Capítulo 7

Programación Dinámica.

7.1. Introducción.

En el método *Divide y Vencerás* se detalló como es posible descomponer un problema en subproblemas del mismo carácter que el original, y que éstos podían seguir descomponiéndose hasta obtener subproblemas ya resueltos, y como combinando las soluciones de esos subproblemas se podía obtener la solución del problema original. A veces los subproblemas resultantes son idénticos lo cual da lugar a cálculos repetitivos que merman la eficiencia del algoritmo. En cambio, si se resuelve cada subproblema distinto una sola vez y se conserva su resultado, se obtiene un algoritmo mucho más eficiente.

Una de las idea que subyace en la programación dinámica está enfocada en este aspecto, no calcular dos veces lo mismo. Para ello, se puede usar una tabla de resultados que se va rellenando a medida que se van resolviendo los subproblemas, y comprobar los valores de la tabla antes de resolver un subproblema para ver si ya se había resuelto anteriormente. Según este enfoque, la programación dinámica es un método ascendente, ya que se resuelven primero los subproblemas más pequeños y por tanto más simples, y combinando sus soluciones se obtienen las soluciones de subproblemas sucesivamente más grandes hasta llegar a la del problema original. Un ejemplo de repetición de llamadas se analizó cuando se trató el algoritmo recursivo para obtener el término n de la sucesión de Fibonacci.

Otro enfoque de la programación dinámica se emplea en problemas de optimización que satisfacen el principio de optimalidad de Bellman. Usando este principio, se obtiene la solución de un problema a partir de ciertas propiedades de sus subsoluciones. Este principio no siempre es aplicable y se cumple cuando toda subsolución de una solución óptima también es óptima. Un ejemplo que cumple el principio de optimalidad de Bellman podría ser el siguiente:

Si el camino más corto para ir de Córdoba a Barcelona pasa por Castellón, entonces la subsolución que conduce de Córdoba a Castellón también es óptima,

al igual que la subsolución que conduce de Castellón a Barcelona. Esto se puede demostrar fácilmente ya que si el camino óptimo para ir de Córdoba a Castellón no fuese el subcamino de la solución óptima, esa solución no podría ser óptima, ya que existiría una forma más corta de recorrer la primera parte del camino.

Como se acaba de comentar, este principio aunque parece evidente, no siempre es aplicable. Vamos a ver un par de casos relacionados con el mismo problema, que no cumplen el principio de optimalidad.

- Supongamos que queremos obtener el camino más rápido entre Córdoba y Barcelona. Si éste pasa por Castellón, podría pensarse que lo mejor sería ir lo más rápido posible hasta Castellón y después ir lo más rápido posible desde Castellón a Barcelona. La idea no sería buena ya que al ir muy rápido desde Córdoba a Castellón, podríamos agotar la gasolina antes y tener que repostar entre Castellón y Barcelona, con lo cual perderíamos tiempo y probablemente no sería la solución más rápida. Esto sucede ya que los subcaminos no son independientes, como ocurría en el caso anterior, ya que comparten un recurso, y por tanto la selección óptima de un subcamino puede impedir que sea óptima la del resto de subcaminos.

- Supongamos por ejemplo que queremos obtener el camino simple más largo (no se repiten nodos, ya que podríamos entrar en ciclos infinitos) entre Córdoba y Barcelona. Si sabemos que pasa por Castellón, la subsolución que va de Córdoba a Castellón no tiene porque ser la solución del camino más largo para ir de Córdoba a Castellón, ya que la solución a este problema podría contener ciudades que se utilizan en la subsolución que va desde Castellón a Barcelona y por tanto no podrían formar parte de la subsolución ya que estarían repetidos.

La mayor dificultad de este principio estriba en fijar los subproblemas que resuelven las subsoluciones de la solución del problema, de forma tal que en tales subproblemas se cumpla el principio de optimalidad de Bellman. Una vez fijados los subproblemas, y comprobando que cumplen el principio, se puede obtener una solución del problema planteado siguiendo el método general que se explica a continuación. Por ejemplo, en el caso citado habría que comprobar si el subproblema de calcular el camino mínimo de Córdoba a Castellón puede servir para hallar el camino más corto entre Córdoba y Barcelona.

7.2. El método general.

El hecho de usar subsoluciones implica que la solución del problema se pueda dividir en etapas, además de comprobar que la sucesión de subsoluciones óptimas

es la solución óptima del problema, con lo cual se cumpliría el principio de optimalidad. En la aplicación del método se pueden plantear dos posibilidades, resolverlo *hacia delante* o resolverlo *hacia detrás* en función de que las subsoluciones se obtengan en un sentido u otro (desde el origen hasta el destino o desde el destino hasta el origen). En la versión hacia delante, supongamos que el problema parte de una etapa inicial A y tiene que llegar a una etapa final B. Supongamos también que en una primera etapa correspondiente a la primera subsolución del problema, tenemos una serie de posibilidades $C_1, C_2, \ldots, C_n$, y la solución nos lleva a un punto intermedio C_i, desde el cual quedará por resolver el problema desde C_i hasta el destino B. El principio de optimalidad dice que si la solución óptima nos lleva, tras la primera etapa a C_i, la continuación hasta B ha de ser solución óptima del problema de ir de C_i a B. De todo ello se puede deducir que la solución óptima del problema original es la mejor de todas las posibles que se forman considerando las soluciones que como primera etapa nos llevan a C_i, para continuar con la solución óptima de ir de C_i a B, haciendo variar i.

Teniendo en cuenta que todas las soluciones del problema tienen una cantidad de etapas inferior a una constante n, se puede plantear el siguiente algoritmo recursivo:

$$Dinamico(A, B) = Optimo((A, C_i) + Dinamico(C_i, B))$$

Siendo C_i el conjunto de pasos intermedios alcanzables desde A en una sola etapa. Este algoritmo funciona, pero al barrer todas las posibilidades es prácticamente inviable en problemas cuya solución tiene un alto número de etapas. Si por ejemplo todas las posibles soluciones tuviesen n etapas y siempre hubiese k posibilidades al elegir una etapa, el algoritmo emplearía un tiempo de orden k^n. La eficiencia del algoritmo se puede mejorar evitando cálculos idénticos reiterados, por ejemplo si D_j en la segunda etapa, es alcanzable en una etapa desde diversos C_i procedentes de la primera etapa, el algoritmo recursivo calcularía varias veces $Dinamico(D_j, B)$. En definitiva tendríamos el problema descompuesto en tres etapas, cuyas soluciones son:

- Ir de A a C_i en una etapa
- Ir de C_i a D_j en una etapa
- $Dinamico(D_j, B)$

Las dos primeras formarían las soluciones del subproblema de ir de A a D_j, que según el principio de optimalidad solo la mejor de todas las soluciones de este subproblema podría formar parte de la solución óptima del problema para ir de A a B. Por tanto, antes de realizar una llamada recursiva hay que decidir cuales realmente valen la pena. Este razonamiento para dos etapas se puede generalizar para cualquier numero de etapas, lo que implicaría la desaparición de las llamadas recursivas. Como se ha podido apreciar, la idea es ir resolviendo los subproblemas

resolubles en $n+1$ etapas basándonos en las soluciones óptimas de los subproblemas resolubles en n etapas, previamente encontradas. En cuanto al método de resolución *hacia detrás* el razonamiento es simétrico, considerando en primer lugar la última etapa, prosiguiendo el estudio de las soluciones desde el final hasta el principio. A continuación vamos a estudiar algunos problemas cuya resolución se basa en este método.

7.3. Ejemplos.

7.3.1. La competición internacional.

Se tienen dos equipos A y B que se van a enfrentar en una competición varias veces, ganando aquel que consiga n victorias (ejemplo de los playoffs de la NBA) y no es posible un empate en un partido. Evidentemente, se enfrentarán como máximo $2n-1$ veces. Suponemos que p es la probabilidad de que el equipo A gane un partido, por tanto $1-p$ será la probabilidad de que B gane un partido. Se pretende calcular la probabilidad de que el equipo A gane la competición. Para dicha estimación vamos a usar una matriz cuadrada de orden n, en la cual un elemento $P(i,j)$ de la matriz reflejará la probabilidad de que A gane la competición cuando a A le faltan i victorias por conseguir y a B le faltan j victorias. Por ejemplo, antes de comenzar la competición, la probabilidad de que A sea vencedor se reflejará en el elemento $P(n,n)$, ya que a ambos les faltan n victorias para ganar la competición. En dicha matriz hay elementos cuyo valor es conocido a priori, los $P(0,i)=1$, porque si a A le quedan cero victorias por conseguir ya habrá ganado la competición. Por la razón inversa los $P(j,0)=0$, ya que sería imposibe que el equipo A ganase la competición si al B le quedan 0 victorias. El elemento $P(0,0)$ no tiene sentido ya que los dos equipos no pueden ganar la competición a la vez. Teniendo en cuenta que A gana un partido con probabilidad p y lo pierde con probabilidad $1-p$, se puede plantear la siguiente fórmula recursiva para obtener un elemento $P(i,j)$ cualquiera:

$$P(i,j) = pP(i-1,j) + (1-p)P(i,j-1)$$

donde $i,j \geq 1$. La fórmula es lógica, ya que desde desde el estado (i,j) se puede llegar al estado $(i-1,j)$, si el equipo A gana el siguiente partido o al estado $(i,j-1)$ si el equipo B gana el siguiente partido. Por tanto se podría implementar el siguiente algoritmo recursivo para obtener un elemento $P(i,j)$.

```
Algoritmo P(i, j, p)
  inicio
    si i = 0 entonces
      devolver 1
    sino
```

```
        si j = 0 entonces
          devolver 0
        sino
          devolver pP(i − 1, j, p) + (1 − p)P(i, j − 1, p)
        finsi
      finsi
    fin
```

De forma intuitiva, se puede ver que la complejidad del algoritmo es exponencial, ya que se generan dos llamadas recursivas al estilo del algoritmo recursivo para obtener el término general de la sucesión de Fibonacci.

Al igual que en la sucesión de Fibonacci, se puede apreciar que en esta función se producen muchas llamadas idénticas. Para subsanar este inconveniente, se puede tener en cuenta que cada elemento de la matriz se puede obtener a partir del que está justo encima de él en la matriz, y del que está a su izquierda, además de considerar los casos elementales ya conocidos que son los elementos de la primera fila y primera columna. De esta forma, se puede plantear el algoritmo siguiente que obtiene todos los elementos de la matriz rellenándola por filas.

```
Algoritmo Competicion(p, n; ; P)
  inicio
    para s de 1 a n hacer
      P(0, s) ← 1
      P(s, 0) ← 0
    finpara
    para i de 1 a n hacer
      para j de 1 a n hacer
        P(i, j) = pP(i − 1, j) + (1 − p)P(i, j − 1)
      finpara
    finpara
  fin
```

En este caso el tiempo de ejecución el algoritmo será $O(n^2)$, ya que se resuelve recorriendo la matriz resultado por filas y rellenándola durante el recorrido.

7.3.2. Caminos mínimos entre todos los pares de nodos en un grafo (Algoritmo de Floyd).

Este problema ya fue tratado en el módulo de grafos de la asignatura de *Estructuras de Datos*. En este apartado vamos a verificar que la técnica utilizada se basa en el principio de optimalidad de Bellman, y por tanto hace uso de la programación dinámica. Está claro que el principio de optimalidad se cumple, ya que si k es un nodo intermedio en el camino mínimo que va de i a j, entonces la parte del camino que va desde i a k y la que va de k a j también han de ser óptimas. Para aplicar la idea, se toma una matriz cuadrada D, de orden n, siendo n el número de nodos,

en la cual finalmente obtendremos las distancias mínimas entre todos los pares de nodos. Inicialmente en D se almacenarán las distancias directas (sin usar ningún nodo intermedio), entre todos los pares de nodos. En sucesivas iteraciones, iremos comprobando si esas distancias se ven acortadas considerando nuevos nodos intermedios, de forma tal que en la iteración k, obtendremos las distancias más cortas usando única y exclusivamente los nodos $1, 2, 3, \ldots, k$. Al cabo de n iteraciones, en D almacenaremos las distancias mínimas que utilicen algunos de los n nodos como nodos intermedios. En la iteración k, se comprueba para cada par de nodos i, j si la distancia entre ambos se ve acortada usando k como nodo intermedio. En las iteraciones anteriores se ha obtenido la distancia mínima entre todos los pares de nodos tomando como nodos intermedios los $k-1$ primeros. De esta forma en la iteración k se realiza la siguiente acción:

$$D_k(i,j) = min\{D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j)\}$$

en la cual se está usando el principio de optimalidad para calcular el camino más corto que va de i a j pasando por k.

7.3.3. Problema del cambio (versión 2).

Otro ejemplo típico es el problema del cambio. Este problema ya se resolvió mediante un algoritmo voraz que desafortunadamente solo funciona bien en casos concretos (sistemas monetarios perfectos). En el caso de ciertos valores de monedas, o en caso de que el número de monedas de cada tipo no sea ilimitado, no proporciona una solución óptima. Por ejemplo si en un sistema tuviésemos monedas de 1, 4, y 5 céntimos y tuviésemos que devolver un cambio de 8 céntimos, la solución que proporcionaría el algoritmo voraz sería 1 moneda de 5 céntimos, y 3 de 1 céntimo. Sin embargo la solución óptima serían dos monedas de 4 céntimos.

Para resolver este problema usando el método de la programación dinámica, tendríamos descomponer el problema en subproblemas más fáciles de resolver y cuya solución fuese óptima; y a partir de estas subsoluciones óptimas obtener la solución óptima del problema de partida. Para ello, se podría generar una tabla donde fuesen apareciendo los resultados de los subproblemas intermedios útiles y que se combinarán para obtener la solución del problema inicial.

Supongamos que el sistema monetario considerado tiene n monedas distintas y que la moneda i-ésima con $1 \leq i \leq n$ tiene un valor $v_i > 0$. Suponemos que hay un suministro ilimitado de monedas de cada tipo. Supongamos que al cliente hay que darle un cambio de valor N unidades monetarias empleando el menor número de monedas. En este caso se generaría una tabla c de n filas y $(N+1)$ columnas, donde cada fila se corresponde con un tipo de moneda y cada columna contendría los valores que van desde 0 hasta N unidades monetarias. Cada elemento i, j de la tabla será el mínimo número de monedas utilizadas para obtener un cambio j

usando solamente monedas que van desde la 1 hasta la i-ésima (ordenadas en valor creciente). Si indexamos como en C++, las filas y columnas desde 0, la solución del problema estará dada por el elemento de la matriz $c(n-1, N)$.

Para rellenar la tabla, es evidente que se puede inicializar con 0 en la primera columna. Después, la tabla se puede rellenar o bien por orden creciente de filas (rellenando cada fila de izquierda a derecha) o por orden creciente de columnas (rellenando cada columna de arriba a abajo). Para conseguir una cantidad j cualquiera usando monedas que van de 1 a la i se pueden plantear dos estrategias:

- No utilizando la moneda i, aunque esté permitido, en cuyo caso $c(i, j) = c(i-1, j)$ (se recuerda que c almacena el número de monedas usadas)

- Usar al menos una moneda i, en cuyo caso antes de usar la moneda se tenía la cantidad $j - v_i$ unidades, y después de usar la moneda i el nuevo número de monedas será $1 + c(i, j - v_i)$. Por lo tanto $c(i, j) = 1 + c(i, j - v_i)$.

Como queremos minimizar el número de monedas usadas, se selecciona aquella alternativa que sea mejor. Por lo tanto:

$$c(i, j) = \text{mín}\{c(i-1, j), 1 + c(i, j - v_i)\}$$

De esta forma, se van resolviendo los posibles estados a partir de la solución óptima de estados previamente calculados.

Cuando $i = 0$ o cuando $j < v_i$, alguno de los elementos a comparar se salen de la tabla, y hay que tenerlo en cuenta. La moneda de valor 1 hay que usarla siempre para que haya solución para cualquier cambio, y ésta será usada en la fila 0. En ese caso, los elementos de la fila 0, se rellenan con la expresión $1 + c(0, j - v_0)$ para todo j >0. La tabla 7.1 ilustra el procedimiento para el ejemplo citado.

Cantidad:	0	1	2	3	4	5	6	7	8
$v_0 = 1$	0	1	2	3	4	5	6	7	8
$v_1 = 4$	0	1	2	3	1	2	3	4	2
$v_2 = 5$	0	1	2	3	1	1	2	3	2

Cuadro 7.1: Tabla de secuencias y beneficios para el ejemplo.

Para ver como se rellena la tabla, supongamos que la rellenamos por filas. Los elementos de la columna 0, valen todos 0, y todos los elementos de la fila 0 se rellenan con uno más el valor que queda una posición a su izquierda (debido a que el valor de la moneda 1 es 1) $1 + c(0, j - v_0)$. De ahí que se obtengan los valores $1, 2, 3, 4, \ldots, 8$.

Los elementos de la fila 1 se rellenarán eligiendo el mínimo entre el valor que hay encima de cada elemento ($c(0,j)$) y uno más el valor que queda 4 posiciones a su izquierda (debido a que el valor de la moneda 2 es 4). Hasta la columna 3 el elemento que queda 4 posiciones a la izquierda se sale de la tabla y por tanto se selecciona el elemento que hay encima. El elemento $c(1,4) = min\{c(0,4), 1 + c(1,0)\} = min\{4,1\} = 1$. De la misma manera se rellenarán los elementos $c(1,5), c(1,6), c(1,7)$. El elemento $c(1,8)$ se obtiene como $c(1,8) = min\{c(0,8), 1 + c(1,4)\} = min\{8,2\} = 2$.

Los elementos de la fila 2 se rellenarán eligiendo el mínimo entre el valor que hay encima de cada elemento ($c(1,j)$) y uno más el valor que queda 5 posiciones a su izquierda (debido a que el valor de la moneda 3 es 5). Hasta la columna 4 el elemento que queda 5 posiciones a la izquierda se sale de la tabla y por tanto se selecciona el que hay encima. El elemento $c(2,5) = min\{c(2,5), 1 + c(1,0)\} = min\{2,1\} = 1$. De la misma manera se rellenarán los elementos $c(2,6), c(2,7)$. Por último, el elemento $c(2,8)$ se obtiene como $c(2,8) = min\{c(1,8), 1 + c(2,3)\} = min\{2,3\} = 2$.

Como se puede apreciar, la tabla nos da la solución para todos los casos en los que haya que dar un cambio de 8 unidades o menos, y se pague con las i primeras monedas con $1 \leq i \leq 3$.

El algoritmo podría quedar como sigue:

```
Algoritmo cambio2(v(n), N; ; C(n, N))
  inicio
    Rellenamos columna 0
    para i de 0 a n − 1 hacer
      c(i, 0) ← 0
    finpara
    para i de 0 a n − 1 hacer
      para j de 1 a N hacer
        si i = 0 entonces
        Rellenamos fila 0
          c(i, j) ← 1 + c(i, j − v(0))
        sino
        Rellenamos el resto de filas
          si j < v(i) entonces
            c(i, j) ← c(i − 1, j)
          sino
            c(i, j) ← mín(c(i − 1, j), 1 + c(i, j − v(i))
          finsi
        finsi
      finpara
    finpara
```

fin

En este algoritmo se ha aplicado el principio de optimalidad ya que cuando se calcula $c(i,j)$ como el mínimo de $c(i-1,j)$ y $1+c(i,j-v(i))$ se ha dado por supuesto si $c(i,j)$ proporciona el óptimo del problema para las i primeras monedas y un cambio de j unidades, entonces $c(i-1,j)$ y $c(i,j-v(i))$ también son soluciones óptimas de los problemas que representan.

Ahora quedaría por determinar que cantidad de cada moneda forma parte de la solución. Ello es simple una vez que se tiene la tabla rellena. Como se ha visto $c(n-1,N)$ indica el número total de monedas necesarias para el cambio final, y cada elemento de la tabla $c(i,j)$ indica las monedas necesarias de tipo menor o igual a i para dar un cambio de j unidades. Pues bien, para un elemento cualquiera $c(i,j)$, si $c(i,j) = c(i-1,j)$ indica que no se va a necesitar ninguna moneda de tipo i en el cambio, y pasaríamos a ver el elemento $c(i-1,j)$. En caso de que $c(i,j) = 1+c(i,j-v(i))$ entonces se contabiliza una moneda de tipo i que vale $v(i)$ y pasaríamos a ver el elemento $c(i,j-v(i))$. En el caso de que $c(i,j) = c(i-1,j)$ y $c(i,j) = 1+c(i,j-v(i))$ se podría seguir cualquiera de los dos caminos posibles. De esta forma se seguirán dando pasos hasta llegar a un elemento $c(x,0)$ y ya no quedará nada por abonar.

7.3.4. El problema de la mochila (versión 2).

Este problema es una versión del problema analizado en el tema de los Algoritmos voraces, con la particularidad de que los materiales no se pueden fragmentar en trozos más pequeños. Esto implica que si se usa un material en la solución, éste ha de ser usado por completo. Los datos son:

- El volumen de la mochila V.
- Los n materiales $m_0, m_1, ..., m_{n-1}$
- Sus volúmenes $v_0, v_1, ..., v_{n-1}$
- Sus precios unitarios $p_0, p_1, ..., p_{n-1}$.

Las soluciones, serán tuplas de reales $(x_0, x_1, ..., x_{n-1})$, que indicarán el volumen de cada material que almacenamos en la mochila, de forma tal que $x_i = \{0, v_i\}$ y $\sum_{i=0}^{n-1} x_i \leq V$. Es decir, un material o se almacena en su totalidad o no se almacena.

Dado que el problema es similar al abordado en el tema anterior, podría pensarse que una solución basada en un algoritmo voraz, similar al del tema anterior, podría ser la óptima. Para ello supongamos que se van ordenando los objetos por orden descendente de precio unitario. Si la mochila no está llena se selecciona un material completo si es posible, antes de pasar al siguiente material. Por desgracia la idea

no funciona cuando hay que seleccionar un material por completo. Ello se puede demostrar con el siguiente caso:

- Tenemos tres materiales de volúmenes $6, 5, 5$ y precios unitarios $8, 6, 5$ respectivamente.
- El volumen de la mochila es 10.
- El algoritmo voraz seleccionaría primero el material de precio 8 que ocuparía un volumen de 6 y ya no podría seguir seleccionando materiales, ya que la mochila no tendría capacidad suficiente. El coste de la mochila sería 48.
- Seleccionando los dos materiales de valor 5 y volumen 5 tendríamos un coste de la mochila de 55, que es mejor que el anterior, y la mochila quedaría completa.

Este problema se podría resolver de forma similar al del cambio, creando una tabla C de n filas y $(V + 1)$ columnas que representase los distintos subproblemas, donde hay una fila para cada material disponible y una columna para cada volumen desde 0 hasta V. El elemento $C(i, j)$ sería el coste máximo de los materiales que se pueden introducir en la mochila si se emplean los materiales que van del 1 al i si el límite de volumen de la mochila fuese j. Si indexamos como en C++, las filas y columnas desde 0, la solución del problema estará dada por el elemento de la matriz $c(n - 1, V)$.

Se puede aplicar una idea similar a la del problema del cambio

$$C(i, j) = \text{máx}\{C(i - 1, j), p_i * v_i + C(i - 1, j - v_i)\}$$

en función de que el material i se añada o no a la mochila. En esta fórmula los elementos son:

- $C(i - 1, j)$ sería el valor seleccionado cuando el material i no se añada a la mochila. En este caso el valor obtenido hasta ese momento queda invariable.
- $p_i * v_i + C(i - 1, j - v_i)$ sería el valor nuevo de la mochila si el material i fuese seleccionado. $p_i * v_i$ sería el coste añadido al seleccionar el material i y $C(i - 1, j - v_i)$ sería el coste que tenía la mochila antes de seleccionar el material i y su volumen era $j - v_i$.

Hay que resaltar que en este caso se puede apreciar una pequeña diferencia con respecto al problema del cambio, la diferencia está en el hecho de que el material i sólo se selecciona una vez (en el caso del cambio, una moneda se puede seleccionar varias veces), por lo tanto cuando es seleccionado, el nuevo coste de la mochila será el que tenía antes de usarlo y con un volumen $j - v_i$, más el coste de añadir el material i, es decir $C(i - 1, j - v_i) + p_i * v_i$.

La columna 0 se rellenará con el valor 0, ya que en esa etapa el volumen de la mochila es 0 y, por tanto, su coste es 0. Cuando $i = 0$ o cuando $j < v_i$, alguno de los elementos a comparar se salen de la tabla, y hay que tenerlo en cuenta. En el caso de la fila 0,$i = 0$, se rellenan con el material 0, cuando el volumen de la mochila lo permita, o se les asigna un valor 0, si el volumen de la mochila no permite albergar al material 0. En cualquier elemento de la fila i, si el volumen de la mochila en esa etapa no permite albergar al material i, ($j < v_i$), el material i no entrará en la mochila y $C(i, j) = C(i - 1, j)$.

El algoritmo quedaría como:

Algoritmo $mochila2((n, V; D; C)$
inicio
`Rellenamos columna 0`
para i **de** 0 **a** $n - 1$ **hacer**
$C(i, 0) \leftarrow 0$
finpara
`Rellenamos fila 0`
para j **de** 0 **a** V **hacer**
si $j < D(0).volumen$ **entonces**
$C(0, j) \leftarrow 0$
sino
$C(0, j) \leftarrow D(0).precio * D(0).volumen$
finsi
finpara
`Rellenamos resto de filas`
para i **de** 1 **a** $n - 1$ **hacer**
para j **de** 1 **a** V **hacer**
si $j < D(i).volumen$ **entonces**
$C(i, j) \leftarrow C(i - 1, j)$
sino
$C(i, j) \leftarrow \max(C(i - 1, j), D(i).precio * D(i).volumen + C(i - 1, j - D(i).volumen)$
finsi
finpara
finpara
fin

Supongamos que se tienen 5 materiales de volúmenes $v_i = \{1, 2, 5, 6, 7\}$ y de precios $p_i = \{1, 3, 3'60, 3'67, 4\}$ y que el volumen máximo es de $V = 11$

La tabla 7.2 ilustra el procedimiento para el ejemplo citado. En este caso, el valor máximo de la mochila será de 40. Para ver como se rellena la tabla, supongamos que la rellenamos por filas. Los elementos de la columna 0 valen todos 0 como es evidente, los elementos de la fila 0 solo pueden albergar al material 0 cuando el volumen de la etapa sea mayor o igual que el del material 0, en caso contrario la

Volumen límite:	0	1	2	3	4	5	6	7	8	9	10	11
$v_0 = 1, p_0 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
$v_1 = 2, p_1 = 3$	0	1	6	7	7	7	7	7	7	7	7	7
$v_2 = 5, p_2 = 3'6$	0	1	6	7	7	18	19	24	25	25	25	25
$v_3 = 6, p_3 = 3'67$	0	1	6	7	7	18	22	24	28	29	29	40
$v_4 = 7, p_4 = 4$	0	1	6	7	7	18	22	28	29	34	35	40

Cuadro 7.2: Tabla de secuencias y beneficios para el ejemplo.

mochila está vacía y su coste es 0. Como $v_0 = 1$, el material 0 entra en la mochila en todas las etapas de la fila 0, y como $p_0 * v_0$ es igual a 1, todos los elementos de la fila 0 valen 1.

Los elementos de la fila 1 se calculan como

$$C(1,j) = max\{C(0,j), p_1 * v_1 + C(0, j - v_1)\} = max\{C(0,j), 2*3 + C(0, j-2)\}$$

.

- Cuando $j = 1$ se obtiene que $j < v_1$ y por tanto $C(1,1) = C(0,1)$.
- Cuando $j = 2$ se tiene que $C(1,2) = max\{C(0,2), 2 * 3 + C(0,0)\} = max\{1, 6 + 0\} = 6$.
- Cuando $j = 3$ se tiene que $C(1,3) = max\{C(0,3), 2 * 3 + C(0,1)\} = max\{1, 6 + 1\} = 7$.
- De la misma forma se obtienen los restantes elementos de la fila 1.

Los elementos de la fila 2 se calculan como

$$C(2,j) = max\{C(1,j), p_2 * v_2 + C(1, j - v_2)\} = max\{C(1,j), 5*3'6 + C(1, j-5)\}$$

.

- Para todos los valores de j que van de 1 a 4, $j < v_2$ y por tanto en esos casos $C(2,j) = C(1,j)$.
- Para $j = 5$ se tiene que $C(2,5) = max\{C(1,5), 5 * 3'6 + C(1,0)\} = max\{7, 18 + 0\} = 18$.
- Para $j = 6$ se tiene que $C(2,6) = max\{C(1,6), 5 * 3'6 + C(1,1)\} = max\{7, 18 + 1\} = 19$.
- Para $j = 7$ se tiene que $C(2,7) = max\{C(1,7), 5 * 3'6 + C(1,2)\} = max\{7, 18 + 6\} = 24$.

- Para $j = 8$ se tiene que $C(2,8) = max\{C(1,8), 5 * 3'6 + C(1,3)\} = max\{7, 18+7\} = 25$.

- Para j desde 9 a 11 se obtiene lo mismo que para $j = 8$.

Los elementos de las filas 3 y 4 se calculan de manera similar.

Al igual que en el problema del cambio, a partir de la tabla se pueden obtener los materiales usados para la mochila. Si analizamos $C(4,11)$, veremos que $C(4,11) = C(3,11)$ pero $C(4,11) \neq C(3,11-v_4) + p_4 * v_4$, con lo cual el material m_4 no está incluido. Seguidamente se observa que $C(3,11) \neq C(2,11)$ pero $C(3,11) = C(2,11-v_3)+p_3*v_3$, con lo cual se deduce que el material m_3 entra en la mochila. A continuación se observa que $C(2,5) \neq C(1,5)$ pero $C(2,5) = C(1,5-v_2)+p_2*v_2$, con lo cual se deduce que el material m_2 está en la mochila. De la misma forma tenemos que $C(2,0) = C(1,0)$ y que $C(1,0) = C(0,0)$, con lo cual la mochila no incluye a los materiales m_1 ni m_0. Por lo tanto la mochila solo contendría a los materiales m_2 y m_3.

Un algoritmo voraz incluiría primero el material m_4 y después parte del material m_3. El coste total de la mochila para el voraz sería de $7 * 4 + 4 * 3{,}67 = 42{,}33$, que es mayor que la solución óptima obtenida por el algoritmo basado en programación dinámica.

7.3.5. Camino más corto en un grafo polietápico.

Un grafo polietápico es un grafo dirigido $G = \{N, L\}$ en el cual el conjunto N de nodos se puede particionar en k conjuntos disjuntos N_i con $1 \leq i \leq k$, donde cada lado $l(u,v)$ del conjunto de lados L enlaza dos nodos $u \in N_i$ y $v \in N_{i+1}$ para cualquier $1 \leq i \leq k$. Los conjuntos son de tal forma que N_1 y N_k contienen un solo nodo, que son los nodos inicial y final del grafo, y el resto de conjuntos disjuntos conforman una etapa intermedia del grafo. El problema consiste en obtener el camino de menor coste que enlaza el nodo inicial y el nodo final del grafo. El grafo de la figura 7.1 es un ejemplo ilustrativo de este tipo de grafos, y el camino $\{1,2,6,9,11\}$ sería el camino mínimo de ese grafo polietápico. En este caso el grafo tendría 5 etapas.

Se puede apreciar que todos los caminos que enlazan el primer y el último nodo tienen el mismo número de lados, que es $k-1$ siendo k el número de etapas (conjuntos disjuntos) del grafo. Si llamamos 0 a la etapa inicial, correspondiente al primer nodo, y n a la etapa final, correspondiente al último nodo. Cualquier camino será de la forma $N_{i0}, N_{i1}, N_{i2}, \ldots, N_{in-1}, N_{in}$ donde N_{ij} será un nodo de la etapa j y N_{i0} y N_{in} son los nodos inicial y final del grafo. Como el coste de un camino es la suma de los costes de los lados que lo componen, es evidente que se cumple el principio de optimalidad de Bellman.

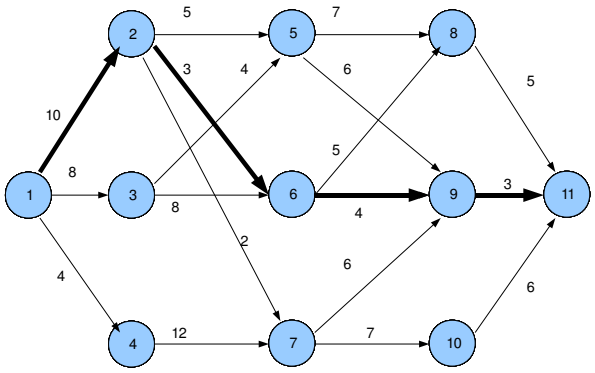

Figura 7.1: Grafo polietápico.

Si encontramos el camino mínimo entre el nodo origen del grafo (A) y un nodo Y, siendo Y un nodo cualquiera del grafo que podría corresponder a cualquier etapa, haciendo que Y sea el nodo final B, tendremos el camino mínimo en el grafo. Sabiendo que todos los caminos que van desde A hasta Y tienen el mismo número de lados, se pueden obtener los caminos mínimos desde el nodo origen A hasta un nodo de la etapa i-ésima haciendo crecer a i de manera progresiva. Aplicando el principo de optimalidad, podemos plantear que si Y pertenece a una etapa E_{i+1}, todo camino que va de A a Y consta de un subcamino que va de A a X, siendo X un nodo de la etapa E_i, más el lado (X, Y). Si el cálculo se ha ido realizando progresivamente haciendo crecer la etapa, el camino óptimo desde A a $X \in E_i$ ha sido previamente calculado.

Si llamamos $c(i, j)$ al coste del camino mínimo entre dos nodos, tendremos que si A es el nodo origen e Y es un nodo perteneciente a una etapa cualquiera E_{i+1} se puede plantear la siguiente fórmula para obtener el coste $c(A, Y)$:

$$c(A, Y) = \min_{X \in E_i} \{c(A, X) + d(X, Y)\}$$

siendo $d(X, Y)$ el coste del lado (X, Y).

Si $C(i, j)$ es el camino mínimo que va del nodo i al j, entonces $C(A, Y) = C(A, X) \cup (X, Y)$. Como es fácil de ver, en la primera etapa solo hay una posibilidad para alcanzar un nodo Y cualquiera. Por tanto, en este caso $C(A, Y) = (A, Y)$ y $c(A, Y) = d(A, Y)$. Vamos a ver como se desarrollaría la solución para el ejemplo de la figura 7.1.

- En la primera etapa tennemos:

- $c(1,2) = 10, C(1,2) = \{1,2\}$
- $c(1,3) = 8, C(1,3) = \{1,3\}$
- $c(1,4) = 4, C(1,4) = \{1,4\}$

- En la segunda etapa tenemos:
  - $c(1,5) = \text{mín}\{c(1,2) + d(2,5), c(1,3) + d(3,5)\} = 12, C(1,5) = \{1,3,5\}$
  - $c(1,6) = \text{mín}\{c(1,2) + d(2,6), c(1,3) + d(3,6)\} = 13, C(1,6) = \{1,2,6\}$
  - $c(1,7) = \text{mín}\{c(1,2) + d(2,7), c(1,4) + d(4,7)\} = 12, C(1,7) = \{1,2,7\}$
- En la tercera etapa tenemos:
  - $c(1,8) = \text{mín}\{c(1,5) + d(5,8), c(1,6) + d(6,8)\} = \text{mín}\{19,18\} = 18, C(1,8) = \{1,2,6,8\}$
  - $c(1,9) = \text{mín}\{c(1,5) + d(5,9), c(1,6) + d(6,9), c(1,7) + d(7,9)\} = \text{mín}\{18,17,18\} = 17, C(1,9) = \{1,2,6,9\}$
  - $c(1,10) = \text{mín}\{c(1,7) + d(7,10)\} = 19, C(1,10) = \{1,2,7,10\}$
- En la última etapa tenemos:
  - $c(1,11) = \text{mín}\{c(1,8)+d(8,11), c(1,9)+d(9,11), c(1,10)+d(10,11)\} = \text{mín}\{23,20,25\} = 20, C(1,11) = \{1,2,6,9,11\}$

Como resultado se obtiene el camino $\{1,2,6,9,11\}$ y su coste será de 20. Para la implementación del algoritmo vamos a usar un vector de $n+1$ elementos para guardar las etapas y los nodos que pertenecen a cada una de las etapas. El tipo $Etapa$ que reflejará la información de cada etapa guardará lo siguiente:

- $numeroNodosEtapa$, que indicará el número de nodos de esa etapa.
- $nodo$ vector de tipo $Nodo$ que contiene los nodos de esta etapa.

A su vez utilizaremos un tipo $Nodo$ que almacenará lo siguiente:

- $etiqueta$ almacenará la etiqueta del nodo.
- $etapa$ almacena la etapa a la que pertenece el nodo.
- $indice$ indica la posición que ocupa el nodo en esa etapa.

El algoritmo devolverá un vector de costes ($coste$), correspondiente al coste mínimo de cada camino a cualquier nodo del grafo, y un vector de caminos ($camino$) que guardará el camino mínimo a cada nodo.

Algoritmo $caminoMinimoGrafoPolietapico(Etapa, n; ; coste, camino)$
inicio
Inicialmente el coste del camino al nodo origen es 0
y el camino coincide con el mismo nodo
$coste(Etapa(0).nodo(1).etiqueta) \leftarrow 0$
$camino(Etapa(0).nodo(1).etiqueta) \leftarrow Etapa(0).nodo(1).etiqueta$
Calculamos directamente los costes y caminos de la primera etapa
$origen \leftarrow Etapa(0).nodo(1).etiqueta$
para i **de** 1 **a** $Etapa(1).numeroNodosEtapa$ **hacer**
$destino \leftarrow Etapa(1).nodo(i).etiqueta$
$coste(destino) \leftarrow d(origen, destino)$ el coste será el peso del lado
$camino(destino) \leftarrow origen$ el camino hasta destino lo compondrá solo el ori
finpara
Ahora se calculan el resto de etapas
para i **de** 2 **a** n **hacer**
se recorren los nodos de cada etapa
para j **de** 1 **a** $Etapa(i).numeroNodosEtapa$ **hacer**
$destino \leftarrow Etapa(i).nodo(j).etiqueta$
$minimoCoste \leftarrow \infty$
$indiceMinimo \leftarrow 0$
se recorren los nodos de la etapa anterior para ver los conectados
al nodo analizado
para k **de** 1 **a** $Etapa(i-1).numeroNodosEtapa$ **hacer**
$origen \leftarrow Etapa(i-1).nodo(k).etiqueta$
se comprueba si origen y destino están conectados
si $d(origen, destino) < \infty$ **entonces**
si $coste(origen) + d(origen, destino) < minimoCoste$ **entonces**
$minimoCoste \leftarrow coste(origen) + d(origen, destino)$
$indiceMinimo \leftarrow Etapa(i-1).nodo(k).etiqueta$
finsi
finsi
finpara
Acabamos de seleccionar el camino minimo hasta el nodo evaluado
$coste(destino) \leftarrow minimoCoste$ se almacena el minimo coste en el nodo dest
Ahora se añade el nodo de la etapa anterior en el camino minimo
al camino mínimo hasta dicho nodo
$camino(destino) \leftarrow camino(indiceMinimo) \cup indiceMinimo$
finpara
finpara

fin

7.3.6. El problema del viajante de comercio.

La solución del problema se puede plantear como una sucesión de decisiones que verifique el principio de optimalidad de Bellman. La idea se basa en generar una solución mediante la búsqueda sucesiva de recorridos mínimos de tamaño 1, 2, 3, etc.

Considerando el grafo g con un conjunto de nodos N y de lados L y siendo m su matriz de conexión, cada recorrido del viajante que parte del nodo n_1 estará formado por un lado L_{n_1,n_k}, para algún nodo n_k perteneciente a $N - \{n_1\}$ y un camino de n_k al nodo n_1. Si el recorrido es óptimo también ha de ser óptimo el camino de n_k al nodo n_1, pues si no lo fuese obtendríamos un camino mejor que el óptimo, lo cual es imposible. Por tanto, se cumple el principio de optimalidad de Bellman. Siguiendo este razonamiento, se puede plantear una solución recursiva, similar a la planteada en el método general. Sea $d(n_i, S)$ la longitud del camino mínimo que partiendo del nodo n_i pasa por todos los nodos del conjunto S y vuelve al nodo n_i. La solución al problema del viajante se puede representar como $d(n_1, N - \{n_1\})$, y se podría obtener de la siguiente forma:

$$d(n_1, N - \{n_1\}) = \min_{2 \leq k \leq n} \{L_{n_1,n_k} + d(n_k, N - \{n_1, n_k\}\}$$

Aquí se puede apreciar la diferencia que existe entre la estrategia de este algoritmo y la que se sigue en los algoritmos voraces. En los algoritmos voraces se ha de escoger una de las posibles opciones en cada paso, y una vez tomada o desechada, ya no vuelve a ser considerada nunca. Son algoritmos que no guardan *historia*, y por tanto no siempre funcionan. Sin embargo, en la Programación Dinámica la solución al problema total se va construyendo de otra forma: a partir de las soluciones óptimas para problemas más pequeños. No obstante, la solución tiene un inconveniente derivado de la necesidad de usar una estructura de datos que permita reutilizar los cálculos. Tal estructura debería contener las soluciones intermedias necesarias para la estimación de $d(n_1, N - \{n_1\})$, pero estas son demasiadas. La estructura sería una tabla con n filas, y 2^n columnas, pues éste es el cardinal de las partes del conjunto N, que son todas las posibilidades que puede tomar el segundo parámetro de d en su definición. 2^n sería el número de conjuntos posibles que se podrían obtener de un conjunto de n elementos y se obtiene de la expresión:

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \binom{n}{3} + \cdots + \binom{n}{n}$$

.

Para ver el funcionamiento del método, se va a detallar un ejemplo de un grafo dirigido de 4 nodos, en el cual el viajante parte del nodo 1, cuya matriz de conexión es la siguiente:

$$\begin{pmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{pmatrix}$$

En primer lugar se van a calcular los caminos óptimos que finalizan en el nodo 1, y que solo tienen un nodo previo. Estos caminos solucionarán los $d(i, \phi) = L_{i,1}, i = 2, 3, \ldots, n$. De esta forma tendríamos:

- $d(2, \phi) = L_{2,1} = 5$
- $d(3, \phi) = L_{3,1} = 6$
- $d(4, \phi) = L_{4,1} = 8$

En segundo lugar se obtienen los caminos óptimos que llegan al nodo 1, con dos nodos previos:

- $d(2, \{3\}) = L_{2,3} + d(3, \phi) = 9 + 6 = 15$
- $d(2, \{4\}) = L_{2,4} + d(4, \phi) = 10 + 8 = 18$
- $d(3, \{2\}) = L_{3,2} + d(2, \phi) = 13 + 5 = 18$
- $d(3, \{4\}) = L_{3,4} + d(4, \phi) = 12 + 8 = 20$
- $d(4, \{2\}) = L_{4,2} + d(2, \phi) = 8 + 5 = 13$
- $d(4, \{3\}) = L_{4,3} + d(3, \phi) = 9 + 6 = 15$

En tercer lugar se obtienen los caminos óptimos que llegan al nodo 1, con tres nodos previos.

- $d(2, \{3, 4\}) = \text{mín}\{L_{2,3} + d(3, \{4\}), L_{2,4} + d(4, \{3\})\} = \text{mín}\{9 + 20, 10 + 15\} = 25$
- $d(3, \{2, 4\}) = \text{mín}\{L_{3,2} + d(2, \{4\}), L_{3,4} + d(4, \{2\})\} = \text{mín}\{13 + 18, 12 + 13\} = 25$
- $d(4, \{2, 3\}) = \text{mín}\{L_{4,2} + d(2, \{3\}), L_{4,3} + d(3, \{2\})\} = \text{mín}\{8 + 15, 9 + 18\} = 23$

Por último, se obtiene el caminos óptimo que llega al 1, con cuatro nodos previos, y que comienzan por el nodo 1:

- $d(1,\{2,3,4\}) = \text{mín}\{L_{1,2}+d(2,\{3,4\}), L_{1,3}+d(3,\{2,4\}), L_{1,4}+d(4,\{2,3\})\} = \text{mín}\{10+25, 15+25, 20+23\} = 35$

De esta forma obtendríamos que el camino óptimo tiene una distancia de 35, la otra cuestión consiste en calcular la ruta óptima. Para ello hay que ir guardando el índice que va optimizando los sucesivos caminos que se van obteniendo en las distintas etapas, y estos índices conforman el camino mínimo. Con este propósito vamos a usar una función I, que va a ir guardando el valor que minimiza a $d(i, N-\{i\})$ en las sucesivas etapas. En el ejemplo tendremos:

- $I(2,\{3\}) = 3$
- $I(2,\{4\}) = 4$
- $I(3,\{2\}) = 2$
- $I(3,\{4\}) = 4$
- $I(4,\{2\}) = 2$
- $I(4,\{3\}) = 3$
- $I(2,\{3,4\}) = 4$
- $I(3,\{2,4\}) = 4$
- $I(4,\{2,3\}) = 2$
- $I(2,\{3,4\}) = 4$
- $I(1,\{2,3,4\}) = 2$

Por tanto, la ruta óptima es:
$1 \to I(1,\{2,3,4\}) = 2 \to I(2,\{3,4\}) = 4 \to I(4,\{3\}) = 3 \to 1$
Para obtener la función d se podría usar el siguiente algoritmo recursivo.

```
Algoritmo d(i, S)
  inicio
    si vacia(S) = cierto entonces
      devolver L(i, 1)
    sino
      masCorto ← ∞
      para j ∈ S hacer
        distancia ← L(i, j) + d(j, S − j)
        si distancia < masCorto entonces
          masCorto ← distancia
        finsi
```

```
        finpara
        devolver masCorto
    finsi
fin
```

S es el conjunto de nodos que se van a incluir en el recorrido, excepto el inicial. En la primera llamada S contendrá a todos los nodos menos el origen.

7.3.7. El problema de la aproximación poligonal óptima (I)

Éste es otro problema que se puede resolver mediante programación dinámica. El problema se puede plantear de diversas formas. La forma que se aborda en este apartado es la siguiente: Dada una curva de N puntos en dos dimensiones y en formato digital, de forma tal que los puntos de la misma no tienen ninguna discontinuidad, obtener la aproximación poligonal de M puntos que mejor se adapta a la curva generando el menor error posible. Este problema es conocido por $min - \epsilon$. El error se cuantifica sumando los cuadrados de las distancias de todos los puntos de la curva a los segmentos de la aproximaciópn poligonal (Figura 7.2. Dicho error se denomina ISE (Integral square error) y se obtiene mediante la expresión:

$$ISE = \sum_{i=1}^{N} e_i^2 \tag{7.1}$$

donde:

- e_i es la distancia desde P_i al segmento que lo aproxima.
- N es el número de puntos de la curva.

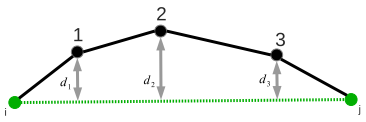

Figura 7.2: Cálculo del valor de ISE.

Si se quisieran evaluar todas las posibilidades y seleccionar la mejor, el número de posibilidades sería $C_{N,M}$. Para poder abordar el problema usando la técnica de programación dinámica, hay que definir la descomposición en problemas parciales cuya solución óptima nos servirá para obtener la solución óptima del problema final. Dichos problemas parciales se pueden plantear siguiendo la siguiente definición recursiva para el error:

$$E(n,m) = \min_{m-1 \leq j \leq n-1} \{E(j, m-1) + e(P_j, P_n)\}$$

donde:

- $E(n,m)$, es el error acumulado (ISE), cuando se aproximan los n primeros puntos de la curva con m puntos.
- $E(j, m-1)$ es el error acumulado (ISE), cuando se aproximan los j primeros puntos de la curva con $m-1$ puntos.
- $e(P_j, P_n)$ es el error acumulado (ISE), del segmento que une el punto j con el n

Obsérvese que j varía de $m-1$ a $n-1$, ya que para ajustar los n primeros puntos a partir de los $n-1$ primeros, y usando m puntos, se parte de todos los problemas ya resueltos en la etapa anterior. En la etapa anterior hay que resaltar lo siguiente:

- Cuando $j = m-1$, es el caso en que se ajustan los $m-1$ primeros puntos con $m-1$ puntos, que no pueden ser otros que los $m-1$ primeros. En este caso el $E(m-1, m-1)$ sería 0.
- Cuando $j = n-1$, estamos en el caso más extremo, ya que se ajustarían con $m-1$ puntos, los $n-1$ primeros puntos, y nos faltaría el segmento que va del punto P_{n-1} al punto P_n para completar los m puntos de la solución.

Resumiendo, j varía entre $m-1$ y $n-1$ ya que es el rango posible de distribución de los $m-1$ primeros puntos de la aproximación, cuando se aproximan los $n-1$ primeros puntos de la curva.

Los casos particulares se resumen en:

- $E(1,1) = 0$
- $E(n,1) = \infty$ para $n = 2, \ldots, N$

Por último, el error que hay que calcular es $E(N, M)$.

Para ver el funcionamiento del algoritmo, se va a ver el desarrollo del mismo para la curva cerrada de la figura 7.3. La curva tiene 6 puntos y se va a obtener la aproximación poligonal óptima de 4 puntos. Hay que tener en cuenta que el último punto de la curva y el de la aproximación estará duplicado ya que es una curva cerrada.

La matriz father, en el elemto i,j, contendrá al punto que precede al punto i, cuando éste ocupa la posición j en la aproximación poligonal. El valor almacenado es el índice que minimiza la expresión recursiva.

Inicialmente se tiene que:

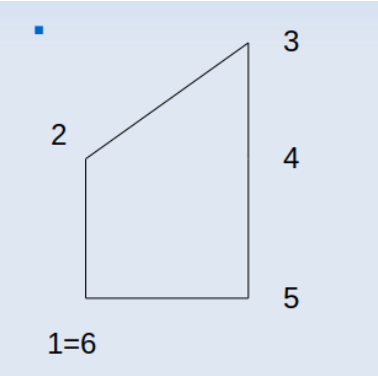

Figura 7.3: Ejemplo de curva cerrada para obtener la aproximación poligonal óptima de 4 puntos .

- $E(1,1) = 0; E(2,1) = E(3,1) = E(4,1) = E(5,1) = \infty$
- $e(1,2) = 0; e(1,3) = 1/5; e(1,4) = 1; e(1,5) = 6; e(1,6) = 9$
- $e(2,3) = 0; e(2,4) = 1; e(2,5) = 5/2; e(2,6) = 3$
- $e(3,4) = 0; e(3,5) = 0; e(3,6) = 1$
- $e(4,5) = 0; e(4,6) = 1/2$
- $e(5,6) = 0$

El desarrollo quedaría como sigue:

- Para $m = 2$
  - Para $n = 2$
    - $E(2,2) = \text{mín}\{E(1,1) + e(1,2)\} = 0$
    - $Father[2,2] = 1$
  - Para $n = 3$
    - $E(3,2) = \text{mín}\{E(1,1) + e(1,3), E(2,1) + e(2,3)\} = \text{mín}\{0 + 1/5, \infty + 0\} = 1/5$
    - $Father[3,2] = 1$
  - Para $n = 4$
    - $E(4,2) = \text{mín}\{E(1,1)+e(1,4), E(2,1)+e(2,4), E(3,1)+e(3,4)\} = \text{mín}\{0 + 1, \infty + 1, \infty + 0\} = 1$
    - $Father[4,2] = 1$

- Para $n = 5$
  - $E(5,2) = \text{mín}\{E(1,1)+e(1,5), E(2,1)+e(2,5), E(3,1)+e(3,5), E(4,1)+e(4,5)\} = \text{mín}\{0+6, \infty+5/2, \infty+0, \infty+0\} = 6$
  - $Father[5,2] = 1$
- Para $n = 6$
  - $E(6,2) = \text{mín}\{E(1,1)+e(1,6), E(2,1)+e(2,6), E(3,1)+e(3,6), E(4,1)+e(4,6), E(5,1)+e(5,6)\} = \text{mín}\{0+9, \infty+3, \infty+1, \infty+1/2, \infty+0\} = 9$
  - $Father[6,2] = 1$

■ Para $m = 3$

- Para $n = 3$
  - $E(3,3) = \text{mín}\{E(2,2) + e(2,3)\} = \text{mín}\{0+0\} = 0$
  - $Father[3,3] = 2$
- Para $n = 4$
  - $E(4,3) = \text{mín}\{E(2,2) + e(2,4), E(3,2) + e(3,4)\} = \text{mín}\{0+1, 1/5+0\} = 1/5$
  - $Father[4,3] = 3$
- Para $n = 5$
  - $E(5,3) = \text{mín}\{E(2,2)+e(2,5), E(3,2)+e(3,5), E(4,2)+e(4,5)\} = \text{mín}\{0+5/2, 1/5+0, 1+0\} = 1/5$
  - $Father[5,3] = 3$
- Para $n = 6$
  - $E(6,3) = \text{mín}\{E(2,2)+e(2,6), E(3,2)+e(3,6), E(4,2)+e(4,6), E(5,2)+e(5,6)\} = \text{mín}\{0+3, 1/5+1, 1+1/2, 6+0\} = 6/5$
  - $Father[6,3] = 3$

■ Para $m = 4$

- Para $n = 4$
  - $E(4,4) = \text{mín}\{E(3,3) + e(3,4)\} = \text{mín}\{0+0\} = 0$
  - $Father[4,4] = 3$
- Para $n = 5$
  - $E(5,4) = \text{mín}\{E(3,3) + e(3,5), E(4,3) + e(4,5)\} = \text{mín}\{0+0, 1/5+0\} = 0$
  - $Father[5,4] = 3$

- Para $n = 6$
  - $E(6,4) = \text{mín}\{E(3,3)+e(3,6), E(4,3)+e(4,6), E(5,3)+e(5,6)\} = \text{mín}\{0+1, 1/5+1/2, 1/5+0\} = 1/5$
  - $Father[6,4] = 5$

Ahora se deducen los puntos que conforman el óptimo. Para ello hay que usar los valores almacenados en la matriz father.

- El último punto para obtener el polígono óptimo de 4 puntos es el 6 (que coincide con el primero).
- $Father[6,4] = 5$ indica que el punto que precede al 6, cuando éste ocupa la posición 4 en la aproximación es el punto 5.
- $Father[5,3] = 3$ indica que el punto que precede al 5, cuando éste ocupa la posición 3 en la aproximación es el punto 3.
- $Father[3,2] = 1$ indica que el punto que precede al 3, cuando éste ocupa la posición 2 en la aproximación es el punto 1.
- Finalmente, indicar que la aproximación poligonal óptima de 4 puntos la componen los puntos 1, 3, 5 y 6.

Por último, hay que resaltar que éste algoritmo proporciona la aproximación poligonal óptima que cominenza por el punto inicial de la curva y su complejidad es $O(MN^2)$. En el caso de que la curva fuese cerrada, para obtener el óptimo habría que aplicar el algoritmo tomando los N puntos de la curva como puntos de comienzo, y seleccionar aquella solución que proporcione el menor error. En el caso de curvas cerradas la complejidad sería $O(MN^3)$.

7.3.8. El problema de la aproximación poligonal óptima (II)

Otro método para abordar este problema se basa en el algoritmo A^*. Según este método se construye un grafo dinámicamente. Dicho grafo representa el árbol de búsqueda de las posibles aproximación poligonales. Cada nodo del grafo almacena:

- Un punto de la curva.
- El costo o error acumulado ISE desde el punto inicial de la aproximación hasta dicho punto.
- Un rango que representa el lugar que ocupa en la aproximación.
- El punto previo a dicho punto en la posible aproximación poligonal.

Un lado entre dos nodos cualesquiera del grafo representa un lado de la aproximación poligonal y el error del segmento correspondiente al lado es asociado al lado (costo del lado). La aproximación óptima viene dada por el camino más corto desde el nodo de comienzo hasta el nodo final (coincidirá con el primero cuando sea una curva cerrada). Como a priori se sabe el número de punto de la aproximación M, el sucesor de un nodo con rango $M-1$ ha de ser el último, por otra parte, el último punto no puede seleccionarse con rango menor de M.

A la hora de implementarlo, hay que tener en cuenta las siguientes consideraciones:

- Usa dos listas: lista abierta y lista cerrada.
- La lista abierta que contiene nodos que aún tienen que ser examinados. Dichos nodos se ordenan ascendentemente según su costo (error del camino que llega a ese nodo con ese rango). Inicialmente la Lista abierta solo contiene al nodo inicial con rango 1 y coste 0.
- En cada paso se extrae el nodo de coste mínimo de la lista abierta, nodo x con rango r, y se pasa a la cerrada, que inicialmente está vacía.
- Si el nodo x, de rango r, no es el último, se examinan todos los posibles nodos sucesores y con un rango de $r+1$ y con coste igual al de x más el del lado (x, y). Evidentemente estos nodos y de rango $r+1$ serán aquellos nodos que puedan conducir a una solución factible. Ahora se presentan dos casos:
  - Si y con rango $r+1$ no está en la lista abierta, se introduce en ella y el punto previo para el punto y será el punto x.
  - Si y con rango $r+1$ está en la lista abierta y el nuevo coste es menor, se actualiza el coste y el nodo previo. Si el coste no es menor, el nodo se deja como está.
- Si se selecciona como mínimo de la lista abierta el último nodo con rango M, no es posible seleccionarlo con otro rango, el algoritmo termina y la aproximación se obtiene volviendo al inicio de punto previo en punto previo.

Para ver el funcionamiento del método, se va a aplicar en el mismo ejemplo del apartado anterior. Cada nodo se representará con cuatro valores: punto, coste, rango y punto previo. LA_i representará a la lista abierta en el paso i y LC_i representará a la cerrada. Igual que en el caso anterior obtendremos la aproximación óptima de 4 puntos.

- Inicio:
  - $LA_0 = \{(1, 0, 0, 0)\}$

- $LC_0 = \{\}$

- Paso 1:
  - Nodo $minimo_1 = (1, 0, 0, 0)$
  - $LC_1 = \{(1, 0, 0, 0)\}$
  - $LA_1 = \{(2, 0, 1, 1), (3, 1/5, 1, 1), (4, 1, 1, 1)\}$. No hay más porque hay que garantizar que $M = 4$.

- Paso 2:
  - Nodo $minimo_2 = (2, 0, 1, 1)$
  - $LC_2 = LC_1 + (2, 0, 1, 1)$
  - $LA_2 = \{(3, 1/5, 1, 1), (4, 1, 1, 1), (3, 0, 2, 2), (4, 1, 2, 2)(5, 5/2, 2, 2)\}$. No hay más porque hay que garantizar que $M = 4$.

- Paso 3:
  - Nodo $minimo_3 = (3, 0, 2, 2)$
  - $LC_3 = LC_2 + (3, 0, 2, 2)$
  - $LA_3 = \{(3, 1/5, 1, 1), (4, 1, 1, 1), (4, 1, 2, 2), (5, 5/2, 2, 2), (6, 1, 3, 3)\}$.

- Paso 4:
  - Nodo $minimo_4 = (3, 1/5, 1, 1)$
  - $LC_4 = LC_3 + (3, 1/5, 1, 1)$
  - $LA_4 = \{(4, 1, 1, 1), (4, 1/5, 2, 3), (5, 1/5, 2, 3), (6, 1, 3, 3)\}$. El nodo 4 con rango 2 ha mejorado su coste, al igual que el 5 con rango 2. Como consecuenca de ello su nuevo previo será el contenido en el mínimo de este paso (3).

- Paso 5:
  - Nodo $minimo_5 = (4, 1/5, 2, 3)$
  - $LC_5 = LC_4 + (4, 1/5, 2, 3)$
  - $LA_5 = \{(4, 1, 1, 1), (5, 1/5, 2, 3), (6, 7/10, 3, 4)\}$. El nodo 6 con rango 3 ha mejorado su coste. Como consecuenca de ello su nuevo previo será el contenido en el mínimo de este paso (4). $7/10$ se obtiene de $1/2+1/5$

- Paso 6:
  - Nodo $minimo_6 = (5, 1/5, 2, 3)$

- $LC_6 = LC_5 + (5, 1/5, 2, 3)$
- $LA_6 = \{(4, 1, 1, 1), (6, 1/5, 3, 5)\}$. El nodo 6 con rango 3 ha mejorado su coste. Como consecuenca de ello su nuevo previo será el contenido en el mínimo de este paso (5). $1/5$ se obtiene de $1/5 + 0$.

- Paso 7:
  - Nodo $minimo_7 = (6, 1/5, 3, 5)$
  - Al seleccionar el 6 con rango 3 (4 puntos incluyendo el primero 2 veces para cerrar el contorno) se ha llegado al final del algoritmo.
  - El error total es $1/5$ (costo del nodo) y el contorno se obtiene a través del punto previo de la siguiente manera: $previo(6, 1/5, 3, 5) = 5$, $previo(5, 1/5, 2, 3) = 3$, $previo(3, 1/5, 1, 1) = 1$. Por lo tanto, la aproximación óptima será la formada por los puntos 1, 3, 5 y 6.

Al igual que en el método anterior, el óptimo se obtiene para un punto inicial fijo.

Para acortar el tiempo computacional del algoritmo, su complejidad computacional no puede ser mejorada, se pueden usar estrategias de poda basadas en algoritmos subóptimos.

7.4. Cuestiones sobre el tema

1. Describe el método general de la programación dinámica, escribiendo la fórmula recursiva que representa el método general.

2. Enuncia el principio de optimalidad de Belman. ¿Se cumple siempre este principio?. Indica un ejemplo que justifique tu respuesta.

3. Describe el algoritmo de la competición internacional. Escribe la fórmula general para obtener cualquiera de los posibles estados que se producen a lo largo de la competición.

4. ¿Cual es el orden de complejidad del algoritmo de la competición internacional en sus versiones recursiva e iterativa?. Justifica tu respuesta.

5. Como sería la fórmula general recursiva del algoritmo de la competición internacional si se enfrentasen tres jugadores (A, B y C) de forma tal que gana la competición aquel que gane n veces, sabiendo que la probabilidad de que gane A en un enfrentamiento es p, la de que gane B es q y la de que gane C es 1-p-q. Escribe también los casos particulares.

6. Indica como se resuelve el problema del cambio usando programación dinámica, describiendo la expresión recursiva que lo resuelve.

7. Resuelve el problema del cambio usando programación dinámica para el caso siguiente: Cambio = 6, valores de las monedas = 1, 3, 4. Indica paso a paso las monedas que se obtienen de cada tipo en la solución final.

8. ¿Podría tener más de una solución le problema del cambio?. Justifica tu respuesta.

9. Resuelve el problema de la mochila si se tienen 4 materiales de volúmenes 1, 2, 3, 4 y de precios unitarios (por unidad de volumen) de 1, 3, 4, 5 sabiendo que el volumen de la mochila es de 6.

10. Indica como se resuelve el problema de la mochila usando programación dinámica, describiendo la expresión recursiva que lo resuelve.

11. ¿Podría tener más de una solución le problema de la mochila?. Justifica tu respuesta.

12. ¿Qué pequeña diferencia existe entre las fórmulas recursivas para obtener la solución al problema del cambio y al problema de la mochila?

13. ¿En qué consiste el problema del camino mínimo en un grafo polietápico?. ¿Se puede obtener más de una solución?. Justifica tu respuesta.

14. ¿Cuales son las fórmulas recursiva a partir de la cual obtenemos la distancia y caminos mínimos en un grafo polietápico?. Indica lo que significa cada uno de los elementos de las fórmulas.

15. Describe como se puede resolver el problema del viajante de comercio mediante programación dinámica. Usa para ello la fórmula recursiva general que lo resuelve.

16. ¿Cuantos posibles caminos hay que evaluar en el problema del viajante de comercio?: ¿De qué orden de complejidad es el algoritmo del viajante de comercio basado en programación dinámica?. Justifica tu respuesta.

17. Escribe la fórmula recursiva general para obtener la aproximación poligonal óptima, usando programación dinámica, indicando lo que significa cada uno de sus elementos.

18. ¿Cómo se pueden obtener los puntos de la aproximación poligonal óptima a partir de la fórmula general cuando se usa programación dinámica?.

19. ¿Cual es el orden de complejidad del algoritmo para obtener la aproximación poligonal óptima en el caso de una curva cerrada y en el caso de una curva abierta?. A qué se debe la diferencia entre ambas complejidades?.

20. ¿Podríamos mejorar la complejidad del algoritmo de la aproximación poligonal óptima usando alguna estrategia de poda?.