new FloydProblem(i,j,0);
       }

       public static FloydProblem of(Integer i,
                     Integer j,Integer k) {
              return new FloydProblem(i,j,k);
       }

       public List<Boolean> actions() {
              if(this.isBaseCase()) return List.of();
              else if(i==k || k==j) return List.of(false);
              else return List.of(false,true);
       }
```

```
        public List<FloydProblem> neighbors(Boolean a) {
          List<FloydProblem> r=null;
          if(!a) r = List.of(FloydProblem.of(i,j,k+1));
          else r=List.of(FloydProblem.of(i,k,k+1),
               FloydProblem.of(k, j, k+1));
          return r;
        }

        public Boolean isBaseCase() {
               return this.i.equals(this.j) ||  k == n;
        }
```

```
        public Double baseCaseSolution() {
               Double r = null;
               if(this.i.equals(this.j)) r = 0.;
               else if(k ==n && FloydPD.graph.containsEdge(
                            this.i, this.j)){
                      Double w = FloydPD.graph
                            .getEdge(i, j).weight();
                      r = w;
               } else if(k ==n && !FloydPD.graph
                            .containsEdge(this.i, this.j)) {
                      r = null;
               }
               return r;
        }
}
```

Con estos elementos implementamos el algoritmo de PD. Vamos a hacer ahora una implementación centralizada frente a la implementación genérica anterior donde el algoritmo estaba distribuido entre vértices y aristas y un objeto compartido que tenía la memoria entre otros elementos.

```
public class FloydPD {
        public static Graph<Integer,SimpleEdge<Integer>> graph;
        public static Integer n;
        public Map<FloydProblem,Spf> solutionsTree;
        public static FloydProblem startVertex;

        private FloydPD(FloydProblem startVertex) {
               FloydPD.startVertex = startVertex;
               this.solutionsTree = new HashMap<>();
        }
```

En primer lugar, el tratamiento de los casos base

```
	public Spf search(FloydProblem actual) {
		Spf r = null;
		if (this.solutionsTree.containsKey(actual)) {
			r = this.solutionsTree.get(actual);
		} else if (actual.isBaseCase()) {
			Double w = actual.baseCaseSolution();
			if(w!=null) r = Spf.of(null,w);
			else r = null;
			this.solutionsTree.put(actual, r);
```

Y los casos recursivos.

```
		} else {
			List<Spf> sps = new ArrayList<>();
			for (Boolean a:actual.actions()) {
				List<Spf> spNeighbors =
					new ArrayList<>();
				Double s = 0.;
				for (FloydProblem neighbor :
						actual.neighbors(a)) {
					Spf nb = search(neighbor);
					if (nb == null) {
						spNeighbors = null;
						break;
					}
					spNeighbors.add(nb);
					s+=nb.weight();
				}
				Spf spa = null;
				if(spNeighbors != null) {
					spa = Spf.of(a,s);
				}
				sps.add(spa);
			}
			r = sps.stream()
				.filter(s -> s != null)
				.min(Comparator.naturalOrder())
				.orElse(null);
			this.solutionsTree.put(actual, r);
		}
		return r;
	}
```


```
public List<Integer> solucion(FloydProblem p) {
	Spf s = this.solutionsTree.get(p);
	if(s.a() == null) {
		List<Integer> r = new ArrayList<>();
		r.add(p.i());
		r.add(p.j());
		return r;
	}else {
		List<FloydProblem> vc = p.neighbors(s.a());
		List<Integer> ls0 = solucion(vc.get(0));
		if(s.a()) {
			List<Integer> ls1 = solucion(vc.get(1));
			ls0.remove(ls0.size() - 1);
			ls0.addAll(ls1);
		}
		return ls0;
	}
}
```

La solución al problema, en este caso, es una lista de enteros que contiene los vértices del camino óptimo.

LocalSearch y SimulatedAnnealing

Junto a los algoritmos anteriores es conveniente conocer otros algoritmos capaces de encontrar el óptimo en un problema con restricciones. Son algoritmos aproximados que explotan el conocimiento de las vecindades para encontrar la mejor solución. Para muchos problemas son candidatos ideales.

Estos algoritmos necesitan modelar un *entorno*. Llamamos entorno a un estado, con sus propiedades, más un conjunto de vecinos que tendrán estados cercanos desde algún punto de vista.

La idea es modelar un entorno mediante un grafo virtual cuyos vértices son los estados y las aristas que nos llevarán a sus vecinos.

Debemos disponer de una función que se evalúe sobre los vértices y devuelva un valor real. Sea esa función $f(v)$.

La versión más sencilla de la búsqueda local es un algoritmo iterativo que toma en cada paso la arista que le conduce al vecino v' tal que el incremento $\Delta = f(v') - f(v)$ sea negativo y su valor absoluto el mayor posible. El algoritmo continúa mientras que $|\Delta| > \varepsilon$. El algoritmo es similar en espíritu al método del gradiente que se utiliza en matemáticas para buscar óptimos de funciones. Este algoritmo, como el del gradiente, busca mínimos locales. En cada caso encontramos el mínimo de la cuenca de atracción del primer valor del estado. Para mejorarlo se puede repetir la búsqueda con distintos puntos iniciales.

Existen variantes como la búsqueda local aleatoria que elige aleatoriamente uno de los vecinos de entre los que cumplan $\Delta = f(v') - f(v)$ sea negativo.

Este tipo de algoritmos se les suele denominar también algoritmos de *Hill Climbing* porque emulan el proceso de escalada de una montaña cuando estamos buscando el máximo de la función objetivo. Recordamos que pasar del máximo al mínimo, o viceversa, siempre los podemos consiguir cambiando el signo de la función objetivo.

El algoritmo de *SimulatedAnnealing,* enfriamiento simulado, ya lo vimos al estudiar algoritmos genéticos por lo que no lo repetiremos aquí. Sólo señalar que la transición a un vecino, dentro de un entorno, hace el papel de operador de mutación. Este algoritmo es mejor para buscar mínimos globales.

En resumen si disponemos de un entorno modelado mediante un grafo virtual y sabemos no existen múltiples mínimos locales, es decir solo existe un único minimo global, es adecuado usar algoritmos de búsqueda local. Si pueden existir muchos mínimos y queremos buscar el global entonces usaremos algoritmos de enfriamiento simulado o algoritmos genéticos si añadimos un operador de cruce.

Usos de la búsqueda local y SimulatedAnnealing

Veamos un ejemplo para ilustrar estas ideas.

Problema del Viajante

El problema del viajante (*TSP* por sus siglas en inglés (*Travelling Salesman Problem*)), responde a la siguiente pregunta: dado un grafo no dirigido completo y con peso en las aristas ¿cuál es el camino simple y cerrado que visita cada vértice y sea de peso mínimo?

Partimos de un grafo completo (siempre lo podemos completar añadiendo las aristas que falten con peso muy grandes) con n vértices. Asumimos que tenemos una vista de ese grafo cuyos vértices son enteros. Busquemos un entorno. El estado puede ser un camino simple cerrado que

incluya todos los vértices con el primero y el último repetido. Es la lista $[0,1,\dots,n-1,0]$ o una permutación de ella que mantenga 0 en sus dos extremos. A cada estado le asociamos un peso que es el peso del camino cerrado definido.

En la literatura se puede encontrar un entorno para este problema formado por el estado definido anteriormente y los vecinos que se alcanzan con las acciones del siguiente conjunto:

$$A = \{(i,j), i \in 1\,..\,n-2, j \in 0\,..\,n-2 \mid j-i > 2\}$$

Como vemos cada acción es un par de enteros. Asumiendo que el vértice actual contiene un camino c el vecino según (i,j) se obtendrá:

$$neighbor\big(c,(i,j)\big) = c[0,i] + r(c[i,j]) + c[j,n]$$

Donde $c[i,j]$ es la sublista correspondiente, $r(c)$ la lista invertida y $c1 + c2$ la concatenación de listas.

Con estas ideas podemos implementar un grafo virtual:

```
TravelVertex extends ActionVirtualVertex<TravelVertex,
                    TravelEdge, IntPair> {
       List<Integer> camino;
       Double weight;
       Integer n;
       List<IntPair> actions() {
              return Streams2.allPairs(1,n-1,1,n-1)
                           .filter(p->p.second-p.first>2)
                           .collect(Collectors.toList());
       }
       TravelVertexInteger neighbor(IntPair a) {
              return TravelVertex.of(this.graph,
                     AuxiliaryTsp.neighbor(this.graph,
              this.camino,a.first,a.second));
}
       …
```

Donde *AuxiliaryTsp.neighbor* es un método auxiliar para calcular el vecino según la acción especificada.

A partir de aquí el uso de la búsqueda local es de la forma:

```
Graph<Integer,SimpleEdge<Integer>> graph =
AuxiliaryTsp.generate(50);
List<Integer> camino = … ;
TravelVertex e1 = TravelVertex.of(graph, camino);

EGraph<TravelVertexInteger,TravelEdgeInteger> graph2 =
       EGraph.virtual(e1,null)
              .vertexWeight(v->v.weight()).build();

LocalSearch<TravelVertexInteger,TravelEdgeInteger> m =
        LocalSearch.of(graph,0.,3);

TravelVertexInteger v = m.search().get();
```

El algoritmo de búqueda local se instancia con el método de factoría:

```
LocalSearch<V, E> of(EGraph<V, E> graph,Double error,
       Integer n);
```

Donde graph es el grafo que describe el entorno, error es el error máximo permitido de la función objetivo y n el número de aristas que se escogen para elegir un vecino.

Hay una variante de este algoritmo cuando queremos repetirlo varias veces tomando diferentes vértices iniciales y devolviendo el mejor vértice encontrado. En este caso el método de factoría es:

```
V repeat(EGraph<V, E> graph,Supplier<V> start, Double error,
       Integer n, Integer m);
```

Ahora aparecen dos prámetros más: *m* que indica el número de repeticiones *start* que nos proporciona un vértice inicial.

El uso del enfriamiento simulado es similar:

```
Graph<Integer,SimpleEdge<Integer>> graph =
AuxiliaryTsp.generate(50);
List<Integer> camino = … ;
TravelVertex e1 = TravelVertex.of(graph, camino);
EGraph<TravelVertexInteger,TravelEdgeInteger> graph2 =
       Graphs2.simpleVirtualGraphLast(e1,v->v.getWeight());

SimulatedAnnealingSearch<TravelVertexInteger,
TravelEdgeInteger> m =
       GraphAlg.simulatedAnnealing(graph2,e1,e->e.weight);
m.search();
System.out.println(m.bestWeight);
System.out.println(m.bestVertex);
```

Implementación de la búsqueda local y SimulatedAnnealing

La implementación y los detalles de los parámetros del enfriamiento simulado ya los vimos junto a los algoritmos genéticos.

La implementación de la búsqueda local es exactamente un algoritmo voraz que toma en cada paso la acción que hace descender al máximo la función objetivo especificada y termina cuando el valor absoluto del error es menor que una cantidad especificada.

Otras búsquedas en grafos

Junto a los algoritmos anteriores disponemos de otros algoritmos generales para recorrer los vértices de un grafo y hacer búsquedas de vértices con determinadas propiedades. Son los recorridos en anchura, en profundidad y el topológico. Todos ellos, junto con el algoritmo *A**, los algoritmos voraces y la búsqueda local, implementan un iterador. Como hemos estudiado a partir de un iterador podemos obtener un stream y usarlo para recorrer los vértices de un grafo en el orden establecido por el recorrido concreto. Los métodos de stream nos permitirán filtrar, transformar, acumular, encontrar el primero que cumple una propiedad, etc.

Los métodos disponibles en el repositorio son *AStart, AStartRandom, GreedyOnGraph, Greedy, LocalSearch, BreadthSearch, DephtSearch, DephtPostSearch y TopologicalSearch.*

Dedicaremos ahora un poco más de atención a la búsqueda en anchura y profundidad su uso y su implementación.

Usos de la búsqueda en anchura, en profundidad y recorrido topológico

Para usar la búsqueda en anchura necesitamos un grafo y un vértice inicial. A partir de aquí construimos un algoritmo de búsqueda en anchura y lo usamos para responder a preguntas como:

- Encontrar todos los *vértices que están a n pasos del vértice inicial*. La idea es usar el método *stream()* y filtrar los vértices cuya distancia al vértice inicial sea la dada. Este algoritmo dispone del método *Double distanceToOrigen(v).*
- Encontrar los vértices de la *componente conexa* a la que pertenece el vértice inicial. La idea es usar el método *stream()* y acumular todos los vértices en un conjunto.
- Decidir *si hay un camino desde un vértice v1 a otro v2*. Se usa el método *stream()* sobre el algoritmo con vértice inicial *v1* y se pregunta si el segundo se encuentra: es decir *filter(v->v.equals(v2).findFirst().*
- Encontrar un *árbol de recubrimiento* aunque no mínimo. Este árbol estará formado por las aristas contenidas en *Map<V,E> edgeToOrigin* asociadas a cada vértice del grafo.
- Encontrar el *camino mínimo en número de pasos* desde cada vértice hasta el vértice inicial. Para cada vértice este camino viene definido por las aristas consecutivas que se van encontrando en *edgeToOrigin.*
- Encontrar las *componentes conexas* del grafo. La idea es usar *stream()* para acumular todos los vértices de una componente conexa, eliminarlos del grafo y continuar con un vértice del resto del grafo para la segunda componente conexa y así hasta terminar todos los vértices. Las *componentes débilmente conexas* en grafos dirigidos se hacen por el mismo procedimiento.
- Decidir *si un grafo es conexo*. Aplicar el punto anterior y mirar el número de componentes conexas. Para que sea conexo solo debe haber una componente conexa.

Para usar la *búsqueda en profundidad* necesitamos un grafo y un vértice inicial. A partir de aquí construimos un algoritmo de búsqueda en profundidad y lo usamos para responder a preguntas. El recorrido en

profundidad que usaremos normalmente se llama recorrido en *preorden* que implementaremos iterativamente. Asociado al *preorden*, como en el caso de la búsqueda en anchura, existe un árbol de recubrimiento que no tiene por qué ser mínimo y que va construyendo el algoritmo. También existen en *postorden*, *postorden inverso* e *inorden*.

El recorrido en *preorden* usa una pila y visita a cada vértice antes que a sus vecinos con respecto al árbol de recubrimiento.

El recorrido en *postorden* usa una pila y visita a cada vértice después que a sus vecinos con respecto al árbol de recubrimiento.

El recorrido en *postorden* inverso es el inverso del *postorden*.

El recorrido en profundidad en *preorden* se usa para calcular las componentes fuertemente conexas y para clasificar el tipo de aristas en grafos dirigido. No veremos detalles sobre estos temas.

El recorrido topológico necesita un grafo dirigido sin ciclos. Las aristas en el grafo indican precedencia. Los vértices se van generado de tal manera que cualquier vértice viene detrás de los vértices que lo preceden según indican las aristas.

Implementación de la búsqueda en anchura, en profundidad y topológico

Estos algoritmos ya vienen implementados en *jgrapht*. Aquí comentaremos la implementación ofrecida en el repositorio.

Las implementaciones de estos algoritmos que se ofrecen en el repositorio son sencillas. Son algoritmos iterativos que parten de un vértice inicial. En su estado tienen un *Map<V,E>* para guardar las aristas que indican los caminos hacia el vértice inicial. Adicionalmente la búsqueda en anchura tiene una cola y la búsqueda en profundidad una pila.

Como los algoritmos voraces estos algoritmos implementan un iterador que va proporcionando los vértices sucesivamente. El iterador empieza con el vértice inicial en la cola, anchura, o pila, profundidad, y acaba cunado están vacías. En cada paso saca un vértice de la cola o pila, busca

sus vecinos no visitados (lo que no están en las claves del *Map*), los añade a la cola o pila y actualiza el *Map*.

El *postorden* también implementa un iterador y usa dos pilas. En primer lugar, hace el recorrido en *preorden* y cada vez que saca un vértice de la primera pila lo apila en la segunda. Terminado este paso comienza a sacar los vértices de la segunda pila. Para cada vértice hay dos momentos que se llaman *previsita* y *postvisita* que corresponden respectivamente a cuando se sacan de la primera y la segunda pila.

El recorrido topológico se implementa mediante un recorrido en *postorden* inverso. Es en definitiva es como el *postorden* pero sustituyendo la segunda pila por una cola. La primera parte es igual, pero en la segunda se van sacando de la cola.

Cuándo usar cada técnica

Los *algoritmos voraces, greedy,* parten de un grafo, real o virtual, un vértice inicial y un vértice final o un predicado que determina el conjunto de vértices finales. En cada vértice se aplica una determina acción, defina mediante una función, que llamaremos acción voraz. Son algoritmos iterativos que pretenden encontrar un camino del vértice inicial al final pero no siempre lo encuentran. Dependiendo de la calidad de la acción voraz el algoritmo puede encontrar un camino o no y en su caso el peso del camino puede estar más o menos cercano al óptimo.

Los algoritmos *A** parten de un grafo, real o virtual, un vértice inicial, un vértice final y una heurística. Buscan el camino mínimo del vértice inicial al final. La definición del peso del camino es muy importante para la definición de camino mínimo.

Los grafos que necesitamos pueden tener caminos cerrados y entre los vértices no tiene porqué existir una noción de tamaño.

El algoritmo nos devuelve un *GraphPath<V,E>* que posteriormente habrá que transformar en la solución buscada.

Los algoritmos *de Backtracking* parten de un grafo, real o virtual, un vértice inicial, un vértice final y una heurística. Pueden definirse casos base en vez del vértice final pero ya hemos visto como añadir aristas al grafo para eliminar los casos base. Usan funciones de cota que se definen a partir de la heurística. Buscan el camino mínimo, o máximo, del vértice

inicial al final. La definición del peso del camino es muy importante como antes.

Los grafos que necesitamos ahora no pueden tener caminos cerrados y entre los vértices tiene existir una noción de tamaño tal que el tamaño de los hijos es menor que el del padre.

El algoritmo nos devuelve una *solución del problema, un número dado de soluciones o todas las soluciones.* Es el algoritmo adecuado cuando queremos calcular más de una solución, todas las soluciones o cuando queriendo calcular la óptima no hay solapamiento de soluciones o muy poco.

Los algoritmos *de Programación Dinámica de Reducción* parten de un grafo, real o virtual, un vértice inicial, un vértice final y una heurística. Pueden definirse casos base en vez del vértice final pero ya hemos visto como añadir aristas al grafo para eliminar los casos base. Usan funciones de cota que se definen a partir de la heurística. Buscan el camino mínimo, o máximo, del vértice inicial al final. La definición del peso del camino es muy importante como antes.

Los grafos que necesitamos ahora no pueden tener caminos cerrados y entre los vértices tiene existir una noción de tamaño tal que el tamaño de los hijos es menor que el del padre.

La programación dinámica, a diferencia del *Backtracking*, usa memoria. Es adecuada, frente al *Backtracking*, donde haya muchos caminos del vértice inicial al final con vértices compartidos.

El algoritmo nos devuelve un *GraphPath<V,E>* que posteriormente habrá que transformar en la solución buscada.

Los algoritmos *de Programación Dinámica de General* parten de un hipergrafo, real o virtual, un vértice inicial y un conjunto de casos base. Buscan el árbol mínimo, o máximo, del vértice inicial a los casos bases.

Los hipergrafos que necesitamos ahora no pueden tener caminos cerrados y entre los vértices tiene que existir una noción de tamaño tal que el tamaño de los hijos es menor que el del padre.

La programación dinámica general usa memoria.

El algoritmo nos devuelve un *GraphTree<V,E>* que posteriormente habrá que transformar en la solución buscada.

Lo *algoritmos de búsqueda local* y *SimulatedAnnealing* necesitan modelar un *entorno*. Llamamos entorno a un estado, con sus propiedades, más un conjunto de vecinos que tendrán estados cercanos desde algún punto de vista. Un entorno se puede modelar mediante un grafo virtual cuyos vértices son los estados y las aristas que nos llevarán a sus vecinos. En ambas técnicas debemos disponer de una función que se evalúe sobre los vértices y devuelva un *Double*. Sea esa función $f(v)$.

La búsqueda local es un algoritmo voraz que toma en cada paso la arista que le conduce a un vecino v' tal que el incremento $\Delta = f(v') - f(v)$ sea negativo y su valor absoluto el mayor posible. *SimulatedAnnealing* es un algoritmo aleatorio que tiene un funcionamiento descrito en al sección de *Implementación de búsqueda local y SimulatedAnnealing*.

Los algoritmos de búsqueda en anchura, profundidad y orden topológico se utilizan para responder a las preguntas descritas en la sección *Implementación de la búsqueda en anchura, profundidad y topológica*.

Un catálogo de problemas

Veamos aquí un catálogo de problemas que pueden ser resueltos mediante diversas técnicas. Las soluciones serán más o menor detalladas según los casos.

Recubrimiento de vértices

El *Problema del Recubrimiento de Vértices* (*Vertex Cover*) de un grafo consiste en buscar un subconjunto mínimo de vértices tal que cada arista del conjunto es incidente al menos a uno de los vértices escogidos y la suma de sus pesos es mínima. Escogiendo las variables binarias $x_j,\ j \in [0, m)$ que tomarán el valor 1 si el vértice x_j es escogido, el Problema del *Vertex Cover* puede ser formulado como:

$$\min \sum_{j=0}^{m-1} w_j x_j$$
$$x_u + x_v \geq 1, \quad (u, v) \in E$$
$$bin\ x_j, \quad j \in [0, m)$$

Donde por $(u, v) \in E$ queremos representar el conjunto de aristas del grafo y para cada arista sus respectivos extremos.

El problema puede ser resuelto mediante algoritmos genéticos escogiendo un cromosoma binario de tamaño m y función de fitness en función del vector de decode d:

$$ft(d) = -\sum_{j=0}^{m-1} w_j d[j] - k \sum_{i,j=0|(i,j)\in g}^{m-1} dge(d[i] + d[j] - 1)$$

Para resolverlo por las otras técnicas debemos imaginar un grafo de problemas. Este grafo es distinto al grafo original. Las propiedades de los vértices serían:

- *j*: Integer, con valores es *[0,m),*
- *ve:* Set<Integer>, índices de vértices ya escogidos

Las alternativas enteras con valores 0, 1. Se escogería 1 solo en el caso que j no sea vecino a alguno de los vértices ya escogidos.

Los pesos de las aristas $a * w_j$, los problemas finales vendrán definidos por *j == m*.

Como una primera aproximación podemos tomar la heurística *(va,p,vf)->0.* Se propone buscar una mejor heurística.

Problema de la Mochila

El problema de la *Mochila* parte de una lista de objetos L de tamaño. Cada objeto ob_i de la lista es de la forma $ob_i = (w_i, v_i, m_i)$ dónde w_i, v_i, m_i son, respectivamente, su peso, su valor unitario y el número de unidades disponibles. La mochila tiene una capacidad C. El problema busca ubicar en la mochila el máximo número unidades de cada objeto, teniendo en cuenta las disponibles, que quepan en la mochila para que el valor de estos sea máximo. Si x_i es el número de unidades del objeto i en la mochila el problema puede modelarse como:

$$\max \sum_{i=0}^{n-1} x_i v_i$$
$$\sum_{i=0}^{n-1} x_i w_i \leq C$$
$$x_i \leq m_i, \quad i \in [0, n-1]$$
$$int \quad x_i, \quad i \in [0, n-1]$$

El problema puede ser resuelto mediante algoritmos genéticos escogiendo un cromosoma de Rango de tamaño n, con valores en el rango $[0, m_i]$ para cada valor de i y función de fitness:

$$ft(d) = -\sum_{i=0}^{n-1} d[i] * v_i - k\, dle(-C + \sum_{i=0}^{n-1} d[i] * w_i)$$

Para obtener un grafo de problemas lo generalizamos:

Objeto:

- Valor: Integer, *v*
- Peso: Integer, *w*
- ValorUnitario: Double, derivada, v/w
- Objetos: List<Objeto>, compartida

MochilaVertex:

Propiedades:

- i: Integer, básica
- pr: Integer, Pero restante, básica
- n: Integer, compartida
- C: Integer, compartida
- t: Integer, derivada n-i

Interpretación:

Encontrar la elección de monedas cuyo peso es menor que pr y su valor es máximo teniendo en cuenta solo los objetos de i hasta el final

Igualdad

- Dos problemas son iguales si lo son i,pr

Es válido

- i>=0,i<=n,pr>=0

Factoría:

- inicial(): Crea el problema (0,C)
- final(): Crea el problema (n,0)
- goal(v) = p.i ==n

Casos base:

1. p.pr == 0
2. p.i == n-1

Solución caso Base

1. Tiene solución
2. Tiene solución

Acciones:

Sea $n_i = \frac{pr}{w_i}$. Debido a que $pr - a * w_i \geq 0$

1. $A_i = \{0\}$
2. $A_i = \{n_i\}$
3. $A_i = \{0..\min(m_i, n_i)\}$, caso general

Vecino

Caso general: $neighbord(a) = (i + 1, pr - a * w_i)$

Casos Base 1: $neighbord(0) = (n, 0)$

Casos Base 2: $neighbord(n_i) = (n, pr - a * w_i)$

Peso de la arista

Peso: $w = a * v_i$

Peso del camino:

Suma de los pesos de las aristas

Solución Voraz

- Ordenar los objetos por la razón $^{v_i}/_{w_i}$ de mayor a menor
- Acción: $n_i = \frac{vr}{w_i}$

Heurística:

- Ordenar las monedas por la razón $^{v_i}/_{w_i}$ de mayor a menor
- Acción: $n_i = \frac{vr}{w_i}$, siendo de tipo real y la división real

Solución:

- La solución más adecuada serían dos *Multiset<Integer>* que indicarían el número de unidades de cada objeto
- La solución se calcula a partir de los detalles del camino óptimo.

El código se puede encontrar en el repositorio

Problema de las monedas etruscas

Tenemos un sistema monetario donde las conchas marinas hacen de monedas. Existen *n* tipos de conchas/monedas. Cada tipo *i* tiene un valor, v_i, y un peso, w_i. El Banco Central Etrusco (BCE) sólo admite cambios por valor *V*. Es decir, los clientes pueden dar dinero por valor total *V*, y a cambio se les devuelve la misma cantidad en las monedas que el cliente desee. Nuestro propósito es dar las monedas con mayor peso total, y a cambio recibir las monedas con menor peso.

En definitiva, debes encontrar una combinación de monedas con valor total *V*, y donde el peso total sea el máximo. Y, por otro lado, también debes

encontrar una combinación por valor V y donde el peso total sea el mínimo. Se supone que existe un número ilimitado de monedas de todos los tipos. Sólo se admitirá como válida la solución óptima.

En caso de que existan varias soluciones óptimas posibles, debes elegir aquella que use más veces las monedas con menor índice. Si no existe ninguna solución (es decir, no es posible llegar al valor V con los tipos de monedas dados), debes indicarlo.

Un modelo de problema es escoger variables x_i, y_i que indiquen el número de unidades de cada tipo elegidas para entregar y para recibir. Podemos deducir que el número máximo de unidades de la moneda i será $m_i = V/v_i$ donde suponemos valores y división enteros.

Con ese dato el modelo del problema sería:

$$\max \sum_{i=0}^{n-1} x_i w_i - y_i w_i$$

$$\sum_{i=0}^{n-1} x_i v_i = V$$

$$\sum_{i=0}^{n-1} y_i v_i = V$$

$$x_i \leq m_i, \quad i \in [0, n-1]$$
$$y_i \leq m_i, \quad i \in [0, n-1]$$
$$int \quad x_i, y_i, \quad i \in [0, n-1]$$

Este problema se puede resolver adecuadamente por *PLI* o por algoritmos genéticos. La función de fitness es fácil de escribir.

Este problema se puede dividir en dos problemas:

$$\max \sum_{i=0}^{n-1} x_i p_i$$

$$\sum_{i=0}^{n-1} x_i v_i = V$$

$$x_i \leq m_i, \quad i \in [0, n-1]$$
$$int \quad x_i, \quad i \in [0, n-1]$$

$$\min \sum_{i=0}^{n-1} y_i p_i$$
$$\sum_{i=0}^{n-1} y_i v_i = V$$
$$y_i \leq m_i, \quad i \in [0, n-1]$$
$$int \quad y_i, \quad i \in [0, n-1]$$

Para obtener un grafo de problemas lo generalizamos:

Moneda:

- Valor: Integer, v
- Peso: Integer, w
- PesoUnitario: Double, derivada, w/v
- Monedas: List<Moneda>, compartida

MonedasVertex:

Propiedades:

- i: Integer, básica
- vr: Integer, Valor restante, básica
- n: Integer, compartida
- V: Integer, compartida
- t: Integer, derivada n-i

Interpretación:

Encontrar la elección de monedas cuyo valor suma vr y su peso es máximo teniendo en cuenta solo las monedas de i hasta el final

Igualdad

- Dos problemas son iguales si lo son i,vr

Es válido

- i>=0,i<=n,vr>=0

Factoría:

- inicial(): Crea el problema (0,V)
- final(): Crea el problema (n,0)
- goal(v) = p.i ==n // también lo podemos llamar caso base

Casos finales con solución (goalHasSolution)

- v.vr = 0

Casos base:

1. p.vr == 0
2. pi. == n-1

Solución caso Base

1. Tiene solución si vr ==0
2. Tiene solución si vr es divisible por v_i

Acciones:

Sea $n_i = \frac{vr}{v_i}$. Debido a que $vr - a * v_i \geq 0$

- $A_i = \{\}$, si es el problema final
- $A_i = \{0\}$, si vr ==0
- Si p.i. == n-1
  - $A_i = \{n_i\}$ si vr es divisible por v_i
  - $A_i = \{\}$ si vr no es divisible por v_i
- $A_i = \{0..n_i\}$, caso general

Vecino

Caso general: $neighbord(a) = (i + 1, vr - a * v_i)$

Casos Base 1: $neighbord(0) = (n, 0)$

Peso de la arista

Peso: $w = a * w_i$

Peso del camino:

Suma de los pesos de las aristas

Solución Voraz

- Maximizar
  - Ordenar las monedas por la razón $^{p_i}/_{v_i}$ de mayor a menor
  - Acción: $n_i = \frac{vr}{v_i}$
- Minimizar
  - Ordenar las monedas por la razón $^{p_i}/_{v_i}$ de menor a mayor
  - Acción: $n_i = \frac{vr}{v_i}$

Debido a la restricción de igualdad en el modelo puede que no exista solución voraz para algunos valores de V, el valor de las monedas a intercambiar. Pero como solo queremos una solución voraz podemos relajar a restricción sustituyéndola por

$$\sum_{i=0}^{n-1} x_i v_i \leq V'$$

Con $V' > V$. Buscaremos el valor más pequeño de V' tal que exista una solución voraz con

$$\sum_{i=0}^{n-1} x_i v_i = V'$$

Usaremos ese valor voraz para el problema original.

Heurística:

- Maximizar
  - Ordenar las monedas por la razón $^{p_i}/_{v_i}$ de mayor a menor
  - Acción: $n_i = \frac{vr}{v_i}$, valor real y división real

- Minimizar
  - Ordenar las monedas por la razón p_i/v_i de menor a mayor
  - Acción: $n_i = \frac{vr}{v_i}$, valor real y división real

Solución:

- La solución más adecuada serían dos *Multiset<Integer>* que indicarían el número de unidades de cada moneda entregadas y recibidas
- La solución se calcula a partir de los detalles del camino óptimo.

El código se puede encontrar en el repositorio

Problema de las estaciones de bomberos

Para ilustrar este modelo, consideremos el siguiente problema de localización: Una ciudad está considerando la ubicación de sus estaciones de bomberos. La ciudad se compone de n barrios. Cada barrio es vecino de otros barrios y la relación de vecindad se puede representar mediante un grafo no dirigido cuyos vértices representan los barrios y existe una arista entre dos barrios si son vecinos. Una estación de bomberos se puede colocar en cualquier barrio y es capaz de gestionar los incendios, tanto para ese barrio como para cualquier barrio vecino. El objetivo es minimizar el número de estaciones de bomberos.

Este problema podemos considerarlo como un ejemplo recubrimiento de vértices anterior. Tomando de nuevo las variables binarias $x_j,\ j \in [0, m)$, que tomarán el valor 1 si el barrio v_j es escogido, el problema de las estaciones de bomberos puede ser formulado como:

$$min \sum_{j=0}^{m-1} x_j$$

$$(\sum_{j:N(i)} x_j) \geq 1, \quad i \in [0, m)$$

$$bin\ x_j, \quad j \in [0, m)$$

El problema puede ser resuelto mediante algoritmos genéticos escogiendo un cromosoma binario de tamaño *m.*

Un modelo alternativo puede ser:

$$min \sum_{j=0}^{m-1} x_j$$
$$\bigcup_{i=0|x_i=1}^{m-1} N(i) = \mathrm{U}$$
$$bin\ x_j, \quad j \in [0, m)$$

Siendo $N(i)$ en conjunto de vértices vecinos de i incluido él mismo y $U = \{0, \dots, m-1\}$. A partir de este nuevo modelo se puede resolver mediante algoritmos genéticos escogiendo un cromosoma binario de tamaño m y función de fitness en función del vector de decode d:

$$ft(d) = -\sum_{j=0}^{m-1} w_j d[j] - k\ \ dEqS(S_{i=0|d[i]=1}^{m-1} N(i), S_{i=0}^{m-1} i)$$

Donde $dEqS$ es la distancia a la restricción igualdad entre dos conjuntos:

```
Boolean equals(Set<E> s1, Set<E> s2);
Double deqSet(Set<E> s1, Set<E> s2) {
	Set<E> cp = new HashSet<>(s1);
	cp.removeAll(s2);
	Integer n = cp.size();
	return n*n;
}
```

Para resolverlo por el resto de las técnicas lo generalizamos:

EstacionesVertex:

Propiedades:

- i: Integer
- bc: Set<Integer>, Barrios ya cubiertos
- n: Integer, compartida
- t: Integer, derivada n-i

Interpretación:

Encontrar los barrios desde i hasta el final donde se ubicarán estaciones de bomberos, cuyo número sea mínimo, con el conjunto bc de barrios ya cubiertos.

Igualdad

- Dos problemas son iguales si lo son i,bc

Es válido

- i>=0,i<=n

Factoría:

- inicial(): Crea el problema (0,{})
- goal(v) = p.i ==n

Casos base:

1. p.i ==n
2. pi. == n-1

Solución casos Base

1. Tiene solución
2. Tiene solución si $bc + (m-1) = \{0, \ldots, m-1\}$

Acciones:

$A_i = \{0,1\}$

En el caso base 2 las acciones serían $A_i = \{1\}$, si tiene solución y $A_i = \{\}$ si no la tiene.

Vecino

Caso general: $neighbord(0) = (i+1, bc)$

Caso general: $neighbord(1) = (i+1, bc + N(i))$

Pero de la Arista

Peso: $w = a$

Peso del camino:

Suma de los pesos de las aristas

Solución Voraz: Escoger el barrio actual si no está cubierto y no escogerlo si lo está

Heurística: No escoger ningún barrio desde el actual hasta el final.

Solución: La solución adecuada sería *Set<Integer>* con el conjunto de los barrios elegidos para ubicar estaciones de bomberos

Problema de la asignación

En este problema tenemos una lista de agentes L y una lista de tareas T ambas del mismo tamaño n. El coste de que el agente i realice la tarea j sea $c(i,j)$. Se pretende asignar a cada agente una tarea y sólo una de tal forma que se ejecuten todas las tareas con el coste mínimo.

Asumimos las variables binarias x_{ij} toman valor 1 si el agente i ejecuta la tarea j y cero si no la ejecuta. El problema puede ser modelado de la forma:

$$\min \sum_{i=0,j=0}^{n-1,n-1} x_{ij}\, c(i,j)$$

$$\sum_{j=0}^{n-1} x_{ij} = 1, \quad i \in [0, n-1]$$

$$\sum_{i=0}^{n-1} x_{ij} = 1, \quad j \in [0, n-1]$$

$$bin\ x_{ij}, \quad i \in [0, n-1],\ j \in [0, n-1]$$

Ahora las variables son binarias, toman valores cero y uno y, de nuevo, tenemos un Problema de Programación Lineal Entera. El primer conjunto de restricciones indica que cada agente i tiene que realizar una tarea y

sólo una. El segundo conjunto de restricciones indica que cada la tarea *j* tiene que ser realizada por un agente y sólo uno. Este modelo puedo ser resuelto mediante PLI.

Un segundo modelo para este problema es:

$$min \sum_{i=0}^{n-1} c(i, x_i)$$
$$P_{i=0}^{n-1}(x_i, i)$$
$$int\ x_i, \quad i \in [0, n-1]$$

En este segundo modelo la variable x_i indica la tarea que se le asigna al agente i.

Este problema se puede resolver mediante algoritmos genéticos con un cromosoma de tipo permutación de tamaño *n*. Para resolverlo por el resto de las técnicas lo generalizamos:

AsignacionVertex:

Propiedades:

- i: Integer
- tna: Set<Integer>, tareas no asignadas todavía, derivada
- n: Integer, compartida
- t: Integer, derivada n-i
- c(i,j): Double, Coste de asignar la tarea j al agente i. Compartida

Interpretación:

Encontrar la asignación con el coste mínimo de las tareas no asignadas a los agentes desde i hasta el final.

Igualdad

- Dos problemas son iguales si lo son i,tna

Es válido

- i>=0,i<=n

Factoría:

- inicial(): Crea el problema (0,{0..n-1})
- final(): Crea el problema (n,{})
- goal(v) = p.i ==n

Casos base:

1. p.i ==n
2. pi. == n-1

Solución casos Base

1. Tiene solución
2. Tiene solución si $|tna| = 1$

Acciones:

$A_i = tna$

Vecino

Caso general: $neighbord(a) = (i + 1, tna - a)$

Peso de la arista

Peso: $w = c(i, a)$

Peso del camino:

Suma de los pesos de las aristas

Solución Voraz: Escoger la acción $\underset{a \in A_i}{argmin}\, c(i, a)$

Heurística: $(n - i) * c,\ c = \min_{i:i..n-1, a:A_i} c(i, a)$

Solución: La solución adecuada sería *Map<Integer,Integer>* que recogería la tarea asignada a cada agente.

Problema de tareas y procesadores

El problema se formula de la siguiente manera: Dado una lista de n tareas con duraciones d_i y un conjunto de m procesadores buscar la asignación de tareas a procesadores tal que el tiempo total de ejecución sea mínimo.

Escogiendo las variables binarias x_{ij} que toman valor 1 si la tarea i es asignada al procesador j el problema puede ser formulado como:

$$\min \quad T$$
$$\sum_{i=0}^{n-1} d_i x_{ij} \leq T, \quad j = 0, \dots, m-1$$
$$\sum_{j=0}^{m-1} x_{ij} = 1, \quad i = 0, \dots, n-1$$
$$bin \ x_{ij}, \quad i = 0, \dots, n-1, j = 0, \dots, m-1$$

Con ese modelo se puede resolver mediante PLI.

Un segundo modelo para este problema es:

$$\min \max_{j:0..n-1} \Sigma_{i=0|x_i=j}^{n-1} d_i$$
$$0 \leq x_i < m, \quad i \in [0, n-1]$$
$$int \ x_i, \quad i \in [0, n-1]$$

En este segundo modelo la variable x_i indica el procesador al que se asigna la i.

Este problema se puede resolver mediante algoritmos genéticos con un cromosoma de tipo rango de tamaño n.

Para resolverlo por el resto de las técnicas lo generalizamos:

TareasYProcesadoresVertex:

Propiedades:

- i: Integer, básica
- cargas: List<Double> de tamaño m, cargas de los procesadores, básica

- cm: Double, derivada, carga del procesador más cargado
- nMin: Integer, derivada, procesador menos cargado
- n: Integer, compartida, número de tareas
- m: Integer, compartida, número de procesadores
- t: Integer, derivada n-i, tamaño

Invariante:

- $cm = \max_{j:0..m-1} cargas[j]$
- $nMin = \operatorname{argmin}_{j:0..m-1} cargas[j]$

Interpretación:

Encontrar la asignación a los procesadores de las tareas desde i hasta el final y con el coste mínimo

Igualdad

- Dos problemas son iguales si lo son i, cargas

Es válido

- i>=0, i<=n

Factoría:

- inicial(): Crea el problema (0,[0,0,0…,0])
- final(): Crea el problema (n,[0,0,0…,0])
- goal(v) = p.i ==n

Casos base:

1. p.i ==n

Solución casos Base

1. Tiene solución

Acciones:

$A_i = \{0..m-1\}$

Vecino

$$neighbord(a) = (i + 1, cargas + (a, d_i))$$

Peso de la arista: No podemos asignar un peso a la arista tal que la suma sus pesos sea el peso del camino a minimizar

Peso del camino: Podemos asignar un peso a un camino completo que sean una función del último vértice del camino. En concreto la propiedad *cm* del último vértice

Solución Voraz: Escoger la acción $a = nMin$

Heurística: Escoger la acción *v.cm*

Solución: Un tipo con las propiedades

- Asignación: Map<Integer,List<Integer>>. Asignación de tareas a cada procesador
- *cargaMaxima: Double,* Carga del procesador más cargado

Coloreado de grafos

El *Problema de Coloreado de Grafos* consiste en buscar el mínimo número de colores tal que dando un color a cada vértice sea distinto el color asociado a dos vértices vecinos. Para modelarlo como un Problema de Programación Lineal Entera partimos de las variables binarias $y_k,\ k \in [0, m)$, que tomarán el valor 1 si el color k es usado y n es el número de vértices del grafo. Además, introducimos las variables binarias x_{ik} que serán 1 si el vértice i se le asigna el color k. El modelo del problema es:

$$\min \sum_{k=0}^{m-1} y_k$$

$$\sum_{k=0}^{m-1} x_{ik} = 1, \quad i = 0, \dots, n-1$$

$$x_{ik} \leq y_k, \qquad i = 0, \dots, n-1, k = 0, \dots, m-1$$

$$x_{ik} + x_{jk} \leq 1, \qquad (i,j) \in E, k = 0, \dots, m-1$$

$$bin \ \ x_{ik}, \ \ y_k, \quad i = 0, \dots, n-1, k = 0, \dots, m-1$$

La restricción (1) garantiza que cada vértice tiene color y solo uno. La restricción (2) indica que si el vértice i recibe de color k este color es usado. La restricción (3) que si los vértices i, j son vecinos no pueden tener el mismo color.

La restricción $a \leq b$, asumiendo que las variables toman valores binarios, tiene como soluciones posibles para (a, b) los pares $\{(0,0), (0,1), (1,1)\}$ y por lo tanto si $a = 1$ implica que $b = 1$.

La restricción $a + b \leq 1$, asumiendo que las variables toman valores binarios, tiene como soluciones posibles para (a, b) los pares $\{(0,0), (0,1), (1,0)\}$ y, por lo tanto, una de las dos variables toma valor 1 o ninguna de ellas, pero no las dos a la vez.

Aunque hay otra forma mejor como ahora veremos este problema se podría resolver mediante algoritmos genéticos escogiendo un cromosoma binario de tamaño $n * m + m$. Para ello la primera tarea es ubicar las variables del problema en el vector decode. Una posibilidad es la siguiente: $x_{ik} = d[m * i + k], y_k = d[m * n + k]$.

Un modelo alternativo sería con las variables de tipo entero x_i que indicaría el color del vértice i. Si representamos por $N(i)$ los vértices vecinos de v_i sin incluir él mismo el problema puede ser formulado como:

$$\begin{gathered} \min CD_{i=0}^{n-1}\, x_i \\ x_i \notin S_{k:N(i)} x_k, \quad i = 0, \dots, n-1 \\ 0 \leq x_i < m, \quad i = 0, \dots, n-1 \\ int \quad x_i \quad i = 0, \dots, n-1 \end{gathered}$$

Recordemos que $S_{k:N(i)} x_k$ es el conjunto de colores de los vecinos del vértice v_i. Para resolverlo por el resto de las técnicas lo generalizamos:

ColoreadoVertex:

Propiedades:

- i: Integer, básica
- cav: Map<Integer,Integer>, colores asignados a cada vértice en 0..i-1, básica
- ca: Set<Integer>, colores ya asignados, derivada

- nc: Integer, número de colores ya asignados, derivada
- cv: Set<Integer>, colores asignados a los vecinos de i que ya tienen color, derivada
- m: Integer, número de colores, obtenido previamente mediante una Solución Voraz
- n: Integer, compartida, número de vértices
- t: Integer, derivada n-i, tamaño

Invariante

- $ca = cav.values()$
- $nc = |ca|$
- $cv = S_{k:N(i)} x_k$

Interpretación:

Encontrar la asignación del mínimo número de colores a los vértices desde i hasta el final teniendo en cuenta los colores ya asignados a los vértices previos

Igualdad

- Dos problemas son iguales si lo son i, cav

Es válido

- i>=0, i<=n, $x_i \notin cv$

Factoría:

- inicial(): Crea el problema (0,{}]
- goal(v) = p.i ==n

Casos base:

1. p.i ==n

Solución casos Base

1. Tiene solución

Acciones:

$A_i = \{0..m-1\} - cv$

Vecino

Caso general: $neighbord(a) = (i+1, cav + (i,a))$

Peso de la arista: No asignamos peso a la arista

Peso del camino: La propiedad *nc* del último vértice

Solución Voraz:

- Si $|ca - cv| > 0$ escoger aleatoriamente una de las acciones del conjunto $ca - cv$
- Si $|ca - cv| = 0$ escoger aleatoriamente una de las acciones del conjunto $\{0..n-1\} - cv$

Heurística: el peso del último vértice

Solución: Map<Integer,Integer> que recoja el color asignado a cada vértice

Problema de empaquetado en contenedores (Pack)

El problema se formula de la siguiente manera: Disponemos de un número ilimitado de contenedores tamaño V y una lista de n elementos con tamaños $v_j \leq V$, encontrar el mínimo número de contenedores necesarios para empaquetar todos los elementos.

Escogiendo las variables binarias x_{ij} que toman valor *1* si el elemento i es empaquetado en el contenedor j y las variables y_j que toman el valor *1* si el contenedor i es usado, el problema puede ser formulado como:

$$\min \sum_{j=0}^{m-1} y_j$$

$$\sum_{i=0}^{n-1} v_j x_{ij} \leq V y_j, \quad j = 0, \dots, m-1$$

$$\sum_{j=0}^{m-1} x_{ij} = 1, \quad i = 0, \dots, n-1$$

$$bin\ y_j, x_{ij}, \quad i = 0, \dots, n-1, j = 0, \dots, m-1$$

Un modelo alternativo sería con las variables de tipo entero x_i que indica el contenedor donde colocar el objeto i. Asumimos un número de contenedores m que será menor o igual que n y que podremos estimar previamente mediante una Solución Voraz.

$$\min CD_{i=0}^{n-1}\ x_i$$
$$0 \leq \Sigma_{i=0|x_i=k}^{n-1} v_i < V, \quad k = 0, \dots, m-1$$
$$0 \leq\ x_i < n, \quad i = 0, \dots, n-1$$
$$int\ \ x_i \quad i = 0, \dots, n-1$$

Con este modelo podemos resolver el problema mediante algoritmos genéticos escogiendo un cromosoma de rango de tamaño n, donde los valores estarán en el rango *[0,n)* y la función de fitness de la forma:

ft(d)= $-CD_{i=0}^{n-1} d[i] - k\ \sum_{k=0}^{n-1} dle(-V + \Sigma_{i=0|d[i]=k}^{n-1} v_i)$

Para resolverlo por el resto de las técnicas lo generalizamos:

PackVertex:

Propiedades:

- i: Integer
- *carga: Map<Integer,Integer>,* ocupación de los contenedores. Básica. Las claves serán los contenedores parcial o totalmente ocupados. Si un contenedor no pertenece a las claves está vacío

- *nc: Integer,* número de contenedores parcial o totalmente ocupados, derivada, igual al número de claves de oc.
- *cMax: Integer,* uno de los contenedores más ocupado
- *cMin: Integer*, uno de los contenedores menos ocupado.
- *n: Integer,* compartida, número de objetos
- *t: Integer,* derivada n-i, tamaño

Interpretación:

Encontrar la asignación de los objetos desde i hasta el final en los contenedores asumiendo que ya están parcialmente ocupados

Igualdad

- Dos problemas son iguales si lo son i, as

Es válido

- i>=0, i<=n, oc[i] <=V

Factoría:

- inicial(): Crea el problema (0,{}]
- goal(v) = p.i ==n

Casos base:

1. p.i ==n

Solución casos Base

1. Tiene solución porque asumimos que el número de contenedores es suficiente

Acciones:

$A_i = a: \{0..nc+1\} | oc[a] + v(i) \leq V$

Ordenamos las acciones con el contenedor más cargado el primero

Vecino

Caso general: $neighbord(a) = (i + 1, as + (i, a))$

Peso de la arista: La diferencia entre el número de contenedores del destino y de la fuente.

Peso del camino: El número de contenedores del último vértice (alternativamente la suma de los pesos de las aristas)

Solución Voraz: Escoger entre las alternativas disponibles aquella en la que $carga[a]$ sea máximo

Heurística: La suma del número de contenedores del último vértice más la diferencia entre el número de objetos y el índice actual (alternativamente la diferencia entre el número de objetos y el índice actual si consideramos el camino de tipo Sum).

Solución: Map<Integer,List<Integer>> que los objetos asignados a cada contenedor.

Problema de las reinas

El problema consiste en colocar n reinas en un tablero de ajedrez $n \times n$ de tal manera 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 $0..n-1$.

Escogiendo las variables binarias x_{ij} que toman valor 1 si ubicamos una reina en la casilla i, j el problema puede ser formulado como:

$$\sum_{i=0}^{n-1} x_{ij} = 1, \qquad j = 0, \dots, n-1$$
$$\sum_{j=0}^{n-1} x_{ij} = 1, \qquad i = 0, \dots, n-1$$
$$\sum_{i,j|j-i=dp} x_{ij} \leq 1, \qquad dp = dp1, \dots, dp2$$
$$\sum_{i,j|j+i=ds} x_{ij} \leq 1, \qquad ds = ds1, \dots, ds2$$
$$bin\ x_{ij}, \quad i,j = 0, \dots, n-1$$

El índice *dp* representa los distintos valores de la diagonal principal y *ds* los de la diagonal segundaria: las ecuaciones respectivas son $y = x + dp, y = -x + ds$. Los valores mínimos y máximos de dp, ds son por lo tanto: $dp1 = -n, dp2 = n, ds1 = 0, ds2 = 2n$. Observando que en las diagonales extremas sólo hay una casilla los rangos de valores anteriores pueden ser reducidos a $dp1 = -n + 1, dp2 = n - 1, ds1 = 1, ds2 = 2n - 1$. En este problema no se trata de encontrar un mínimo. Se trata de encontrar una solución y puede haber muchas.

A partir de este modelo el problema se podría resolver mediante algoritmos genéticos, aunque hay otros modelos mejores para este propósito. Se deja como ejercicio la elección del cromosoma y la función de fitness.

El problema puede ser modelado alternativamente usando las variables enteras $x_i, \quad i: [0, n-1]$, que indica que una reina se colocará en la casilla (i, x_i) y las restricciones:

$$P_{i=0}^{n-1}(x_i, i)$$
$$AD_{i=0}^{n-1}(x_i - i)$$
$$AD_{i=0}^{n-1}(x_i + i)$$
$$int\ x_i, \quad i: [0, n-1]$$

La razón es que las filas ocupadas serán los valores de las x_i, las diagonales principales ocupadas serán los valores $x_i - i$ y las diagonales secundarias ocupadas serán los valores $x_i + i$.

Con este nuevo modelo el problema se puede resolver mediante algoritmos genéticos eligiendo un cromosoma de permutación de tamaño *n* y secuencia normal [0,1,…,n-1]. La función de fitness en función del vector de decode es:

$$ft(d) = -dAd_{i=0}^{n-1}(d[i] - i) - dAd_{i=0}^{n-1}(d[i] + i)$$

Como podemos ver la primera restricción y la restricción de rango ya están tenidas en cuenta en los valores generados por el cromosoma. Por lo tanto, solo la restricción segunda y tercera se incluye en la función de fitness mediante distancias a estas restricciones. Ya vimos anteriormente la restricción allDifferent y la distancia a la misma.

Para resolverlo por el resto de las técnicas lo generalizamos:

ReinasVertex:

Propiedades:

- i: Integer, básica
- fo: List<Integer>, filas ocupadas, básica
- dp: Set<Integer>, diagonales principales ocupadas, derivada
- ds: Set<Integer>, diagonales secundarias ocupadas, derivada
- vl: Set>Integer, valores libres de las filas para colocar en columna i, derivada
- n: Integer, compartida, número columnas igual al de filas
- t: Integer, derivada n-i, tamaño

Invariante:

- $dp = S_{k:0..i-1}(fo[i] - i)$
- $ds = S_{k:0..i-1}(fo[i] + i)$
- $vl = S_{k:0..i-1|p(k)}k, \quad p(k) \equiv k \notin fo \wedge (k - i) \notin dp \wedge (k + i) \notin ds$

Interpretación:

Encontrar la asignación reinas desde i hasta el final asumiendo que ya están colocadas las reinas de 0..i-1

Igualdad

- Dos problemas son iguales si lo son i, fo

Es válido

- i>=0, i<=n, |fo|=|dp|=|ds|

Factoría:

- inicial(): Crea el problema (0,{}]
- goal(v) = p.i ==n

Casos base:

1. p.i ==n

Solución casos Base

1. Tiene solución

Acciones:

$A_i = a: \{0..n-1\} | a \notin fo, a - i \notin dp, a + i \notin ds$

Vecino

Caso general: $neighbord(a) = (i + 1, fo + a)$

Peso de la arista: Peso de la arista 1

Peso del camino: Suma de los pesos de las aristas

Solución Voraz: Escoger aleatoriamente entre las alternativas disponibles según la metaheurística.

Heurística: 0.

Solución: Map<Integer,Integer> que recoja la fila donde se colocará la reina en cada columna.

Problema del sudoku

El problema consiste en rellenar con los enteros $[1..n]$ las casillas de un tablero $n \times n$ de tal manera que cada fila, cada columna y cada subtabla tenga todos los enteros $[1..n]$ y una sola vez cada uno de ellos. Asumiendo subtablas cuadradas de lado k, $(n = k^2)$, a cada casilla le podemos asignar las coordenadas i, j que toman valores en $0..n-1$, siendo la casilla (0,0) la inferior izquierda. A cada subtabla le asociamos el índice t que toma valores en $0..n-1$ y se cumple $t = \left(\frac{j}{k}\right) k + i/k$ siendo / la división entera. Por ejemplo, para $(i, j) = (2,4)$, $n = 9, k = 3$ tenemos $t = \left(\frac{4}{3}\right) 3 + \frac{2}{3} = 3$. Concretemos el problema para $n = 9$. Cada casilla tiene también asociado un entero p: $0..n^2 - 1$ que indica su posición en una lista de casillas tomadas por filas de abajo arriba. Es decir $p = i * n + j$ y, a su vez, dado p, $i = \frac{p}{n}, j = p\%n$. Por último cada casilla tiene la propiedad d que indica si su valor está definido al principio y el valor v fijado para esa casilla.

Sea $\varphi(i, j, t) = \{(i, j) | t = \left(\frac{j}{k}\right) k + i/k\}$ y escogiendo las variables binarias x_{ijv} que toman valor 1 si la casilla i, j toma el valor v el problema puede ser formulado como:

$$\sum_{i=0}^{n-1} x_{ijv} = 1, \qquad j = 0, \dots, n-1, v = 1, \dots, n$$

$$\sum_{j=0}^{n-1} x_{ijv} = 1, \qquad i = 0, \dots, n-1, v = 1, \dots, n$$

$$\sum_{(i,j):\varphi(i,j,t)} x_{ijv} = 1, \qquad t = 0, \dots, n-1, v = 1, \dots, n$$

$$\sum_{v=1}^{n} x_{ijv} = 1, \qquad i = 0, \dots, n-1, j = 0, \dots, n-1$$

$$bin\ x_{ijv}, \qquad i, j = 0, \dots, n-1, v = 1, \dots, n$$

La primera restricción indica que un valor v, sólo puede aparecer una vez en la fila j. La segunda restricción que un valor v sólo puede aparece una vez en la columna i. La tercera restricción indica que un valor v sólo puede aparece una vez en la subtabla t. La cuarta que en cada casilla sólo puede haber un valor. Los problemas de sudokus suelen venir enunciados con un conjunto de casillas con un valor asignado. Esto añadirá más restricciones a las propuestas previamente. Así si la casilla (2,3) tiene asignado en el enunciado del problema un 7 implicará que $x_{2,3,7} = 1, x_{2,3,v} = 0, v \neq 7$.

En este problema no se trata de encontrar un mínimo. Se trata de encontrar una solución y puede haber muchas.

Un modelo alternativo más compacto es:

$$\begin{aligned} P_{i=0}^{n-1}(x_{ij}, i+1), &\qquad j \in [0, n-1] \\ AD_{j=0}^{n-1} x_{ij}, &\qquad i \in [0, n-1] \\ AD_{(i,j)=0|\varphi(i,j,t)}^{n-1} x_{ij}, &\qquad t \in [0, n-1] \\ int\ \ x_{ij}, &\quad i,j \in [0, n-1] \end{aligned}$$

Ahora las variables x_{ij} son valores enteros en el rango $[1, n]$. La primera restricción indica que los valores en cada fila deben ser una permutación del conjunto *{1,2,…,n}*. La segunda que los valores en cada columna deben ser una permutación del mismo conjunto y la tercera que los valores en cada subtabla también.

El problema se puede abordar mediante algoritmos genéticos a partir de este modelo, pero esta técnica no aporta buenos resultados. En la literatura siguen apareciendo propuestas para resolver este problema mediante algoritmos genéticos. Pero este problema se resuelve bien mediante Programación Lineal Entera con el modelo anterior o mediante backtracking.

Para resolverlo backtracking lo generalizamos para definir un grafo de problemas:

Casilla:

- i: Integer, número de la columna
- j: Integer, número de la fila

- p: Integer, posición de la casilla en una secuencia por filas
- t: Integer, número de la subtabla
- v: Integer, puede ser null si el valor no está definido todavía
- d: Boolean, si está definida al principio

SudokuVertex:

Propiedades:

- lc: List<Casilla>, compartida
- i: Integer
- *indices: List<Integer>,* indices a las casillas libres en lc.
- *vl(k): Set<Integer>, k:i..n,* valores libres en la casilla de índice indices[k], derivada
- *c(k): Casilla, k:0..n,* casilla cuyo índice es indices[k], derivada
- *er: Integer*, número de errores, derivada
- n: Integer, compartida, número de casillas
- t: Integer, derivada n-i, tamaño

Invariante

La lista de indices se mantiene ordenada, en primer los indices cuyas casillas están definidas, luego de menor a mayor según la propiedad */vl(indices)/.*

Las propiedades *vl(k)* y *er* se calculan a partir de las casillas ya asignadas

Interpretación:

Encontrar la asignación de valores a casillas cuyos índices están desde indices[i] hasta indices[n-1], asumiendo que ya están colocados los valores para las casillas cuyos índices están entre indices[0] e indices[i-1].

Igualdad

- Dos problemas son iguales si lo son i, lc

Es válido

- i>=0, i<=n, |vl(k)|>0,k=i..n-1

Factoría:

- inicial(): Crea el problema (0,cd]), cd es la lista de casillas definidas
- goal(v) = p.i ==n

Casos finales con solución (goalHasSolution)

- v.er = 0

Casos base:

1. p.i ==n

Solución casos Base

1. Tiene solución

Acciones:

$A_i = a: vl(i)$

Vecino

Caso general: $neighbord(a) = (i + 1), c(i).v = a$

Peso de la arista: No se puede asignar un peso a la arista

Peso del camino: El número de errores del último vértice

Metaheurística: En el problema de las reinas se asignan las columnas de forma consecutiva. Aquí introducimos el concepto de metaheurística: consiste en rellenar primero las casillas más restringidas: las que tienen menos valores libres. Es decir, menor valor de */vl(k)/.* Esto implica cada vez que se crea un vértice ordenar los índices desde *i* hasta *n-1* según el valor de */vl(k)/* de menor a mayor.

Solución Voraz: Escoger aleatoriamente entre las alternativas disponibles.

Heurística: 0

Solución: Map<Casilla,Integer> que recoja el valor asignado a cada casilla.

Transformación de secuencias

Dadas dos secuencias s1 y s2 encontrar el mínimo número de transformaciones, que aplicadas secuencialmente, transforman la primera en la segunda. Por ejemplo transformar "*cbrrrarreterb*" en "*carretera*".

Las transformaciones disponibles son:

- *(A):* Añadir al final de *s1* el carácter *i* de *s2*
- *(E):* Eliminar el carácter en posición *i* de *s1*
- *(C):* Cambiar el carácter que aparece en posición *i* de *s1* por el que aparece en la misma posición de *s2*

Los valores de las acciones son los del tipo enumerado $As = \{A, E, C, M\}$. Se ha añadido la acción *M* (aumentar en uno el valor de i del estado) para simplificar el modelado el problema. Los valores estado, el tipo *Vs*, serán pares *(i,s)* entero-string. Un estado, vértice será válido si *i* es un indice válido en las cadenas *s* y *s2*.

$$isValid(v) \equiv 0 \leq i, i \leq |s|, i \leq s2$$

Adicionalmente tendremos la variable comapartida *s2*. El nuevo estado tras aplicar una acción válida:

$$neighbord(i, s, a) = \begin{cases} (i+1, s + s2[i]), & a = A \\ (i+1, s + (i, s2[i])), & a = C \\ (i, s - i), & a = E \\ (i+1, s), & a = M \end{cases}$$

A partir de lo anterior podemos ver que desde un estado v1 a otro estado vecino v2 hay una única arista que denotaremos por *a(v1,v2).*

Donde por $s + (i, s2[i])$ indicamos que se cambie el carácter *i* de *s* por *s[i]* y por $s - i$ eliminar el carácter en la posición *i* de *s*. Las acciones aplicables en el estado (i,s) son:

$$As(i, s) = \begin{cases} \{A\}, & n - i = 0, & n2 - i > 0 \\ \{E\}, & n - i > 0, & n2 - i = 0 \\ \{C, E\}, & n - i > 0, & n2 - i > 0, s[i] \neq s2[i] \\ \{M\}, & n - i > 0, & n2 - i > 0, s[i] = s2[i] \end{cases}$$

Donde *n2* es la longitud de la cadena *s2* y *n* la longitud de *s*. Para acabar de definir el modelo tenemos que indicar el peso de cada acción, cual es el conjunto de acciones aplicables en un estado dado y cuál es el nuevo estado tras aplicar una acción posible.

$$w(a) = \begin{cases} 1, & a = A \\ 1, & a = C \\ 1, & a = E \\ 0, & a = M \end{cases}$$

Dos vértices son vecinos si hay una arista que los conecte.

$$isNeighbord(v1, v2) \equiv \wedge_{a:As(v1)}\ neighbor(v1, a) = v2$$

La función $isNeighbord(v1, v2)$ es equivalente al método $v1.isNeighbord(v2)$. Igualmente, $neighbor(v1, a)$ es equivalente a $v1.neighbor(a)$.

A partir del tipo *Vs* y los predicados *isValid* y *isNeighbor* podemos definir por comprensión el grafo *g* de tipo *Graph<Vj,DefaultEdge>.*

$$sv = \{v{:}\, Vs \mid v.isValid()\}$$
$$se = \{v1{:}\, sv, v2{:}\, sv \mid v1.isNeighbord(v2)\}$$
$$g = (sv, se)$$

A partir del grafo *g* nuestro problema es un problema de camino mínimo en ese grafo. Con esos tipos ya podemos diseñar un modelo:

$$min \sum_{i=0}^{r-2} w(a(x_i, x_{i+1})$$
$$x_0 = (0, s1)$$
$$x_{r-1} = (|s2|, s2)$$
$$OP_{i=0|g}^{r-1} x_i$$
$$int\ r$$
$$Vs\ x_i, i \in [0, r-1]$$

El modelo anterior tiene prácticamente todos los detalles que necesitamos para definir un grafo de problemas. Los valores del estado definen los vértices y las acciones las aristas.

SecuenciasVertex:

Propiedades:

- k, entero, básica
- s, String, básica
- n, Integer, tamaño de s, derivada
- s1, String, compartida
- s2, String, compartida
- n1, Integer, tamaño de s1, derivadas
- n2, Integer, tamaño de s2, derivada
- nd: Integer, número de caracteres diferentes en s[i:] y s2[i:] asumiendo que si no existen son considerados como diferentes.

Invariante

$$s[0:k] = s2[0:k]$$

Interpretación:

Encontrar los cambios en los caracteres desde k hasta el final para conseguir que s sea igual a s2.

Igualdad

- Dos problemas son iguales si lo son k, s

Es válido

- Siempre que cumpla el invariante

Factoría:

- inicial(): Crea el problema (0,s1)
- goal(v) = v =(n2,s2)

Casos base:

1. k ==n

Solución casos Base

1. Tiene solución

Acciones:

$$\boldsymbol{A_i} = \begin{cases} \{A\}, & n-k=0, & n2-k>0 \\ \{E\}, & n-k>0, & n2-k=0 \\ \{C,E\}, & n-k>0, & n2-k>0, s[k] \neq s2[k] \\ \{M\}, & n-k>0, & n2-k>0, s[k] = s2[k] \end{cases}$$

Hemos añadido una acción adicional, *M*, con peso cero para resolver el problema. Esta acción no estará en la lista solución.

Vecino

$$nb(k,s,a) = \begin{cases} (k+1, s+s2[k]), & a = A \\ (k+1, s+(k, s2[k])), & a = C \\ (k, s-k), & a = E \\ (k+1, s), & a = M \end{cases}$$

Peso de la arista:

Depende de la acción tomada

$$w(a) = \begin{cases} 1, & a = A \\ 1, & a = C \\ 1, & a = E \\ 0, & a = M \end{cases}$$

Peso del camino: Suma de los pesos de las aristas

Solución Voraz: Escoger aleatoriamente una entre las alternativas disponibles.

Heurística: Propiedad *nd* del vértice: Número de caracteres distintos en las *subcadenas s[i:] y s2[i:]*

Solución: List<(As,Integer)>

Subsecuencia común más larga

Dada una secuencia $X = \{x_0, x_1, \dots, x_{m-1}\}$ decimos que $Z = \{z_0, z_1, \dots, z_{k-1}\}$ es una subsecuencia de X (siendo $k \leq m$) si existe una secuencia creciente $\{i_0, i_1, \dots, i_{k-1}\}$ de índices de X tales que para todo $j = 0, 2, ..., k-1$ tenemos $x_{i_j} = z_j$. Dadas dos secuencias X e Y, decimos que Z es una subsecuencia común de X e Y si es subsecuencia de X y subsecuencia de Y. Es decir, buscamos alinear caracteres iguales entre las cadenas X, Y, siempre y cuando aparezcan en el mismo orden.

Deseamos determinar la subsecuencia de longitud máxima común a dos secuencias dadas.

Ejemplo: Si $X = \{a, \mathbf{b}, \mathbf{c}, \mathbf{b}, d, \mathbf{a}, b\}$, $Y = \{\mathbf{b}, d, \mathbf{c}, a, \mathbf{b}, \mathbf{a}\}$ entonces $Z = \{b, c, b, a\}$ es una subsecuencia de ambas y, además, la más larga.

El problema que nos planteamos es: dadas dos secuencias $X = \{x_0, x_1, \dots, x_{n-1}\}$, $Y = \{y_0, y_1, \dots, y_{m-1}\}$ encontrar otra secuencia $Z = \{z_0, z_1, \dots, z_{k-1}\}$ que sea la subsecuencia común a ambas más larga posible.

Escogemos las variables binarias v_{ij} que indican que el carácter i de la cadena X se alinea al carácter j de la cadena Y para aparecer en la subcadena resultante Z. Asumimos la función $coincide(i, j)$ que devuelve us 1 si los respectivos caracteres de X e Y coinciden y 0 en otro caso. Un modelo del problema con estas variables es:

$$
\begin{aligned}
& max \sum_{i:0..n-1, j:0..m-1} v_{ij} \\
v_{ij} \leq coincide(i,j), \quad & i \in (0, n-1), j \in (0, n-1) \\
\sum_{i:0..n-1} v_{ij}, \quad & j \in (0, m-1) \\
\sum_{j:0..m-1} v_{ij}, \quad & i \in (0, n-1) \\
v_{i1j1} + v_{i2j2} \leq 1, \quad & i1, i2, j1, j2 \in (0, n-1), i2 > i1, j2 < j1 \\
bin\ v_{ij}, \quad & i \in (0, n-1), j \in (0, m-1)
\end{aligned}
$$

Como vemos se maximiza el número de caracteres que se alinean de ambas cadenas. La primera restricción indica que si los caracteres i, j se alinean deben coincidir. La segunda restricción indica que cada carácter de Y se alinea con uno de X como máximo. La tercera que cada carácter de

X se alinea con uno de Y como máximo. La última restricción indica que si alinean los pares $(i1, j1)$ y $(i2, j2)$ tienen que estar en el mismo orden. Es decir si no están en el mismo orden, $i2 > i1, j2 < j1$, solo uno de los pares puede estar alineado.

El modelo anterior es adecuado para resolver el problema mediante *PLI.*

Veamos como modelar el problema mediante un grafo. Sea *Vs* el tipo de los vértices cuyas propiedades son:

Propiedades de Vs

Integer m, El tamaño de la cadena X
Integer n, El tamaño de la cadena Y
Integer i, Un índice en la cadena X
Integer j, Un índice en la cadena Y
Boolean isValid() ≡ i>=0, i<n, j>=0, j<m
Vs neighbor(As a), el vértice vecino si escogemos a
Set<As> actions(), Las acciones posibles

Tomamos el par *(i,j)* como propiedades del vértice del grafo. Los valores de las acciones son los del tipo enumerado $As = \{A, B, C\}$.

El resultado del método *actions()* se determina con las reglas: si $x_i = y_j$ sólo tenemos una alternativa posible que llamaremos C. Si $x_i \neq y_j$ hay dos alternativas disponibles que representaremos por A y B según incrementemos el índice de una u otra lista.

$$actions() = \begin{cases} C, & x_i = y_j \\ \{A, B\}, & x_i \neq y_j \end{cases}$$

El método *neighbor(a)*:

$$neighbord(a) = \begin{cases} (i+1, j+i), & a = C \\ (i+1, j), & a = A \\ (i, j+1), & a = B \end{cases}$$

Un modelo del problema podría ser:

$$\min r$$
$$x_0 = (0,0)$$
$$x_{r-1} = (i,m)\ \ o\ \ (n,j)$$
$$OP_{i=0|g}^{r-1} x_i$$
$$int\ r$$
$$Vs\ x_i, i \in [0, r-1]$$

El modelo describe un camino de peso máximo en un grafo cuyos vértices tienen las propiedades (i, j) y las acciones una del tipo *As* anterior.

Para acabar de especificar el grafo virtual extendido hay que fijar el vértice inicial, el final, el peso de las aristas y la heurística.

Las ideas anteriores ordenadas son:

ScmlVertex:

Propiedades:

- i: Integer, básica
- j: Integer, básica
- X: List<E>, compartida
- Y: List<E>, compartida
- n: Integer, compartida, tamaño de X
- m: Integer, compartida, tamaño de Y
- t: Integer, derivada, t = n-i+m-j

Interpretación:

*E*ncontrar la subsecuencia común máxima entre *X[i:n]* y *Y[j:n]*

Igualdad

- Dos problemas son iguales si lo son i, j

Es válido

- i>=0, i<m, j>=0, j<n,

Factoría:

- inicial(): Crea el problema (0,0)

Problema Final:

1. (i,m) o (n,j)

Solución caso Final

1. Tiene [] como solución

Acciones:

$$\boldsymbol{A_i} = \begin{cases} \{A, B\}, & X[i] \neq Y[i], i < n, j < m \\ \{C\}, & X[i] = Y[i], i < n, j < m \end{cases}$$

Vecino

$$neighbord(a) = \begin{cases} (i+1, j), & a = A \\ (i, j+1), & a = B \\ (i+1, j+1), & a = C \end{cases}$$

Peso de la arista: Depende de la acción tomada $w(a) = \begin{cases} 0, & a = A \\ 0, & a = B \\ 1, & a = C \end{cases}$

Peso del camino: Suma de lo pesos de las aristas

Solución Voraz: Escoger aleatoriamente una entre las alternativas disponibles.

Heurística*: min(n-i,m-j)*

Solución: *List<E>*

Problema jarras

Se dispone de dos jarras inicialmente vacías, *J1* y *J2*, de capacidades en litros *C1, C2*. Se desea que las jarras contengan una cierta cantidad de litros de agua: *CFJ1* y *CFJ2*.

Como las jarras no poseen marcas de medida, la única manera de conseguirlo es haciendo trasvases de agua entre las mismas. Las operaciones posibles son:

Operación	Descripción	Nombre
vaciar J1	Vacía completamente el contenido de J1 al suelo.	VJ1
volcar J1 en J2	Vuelca el contenido de la jarra J1 en la J2. Si ésta se llena, el agua restante de la J1 cae al suelo y desaparece. J1 siempre queda vacía.	VJ1J2
echar J1 en J2	Echa el contenido de la jarra J1 en la J2. Si ésta se llena, el agua restante de la J1 se conserva en dicha jarra (no cae al suelo).	EJ1J2
llenar J1	Llena completamente la jarra J1 desde el grifo.	LLJ1
llenar J2	Llena completamente la jarra J2 desde el grifo.	LLJ2
vaciar J2	Vacía completamente el contenido de J2 al suelo.	VJ2
volcar J2 en J1	Vuelca el contenido de la jarra J2 en la J1. Si ésta se llena, el agua restante de la J2 cae al suelo y desaparece. J2 siempre queda vacía.	VJ2J1
echar J2 en J1	Echa el contenido de la jarra J2 en la J1. Si ésta se llena, el agua restante de la J2 se conserva en dicha jarra (no cae al suelo).	EJ2J1

Para modelar el problema diseñamos un grafo virtual el tipo de cuyos vértices es una tupla de enteros $Vj = (c1, c2)$, es decir la cantidad de agua en las jarras 1 y 2 respectivamente. Un vértice es válido si las cantidades c1, c2 no son negativas:

$$v.isValid() \equiv v.c1 \geq 0, v.c2 \geq 0$$

Los valores de las acciones son los del tipo enumerado *AccionJarras* que representaremos por Aj. Siendo Aj el tipo enumerado:

Aj = [VJ1, VJ1J2, EJ1J2, LLJ1, LLJ2, VJ2, VJ2J1, EJ2J1]

La tabla anterior de arriba nos define el método v1.*neighbor(a)* que nos da el vecino de *v* si tomamos la acción a. Con esa función podemos definir el conjunto de acciones válidas en el vértice *v* como:

$$Aj(v) = \{a: Aj \mid v.neighbor(a).isValid()\}$$

Cada acción válida del conjunto *Aj(v)* define la arista: *e(v,a)* que parte de *v* tomando la acción válida *a*.

$$e(v, a) = (v, v.neighbord(a))$$

Por otra parte, dos vértices son vecinos si hay una arista que los conecte.

$$isNeighbord(v1, v2) \equiv \exists_{a:Aj(v1)}\, neighbor(v1, a) = v2$$

La función $isNeighbord(v1, v2)$ es equivalente al método $v1.isNeighbord(v2)$. Igualmente, $neighbor(v1, a)$ es equvalente a $v1.neighbor(a)$.

A partir del tipo *Vp* y los predicados *isValid* y *isNeighbor* podemos definir por comprensión el grafo *g* de tipo *Graph<Vj,DefaultEdge>.*

$$g = (sv, se)$$
$$sv = \{v: Vj \mid v.isValid()\}$$
$$se = \{v1: sv, v2: sv \mid v1.isNeighbord(v2)\}$$

A partir del grafo g nuestro problema es un problema de camino mínimo en ese grafo. Con esos tipos ya podemos diseñar un modelo:

$$min\, r$$
$$x_0 = v_0$$
$$x_{r-1} = v_1$$
$$OP_{i=0|g}^{r-1} x_i$$
$$int\, r$$
$$Vj\ x_i, i \in [0, r-1]$$

Generalizamos el problema

JarrasVertex:

Propiedades:

- c1, entero, básica, cantidad en la jarra 1
- c2, entero, básica, cantidad en la jarra 2
- cP1, entero, compartida, capacidad de la jarra 1
- cP2, entero, compartida, capacidad de la jarra 2
- cI1, entero, compartida, cantidad inicial en la jarra 1
- cI2, entero, compartida, cantidad inicial en la jarra 2
- cF1, entero, compartida, cantidad final en la jarra 1
- cI1, entero, compartida, cantidad final en la jarra 2

Invariante

$$0 \leq c1 \leq cP1 \;\&\&\; 0 \leq c2 \leq cP2$$

Interpretación:

Encontrar los cambios en los caracteres desde i hasta el final para conseguir que s sea iguala s2.

Igualdad

- Dos problemas son iguales si lo son c1, c2

Es válido

- Siempre que cumpla el invariante

Factoría:

- inicial(): Crea el problema (cI1,cI2)
- final(): Crea el problema (cF1,cF2)

Casos base:

1. Si es el problema final

Solución casos Base

1. Tiene solución

Acciones:

$$A_i = \begin{cases} \{VJ1\}, & c1 > 0, \\ \{VJ2\}, & c2 > 0, \\ \{VJ1J2, EJ1J2\}, & c1 > 0 \wedge\ c2 < cP2 \\ \{VJ2J1, EJ2J1\}, & c1 < cP1 \wedge\ c2 > 0 \\ \{LLJ1\}, & c1 < cP1, \\ \{LLJ2\}, & c2 < cP2 \end{cases}$$

Vecino

Caso general:

$$nb(c1, c2, a) \begin{cases} (0, c2), & a = VJ1 \\ (c1, 0), & a = VJ2 \\ (0, min(c1 + c2, cP2)), & a = VJ1J2 \\ (min(c1 + c2, cP1), 0), & a = VJ2J1 \\ (max(0, c1 + c2 - cP2), min(c1 + c2, cP2)), & a = EJ1J2 \\ (min(c1 + c2, cP1), max(0, c1 + c2 - cP1)), & a = EJ2J1 \\ (cP1, c2), & a = LLJ1 \\ (c1, cP2), & a = LLJ2 \end{cases}$$

Peso de la arista: Peso 1.

Peso del camino: Suma de los pesos de las aristas

Solución Voraz: Escoger aleatoriamente una entre las alternativas disponibles.

Heurística: 0

Solución: List<AccionJarras>

Detalles de implementación.

Cuando las acciones disponibles en un vértice, los vértices vecinos o ambos están especificados por una secuencia compleja de *if-then-else* es más conveniente diseñar una clase que represente las acciones específicas del problema. Esta clase debe extender la interface:

```
interface Action<V> {
	public V neighbor(V v);
	public boolean isApplicable(V v);
	public Double weight(V v);
}
```

A partir de aquí se trata de crear un objeto por cada acción y tener disponible una lista de objetos con todas las acciones posibles.

Robots

Se tienen n=4 tipos de materiales (A, B, C y D). Para cada uno, existe un robot que lo puede generar, pero fabricar ese robot tiene un coste en materiales. Cuando se fabrica un robot, éste produce 1 unidad de su material por minuto indefinidamente. Se empieza con 1 robot de A y 0 unidades de todos los materiales.

Al principio de cada minuto (estado) se puede decidir si se construye un robot de algún tipo (siempre que se tengan los materiales necesarios) o no. El robot construido empieza a producir en el minuto siguiente, y en caso de que se construya, los materiales necesarios para su construcción se restan de los existentes en el estado. También se puede decidir no construir nada en ese minuto.

A continuación, se actualizan los materiales añadiendo lo que han fabricado los robots existentes. Este orden es importante, ya que la

decisión de construir o no un robot en un estado se toma antes de que se actualicen los materiales.

El objetivo es saber cual es la cantidad máxima de material de tipo D que se pueden fabricar tras *T*= 24 minutos.

Sea $\mathrm{mt}(i,j), i,j \in [0,n)$ la cantidad de material *j* necesaria para construir un robots de tipo *i.* Usaremos la notación mt(i) para indicar el material necesario para construir el robots *i.*

Ej: *mt = [[4,0,0,0],[2,0,0,0],[3,14,0,0],[3,0,7,0]]*

Para modelar el problema escogemos un estado definido por el tipo *Vr* cuyas variables básicas son: *List<Integer> r, List<Integer> x, Integer t*. Las listas *r* y *x* tienen el mismo tamaño *n*, indican el número de robots y la cantidad de material de cada tipo respectivamente. Un vértice válido es aquel en el cual todas las casillas de *r* y las de x son positivas o cero. La variable t, que indica el tiempo debe ser menor o igual a T y mayor o igual a cero. Estas propiedades definen el preficado *isValid(v).*

Las acciones disponibles son el conjunto $Ar = \{-1,0,..n-1\}$. La acción a significa crear un robots de tipo a. Para que eso sea posible debe existir material suficiente para poderlo crear y si es -1 no se crea ningún robots. Podemos asumir, por tanto, que *mt(-1) = [0,0,0,0].*

El conjunto de acciones válidas en el vértice *v* será entonces:

$$Ar(v) = \{-1\} + \{a: Ar \mid \forall_{i=0}^{n-1} m(a,j) \leq v.x()[j]\}$$

A cada acción válida le corresponde un vecino $neighbord(v,a)$:

$$neighbord(v,a) = \begin{cases} (v.r, v.x + nm(v.r), t+1), & a = -1 \\ (v.r + nr(a), x + nm(v.r) - mc(a), t+1), & a \geq 0 \end{cases}$$

Donde $nm(v.r) = v.r$ son las unidades producidas por los robots existentes a razón de una unidad por año, $mc(a) = mt(a)$ es la cantidad de material consumido al crear un robot de tipo a y por $nr(a)$ un vector con un 1 en la posición a.

Dos vértices son vecinos si hay una arista que los conecte.

$$isNeighbord(v1, v2) \equiv \exists_{a:Ar(v1)}\, neighbor(v1, a) = v2$$

A partir del tipo *Vr* y los predicados *isValid* y *isNeighbor* podemos definir por comprensión el grafo *g* de tipo *Graph<Vr,DefaultEdge>.*

$$g = (sv, se)$$
$$sv = \{v{:}\, Vj \mid v.isValid()\}$$
$$se = \{v1{:}\, sv, v2{:}\, sv \mid v1.isNeighbord(v2)\}$$

A partir del grafo *g* nuestro problema es un problema de camino mínimo en ese grafo. Con esos tipos ya podemos diseñar un modelo:

$$max\ x_i\ [3]$$
$$x_0 = ([1,0,0,0],[0,0,0,0])$$
$$x_T.t = T$$
$$OP_{i=0|g}^{T} x_i$$
$$Vr\ x_i, i \in [0, T]$$

A partir de ese modelo podemos resolver este problema medainte BT, A* o DPR. Una heuristica podría ser $hu(v) = 3.* (T - v.t())$.

Estantería

Se tiene una colección de libros que se quieren colocar en una estantería compuesta por distintos estantes. Cada libro tiene una altura y una anchura determinada, mientras que todos los estantes tienen la misma anchura, pero distinta altura. Determinar una distribución de libros en los estantes de forma que se maximice el número de libros colocados.

Datos

Libro:

- h_i: altura del libro i
- w_i: anchura del libro i
- n: número de libros

Estante:

- g_j: altura del estante j
- A: anchura los estantes
- m: número de estantes, de *0* a *m-1*

Primer Modelo

Variables

- Dependiendo de los datos los libros podrán ser colocados en los estantes o no. Sean las variables x_{ij} con valor uno, cero indicando si el libro i se ha colocado en el estante j.

Modelo

$$\max \Sigma_{i,j=0}^{n-1,m-1} x_{ij}$$
$$h_i x_{ij} \leq g_j, \quad i \in [0, n-1], j \in [0, m-1]$$
$$\Sigma_{i=0}^{n-1} w_i x_{ij} \leq A, \quad j \in [0, m-1]$$
$$\Sigma_{j=0}^{m} x_{ij} \leq 1, \quad i \in [0, n-1]$$
$$bin \quad x_{ij}, \quad i \in [0, n-1], j \in [0, m-1]$$

Segundo Modelo

Variables

- Dependiendo de los datos los libros podrán ser colocados en los estantes o no. Sean las variables x_i cuyo valor indicará donde se ha colocado el libro i. Los valores de estas variables verifican $0 \leq x_i \leq m$. Si $x_i = m$ el libro no estará colocado en ningún estante.

Modelo

$$\min \left| C_{i=0|p1(i)}^{n-1} x_i \right|$$
$$h_i \leq g_j, \quad i \in [0, n-1], j \in [0, m-1], p(i,j)$$
$$(\Sigma_{i=0|p2(i,j)}^{n-1} w_i) \leq A, \quad j \in [0, m-1]$$
$$0 \leq x_i \leq m, \quad i \in [0, n-1]$$
$$int \quad x_i, \quad i \in [0, n-1]$$

En lo anterior usamos el predicado $p2(i,j) \equiv (x_i = j)$ que indica si el libro i está en el estante j y el $p1(i) \equiv (x_i = m)$ que indica si no está colocado en ningún estante.

La restricción del sumatorio nos invita a introducir un acumulador del espacio restante en cada estante. Sea este acumulador *r(j).*

El problema generalizado tendrá las propiedades: índice *i*, acumuladores *r(i).*

El problema será válido si *0<=i<=n, r(i)>=0.*

La alternativa a, en principio en el rango *0..m*, se filtrará, porque no conduce a un problema válido, si *r(j)* se hace negativo o el libro *i* no cumple la restricción $h_i \leq g_a$. Siempre estará disponible la alternativa *a =m.*

Tareas solapadas

Se tienen n tareas y cada una está representada por los siguientes 3 elementos: hora de inicio, duración, ganancia asociada. Encontrar el subconjunto de tareas que sin solaparse proporcionen la ganancia máxima.

Datos

Tarea:

- s_i: inicio
- d_i: duración
- g_i: ganancia
- n: número de tareas
- $sp(i,j) = \max(s_i + d_i - s_j, 0)$. Solapamiento entre las tareas i,j. Compartida.
- Las tareas se mantienen ordenadas por s_i

Variables:

- x_i con valores 1,0 indicando si se ha escogido la tarea *i* o no.

Modelo:

$$\max \Sigma_{i=0}^{n-1} g_i * x_i$$
$$x_i + x_j \leq 1, \qquad i,j \in [0, n-1], i < j, sp(i,j) > 0,$$
$$bin \;\; x_i, \quad i \in [0, n-1]$$

El problema generalizado tendrá las propiedades: índice *i*, tareas ya escogidas *h.*

El problema será válido si *0<=i<=n.*

La alternativa *a*, en principio en el rango *0..1*, se eliminará el valor 1 si alguna de las tareas ya escogidas, h, se solapan con la tarea *i*.

TareasSolapadas*Vertex:*

Propiedades:

- index: Integer, básica
- te: Set<Integer>, básica, tareas ya escogidas
- n: Integer, compartida, número de tareas

Interpretación:

Encontrar la elección de tareas, desde index hasta el final, que maximicen las ganancias totales y n se solapen

Igualdad

- Dos problemas son iguales si lo son index, te

Es válido

- index>=0,index<=n

Factoría:

- inicial(): Crea el problema (0,{})
- goal(v) = p.index ==n

Casos base:

1. p.i == n-1

Solución caso Base

1. Tiene solución

Acciones:

- A_i ={0} si $\sum_{j:te} sp(index, j) > 0$
- A_i ={0,1} en otro caso

Vecino

Caso general:

- $neighbord(0) = (i + 1,\ te)$
- $neighbord(1) = (i + 1, te + index)$

Peso de la arista

a*g_i

Peso del camino:

La suma de los pesos de las aristas

Solución Voraz

- Acción Voraz: max(a: A_i)

Heurística:

- $\sum_{i:index..n-1} g_i$

Solución:

- La solución más adecuada sería *Set<Integer>* que indicarían las tareas elegidas. En la solución habría que incluir las propiedades derivadas adecuadas

Recubrimiento de conjuntos

Se tiene un conjunto U (llamado el universo) de n elementos de tipo entero e_j, $j \in [0, n-1]$, y un conjunto S de m conjuntos s_i, cuya unión es igual al universo. Cada conjunto s_i tiene un peso $w_i \geq 0$ asociado. El problema de cobertura de conjuntos consiste en identificar el subconjunto de S cuya unión es igual al universo U y la suma de los pesos de los conjuntos escogidos es mínima.

Definiendo las variables binarias x_i que indicarán si se ha escogido o no el conjunto s_i podemos modelar el problema como

$$min \sum_{i=0}^{m-1} w_i x_i$$

$$\sum_{i=0|e_j \in s_i}^{m-1} x_i \geq 1, \quad j \in [0, n)$$

$$bin\, x_i, \quad i \in [0, m)$$

El problema puede ser resuelto mediante algoritmos genéticos escogiendo un cromosoma binario de tamaño m y función de fitness en función del vector de decode *d*:

$$ft(d) = -\sum_{i=0}^{m-1} w_j d[j] - k \sum_{j=0}^{n-1} dge(-1 + \sum_{i=0|e_j \in s_i}^{m-1} x_i)$$

Para resolverlo por las otras técnicas debemos imaginar un grafo de problemas. Las propiedades del vértice serían: *j* de tipo entero con valores es *[0,m),* las alternativas enteras con valores 0, 1, los pesos de las aristas $a * w_j$, los problemas finales vendrán definidos por *j == m*.

Como una primera aproximación podemos tomar la heurística *(va,p,vf)->0.*

Se propone como ejercicio ampliar las propiedades del vértice para poder filtrar las acciones. Es decir, no escoger 1 si el conjunto *j* ya está cubierto. Igualmente se propone buscar una mejor heurística.

Concretado el grafo, como hemos visto, podemos resolver el problema mediante A*, BT y DPR.

Con las variables binarias x_i que indicarán si se ha escogido o no el conjunto s_i podemos modelar el problema también como:

$$\min \Sigma_{i=0}^{m-1} w_i \, x_i$$
$$\cup_{i=0|x_i=1}^{n-1} s_i = \mathrm{U}$$
$$bin \quad x_i, \quad i \in [0, m)$$

Conjuntos*Vertex:*

Propiedades:

- i: Integer, básica
- ec: Set<Integer>, básica, elementos ya cubiertos
- n: Integer, compartida, número de tareas

Interpretación:

Encontrar la elección de conjuntos, desde index hasta el final, que minimicen el peso total asumiendo es el conjunto de los elementos ya elegidos

Igualdad

- Dos problemas son iguales si lo son index, ec

Es válido

- i>=0,i<=n

Factoría:

- inicial(): Crea el problema (0,{})
- goal(v) = p.i ==n

Casos base:

1. p.i == n-1

Solución caso Base

1. Tiene solución si $ce \cup s_{n-1} = U$

Acciones:

- $A_i = \{0\}$ si $i = n-1, ce = U$
- $A_i = \{1\}$ si $i = n-1, ce \neq U, ce \cup s_{n-1} = U$
- $A_i = \{\}$ si $i = n-1, ce \neq U, ce \cup s_{n-1} \neq U$
- $A_i = \{0\}$ si $i < n-1, ce \cap s_i = \emptyset$
- $A_i = \{1,0\}$ si $i < n-1, ce \cap s_i \neq \emptyset$

Vecino

Caso general:

- $neighbord(0) = (i+1,\ ec)$
- $neighbord(1) = (i+1, ec + s_i)$

Peso de la arista

a*w_i

Peso del camino:

La suma de los pesos de las aristas

Solución Voraz

- Acción Voraz: max(a: A_i)

Heurística:

- 0

Solución:

- La solución más adecuada sería *Set<Integer>* que indicarían los indices de los conjuntos elegidos. En la solución habría que incluir las propiedades derivadas adecuadas

Multiconjunto de enteros

Dado un conjunto de m números enteros estrictamente positivos, encontrar el multiconjunto (se pueden repetir varias veces cada número) formado por números del conjunto anterior que sume exactamente N, y que tenga el menor tamaño. El tamaño de un multiconjunto es la suma de todas las multiplicidades para cada uno de sus elementos.

Sean $e_i,\ i\ \in [0, m-1]$ los números enteros dados y x_i el número de veces que se repetirá cada uno, sus multiplicidades. El modelo del problema es

$$min\ \Sigma_{i=0}^{m-1}\ x_i$$
$$\Sigma_{i=0}^{m-1} x_i e_i = N$$
$$int\quad x_i,\quad i \in [0, m)$$

MultiConjuntos*Vertex:*

Propiedades:

- index: Integer, básica
- sr: Integer, suma restante
- n: Integer, compartida, número de elementos

Es válido

- index>=0,index<=m

Factoría:

- inicial(): Crea el problema (0,{},N)
- goal(v) = p.index ==m

Casos base:

1. p.index == m-1

Solución caso Base

1. Tiene solución si existe a >= 0 tal que $sr - a * e_i = 0$

Acciones:

2. A_i ={a} si p.index == m-1, sr es divisible por e_i, $a = {}^{sr}/_{e_i}$
   - A_i ={} si p.index == m-1, sr no es divisible por e_i
   - A_i ={0..n_i} caso general, $n_i = {}^{sr}/_{e_i}$

Vecino

Caso general:

- $neighbord(a) = (\mathrm{i} + 1,\ \mathrm{sr} - \mathrm{a} * \mathrm{e_i})$

Peso de la arista

a

Peso del camino:

La suma de los pesos de las aristas

Solución Voraz

- Acción Voraz: max(a: A_i)

Heurística:

- 0

Solución:

- La solución más adecuada sería *MultiSet<Integer>* que indicarían el número de veces que se elige cada número. En la solución habría que incluir las propiedades derivadas adecuadas.

Academia

Una academia de ingles tiene n alumnos a ser repartidos en m grupos (n múltiplo de m). Cada grupo tiene distinto horario y profesor. De cada alumno se conoce la afinidad que tiene para pertenecer a cada uno de los grupos (valor entero en el rango [0,5]). Se desea conocer el reparto de alumnos en grupos, de forma que todos los grupos deben tener el mismo

número de alumnos, maximizando la afinidad total conseguida para todos los alumnos, y teniendo en cuenta que no está permitido asignar un alumno a un grupo para el que presente afinidad 0.

Un modelo es definir las variables enteras $x_i,\ i\ \in [0, n-1]$ que indican el grupo asignado al alumno i. Sea $af(i,j),\ i\ \in [0, n-1], j\ \in [0, m-1]$ la afinidad del alumno i con el grupo j. El modelo del problema es:

$$max\ \Sigma_{i=0}^{n-1}\ af(i, x_i)$$
$$C_{i=0|x_i=j, a(i,x_i)>0}^{n-1} x_i = \frac{n}{m},\ \ j \in [0, m)$$
$$0 \leq x_i < m,\ \ i \in [0, n)$$
$$int\ \ x_i\ ,\quad i \in [0, n)$$

AcademiaVertex:

Propiedades:

- index: Integer, básica
- pl: Integer[m], básica, plazas libres por grupos
- n: Integer, compartida, número de alumnos
- m: Integer, compartida, número de grupos
- Neg: Integer, compartida, número estudiantes por grupo

Interpretación:

Encontrar la asignación de estudiantes a grupos, desde index hasta el final, que optimicen la afinidad.

Igualdad

- Dos problemas son iguales si lo son index, plazasLibres

Es válido

- index>=0,index<=n,(plazasLibres[i]>=0, i>=0,i<m)

Factoría:

- inicial(): Crea el problema (0,[Neg,…,Neg])
- goal(v) = p.i ==n // también lo podemos llamar caso base

Casos base:

3. p.i == n-1

Solución caso Base

4. Tiene solución si $|\{a: 0..m-1 | pl[a] > 0\}| = 1$

Acciones:

- $A_i = \{a\}$, si p.i == n-1 y $pl[a] - 1 = 0$ y $pl[b] = 0, b \neq a$
- $A_i = \{\}$, si p.i == n-1 y si $|\{a: 0..m-1 | pl[a] > 0\}| \neq 1$
- $A_i = \{a: 0..m-1 | pl[a] > 0\}$

Vecino

Caso general: $neighbord(a) = (i+1, pl'), \ pl'[a] = pl[a] - 1$

Peso de la arista

Peso: $w = af(i, a)$

Peso del camino:

Suma de los pesos de las aristas

Solución Voraz

- Acción Voraz: $\underset{j:0..m-1}{\text{argmax}}(af(index, j) | pl[j] > 0)$

Heurística:

- $\sum_{i=index}^{n-1} \max_{j:0..m-1} (af(i, j) | pl[j] > 0)$

Solución:

- La solución más adecuada sería *List<Integer>* que indicarían el número del grupo al que se asigna cada alumno

Bufete

Un bufete de abogados cuenta con un equipo de n personas que deben analizar m casos relacionados entre sí ($m \geq n$), y deben terminar dicho análisis global lo antes posible para lo que trabajaran en paralelo. Cada caso será analizado por un único abogado, y cada abogado puede analizar varios casos. Se conoce el tiempo (en horas) que se estima que tarda cada abogado en analizar cada caso concreto (dicho tiempo puede diferir para cada caso en función de que abogado realice el análisis). Determine cual es la mejor asignación de casos a abogados para conseguir el objetivo indicado (terminar de analizar todos los casos lo antes posible).

Un modelo del problema es

$$\min T$$
$$\sum_{i=0}^{n-1} x_{ij} = 1, \quad j \in [0, m-1]$$
$$\sum_{j=0}^{m-1} x_{ij}\, c(i,j) < T, \quad i \in [0, n-1]$$
$$bin\ x_{ij}, \quad i \in [0, n-1], \ \ j \in [0, m-1]$$

Donde asumimos las variables binarias x_{ij} toman valor 1 si el abogado i analiza el caso j y cero si no lo analiza y $c(i,j)$ el tiempo que tarda el abogado i en analizar el caso j.

Otro modelo para este problema es:

$$\min \max_{i:0..n-1} \sum_{j=0|x_j=i}^{m-1} c(i,j)$$
$$0 \leq x_j < n, \quad j \in [0, m-1]$$
$$int\ x_j, \quad j \in [0, m-1]$$

En este modelo la variable x_j indica el abogado al que se asigna el caso j.

BufeteVertex:

Propiedades:

- index: Integer, básica
- ca: Integer[n], básica, carga de abogados
- *cMax: Integer,* carga del abogado más cargado
- *aMin, Integer,* abogado menos cargado
- n: Integer, compartida, número de abogados
- m: Integer, compartida, número de tareas

Interpretación:

Encontrar la asignación de tareas a abogados, desde index hasta el final, que minimicen cMax.

Igualdad

- Dos problemas son iguales si lo son index, ca

Es válido

- index>=0,index<=n

Factoría:

- inicial(): Crea el problema (0,new Integer[n])
- goal(v) = p.i ==n

Casos base:

1. p.i == n-1

Solución caso Base

1. Tiene solución

Acciones:

- $A_i = \{a\}, a =$ a = $\underset{a}{\arg\min}(ca[a] + c(a,i))$, si p.i = n-1
- $A_i = \{a: 0..n-1\}$, caso general

Vecino

Caso general: $neighbord(a) = (i+1, ca'),\ \ ca'[a] = ca[a] + c(a,i)$

Peso de la arista

No se le puede asignar peso a la arista

Peso del camino:

La propiedad cMax del último vértice

Solución Voraz

- Acción Voraz: a = $\underset{a}{\text{arg\_min}}(ca[a] + c(a,i))$

Heurística:

- La propiedad cMax del último vértice

Solución:

- La solución más adecuada sería *Map<Integer,Integer>* que indicarían el abogado al que se asigna cada tarea. En la solución habría que incluir las propiedades derivadas adecuadas

Productos y precios

Se tienen n productos, cada uno de los cuales tiene un precio y presenta una serie de funcionalidades (el mismo producto puede tener más de una funcionalidad). Se desea diseñar un lote con una selección de dichos productos que cubran un conjunto de funcionalidades deseadas entre todos productos seleccionados al menor precio.

Sea un conjunto finito de funcionalidades F que podemos representar por una lista de tamaño q de funcionalidades y un conjunto de índices $D = \{d_0, d_1, \ldots, d_{m-1}\}$ de tamaño m que indique los índices a las funcionalidades deseadas.

Consideremos la función $\varphi(j) = \{i: 0..n-1 |\, d_j \in f_i\}$. Esta función devuelve un conjunto que contiene los índices de los productos que ofrecen la funcionalidad d_j. Un modelo del problema es

$$\min \sum_{i=0}^{n-1} p_i x_i$$
$$(\sum_{i:\varphi(j)} x_i) \geq 1, \quad j \in [0, m)$$
$$bin\ x_i, \quad i \in [0, n)$$

Sean las variables binarias $x_i, i \in [0, n-1]$ que indica si se ha elegido el producto i. A su vez p_i y f_i sean el precio y las funcionalidades ofrecidas, un conjunto de índices, del producto i.

Otro modelo del problema es

$$\min \sum_{i=0}^{n-1} p_i\ x_i$$
$$\mathrm{U}_{i=0|x_i=1}^{n-1} f_i \supset D$$
$$bin\ x_i, \quad i \in [0, n-1]$$

Aquí considerado como D un conjunto obtenido de la lista con el mismo nombre.

ProductosVertex:

Propiedades:

- index: Integer, básica
- *fc: Set<Integer>, básica,* las funcionalidades ya cubiertas
- n: Integer, compartida, número de productos
- m: Integer, compartida, número de funcionalidades

Interpretación:

Encontrar la elección adecuada de productos, desde index hasta el final, que minimicen el precio total y cubran todas las funcionalidades

Igualdad

- Dos problemas son iguales si lo son index, fc

Es válido

- index>=0,index<=n

Factoría:

- inicial(): Crea el problema (0,{})
- goal(v) = p.i ==n

Casos base:

1. p.i == n-1

Solución caso Base

1. Tiene solución si ya están todas las funcionalidades cubiertas o si $fc \cup f_{n-1} \supseteq D$

Acciones:

- En el caso p.i = n-1
  - {0} Si $fc \supseteq D$
  - {1} Si ! $(fc \supseteq D), fc \cup f_{n-1} \supseteq D$
  - {} Si $fc \cup f_{n-1} \supseteq D$
- $A_i = \{a: 0..1\}$, caso general
  - 0 siempre
  - 1 si $|fc \cup f_{n-1}| > fc$

Vecino

Caso general:

- $neighbord(0) = (i+1,\ fc)$
- $neighbord(1) = (i+1, fc \cup f_i)$

Peso de la arista

a*p_i

Peso del camino:

La suma de los pesos de las aristas

Solución Voraz

- Acción Voraz: max(a: A_i)

Heurística:

- 0

Solución:

- La solución más adecuada sería *Set<Integer>* que indicarían los productos elegidos. En la solución habría que incluir las propiedades derivadas adecuadas

Partición de conjuntos

Dado un conjunto de enteros determinar si puede particionarse en tres subconjuntos de manera que la suma de elementos en los tres subconjuntos sea la misma, y que el tamaño de uno de ellos sea lo menor posible.

Sean $e_i,\ i\ \in [0, n-1]$ los n elementos de tipo entero del conjunto de partida U y $x_{ij}, i\ \in [0, n-1], j \in \{0,1,2\}$ variables binarias que indican a que el elemento e_i se asigna al subconjunto j. Un modelo del problema es

$$\min \sum_{i=0|j=0}^{n-1} x_{ij}$$

$$\sum_{j=0}^{2} x_{ij}\ = 1, \quad i \in [0, n-1]$$

$$\sum_{i=0}^{n-1} x_{ij} e_i\ = \frac{1}{3} \sum_{i=0}^{n-1} e_i\,, \quad j \in \{0,1,2\}$$

$$bin\ x_{ij}, \quad i \in [0, n-1], j \in \{0,1,2\}$$

Otro modelo del problema con $x_i, i \in [0, n-1]$ variables enteras que indiquen a que subconjunto se asigna el elemento i. Sea $N = \frac{1}{3}\sum_{i=0}^{n-1} e_i$ un modelo puede ser:

$$\min C_{i=0|x_i=0}^{n-1} x_i$$
$$x_i \leq 2, \quad i \in [0, n-1]$$
$$\sum_{i=0|x_i=j}^{n-1} e_i = N, \quad j \in \{0,1,2\}$$
$$int\ x_i, \quad i \in [0, n-1]$$

Particion*Vertex:*

Propiedades:

- index: Integer, básica
- vr: Integer[3], básica, N – la suma de los elementos asignados a cada conjunto
- n: Integer, compartida, número de elementos

Interpretación:

Encontrar la elección adecuada de productos, desde index hasta el final, que minimicen el precio total y cubran todas las funcionalidades

Igualdad

- Dos problemas son iguales si lo son index, fc

Es válido

- index>=0,index<=n, vr[i]>=0, i en 0..2

Factoría:

- inicial(): Crea el problema (0,[N,N,N])
- goal(v) = p.i ==n

Casos base:

1. p.i == n-1

Solución caso Base

1. Tiene solución si existe a tal que $vr[i] = 0, i \neq a, vr[a] - e_i = 0$

Acciones:

- En el caso p.i = n-1
  - {a} Si $vr[i] = 0, i \neq a, vr[a] - e_i = 0$
  - {} En otro caso
- $A_i = \{a: 0..2 | vr[a] - e_i \geq 0\}$, caso general

Vecino

Caso general: $neighbord(0) = (i + 1,\ vr'), vr'[a] =\ vr[a] - e_i$

Peso de la arista

a==0?1:0

Peso del camino:

La suma de los pesos de las aristas

Solución Voraz

- Acción Voraz: $\underset{j:A_i}{\text{argmax}}\, vr[a]$

Heurística:

- 0.

Solución:

- La solución más adecuada sería *Map<Integer,Integer>* que indicarían los conjuntos asignados a cada elemento. En la solución habría que incluir las propiedades derivadas adecuadas

Camino cerrado

Se tiene un grafo g y un predicado sobre sus aristas. Se desea saber cuál es el camino simple y cerrado más corto que pase por todos los vértices del grafo una sola vez y que contiene al menos una arista que cumple el predicado.

Asumimos que los vértices pueden ser representados por números enteros. Sea x_i una variable entera que representa el vértice en la posición i del recorrido, y_{ij} una variable binaria que indica si se ha escogido la arista (i, j). Sea φ el conjunto de las aristas que cumplen el predicado dado y w_{ij} el peso de la arista (i, j).

El problema puede ser enfocado como una red de flujo en grafos con ciclos. Definimos las siguientes variables:

- Dos variables binarias por arista. 1 si pasa flujo. 0 si no pasa
- Una variable entera por vértice para indicar el orden del vértice en el camino
- Leyes de *Kirchoff* para todos vértices
- Adicionalmente una restricción para impedir bucles cerrados de tamaño menor que n pero permitir el de tamaño n que incluye todos los vértices.

$$min \sum_{i=0,j=0|\,(i,j)\in g}^{n-1} w_{ij}y_{ij}$$

$$\sum_{i=0|\,(i,j)\in g}^{n-1} y_{ij} = 1, \qquad j \in [1, n-1]$$

$$\sum_{j=0|\,(i,j)\in g}^{n-1} y_{ij} = 1, \qquad i \in [1, n-1]$$

$$x_i - x_j + n\,y_{ij} \leq n-1, \qquad i \in [1, n-1], j \in [1, n-1]$$

$$x_i \leq n-1, \qquad i \in [0, n-1]$$

$$x_0 = 0$$

$$\sum_{i,j=0|\varphi(i,j)}^{n-1} y_{ij} \geq 1$$

$$int\ x_i, \qquad i \in [0, n-1]$$

$$bin\ y_{ij}, \qquad i \in [0, n-1], j \in [1, n-1], (i,j) \in g$$

Un modelo alternativo sería escoger las variables enteras x_i que representen el vértice en la posición *i* del recorrido.

$$min \sum_{i=0}^{n-1} w(e(x_i))$$

$$\exists_{i=0}^{n-1}\,(e(x_i)\ \in \varphi)$$

$$CP_{i=0|g}^{n-1} x_i$$

$$int\ x_i, i \in [0, n-1]$$

Con $e(x_i)$ la arista que conecta el vértice x_i del recorrido con el siguiente $x_{(i+1)\%n}$, $w(e(x_i))$ su peso y φ el conjunto de aristas que cumplen la propiedad. La restricción $CP_{i=0|g}^{n-1} x_i$ indica que las variables forman un camino simple cerrado del grafo. La restricción con el operador ∧ puede ser reformulada como $\sum_{i=0|e(x_i)\,\in\varphi}^{n-1} 1 \geq 1$

ParticionVertex:

Propiedades:

- cm: List<Integer>, básica, camino ya escogido
- na: Integer, numero de aristas que cumplen la restricción
- n: Integer, compartida, número de vértices
- g: Graph<Integer,SimpleEdge<Integer>, grafo

Interpretación:

Encontrar el camino cerrado mínimo más corto, desde index hasta el final, considerando que se ya se ha recorrido cm.

Igualdad

- Dos problemas son iguales si lo son index, cm

Es válido

- cm es un camino

Factoría:

- inicial(): Crea el problema (0,[0],0)
- goal(v) = |cm| ==n+1

Casos base:

1. |cm| == n

Solución caso Base

1. Tiene solución si existe la arista (index,0) y na >1 o la arista (index,0) cumple la condición

Acciones:

- En el caso *|cm| == n*
  - {a} Si $a \in graph.edgesOf(n-1), (na \geq 1 \; o \; a \in \varphi)$
  - {} En otro caso
- $A_i = \{a: g.edgesOf(index) | ov(a, index) \notin cm\}$, caso general

Vecino

Caso general:

$$neighbord(a) = (ov(a, index), cm + ov(index), a \in \varphi : na + 1 : na)$$

Peso de la arista

g.getEdgeWeight(a)

Peso del camino:

La suma de los pesos de las aristas

Solución Voraz

- Acción Voraz: $\underset{a:A_i}{\operatorname{argmax}}\, g.getEdgeWeight(a)$

Heurística:

- 0.

Solución:

- La solución más adecuada sería *Lis<Integer>* que indicarían el orden de los vértices que forma el camino. En la solución habría que incluir las propiedades derivadas adecuadas

Ruta de tren

Hay n estaciones en la ruta de un tren. El tren va desde la estación *0* hasta la *n-1*. El coste del billete para el par de estaciones *(i, j)* es *c(i, j).* Además, se cumple que *c(i, k) > c(i, j)* para todo *k > j.* Diseñe un algoritmo que minimice el coste para llegar desde el origen al destino, indicando también los billetes que habría que comprar. Los datos de los costes y los posibles caminos vienen proporcionados en un grafo dirigido cuyos pesos de las aristas dan los costes y sólo existen aristas de vértices *i, j* que cumplen *j>i*. Es, por lo tanto, un grafo dirigido.

Datos

- *g*: *Graph<Integer,SimpleEdge<Integer>>*
- *c(i,j):* coste del billete de i a j, dado por *c(i,j) = graph.getEdge(i,j).getEdgeWeight()*
- *n*, número de estaciones, n = graph.vertexSet().size();

Un primer modelo

Variables:

- x_{ij} con valores *1,0* indicando si coge la arista que va del vértice i al j.

Modelo

$$min\ \Sigma_{i,j|p(i,j)} c(i,j) * x_{ij}$$
$$\Sigma_{j|p(0,j)} x_{0j} = 1$$
$$\Sigma_{i|p(i,n-1)} x_{i,n-1} = 1$$
$$\Sigma_{j|p(j,i)} x_{ji} - \Sigma_{j|p(i,j)} x_{ij} = 0, \quad i \in [1, n-2]$$
$$bin\ \ x_{i,j}, \quad i,j \in [0, n-1], p(i,j)$$

El modelo está inspirado en una red de flujo donde se obliga a que el flujo por las aristas sea cero o uno y la fuente y sumidero crean y consumen uno respectivamente. Hemos usado el predicado $p(i,j) \equiv (i,j) \rightarrow g.containsEdge(i,j)$ y los coste $c(i,j)$ desde el vértice *i* al *j*.

Segundo Modelo

Variables:

- x_i con valores enteros indicando el número de una de las estaciones donde se para.
- r es el número de ciudades del trayecto

Modelo

$$min \sum_{i=0}^{r-2} c(x_i, x_{i+1})$$
$$x_0 = 0$$
$$x_{r-1} = n - 1$$
$$OP_{i=0|g}^{r-1} x_i$$
$$x_i < n, i \in [0, n)$$
$$2 \leq r < n$$
$$int\ r$$
$$int\ x_i, i \in [0, r-1]$$

En ambos casos el problema generalizado tendrá las propiedades: índice *i.*

El problema será válido si *0<=i<n.*

La alternativa *a* en el rango *i..n-1*.

Elementos y contenedores

Se tiene un conjunto de elementos y un conjunto de contenedores. De cada contenedor se conoce su tipo y su tamaño. De cada elemento se conocen los tipos de contenedores en los que se puede ubicar, y su tamaño. Se desea ocupar totalmente el mayor número de contenedores posible haciendo uso de los elementos de los que se dispone.

Se quiere saber los elementos que ubican cada uno de los contenedores seleccionados, es decir, en qué contenedor está ubicado cada elemento seleccionado.

Modelo

De lo anterior consideramos necesarias variables binarias $x_{ij},\ i \in [0, n)$, $j \in [0, m)$ que indican si el elemento i se ha ubicado en el contenedor j. Consideremos también variables binarias $y_j,\ j \in [0, m)$ que indican si el contenedor j está totalmente ocupado. Sea $e_i,\ i \in [0, n)$ el tamaño del elemento i. Sea $c_j,\ j \in [0, m)$ el tamaño del contenedor j. Sea $w_{ij},\ i \in$

$[0,n),\ j \in [0,m)$ una función booleana que indica si el elemento i puede ubicarse en el contenedor j. El modelo del problema sería:

$$max\ \Sigma_{j=0}^{m-1}\ y_i$$
$$x_{ij} \leq w_{ij}, \qquad i \in [0,n), j \in [0,m)$$
$$\Sigma_{i=0}^{n-1}\ e_i x_{ij}\ =\ c_j y_j, \qquad j \in [0,m)$$
$$\Sigma_{j=0}^{m-1}\ x_{ij} \leq 1, \qquad i \in [0,n)$$
$$bin\ \ x_{ij}, i \in [0,n), j \in [0,m)$$
$$bin\ \ y_j, j \in [0,m)$$

Es un modelo adecuado para resolver el problema mediante PLE.

Modelo alternativo

Consideremos variables enteras $x_i,\ i \in [0,n]$, que indican el contenedor en el que se ha ubicado elemento i, de forma que si $x_i = m$, el i-ésimo elemento no será ubicado en ningún contenedor. Sea $e_i,\ i \in [0,n)$ el tamaño del elemento i. Sea $c_j,\ j \in [0,m)$ el tamaño del contenedor j. Sea $z(i,j),\ i \in [0,n),\ j \in [0,m)$ una función booleana que devuelve true si el elemento i no puede ubicarse en el contenedor j. Consideremos también funciones $\varphi(j) \equiv \Sigma_{i=0\,|\,x_i=j}^{n-1}\ e_i = c_j,\ j \in [0,m)$, que indican si el contenedor j está totalmente ocupado. El modelo del problema sería:

$$max\ C_{j=0|\varphi(j)}^{m-1}\ 1$$
$$x_i \leq m, i \in [0,n]$$
$$\forall_{i=0}^{n-1} z(i,x_i)$$
$$\Sigma_{i=0\,|\,x_i=j}^{n-1}\ e_i \leq\ c_j, \quad j \in [0,m)$$
$$intx_i, i \in [0,n]$$

Es un modelo adecuado para resolver le problema mediante Algoritmos Genéticos.

Para resolverlo por el resto de las técnicas lo generalizamos:

VertexContenedores:

Propiedades:

- *i: Integer*
- *cR: List<Integer>,* la capacidad restante en cada contenedor
- *cC: Set<Integer>,* derivada, contenedores completos.

Interpretación:

Encontrar la asignación de los elementos desde i hasta el final en los contenedores que quepan, puedan ublicarse y no estén llenos.

Igualdad

- Dos problemas son iguales si lo son i, capRest

Es válido

- *i>=0, i<=n, capRest[i]>=0*

Factoría:

- *inicial():* Crea el problema *(0,[c0,c1,...]).* Donde c0, etc., son la capacidades de los contenedores
- *goal(v) = v.i ==n*

Caso base:

- v.i ==n

Solución caso base

- Tiene solución porque asumimos que el número de contenedores es suficiente

Acciones:

$A_i = a: \{0..m\} | a \notin cC, pu(i,a), t(i) \leq cR(a)\}$

Donde $t(i)$ es el tamaño del elemeno i y $pu(i,a)$ indica que el elemento i se puede ubicar en el contenedor a.

Vecino

Caso general: $neighbord(a) = (i + 1, cR(a) - t(i))$

Peso de la arista: No asignamos peso a la arista

Peso del camino: La propiedad cC del último vértice

Solución Voraz:

- Escoger aleatoriamente una alternativa de entre las posibles. Repetir varias veces y quedarse con la mejor.
- Escoger el contenedor que queda más lleno tras incorporar el elemento (o alternativamente el más vacío). Repetir varias veces y quedarse con la mejor.

Podemos ordenar los elementos según su tamaño de mayor a menor (o alternativamente de menor a mayor).

Experimentar con las opciones anteriores para ver la mejor combinación.

Heurística: El número de contenedores llenos más la diferencia entre el número total de contenedores y el índice actual. Es una heurística admisible.

Solución: Map<Integer,List<Integer>> que indica los elementos asignados a cada contenedor.

Productos y componentes

Una fábrica elabora distintos tipos de componentes. La unidad de cada tipo de componente tarda un tiempo de producción más un tiempo manual en elaborarse. Cada semana, el tiempo total de producción y el tiempo total de acabado manual no deben exceder unos límites establecidos (Tp y Te, respectivamente). Finalmente, estos componentes se ensamblan para obtener ciertos productos acabados. Cada producto está compuesto de varias unidades de cada componente, y se venden por precios diferentes. Cada semana se pueden vender como máximo unas

determinadas unidades de cada producto. Se desea planificar la producción semanal para maximizar los ingresos totales.

Se quiere conocer cuántas unidades de cada producto deben fabricarse.

Modelo

De lo anterior consideramos necesarias variables enteras $x_i,\ i \in [0,n)$ que indican cuantas unidades del producto i serán fabricadas. Sea $v_i,\ i \in [0,n)$ el precio de venta del producto i. Sea $m_i,\ i \in [0,n)$ el máximo número de unidades del producto i que pueden venderse semanalmente. Sea $u_{ij},\ i \in [0,n),\ j \in [0,m)$ el número de unidades del componente j que requiere el producto i. Sea $t_j,\ j \in [0,m)$ el tiempo de producción del componente j. Sea $e_j,\ j \in [0,m)$ el tiempo de elaboración manual del componente j. Sea $tp(i) = \Sigma_{j=0}^{m-1} t_j u_{ij}$ y $te(i) = \Sigma_{j=0}^{m-1} e_j u_{ij}$

El modelo del problema sería:

$$
\begin{gathered}
max\ \Sigma_{i=0}^{n-1}\ v_i x_i \\
x_i \leq m_i, \qquad i \in [0,n) \\
\Sigma_{i=0}^{n-1}\ tp(i)\ x_i \quad \leq Tp \\
\Sigma_{i=0}^{n-1}\ te(i)\ x_i \quad \leq Te \\
int\ \ x_i, i \in [0,n)
\end{gathered}
$$

El modelo es adecuado para resolver el problema mediante PLI o por Algoritmos Genéticos.

Para resolverlo por el resto de las técnicas lo generalizamos:

VertexProductos:

Propiedades:

- *i: Integer*
- *tpR: List<Integer>,* tiempo de producción restante
- *teR: Set<Integer>,* tiempo de elaboración restante

Interpretación:

Encontrar el número de unidades a producir de lo s productos desde i hasta el final teniendoo en cuenta que los tiempos restantes de producción y elaboración son *tpR* y *teR*.

Igualdad

- Dos problemas son iguales si lo son i, tpR,TeR

Es válido

- *i>=0, i<=n, tpR>=0, teR>=0*

Factoría:

- *inicial():* Crea el problema *(0,Tp,Te).*
- *goal(v) = v.i ==n*

Caso base:

- v.i ==n

Solución caso base

- Tiene solución

Acciones:

$A_i = a: \{0..m_i\} | a * tp(i) \leq tpR, a * te(i) \leq teR\}$

Vecino

Caso general: $neighbord(a) = (i + 1, tpR - a * tp(i), teR - a * te(i))$

Peso de la arista: $a * v_i$

Peso del camino: La suma de los pesos de las aristas

Solución Voraz:

Escogemos como acción voraz la mayor cantidad de unidades del producto *i* que pueden ser fabricadas y elaboradas en el tiempo restante de fabricación y elaboración.

Ordenamos los productos según el beneficio esperado al comienzo. El beneficio esperado es la cantidad máxima de unidades anterior multiplicada por su precio.

Heurística: Escogemos como acción heurística la acción voraz más uno.

Solución: Multiset<Integer> que indica el número de unidades del producto *i*.

Algoritmo de Floyd

Veamos el algoritmo de Floyd para calcular el camino mínimo en un grafo. Es un algoritmo de Programación Dinámica General donde tras tomar una acción nos encontramos con dos subproblemas y por lo tanto nos aparece un hipergrafo en lugar de un grafo.

Asumimos que cada vértice del grafo está indexado mediante un entero comprendido en $[0, n-1]$ dónde n es el número de vértices. Esto siempre lo podemos conseguir mediante una vista de un grafo cuyos vértices sean enteros y la arista del tipo *SimpleEdge<Integer>* por ejemplo. Queremos encontrar el *Camino Mínimo* entre dos ciudades dadas *c1, c2*. Este problema se puede resolver de muchas otras formas como hemos visto. Aquí vamos a resolver el problema para que sirva de ejemplo de uso del algoritmo de Programación Dinámica. Para ello lo generalizamos a este otro: encontrar el *Camino Mínimo* de *i* a *j* usando como camino intermedio ciudades cuyos índices estén en el conjunto $[0, k)$. Con este planteamiento cada problema lo podemos representar por (i, j, k, g). Donde i, j toman valores en $[0, n-1]$ y k en $[0, n]$. El valor de $k = n$ indica que el camino intermedio no contiene ninguna ciudad.

Un camino en el grafo anterior lo vamos a representar por una secuencia de ciudades conectadas cada una a la siguiente mediante una arista. Cada

camino tiene una longitud que es la suma del valor de sus aristas. La no existencia de camino lo representaremos por ⊥.

Representamos por g el grafo de partida y por $e(i,j)$ la arista entre i y j, si la hay, y por $w(i,j)$ su peso. La primera decisión es escoger el tipo de alternativas. Para cada problema tenemos dos alternativas $\{Y,N\}$. La primera alternativa representa pasar por la ciudad k. La segunda no pasar. Veamos los detalles del problema:

FloydVertex:

Propiedades:

- i: Integer
- j: Integer
- k: Integer
- g: Graph<Integer,SimpleEdge<Integer>, compartida
- v1: Integer, compartida, ciudad inicial
- v2: Integer, compartida, ciudad final
- t: Integer, derivada, t = n-k
- n: Integer, compartida, número de vértices del grafo

Interpretación:

*E*ncontrar en el grafo g el *Camino Mínimo* de i a j usando en el camino intermedio ciudades cuyos índices estén en el conjunto $[k,n)$.

Igualdad

- Dos problemas son iguales si lo son i, j, k

Es válido

- i>=0, i<n, j>=0, j<n, k>=0, k<=n

Factoría:

- inicial(): Crea el problema (v1,v2,0)

Casos base:

1. (i,j,n)
2. (i,j,k) y e(i,j) pertenece al grafo

Solución casos Base

1. Tiene solución si e(i,j) pertenece al grafo y es [i,j]. En otro caso no tiene.
2. Tiene solución y es [i,j]

Acciones:

$A_i = \{Y, N\}$

Vecino

Caso general:

- $neighbords(N) = [(i, j, k+1)]$
- $neighbords(Y) = [(i, k, k+1), (k, j, k+1)]$

Nota: Ahora el grafo es un *hypergrafo*, las aristas *hiperaristas* y las soluciones de este tipo de problemas son árboles en vez de caminos. *Peso del árbol:* Suma de los pesos de los árboles hijos. *Solución: List<Integer>* con los vértices que incluyan el camino.

Multiplicación de matrices encadenadas

Dada una secuencia de matrices $M = \{m_0, m_1, \dots, m_{k-1}\}$ donde la matriz m_i tiene f_i filas y c_i columnas con $c_i = f_{i+1}$ queremos obtener su producto. Este producto puede obtenerse de diversas maneras según decidamos agruparlas. Para multiplicar dos matrices m_i, m_{i+1} es necesario hacer $f_i c_i c_{i+1}$ (o $f_i f_{i+1} c_{i+1}$) multiplicaciones. Queremos obtener la forma de agruparlas para conseguir el número mínimo de multiplicaciones.

Para abordar el problema lo generalizamos introduciendo dos índices i, j. El problema generalizado busca la forma óptima de agrupar las matrices

en la sublista $M[i,j]$. Si asumimos que $n_{i,j}$ es el número óptimo de multiplicaciones en $M[i,j]$ podemos observar que se cumple la relación $n_{i,j} = n_{i,k} + n_{k,j} + f_i f_k c_{j-1}$ para todos los $s \in (i+1, j)$. Es decir si dividimos el intervalo (i,j), en el sentido de Java, en dos partes (i,k), (k,j), el número de multiplicaciones necesarias será la suma de las multiplicaciones de cada subgrupo más las que hacen falta para multiplicar los resultados de los dos bloque resultantes. Hemos de tener en cuenta que la matriz resultante del subgrupo izquierdo tendrá f_i filas y f_k columnas y el subgrupo derecho f_k filas y c_{j-1} columnas.

El problema se nos presenta como un problema recursivo donde la solución de un problema se puede expresar a partir de las soluciones de los subproblemas pero tenemos todos los valores de k como opciones. Estas opciones son las alternativas posibles en el planteamiento de Programación Dinámica.

La solución que vamos buscando es un *String* con los paréntesis puestos adecuadamente para multiplicar las matrices de la lista.

El problema generalizado será:

Matriz:

- f: Integer, numero de filas
- c: Integer, numero de columnas

MatrizVertex:

Propiedades:

- i: Integer
- j: Integer
- lm: List<Matriz>, compartida
- t: Integer, derivada, t = j-i
- n: Integer, compartida, n= número de matrices en lm

Interpretación:

*E*ncontrar una forma de agrupar las matrices en *ls[i,j]* que minimice el número de multiplicaciones necesarias para conseguir el producto de todas las matrices en *ls[i,j]*.

Igualdad

- Dos problemas son iguales si lo son i, j

Es válido

- i>=0, i<n, j>i, j<n

Factoría:

- inicial(): Crea el problema (0,n)

Casos base:

1. n-i =0
2. n-i = 1
3. n-i=2

Solución casos Base

1. Tiene solución con peso 0
2. Tiene solución con peso 0
3. Tiene solución con peso $f_i c_i c_{i+1}$

Acciones:

$A_i = i + 1..j - 1$

Vecino

Caso general:

- $neighbords(a) = [(i, a), (a, j)]$

Nota: Ahora el grafo es un *hypergrafo*, las aristas *hiperaristas* y las soluciones de este tipo de problemas son árboles en vez de caminos.

Peso del árbol: Las soluciones de este problema son árboles. Cada árbol tiene asociado un vértice del hipergrafo virtual, un peso, una acción y unos vecinos siguiendo esa acción. En este caso será un caso base o tendrá dos hijos. El peso de un árbol t con hijos t_0, t_1 es

$$t.n = t_0.n + t_1.n + lm[t_0.i].f * lm[t_0.i].f * lm[t_1.j].c$$

Solución: String

Notación y catálogo de restricciones de uso general

Para uso en todo el libro usaremos una notación compacta que tiene una equivalencia directa en Java. Esta notación cuenta con tipos de datos conocidos y operadores sobre los mismos.

Tipos de datos

Los tipos de datos que usaremos serán listas, conjuntos, multiconjuntos, diccionarios y bidiccionarios.

Las *listas* son agregados de elementos que pueden repetirse y que están indexados por un entero en un rango. El orden es importante en las listas. Las representaremos como $r = [e_0, \dots, e_{n-1}]$. El tipo que representa las listas en Java es *List<T>*.

Los *conjuntos* son agregados de elementos que no se pueden repetir, no están indexados y donde no importa el orden. Los representamos como $s = \{e_0, \dots, e_{n-1}\}$. El tipo que representa los conjuntos en Java es *Set<T>*.

Los *multiconjuntos* son agregados de elementos que se pueden repetir, no están indexados y donde no importa el orden. Los representamos como $ms = \{e_0 : f_0, \dots, e_{:n-i} : f_{n-1}\}$. Donde las f_i son enteros que indican el número de veces que se repite el elemento e_i El tipo que representa los

multiconjuntos en Java es *Multiset<T>* y su implementación puede encontrarse en el repositorio. Este tipo, junto a los métodos de los conjuntos, tiene los métodos *count(E e),* que nos proporciona el número de repeticiones del elemento e, y *add(E e, Integer n)*, que añade n repeticiones del elemento *e.*

Los *diccionarios* son conjuntos de pares formados por una clave y un valor que modelan una función inyectiva entre el conjunto de las claves, los primeros elementos de los pares, y el de los valores, los segundos elementos. Los representamos como $d = \{k_0 : v_0, \dots, k_{:n-i} : v_{n-1}\}$. Donde las k_i son las claves y los v_i los valores asociados. En este tipo las claves no pueden estar repetidas. El tipo que representa los diccionarios en Java es *Map<K,V>*.

Los *bidiccionarios* son conjuntos de pares formados por una clave y un valor que modelan una función biyectiva entre el conjunto de las claves, los primeros elementos de los pares, y el de los valores, los segundos elementos. Los representamos como $d = \{k_0 : v_0, \dots, k_{:n-i} : v_{n-1}\}$. Donde las k_i son las claves y los v_i los valores asociados. En este tipo las claves y los valores no pueden estar repetidos. El tipo que representa los diccionarios en Java es *BiMap<K,V>.* Este tipo tiene el método *inverse()* que devuelve un *BiMap<V,K>* que es el inverso del anterior *asMap()* que nos devuelve un objeto de tipo *Map<K,V>.*

Los flujos de datos los representaremos según los casos por los tipos de Java *Iterable<E>* o el tipo *Stream<E>.* Usaremos uno u otro según la conveniencia en cada caso, pero preferiremos el uso del segundo frente al primero.

Los grafos de distintos tipos los representaremos por el tipo *Graph<V,E>* de la librería *JGraphT*.

Los caminos en los grafos los se representan por el tipo *GraphPath<V,E>* también de *JGraphT*.

Junto con los tipos usaremos una notación para designar de forma compacta las propiedades y operaciones más usuales.

- $|r|$: Tamaño del conjunto, lista, número de claves de un diccionario o número de elementos distintos en un multiconjunto.

- $||r||$: Número de elementos de un multiconjunto teniendo en cuenta las veces que se repite cada elemento.
- g_v, g_e: El conjunto de vértices y de aristas de un grafo respectivamente.
- d_k, d_v: El conjunto de claves y valores de un diccionario respectivamente.
- $r + e$: Añadir un elemento al final de una lista o a un conjunto.
- $r + (k, v)$: Añadir un par clave-valor a un diccionario o una lista, posiblemente sustituyendo al antiguo valor asociado a la clave. En el caso de una lista k debe ser entero y entonces se actualiza la posición *r[k].*
- $r - e$: Eliminar un elemento de un conjunto, lista o diccionario. En el caso de una lista si e es un entero se elimina la casilla correspondiente.
- $r[k], r_k$: Elemento de índice k en una lista si k es un entero, el valor asociado a la clave k en un diccionario o el número de apariciones de k en un multiconjunto.
- $r[i: j]$: Sublista desde la casilla i a la j sin incluir la j.
- $r[: i]$: Sublista de 0 a i sin incluir la i.
- $r[i:]$: Sublista desde i hasta el final.
- $r1 + r2$: Concatenar listas, unir dos conjuntos, multiconjuntos o dos diccionarios teniendo en cuenta, en este último caso, si hay claves comunes en *r1* y *r2* los valores asociados a la clave común serán los de *r2*.
- $r1 - r2$: Eliminar de *r1* los elementos en *r2*. Si son conjuntos es la diferencia de conjuntos. Si son diccionarios el diccionario cuyas claves son la diferencia de las claves. Si son multiconjuntos el formado por los elementos de *r1* restándole las de *r2* con un mínimo de cero. Si son listas se elimina de la primera todas las ocurrencias de los elementos de la segunda.
- $r1 \oplus r2$: Concatenar dos listas tales que el último elemento de la primera coincide con el primero de la segunda y manteniendo solo una vez el elemento repetido.
- Los tipos de las funciones los expresaremos de forma compacta indicando los tipos de los parámetros y el resultado. Así $E \times T \rightarrow$

R es el tipo de una función, o una lambda expresión que toma dos parámetros de tipos E, T y devuelve un resultado de tipo R.

Operadores

Prácticamente todo el código en java con *streams* puede compactarse en operadores con la forma:

$$O_{e:d|p} f(e)$$

Donde e es un elemento que recorre los valores de un dominio d, usualmente un rango de valores o un flujo de datos, que es filtrada por un predicado $p(e)$, transformada por una función $f(e)$ y acumulado por el operador O.

En la mayoría de los casos cuando el dominio es un rango simple de enteros se explicita:

$$O_{i=0|p}^{n} f(i)$$

Algunos ejemplos y sus equivalencias en Java:

Sum: $\Sigma_{i=0|p(i)}^{n-1} f(i)$:

```
IntStream.range(0,n).boxed().filter(p).mapToInt(f).sum();
```

Máximo: $\max_{i:0..n-1} f(i)$. Tenemos la equivalencia:

$$\max_{i:0..n-1} f(i) = \mathrm{y} \quad \equiv \quad \min \mathrm{y}, \qquad \mathrm{V}_{i=0}^{n-1} f(i) \leq \mathrm{y}$$

```
IntStream.range(0,n).boxed().map(f).max(cmp).get();
```

Mínimo: $\min_{i:0..n-1} f(i)$: Tenemos la equivalencia:

$$\min_{i:0..n-1} f(i) = y \quad \equiv \quad \max y, \qquad \mathrm{V}_{i=0}^{n-1} f(i) \geq \mathrm{y}$$

```
IntStream.range(0,n).boxed().map (f).min(cmp).get();
```

Argmax: $\underset{i:0..n-1}{\text{argmax}}\, f(i)$. Tenemos la equivalencia:

$$\underset{i:0..n-1}{argmax}\, f(i) = k \quad \equiv \quad \vee_{i=0}^{n-1} f(i) \leq f(k), \qquad k \in [0..n-1]$$

```
IntStream.range(0,n).boxed()
       .max(Comparator.comparing(f).get();
```

Argmin: $\underset{i:0..n-1}{\text{argmin}}\, f(i)$. Tenemos la equivalencia:

$$\underset{i:0..n-1}{argmin}\, f(i) = k \quad \equiv \quad \vee_{i=0}^{n-1} f(i) \geq f(k), \qquad k \in [0..n-1]$$

```
IntStream.range(0,n).boxed()
       .min(Comparator.comparing(f).get();
```

Unión: $\cup_{i=0|p(i)}^{n-1} f(i)$:

```
IntStream.range(0, n).boxed().filter(pd)
             .map(f).reduce((s1,s2)->Set2.union(s1,s2)).get();
```

Intersección: $\cap_{i=0|p(i)}^{n-1} f(i)$:

```
IntStream.range(0, n).boxed().filter(pd)
             .map(f)
             .reduce((s1,s2)->Set2.intersection(s1,s2)).get();
```

ToList: $L_{i=0|p(i)}^{n-1} f(i)$:

```
IntStream.range(0,n).boxed().filter(p)
       map(f).toList();
```

ToSet: $S_{i=0|p(i)}^{n-1} f(i)$:

```
IntStream.range(0,n).boxed().filter(p)
       map(f).collect(Collectors.toSet());
```

ToMultiset: $MS_{i=0|p(i)}^{n-1} f(i)$:

```
Multiset.of(IntStream.range(0,n).boxed().filter(p).
       map(f).collect(Collectors.toList()));
```

Grouping: $G_{i=0|key}^{n-1} f(i)$: Forma grupos a partir de $f(i)$ con la misma clave *key(f(i))*. Si los tipos de las funciones *f, key* son respectivamente $Integer \rightarrow E$ y $E \rightarrow K$ el resultado es de tipo *Map<K,List<E>>*

```
IntStream.range(0,n).boxed().map(f)
      .collect(Collectors.groupingBy(key));
```

Grouping y transformación: $GT_{i=0|key,t}^{n-1} f(i)$: Forma grupos a partir de $f(i)$ con la misma clave *key(f(i)) y* los transforma con la función *t.* Si los tipos de las funciones *f, key,t* son respectivamente $Integer \rightarrow E$, $E \rightarrow K$ y $E \rightarrow T$ el resultado es de tipo *Map<K,List<T>>.*

```
IntStream.range(0, n).boxed().map(f)
      .collect(Collectors.groupingBy(key,
            Collectors.mappping(f,Collectors.toList())));));
```

Grouping, transformación y reducción: $GR_{i=0|key,t,bo}^{n-1} f(i)$: Forma grupos a partir de $f(i)$ con la misma clave *key(f(i)).* Transforma cada grupo *e* del grupo con la función t(e) y finalmente agrupa los elementos de cada grupo según el operador binario *bo*. Si los tipos de las funciones *f, key,t,bo* son respectivamente $Integer \rightarrow E$, $E \rightarrow K$, $E \rightarrow R$ y $R \times R \rightarrow R$ el resultado es de tipo *Map<K,R>.*

```
IntStream.range(0, n).boxed().map(f)
      .collect(Collectors.groupingBy(key,
            Collectors.collectingAndThen(
                  Collectors.mapping(t,
                        Collectors.reducing(bo)),
                              r -> r.get())));
```

Contador: $C_{i=0|p(i)}^{n-1} f(i)$: Cuenta el número de elementos que cumplen el predicado

```
IntStream.range(0,n).boxed().filter(i->p(i)).count();
```

Contador de diferentes: $CD_{i=0|p(i)}^{n-1} f(i)$:

```
IntStream.range(0,n).boxed().filter(i->p(i))
.map(i->f(i)).distinct().count();
```

Restricciones

Las restricciones son funciones que devuelven un valor de tipo lógico tras operar sus resultados.

Las restricciones se pueden combinar para formar restricciones más complejas con los operadores lógicos $\wedge$, $\vee$, ! (*and, or* y *not* respectivamente), y los operadores: $\Rightarrow, \Leftrightarrow$ (de implicación y doble implicación respectivamente. En muchos casos una simple coma sustituirá al operador $\wedge$.

Con el símbolo $\equiv$ indicaremos que dos restricciones son equivalentes.

La *restricciones relacionales* más conocidas son $>, \geq, =, <, \leq$ que denominaremos, respectivamente *GT,GE,EQ,LT,LE:* mayor que, mayor o igual, igual, menor que, menor o igual. Incluimos aquí también las de inclusión de conjuntos y multiconjuntos: $\subseteq, \subset, \supset, \supseteq$ y los de pertenencia $\in$. Un multiconjuntos está incluido en otro cuando el número de repeticiones de cada elemento en el primero es menor o igual que en el segundo.

Otras restricciones muy usadas son las siguientes:

AllMatch: $\forall_{i=0}^{n-1} f(i)$: La restricción obliga a que todos los $f(i)$ son verdaderos.

```
IntStream.range(0,n).boxed().map(f).allMatch(f);
```

AnyMatch: $\exists_{i=0}^{n-1} f(i)$: La restricción obliga a que alguno de los $f(i)$ es verdadero.

```
IntStream.range(0,n).boxed().map(f).anyMatch(f)
```

NoneMatch: $\nexists_{i=0}^{n-1} f(i)$*:* La restricción obliga a que ninguno de los $f(i)$ es verdadero.

```
IntStream.range(0,n).boxed().map(f).noneMatch(f);
```

ValueIn: $I_{i=0}^{n-1}(x, f(i))$: Es una restricción que obliga a que la variable x tome uno de los valores de la lista $\mathrm{L}_{i=0}^{n-1} f(i)$.

$$I_{i=0}^{n-1}(x, f(i)) \equiv \exists_{i=0}^{n-1}(x = f(i))$$

AllDifferents: $\mathrm{AD}_{i=0|p(i)}^{n-1} f(i)$: Es una restricción que toma una lista de valores enteros filtrada por un predicado y los obliga a que sean distintos.

$$AD_{i=0|p(i)}^{n-1} f(i) \equiv |L_{i=0|p(i)}^{n-1} f(i)| = |S_{i=0|p(i)}^{n-1} f(i)|$$

```
Integer n1 = IntStream.range(0, n).boxed().filter(pd)
             .map(f).collect(Collectors.toSet()).size();
Integer n2 = (int)IntStream
             .range(0, n).boxed().filter(pd).count();
return n1.equals(n2);
```

Permutación: $P_{i=0,j=0}^{n-1,m-1}(f(i), g(j))$. Es una restricción que obliga a la lista de $L_{i=0}^{n-1} f(i)$, de *n* elementos, que sea una permutación de un prefijo *de* la lista $L_{i=0}^{m-1} v(i)$, de *m* elementos, con *n <=m*.

$$P_{i=0,j=0}^{n-1,m-1}(f(i), g(j)) \equiv MS_{i=0}^{n-1} f(i) \subseteq MS_{j=0}^{m-1} g(j)$$

Esta restricción es equivalente a la combinación de alguna de las anteriores:

$$P_{i=0}^{n-1}(f(i), i) \equiv AD_{i=0}^{n-1} f(i), \forall_{i=0}^{n-1} 0 \leq f(i) < n$$

Aqui la yuxtaposición de restricciones separadas por como indica combinación con el operador and.

Open Path: $OP_{i=0|g}^{n-1} f(i)$: Los valores de $f(i)$ son los vértices de un grafo g y forman un camino abierto en el mismo. Esta restricción puede ser expresada en función de las anteriores en la forma:

$$OP_{i=0|g}^{n-1} f(i) \equiv \mathrm{AD}_{i=0}^{n-1} f(i),\ \forall_{i=0}^{n-1} f(i) \in g_v,\ \forall_{i=0}^{n-2}(f(i), \mathrm{f(i+1)}) \in g_e$$

Si podemos afirmar con otras restricciones que los valores $f(i)$ son vértices del grafo entonces podemos simplificar lo anterior a:

$$OP_{i=0|g}^{n-1} f(i) \equiv \mathrm{AD}_{i=0}^{n-1} f(i),\ \forall_{i=0}^{n-2}(f(i), \mathrm{f(i+1)}) \in g_e$$

Closed Path: $CP_{i=0|g}^{n-1} f(i)$. Los valores de f(i) son los vértices de un grafo g y forman un camino cerrado en el mismo. Esta restricción puede ser expresada en función de las anteriores en la forma:

$$CP_{i=0|g}^{n-1} f(i) \equiv AD_{i=0}^{n-1} f(i),\ \forall_{i=0}^{n-1} f(i) \in g_v,\ \forall_{i=0}^{n-1} (f(i), f((i+1)\%n)) \in g_e$$

Si podemos afirmar con otras restricciones que los valores $f(i)$ son vértices del grafo entonces podemos simplificar lo anterior a:

$$CP_{i=0|g}^{n-1} f(i) \equiv AD_{i=0}^{n-1} f(i),\ \forall_{i=0}^{n-1} (f(i), f((i+1)\%n)) \in g_e$$

Distancias a restricciones

La distancia a una restricción es una función que devuelve un entero o real tomando los mismos argumentos que la restricción. Devuelve un cero si la restricción se cumple y un valor positivo si o se cumple. La idea general es convertir una restricción como $c(x)$ en una función numérica $dc(x)$ que llamaremos *función de distancia*, que cumpla $dc(x) = 0$ cuando se cumpla la restricción y $dc(x) \geq 0$ cuando no se cumpla. Es decir:

$$r = dc(x) = \begin{cases} r = 0, & c(x) \\ r > 0, & !\, c(x) \end{cases}$$

Veamos algunas distancias a restricciones que usaremos.

Distancia a las restricciones relacionales

$$x \geq 0,\ \ dGe(x) = \begin{cases} 0, & x \geq 0 \\ x^2, & x < 0 \end{cases}$$

$$x \leq 0,\ \ dLe(x) = \begin{cases} 0, & x \leq 0 \\ x^2, & x > 0 \end{cases}$$

$$x = 0,\ \ dEq(x) = \begin{cases} 0, & x = 0 \\ x^2, & x \neq 0 \end{cases}$$

El código se muestra abajo junto a las distancias a otras restricciones más complejas. Algunas de ellas son:

```
public Double dGe(Double in) {
	Double r = 0.;
	if(in < 0) {
		r = in*in;
	}
	return r;
}
```

```
public Double dLe (Double in) {
	Double r = 0.;
	if(in > 0) {
		r = in*in;
	}
	return r;
}
```

```
public Double dEq (Double in) {
	Double r = 0.;
	if(in != 0) {
		r = in*in;
	}
	return r;
}
```

Distancia a la Igualdad entre dos conjuntos: *dEqS*

$$dEqS_{i=0}^{n-1}(f(i), g(i)) = (|S_{i=0}^{n-1}f(i) - S_{i=0}^{n-1}g(i))|)^2$$

Distancia a la Igualdad entre dos multiconjuntos: *dEqMs*

$$dEqMs_{i=0}^{n-1}(f(i), g(i)) = (||Ms_{i=0}^{n-1}f(i) - Ms_{i=0}^{n-1}g(i))||)^2$$

Distancia a AllMatch: *dAl*

$$dAl_{i=0}^{n-1}f(i) = (\sum_{i=0|!f(i)}^{n-1} 1)^2$$

Distancia a AnyMatch: *dAn*

$$dAn_{i=0}^{n-1}f(i) = (\exists_{i=0}^{n-1}f(i)?\,0:1)^2$$

Distancia a NoneMatch: *dNm*

$$dNm_{i=0}^{n-1} f(i) = (\sum_{i=0|f(i)}^{n-1} 1)^2$$

Distancia a AllDiferents: *dAd*

$$dAd_{i=0|p(i)}^{n-1} f(i) = (|L_{i=0|p(i)}^{n-1} f(i)| - |S_{i=0|p(i)}^{n-1} f(i)|)^2$$

```
public <E> Double distanceToAllDifferents(List<E> ls) {
	Integer n = ls.size();
	Integer m =
	ls.stream().collect(Collectors.toSet()).size();
	return (double)(n-m)*(n-m);
}
```

Distancia a Permutación: *dP*

$$dP_{i=0,j=0}^{n-1,m-1}(f(i), g(j) = (||Ms_{i=0}^{n-1}(f(i) - Ms_{j=0}^{m-1} g(j)||)^2$$

Distancia a Open Path: *dOp*

$$dOp_{i=0|g}^{n-1} f(i) =$$

$$dAd_{i=0}^{n-1} f(i) + dAl_{i=0}^{n-1} f(i) \in g_v + dAl_{i=0}^{n-2}(f(i), f(i+1)) \in g_e$$

Si podemos afirmar con otras restricciones que los valores $f(i)$ son vértices del grafo entonces podemos simplificar lo anterior a:

$$dAd_{i=0}^{n-1} f(i) + dAl_{i=0}^{n-2}(f(i), f(i+1)) \in g_e$$

Distancia a Closed Path: *dCp*

$$dCp_{i=0|g}^{n-1} f(i) =$$

$$dAd_{i=0}^{n-1} f(i) + dAl_{i=0}^{n-1} f(i) \in g_v + dAl_{i=0}^{n-1}(f(i), f((i+1)\%n) \in g_e$$

Si podemos afirmar con otras restricciones que los valores $f(i)$ son vértices del grafo entonces podemos simplificar lo anterior a:

$$dCp_{i=0|g}^{n-1} f(i) =$$

$$dAd_{i=0}^{n-1} f(i) + dAl_{i=0}^{n-1}(f(i), f((i+1)\%n) \in g_e$$

Bibliografía

Para trabajar con grafos usaremos la librería JgraphT que no proporciona implementaciones en Java de distintos tipos de grafos y una abundante colección de algoritmos sobre los mismos que podemos encontrar en el API.

El algoritmo Simplex se refiere a un conjunto de métodos usados para resolver problemas de programación lineal, en los cuales se busca el valor óptimo de una función lineal sobre un conjunto de variables que satisfaga un conjunto de inecuaciones lineales.

Gurobi es una herramienta informática que implementa algoritmos para resolver problemas de Programación Lineal Entera Extendida.

Gurobi acepta ficheros en el formato LP. Este formato es aceptado por otras herramientas informática que resuelven problemas de restricciones. Aquí se pueden ver otros tipos de restricciones no incluidas aquí como restricciones cuadráticas. También una posible clasificación de las restricciones en *lazy* o no, según la prioridad que queramos asignarle.

Graphviz es una herramienta muy adecuada para visualizar grafos. Acepta ficheros en formato *dot* o *gv* que puede ser generados por JgraphT. También existe una herramienta online para conseguir ese objetivo.

Libro de introducción a Java: Fundamentos de programación: JAVA en la Colección de Manuales del I3US

Libro de introducción a Python: Fundamentos de programación: PYTHON en la Colección de Manuales del I3US

Análisis y Tipos de Datos en C, autor Miguel Toro, por aparecer en la Colección de Manuales del I3US.

Análisis y Diseño de Algoritmos y Tipos de Datos, autor Miguel Toro, por aparecer en la Colección de Manuales del I3US.

Una introducción a la Programación Lineal Entera, algunos algoritmos para resolver problemas modelados con este paradigma y la complejidad de los mismos se puede encontrar en Wikipedia aqui.

Una discusión sobre las posibilidades de implementar el valor absoluto en Programación Lineal Entera puede encontrarse aquí y aqui.

Un libro muy recomendable para ampliar conocimientos de algoritmos en general y en concreto de DPR, es Introduction to Algorithms de T. H. Cormen entre otros autores.

Para ampliar conocimientos sobre Programación Dinámica se puede consultar libro Dynamic Programming de Bellman.

Otros detalles, bibliografía y variantes del algoritmo A* se puede encontrar en Wikipedia.

Una visión general sobre el campo de los Algoritmos Genéticos , sus limitaciones y aplicaciones puede encontrase aquí y en Wikipedia aqui y aquí.

La librería de Apache de Genéticos la hemos usado como base de las implementación de algoritmos genéticos que usamos en el repositorio.

La librería de Apache la hemos usado también para disponer de tipos como *Complex, Fraction*, etc. y los tipos para resolver problemas de algebra lineal como matrices, etc.

Para diseñar el lenguaje *LSI,* hemos usado Antlr4 que nos permite escribir la gramática de un lenguaje y generar un reconocer del mismo, más un conjunto de métodos para visitar cada uno de los vértices del árbol de sintaxis abstracta correspondiente.

Para consultar otros tipos de cromosomas, pueden verse las ideas de Gene expression programming.

Miguel Toro es doctor en Ingeniero Industrial por la Universidad de Sevilla y catedrático del Departamento de Lenguajes y Sistemas Informáticos de la misma universidad.

Ha sido director de la Oficina de Transferencia de Resultados de la Investigación (OTRI) y Director General de Investigación, Tecnología y Empresa de la Junta de Andalucía.

Ha tenido un papel activo en la Agencia Andaluza de Evaluación de la Calidad y Acreditación Universitaria (AGAE), ha sido miembro del Consejo Asesor de la Agencia Nacional de Evaluación de la Calidad y Acreditación (ANECA) y colabora asiduamente con varias agencias nacionales de evaluación universitaria.

Ha sido el Presidente de Sistedes (Sociedad Nacional de Ingeniería del Software y Tecnologías de Desarrollo de Software) y el Presidente de la Sociedad Científica Informática de España (SCIE) que engloba a los informáticos de las universidades españolas.

Ha recibido el Premio Fama de la Universidad de Sevilla, el Premio Sistedes de la Sociedad Nacional de Ingeniería del Software y Tecnologías de Desarrollo de Software en reconocimiento a su labor de promoción y consolidación de la Informática en España.

Ha recibido el Premio Nacional de Informática José García Santesmases a la trayectoria profesional otorgado por la Sociedad Científica Informática de España.

Actualmente es director del Instituto de Ingeniería Informática de la Universidad de Sevilla

Este libro está especialmente orientado a la enseñanza de la asignatura Análisis y Diseño de Datos y Algoritmos, que es la continuación natural de Fundamentos de Programación. Este volumen de *Problemas, modelos, grafos y algoritmos* sigue al de *Análisis y diseño de algoritmos y tipos de datos*, también publicado en esta colección. El lector interesado puede consultar, además, *Fundamentos de programación: Python* y *Fundamentos de programación: Java*. Para abordar el diseño de algoritmos, es necesario tener asimilados los elementos de la programación en algún lenguaje. El dominio de Java y sus peculiaridades se hace imprescindible para acometer el seguimiento del contenido. Tras las técnicas revisadas en el volumen anterior, *Análisis y Diseño de Algoritmos y Tipos de Datos*, sobre diseño de algoritmos iterativos y recursivos, abordamos ahora un conjunto de técnicas algorítmicas de uso general: Programación Lineal Entera, Algoritmos Genéticos, Algoritmos A*, Backtracking, Programación Dinámica y otras. Al final del manual, se incluye una serie de ejemplos, muchos de ellos resueltos a partir del uso de las técnicas descritas. Este texto es el resultado de la experiencia de varios años de enseñanza de la asignatura Análisis y Diseño de Algoritmos en la Universidad de Sevilla.

Editorial Universidad de Sevilla