Introducción a la Programación

Tema 14. Programación Dinámica

1. Introducción

La **Programación Dinámica** tiene distintas acepciones en la literatura. Aquí llamaremos **Programación Dinámica** a una generalización de la técnica de *Divide y Vencerás* (posiblemente en su versión de *Reducción*) con o sin memoria. Esencialmente la generalización consiste en considerar, para dividir un problema en subproblemas, un conjunto de **alternativas** u **opciones**. Para dividir un problema en subproblemas se considera una de las alternativas, se divide el problema en subproblemas, se resuelven, se combinan las soluciones y luego se combinan las soluciones obtenidas a partir de cada alternativa. Aparecen nuevos elementos con respecto a la técnica de *Divide y Vencerás*: alternativas y combinación de las soluciones obtenidas tras escoger las diferentes alternativas.

Suponemos que las alternativas están descritas por un conjunto finito que denominaremos A. Por $a, a_1, a_2, \dots$ representaremos valores concretos de las alternativas. El conjunto de alternativas disponibles puede depender del problema en cuestión. Cuando hay dudas el conjunto de alternativas disponibles para resolver un problema X lo representaremos como A_X y sin subíndice cuando no haya dudas sobre el conjunto en cuestión. Para resolver un problema dado X, en la técnica de la *Programación Dinámica,* tomamos una de las alternativas y posteriormente dividimos el problema en subproblemas. Tras elegir una de las alternativas del conjunto A_X y el problema se divide en los subproblemas $X_1^a, X_2^a, \dots, X_k^a$. En general el número de subproblemas depende de la alternativa escogida.

Si *k = 1* (sólo un subproblema), como en el caso de *Divide y Vencerás*, denominaremos a la técnica *Programación Dinámica con Reducción*.

1.1 Algoritmo de la Programación Dinámica

Veamos el esquema general de un problema de *Programación Dinámica*. Como vimos anteriormente un problema, que se resuelve por la técnica de *Divide Y Vencerás*, adopta la forma:

$$f(X) = \begin{cases} S_0, & si\ es\ un\ caso\ base \\ c(X, f(X_1), f(X_2), \dots, f(X_k)) & si\ es\ un\ caso\ recursivo \end{cases}$$

Donde *X* representa el problema de tamaño *n*, X_i los subproblemas de tamaño n_i con $n_i < n$, c la función que combina las soluciones de los subproblemas (en el contexto del problema a resolver) y s_b la solución del caso base de los que puede haber más de uno. Representaremos este por $\perp$ la no existencia de solución. En la técnica de *Divide y Vencerás* si uno de los problemas no tiene solución (solución $\perp$) entonces el problema completo tampoco la tiene.

El esquema de un problema que se resuelve por Programación Dinámica comparte elementos con el anterior pero es más general. Su forma es:

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

Donde, como vimos antes, $X_1^a, X_2^a, \dots, X_k^a$ son los sub-problemas en los que se puede dividir X tras optar por la alternativa a del conjunto A_X. Asumimos que los casos base no tienen ningún subproblema y tampoco tienen subproblemas aquellos problemas para los que $A_X = \emptyset$.

Cada problema tiene asociado un tamaño $n(X)$ y los tamaños de los subproblema deben ser menores que el tamaño del problema original: $n(X_l^a) < n(X)$ con $l \in [1, k], a \in A_X$.

Siendo $s_i^a = f(X_i^a), i \in [1, k]$ las soluciones de los subproblemas la operación $s^a = c(X^a, s_1^a, \dots, s_k^a)$ construye la solución del problema asumiendo que se ha escogido la alternativa a. La operación $s = sA_{a\in A_X}(X, s^a)$ construye la solución del problema original X a partir de las soluciones obtenidas para cada alternativa. A la operación *c* la denominaremos *combinaSoluciones* y a la operación *cA seleccionaAlternativa.* A partir de X, a se pueden obtener los subproblemas $X_1^a, X_2^a, \dots, X_k^a$.

Las operaciones *c* y *sA* tienen distintos comportamientos con respecto a los subproblemas que no tienen solución. El operador *c* devuelve el valor ⊥ cuando alguno de los subproblemas no tiene solución: $\perp = c(X^a, s_1^a, \dots, \perp, \dots, s_k^a)$. La operación *sA* devuelve ⊥ cuando no existe solución para ninguna alternativa. El operador *cA* verifica:

$$s = sA_{a\in A_X, s^a\neq\perp}(X, s^a), \qquad \perp = sA_{a\in A_X}(X, \perp), \qquad \perp = sA_{a\in A_X=\emptyset}(X, s^a)$$

Donde hemos denotado por s^a la solución que se obtiene para el problema si escogemos la alternativa a. Vemos que si $A_X = \emptyset$, es decir no hay alternativas, entonces las solución es ⊥, es decir no hay solución. Esta idea estará implícita (aunque en algunos caso la explicitaremos) en los problemas posteriores.

Como hemos comentado antes el conjunto de alternativas A_X depende del problema X que estamos considerando. En cada problema concreto habrá que detallar el cálculo de este conjunto a partir de las propiedades del problema.

1.2 Programación Dinámica con Memoria y solución de un conjunto de problemas

La *Programación Dinámica* puede usar memoria como en la técnica de *Divide y Vencerás con Memoria*. En general, si no decimos lo contrario, asumimos que la Programación Dinámica es con uso de Memoria. El esquema que hace explícito este uso de la memoria es:

$$fmg(X, m) = \begin{cases} m, & si\ X \in m \\ m + (X, s_b), & si\ es\ un\ caso\ base \\ m + (X, \perp), & si\ A_X = \emptyset \\ m + (X, s), & si\ es\ un\ caso\ recursivo \end{cases}$$

Donde $s_i^a = fmg(X_i^a, m), i = 1, \dots, k$

$$s = sA_{a\in A_X}(c(X, a, s_1^a, s_2^a, .., s_k^a))$$

Como vemos el problema generalizado anterior devuelve no sólo la solución al problema planteado. También devuelve las soluciones a todos los subproblemas que se han tenido que resolver. Esto nos permite aprovechar los posibles subproblemas compartidos y resolver un conjunto de problemas en general. Un **conjunto de problemas** lo podemos caracterizar por un rango de valores de una o varias propiedades de un problema. Por lo tanto especificar un conjunto de problemas consiste, desde el punto de vista que nos interesa aquí, en dar un problema y unos rangos donde se moverán los valores de alguna de sus variables. Un problema lo representaremos por (SX, n). El conjunto de problemas estará formado por una secuencia de problemas X_i cada uno de los cuales está indexado por un índice $i = 1, .. n$.

Usando el esquema anterior la solución de un conjunto de problemas $fc(SX, n)$ (y de los todos los subproblemas que es necesario resolver) mediante *Programación Dinámica con Memoria* es de la forma:

$$fcg(X, i, n, m) = \begin{cases} fcg\big(X, i+1, n, fmg(X_{i+1}, m)\big), & i+1 < n \\ fmg(X_{i+1}, m), & i+1 = n \end{cases}$$

$$fc(X, n) = fcg(X, 0, n, \emptyset)$$

La solución de un problema concreto X_i del conjunto de problemas podemos obtenerla buscando en el conjunto de pares problema-solución devuelto. Es decir

$$f(X_i) = fc(X, n)(X_i)$$

Si solo tenemos un problema de partida entonces su solución es directamente:

$$f(X) = fgm(X, \emptyset)(X)$$

Como vemos arriba la *Programación Dinámica con Memoria* (de un problema o un conjunto de problemas) generaliza el problema añadiendo una propiedad que mantiene en una memoria las soluciones de los problemas ya resueltos. Resolver un conjunto de problemas es resolver consecutivamente uno tras otro usando la memoria devuelta por los que han sido resueltos previamente. Como vemos siempre aparece un nuevo caso base que ocurre si el problema considerado pertenece al dominio de la memoria. Es decir a los problemas ya resueltos. Como hemos comentado arriba si el conjunto de las alternativas está vacío el problema no tiene solución. Por defecto asumimos que la *Programación Dinámica* siempre usa memoria aunque puede haber casos concretos en que no sea así. En las secciones siguientes hablaremos sólo de *Programación Dinámica* independientemente de que se haga con memoria o sin ella (aunque por defecto asumiremos que es con memoria).

2. Problemas y Grafos

Veamos ahora las relaciones que existen entre la técnica de la **Programación Dinámica** y los grafos. Tal como hemos visto anteriormente la **Programación Dinámica** generaliza a **Divide y Vencerás** (con o sin memoria).

Al resolver un problema aparecen un conjunto de problemas dónde el que queremos, o lo que queremos resolver, están incluidos. Cada problema en ese conjunto tiene diversas propiedades: unas comunes a todos los problemas del conjunto y otras individuales a cada uno de ellos. Así cada problema, dentro del contexto del conjunto de problemas considerado puede identificarse por un conjunto de valores para las propiedades individuales consideradas. Asociado a cada problema X aparecen un conjunto de alternativas, A_X que depende del problema. Tal como vimos el esquema de la Programación Dinámica es de la forma:

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

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

El conjunto de problemas más las relaciones entre problemas y subproblemas definen implícitamente un grafo dirigido. Este grafo es un grafo bipartito. Es decir un grafo con dos tipos de vértices: **vértices problemas** y **vértices alternativas**. Cada *vértice problema* contiene un problema del conjunto de problemas. Cada *vértice alternativa* contiene un problema más una alternativa escogida de entre su conjunto de alternativas. Cada problema del conjunto de problemas lo representamos por $X, X_1, \dots, X_r$. Las alternativas para un problema dado las representaremos por (X, a). Los subproblemas para una alternativa dada los representamos por X_i^a. Cada problema se asocia a un vértice y hay una arista entre los vértices X y (X, a) si hay una alternativa $a \in A_X$ y entre (X, a) y X_i^a para cada entero *i* que represente un subproblema de X tomando la alternativa a. Cada arista entre los vértices $X, (X, a)$ la representamos de la forma $X \rightarrow_a (X, a)$ y cada arista entre los vértices $(X, a), X_i^a$ la representamos de la forma $(X, a) \rightarrow_i X_i^a$.

En estos grafos las hojas de este son casos base o problemas cuyo conjunto de alternativas es vacío.

En estos grafos cada vértice tiene asociado un operador que es capaz de calcular la solución de un problema a partir de las de los subproblemas. Los vértices *OR* (problemas) tienen asociado el operador *sA(…).* Este operador calcula la solución del problema entre las diferentes soluciones calculadas para cada una de las alternativas. Los vértices AND tienen asociado el operador *c(…)* que obtiene la solución de un problema, tras tomar una alternativa, combinando las soluciones de los subproblemas.

2.1 Caracterización de las soluciones de un problema de Programación Dinámica

Los problemas de un conjunto de problemas que se quieren resolver mediante *Programación Dinámica* se organizan en un grafo dirigido y bipartito. Los problemas y sus alternativas constituyen los vértices del grafo. De cada vértice problema (cada problema del conjunto de problemas) parten un conjunto de aristas dirigidas que conducen a los correspondientes vértices alternativas. De cada vértice alternativa parten aristas que conducen a los subproblemas alcanzados

Las soluciones, si las hay, a cada problema del conjunto de problemas representado en el grafo pueden ser caracterizadas por un subgrafo del grafo anterior. Este subgrafo tiene como vértice maximal al problema cuya solución queremos caracterizar, todas las hojas son casos base, de cada vértice problema partes aristas que conducen a problemas con solución. Si no es posible obtener un subgrafo con las características anteriores el problema no tiene solución.

Dado un subgrafo que representa una solución es posible calcularla recorriendo el grafo de abajo arriba. Partimos de las soluciones de los casos base incluidos en la solución y vamos aplicando los operadores de combinación c(…) a las soluciones de los subproblemas para ir obteniendo las soluciones de los problemas.

Un problema puede tener muchas soluciones cada una de las cuales vendrá representada por un subgrafo. Si el problema es de optimización entonces se trata de escoger la mejor de todas ellas. Este trabajo lo va haciendo el operador *sA(…)* al escoger para cada problema la alternativa que proporciona la mejor solución.

En cada caso la **solución** buscada será de un tipo determinado que habrá que especificar. Si el problema es de optimización, para caracterizarlo, habrá que indicar, además, una **propiedad de la solución a optimizar**. En cualquier caso para identificar de forma única la solución es suficiente con indicar cuál es la alternativa elegida para cada problema. Por lo que hemos comentado arriba conocidas las alternativas elegidas para cada problema conocemos el subgrafo que define la solución y por lo tanto el valor de la misma. Por cuestiones de eficiencia, en general, es preferible primero buscar el subgrafo que define la solución recordando (en la memoria del algoritmo) la alternativa escogida y el valor de la propiedad de la solución a optimizar. Posteriormente reconstruir el valor de la solución pedida. Al par formado por la alternativa elegida, valor de la propiedad relevante de la solución lo llamaremos **solución parcial**. La soluciones parciales las representaremos por pares del tipo (a, p). Dadas las soluciones parcial para un problema y los subproblemas involucrados es posible reconstruir la solución. Para cada problema debemos indicar la forma de reconstruir la solución a partir de la solución parcial.

Veamos un ejemplo para fijar las ideas anteriores.

2.2 Problema de la Mochila

El problema se enuncia así:

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

Para definir los problemas del conjunto de problemas trataremos cada problema como un objeto (independientemente de su implementación final) y diseñaremos un tipo cuyas instancias serán los problemas. Igual que en el diseño de tipos tendremos, para los problemas, propiedades básicas y derivadas, compartidas e individuales, etc.

Partimos de un conjunto de objetos disponibles para almacenar en la mochila que representamos como una propiedad compartida del conjunto de problemas. Es la propiedad *od, List<ObjetoMochila>* ordenada y sin repetición, consultable. Cada objeto, de los disponibles, tendrá las propiedades (v, p, m, q) . Estas propiedades representan, respectivamente el valor unitario, el peso unitario, el número máximo de unidades permitido y el valor por unidad de peso. La última propiedad es derivada y se calcula $q = \frac{v}{p}$.

La lista *OD* se mantiene ordenada de menor a mayor valor por la propiedad *q*. Es decir valor por unidad de peso de cada uno de los objetos. La razón para este orden la veremos más adelante.

Cada objeto de la lista anterior viene identificado pos su posición i en la misma y lo representamos por $o_i, i \in [0, r-1]$. Siendo *r* el número de objetos disponibles diferentes. Sea v_i el valor (entero), p_i el peso (entero), m_i el número máximo de unidades del objeto *i* (entero) y q_i el cociente (real) entre el valor y el peso. Es decir $q_i = \frac{v_i}{p_i}$. A partir de esas propiedades podemos modelar el tipo *ObjetoMochila*.

El tipo *SolucionMochila* es un subtipo de *MultiSet<ObjetosEnMochila>* con las propiedades adicionales *ValorTotal* (entero, consultable) de los objetos almacenados en la mochila y *PesoTotal* (entero, consultable) de los objetos en la mochila. Una solución la representamos por lo tanto por (ms, vt, pt). Donde ms es el multiconjunto de objetos en la mochila, vt el peso total y pt el peso total. Como solución parcial escogemos el par formado por la alternativa y el valor total de la solución.

Como en la mayoría de los casos de problemas resueltos mediante *Programación Dinámica* generalizamos el problema. Una forma de generalizarlo es resolver el problema para una *capacidad* de la mochila (que representaremos por *C*) usando solamente un subconjunto de los objetos disponibles. El subconjunto de objetos los representamos por un índice entero *J* que representa el subconjunto de objetos disponibles cuyos índices son menores o iguales a *J* dentro de la lista *OD*.

El problema generalizado tiene las propiedades individuales: *C (*entero mayor o igual a cero, consultable*), J (*entero en *[0,r)*, consultable). Dónde r es el número de objetos disponibles distintos. En este contexto dos problemas son iguales si tienen iguales sus propiedades individuales: *C, J.*

Las alternativas disponibles podemos representarlas con números enteros: El número de unidades que almacenaremos del objeto o_j si la capacidad de la mochila es *C*. Las alternativas disponibles están en el conjunto $A_{c,j} = \{0,1,2,\dots,k\}$ con $k = \min\left(\frac{c}{p_j}, m_j\right)$. Donde la división es entera.

El objetivo del problema es encontrar una solución cuya propiedad *PesoTotal* sea menor o igual a la *capacidad C* de la mochila y tenga el mayor valor posible para la propiedad *ValorTotal*.

Con esos elementos de diseño para acabar de concretar el algoritmo tenemos que dar los detalles de las funciones de combinación y de combinación de alternativas. Si buscamos la solución directamente Estos son:

$$c((c, j, od), a, s) = s + (od(j), a)$$

Donde por $s + (od(j), a)$ indicamos añadir a unidades del objeto en la posición j de od al multiconjunto s. Alternativamente podemos considerar en primer lugar la solución parcial (par formado por la alternativa elegida y el valor total de la solución):

$$c((c, j, od), a, vt) = (a, vt + a * v_j)$$

El operador para combinar las alternativas es (en las dos versiones anteriores):

$$sA_{a \in A_X}(s^a) = \max_{a \in A_X} s^a$$

Donde s^a es el multiconjunto solución del subproblema resultante cuando escogemos la alternativa a. Las soluciones se ordenan de mayor a menor según su propiedad valor total.

$$sA_{a \in A_X}((a, vt)) = \max_{a \in A_X}((a, vt))$$

Ahora el máximo se toma según el valor de vt.

Los casos base serían problemas del tipo $(c, 0, od)$. La solución directa del caso base es:

$$\emptyset + (od(0), a_b)$$

La solución parcial:

$$(a_b, a_b * v_0)$$

Otros casos bases posibles son los de la forma $(0, j, od)$ cuya solución parcial es $(0,0)$.

Donde $a_b = \min(\frac{c}{p_0}, m_0)$ y $v_0, p_0 m_0$ representan, respectivamente el valor unitario, el peso unitario y el número máximo de unidades del objeto que ocupa la posición 0 en los objetos disponibles od.

Para cada alternativa hay un solo subproblema que bien dado por:

$$(c, j, od)_1^a = (c - a * p_j, j - 1, od)$$

A partir de las soluciones parciales la forma de reconstruir la solución bien dada por la función:

$$rc(X, m) = \begin{cases} \perp, & m_a(X) = \perp \\ \emptyset + (od(0), m_a(X)), & si\ es\ caso\ base \\ rc\left(X_1^{m_a(X)}, m\right) + (od(j), m_a(X)), & si\ es\ caso\ recursivo \end{cases}$$

Donde asumimos que tenemos disponible la memoria que guarda para cada problema X la información asociada a la solución parcial. La más importante para reconstruir la solución es alternativa escogida en la mejor solución que representaremos por $m_a(X)$ (este valor asumimos que será $\perp$ si el problema no tiene solución). Además designamos por X_1^a el subproblema de X cuando se escoge la alternativa $a = m_a(X)$. La memoria guardada otra información: el valor de la propiedad a optimizar que representaremos por $m_v(X)$.

Todas las ideas anteriores las podemos reunir en una ficha para el problema.

<table>
<tr><th colspan="2">Problema de la Mochila</th></tr>
<tr><td colspan="2">Técnica: Programación Dinámica</td></tr>
<tr><td colspan="2">Tamaño: J</td></tr>
<tr><td>Propiedades Compartidas</td><td>OD, List <ObjetoMochila>, ordenada de mayor a menor por el valor unitario.
N, OD.size()</td></tr>
<tr><td>Propiedades Individuales</td><td>C, entero no negativo
J, entero en [0,ObjetosDisponibles.Size)</td></tr>
<tr><td colspan="2">Solución: $s = (ms, vt, pt)$</td></tr>
<tr><td colspan="2">Objetivo: Encontrar s tal que $pt \leq c$ y vt tenga el mayor valor posible</td></tr>
<tr><td colspan="2">Solucion Parcial: (a,vt)</td></tr>
<tr><td colspan="2">Alternativas: $A_{c,j,od} = \{a: k..0\}, k = \min\left(\frac{c}{p_j}, m_j\right)$</td></tr>
<tr><td colspan="2">Instanciación

$pm(c, od) = pmg(c, 0, od), \quad r = od.size()$</td></tr>
<tr><td colspan="2">Problema Generalizado

$pmg(p) = \begin{cases} (0,0), & c = 0 \\ (n, nv_j), & n = \min\left(\frac{c}{p_0}, m_0\right), j = N - 1 \\ sA_{a \in A_{c,j,od}}(cS(p, a, pmg(sp))), & en\ otro\ caso \end{cases}$</td></tr>
</table>

$p = (c, j, od)$ $sp = (c - ap_j, j + 1, od)$
$cS\big(p, a, (a', vt)\big) = (a, vt + av_j)$
$sA_{a \in A_{c,j,od}}((a, vt))$: Elige la solución parcial con mayor de *vt*.
Función de reconstrucción $rc(X, m) = \begin{cases} \perp, & m_a(X) = \perp \\ \emptyset + (od(0), m_a(X)), & si\ es\ caso\ base \\ rc\left(X_1^{m_a(X)}, m\right) + (od(j), m_a(X)), & si\ es\ caso\ recursivo \end{cases}$

Complejidad	*r*c*

Una ficha más detallada puede encontrarse en Ficha de Mochila

3. Casos particulares y variantes de la Programación Dinámica

3.1 Caso de Reducción

El caso particular de **Reducción,** en el contexto de **la Programación Dinámica**, se da cuando el número de subproblemas para cada alternativa es igual a uno ($k = 1$). El esquema para este caso particular es:

$$f(X) = \begin{cases} s_b, & si\ es\ un\ caso\ base \\ sA_{a \in A_X}(c\big(X, a, f(X_1^a)\big)), & si\ es\ un\ caso\ recursivo \end{cases}$$

Cada problema X se reduce al problema X_1^a después de tomar la alternativa $a \in A_X$. En el caso de reducción, por haber un solo subproblema para cada problema tras tomar una alternativa, los subgrafos que definen una solución pueden definirse unívocamente por secuencias de alternativas. A partir de un problema cada alternativa nos conduce a un sólo subproblema. Es decir una solución en el caso de reducción es un camino en el grafo que conecta el problema con un caso base. Resolver un problema, desde esta perspectiva, es encontrar un camino, si existe, desde el vértice que representa el problema hasta una hoja que sea un caso base. Encontrar todas las soluciones al problema es encontrar todos los caminos desde el vértice que representa el problema que acaban en casos base. Conocidas todas las soluciones la mejor, con respecto a una propiedad dada de la solución, es aquella que tiene mayor (o menor) valor de esa propiedad entre todas las soluciones encontradas.

En el caso de *reducción*, pues, las soluciones a un problema vienen caracterizadas por **caminos en el grafo** que partiendo de un problema acaben en un caso base. Los **caminos en el grafo** serán secuencias de alternativas que representaremos por $\{a_1, a_2, \dots, a_r\}$. Partiendo de un problema dado X_0 y siguiendo la secuencia de alternativas $\{a_1, a_2, \dots, a_r\}$ alcanzaremos un problema que representaremos por X_r. Desde la perspectiva anterior una solución para el problema X_0 puede representarse como una secuencia de alternativas $\{a_1, a_2, \dots, a_r\}$ con $a_1 \in A_{X_0}$ y X_r un caso base.

Alternativamente una secuencia de alternativas (un camino en el grafo) puede acabar en un problema cuyo conjunto de alternativas esté vacío. Ese camino no define una solución. Un problema (en el caso de *reducción*) para el que no es posible encontrar un camino que lo conecte a un caso base no tiene solución.

Si el problema, además de reducción, es de optimización entonces el valor de la propiedad a optimizar

3.2 Programación Dinámica con Filtro y estrategia Voraz

Usando mecanismos para afinar el cálculo del conjunto A_X podemos obtener un caso particular de la *Programación Dinámica* que denominaremos *Programación Dinámica con Filtro.* Suponemos que el problema de optimización es de minimización. Si fuera de maximización habría que convertirlo en uno de minimización.

Puede usarse para resolver un problema o un conjunto de problemas. La idea consiste en tener disponible un buen valor de la propiedad a optimizar y eliminar del conjunto A_X aquellas alternativas de las que podemos asegurar que tomándolas no alcanzaremos una solución con un valor mejor. Y la forma de llevarlo a cabo es generalizar el problema con dos nuevas propiedades. La primera va es el **valor acumulado** (según las alternativas ya escogidas) para la propiedad que queremos optimizar. La segunda mv es el **mejor valor** obtenido hasta el momento de la propiedad a optimizar. Es una variable compartida por todos los problemas. Disponemos, además, de una función $ct(X, a)$, la función de cota, que para cada problema y alternativa escogida es capaz de calcular una cota inferior para el valor de la propiedad a optimizar de la solución de ese problema si tomáramos esa alternativa.

Como hemos explicado arriba, si estamos en un caso de reducción, partiendo de un problema dado X_0 y siguiendo la secuencia de alternativas $\{a_1, a_2, \dots, a_r\}$ alcanzaremos un problema que representaremos por X_r. El problema generalizado (X_r, va, mv) representa al problema X_r que ha sido alcanzado desde X_0 siguiendo la secuencia de alternativas $\{a_1, a_2, \dots, a_r\}$. El valor acumulado de la propiedad a optimizar después de escoger las alternativas $\{a_1, a_2, \dots, a_r\}$ es va y mv es el mejor valor de esa propiedad a optimizar encontrado por el algoritmo.

Para hacer un tratamiento más homogéneo suponemos que la función de cota $ct(X, a))$ puede aplicarse si X es un caso base. En ese caso asumimos que hay una única alternativa y la función de cota nos proporciona directamente el valor de la propiedad a optimizar.

Por otra parte podemos ver que $va + ct(X, a)$ es una cota inferior para el valor de la propiedad a optimizar del problema original X_0. Luego si siguiendo esa alternativa no se cumple $va + ct(X, a) < mv$ entonces puede ser descartada.

Igualmente si estando en un problema X escogemos la alternativa a entonces se produce un incremento del valor acumulado dado por $incva(X, a)$.

En este esquema es muy importante usar una Estrategia Voraz para alcanzar cuanto antes el mejor valor posible para la variable *mejorValor*. Esto nos permitirá filtrar los más rápidamente

posible. La estrategia voraz la diseñaremos para que se ejecute la primera. Es decir la asociamos a la rama más a la izquierda del grafo de ejecución, la que se sigue en primer lugar. Esta estrategia voraz se define diseñando un **orden de las alternativas** y escogiendo la mejor en cada caso. Escoger la mejor alternativa de las disponibles hasta alcanzar un caso base constituye un algoritmo que denominamos **Voraz**. Es un tipo de algoritmo que puede implementarse de forma iterativa o asociarse a una de las ramas del grafo de ejecución del Algoritmo de Programación Dinámica. Podemos deducir de esto que un algoritmos Voraz puede que no alcance una solución o si la alcanza no sea la óptima pero será un algoritmo rápido y eficiente.

En efecto, como podemos deducir del esquema anterior, cada vez que alcanzamos un caso base podemos reconstruir una solución y por lo tanto el valor de su propiedad a optimizar. Es muy importante, por lo tanto, que las alternativas se ordenen de tal forma que los primeros casos base que se alcancen den lugar a soluciones lo más cercanas posibles a la solución óptima. Esto permitirá que el mecanismo de filtro funcione de la mejor forma posible. El orden de las alternativas es específico para cada problema y, en general, es una información (una heurística) que, cuando se tiene, permite implementar algoritmos mucho más eficientes.

La actualización del mejor valor, $actMv(s)$, se hace vez que obtenemos una solución s para un problema (ya sea un caso base, un caso recursivo o tengamos un problema que ya tiene solución). Eso es posible porque en ese momento encontramos una solución al problema original cuyo valor es el acumulado hasta ese punto más el valor de la propiedad para el problema ya resuelto. Este detalle no parece explícito en el esquema anterior.

El problema original $f(X)$ es una instancia del problema generalizado dando valores iniciales adecuados a las variables va, mv.

$$f(X) = fg(X, va_0);\ mv = mv_0$$

El problema de la mochila visto anteriormente puede resolverse mediante *Programación Dinámica con Filtro.* Una buena función de cota es:

$$cta(c, j, od, a) = av_j + ct\big(c - ap_j, j - 1\big)$$

$$ct(c, j) = \begin{cases} 0, & c = 0 \\ xv_0, & j = 0, x = \min\,(c//p_0, m_0) \\ xv_j + ct\big(c - xp_j, j - 1\big), & j > 0, x = \min\,(c//p_j, m_j) \end{cases}$$

Donde por $c//p_0$ hemos querido indicar que la división hay que hacerla, en este caso, convirtiendo los enteros a reales y dividiendo posteriormente para obtener un número decimal.

3.3 Programación Dinámica Aleatoria

En la técnica de *Divide y Vencerás* vimos la posibilidad de incluir en el algoritmo una variable aleatoria para decidir el tamaño de los subproblemas. Esta idea puede ser usada también en la *Programación Dinámica*. Aquí tenemos la posibilidad adicional de escoger aleatoriamente una de las alternativas disponibles en todos o en un subconjunto de los problemas. La idea es

escoger aleatoriamente una de las alternativas posibles un número especificado de veces y posteriormente continuar con el algoritmo de *Programación Dinámica*. Por ejemplo escoger aleatoriamente solo una de las alternativas para los problemas de tamaño mayor a o igual a N y escoger todas las alternativas posibles para tamaños menores a N. Enfocado de esta manera el algoritmo de *Programación Dinámica Aleatoria* puede encontrar la solución con probabilidad una p que habrá que estimar y fallará en el resto de los casos. Esa probabilidad p dependerá evidentemente del valor de N escogido. Se trata de repetir el algoritmo hasta encontrar una solución.

$$ft(X) = \begin{cases} s_b, & si\ es\ un\ caso\ base \\ c\left(X, a', ft\left(X_1^{a'}\right), ft\left(X_2^{a'}\right), \dots, ft\left(X_k^{a'}\right)\right), & si\ t(X) \geq N, a' \in A_X \\ sA_{a \in A_X}\left(c\left(X, a, ft(X_1^a, N), ft(X_2^a, N), \dots, ft(X_k^a)\right)\right), & si\ t(X) < N \end{cases}$$

$$f(X) = \begin{cases} ft(X), & si\ ft(X)\ tiene\ solución \\ f(X), & si\ ft(X)\ no\ tiene\ solución \end{cases}$$

Donde a' es una de las alternativas posibles escogida aleatoriamente. Esta técnica es adecuada para resolver problemas que buscan una solución para un conjunto de restricciones. Si el problema a resolver es de optimización con esta técnica obtendremos una solución pero no la mejor. La técnica puede ser adaptada para encontrar un número dado de soluciones si las hay o encontrar una mejor aproximación a la mejor solución.

Si los problemas a resolver tienen varias soluciones los operadores $cA(\dots), sb$ tienen que escoger aleatoriamente una de ellas.

Ejemplos de este tipo son el Problema de las Reinas o el Problema del Sudoku. Ambos se resuelven de forma similar.

Esta técnica se puede combinar con la técnica de filtro vista antes y por supuesto con la memoria usual en programación dinámica.

3.4 Búsqueda de todas las soluciones

Como hemos visto anteriormente si el problema es de optimización la solución del problema está asociada a una de las alternativas del mismo. En este caso la solución parcial (a, p) del problema contendrá el valor de la alternativa a través de la cual se obtiene el valor de la propiedad a optimizar y el valor óptimo de la misma.

En otros casos (porque no es un Problema de Optimización o por otras razones) la solución de un problema no está asociada a una alternativa en particular. En estos casos el operador *sA(…), seleccionaAlternativa*, no escoge la solución parcial asociada a una alternativa. Tiene que construir un valor de la propiedad que se calcula a partir de los valores de la propiedad alcanzados a través de las diferentes alternativas. En estos casos hacemos que en la solución parcial la alternativa tenga un valor *null* y la propiedad el correspondiente valor calculado.

Un caso de la situación anterior, pero hay muchos más, es contar el número de soluciones de un problema cuando tiene más de una (posiblemente por no ser un problema de optimización o por otra razones). Ahora p recoge el número de soluciones del problema y a será *null*. La solución reconstruida, en la búsqueda de todas las soluciones, será ahora del tipo $Set < S >$ dónde *S* es el tipo de una solución del problema. Es conveniente recordar que los operadores *c (…)* y *sA(…)* que hay que tener en cuenta, también, para reconstruir la solución. En el caso de *c(…), combinaSoluciones*, la solución será *null* si lo es la de algún subproblema. En caso de *sA(…)* hay que filtrar las soluciones *null* antes de calcular la solución reconstruida o la propiedad de la solución parcial.

Para el caso de *Reducción* veremos, en el capítulo siguiente, técnicas específicas para encontrar todas las soluciones.

3.5 Búsqueda de sólo una solución

Cuando el problema puede tener muchas soluciones podemos tener interés sólo en una de ellas o varias de las posibles aunque no todas. Es un caso particular de la búsqueda de todas las soluciones. Se trataría de buscarlas todas y escoger de entre ellas aquellas en las que estemos interesados.

En el caso de Reducción la *Programación Dinámica* puede ser ajustada fácilmente para obtener una o varias de las soluciones de un problema. En este caso sabemos que hay una solución cuando alcanzamos un caso base desde el problema original. En ese momento podemos reconstruir la solución y hacer que el algoritmo termine, si sólo queremos una, o que termine cuando hayamos alcanzado varias veces los casos base. Para el caso de *Reducción* veremos, en el capítulo siguiente, técnicas específicas para encontrar las primeras soluciones.

4. Problema de las Reinas

El problema se enuncia así:

Colocar N reinas en un tablero de ajedrez $N \times N$ de manera tal que ninguna de ellas amenace a alguna 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 $[0, N)$.

Cada reina la representamos por $(c, f, dp, ds), f \in [0, N-1], c \in [0, N-1]$. Siendo c, f, dp, ds, la columna, la fila, la diagonal principal y la diagonal secundaria donde está colocada la reina. La columna y la fila son números enteros. La diagonal principal y la diagonal secundaria son representadas por números enteros que las identifican de manera única. Siendo estos enteros calculados como $dp = f - c, ds = f + c$. El tipo *Reina* tiene las propiedades individuales anteriores.

La solución buscada es una lista de reinas. La solución podemos dotarla de varias propiedades derivadas: fo, dpo, dso, lista de filas ocupadas, conjunto de diagonales principales ocupadas y conjunto de diagonales secundarias ocupadas. Como solución parcial, a partir de la cual

podemos reconstruir la solución, podemos escoger una lista de enteros donde en cada posición c se ubica una fila f. A partir de c, f podemos crear la reina correspondiente.

Generalizamos el problema: asumiendo ya colocadas $r < N$ reinas en las columnas $0, \dots, r-1$ colocar las $N - r$ restantes. Las reinas ya colocadas tienen ocupadas un conjunto de filas, de diagonales principales y secundarias. El problema generalizado tiene entonces las propiedad individual fo (lista de enteros, consultable) que es la lista de las filas ocupadas. Es decir en la posición cero se guarda la fila dónde está colocada la reina ubicada en la columna cero, etc. El problema tiene otras propiedades derivadas: r (entero, consultable) el número de reinas ya colocadas igual al tamaño de fo, dpo (conjunto de enteros, consultable) que contiene las diagonales principales ya ocupadas y dso (conjunto de enteros, consultable) que contiene las diagonales secundarias ya ocupadas. En este contexto dos problemas son iguales si tienen iguales la propiedad fo. El tamaño del problema es el número de reinas que nos quedan por colocar: $N - r$.

El problema generalizado tiene las propiedades

$$prg(N, r, fo, dpo, dso)$$

El problema original es una instancia del conjunto de problemas generalizados anteriores

$$pr(N) = prg(N, 0, \emptyset, \emptyset, \emptyset)$$

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

$$A_{N,r,fo,dpo,dso} = \{f \in [0, N) \mid f \notin fo, f - r \notin dpo, f + r \notin dso\}$$

Las alternativas son por lo tanto números enteros.

El objetivo del problema es encontrar una solución que tenga un número de reinas r igual a N y ninguna reina amenace a otra. Esto es equivalente a decir que los cardinales de fo, dpo, dso (propiedades derivadas de la solución) sean iguales a N.

El caso base es cuando tengamos colocadas todas las reinas $N = r$. La solución puede obtenerse de las propiedades del problema. Es decir la solución parcial es fo directamente.

$$prg(p) = \begin{cases} \bot, & A = \emptyset \\ fo, & N = r \\ cA_{f \in A}(c(p, f, prg(sp))), & r < N \end{cases}$$

$$p = (N, r, fo, dpo, dso)$$

$$sp = (N, r + 1, fo + f, dpo + d1, dso + d2), d1 = f - r, d2 = f + r$$

$$A_{N,r,fo,dpo,dso} = \{f \in [0, N) \mid f \notin fo,\ f - r \notin dpo,\ f + r \notin dso\}$$

$$c(p, f, s = s)$$

El operador $cA(\ldots)$si sólo queremos una solución tiene la forma:

$$cA_{f \in A}(s^f) = s$$

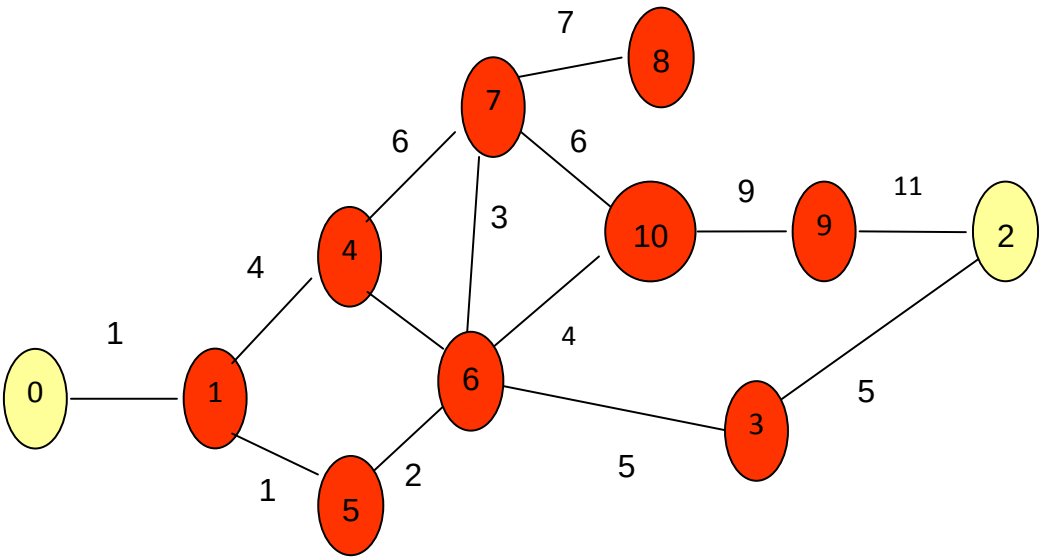
Donde s es cualquiera de las soluciones. Si queremos obtener todas las soluciones podemos usar las ideas vistas más arriba.

Este problema se adapta bien para usar la técnica de la *Programación Dinámica Aleatoria.* Como hemos explicado el problema necesita tener una probabilidad de encontrar la solución al primer intento para que el esquema funcione. Ese es el caso para este problema.

5. Problema del Camino Mínimo

Enunciado

Sea el grafo g que sea muestra abajo donde los vértices son *Ciudades* y las aristas caminos entre ellas. Cada arista está etiquetada por un entero que es la longitud entre las ciudades respectivas



Cada vértice del grafo lo indexamos mediante un entero. Asumimos que el índice está comprendido en $[0, n-1]$ dónde n es el número de ciudades. Queremos encontrar el *Camino Mínimo* entre dos ciudades dadas que representaremos por i, j. Este problema se puede resolver de muchas otras formas como veremos más adelante. 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, g). Donde i, j toman valores en $[o, n-1]$ y k en $[-1, n-1]$. El valor de $k = -1$ 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 $\perp$. Como soluciones parciales escogemos, de forma similar a otros casos, el par formado por la alternativa y la longitud del camino.

Representamos por g el grafo de partida y por (i,j) la arista, si la hay entre i y j y por $|(i,j)|$ su longitud. 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.

El problema lo podemos esquematizar de la forma:

$$cmg(p) = \begin{cases} (N,0), & i = j \\ \perp, & i \neq j \wedge k = -1 \wedge (i,j) \notin g \\ (N,|(i,j)|), & i \neq j \wedge k = -1 \wedge (i,j) \in g \\ \min(cmg(sp_{Y,1}) + cmg(sp_{Y,2}), & \\ \quad cmg(sp_{N,1})), & i \neq j \wedge k \geq 0 \end{cases}$$

$p = (i,j,k,g)$
$sp_{Y,1} = (i,k,k-1,g)$
$sp_{Y,2} = (k,j,k-1,g)$
$sp_{N,1} = (i,k,k-1,g)$
$(a,l1) < (b,l2) = l1 < l2$

Del esquema anterior podemos deducir los operadores $c(\ldots), cA(\ldots)$ y el número de subproblemas para cada alternativa.

El problema original es una instancia del conjunto de problemas generalizados

$$cm(i,j,g) = cmg(i,j,n-1,g)$$

La función de reconstrucción, asumiendo la memoria m, es:

$$fr(p,m) = \begin{cases} \perp, & p \notin m \\ (), & p \in m, i = j \\ (i,j), & p \in m, i \neq j, k = -1 \\ fr(sp_{N,1},m), & p \in m, i \neq j, k \geq 0, \ m_a(p) = N \\ fr(sp_{Y,1},m) + fr(sp_{Y,2},m), & p \in m, i \neq j, k \geq 0, \ m_a(p) = Y \end{cases}$$

Donde $p, sp_{Y,1}, sp_{Y,2}, sp_{N,1}, (i,j)$ representan lo mismo que anteriormente. Además dotamos a dos caminos $l1, l2$ de la operación de concatenación $l1 + l2$ y la de inicialización $l = (i,j)$ que convierte una arista en un camino de un solo tramo.

Como hemos visto más arriba la solución de un problema de *Programación Dinámica con Memoria* pueden generalizarse fácilmente para buscar la **solución de conjuntos de problemas** relacionados. Como vimos se trataba de indicar el conjunto de problemas a resolver y aprovechar los cálculos guardados en la memoria de unos problemas para otros. En este caso, con estas ideas, podríamos buscar los *Caminos Mínimos* entre dos ciudades de grafo cualesquiera. Para ello proporcionaríamos un Iterable que contenga todos los pares de Ciudades del grafo. Al resolver cada problema se utilizarían las soluciones a los subproblemas que se comparten con el resto de problemas del Iterable. Esta esquema, para la búsqueda de los Caminos Mínimos entre cada dos ciudades es conocida como el **Algoritmo de Floyd-Warshall.** El algoritmo descrito aquí es la versión recursiva. Usualmente en la literatura se presenta la versión iterativa que puede obtenerse fácilmente de la versión recursiva y que veremos en capítulos posteriores.

6. Corrección y complejidad de los Algoritmos de Programación Dinámica

6.1 Corrección de un algoritmo de Programación Dinámica

Para demostrar la corrección:

- Demostrar que las soluciones de los casos base son correctas
- Suponiendo que son correctas las soluciones de los subproblemas demostrar que la combinación de sus soluciones mediante el operador *c(…)* produce la solución correcta.
- Suponiendo que son correctas las soluciones para cada una de las alternativas demostrar que la combinación de sus soluciones mediante el operador *cA(…)* produce la solución correcta para del problema completo.
- Demostrar que el tamaño va decreciendo estrictamente cuando pasamos de un problema a sus subproblemas y esa secuencia decrece alcanzando alguno de los casos base.
- Dado un problema demostrar que las alternativas elegidas cubren todas las posibilidades.

6.2 Complejidad de los problemas de Programación Dinámica

Al ser la *Programación Dinámica* una extensión de la técnica de *Divide y Vencerás con Memoria* podemos usar la misma metodología para el cálculo de la complejidad. Para resolverlo debemos resolver un subconjunto de problemas. Para un problema x sea $g(x)$ el tiempo necesario para calcular las alternativas, los subproblemas, el tiempo para combinar las soluciones de los subproblemas con el operador $c(\dots)$ y posteriormente con el $cA(\dots)$. Sea I el conjunto formado por los problemas que hay que resolver para poder obtener la solución de X de tamaño n. Entonces tiempo de ejecución podemos expresarlo como:

$$T(n) = \sum_{x \in I} g(x)$$

Esto es debido a que, supuesto calculados los subproblemas, el tiempo para calcular un problema x es $g(x)$. Luego el tiempo necesario para calcular un problema es la suma de lo que se requiere para cada uno de los subproblemas que es necesario resolver.

Un caso particular, muy usado, se presenta cuando $g(x) = k$. Entonces la fórmula anterior queda:

$$T(n) = \sum_{x \in I} g(x) = k \sum_{i \in I} 1 = k|I| = \Theta(|I|)$$

El problema, para el cálculo de la complejidad, es calcular $\Theta(|I|)$. Veremos ejemplos posteriormente.

7. Versiones iterativas de algoritmos de Programación Dinámica

En muchos casos es posible encontrar un esquema iterativo a partir de otro recursivo de *Programación Dinámica*. Para ello debemos pensar, como para todos los algoritmos iterativos, en un estado, el valor inicial de este estado y el cambio de ese estado en cada iteración. No hay una forma general de hacer esto pero, como vimos en las transformaciones recursivo-iterativas, podemos considerar dos enfoques: uno hacia abajo (bastante sistemáticos si el problema es recursivo lineal final, y otro de abajo arriba que parte de los casos base y a partir de ellos va construyendo las soluciones.

Veamos algún ejemplo del segundo enfoque dejando otros detalles para capítulos próximos. En todos los algoritmos iterativos tenemos que diseñar un estado. En los casos en que la solución de un problema de tamaño k depende solamente de subproblemas de tamaño $k-1$ podemos elegir el estado para que guarde las soluciones de todos los problemas de un tamaño dado n y sus soluciones. El estado inicial contendrá las soluciones a los problemas de tamaño pequeño asociados a los casos base. La transición del estado que contiene los problemas de tamaño n y sus soluciones a los de tamaño $n+1$ puede obtenerse del planteamiento recursivo del problema.

En efecto con el planteamiento anterior el problema X sería de tamaño n y los problemas $X_1^a, X_2^a, \ldots, X_r^a$ de tamaño $n-1$. El estado podemos diseñarlo para que contenga las soluciones a los problemas de tamaño n como hemos comentado. El estado podría ser diseñado, en general, como una función que teniendo como dominio los problemas de tamaño n nos proporcione la solución para cada uno de ellos (o posiblemente la no existencia de solución). Si los problemas de tamaño n pueden ser identificados por dos propiedades de tipo entero entonces la función puede ser implementada por una tabla.

La transición de un estado al siguiente consistirá encontrar la solución de cada problema de tamaño $n+1$ usando la fórmula de la *Programación Dinámica* anterior. El esquema de actualización del estado sería:

Para cada problema X de tamaño n {

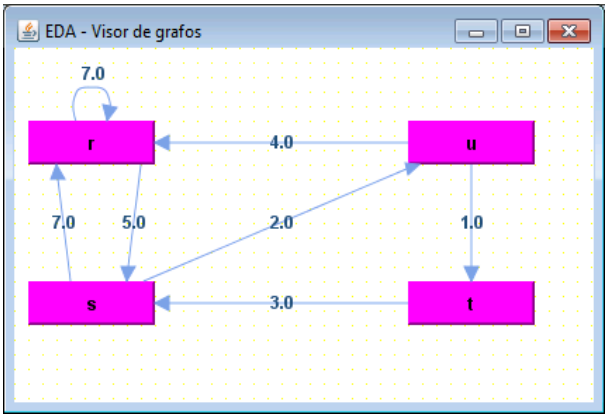
$$f(X) = cA_{a \in A_X}(c(X, a, f(X_1^a), f(X_2^a), \dots, f(X_r^a))$$

}

El bucle anterior define la transición del estado *e* al *e'* siendo $f(X)$ la solución del problema *X* en el estado *e'* y $f(X_i^a)$ las soluciones disponibles en el estado *e*. Es decir calcula las soluciones de los problemas de tamaño n a partir de las soluciones de los problemas de tamaño $n-1$. El estado tiene que inicializarse con las soluciones de los casos base.

Las ideas anteriores pueden ser adaptadas para encontrar la versión iterativa del algoritmo recursivo de *Floyd-Warshall* arriba. Como vimos allí cada problema se representaba por la tripleta (i, j, k, g) siendo $k+1$ el tamaño del problema. Por lo tanto los problemas de tamaño k pueden representar por el par de enteros (i, j) ya que le grafo es una propiedad compartida. Tal como hemos comentado antes la función, que para cada estado, nos proporciona la solución de un problema puede implementarse por una tabla. Esta tabla t_k nos dará el camino del vértice i al j usando vértices en el conjunto $[0, k]$ o $\perp$ si no hay camino. Designamos entonces por $t_k(i, j)$ el camino y por $|t_k(i, j)|$ su longitud.

Ejemplo: Hallar el camino mínimo en el grafo con las siguientes distancias:



El caso base es para $k = -1$. Es decir caminos de i a j sin usar ningún vértice intermedio. El estado inicial viene representado en la tabla de distancias, t_{-1}, siguiente:

$$c = q_{[0]} = \begin{array}{c} \begin{array}{cccc} r & s & t & u \end{array} \\ \begin{pmatrix} 7 & 5 & \infty & \infty \\ 7 & \infty & \infty & 2 \\ \infty & 3 & \infty & \infty \\ 4 & \infty & 1 & \infty \end{pmatrix} \end{array} \begin{array}{c} r \\ s \\ t \\ u \end{array}$$

El la fila *i* columna *j* tenemos la distancia del camino de *i* a *j* para los problemas de tamaño *k =-1 (tamaño 0)* es decir aristas en el grafo. Hemos representado por infinito el hecho de no existir arista de *i* a *j*.

El algoritmo recursivo de *Floyd-Warshall* dos daba las soluciones de los problemas de tamaño k a partir de los de tamaño *k-1* mediante la expresión:

$$cm(i,j,k) = \min\left(cm(i,k,k-1) + cm(k,j,k-1)), cm(i,j,k-1)\right)$$

Aplicando la expresión al estado representado por la tabla anterior obtenemos el estado siguiente. Recordamos que *cm(i,j,k)* nos devolvería el camino mínimo de i a j para los problemas de tamaño k (estado e') y *cm(i,j,k-1)* nos devolvería el camino mínimo de i a j para los problemas de tamaño k-1 (estado e). Aplicando la fórmula de manera iterativa llegaremos a los problemas de tamaño *n* (siendo *n* el número de vértices del grafo) que dos da la solución buscada. Además debemos ir actualizando otra tabla que guarde el camino mínimo de *i* a *j* para los problemas de tamaño *k*.

El esquema concreto es de la forma:

Inicializar t_{-1}*;*
k=-1;
Mientras k<n {
    Para cada i {
        Para cada j {
            $t_{k+1}(i,j) = \min\left(t_k(i,k) + t_k(k,j)), t_k(i,j)\right)$*;*
        }
    }
    k=k+1;
}

A la hora de implementar el algoritmo sólo necesitamos un estado con la tabla t_k. Como cada índice recorre *n* valores la complejidad del algoritmo es $\Theta(n^3)$ siendo n el número de vértices del grafo. Esta complejidad es la misma que para el caso recursivo.

Los sucesivos estados asociados al problema del grafo anterior son:

1ª Iteración:

$$q_{[1]} = \begin{array}{c} \begin{array}{cccc} r & s & t & u \end{array} \\ \begin{pmatrix} 7 & \underline{5} & \infty & \infty \\ \underline{7} & \underline{12} & \infty & \underline{2} \\ \infty & \underline{3} & \infty & \infty \\ 4 & \underline{9} & 1 & \infty \end{pmatrix} \end{array} \begin{array}{c} r \\ s \\ t \\ u \end{array}$$

2ª Iteración:

$$q_{[2]} = \begin{array}{c} \begin{matrix} r & s & t & u \end{matrix} \\ \begin{pmatrix} 7 & 5 & \underline{\infty} & 7 \\ 7 & 12 & \underline{\infty} & 2 \\ \underline{10} & \underline{3} & \underline{\infty} & \underline{5} \\ 4 & 9 & 1 & 11 \end{pmatrix} \end{array} \begin{matrix} r \\ s \\ t \\ u \end{matrix}$$

3ª Iteración:

La matriz de distancia después de esta iteración es:

$$q_{[3]} = \begin{array}{c} \begin{matrix} r & s & t & u \end{matrix} \\ \begin{pmatrix} 7 & 5 & \infty & \underline{7} \\ 7 & 12 & \infty & \underline{2} \\ 10 & 3 & \infty & \underline{5} \\ \underline{4} & \underline{4} & \underline{1} & \underline{6} \end{pmatrix} \end{array} \begin{matrix} r \\ s \\ t \\ u \end{matrix}$$

4ª Iteración:

La matriz de distancia después de esta iteración es:

$$q_{[4]} = \begin{array}{c} \begin{matrix} r & s & t & u \end{matrix} \\ \begin{pmatrix} 7 & 5 & 8 & 7 \\ 6 & 6 & 3 & 2 \\ 9 & 3 & 6 & 5 \\ 4 & 4 & 1 & 6 \end{pmatrix} \end{array} \begin{matrix} r \\ s \\ t \\ u \end{matrix}$$

Ya se han hecho todas las iteraciones posibles. Por tanto, el camino mínimo entre 2 vértices cualesquiera del grafo será el obtenido en tabla final anterior.

Las ideas presentadas anteriormente son aplicables a lo obtención de algoritmos iterativos a partir de los algoritmos recursivos de Programación Dinámica en los cuales la solución para un problema de tamaño k se exprese a partir de la solución de problemas de tamaño $k-1$. Para que le estado sea implementable de manera adecuada el número de problemas de un tamaño dado k debe ser acotado y suficientemente pequeño para que sus soluciones puedan guardarse eficientemente.

8. Programación Dinámica y Algoritmo A*

Como hemos visto más arriba cuando los problemas de Programación Dinámica son de reducción cada problema, tras escoger una alternativa, tiene un solo subproblema. El conjunto de problemas define, por lo tanto, un grafo implícito dónde los vértices son los problemas. En este grafo existe una arista de un problema a cada uno de sus subproblemas. Pro generalidad podemos definir un nuevo problema, el problema final, a cual se reducen todos los casos base.

En ese diseño cada arista puede ser etiquetada con el valor de la propiedad objetivo, si es un problema de optimización, que se acumula al pasar de un problema a un subproblema. Recordemos que una solución de un problema de Programación Dinámica de Reducción es un camino desde el Problema Inicial al que hemos definido como Problema Final. El camino que da la solución óptima es aquel cuya suma de los valores asignados a las aristas sea óptima.

Así planteado el problema de Programación Dinámica de Reducción y Optimización se transforma en un problema de Camino Mínimo que puede ser resuelto por el Algoritmo A*.

Veremos algún ejemplo más abajo.

9. Detalles de Implementación

9.1 Tipos y Algoritmos Genéricos para la Programación Dinámica

El esquema de programación dinámica puede ser programado de forma genérica. Para ello hace falta diseñar una interfaz que deben implementar los problemas que se quieran resolver por esta técnica. El tipo será genérico y sus parámetros

- *S*: Tipo de la solución del problema
- *A*: Tipo de la alternativa
- *T*: El tipo de la solución parcial

Los problemas que se quieran resolver por la técnica de Programación Dinámica deben implementar el tipo *ProblemaPD<S,A,T>.* .

```
public interface ProblemaPD<S,A,T> {
	Tipo getTipo();
	int size();
	boolean esCasoBase();
	Sp<A, T> getSolucionCasoBase();
	Sp<A, T> seleccionaAlternativa(List<Sp<A, T>> ls);
	ProblemaPD<S,A,T> getSubProblema(A a, int i);
	Sp<A, T> combinaSolucionesParciales(A a, List<Sp<A, T>> ls);
	Iterable<A> getAlternativas();
	int getNumeroSubProblemas(A a);
	S getSolucionReconstruida(Sp<A,T> sp);
	S getSolucionReconstruida(Sp<A,T> sp, List<S> ls);
	Double getObjetivo();
	Double getObjetivoEstimado(A a);
}
```

La última versión de este tipo puede encontrarse en el API.

Para hacer el algoritmo más flexible se han diseñado las soluciones parciales como pares alternativa-valor. Estos pares son implementados por la clase `Sp<A, T>`.

Veamos cada uno de los métodos.

- *int size():* Calcula el tamaño del problema
- *boolean isSolucion(S s):* Decide si s es una solución del problema
- *boolean esCasoBase():* Decide si s el problema es un caso base
- *Sp<A, T> getSolucionCasoBase():* Obtiene la solución del problema asumiendo que es un caso base
- *Sp<A, T> seleccionaAlternativa(List<Sp<A, T>> ls):* Combina las soluciones obtenidas para las diferentes alternativas. Asumimos que si *ls* está vacía el problema no tiene solución. La no existencia de solución lo podemos representar de muchas formas. Una posibilidad, que seguiremos aquí, es representar la no solución por el valor *null.* Este método debe tener en cuenta, además, que tiene que descartar aquellas alternativas que no alcancen soluciones (en ese caso representadas por el valor *null*). Tras el filtro de los valores *null* se debe escoger la solución parcial asociada a una alternativa, en cuyo caso el par devuelto es de la forma (a, p), o combinar las propiedades asociadas a las diferentes alternativas. En este último caso el par devuelto será de la forma $(null, pc)$ dónde pc es el valor de la propiedad combinada.
- *ProblemaPD<S,A,T> getSubProblema(A a, int i):* Obtiene el subproblema *i después de tomar la alternativa a.*. Los problemas se numeran desde cero.
- *Sp<A, T> combinaSolucionesParciales(A a, List<Sp<A, T>> ls):* Combina las soluciones de los subproblemas. Ha de tener en cuenta que si no hay solución para un subproblema (solución *null*) entonces no hay solución para el problema.
- *Iterable<A> getAlternativas():* Devuelve un iterable con las alternativas posibles para ese problema.
- *int getNumeroSubProblemasAnd(A a):* Número de subproblemas dada una alternativa.
- *S getSolucionReconstruida(Sp<A,T> sp):* Método proporcionado para reconstruir la solución asumiendo que es un caso base con solución parcial *sp*.
- *S getSolucion(Sp<A,T> sp, List<S> ls):* Método proporcionado para reconstruir la solución a partir de las soluciones de los subproblemas. Se asume que no es un caso base y que su solución parcial está en sp y las soluciones de los hijos en ls.
- *S getSolucion(List<S> ls):* Método proporcionado para reconstruir la solución a partir de las soluciones encontradas tras escoger las diferentes alternativas. Si la solución del problema está asociada a una alternativa dada este método no es necesario y puede devolver null.
- *Double getObjetivoEstimado(A a):* Es una cota inferior del valor estimado de la propiedad a optimizar del problema inicial si tomamos la alternativa *a* en el problema actual si estamos minimizando y una cota superior si estamos maximizando.
- *Double getObjetivo():* Es el valor de la propiedad a optimizar del problema inicial

Normalmente la implementación de los métodos anteriores se lleva a cabo añadiendo a cada problema una propiedad adicional que es el valor acumulado de la propiedad a optimizar desde el problema original hasta el problema actual. Para cada problema, además, se diseña un método que nos proporcione una cota del valor de la propiedad objetivo para el problema actual si elegimos una alternativa concreta entre las posibles. Llamaremos a este método función de cota.

Para que el filtro elimine sólo las alternativas que no conducen a una mejor solución, si el problema es de maximización, la función de cota debe cumplir:

$$p.getObjetivoEstimado(a) > pr^*$$

Dónde p es el problema, a la alternativa elegida y pr^* el máximo valor de la propiedad objetivo. Si el problema es de minimización entonces se debe cumplir:

$$p.getObjetivoEstimado(a) < pr^*$$

Es decir la cota debe ser una cota superior en el caso de maximización y una cota inferior en el caso de minimización.

El método *getObjetivoEstimado(a)* se puede implementar a partir del valor acumulado y la función de cota estimada. El método *getObjetivo()* se puede implementar devolviendo el valor acumulado pero sólo se puede llamar en los casos base.

Definidas las interfaces anteriores el algoritmo de la *Programación Dinámica* puede ser implementado en una clase reutilizable. Es la clase *AlgoritmoProgramacionDinamica<S,A,T>.* El núcleo del algoritmo es el algoritmo privado *pD* que se muestra a continuación. Aunque la última versión de este tipo puede encontrarse en el API.

Veamos los detalles del algoritmo. Las soluciones parciales obtenidas para los subproblemas se guardan en la variable *map* de tipo *Map<ProblemaPD<S,A,E>,E>*. Para resolver el problema seguimos los siguientes pasos:

- Comprobamos si el problema ha sido ya resuelto. Si ha sido resuelto devolvemos su solución (que puede ser el valor *null* que representa la no solución).
- Si el problema no ha sido resuelto preguntamos si el problema es un caso base y si lo es lo resolvemos y guardamos el resultado en la memoria.
- Si el problema es un caso recursivo obtenemos todas las alternativas posibles después de filtrarlas si estamos en la modalidad *Randomize.* Para cada una de ellas que pase el filtro (si es en la modalidad *Con Filtro*) calculamos los subproblemas. Resolvemos cada subproblema y obtenemos la solución con el método *combinaSoluciones* y posteriormente combinamos todas las soluciones (con el método *combinaAlternativas)* para obtener la solución del problema. Luego guardamos la solución en la memoria.
- Para comprobaciones posteriores podemos guardar para cada problema la lista de las alternativas recorridas.
- Si es un problema *Con Filtro* hay que ver si las alternativas pasan el filtro y actualizar el mejor valor.

- El método ejecuta resuelve una secuencia de problemas o itera hasta que encuentra solución para el primero de ellos.
- La clase tiene varias propiedades públicas que pueden ser consultadas y actualizadas: *solucionesParciales, isRandomize, conFiltro, tipo.* La primera podemos usarla para consultar la solución parcial de cada problema y poder reconstruir la solución a partir de ella. Las siguientes nos sirven para escoger el tipo específico de *Programación Dinámica*.

La reconstrucción de la solución puede hacerse por el *método getSolucion(ProblemaPD<S,A,T> pd)* que se explicita más arriba.

Es importante hacer notar:

- El algoritmo puede resolver un problema o un conjunto de problemas (pasándole un iterable). Cuando resuelve un conjunto de problemas usa la memoria acumulad en un problema para resolver los siguientes.

10. Un problema concreto: el Problema de la Mochila

Veamos los detalles del *Problema de la Mochila*. Pero, como veremos, la mayoría de las decisiones se repiten en muchos problemas. Para cada tipo implementaremos la factoría correspondiente.

Los objetos que están disponibles para introducir en la mochila tienen las propiedades: *código, valor unitario, peso unitario, número máximo de unidades en la mochila y ratio valor peso*. Diseñamos la clase *ObjetoMochila* con esas propiedades y dos métodos de factoría: uno que toma los valores de las propiedades básicas y otro que toma una cadena de caracteres con la información para las propiedades básicas separada por comas. Hacemos que los objetos del tipo tengan un orden natural dado por la propiedad ratio valor peso y si esta es igual por el código. Lo diseñamos como un tipo inmutable.

La solución del Problema de la Mochila, tal como hemos comentado anteriormente, es un multiconjunto de objetos con las propiedades adicionales *valor total* y *peso total* de la mochila. Para ello implementamos la clase *SolucionMochila*. Su orden natural será según el valor total de la mochila. Dotamos a los objetos de este tipo de tres propiedades: *ValorTotal*, *PesoTotal* y *Objetos*. La tercera, de tipo *Multiset<ObjetoMochila>* es la única propiedad básica las otras dos son derivadas. La clase tendrá dos constructores: uno que construye una solución vacía También un método para añadir varias unidades de un objeto a la solución. La diseñamos como un tipo inmutable.

La propiedad a optimizar es de tipo entero, las alternativas también de tipo entero. El Problema de la Mochila es un problema de optimización donde se da una capacidad para la mochila y una lista de objetos disponibles. Estas propiedades del problema original estarán disponibles para todos los problemas. Vamos a resolver el problema mediante la técnica *Con Filtro.*

Diseñamos la clase *ProblemaMochila* como una factoría para crear problemas generalizados, para hacer algunos trabajos previos como leer los objetos disponibles de un fichero y ordenarlos. Estos objetos disponibles ya ordenados es una propiedad de esta clase que compartirán todos los problemas generalizados.

La clase *ProblemaMochilaPDF* es el tipo de los problemas generalizados que se van a resolver por la técnica de la *Programación Dinámica con Filtro*.

Para este problema ya vimos las propiedades de los problemas generalizados. Estos tienen las propiedades individuales: *capacidad, index* y *valor acumulado*. También tienen la propiedades compartidas: *objetosDisponibles* y *mejorValorEncontrado.* Otras propiedades compartidas es el orden sobre las soluciones parciales (sobre el tipo *Sp<Integer,Integer>).* Dos problemas son iguales si tienen iguales *capacidad y index.*

El código es como sigue. La última versión de este tipo puede encontrarse en el API.

```
public class ProblemaMochilaPDF implements ProblemaPDF<SolucionMochila,
       Integer, Integer> {



       public static ProblemaMochilaPDF create(ProblemaMochila problema,
         Integer valorAcumulado) {
              return new ProblemaMochilaPDF(problema, valorAcumulado);
       }

       public static ProblemaMochilaPDF create() {
              AlgoritmoPD.tipo = AlgoritmoPD.Tipo.Max;
              return new ProblemaMochilaPDF(ProblemaMochila.create(), 0);
       }

       private ProblemaMochila problema;
       private Integer valorAcumulado;

       private ProblemaMochilaPDF(ProblemaMochila problema, Integer
          valorAcumulado) {
              super();
              this.problema = problema;
              this.valorAcumulado = valorAcumulado;
       }

       @Override
       public int size() {
              return problema.getIndex();
       }

       @Override
       public boolean esCasoBase() {
              return problema.getIndex() == 0 || problema.getCapacidad() == 0;
       }

       @Override
       public Sp<Integer,Integer> getSolucionCasoBase() {
              Preconditions.checkState(this.esCasoBase(),
```

```
                "El problema "+toString()+ " no es un caso base");
        Integer numeroDeUnidades =
        ProblemaMochila.numeroEnteroMaximoDeUnidades(
                problema.getIndex(), problema.getCapacidad());
        return new Sp<Integer,Integer>(numeroDeUnidades,
                numeroDeUnidades*ProblemaMochila.getValorObjeto(
                        problema.getIndex()));
    }

    @Override
    public Sp<Integer,Integer> combinaAlternativas(
       List<Sp<Integer,Integer>> ls) {
        Sp<Integer,Integer> e = null;
        if(!ls.isEmpty()){
                e = Collections.max(ls);
        }
        return e;
    }

    @Override
    public ProblemaPD<SolucionMochila, Integer, Integer>
       getSubProblema(Integer a, int i) {
        Preconditions.checkArgument(i==0,
             "Solo hay un problema y se ha recibido  i = "+i);
        int index = problema.getIndex();
        ProblemaMochila p =ProblemaMochila.create(
                problema.getCapacidad()-
                      a*ProblemaMochila.getPesoObjeto(index),index-1);
        Integer acumulado = this.valorAcumulado
                +a*ProblemaMochila.getValorObjeto(index);
        ProblemaMochilaPDF pr = create(p,acumulado);
        return pr;
    }

    @Override
    public Sp<Integer,Integer> combinaSoluciones(
       Integer a, List<Sp<Integer,Integer>> ls) {
        Preconditions.checkArgument(ls.size()==1,
            "Solo hay un problema y se ha recibido  = "+ls.size());
        Sp<Integer,Integer> e = null;
        int index = problema.getIndex();
        if(ls!=null) e =ls.get(0);
        if(e!=null && e.alternativa!=null) e = new Sp<Integer,Integer>(
           a,ls.get(0).propiedad
                +a*ProblemaMochila.getValorObjeto(index));
        return e;
    }

    @Override
    public Iterable<Integer> getAlternativas() {
        Iterable<Integer> ite = Iterables2.fromArithmeticSequence(
            ProblemaMochila.numeroEnteroMaximoDeUnidades(
                problema.getIndex(), problema.getCapacidad()),-1,-1);
        return ite;
    }


    @Override
    public int getNumeroSubProblemas(Integer a) {
```

```
            return this.esCasoBase() ? 0 : 1;
      }

      @Override
      public SolucionMochila getSolucion(Integer a,
         List<SolucionMochila> ls) {
            SolucionMochila s;
            if(ls.isEmpty()){
                  s = SolucionMochila.create();
            } else {
                  s = ls.get(0);
            }
            if(a>0){
                  s = s.add(ProblemaMochila
                        .getObjeto(problema.getIndex()), a);
            }
            return s;
      }

      @Override
      public Integer getObjetivoEstimado(Integer a) {
            returnthis.valorAcumulado
                  +problema.getCotaSuperiorValorEstimado(a);
      }

      @Override
      public Integer getObjetivo() {
            return this.valorAcumulado;
      }
```

Para implementar lo anterior necesitamos una clase *ProblemaMochila* que gestione la creación de problemas, el cálculo de sus propiedades, los detalles del problema inicial, la estimación de la cota para el valor de la propiedad de cada problema, etc.

La última versión de este tipo puede encontrarse en el API.

```
public class ProblemaMochila {

      private static List<ObjetoMochila> objetosDisponibles;
      private static Ordering<ObjetoMochila> ordenObjetos;
      public static Integer capacidadInicial;

      public static void leeObjetosDisponibles(String fichero){
            ordenObjetos = Ordering.<ObjetoMochila>natural();
            Iterable<String> is = Iterables2.fromFile(fichero);
            objetosDisponibles = Lists.newArrayList();
            for(String s : is){
                  objetosDisponibles.add(ObjetoMochila.create(s));
            }
            Collections.sort(getObjetosDisponibles(), ordenObjetos);
      }

      public static List<ObjetoMochila> getObjetosDisponibles() {
            return objetosDisponibles;
      }
```

```
public static Ordering<ObjetoMochila> getOrdenObjetos() {
	return ordenObjetos;
}

public static ProblemaMochila create(Integer capacidad, Integer index) {
	return new ProblemaMochila(capacidad, index);
}

public static ProblemaMochila create() {
	return new ProblemaMochila(
		ProblemaMochila.capacidadInicial,
			ProblemaMochila.getObjetosDisponibles().size()-1);
}

private Integer capacidad;
private Integer index;



private ProblemaMochila(Integer capacidad, Integer index) {
	super();
	this.capacidad = capacidad;
	this.index = index;
}

public Integer getCapacidad() {
	return capacidad;
}

public Integer getIndex() {
	return index;
}

public static ObjetoMochila getObjeto(int index){
	return ProblemaMochila.getObjetosDisponibles().get(index);
}

public static Integer getValorObjeto(int index){
	return ProblemaMochila.getObjetosDisponibles()
		.get(index).getValor();
}

public static Integer getPesoObjeto(int index){
	return ProblemaMochila.getObjetosDisponibles()
		.get(index).getPeso();
}

public static Integer getNumMaxDeUnidades(int index){
	return ProblemaMochila.getObjetosDisponibles()
		.get(index).getNumMaxDeUnidades();
}

public Integer getNumMaxDeUnidades(){
	return ProblemaMochila.getNumMaxDeUnidades(this.index);
}

public static Integer numeroEnteroMaximoDeUnidades(Integer index,
	Integer capacidad){
```

```
		return Math.min(capacidad/ProblemaMochila.getPesoObjeto(index),
			ProblemaMochila.getNumMaxDeUnidades(index)) ;
	}

	public static Double numeroRealMaximoDeUnidades(Integer index,
		Double capacidad){
		return Math.min(capacidad/ProblemaMochila
			.getPesoObjeto(index),
				ProblemaMochila.getNumMaxDeUnidades(index)) ;
	}

	public ProblemaMochila getSubProblema(Integer a) {
		int index = this.getIndex();
		return ProblemaMochila.create(this.getCapacidad()
			-a*ProblemaMochila.getPesoObjeto(index),index-1);
	}



	public IntStream getAlternativas() {
		IntStream r;
		if (this.isFinal()) {
			r = IntStream.empty();
		} else if (this.getCapacidad() == 0) {
			r = IntStream.of(0);
		} else if (this.getIndex() == 0) {
			r = IntStream.of(this.getNumMaxDeUnidades());
		} else {
			r = IntStream.rangeClosed(0,
				ProblemaMochila.numeroEnteroMaximoDeUnidades(
					this.getIndex(), this.getCapacidad()));
		}
		return r;
	}

	public Integer getAlternativa(ProblemaMochila p) {
		Preconditions.checkArgument(hayAlternativa(p));
		int index1 = this.index;
		int index2 = p.index;
		Preconditions.checkArgument(index1-index2 == 1);
		int capacidad1 = this.capacidad;
		int capacidad2 = p.capacidad;
		int peso = ProblemaMochila.getPesoObjeto(index1);
		Preconditions.checkArgument((capacidad1-capacidad2)%peso ==0);

		return (capacidad1-capacidad2)/peso;
	}

	public boolean hayAlternativa(ProblemaMochila p) {
		int index1 = this.index;
		int index2 = p.index;
		if(index1-index2 != 1) return false;
		int capacidad1 = this.capacidad;
		int capacidad2 = p.capacidad;
		int peso = ProblemaMochila.getPesoObjeto(index1);
		return (capacidad1-capacidad2)%peso ==0;
	}

	public boolean isFinal(){
```

```
		return this.index < 0;
	}

	public Integer getCotaSuperiorValorEstimado(){
		Double r = 0.;
		Double c = (double)getCapacidad();
		Double nu;
		int index = getIndex();
		while(true) {
			if(index < 0 || c<= 0.) break;
			nu = ProblemaMochila.numeroRealMaximoDeUnidades(index,c);
			r = r+nu*ProblemaMochila.getValorObjeto(index);
			c = c-nu*ProblemaMochila.getPesoObjeto(index);
			index--;
		}
		return (int)Math.ceil(r);
	}

	public Integer getCotaSuperiorValorEstimado(Integer a){
		Double r = 0.;
		Double c = (double)getCapacidad();
		Double nu =(double) a;
		int index = getIndex();
		while(true) {
			r = r+nu*ProblemaMochila.getValorObjeto(index);
			c = c-nu*ProblemaMochila.getPesoObjeto(index);
			index--;
			if(index < 0 || c<= 0.) break;
			nu = ProblemaMochila.numeroRealMaximoDeUnidades(index,c);

		}
		return (int)Math.ceil(r);
	}

	public Integer getCotaInferiorValorEstimado() {
		Integer r = 0;
		Integer c = getCapacidad();
		Integer nu;;
		int index = getIndex();
		while(true) {
			if(index < 0 || c<= 0.) break;
			nu = ProblemaMochila.numeroEnteroMaximoDeUnidades(index,c);

			r = r+nu*ProblemaMochila.getValorObjeto(index);
			c = c-nu*ProblemaMochila.getPesoObjeto(index);
			index--;
		}
		return (int)Math.ceil(r);
	}

	public Integer getCotaInferiorValorEstimado(Integer a) {
		Integer r = 0;
		Integer c = getCapacidad();
		Integer nu = a;
		int index = getIndex();
		while(true) {
			r = r+nu*ProblemaMochila.getValorObjeto(index);
			c = c-nu*ProblemaMochila.getPesoObjeto(index);
			index--;
```

```
                    if(index < 0 || c<= 0.) break;
                    nu = ProblemaMochila.numeroEnteroMaximoDeUnidades(index,c);

             }
             return (int)Math.ceil(r);
      }


      public boolean isSolucion(SolucionMochila s){
             boolean r = s.getPeso() <= this.getCapacidad();
             for(ObjetoMochila e: s.elements()){
                    if(!r) break;
                    r = s.count(e) <= e.getNumMaxDeUnidades();
             }
             return r;
      }
}
```

Para simplificar el código se han diseñado do métodos privados *(numeroEnteroMaximoDeUnidades, numeroRealMaximoDeUnidades*) que calculan el número máximo de unidades del objeto actual (objeto ubicado en el índice I) que podemos colocar en la mochila.

Igualmente se han diseñado método que nos dan la cotaSuperior y cotaInferior del valor de propiedad objetivo asumiendo tomada una alternativa o sin asumir.

También se ha diseñado métodos que no indican si hay alternativa para pasar de un problema a otro. El subproblema que obtenemos tras escoger un alternativa o la alternativa, si la hay que nos permite pasar de un problema otro.

Como el problema de la Mochila es un problema de Reducción también podríamos resolverlo con el Algoritmo A*. Para ello tenemos que diseñar un grafo virtual y ajustar el grafo para que pueda ser resuelto por el algoritmo A*. Para ello diseñamos las clases *MochilaGrafo* y *MochilaVertex*.

La última versión de este tipo puede encontrarse en el API.

```
public class MochilaGrafo
             extends UndirectedSimpleVirtualGraph<ProblemaMochilaVertex,
                    DefaultSimpleEdge<ProblemaMochilaVertex>>
             implements AStarGraph<ProblemaMochilaVertex,
                    DefaultSimpleEdge<ProblemaMochilaVertex>> {



      public static MochilaGrafo create(EdgeFactory<ProblemaMochilaVertex,
               DefaultSimpleEdge<ProblemaMochilaVertex>> edgeFactory) {
             return new MochilaGrafo(edgeFactory);
      }

      public static MochilaGrafo create(EdgeFactory<ProblemaMochilaVertex,
                    DefaultSimpleEdge<ProblemaMochilaVertex>> edgeFactory,
```

```
			ProblemaMochilaVertex[] vs) {
				return new MochilaGrafo(edgeFactory, vs);
		}

		private MochilaGrafo(EdgeFactory<ProblemaMochilaVertex,
				DefaultSimpleEdge<ProblemaMochilaVertex>> edgeFactory,
			ProblemaMochilaVertex[] vs) {
				super(edgeFactory, vs);
		}

		private MochilaGrafo(EdgeFactory<ProblemaMochilaVertex,
				DefaultSimpleEdge<ProblemaMochilaVertex>> edgeFactory) {
			super(edgeFactory);
		}

		@Override
		public double getVertexWeight(ProblemaMochilaVertex vertex) {
			return 0;
		}

		@Override
		public double getVertexWeight(ProblemaMochilaVertex vertex,
				DefaultSimpleEdge<ProblemaMochilaVertex> edgeIn,
				DefaultSimpleEdge<ProblemaMochilaVertex> edgeOut) {
			return 0;
		}

		@Override
		public double getWeightToEnd(ProblemaMochilaVertex startVertex,
				ProblemaMochilaVertex endVertex,
				Predicate<ProblemaMochilaVertex> goal,
				Set<ProblemaMochilaVertex> goalSet) {
			ProblemaMochila actual = startVertex.getProblema();
			return -actual.getCotaSuperiorValorEstimado();
		}

		@Override
		public double getEdgeWeight(DefaultSimpleEdge<ProblemaMochilaVertex> e){
			ProblemaMochila source = e.getSource().getProblema();
			ProblemaMochila target = e.getTarget().getProblema();
			Integer a = source.getAlternativa(target);
			return -a*ProblemaMochila.getValorObjeto(source.getIndex());
		}
}
```

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

		public static MochilaVertex create() {
			return new MochilaVertex(ProblemaMochila.create());
		}

		public static MochilaVertex create(ProblemaMochila problema) {
			return new MochilaVertex(problema);
		}
```

```
private ProblemaMochila problema;

private MochilaVertex(ProblemaMochila problema) {
	super();
	this.problema = problema;
}

public static SolucionMochila getSolucion(List<MochilaVertex> ls){
	SolucionMochila s = SolucionMochila.create();
	for(int i = 0; i < ls.size()-1; i++){
		int index1 = ls.get(i).problema.getIndex();
		ProblemaMochila p1 = ls.get(i).problema;
		ProblemaMochila p2 = ls.get(i+1).problema;
		ObjetoMochila ob = ProblemaMochila.getObjeto(index1);
		Integer a = p1.getAlternativa(p2);
		if(a>0){
			s = s.add(ob, a);
		}
	}
	return s;
}

@Override
public Set<MochilaVertex> getNeighborListOf() {
	return problema.getAlternativas()
		.<ProblemaMochila>mapToObj(
			(int a)-> this.problema.getSubProblema(a))
		.<MochilaVertex>map(
				(ProblemaMochila p)->MochilaVertex.create(p))
		.collect(Collectors.<MochilaVertex>toSet());
}

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

@Override
public boolean isNeighbor(MochilaVertex e) {
	return this.problema.hayAlternativa(e.problema);
}

public ProblemaMochila getProblema() {
	return problema;
}

@Override
public String toString() {
	return problema.toString();
}
```

```
}
```

11. Representación del grafo de problemas y alternativas

En muchos casos es interesante mostrar gráficamente el conjunto de problemas, las alternativas y los subproblemas a los que conducen.

Hemos decidido representar los problemas mediante un rectángulo con una etiqueta que será la devuelta por el toString del problema. Los problemas que no tengan solución los representamos dentro de un rombo.

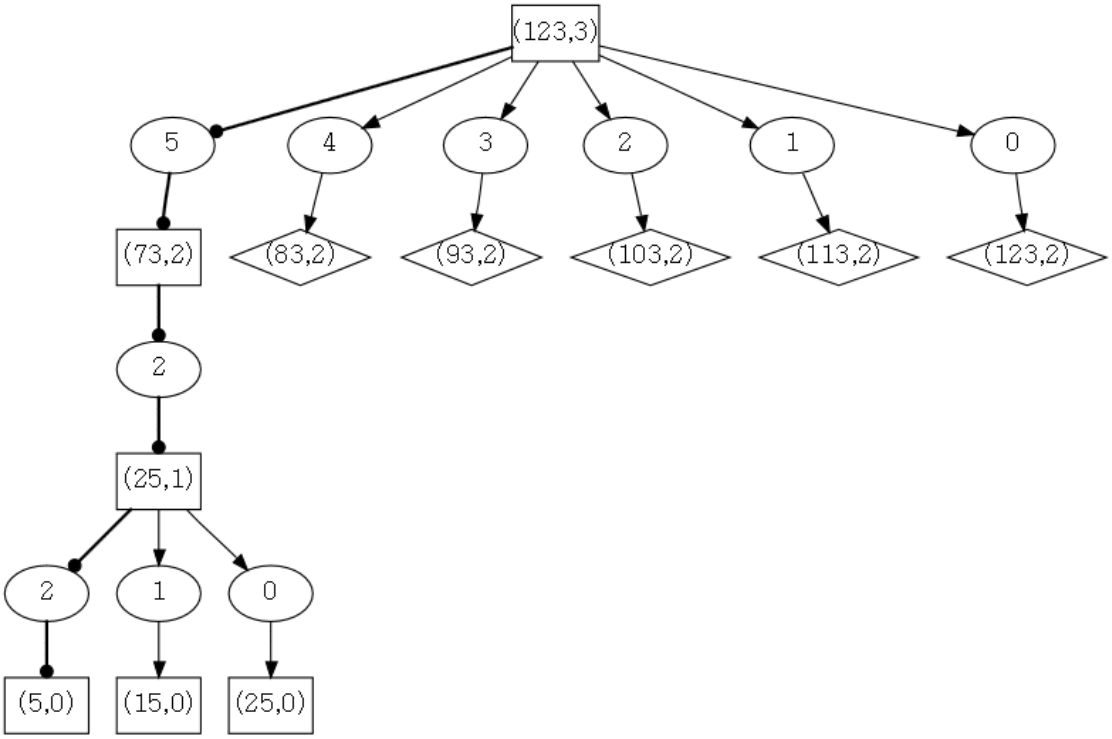
Las alternativas las representamos mediante un círculo con una etiqueta que es su valor.

Para conseguir ese objetivo generamos un fichero intermedio que puede ser procesado por las herramientas que vimos en el capítulo de grafos.

Asumiendo que en el problema de la mochila con capacidad 123 y los objetos disponibles

```
[<1,24,15,2>, <3,24,10,7>, <0,60,24,2>, <2,25,10,5>]
```

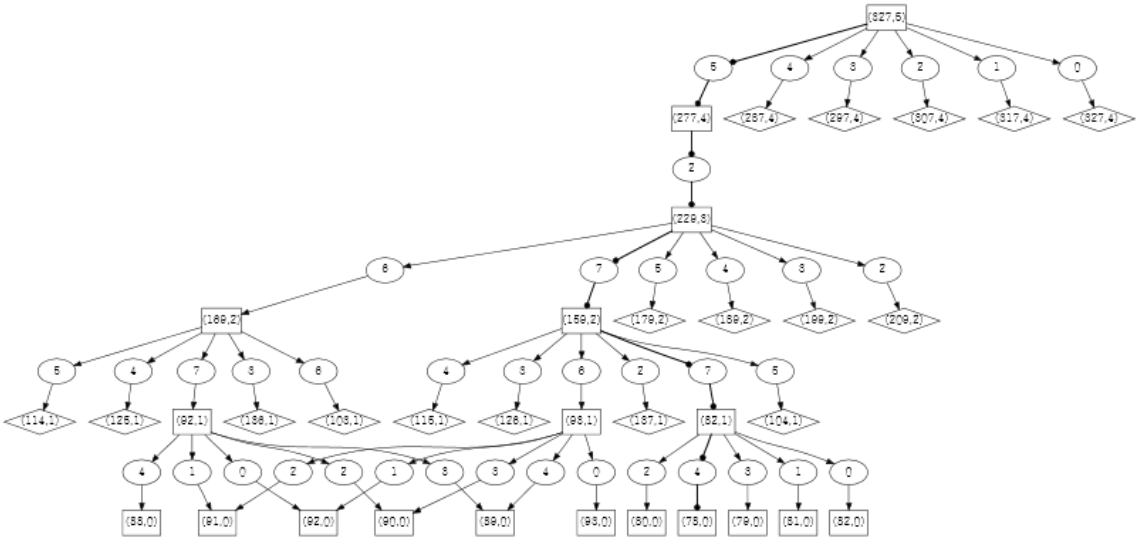
El grafo resultante es:



Si planteamos el problema de la mochila con capacidad 327 y con los objetos disponibles

[<1,24,15,2>, <5,2,1,4>, <4,24,11,7>, <3,24,10,7>, <0,60,24,2>, <2,25,10,5>]

El grafo resultante es:



El código para generar el fichero a partir de la memoria del problema de programación dinámica es:

```
public void showAllGraph(String nombre,String titulo,ProblemaPD<S,A,E> pd){
		setFile(nombre);
		file.println("digraph "+titulo+" {  \n size=\"100,100\"; ");

		showAll(pd);
		file.println("}");
}

private void marcarEnSolucion(ProblemaPD<S,A,E> pd){
	if(solucionesParciales.containsKey(pd)){
		Sp<A,E> e = solucionesParciales.get(pd);
		if(e!=null){
			e.estaEnLaSolucion =true;
			A alternativa = e.alternativa;
			for(int i = 0;
				  i < pd.getNumeroSubProblemas(alternativa); i++){
				ProblemaPD<S,A,E> pds=
					pd.getSubProblema(alternativa,i);
				marcarEnSolucion(pds);
			}
		}
	}
}

private String problema(ProblemaPD<S,A,E> p, Sp<A,E> e){
	String s= "    "+\""+p+"\"";
	if(e!=null){
		s = s+" [shape=box]";
	} else{
		s = s+" [shape=diamond]";
```

```
        }
        return s+";";
}


private String alternativa(ProblemaPD<S,A,E> p, A alternativa){
        String s = "    "+\""+p+","+alternativa+"\""+"
                        [label="+alternativa+"]";
        return s+";";
}

private String aristaProblemaToAlternativa(ProblemaPD<S,A,E> p,
                    A alternativa, Sp<A,E> e){
        String s = "    "+\""+p+"\""+" -> "+"\""+p+","+alternativa+"\"";
        if(e.estaEnLaSolucion && e.alternativa.equals(alternativa)){
                s = s+ "[style=bold,arrowhead=dot]";
        }
        return s+";";
}

private String aristaAlternativaToProblema(ProblemaPD<S,A,E> p,
                  A alternativa, ProblemaPD<S,A,E> ps, Sp<A,E> e){
        String s = "    "+\""+p+","+alternativa+"\""+" ->
                    "+\""+ps+"\"";
        if(e.estaEnLaSolucion && e.alternativa.equals(alternativa)){
                s = s+ "[style=bold,arrowhead=dot]";
        }
        return s+";";
}

private void showAll(ProblemaPD<S,A,E> p){
        marcarEnSolucion(p);
        for(ProblemaPD<S,A,E> pd:solucionesParciales.keySet()){
                Sp<A,E> e = solucionesParciales.get(pd);
                if(e!=null)file.println(problema(pd,e));
                if(e!=null && e.alternativas!=null){
                        for(A alternativa:e.alternativas){
                           for(int i = 0; i <
                                        pd.getNumeroSubProblemas(alternativa); i++){
                               ProblemaPD<S,A,E> pds=
                                            pd.getSubProblema(alternativa,i);
                               if(solucionesParciales.get(pds)==null)
                                             file.println(problema(pds,null));
                               file.println(alternativa(pd,alternativa));
                               file.println(aristaProblemaToAlternativa(
                                        pd,alternativa,e));
                                file.println(aristaAlternativaToProblema(
                                        pd,alternativa,pds,e));
                           }
                        }
                }
        }
}
```

La última versión de este tipo puede encontrarse en el API.

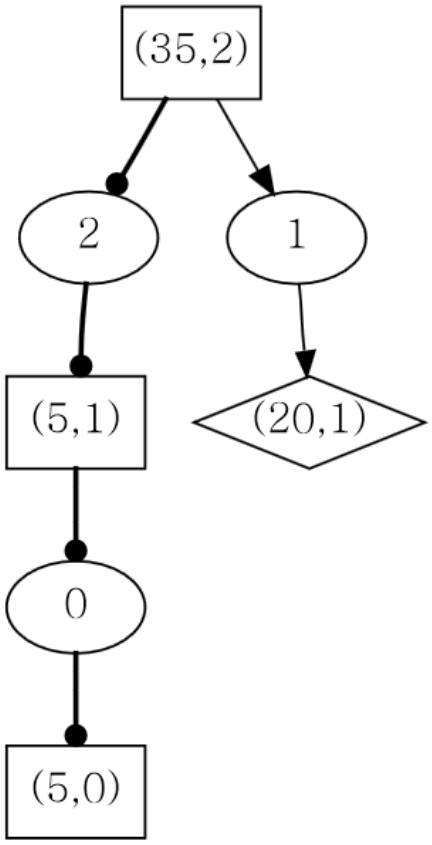
La idea ha sido ir generando las definiciones de los vértices del grafo a dibujar: se han incluido los problemas y las alternativas. Posteriormente se han generado las aristas entres ellos.

Como identificador único en el fichero de cada vértice de tipo problema se ha usado so *toString*. El identificador único de los vértices de tipo alternativa ha sido la concatenación del *toString* del problema más el valor de la alternativa.

Para un problema de la mochila con capacidad 35 y con los objetos disponibles

```
[<2,1,1,10>, <0,60,24,2>, <1,74,15,2>]
```

Y el grafo asociado



Siendo el fichero generado:

```
digraph Mochila {
    size="100,100";
    "(35,2)" [shape=box];
    "(35,2),2" [label=2];
    "(35,2)" -> "(35,2),2"[style=bold,arrowhead=dot];
    "(35,2),2" -> "(5,1)"[style=bold,arrowhead=dot];
    "(20,1)" [shape=diamond];
    "(35,2),1" [label=1];
    "(35,2)" -> "(35,2),1";
    "(35,2),1" -> "(20,1)";
    "(5,1)" [shape=box];
    "(5,1),0" [label=0];
```

```
    "(5,1)" -> "(5,1),0"[style=bold,arrowhead=dot];
    "(5,1),0" -> "(5,0)"[style=bold,arrowhead=dot];
    "(5,0)" [shape=box];
}
```

12. Problemas que se resuelven con Programación Dinámica

Resolvemos en este apartado varios problemas de Programación Dinámica. Para cada uno de ellos ofrecemos una ficha que resume las principales características del mismo. También ofrecemos la ficha para los problemas que hemos visto previamente aunque no repetimos la explicación anterior. La ficha del problema es un elemento valioso para concretar muchos detalles del algoritmo sin entrar en detalles de código.

12.1 Problema del Cambio de Monedas

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

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

El tipo *SolucionMoneda* es un *Multiset<Moneda>* con las propiedades adicionales *Valor* (entero, consultable) de las monedas incluidas en la solución, *NúmeroDeMonedas* (entero, consultable) incluidas en la solución. Podemos diseñar una solución parcial del problema con una lista de enteros que representan las opciones tomadas y dos propiedades adicionales *Valor* y *NumeroDeMonedas.*

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

Es muy usual que para resolver un problema por esta técnica haya que generalizarlo en primer lugar. Hay varias formas posibles de hacer la generalización. Una forma es resolver el problema planteado dada una cantidad *N* con un subconjunto de las monedas del sistema monetario. Las monedas disponibles, aunque es un conjunto, las hemos representado por su forma normal, una lista ordenada sin repeticiones. Una moneda concreta la podemos representar por su índice en *Monedas*. Un subconjunto de monedas por otro índice *J* que representa el subconjunto de monedas cuyos índices son menores o iguales a *J*.

El problema generalizado tiene las propiedades individuales: C, *Cantidad (*entero, consultable*), J (*entero en [0,|*Monedas*|), consultable). En este contexto dos problemas son iguales si tienen iguales sus propiedades individuales: *C, J.*

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

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

Todas las ideas anteriores están resumidas en la Ficha siguiente:

Cambio de Monedas	
Técnica: Programación Dinámica	
Tamaño: J	
Propiedades Compartidas	*M, List <Moneda>, ordenada de mayor a menor* *N = M.size()*
Propiedades Individuales	*C, entero no negativo* *J, entero en [0,N)*
Solución: SolucionMoneda	
SolucionParcial: (a,n)	
Objetivo: Encontrar (m,v,n) *tal que* $v = c$ *y* n *tenga el menor valor posible*	
Alternativas: $A_{c,j} = \{r..0\},\ r = \frac{c}{v_j}$,	
Instanciación $Inicial\ =\ cm(c,0)$	
$cm(c,j)\ = \begin{cases} (0,0), & c = 0 \\ (n,n), & n = \frac{c}{v_j}, n = cv_j, j = N-1 \\ \perp, & n = \frac{c}{v_j}, n \neq cv_j, j = N-1 \\ sA_{a \in A_{c,j}}\big(cS(a, cm(c - av_j, j+1))\big), & en\ otro\ caso \end{cases}$	
$cS\big(a,(a_1,n)\big) = (a, n+a)$: Añade a unidades de la moneda m_j a la solución s	
$sA_{a \in A_{c,j}}(s_a^{m,v,n})$: Elige la solución con menor *n*	
Solución Reconstruida $sr((a,v)) = \begin{cases} (), & c = 0 \\ n \times m_j, & n = \frac{c}{v_j}, n = cv_j, j = N-1 \end{cases}$ $sr((a,v),s_1)\begin{cases} s_1, & a = 0 \\ s_1 + a \times m_j, & a \neq 0 \end{cases}$	
Complejidad	

Los operadores *sA* y *cS* toman un parámetro adicional que no hemos explicitado. Este parámetro adicional es el problema actual. En la medida que tanto *sA* como *cS* se implementan como métodos de los objetos que representan los problemas ese parámetro adicional está implícito.

Un tema importante es el orden con el que tomamos las diferentes alternativas. En algunos problemas puede ser relevante. En este caso las alternativas las tomamos de mayor a menor. Es decir primero la de valor *r*.

Otro tema importante son los casos base considerados. En este caso hemos considerado tres casos base. Los dos importantes son el 2 y el 3. El 1 es un caso particular del 2 que añadimos para considerar la posibilidad de *c = 0* para cualquier valor de *j.* Como es un problema de igualdad (es decir hay que devolver un conjunto de monedas de valor igual a la cantidad c indicada) entonces debemos considerar los casos 2 y 3. En el caso 3 indicamos que no hay solución porque la cantidad c no es múltiplo del valor de la moneda m_o.

El problema se puede resolver con la técnica de *Programación Dinámica con Filtro*. Habría que añadir dos propiedades más: una individual, el número de unidades acumulado, y otra compartida, el número mínimo de unidades encontrado. Las alternativas habría que tomarlas de mayor a menor, las monedas ordenarlas de menor a mayor valor y encontrar una cota inferior del número de unidades que faltarían para completar la cantidad c. La cota la podemos obtener tomando un número fraccionario de monedas siempre ordenadas de menor a mayor valor unitario. Lo dejamos como ejercicio pero planteamos el problema siguiente con esta técnica.

12.2 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 sub-secuencia 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 sub-secuencia común de *X* e *Y* si es sub-secuencia de *X* y sub-secuencia de *Y*. Deseamos determinar la sub-secuencia de longitud máxima común a dos secuencias dadas.

Ejemplo: Si *X = { a, **b, c, b**, d, **a**, b }* e *Y = { **b**, d, **c**, a, **b, a** }* tendremos como sub-secuencia más larga a *Z = { b, c, b, a }*.

El problema que nos planteamos es: dadas dos secuencias $X_m = \{x_0, x_1, \dots, x_{m-1}\}$ y $Y_n = \{y_0, y_1, \dots, y_{n-1}\}$ encontrar otra secuencia $Z_k = \{z_0, z_1, \dots, z_{k-1}\}$ que sea la secuencia común a ambas más larga posible.

Como vemos los datos del problema podemos modelarlos como listas de un tipo genérico *E*. La solución buscada es también una lista del mismo tipo *E*. La solución tiene las propiedades *Z* (*List<E>*, consultable) y otras propiedades y métodos adecuados para gestionar la lista anterior: *Empty, Size, Last, add.* Una solución la representamos por *z* y por z_k el valor que ocupa la posición *k*. La lista vacía la representamos por ϕ.

Como anteriormente generalizamos el problema. El problema generalizado pretende encontrar las sub-secuencia común más larga entre los prefijos de *X* e *Y* que incluyen las casillas menores o iguales a *I*, *J* respectivamente. Las propiedades compartidas del conjunto de problemas son: *X, List<E>*, consultable e *Y, List<E>*, consultable. Las propiedades individuales de cada problema son: *I (*entero *[0,|X|)*, consultable*), J (*entero en *[0,|Y|)*, consultable). En este contexto dos problemas son iguales si tienen iguales sus propiedades individuales: *I, J.*

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 que disminuyamos el índice de una u otra lista.

El objetivo del problema es encontrar una solución *Z* que siendo una sub-secuencia común a *X* e *Y* tenga la longitud máxima.

Todas las ideas anteriores están resumidas en la Ficha siguiente:

<table>
<tr><th colspan="2">Subsecuencia común más larga</th></tr>
<tr><td colspan="2">Técnica: Programación Dinámica</td></tr>
<tr><td colspan="2">Tamaño: I+J</td></tr>
<tr><td>Propiedades Compartidas</td><td>X, List <E>
Y, List<E></td></tr>
<tr><td>Propiedades Individuales</td><td>I, entero no [0,X.Size)
J, entero en [0,Y.Size)</td></tr>
<tr><td colspan="2">Solución: Z</td></tr>
<tr><td colspan="2">Objetivo: Encontrar z tal |z| tenga el mayor valor posible</td></tr>
<tr><td colspan="2">Alternativas: si $x_i \neq y_j\ A_{i,j} = \{X, Y\}, si\ x_i = y_j\ A_{i,j} = \{C\}$</td></tr>
<tr><td colspan="2">$psc(X, Y) = sc(|X|\text{-}1, |Y|\text{-}1)$
$sc(i,j) = \begin{cases} \theta, & i < 0\ ||\ j < 0 \\ sA_{a \in A_{i,j}}\left(c\left(C, sc(i-1, j-1)\right)\right), & x_i = y_j \\ sA_{a \in A_{i,j}}\left(c\left(A, sc(i-1, j)\right), c\left(B, sc(i, j-1)\right)\right), & x_i \neq y_j \end{cases}$</td></tr>
<tr><td colspan="2">$c(A, s)$: Devuelve s
$c(B, s)$: Devuelve s
$c(C, s)$: Devuelve $s + x_i$</td></tr>
<tr><td colspan="2">$sA_{a \in A_{c,j}}(s_a)$: Elige la solución con mayor $|s|$</td></tr>
<tr><td>Complejidad</td><td></td></tr>
</table>

12.3 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 el producto de las mismas. 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. Sean $n_{0,s}$ y $n_{s,k}$ el número de multiplicaciones necesarias para multiplicar los grupos de matrices de *0* a *s-1* y de *s* a *k-1* respectivamente. El producto total $(m_0 m_1, \dots, m_{s-1})(m_s, \dots, m_{k-1})$ requiere $n_{0,s} n_{s,k} + f_0 f_s c_{k-1}$. Es decir el número de multiplicaciones necesarias para cada subgrupo más las que necesitamos para multiplicar las dos matrices resultantes. Hemos de tener en cuenta que la

matriz resultante del subgrupo izquierdo tendrá f_0 filas y f_s columnas y el subgrupo derecho f_s filas y c_{k-1} columnas.

El problema generalizado pretende encontrar el número mínimo de multiplicaciones necesarias para multiplicar el grupo de matrices comprendido en el intervalo *[I,J)*. Para un problema de ese tipo las alternativas disponibles están en colocar el punto donde colocar un paréntesis en *S* con *I < S ≤ J*.

Además del número de multiplicaciones queremos encontrar la forma de colocar los paréntesis que dé lugar a ese número de multiplicaciones. La solución contendrá, también, la expresión con los paréntesis adecuados. Por lo tanto el tipo *SolucionMatrices* tendrá las propiedades: *ExpresionConParentesis*, *String*, consultable, *NumeroDeOperaciones*, entero. La expresión con paréntesis representa la forma concreta de operar las matrices que están en las posiciones indicadas. Así por *(1,(2,3))* representaremos el producto de las matriz en la posición *1* por el resultado de multiplicar las que están en las posiciones *2* y *3*. Las soluciones parciales serán de la forma (a, n) dónde a es la alternativa elegida y n el número de multiplicaciones. La solución reconstruida será una expresión del tipo *(1,(2,3))* donde se indica el orden de multiplicación resultante junto con el número de multiplicaciones. Es decir de la forma *(e,n).*

El objetivo del problema es encontrar una solución para la cual el número de multiplicaciones sea mínimo.

Todas las ideas anteriores están resumidas en la Ficha siguiente:

<table>
<tr><th colspan="2">Multiplicación de Matrices Encadenadas</th></tr>
<tr><td colspan="2">Técnica: Programación Dinámica</td></tr>
<tr><td colspan="2">Tamaño: J-I</td></tr>
<tr><td>Propiedades Compartidas</td><td>M, List <Matriz>
N = M.size()</td></tr>
<tr><td>Propiedades Individuales</td><td>I, entero no [0,M.Size)
J, entero en [I+1,M.Size]</td></tr>
<tr><td colspan="2">Solución: (e,n) de tipo SolucionMatrizEncadenada</td></tr>
<tr><td colspan="2">SolucionParcial: (a,n)</td></tr>
<tr><td colspan="2">Objetivo: Encontrar $r = (a, n)$ tal que n sea mínimo</td></tr>
<tr><td colspan="2">Alternativas: $A = (i, j]$</td></tr>
<tr><td colspan="2">Instanciación

Inicial = m(0,N)</td></tr>
<tr><td colspan="2">Problema Generalizado
$m(i,j) = \begin{cases} (i,0), & j-i=1 \\ (i, f_i c_i c_{i+1}), & j-i=2 \\ sA_{s\in A_{i,j}}\big(cS(a, m(i,a), m(a,j)\big), & j-i>2 \end{cases}$</td></tr>
<tr><td colspan="2">$cS(a,(a_1,n_1),(a_2,n_2))$: Devuelve con (a, n), $n = n_1 + n_2 + f_i f_a c_j$</td></tr>
<tr><td colspan="2">$sA_{s\in A_{i,j}}(r_s)$: Elige la solución con menor n</td></tr>
<tr><td colspan="2">Solución Reconstruida</td></tr>
</table>

$sr((a,n)) = \begin{cases} (i,n), & j-i=1 \\ ((i,j),n), & j-i=2 \end{cases}$ $sr((a,n),e_1,e_2) = ((e_1,e_2),n)$
Complejidad

13. Problemas propuestos

Ejercicio 1

Dadas dos cadenas de caracteres **X** e **Y**, se desea transformar X en Y usando el menor número de operaciones básicas, que pueden ser de tres tipos: eliminar un carácter, añadir un carácter, y cambiar un carácter por otro. Por ejemplo, para transformar la cadena "*carro*" en la cadena "*casa*" se puede proceder de la siguiente forma:

- "*carr*" (eliminando *o* de la posición 5), "*cara*" (cambiando *r* por *a* en la posición 4) y "*casa*" (cambiando *r* por *s* en la posición 3)

Esta transformación es óptima (la hemos escogido a propósito), pero no tiene por qué ser siempre así (piénsese por ejemplo en borrar X y copiar Y). Se pide obtener un algoritmo basado en *Programación Dinámica* que calcule la matriz OB[0..long(X), 0..long(Y)], que contiene en su posición (*i*, *j*) el número mínimo de operaciones básicas para transformar la cadena formada por los primeros *i* caracteres de X en la cadena formada por los primeros *j* caracteres de Y. Obtener además un algoritmo que, a partir de la matriz OB, reconstruya la solución, indicando las instrucciones que han de ejecutarse para reemplazar X por Y.

Ejercicio 2

Escríbase un algoritmo utilizando la técnica de *programación dinámica*, que recibiendo como entrada un natural n, con n ≥ 0, devuelva el número de árboles binarios de n elementos topológicamente distintos (de formas distintas), que hay. Por ejemplo, si f es la función que nos da la solución, se tendrá f(0) = 1, f(1) = 1, f(2) = 2, f(3) = 5, f(4) = 14, etc.

Ejercicio 3

Se tiene una cantidad de dinero a invertir M, de la cual se desea obtener el máximo beneficio. Para ello se dispone de ofertas de *n* bancos, correspondientes cada una de ellas a una función de interés *fi* (*x*) (*i* = 1..n) dependiente del capital invertido, que representa la cantidad de dinero que se obtiene al invertir en el banco *i* la cantidad *x*. Las funciones de interés cumplen que si *x* > *y*, entonces *fi* (*x*) ≥ *fi* (*y*). En general no existe una oferta mejor que ninguna otra para cualquier cantidad, por lo que el beneficio máximo deberá buscarse en un reparto adecuado de la cantidad total a invertir entre todas las ofertas. Obtener mediante ***Programación Dinámica*** un programa que determine el máximo beneficio que puede obtenerse al invertir la

cantidad total M mediante un reparto adecuado entre las ofertas de los *n* bancos considerados. ¿Cómo puede determinarse la cantidad a invertir en cada banco?
Nota: considerar que las cantidades destinadas a cada inversión son valores enteros.

Ejercicio 4

Una red de aeropuertos desea mejorar su servicio al cliente mediante terminales de información. Dado un aeropuerto *origen* y un aeropuerto *destino*, el terminal desea ofrecer la información sobre los vuelos que hacen la conexión y que minimizan el tiempo del trayecto total. Se dispone de una tabla de tiempos *tiempos[1..N,1..N,0..23]* de valores enteros, cuyos elementos *tiempos[i,j,h]* contiene el tiempo que se tardaría desde el aeropuerto *i* al *j* en vuelo directo estando en el aeropuerto *i* a las *h* horas (el vuelo no sale necesariamente a las *h* horas).

Mediante *programación dinámica*:

a) Realizar la función que obtiene el tiempo mínimo de trayecto total entre el aeropuerto *origen* y *destino* si nos encontramos en el aeropuerto *origen* a una hora *h* dada, cuyo prototipo es

```
func vuelo(origen, destino, h: Entero) dev (t: Entero)
```

b) Explicar cómo se encontraría la secuencia de aeropuertos intermedios del trayecto de duración total mínima entre el aeropuerto *origen* y *destino*, estando en el aeropuerto *origen* a la hora *h*.

c) Realizar la función que obtiene la hora *h* a la que habría que partir del aeropuerto *origen* para que el tiempo de trayecto total hasta el aeropuerto *destino* sea **mínimo**, cuyo prototipo es

```
func hora(origen, destino: Entero) dev (h: Entero)
```

NOTAS:
- No hay diferencia horaria entre los aeropuertos.
- Los retardos en los transbordos ya están incluidos en los datos.
- No se deben tener en cuenta situaciones imprevisibles como retardos en vuelos.
- Todos los días el horario de vuelos en cada aeropuerto es el mismo.

Ejercicio 5

Se desea obtener el número de ordenaciones posibles $f(n)$ de n elementos usando las relaciones "<" y "=". Por ejemplo, para tres elementos, se tienen 13 ordenaciones posibles:

a = b = c a = b < c c < a = b a < b = c a = c < b a < b < c a < c < b
c < a < b b < a = c b = c < a b < a < c b < c < a c < b < a

Apoyándose en la cantidad $g(n, d)$ "número de ordenaciones posibles de n elementos, siendo d el número de elementos distintos", obtenga un algoritmo de Programación Dinámica que calcule el número de ordenaciones posibles de n elementos. Justifique la respuesta mediante las relaciones correspondientes.

Ejercicio 6

Sea la matriz L[1..N] tal que L[*i*, *j*] = *verdadero* si existe una arista entre los vértices *i* y *j* de un grafo dirigido G, y L[*i*, *j*] = *falso* en caso contrario. Deseamos calcular una matriz D tal que D[*i*, *j*] = *verdadero* si existe al menos un camino desde *i* hasta *j*, y D[*i*, *j*] = *falso* en caso contrario.

a) Escribir un algoritmo que use la técnica de ***Programación Dinámica*** para calcular la matriz D (encontrar previamente la ecuación en recurrencias para D[*i*, *j*]).
b) Escribir un algoritmo que permita obtener un camino desde un vértice *i* del grafo hasta otro *j*, en caso de existir.

Ejercicio 7

La circulación viaria de una ciudad viene representada por un grafo dirigido en el que los vértices se corresponden con intersecciones de calles y las aristas con las propias vías de tráfico, de manera que se dispone de la anchura, en número de carriles, de cada vía. Así, A[*i*, *j*] representa el número de carrilles de la vía que une la intersección *i* con la intersección *j*, en el sentido "desde *i* a *j*" (0 si no hay una vía que una directamente a las dos intersecciones en el sentido indicado). Se define la anchura de un trayecto entre dos intersecciones a la correspondiente al tramo de menor anchura. Aplicar la técnica de Programación Dinámica para:

a) implementar un algoritmo que obtenga la anchura correspondiente al trayecto de mayor anchura entre cada par de intersecciones.
b) implementar un algoritmo que obtenga el trayecto de mayor anchura desde una intersección *origen* hasta otra *destino*.