6

Programación Dinámica

Introducción

Una forma razonable y comúnmente empleada de resolver un problema es definir o caracterizar su solución en términos de las soluciones de subproblemas del mismo. Esta idea, cuando se emplea recursivamente, proporciona métodos eficientes de solución para problemas en los que los subproblemas son versiones más pequeñas del problema original.

Una técnica de diseño de algoritmos, que sigue esta idea, es la conocida como Divide y Vencerás que hemos estudiado en temas anteriores. Como allí se explicó, consiste en descomponer cada problema en un número dado de subproblemas, resolver cada uno de ellos (quizás empleando el mismo método) y combinar estas soluciones parciales para obtener una solución global. Desde el punto de vista de la complejidad de los algoritmos que se obtienen por este procedimiento, lo mejor es que todos los subproblemas tengan dimensión similar y que la suma de estas sea lo menor posible. No obstante, en muchos casos, dado un problema de tamaño n solo puede obtenerse una caracterización efectiva de su solución en términos de la solución de los subproblemas de tamaño n-1 (n en total), lo que puede desarrollarse recursivamente. En estos casos, la técnica conocida como Programación Dinámica (PD) proporciona algoritmos bastante eficientes. Es el caso, por ejemplo, de los problemas de la mochila, del camino mínimo o del viajante de comercio.

En efecto, para el primero, su solución puede entenderse como el resultado de una sucesión de decisiones. Tenemos que decidir los valores de x_i, $1 \leq i \leq n$. Así, primero podríamos tomar una decisión sobre x_1, luego sobre x_2, etc. Una sucesión óptima de decisiones, verificando las restricciones del problema, será la que maximice la correspondiente función objetivo. En el caso del camino mínimo, una forma de encontrar ese camino desde un vértice i a otro j en cierto grafo dirigido G, es decidiendo que vértice debe ser el segundo, en el camino, cual el tercero, cual el cuarto, etc. hasta que alcanzáramos el vértice j. Una sucesión óptima de decisiones proporcionaría entonces el camino de longitud mínima que buscamos.

El nombre PD y la metodología que subyace en esa técnica lo ha tomado la Teoría de Algoritmos de la Teoría de Control, donde la PD es una técnica para obtener la política óptima en problemas de control con n etapas, caracterizando ésta en términos de la política óptima para un problema similar, pero con n-1 etapas. Por este motivo comenzamos nuestra exposición analizando el método de la PD en

su contexto original, para pasar después a su formulación dentro de la Teoría de Algoritmos.

El Lugar de la Programación Dinámica

El problema más importante en la Teoría de Control es el del control óptimo. Un típico proceso supone en este contexto cuatro tipos de variables,

1. Variables independientes, que son las variables manipulables para el control del proceso.
2. Variables dependientes, que sirven para medir y describir el estado del proceso en cualquier instante de tiempo.
3. Variables producto, que se usan para indicar y medir la calidad de la tarea que realiza el sistema de control, y
4. Ruidos, que son variables incontrolables que dependen del ambiente en el que el sistema lleve a cabo su operación.

El problema general del control óptima de un proceso consiste en determinar la forma en que las variables producto pueden mantenerse en valores óptimos, independientemente de las fluctuaciones que los ruidos, o los cambios de parámetros, puedan causar. Las variables producto se usan para describir el índice de eficacia del control del sistema. Así el problema del control óptimo supone la optimización (maximización o minimización) de ese índice de eficacia.

Generalmente, un sistema físico de n-ésimo orden puede describirse en cualquier instante de tiempo t por medio de un conjunto finito de cantidades

$$(x_1(t), x_2(t),..., x_n(t))$$

Estas cantidades se conocen como las variables de estado del sistema, y constituyen las componentes del vector de estado del sistema: x(t). La relación entre los cambios del parámetro tiempo y las variables de estado del sistema, se establece mediante una sencilla y útil hipótesis como es que la derivada del vector de estado, dx/dt, solo dependa del estado del sistema en curso y no de su historia anterior. Esta hipótesis básica lleva a una caracterización matemática del proceso por medio de una ecuación diferencial vectorial,

$$dx(t)/dt = f[x(t), m(t), t]$$

con la condición inicial $x(0) = x_0$. En esa ecuación m(t) nota el vector de control, y f es una función vectorial de las variables de estado, las señales de control, el tiempo y, posiblemente, las variables ambientales o ruidos externos. En cada instante, el vector de control m debe satisfacer la condición,

$$g(m) \leq 0$$

que refleja las restricciones que el control del sistema tenga. La función g es una función conocida de la señal de control. Entonces, el problema del diseño de un control óptima puede plantearse en los siguientes términos: Dado un proceso que debemos controlar, determinar la ley de control, o sucesión, tal que el conjunto de índices de eficacia, previamente fijado, sea óptimo.

A veces, las variables de estado de un sistema son todas accesibles para medirlas y observarlas. Para los sistemas lineales que tienen esta característica, aun en presencia de ruido, puede tratarse directamente de determinar la ley de control óptima como una función de las variables de estado. Sin embargo, es bastante frecuente que las variables de estado no sean todas accesibles para medirlas y observarlas. Entonces, la ley de control óptima se determina como una función de las mejores estimaciones de las variables de estado, que se calculan a partir de las señales de salida del sistema que se hayan medido. Por tanto, en el caso más general, tendremos que considerar la resolución de problemas de control óptima y de estimación óptima.

La Teoría de Control parte de la caracterización de un sistema mediante sus variables de estado y del diseño del sistema por medio de técnicas basadas en su representación como espacio de estados. En una formulación general, el diseño del control óptimo se ve generalmente como un problema variacional. Pero hay muchos métodos variacionales para optimizar un funcional sobre un espacio de funciones. El rango de tales métodos va desde los métodos clásicos del cálculo de variaciones, a las técnicas numéricas de aproximaciones sucesivas de modelos experimentales. Entre los métodos más frecuentemente usados para el diseño de sistemas de control están,

1.- El Cálculo de Variaciones,

2.- El Principio de Máximo de Pontryaguin, y

3.- La Programación Dinámica.

En todos los casos, el propósito es encontrar la ley de control óptima tal que, dado el funcional del índice de eficacia del sistema, lo optimice.

Pero es muy significativo que los tres métodos tienen en común el uso de principios variacionales. Cada uno de estos tres métodos está íntimamente ligado a alguna formulación bien conocida de la Mecánica clásica. Los primeros, con la ecuación de Euler-Lagrange, los segundos con el Principio de Hamilton y los terceros con la teoría de Hamilton-Jacobi. Particularmente, el Principio de Máximo de Pontryaguin emplea procedimientos, más o menos directos, del cálculo de variaciones, mientras que la PD, aun basándose en principios variacionales, usa relaciones de recurrencia.

La PD la desarrolló Richard E. Bellman a mediados de los 50 (R. Bellman: Dynamic Programming, Princeton University Press, 1957), y es una técnica simple, pero poderosa, que se ha demostrado muy útil para la resolución de problemas de decisión multietápicos. La idea fundamental que subyace en el método de la PD es similar a la de la técnica Divide y Vencerás, en definitiva que un problema difícil de resolver esta encajado en una clase de problemas más simples de resolver, lo que permite encontrar una solución al problema original.

Naturalmente, el objetivo de este tema es el estudio del uso de la PD para el diseño de algoritmos que resuelvan eficientemente problemas por lo que, una vez que la hemos colocado en un lugar de referencias exacto, a continuación nos centramos en ella. Comenzaremos por introducir los procesos de decisión multietápicos estudiándolos desde el punto de vista del diseño de su control óptima para, después, presentar el Principio del Óptimo de Bellman y emplearlo para el diseño de algoritmos que, basados en él, resuelven algunos problemas notables en las Ciencias de la Computación.

Procesos de Decisión Multietápicos: Enfoque Funcional

Uno de los enfoques posibles para el diseño de sistemas con control óptima es la búsqueda sistemática de una estrategia óptima de control en cierto espacio multidimensional de control. En este sentido, el problema de diseñar el sistema óptimo puede entenderse como un problema de decisión multietápico.

En la práctica son abundantes los ejemplos de procesos de decisión multietápica. Quizás el más inmediato de todos ellos sea un juego de naipes. Pero, en el ambiente financiero, los problemas de inversión y de determinación de políticas para hacer seguros, también, son buenos ejemplos de problemas de decisión multietápicos.

Por ejemplo, supongamos que se dispone de una cantidad inicial de dinero x para invertir en un negocio de alquiler de computadores. Este dinero se usara para comprar computadores para tratamiento de textos (CTT) y para conexiones a la red (CCR). Supongamos que se gastan y pesetas en la compra de CTT y, el resto, se dedica a la compra de CCR. Lo que producen anualmente los CTT es una función de la cantidad inicial invertida, y, e igual a g(y) el primer año. Por su lado, lo que producen anualmente los CCR es función del capital restante, x-y, e igual a h(x-y) el primer año.

La política de la empresa determina que al final de cada año, todos los computadores usados de uno y otro tipo se cambien. El cambio de todos los CTT es una función de la cantidad total de dinero invertida en CTT e igual a ay el primer año, $0<a<1$. Por su parte el valor del cambio de los CCR es una función igual a b(x-y), $0<b<1$. Los expertos que dirigen la empresa tienen que tomar una sucesión de decisiones óptimas de modo que el beneficio total a lo largo de un número N de años sea máximo. Para simplificar el problema, supondremos que los beneficios anuales no se reinvierten en el negocio.

Los problemas de decisión multietápica pueden resolverse muy bien por medio del enfoque funcional, según el cual el problema de maximización original se convierte en un problema de determinar la solución de una ecuación funcional que se obtiene como sigue. El beneficio durante el primer año es,

$$Y_1(x,y) = g(y) + h(x-y)$$

Así, para un periodo de un año, la máxima recompensa la da,

$$f_1(x) = \mathrm{Max}_{0\le y\le x}\{Y_1(x,y)\} = \mathrm{Max}_{0\le y\le x}\{g(y) + h(x-y)\} \qquad (1)$$

Nótese que la recompensa máxima es función de la cantidad a invertir inicialmente. Cuando g y h se conocen, ese máximo puede evaluarse mediante un proceso sencillo de derivación. Así que, en este caso de un año, el problema es trivial.

En nuestro caso, el dinero que queda para tratar, tras un año de operación, es la cantidad destinada al cambio de los CTT y los CCR, debido a la hipótesis de que los beneficios no se reinvierten. Por tanto, la cantidad de capital durante el segundo año de operación es,

$$x_1 = ay + b(x-y)$$

que escribiremos como

$$x_1 = y_1 + (x_1 - y_1)$$

donde y_1 es la cantidad gastada en la adquisición de nuevos CTT y $x_1 - y_1$ la gastada en CCR. Entonces, durante el segundo año de operación tenemos,

- Beneficios de la inversión en CTT = $g(y_1)$
- Beneficios de la inversión en CCR = $h(x_1 - y_1)$

y al final del segundo año,

- Valor del cambio de CTT = ay_1
- Valor del cambio de CCR = $b(x_1 - y_1)$

Con lo que el beneficio total tras dos años de operación es,

$$Y_2(x,y,y_1) = g(y) + h(x-y) + g(y_1) + h(x_1-y_1)$$

y la recompensa máxima tras esos dos años es,

$$f_2(x) = \text{Max}_{0\le y\le x,\ 0\le y1\le x1} \{Y_2(x,y,y_1)\} = \text{Max}_{0\le y\le x,\ 0\le y1\le x1} \{g(y)+h(x-y)+g(y_1)+h(x_1-y_1)\}$$

o bien, de acuerdo con la expresión (1),

$$f_2(x) = \text{Max}_{0\le y\le x,\ 0\le y1\le x1} \{g(y) + h(x-y) + f_1(x_1)\}$$

con lo que, sustituyendo tenemos para el máximo beneficio,

$$f_2(x) = \text{Max}_{0\le y\le x} \{g(y) + h(x-y) + f_1[ay + b(x-y)]\}$$

Esta ecuación supone que el máximo producido durante un periodo de dos años, puede determinarse derivando las funciones en ella, de acuerdo con las restricciones que comportan. El valor de y(x) que maximice esas funciones será la decisión óptima a tomar al principio de una operación bianual que comienza con una cantidad x.
El análisis puede repetirse para una operación trianual y obtendríamos que,

$$f_3(x) = \text{Max}_{0\le y\le x} \{g(y) + h(x-y) + f_2[ay + b(x-y)]\}$$

y así sucesivamente. Por tanto, para un periodo de N años, la recompensa máxima vendrá dada por,

$$f_N(x) = \text{Max}_{0\le y\le x} \{g(y) + h(x-y) + f_{N-1}[ay + b(x-y)]\} \qquad (2)$$

Esta es la ecuación funcional básica para este proceso de decisión N-etápico. Comenzando con $f_1(x)$, como lo determina (1), esta ecuación funcional se usa para calcular $f_2(x)$, que luego se usa para calcular $f_3(x)$, y así sucesivamente. En cada etapa, se obtiene también la decisión óptima $y_j(x)$ a tomar al comienzo de la j-ésima etapa. El valor de y(x) que maximiza las funciones entre corchetes en (2) es la decisión óptima a tomar al comienzo de la operación del año N, supuesto que inicialmente comenzamos con una cantidad x.

Para la obtención de la solución por un método directo, tendríamos que resolver las N ecuaciones simultáneas que se obtienen igualando a cero las correspondientes derivadas parciales, supuesto que esas ecuaciones fueran derivables, lo que se prevé complicado.

El Principio del Óptimo

Como hemos visto, el enfoque funcional expuesto sirve para el estudio de los procesos de decisión multietápicos de modo que, aplicando repetidamente la ecuación funcional, se puede encontrar la sucesión de decisiones óptima s para el proceso N-etápico considerado. Aunque el problema de inversión comentado se ha escogido como ilustración, el enfoque funcional proporciona una herramienta muy útil en general para resolver problemas de decisión multietápicos. Sin embargo, ahora nos vamos a dedicar a introducir el Principio del Óptimo, que nos permitirá obtener la ecuación funcional que describe un proceso de decisión multietápico como una consecuencia inmediata.
Consideremos, en primer lugar, un proceso de decisión uni-etápico. Sea x el vector de estado que caracteriza al sistema físico en cuestión en cualquier instante de tiempo. Si el estado del sistema se transforma de x^1 a x^2 por la transformación,

$$x^2 = g(x^1, m_1)$$

esta transformación producirá una respuesta, o recompensa,

$$R_1 = r(x^1, m_1)$$

El problema consiste en elegir una decisión m_1 que maximice la respuesta. A la decisión m_1 se le llama política uni-etápica. Está claro que la solución a este problema de decisión uni-etápico no tiene dificultad, porque el máximo de la respuesta está dado por,

$$f_1(x^1) = \text{Max}_{m1} \{r(x^1, m_1)\}$$

La decisión que da el máximo valor de la salida, se llama decisión óptima, o estrategia de control óptima.
En un proceso de decisión bi-etápico, si el estado del sistema se transforma, primero, de x^1 a x^2 por la transformación,

$$x^2 = g(x^1, m_1)$$

y después, de x^2 a x^3 por,

$$x^3 = g(x^2, m_2)$$

esta sucesión de operaciones produce una única salida, o recompensa total,

$$R_2 = r(x^1, m_1) + r(x^2, m_2)$$

Entonces, en este caso, el problema del diseño óptima consiste en elegir una sucesión de decisiones factibles m_1 y m_2 que maximicen la recompensa total. Este

es un proceso de decisión bi-etápico en el que $r(x^1, m_1)$ es la salida debida a la primera elección de una decisión, y $r(x^2, m_2)$ es la salida tras la elección de la segunda decisión. Esta sucesión de decisiones $\{m_1, m_2\}$ se llama política bi-etápica.
La máxima recompensa estará dada por,

$$f_2(x^1) = \text{Max}_{m1,\, m2} \{r(x^1, m_1) + r(x^2, m_2)\}$$

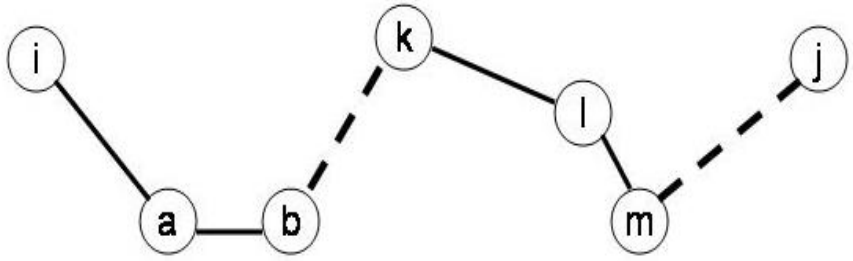
La función de recompensa total ahora se maximiza sobre la política $\{m_1,m_2\}$. A la política que maximiza a R_2 se le llama política óptima.
Como fácilmente puede verse, manejar un proceso de decisión bi-etápico es más complicado que manejar el uni-etápico anterior. La dificultad y la complejidad aumentan conforme lo hace el número de etapas del problema en consideración.
En general, para un proceso N-etápico, el problema consiste en elegir aquella política N-etápica $\{m_1, m_2 \ldots, m_N\}$ tal que maximice la recompensa total,

$$R_N = \sum_j r(x^j, m_j), \quad j \in J = \{1,2,\ldots,N\}$$

El estado del sistema se transforma de x^1 a x^2 por $x^2 = g(x^1, m_1)$, de x^2 a x^3 por $x^3 = g(x^2, m_2)$,..., y finalmente desde x^{N-1} hasta x^N por $x^N = g(x^{N-1}, m_{N-1})$. La recompensa máxima de este proceso N-etápico estará dada por,

$$f_N(x^1) = \text{Max}_{\{mj\}} \{ \sum_j r(x^j, m_j) \}, \; j \in J$$

La política $\{m_j\}$ que determina $f_N(x^1)$ es la política óptima, o estrategia de control óptima. En este caso, las m_j forman una política de control N-etápica.
Como antes se ha comentado, llevar a cabo el procedimiento de maximización por un método directo, requiere la resolución de las N ecuaciones simultaneas que se obtienen igualando a cero las derivadas parciales, con respecto a m_j, $j \in J$, de las cantidades entre los corchetes, lo que puede ser formidablemente complicado en tiempo. Es evidente, entonces, que para resolver un problema de decisión óptima con un elevado número de etapas es necesario un procedimiento sistemático que mantenga al problema en niveles tratables. Tal procedimiento sistemático de resolución puede obtenerse haciendo uso del principio fundamental de la PD, el Principio del Óptimo de Bellman, que establece que: *Una política óptima, o estrategia de control óptima, tiene la propiedad de que, cualquiera que sea el estado inicial y la decisión inicial elegidos, la decisión restante forma una estrategia de control óptima con respecto al estado que resulta a consecuencia de la primera decisión.* Otra forma, equivalente, de expresarlo es: *Una política es óptima si en un periodo dado, cualesquiera que sean las decisiones precedentes, las decisiones que quedan por tomar constituyen una política óptima teniendo en cuenta los resultados de las decisiones precedentes, o de forma mucho más concisa: Una política óptima solo puede estar formada por subpoliticas óptima s.*
En resumen, el Principio del Óptimo de Bellman lo que establece es que una política óptima solo puede estar formada por subpoliticas óptimas.
Analicemos el Principio del Óptimo de una forma intuitiva. Para ello, consideremos la siguiente figura



y supongamos que lo que nos indica es que la política óptima entre el estado i y el j, es la que pasa a través de los estados {a, b, k, l, m}. Bien, pues el Principio del Óptimo de Bellman lo que nos está diciendo es que si, por ejemplo, nosotros tuviéramos que encontrar la política óptima entre el estado k y el j, ésta sería la {k, l, m, j}, o que la política óptima entre los estados b y k es la definida exactamente por el tramo discontinuo de la anterior figura.

Como comentamos, la PD, y por tanto el Principio del Óptimo, que describe las propiedades básicas de las estrategias de control óptima, se basa en que para resolver un problema de decisión óptima especifico, el problema inicial esta encajado en una familia de problemas similares que son más fáciles de resolver. Para procesos de decisión multietápicos esto permitirá reemplazar, el problema de optimización multietápico original, por el de resolver una sucesión de procesos de decisión uni-etápicos que, indudablemente, son más fáciles de manejar.

Como antes hemos deducido, para el proceso N-etápico que considerábamos el problema consiste en elegir aquella política N-etápica $\{m_1, m_2 \ldots, m_N\}$ tal que maximice la recompensa total,

$$R_N = \sum_j r(x^j, m_j) \,, \quad j \in J = \{1,2,\ldots,N\}$$

Ahora, de acuerdo con el Principio del Óptimo, la recompensa total del mismo proceso de decisión multietápico puede escribirse como,

$$R_N = r(x^1, m_1) + f_{N-1}[g(x^1, m_1)]$$

donde el primer término, del miembro de la derecha, es la recompensa inicial y el segundo representa la recompensa máxima debida a las últimas N-1 etapas. Por tanto, la recompensa máxima estará dada por,

$$f_N(x^1) = \text{Max}_{m1} \{r(x^1, m_1) + f_{N-1}[g(x^1, m_1)]\}$$

Esta ecuación es válida solo para $N \geq 2$, pero para $N = 1$ la recompensa máxima estaría dada por,

$$f_1(x^1) = \text{Max}_{m1} \{r(x^1, m_1)\}$$

Es obvio que, aplicando este principio fundamental, un proceso de decisión N-etápico se reduce a una sucesión de N procesos de decisión uni-etápicos, con lo que disponemos de un procedimiento iterativo y sistemático para resolver eficientemente aquel problema.

El enfoque de la PD

A la vista de lo anterior, para abordar la resolución de un problema con la técnica de la PD, habría que verificar, en primer lugar, la naturaleza N-etápica del problema bajo consideración, para caracterizar la estructura de una solución óptima. Posteriormente, se tendría que comprobar el cumplimiento del Principio del Óptimo de Bellman como condición absolutamente necesaria para su aplicación, y si este se verificara, apoyándose en su propio significado en el caso concreto que estemos resolviendo, formular una ecuación de recurrencias que represente la forma de ir logrando etapa por etapa la solución óptima correspondiente. A partir de esa ecuación, a continuación, habrá que determinar la solución óptima resolviendo problemas encajados para, por último, construir la solución. Con esos ingredientes, estaremos en condiciones de poder iniciar la solución del problema que, a su vez, podría realizarse desde el primer estado hasta el último, o bien a la inversa.
De forma más concisa, el desarrollo de un algoritmo de PD correspondería a las siguientes etapas:

1) Caracterizar la estructura de una solución óptima
2) Definir recursivamente el valor de una solución óptima
3) Calcular el valor de una solución óptima en forma encajada de menos a más, y
4) Construir una solución óptima a partir de la información previamente calculada.

Las etapas 1) a 3) constituyen la base de la solución de PD para un cierto problema. La etapa 4 puede omitirse si lo único que buscamos es el valor de una solución óptima. Cuando se realiza la etapa 4, a veces mantenemos información adicional durante la etapa 3 para facilitar la construcción de una solución óptima.
Vemos así que la PD, igual que la técnica Divide y Vencerás, resuelve problemas combinando las soluciones de subproblemas. Como sabemos, los algoritmos Divide y Vencerás particionan el problema en subproblemas independientes, resuelven estos recursivamente, y combinan sus soluciones para obtener la solución del problema original. En contraste, la PD es aplicable cuando los subproblemas no son independientes, es decir, cuando los subproblemas comparten sub-subproblemas. En este contexto, un algoritmo Divide y Vencerás hace más trabajo del necesario, ya que resuelve repetidamente subproblemas comunes. Un algoritmo de PD resuelve cada subproblema solo una vez, salvando su solución en una tabla de cara a evitar el trabajo de recalcular la solución cada vez que se encuentra ese subproblema.
Esta forma de actuar, manteniendo la información relevante en una tabla, invita en algunos casos a resolver problemas que, no siendo típicamente resolubles con PD, permiten este tipo de “enfoque tabular”, que facilita soluciones de una forma rápida. Evidentemente ese enfoque también es posible sobre los problemas directamente resolubles con esa técnica. Sobre este punto volveremos más adelante para ilustrarlo con un ejemplo.
De forma esquemática, los parecidos y diferencias entre estos dos enfoques pueden sintetizarse del siguiente modo:

Programación Dinámica	Divide y Vencerás
Se progresa etapa por etapa con sub-problemas que se diferencian en tamaño en una unidad. Se generan muchas sub-sucesiones de decisiones	Divide el problema en subproblemas independientes, los resuelve recursivamente y combina sus soluciones para obtener la solución total
Es aplicable cuando los sub-problemas obtenidos comparten subproblemas	Los sub-problemas no tienen ninguna característica común de tamaño, ni se sabe su número "a priori"
No se repiten cálculos porque los problemas están encajados	Como los problemas no tienen que ser independientes pueden repetirse cálculos
Siempre obtienen la solución óptima	Siempre se obtienen soluciones óptima s

Por otro lado, el enfoque que define la PD tiene también ciertos parecidos con el que propone la técnica greedy, pero la diferencia esencial entre el método greedy y el de la PD es que en el primero siempre se generaba solo una sucesión de decisiones, mientras que en PD pueden generarse muchas sucesiones de decisiones. Sin embargo las sucesiones que contienen subsucesiones suboptimas no pueden ser óptimas (si se verifica el Principio de Principio del Óptimo), y por tanto no se generaran.
La siguiente tabla muestra estas características de una forma resumida,

Programación Dinámica	Enfoque Greedy
Se progresa etapa por etapa con sub-problemas que se diferencian entre sí en una unidad en sus tamaños	Se progresa etapa por etapa con sub-problemas que no tienen por qué coincidir en tamaño
Se generan muchas sub-sucesiones de decisiones	Solo se genera una sucesión de decisiones
Hay un gran uso de recursos (memoria)	La complejidad en tiempo suele ser baja
En cada etapa se comparan los resultados obtenidos con los precedentes: Siempre se obtiene una solución óptima	Como en cada etapa se selecciona lo mejor de lo posible sin tener en cuenta las decisiones precedentes no hay garantía de obtener el óptimo

Por último, en lo que se refiere a similitudes, otra forma de resolver problemas en los que no es posible conseguir una sucesión de decisiones que, etapa por etapa, formen una sucesión óptima, es intentarlo sobre todas las posibles sucesiones de decisiones, es decir, lo que comúnmente se llama enfoque combinatorio, enumerativo o de la fuerza bruta. Según esa metodología, lo que se hace es enumerar todas esas sucesiones, y entonces tomar la mejor respecto al criterio que se esté usando como objetivo. La PD haciendo uso explícito del Principio del Óptimo de Bellman, a menudo, reduce drásticamente la cantidad de enumeraciones que hay que hacer, evitando la enumeración de algunas

sucesiones que posiblemente nunca podrán ser óptimas. De hecho mientras que el número total de sucesiones de decisión diferentes es exponencial en el número de decisiones (si hay d elecciones para cada una de las n decisiones, entonces hay d^n posibles sucesiones de decisiones), los algoritmos de PD suelen tener eficiencia polinómica.
Ilustremos la aplicación de la técnica de la PD con un ejemplo simple que nos servirá para describir las distintas fases de su desarrollo.

El problema de la subdivisión óptima

Se quiere dividir una cantidad positiva c en n partes de forma que el producto de esas n partes sea máximo. Para determinar la subdivisión óptima aplicamos la técnica de la Programación Dinámica. Sea $f_n(c)$ el máximo producto alcanzable, x el valor de la primera subdivisión, y (c-x) el valor de las (n-1) partes restantes. De acuerdo con el Principio del Óptimo, la ecuación funcional que describe este problema de subdivisión óptima es,

$$f_n(c) = \text{Max}_{\,0\le x\le c}\,\{x f_{n-1}(c-x)\}$$

Esta ecuación es válida para $n \ge 2$. Es obvio que cuando n = 1,

$$f_1(c) = c \text{ y } f_1(c-x) = c - x$$

Así, para n = 2

$$f_2(c) = \text{Max}_{\,0\le x\le c}\,\{x f_1(c-x)\} = \text{Max}_{\,0\le x\le c}\,\{x(c-x)\}$$

y se encuentra que el valor de x que maximiza ese producto es x = c/2. Así la política óptima , es decir, la subdivisión óptima para este proceso de decisión bi-etápico es la definida por $\{m_j\} = \{c/2, c/2\}$, siendo el máximo valor del producto

$$f_2(c) = (c/2)^2$$

Consideremos ahora el caso n = 3. El máximo producto alcanzable es,

$$f_3(c) = \text{Max}_{\,0\le x\le c}\,\{x f_2(c-x)\} = \text{Max}_{\,0\le x\le c}\,\{x(c-x)^2/4\}$$

Realizando cálculos se obtiene que el valor máximo de x es x = c/3. La subdivisión de la parte restante, 2c/3, constituye un proceso de decisión bi-etápico. De acuerdo con el anterior análisis, esa parte se divide en c/3 y c/3, y por tanto la subdivisión óptima para el proceso tri-etápico es $\{m_j\} = \{c/3, c/3, c/3\}$, siendo el valor máximo del producto

$$f_3(c) = (c/3)^3$$

Cuando la cantidad c dada se divide en cuatro partes, el máximo producto que se puede alcanzar es

$$f_4(c) = \text{Max}_{\,0\le x\le c}\,\{x f_3(c-x)\} = \text{Max}_{\,0\le x\le c}\,\{x(c-x)^3/27\}$$

siendo el máximo valor de x, c/4. Ahora la subdivisión de la parte restante, 3c/4, constituye un proceso tri-etápico, de subdivisión óptima c/4, c/4 y c/4, con lo que

la subdivisión óptima de este problema tetra-etápico es $\{m_j\} = \{c/4, c/4, c/4, c/4\}$, y el valor máximo del producto es

$$f_4(c) = (c/4)^4 .$$

Todo el anterior análisis nos lleva a conjeturar que la solución para este problema, cuando n = k, sería

$$\{m_j\} = \{c/k, c/k, ..., c/k\}$$

siendo el máximo valor del producto $f_k(c) = (c/k)^k$.

Realmente puede demostrarse por inducción matemática que las anteriores relaciones son ciertas para cualquier k. De acuerdo con el Principio del Óptimo, se sigue que,

$$f_{k+1}(c) = \text{Max}_{\,0\le x\le c}\ \{x f_k(c-x)\} = \text{Max}_{\,0\le x\le c}\ \{x(c-x)^k/k\ \}$$

lo que por simple cálculo nos lleva a que x = c/(k+1), siendo el máximo valor del producto,

$$f_{k+1}(c) = [c/(k+1)]^{\,k+1}$$

Así, por inducción matemática, la subdivisión óptima del proceso n-etápico puede obtenerse como

$$\{m_j\} = \{c/n, c/n, ..., c/n\}$$

con un valor máximo del producto $f_n(c) = (c/n)^n$

Evidentemente este problema es muy simple y podría resolverse fácilmente con métodos convencionales. Sin embargo hay que destacar que ilustra la idea de la resolución del mismo como un problema de decisión multi-etápico. La formulación del problema en este sentido es lo que realmente da la clave sobre la gran utilidad del enfoque de la Programación Dinámica.

A continuación nos centramos en la aplicación de la técnica de la PD para el diseño de algoritmos, es decir en la ilustración de la aplicación de las anteriores fases para la resolución de un problema con PD, considerando algunas situaciones que ya conocemos de capítulos anteriores

El Problema del Camino Mínimo

Sea G = (N,A) un grafo dirigido en el que N es su conjunto de nodos y A el de sus arcos. Cada arco tiene asociada una longitud, no negativa que viene dada por una matriz L El problema consiste en determinar el camino de longitud mínima que una cualquier par de nodos del grafo.

Para comprobar sobre este problema la aplicabilidad de la técnica de la PD, vemos cada una de las fases en las que se desarrollaría este enfoque.

a) La naturaleza N-etápica de los problemas. Una forma de encontrar el camino mínimo desde un vértice i a otro j en un grafo dirigido G, es decidiendo que vértice debe ser el segundo, cual el tercero, cual el cuarto, etc. hasta que

alcanzáramos el vértice j. Una sucesión óptima de decisiones proporcionaría entonces el camino de longitud mínima que buscamos.

b) Comprobación del Principio del Óptimo. Concretamente, en el problema del camino mínimo, supongamos que i, i_1, i_2, ... i_k, j es el camino mínimo desde el vértice i hasta el j, es decir, comenzando con el vértice inicial i, se ha tomado la decisión de ir al vértice i_1. Como resultado de esta decisión, ahora el estado del problema está definido por el vértice i_1, y lo que se necesita es encontrar un camino desde i_1 hasta j.

Vamos a demostrar por reducción al absurdo, que la sucesión i_1, i_2, ...i_k, j es un camino mínimo entre i_1 y j. Para ello, supondremos que esto no es cierto, y entonces llegaremos a una contradicción. En efecto, si no fuera así, sería porque habría un camino más corto i_1, r_1, r_2, ... r_q, j entre i_1 y j. Pero entonces i, i_1, r_1, r_2, ...r_q, j sería un camino entre i y j más corto que el camino i, i_1, i_2, ...i_k, j. Como esto es claramente contradictorio, se verifica nuestra hipótesis de partida y el Principio del Óptimo puede aplicarse a este problema.

c) Construcción de una ecuación recurrente. El Principio del Óptimo, en sí mismo, podría decirse que es conmutativo en el sentido de que la propiedad de una subpolítica de ser óptima, es independiente de que el estado inicial del proceso se considere a la izquierda del primer estado de esa subpolítica, o a la derecha del mismo. Esto describe en esencia lo que se llama el enfoque adelantado (que de forma similar se explicaría para el retrasado) en la aplicación del Principio del Óptimo.

Consiguientemente, desde el punto de vista adelantado, el Principio del Óptimo se aplicaría como sigue. Sea S_0 el estado inicial del problema. Supongamos que hay que tomar n decisiones d_i , $1 \le i \le n$. Sea $D_1 = \{r_1, r_2, ..., r_j\}$ el conjunto de los posibles valores que podría tomar d_1. Sea S_i el estado del problema tras tomar la decisión r_i, $1 \le i \le j$. Sea Γ_i una sucesión de decisiones óptimas con respecto al estado del problema S_i. Entonces, cuando se verifica el Principio del Óptimo, una sucesión de decisiones óptimas con respecto a S_0 es la mejor de las sucesiones de decisiones $r_i\Gamma_i$, $1 \le i \le j$.

En nuestro caso, sea A_i el conjunto de los vértices adyacentes al vértice i. Para cada vértice $k \in A_i$ sea Γ_k el camino mínimo desde k hasta j. Entonces un camino más corto desde i hasta j es el más corto de los caminos $\{i, \Gamma_k / k \in A_i\}$

Nótese que el Principio del Óptimo también puede aplicarse a estados y decisiones intermedios, y no solo a los extremos. así, sea k un vértice intermedio en una camino mínimo desde i a j, i, i_1, i_2, ...k, p_1, p_2, ... j. Los dos caminos i, i_1, i_2, ...k, y k, p_1, p_2, ... j, deben ser respectivamente los caminos más cortos desde i a k y desde k a j.

Ahora, para estudiar cómo resolver operativamente el problema, es decir, para diseñar un algoritmo que lo resuelva, supondremos que los nodos están numerados de 1 a n, N = {1,2,...,n} y que L da la longitud de cada arco, de modo que $L(i,i) = 0$, $L(i,j) \ge 0$ si i es distinto de j, y $L(i,j) = \infty$ si no existe el arco (i,j).

Como hemos visto, el Principio del Óptimo se aplica del siguiente modo: Si k es un nodo en el camino mínimo que une i con j, entonces la parte de ese camino que va de i hasta k, y la del que va de k hasta j, es también óptima.

Para resolver el problema nos basamos en el célebre Algoritmo de Floyd. De acuerdo con él, construimos una matriz D que da la longitud del camino mínimo entre cada par de nodos. El algoritmo comienza asignando a D, L y, entonces,

realiza n iteraciones. Tras la iteración k, D da la longitud de los caminos mínimos que solo usan como nodos intermedios los del conjunto {1,2,...,k}. Después de n iteraciones tendremos por tanto la solución buscada. En la iteración k, el algoritmo tiene que comprobar, para cada par de nodos (i,j), si existe o no un camino que pase a través de k que sea mejor que el actual camino mínimo que solo pasa a través de los nodos {1,2,...,k-1}. Sea D la matriz después de la k-ésima iteración. La comprobación puede expresarse como,

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

donde hacemos uso del Principio del Óptimo para calcular la longitud del camino más corto que pasa a través de k. También hacemos uso implícitamente del hecho de que un camino mínimo no pasa a través de k dos veces.
En la k-ésima iteración, los valores en la k-ésima fila y k-ésima columna de D no cambian, ya que D(k,k) siempre vale cero. Por tanto, no es necesario proteger estos valores al actualizar D. Esto nos permite trabajar siempre con una matriz D bidimensional, mientras que a primera vista parece que habríamos necesitado una matriz n x n x 2 (o incluso n x n x n). Con esto, el problema se resuelve conforme al siguiente algoritmo,

```
PROCEDIMIENTO FLOYD (D[1..n, 1..n], L[1..n,1..n])
begin
        For i = 1 to n do
            For j = 1 to n D[i,j] = L[i,j]
        For i = 1 to n D[i,i] = 0
        For k = 1 to n do
            For i = 1 to n do
                For j = 1 to n do
                  If D[i,k] + D[k,j] < D[i,j]
                  Then D[i,j] = D[i,k] + D[k,j]
End
```

Es obvio que este algoritmo consume un tiempo $O(n^3)$. Para resolver el problema también podemos usar el algoritmo de Dijkstra, entonces aplicaríamos ese algoritmo n veces, cada una de ellas eligiendo un nodo diferente como origen. Si queremos usar la versión de Dijkstra que trabaja con una matriz de distancias, el tiempo de cálculo total esta en n x $O(n^2)$, es decir en $O(n^3)$. El orden es el mismo que para el algoritmo de Floyd, pero la simplicidad de este supone que, en la práctica, probablemente sea más rápido.
Si queremos saber por dónde va el camino más corto, y no solo su longitud, podemos usar una segunda matriz P, inicialmente con valores cero, en la que P(i,j) al final contendrá el valor de la última iteración que causó un cambio en D(i,j). Si P(i,j) = 0, el camino mínimo entre i y j será directo a través del arco (i,j). Así, si el bucle más interno del algoritmo de Floyd lo cambiamos para que haga que

```
If D(i,k) + D(k,j) < D(i,j)
Then  D(i,j) = D(i,k) + D(k,j)
            P(i,j) = k
```

nos queda el siguiente algoritmo,

```
PROCEDIMIENTO MINIMO
Begin
  For i = 1 to n do
      For j = 1 to n do begin
          D[i,j] = L[i,j];
          P[i,j] = 0
      End;
  For i = 1 to n do
     D[i,i] = 0
  For k = 1 to n do
       For i = 1 to n do
          For j = 1 to n do
                If D[i,k] + D[k,j] < D[i,j] then begin
                   D[i,j] = D[i,k] + D[k,j]
                    P[i,j] = k
          End
End
```

Para obtener los vértices intermedios en el camino mínimo entre i y j, hacemos

```
PROCEDIMIENTO NODOS
Begin
   k := P[i,j];
   If k = 0 then
          Return;
   Path (i,k);
   Writeln (k);
   Path (k,j)
End
```

Si el problema fuera encontrar los caminos entre cada dos vértices de un grafo, es decir, solo en saber si esos caminos existen o no, el algoritmo que podríamos diseñar a partir del de Floyd es el llamado Algoritmo de Warshall. El algoritmo de Warshall en definitiva permite conocer si dos vértices de un grafo están conectados o no.

El Problema de la Mochila 0-1

Recordemos que el Problema de la Mochila se plantea en los siguientes términos: Se tienen n objetos fraccionables y una mochila. El objeto i tiene peso w_i y una fracción x_i ($0 \leq x_i \leq 1$) del objeto i produce un beneficio $p_i x_i$. El objetivo es llenar la mochila, de capacidad M, de manera que se maximice el beneficio, es decir,

$$\text{Max: } \sum_{1 \leq i \leq n} p_i x_i$$

s.a:

$$\sum_{1\le i\le n} w_i x_i \le M$$
$$x_i \in [0, 1],\ 1 \le i \le n$$

Los pesos, w_i, y la capacidad son números naturales, los beneficios, p_i, son reales no negativos.

Una variante importante de este problema es el que se conoce con el nombre de Problema de la Mochila 0-1, en el que xi sólo toma valores 0 ó 1, indicando que el objeto se deja fuera o se mete en la mochila, y con un planteamiento similar al anterior,

$$\text{Max: } \sum_{1\le i\le n} p_i x_i$$
$$\text{s.a:}$$
$$\sum_{1\le i\le n} w_i x_i \le M$$
$$x_i = 0, 1;\ 1 \le i \le n$$

a) Naturaleza N-etápica de los problemas. Sin dificultad, la solución al problema de la mochila 0-1 puede entenderse como el resultado de una sucesión de decisiones. Tenemos que decidir los valores de x_i , $1 \le i \le n$. Así, primero podríamos tomar una decisión sobre x_1, luego sobre x_2, etc. Una sucesión óptima de decisiones, verificando las restricciones del problema, será la que maximice la correspondiente función objetivo.

b) Comprobación del Principio del Óptimo. Notemos Mochila(l,j,Y) al siguiente problema,

$$\text{Max: } \sum_{1\le i\le j} p_i x_i$$
$$\text{s.a:}$$
$$\sum_{1\le i\le j} w_i x_i \le Y$$
$$x_i = 0, 1;\ 1 \le i \le j$$

es evidente que el problema de la mochila 0-1 con capacidad M y n objetos se representa por Mochila(1,n,M).

Como en el anterior caso, vamos a probar la aplicabilidad del Principio del Óptimo de Bellman a este problema por reducción al absurdo.

Sea $y_1, y_2, ..., y_n$ una sucesión óptima de valores 0-1 para $x_1, x_2, ..., x_n$ respectivamente. Si $y_1 = 0$, entonces $y_2, ..., y_n$ debe ser una sucesión óptima para el problema Mochila (2,n,M). Si no lo es, entonces $y_1, y_2, ..., y_n$ no es una sucesión óptima de Mochila (1,n,M). Si $y_1 = 1$, entonces $y_2, ..., y_n$ debe ser una sucesión óptima para el problema Mochila (2,n, M-w_1). Si no lo fuera, entonces habría otra sucesión de valores 0-1, $z_2, z_3, ..., z_n$ tal que

$$\sum_{2\le i\le n} w_i z_i \le M - w_1$$

y

$$\sum_{2\le i\le n} p_i z_i > \sum_{2\le i\le n} p_i y_i$$

Por tanto la sucesión $y_1, z_2, z_3, ..., z_n$ es una sucesión para el problema de partida con mayor valor que la supuesta, lo que es contradictorio. Consiguientemente se puede aplicar el Principio del Óptimo.

c) Construcción de una ecuación recurrente. Sea $g_j(y)$ el valor de una solución óptima del problema Mochila(j+1, n, y). Claramente $g_0(M)$ es el valor

de una solución óptima de Mochila(1,n,M). Las posibles decisiones para x_1 son 0 o 1 ($D_1 = \{0,1\}$). A partir del Principio del Óptimo se sigue que

$$g_0(M) = \text{Max } \{g_1(M), g_1(M-w_1) + p_1 \} \quad (3)$$

Como en el caso de los caminos mínimos, el Principio del Óptimo también puede aplicarse a estados y decisiones intermedios: Sea $y_1, y_2, ...,y_n$ una solución óptima del problema Mochila(1,n,M). Entonces para cada j, $1 \le j \le n$, $y_1, ..., y_j$ e $y_{j+1}, ..., y_n$ deben ser soluciones óptimas de los problemas

$$\text{Mochila } (1, j, \sum_{1 \le i \le j} w_i y_i)$$

y

$$\text{Mochila } (j+1, n, M - \sum_{1 \le i \le j} w_i y_i)$$

respectivamente. Esta observación nos permite generalizar (3) a

$$g_i(y) = \text{Max } \{g_{i+1}(y), g_{i+1}(y - w_{i+1}) + p_i\} \quad (4)$$

La aplicación recursiva del Principio del Óptimo produce una relación de recurrencia del tipo (4). Los algoritmos de PD resuelven estas recurrencias para obtener la solución del caso del problema que estemos considerando. La recurrencia (4) puede resolverse sabiendo que $g_n(y) = 0$ para todo y. A partir de $g_n(y)$ puede obtenerse $g_{n+1}(y)$ usando la formula (4) con i = n-1. Entonces usando $g_{n-1}(y)$ podemos obtener $g_{n-2}(y)$. Repitiendo esto, podremos determinar $g_1(y)$, y finalmente $g_0(M)$ usando (4) con i = 0.

De la misma forma que hemos razonado la obtención de la ecuación recurrente en base al Principio del Óptimo y al enfoque adelantado, podríamos haberlo hecho con respecto al enfoque atrasado, de modo que en este último, la formulación para la decisión x_i se haría en base a sucesiones de decisiones óptimas para $x_1, ..., x_{i-1}$, mirando por tanto hacia atrás la sucesión $x_1, ..., x_n$.

Con todo esto podemos escribir un algoritmo recursivo de manera inmediata,

```
FUNCION MOCHILA1(w,p,M)
Begin
  devuelve g(n,M)
End

FUNCION g(j, c)
Begin
  si j = 0 entonces devuelve 0
    sino
      si c < w[j]
        entonces devuelve g(j-1, c)
        sino
          si g(j-1, c) ≥ g(j-1, c-w[j]) + p[j]
            entonces
              devuelve g(j-1, c)
            sino
              devuelve g(j-1, c-w[j]) + p[j]
End
```

Es fácil observar que este esquema algorítmico no es muy eficiente porque un problema de tamaño n se reduce a dos subproblemas de tamaño (n-1). Luego cada uno de los dos subproblemas se reduce a otros dos, y así sucesivamente. Por tanto, se obtiene un algoritmo exponencial. Sin embargo, el número total de subproblemas a resolver no es tan grande.
En efecto, la función g tiene dos parámetros. Con el primero puede tomar n valores distintos y con el segundo, M valores. Luego sólo hay nM problemas diferentes. Deducimos entonces que si lleváramos a efecto este enfoque recursivo que acabamos de ver, el algoritmo estaría generando y resolviendo el mismo problema muchas veces, repitiendo en consecuencia muchas veces los mismos cálculos. Para evitar esa repetición de cálculos, las soluciones de los subproblemas se deben almacenar en una tabla, es decir, en una matriz n×M.
Así, si tuviéramos: n = 3, M = 15, (p_1, p_2, p_3) = (38, 40, 24) y (w_1, w_2, w_3) = (9, 6, 5), la anterior expresión (4) nos proporcionaría la siguiente tabla,

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$p_1 = 9$	0	0	0	0	0	0	0	0	0	38	38	38	38	38	38	38
$p_2 = 6$	0	0	0	0	0	0	40	40	40	40	40	40	40	40	40	78
$p_3 = 5$	0	0	0	0	0	24	40	40	40	40	40	64	64	64	64	78

que también podemos obtener fácilmente a partir del algoritmo que planteamos a continuación, con una complejidad O(nM)

```
ALGORITMO MOCHILA(w, p, M)
Principio
  para c = 0 hasta M hacer g[0,c] = 0
  para j = 1 hasta n hacer g[j,0] = 0
  para j = 1 hasta n hacer
    para c = 1 hasta M hacer
      si c < w[j]
        entonces
          g[j, c] = g[j-1, c]
        sino
          si g[j-1, c] ≥ g[j-1, c-w[j]] + p[j]
            entonces
              g[j, c] = g[j-1, c]
            sino
              g[j,c] = g[j-1,c-w[j]]+p[j]
Fin
```

El Problema del Viajante de Comercio

Según sabemos del tema de algoritmos greedy, este problema consiste en lo siguiente: Dado un grafo con longitudes no negativas asociadas a sus arcos, queremos encontrar el circuito más corto posible que comience y termine en un

mismo nodo, es decir, un camino cerrado que recorra todos los nodos una y solo una vez y que tenga longitud mínimo.
Sea G = (N,A) nuestro grafo dirigido. Como antes, tomaremos N = {1,2,...,n}, y notaremos la longitud de los arcos por L_{ij}, con L(i,i) = 0, L(i,j) $\geq$ 0 si i es distinto de j, y L(i,j) = ∞ si no existe el arco (i,j).
Suponemos, sin pérdida de generalidad, que el circuito comienza y termina en el nodo 1. Por tanto, está constituido por un arco (1, j), seguido de un camino de j a 1 que pasa exactamente una vez a través de cada nodo de N - {1, ,j}. Si el circuito es óptima, es decir, de longitud mínima, entonces ese es el camino de j a 1 y vale el Principio del Óptimo.
Consideremos un conjunto de nodos S $\subseteq$ N - {1} y un nodo mas i $\in$ N - S, estando permitido que i = 1 solo si S = N - {1}. Definimos el valor g(i,S) para cada índice i, como la longitud del camino más corto desde el nodo i al nodo 1 que pasa exactamente una vez a través de cada nodo de S. Usando esta definición, la longitud de un circuito óptima viene dada por g(1, N - {1}).
Por el Principio del Óptimo, vemos que

$$g(1, N-\{1\}) = \text{Min}_{2\leq j\leq n} [L_{1j} + g(j, N - \{1,,j\})] \qquad (5)$$

Mas generalmente, si i no es igual a 1, el conjunto S no es vacío, no coincide con el conjunto N - {1} e i$\notin$S,

$$g(i,S) = \text{Min}_{j\in S} [L_{ij} + g(j, S - \{j\})] \qquad (6)$$

Además,

$$g(i,\varnothing) = L_{i1}\ ,\ i = 2,3,...n$$

Por tanto, los valores de g(i,S) se conocen cuando S es vacío.
Podemos aplicar (6) para calcular la función g en todos los conjuntos S que contienen exactamente un nodo (que no es el 1). Entonces, de nuevo, podemos volver a aplicar (6) para calcular g en todos los conjuntos S que contienen dos nodos (distintos del 1), y así sucesivamente.
Cuando se conoce el valor de g[j,N - {1,j}] para todos los nodos j, excepto para el nodo 1, podemos usar (5) para calcular g(1,N - {1}) y, definitivamente, resolver el problema
El tiempo que consumirá este algoritmo puede calcularse como sigue,

a) Calcular g(j,$\varnothing$) supone la realización de n-1 consultas de una tabla.
b) Calcular todas las g(i,S) tales que 1 $\leq$ #S = k $\leq$ n-2, supone realizar,

$$(n-1) \times C_{n-2,k} \times k$$

adiciones (n-1 corresponden a los posibles valores que puede tomar la variable i, k provienen de los valores que puede tomar la variable j, y las combinaciones restantes son todos los conjuntos que podemos formar de n-2 elementos tomados de k en k.
c) Calcular g(1, N-{1}) implica n-1 adiciones.

Estas operaciones pueden usarse como referencia para calcular la eficiencia del algoritmo.
Así el tiempo que se lleva el algoritmo en cálculos es,

$$O[2(n-1) + \sum_{k=1..(n-2)} (n-1)\times k\times C_{n-2,k}\ ] = O(n^2 2^n)$$

ya que

$$\sum_{k=1..r} k\times C_{r,k} = r2^{r-1}$$

Este tiempo es bastante considerable pero, sin embargo, es bastante mejor que uno de O(n!) que es el que proporciona el método directo clásico sin contar con el enfoque de la PD. La siguiente tabla, ilustra esto

	Tiempo	**Tiempo**
N	**Método directo**	**PD**
	n!	$n^2 2^n$
5	120	800
10	3.628.800	102.400
15	$1.31x10^{12}$	7.372.800
20	$2.43x10^{18}$	419.430.400

A partir de ella vemos el enorme incremento que sufren el tiempo y el espacio necesarios conforme aumenta n. Por ejemplo, $20^2 2^{20}$ microsegundos es menos de siete minutos, mientras que 20! microsegundos supera las 77 mil años.
Para terminar de fijar ideas, consideremos un grafo definido por la siguiente matriz de distancias,

0	10	15	20
5	0	9	10
6	13	0	12
8	8	9	0

Entonces,

$$g(2, \varnothing) = c_{21} = 5;\ g(3, \varnothing) = c_{31} = 6 \text{ y } g(4, \varnothing) = c_{41} = 8$$

Usando (4) obtenemos

$g(2,\{3\}) = c_{23} + g(3, \varnothing) = 15;$ $\quad g(2,\{4\}) = 18$

$g(3,\{2\}) = 18;$ $\quad g(3,\{4\}) = 20$

$g(4,\{2\}) = 13;$ $\quad g(4,\{3\}) = 15$

A continuación, calculamos g(i,S) para conjuntos S de cardinal 2, $i \neq 1$, $1 \notin S$ e $i \notin S$:

$$g(2, \{3,4\}) = \text{Min}\ [c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\})] = 25$$

$$g(3, \{2,4\}) = \text{Min}\ [c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})] = 25$$

$$g(4, \{2,3\}) = \text{Min}\ [c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})] = 23$$

y finalmente,

$$g(1,\{2,3,4\}) = \text{Min}[c_{12}+g(2,\{3,4\}), c_{13}+g(3,\{2,4\}), c_{21}+g(4,\{2,3\})] = \text{Min}\{35,40,43\} = 35$$

que es la longitud del circuito buscado.

Multiplicación encadenada de matrices

Otro ejemplo de problema resoluble con la técnica de la PD es un algoritmo para resolver el problema de la multiplicación encadenada de matrices. Suponemos una sucesión (cadena) de n matrices $\{A_1, A_2, ...,A_n\}$ que se han de multiplicar, y deseamos calcular el producto $A_1 \times A_2 \times ... \times A_n$. Podemos calcular ese producto usando el algoritmo estándar de multiplicación de pares de matrices como un procedimiento, una vez que hayamos parentizado esa expresión para evitar ambigüedades acerca de cómo han de multiplicarse las matrices.
Un producto de matrices se dice que está completamente parentizado si está constituido por una sola matriz, o el producto completamente parentizado de dos matrices, cerrado por paréntesis. La multiplicación de matrices es asociativa, y por tanto todas las parentizaciones producen el mismo resultado.
Por ejemplo, si la cadena de matrices es la $\{A_1, A_2, A_3, A_4\}$, el producto $A_1 \times A_2 \times A_3 \times A_4$ puede parentizarse completamente de cinco formas distintas:

$(A_1 \times (A_2 \times (A_3 \times A_4)))$,
$(A_1 \times (A_2 \times A_3) \times A_4)$,
$((A_1 \times A_2) \times (A_3 \times A_4))$,
$((A_1 \times (A_2 \times A_3)) \times A_4)$, y
$(((A_1 \times A_2) \times A_3) \times A_4)$.

La forma en que parenticemos un producto de matrices puede tener un fuerte impacto en el costo de evaluar el producto. Consideremos primero el costo de multiplicar dos matrices. Como sabemos, si A es una matriz de dimensión pxq y B de dimensión qxr, el algoritmo estándar consumiría un tiempo proporcional a pxqxr.
Así, para ilustrar los diferentes costos asociados a las distintas parentizaciones de un producto de matrices, consideremos la cadena de tres matrices $\{A_1, A_2, A_3\}$, con dimensiones 10x100, 100x5 y 5x50 respectivamente. Si multiplicamos conforme a la parentización $((A_1 A_2)A_3)$, realizamos $10 \times 100 \times 5 = 5.000$ multiplicaciones para calcular el producto de la matriz (A_1A_2), 10x5, más otras $10 \times 5 \times 50 = 2.500$ multiplicaciones para multiplicar esta matriz por A_3, realizando en total 7.500 multiplicaciones. Así mismo, para el producto $(A_1 (A_2 A_3))$ realizamos $100 \times 5 \times 50 = 25.000$ multiplicaciones para calcular (A_2A_3), más $10 \times 100 \times 50 = 50.000$ multiplicaciones para multiplicar A_1 por esa matriz, teniendo que hacer definitivamente 75.000 multiplicaciones escalares. Por tanto el primer producto es 10 veces más rápido.
El problema de la multiplicación encadena de matrices puede, por tanto, plantearse como sigue: Dada una cadena de n matrices $\{A_1, A_2, ...,A_n\}$, donde para i = 1,2,...,n la matriz A_i tiene de dimensión p_{i-1} x p_i , parentizar completamente el producto de las matrices $A_1 \times A_2 \times ... \times A_n$ de tal forma que se minimice el número total de operaciones escalares que haya que realizar.

Para resolver el problema deberíamos considerar el mejor método posible, y surgen dos:

1) insertar los paréntesis de todas las formas posibles que sean significativamente diferentes), y
2) calcular para cada una el número de multiplicaciones escalares requeridas.

Veamos en primer lugar que la enumeración de todas las parentizaciones posibles no proporciona un método eficiente. Notemos el número de parentizaciones alternativas de una sucesión de n matrices por P(n). Como podemos dividir una sucesión de n matrices en dos (las k primeras y las k+1 siguientes) para cualquier k = 1,2,...,n-1, y entonces parentizar las dos subsucesiones resultantes independientemente, obtenemos la recurrencia:

$$\begin{aligned} P(n) &= 1 && \text{si } n = 1 \\ &= \sum_{k=1..n-1} P(k)\times P(n-k) && \text{si } n \geq 2 \end{aligned}$$

Puede demostrarse que la solución de esta ecuación es la sucesión de los Números de Catalan (Eugène Charles Catalan, 1814-1894):

$$P(n) = C(n-1)$$

donde

$$C(n) = (n+1)^{-1} C_{2n,n}$$

es $\Omega(4^n/n^{3/2})$, por lo que el número de soluciones es exponencial en n y, consiguientemente, el método de la fuerza bruta es una pobre estrategia para determinar la parentización óptima de una cadena de matrices.

Estudiemos por tanto cual es la estructura de una parentización óptima viendo si es posible que nos basemos en la metodología que nos proporciona la PD.

La primera etapa de la técnica de la PD es comprobar la naturaleza n-etápica del problema. Pero en el caso que nos ocupa esto es evidente, debido a que podemos asociar cada etapa al producto por una nueva matriz.

Con esto podemos pasar a comprobar la verificación del Principio del Óptimo, antes de ver cómo podríamos definir una ecuación recurrente. Ahora bien, en el problema de la multiplicación encadenada de matrices, podemos realizar esta segunda etapa como sigue.

Notemos $A_{i..j}$ la matriz que resulte de evaluar el producto $A_i \times A_{i+1} \times ... \times A_j$. Una parentización óptima del producto $A_1 \times A_2 \times ... \times A_n$ divide el producto entre A_k y A_{k+1} para algún entero k en el rango $1 \leq k \leq n$. Es decir, para algún valor de k, primero calculamos las matrices $A_{i..k}$ $A_{k+1..n}$ y entonces las multiplicamos para obtener el producto final $A_{1..n}$. El costo de esta parentización óptima es así el costo de calcular el producto $A_{i..k}$, más el costo de calcular el de $A_{k+1..n}$ más el costo de multiplicar ambas.

La clave está en que la parentización de la primera subcadena $A_1 \times ... \times A_k$ dentro de la parentización óptima de $A_1 \times A_2 \times ... \times A_n$ debe ser una parentización óptima de $A_1 \times A_2 \times ... \times A_k$. ¿Por qué?. Si hubiera una forma menos costosa de parentizar $A_1 \times A_2 \times ... \times A_k$ entonces, sustituyendo esa parentización en la parentización óptima de $A_1 \times A_2 \times ... \times A_n$, obtendríamos otra parentización de $A_1 \times A_2 \times ... \times A_n$ cuyo costo sería inferior a la de la óptima : una contradicción. Un razonamiento

similar es válido para la parentización de la segunda sub-cadena $A_{k+1} \times ... \times A_n$ en la parentización óptima de $A_1 \times A_2 \times ... \times A_n$. Por tanto también debe ser una parentización óptima de $A_{k+1} \times ... \times A_n$.
así, una solución óptima para un caso del problema de la multiplicación encajada de matrices contiene las soluciones de los subproblemas del caso considerado, es decir, se está cumpliendo el Principio del Óptimo de Bellman.
A continuación vamos a definir recursivamente el valor de una solución óptima en términos de las soluciones óptimas de los subproblemas. Para nuestro problema, tomamos como subproblemas los de determinar el mínimo costo de una parentización de $A_i \times A_{i+1} \times ... \times A_j$, $1 \le i \le j \le n$.. Sea m[i,j] el mínimo número de multiplicaciones escalares necesarias para calcular la matriz A; el costo de la forma menos costosa de calcular $A_{1..n}$ deberá ser por tanto m[1,n].
Se puede definir m[i,j] recursivamente como sigue. Si i = j, la cadena consiste de un solo elemento $A_{i..j} = A_i$, por lo que no hay que hacer ninguna multiplicación para calcular el producto. así m[i,i] = 0 para i = 1,2,...,n.
Para calcular m[i,j] cuando i < j, nos aprovecharemos de la estructura de una solución óptima a partir de la primera etapa. Supongamos que la parentización óptima hace el producto $A_i \times A_{i+1} \times ... \times A_j$, tomando como punto de división A_k y A_{k+1}, con $i \le k < j$. Entonces m[i,j] es igual al costo mínimo de calcular los subproductos $A_{i..k}$ y $A_{k+1..j}$ más el costo de multiplicar esas dos matrices resultantes. Como calcular la matriz producto $A_{i..k} \cdot A_{k+1..j}$ supone hacer $p_{i-1} \cdot p_k \cdot p_j$ multiplicaciones, obtenemos:

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j$$

Esta ecuación recurrente supone que conocemos el valor de k, lo que no es cierto. Hay solo j - i posibles valores para k: k = i, i+1,...,j-1. Como la parentización óptima debe usar uno de estos valores de k, solo necesitamos comprobarlos para encontrar el mejor.
Así nuestra definición recursiva del costo mínimo de la parentización del producto de $A_i \times A_{i+1} \times ... \times A_j$ es:

$$m[i,j] = \begin{cases} \text{Min}_{1 \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j\} & \text{si } i < j \\ 0 & \text{si } i = j \end{cases} \quad (5)$$

donde

m[i, k] = Costo óptimo de A_i x … x A_k

m[k+1, j] = Costo óptimo de A_{k+1} x … x A_j

$p_{i-1}p_kp_j$ = Costo de (A_i x … x A_k) x (A_{k+1} x … x A_j)

Los valores m[i,j] dan los costos de las soluciones óptimas de los subproblemas. Definimos también s[i,j] como el índice k en el que podemos dividir el producto $A_i \times A_{i+1} \times ... \times A_j$ para obtener una parentización óptima. Esto es, s[i,j] es aquel k tal que

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j.$$

Llegados a este punto, para el cálculo de los costos óptimas, es relativamente fácil preparar un algoritmo recursivo que, basado en la última ecuación recurrente, calcule el costo mínimo m[1,n] asociado a la multiplicación de $A_1 \times A_2 \times ... \times A_n$. El inconveniente es que este algoritmo va a consumir un tiempo exponencial.
En efecto, el algoritmo quedaría como,

MATRICES-ENCADENADAS-RECURSIVO (p,i,j)
If i = 1
Then Return 0
m[i,j] = ∞
For k = 1 to j -1 do
  q = Matrices-Encadenadas-Recursivo (p,i,k)
    + Matrices-Encadenadas-Recursivo (p,k+1,j) + $p_{i-1} \cdot p_k \cdot p_j$
  If q < m[i,j]
  Then m[i,j] = q
Return m[i,j]

Entonces podemos plantear la siguiente recurrencia,

$$T(1) = 1$$
$$T(n) = 1 + \sum_{k=1..n-1}(T(k) + T(n-k) + 1) \text{ para } n > 1$$

Nótese que para i = 1,2, ..., n-1, cada termino T(i) aparece una vez como T(k) y una vez como T(n-k), por tanto la anterior ecuación puede rescribirse como

$$T(n) = 2 \cdot \sum_{i=1..n-1} T(i) + n$$

Ahora bien, como $T(1) \geq 1 = 2^o$, inductivamente para $n \geq 2$ tenemos

$$T(n) = 2 \cdot \sum_{i=1..n-1} 2^{i-1} + n \geq 2 \cdot \sum_{i=0..n-1} 2^i + n \geq 2(2^{n-1} - 1) + n = (2^n - 2) + n \geq 2^{n-1}$$

Con lo que se demuestra que el tiempo es al menos exponencial en n.
Nótese que tenemos un número relativamente bajo de subproblemas: un problema para cada elección de i y j que satisfaga $1 \leq i \leq j \leq n$, es decir en total $C_{n,2} + n = O(n^2)$. Un algoritmo recursivo puede tratar el mismo subproblema muchas veces en las diferentes ramas de su árbol de recursión, y este solapamiento es una de las propiedades interesantes para la aplicabilidad de la PD. En lugar de calcular las soluciones de (5) de forma recursiva, resolveremos de acuerdo con la técnica de la PD aprovechando su metodología tabular. El siguiente algoritmo supone que la matriz A_i tiene dimensiones $p_{i-1} \cdot p_i$, para cualquier i = 1,2,...,n. El input es una sucesión $\{p_o, p_1, ..., p_n\}$ de longitud n+1, es decir leng[p] = n+1. El procedimiento usa una tabla auxiliar m[1..n,1..n] para ordenar los m[i,j] costos y una tabla auxiliar s[1..n,1..n] que almacena que índice de k alcanza el costo óptima al calcular m[i,j].

ORDEN-CADENA-MATRICES (p)
n = leng[p] - 1
For i = 1 to n do m[i,i] = 0
For l = 2 to n do
  For i = 1 to n - l +1 do
    j = i + l -1
    m[i,j] = ∞

```
    For k = i to j - 1 do
      q = m[i,k] + m[k+1,j] + p_{i-1}·p_k ·p_j
      If q < m[i,j]
      Then m[i,j] = q; s[i,j] = k
Return m y s.
```

El algoritmo rellena la tabla m de un modo que corresponde a la solución del problema de la parentización en cadenas de matrices de longitudes crecientes. La ecuación (5) muestra que el costo m[i,j] de calcular el producto encadenado de j-i+1 matrices depende solo de los costos de calcular los productos de las cadenas con menos de j-i+1 matrices. Es decir, para k = i, i+1,..., j-1, $A_{i..k}$ es un producto de k-i+1 < j-i+1 matrices, y la matriz $A_{k+1..j}$ es un producto de j-k < j-i+1 matrices.
El algoritmo primero toma m[i,i] = 0, i = 1,2,...,n (los mínimos costos para las cadenas de longitud 1) y entonces, durante la primera ejecución del siguiente lazo, usa (5) para calcular m[i,i+1], i = 1,2,...,n-1 (los mínimos costos para las cadenas de longitud l = 2). La segunda vez que pasa el lazo, calcula m[i,i+2], i = 1,2,...,n-2 (los mínimos costos para las cadenas de longitud l = 3), y así sucesivamente.
Una simple inspección a la estructura encajada de los lazos en el anterior algoritmo demuestra que este tiene una eficiencia de $O(n^3)$, ya que hay 3 lazos en l, i y k que, en el peor caso, pueden llegar a tomar el valor n. Por tanto deducimos que este algoritmo es mucho más eficiente que el anterior método en tiempo exponencial que enumeraba todas las posibles parentizaciones, y chequeaba cada una de ellas.
Aunque el anterior algoritmo determina el número óptima de multiplicaciones necesarias para calcular un producto encajado de matrices, no explica como multiplicar las matrices. La siguiente etapa de la técnica de la PD es la de la construcción de una solución óptima a partir de la información calculada.
En nuestro caso, usamos la tabla s[1..n,1..n] para determinar la mejor forma de multiplicar las matrices. Cada elemento s[i,j] almacena el valor k tal que divide óptimamente la parentización de A_{i-1} x A_{i+1} x ... xA_j. Por tanto, sabemos que la multiplicación óptima de matrices final para calcular $A_{1..n}$ es

$$A_{1..s[1,n]} \text{ x } A_{s[1,n]+1..n}$$

La anterior multiplicación de matrices puede efectuarse recursivamente, puesto que s[1,s[1,n]] determina la última multiplicación de matrices al calcular A, y s[s[1,n]+1,n] la última multiplicación de matrices al calcular $A_{1..s[1,n]}$.
El siguiente procedimiento recursivo calcula el producto encadenado de matrices $A_{i..j}$ dadas las matrices de la cadena A = {A_1, A_2,..., A_n }, la tabla s calculada por el algoritmo Orden-Cadena-Matrices, y los índices i y j. El algoritmo usa el procedimiento clásico MULT (A,B) de multiplicación de dos matrices.

```
MULTIPLICA-CADENA-MATRICES (A,s,i,j)
If j > i
Then X = Multiplica-Cadena-Matrices (A,s,i,s[i,j])
   Y = Multiplica-Cadena-Matrices (A,s,s[i,j]+1,j)
   Return MULT (X,Y)
Else Return A_i
```

El problema del “play off”

Una curiosa aplicación de la metodología tabular que inspira la técnica de la PD es la siguiente, que se conoce con el nombre de Problema del “Play Off”. Este problema no es un claro exponente de la aplicabilidad de la PD, sin embargo arroja claridad a las anteriores explicaciones que henos dado sobre porque la metodología tabular de la PD proporciona mejores resultados que los potenciales de un enfoque recursivo. El problema se plantea en los siguientes términos,

Supongamos dos equipos A y B que juegan una final en la que se quiere saber cuál será el primero en ganar n partidos, para cierto n particular, es decir, deben jugar como mucho 2n-1 partidos. El problema del play off es tal final cuando el número de partidos necesarios es n = 4. Se puede suponer que ambos equipos son igualmente competentes y que, por tanto, la probabilidad de que A gane algún partido concreto es 1/2.

Sea P(i,j) la probabilidad de que sea A el ganador final si A necesita i partidos para ganar y B j. Antes del primer partido, la probabilidad de que gane A es P(n,n), y es obvio que

$$P(0,i) = 1 \text{ si } 1 \leq i \leq n$$
$$P(i,0) = 0 \text{ si } 1 \leq i \leq n$$

No estando definida P(0,0).

Así, como A puede ganar cualquier partido con probabilidad 1/2 y perderlo con idéntica probabilidad, tenemos

$$P(i,j) = [P(i-1,j) + P(i,j-1)]/2,\ i \geq 1,\ j \geq 1$$

es decir,

$$\begin{aligned} P(i,j) &= 1 && \text{si } i = 0 \text{ y } j > 0 \\ &= 0 && \text{si } i > 0 \text{ y } j = 0 \\ &= [P(i-1,j)+P(i,j-1)]/2 && \text{si } i > 0 \text{ y } j > 0 \end{aligned} \qquad (6)$$

y podemos calcular P(i,j) recursivamente.

Sea T(k) el tiempo necesario en el peor de los casos para calcular P(i,j), con i + j = k.

Si diseñáramos un algoritmo recursivo para realizar los cálculos que nos interesan, tendríamos que,

$$T(1) = c$$
$$T(k) = 2T(k-1) + d,\ k > 1$$

donde c y d son constantes.

La resolución, fácil, de esa recurrencia nos dice que T(k) consume un tiempo en $O(2^k) = O(4^n)$, si i = j = n, lo que sin lugar a dudas demuestra que no es un método demasiado práctico cuando se supone un n grande.

Intentemos sacar ventaja de la estructura encajada de los sub-problemas, es decir, de la metodología tabular subyacente en la técnica d la PD. Para ello notamos de (6) que otra forma de calcular P(i,j) es rellenando una tabla en la que los elementos de la última fila serían todos ceros, y los de la última columna todos uno. Cualquier otra casilla de la tabla es la media de la casilla anterior y la que está a su derecha.

Por tanto una forma valida de rellenar la tabla es proceder en diagonales, comenzando por la esquina sureste y procediendo hacia arriba a la izquierda a lo largo de las diagonales, que representan casillas con valores constantes de i+j. Así, con los datos del problema, partiríamos de una matriz como la siguiente

				1	**4**
				1	**3**
				1	**2**
				1	**1**
0	0	0	0	**XXX**	**0**
4	**3**	**2**	**1**	**0**	

Teniendo en cuenta (6), podríamos obtener en primer lugar que

$$P(1,1) = (P(0,1) + P(1,0))/2 = (1+0)/2 = ½$$

y sucesivamente toda la tabla siguiente

1/2	21/32	13/16	15/16	1	**4**
11/32	1/2	11/16	7/8	1	**3**
3/16	5/16	1/2	3/4	1	**2**
1/16	1/8	1/4	1/2	1	**1**
0	0	0	0	**XXX**	**0**
4	**3**	**2**	**1**	**0**	

El siguiente algoritmo realiza estos cálculos

ALGORITMO PLAYOFF

```
Begin
  For s := 1 to i+j do begin
     P[0,s] = 1.0;
     P[s,0] = 0.0;
     For k = 1 to s-1 do
        P[k,s-k] = (P[k-1,s-k] + P[k,s-k-1])/2.0
  end;
  Return (P[i,j])
End
```

Es evidente que El lazo más externo se lleva un tiempo $O(\Sigma s)$, para s = 1,2,..,n, es decir, un tiempo en $O(n^2)$ cuando i+j = n. Por tanto, el uso de la PD supone un tiempo $O(n^2)$, muy inferior al del método directo.