Apuntes de Algorítmica

Andrés Marzal
María José Castro
Pablo Aibar

Borrador
6 de febrero de 2007

Capítulo 9
RAMIFICACIÓN Y PODA

Ramificación y poda guarda relación con el método de búsqueda con retroceso en tanto que propone la búsqueda de una solución factible (óptima) explorando ordenadamente un árbol de estados y eliminando los nodos que no pueden conducir a la solución buscada. La estrategia de ramificación y poda se distingue de la búsqueda con retroceso tanto en el orden de recorrido del árbol de estados como en la poda de estados. Quizá sea más apropiado denominar al método «selección, ramificación y poda», pues éstos son sus pasos básicos:

- **Selección**. En cada instante hay una serie de estados no visitados a los que denominamos «**estados activos**» y se selecciona uno de ellos para explorar: el que parezca más prometedor. Este criterio conduce a una exploración por «primero el mejor». (La búsqueda con retroceso, por contra, sigue un criterio «último en entrar, primero en salir», propio de la pila con que se gestiona la exploración por primero en profundidad.)
- **Ramificación**. Es el proceso por el que un estado, que representa un conjunto de soluciones, se divide y genera otros estados que representan a subconjuntos suyos.
- **Poda**. Es la supresión de aquellos estados activos que no pueden contener la solución factible óptima. Consideraremos diferentes criterios de poda: **poda por factibilidad** (que ya conocemos por la búsqueda co retroceso), **poda por cota optimista**, **poda por cota pesimista** y **poda basada en la memorización de resultados intermedios** (esta última propia de la aplicación de ramificación y poda a problemas abordables por programación dinámica).

«Branch and bound» se debería traducir por «ramificación y acotación», pero se ha popularizado el término «ramificación y poda».

Es frecuente aplicar esta técnica de búsqueda a problemas para los que no se conoce un algoritmo eficiente. En tales casos, la estrategia de ramificación y poda presenta un

coste computacional exponencial para el peor de los casos, pero en la práctica, con un diseño adecuado de sus elementos y la debida atención a los detalles de implementación, el tiempo de ejecución puede resultar permisible para problemas de talla moderada. Es más, ciertas variantes de la técnica pueden sacrificar controladamente la optimalidad del resultado para acelerar el tiempo de respuesta, es decir, proporcionan algoritmos de aproximación. Pero también es posible aplicar la técnica de «ramificación y poda» a problemas para los que ya se conoce un algoritmo eficiente: un diseño apropiado puede asegurar un coste para el peor de los casos igual al de los métodos ya conocidos y, en la práctica, acelerar notablemente el cálculo.

9.1. El problema de la mochila discreta

Recordemos ahora el enunciado del problema de la mochila discreta, problema que resolvimos en la sección **??**, en el capítulo dedicado a la programación dinámica. Disponemos de N objetos con pesos positivos $w_1, w_2, \ldots, w_N$ y valores respectivos $v_1, v_2, \ldots, v_N$. La mochila soporta un peso máximo W. Deseamos cargar la mochila con algunos de los objetos de modo que la suma de sus valores sea máxima, pero sin exceder la capacidad de carga de la mochila.

Las soluciones factibles son selecciones de objetos cuyo peso total no excede la capacidad de carga de la mochila:

$$X = \left\{ (x_1, x_2, \ldots x_N) \in \{0,1\}^N \;\middle|\; \sum_{1 \le i \le N} x_i w_i \le W \right\}.$$

La función objetivo es

$$f((x_1, x_2, \ldots, x_N)) = \sum_{1 \le i \le N} x_i v_i,$$

y deseamos encontrar

$$\arg \max_{(x_1, x_2, \ldots, x_N) \in X} \sum_{1 \le i \le N} x_i v_i.$$

9.1.1. Estados y ramificación

Modelaremos el problema siguiendo los mismos pasos que adoptamos al abordar problemas con la estrategia de búsqueda con retroceso. Empezaremos por definir el concepto de estado, que representa un conjunto de soluciones (factibles o no). Un estado es un conjunto de soluciones factibles representado por el prefijo que tienen en común. El estado $(x_1, x_2, \ldots, x_k)$ representa al conjunto $\{(y_1, y_2, \ldots, y_N) \in X \mid y_1 = x_1, y_2 = x_2, \ldots, y_k = x_k$. El número de soluciones (factibles o no) incluidas en el conjunto que representa un estado es 2^{N-k}. Cada solución resulta de sustituir la secuencia de interrogantes por una combinación diferente de ceros y unos.

Pongamos un caso particular: supongamos que hay 5 objetos con pesos $w_1 = 12$, $w_2 = 5$, $w_3 = 6$, $w_4 = 2$ y $w_5 = 6$ y valores $v_1 = 10$, $v_2 = 2$, $v_3 = 3$, $v_4 = 4$ y $v_5 = 2$.

Un estado como $(0,1,0,?)$ representa un conjunto con cuatro soluciones, no todas ellas factibles: $\{(0,1,0,0,0),(0,1,0,0,1),(0,1,0,1,0),(0,1,0,1,1)\}$. Una solución factible como $(0,1,0,0,0)$ pertenece al conjunto representado por un estado como $(0,1,0,?)$. Así, serán lícitas expresiones como $(0,1,0,0,?) \subset (0,1,0,?)$ o $(0,1,0,0,0) \in (0,1,0,?)$.

Hay cierta ambigüedad en las tuplas con N valores fijos: representan estados que contienen una sola solución (conjunto unitario) y describen , a la vez, la propia solución. El contexto decidirá cuándo nos referimos al estado y cuándo a la solución. Una expresión como $(0,1,0,0,0) \in (0,1,0,0,0)$ debería interpretarse como «la solución $(0,1,0,0,0)$ pertenece al conjunto $\{(0,1,0,0,0)\}$. Estos estados recibirán el nombre de **estados unitarios**.

Un estado del que se sepa que no contiene solución factible alguna es un **estado no factible** (por ejemplo, $(0,1,1,?)$). Un estado del que se sabe que el valor de la función objetivo sobre sus soluciones factibles no es mejor que el que ya conocemos para cierta solución, será un **estado no prometedor**.

Si $k < N$, Un estado $(x_1, x_2, \ldots, x_k, ?)$ puede dividirse en dos nuevos estados por medio de una función a la que denominaremos *branch*:

$$branch((x_1, x_2, \ldots, x_k, ?)) = \{(x_1, x_2, \ldots, x_k, 0, ?), (x_1, x_2, \ldots, x_k, 1, ?)\}.$$

En el ejemplo que estamos considerando, la ramificación del estado $(0,1,0,?)$ da lugar a un conjunto con dos estados: $\{(0,1,0,0,?),(0,1,0,1,?)\}$.

La función de ramificación induce un árbol a partir del estado inicial, $(?)$, que representa al conjunto completo de soluciones. Las hojas del árbol son soluciones (factibles o no). El árbol de estados completo para el problema con 5 objetos se muestra en la figura 9.1. Una exploración exhaustiva del árbol que evalúe la función objetivo sobre cada hoja que represente una solución factible permite encontrar la solución óptima. El coste de tal procedimiento es, en nuestro problema, $\Theta(2^N)$.

9.1.2. El conjunto de estados activos y el ciclo «selecciona, ramifica y poda»

Vamos a disponer los estados inexplorados en un conjunto al que denominamos **conjunto de estados activos** y que denotaremos con A. El conjunto A no es una estructura estática: evoluciona con la ejecución del método de búsqueda, que es un procedimiento iterativo. Denotaremos con $A^{(i)}$ el valor de A tras la iteración i-ésima y con $A^{(0)}$ su valor inicial. El conjunto se inicializa con un único estado:

$$A^{(0)} = \{(?)\}.$$

El método de ramificación y poda consiste en, reiteradamente, extraer un estado de A, introducir en A los estados resultantes de ramificar ese estado y eliminar de A los estados no prometedores. En nuestro caso, en la primera iteración sólo puede seleccionarse un estado: el estado inicial. Su ramificación da lugar a dos estados: $(0,?)$ y $(1,?)$. En principio, ambos estados ingresan en A. Pero a continuación, eliminamos de A los estados no prometedores. En nuestro caso, el estado $(1,?)$ no es prometedor, pues $w_1 = 12$ es mayor

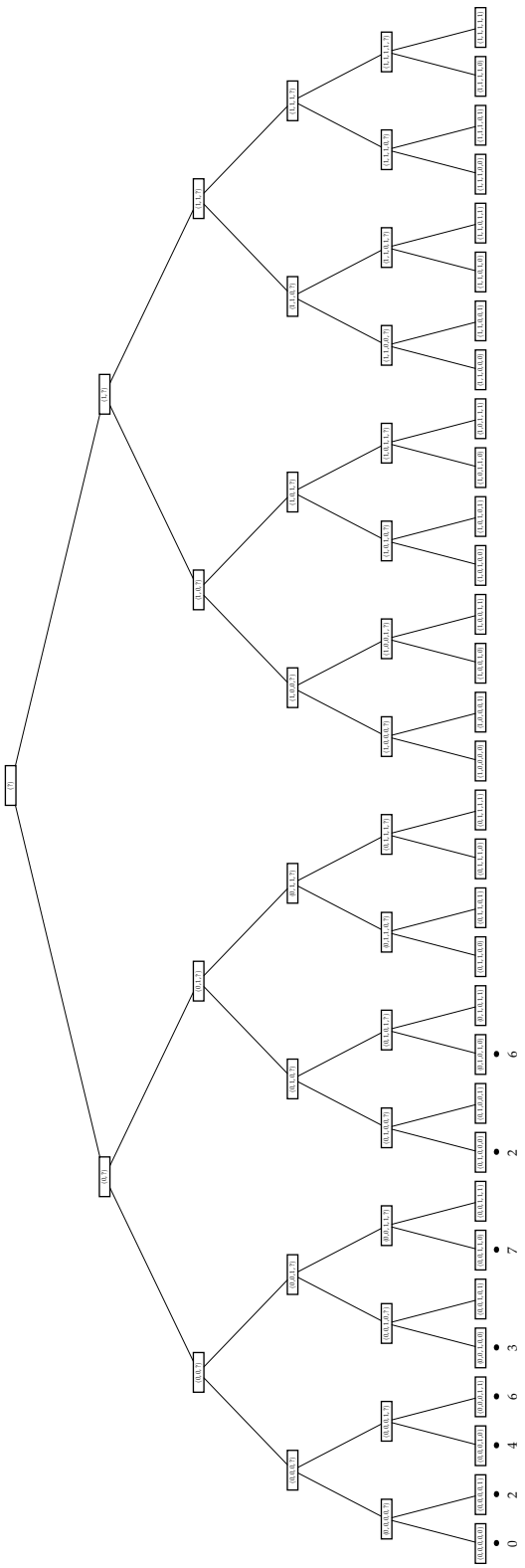


Figura 9.1: Árbol de estados completo (incluyendo estados unitarios con soluciones no factibles) para el problema de la mochila discreta con 5 objetos. Marcamos las soluciones factibles con un punto y el valor de la función objetivo.

que $W = 10$. La lista de estados activos queda, pues, así:

$$A^{(1)} = \{(0, ?)\}.$$

Al eliminar este estado de A estamos efectuando una «poda por factibilidad». Pero hemos de tener en cuenta que siempre resultará más eficiente no insertar el nuevo estado en A que insertarlo en A para, acto seguido, podarlo.

En la siguiente iteración se extrae de A su único estado, $(0, ?)$, se ramifica y se insertan los estados descendientes en A:

$$A^{(2)} = \{(0, 0, ?), (0, 1, ?)\}.$$

Ahora que A contiene más de un estado, ¿cuál seleccionamos? Hay diferentes criterios y cada uno sugiere para A una estructura de datos adecuada:

- Primero en profundidad: A es una pila o cola LIFO.
- Primero en anchura: A es una cola FIFO.
- Primero el mejor: A es una cola de prioridad.

Nótese que el primer criterio nos conduce a un proceso equivalente al de búsqueda con retroceso. Si seguimos el tercer criterio, ya no basta con calificar cada estado como prometedor o no prometedor: si es prometedor, hemos de cuantificar el grado en que nos parece prometedor para decidir cuál de todos es *más* prometedor. Hemos de definir, pues, una función que puntúe los estados de acuerdo con lo prometedores que nos parezcan. Ya que estamos maximizando, consideraremos «estado más prometedor» a aquél para el que la «función de puntuación» de conjuntos de soluciones alcance un valor máximo. Hay infinidad de funciones de puntuación posibles. De cuál «escojamos» depende críticamente la celeridad con que alcancemos la mejor solución factible.

Diseñaremos ahora dos funciones de puntuación diferentes para nuestro problema, cada una de las cuales conducirá a una exploración diferente del árbol de estados (y más adelante presentaremos otras funciones de puntuación).

9.1.3. Una función voraz para la puntuación de estados

Los estados no unitarios son de la forma $(x_1, x_2, \ldots, x_k, ?)$. Usaremos el término «parte conocida» para los k primeros valores del estado, y «parte desconocida» para los $N - k$ valores que aún no han sido asignados. Ante un estado de la forma $(x_1, x_2, \ldots, x_k, ?)$ nos preguntamos qué valores de la parte desconocida proporcionan mayor valor sin desbordar la capacidad de la mochila. Responder a esa pregunta con exactitud es tanto como resolver una instancia del problema de la mochila discreta con los últimos $N - k$ objetos y una capacidad de carga $W - \sum_{1 \leq i \leq k} x_i w_i$. Pero, en principio, resolverla supone un coste temporal considerable, así que nos conformaremos con obtener una aproximación del valor óptimo si podemos hacerlo eficientemente.

Un algoritmo voraz, por ejemplo, puede asignar valor a los $N - k$ elementos indeterminados de la tupla en tiempo $O(N - k)$. No siempre encontrará la solución óptima,

como ya vimos en su momento, pero cabe esperar que tienda a otorgar mayor puntuación al estado más prometedor. Sigamos desarrollando el ejemplo. En $A^{(2)}$ tenemos por el momento dos estados cuyos tres últimos elementos están indeterminados. Si vamos escogiendo objetos «de izquierda a derecha» mientras quepan en la mochila, el estado $(0,0,?)$ se puntuará con el valor que tiene la carga $(0,0,1,1,0)$, es decir, valor 7. El estado $(0,1,?)$, por su parte, se completará vorazmente formando la solución $(0,1,0,1,0)$. Indicamos la puntuación asignada a cada estado activo sobre el mismo:

$$A^{(2)} = \{\overbrace{(0,0,?)}^{7}, \overbrace{(0,1,?)}^{6}\}.$$

Para mayor claridad, expresaremos la función de puntuación separando sus dos componentes: el valor que aportan los objetos de la «parte conocida» y el que estimamos que pueden aportar los de la «parte desconocida»:

$$A^{(2)} = \{\overbrace{(0,0,?)}^{0+7=7}, \overbrace{(0,1,?)}^{2+4=6}\}.$$

Seleccionamos y extraemos el estado más prometedor, $(0,0,?)$, lo ramificamos e insertamos los estados descendientes en A:

$$A^{(3)} = \{\overbrace{(0,0,0,?)}^{0+6=6}, \overbrace{(0,0,1,?)}^{3+4=7}, \overbrace{(0,1,?)}^{2+4=6}\}.$$

Repetimos el proceso seleccionando esta vez el estado $(0,0,1,?)$:

$$A^{(4)} = \{\overbrace{(0,0,0,?)}^{0+6=6}, \overbrace{(0,0,1,0,?)}^{3+0=3}, \overbrace{(0,0,1,1,?)}^{7+0=7}, \overbrace{(0,1,?)}^{2+4=6}\}.$$

Seleccionamos el estado $(0,0,1,1,?)$ y su ramificación produce dos soluciones «completas», $(0,0,1,1,0)$ y $(0,0,1,1,1)$. La segunda no es factible, así que no se tiene en cuenta. La solución factible $(0,0,1,1,0)$ ofrece un beneficio de valor 7. Memorizamos esta solución factible, que es la mejor vista hasta el momento, en una variable adicional $\hat{x}$ (sin necesidad de ingresar en A el estado que la representa):

$$A^{(5)} = \{\overbrace{(0,0,0,?)}^{0+6=6}, \overbrace{(0,0,1,0,?)}^{3+0=3}, \overbrace{(0,1,?)}^{2+4=6}\}, \qquad \hat{x} = \overbrace{(0,0,1,1,0)}^{f(\hat{x})=7}.$$

Mostramos gráficamente los pasos efectuados hasta el momento en la figura 9.2. ¿Puede alguno de los estados activos conducirnos a una solución factible de mayor valor? No es posible saberlo sin explorar cada uno de los estados activos. Hemos de continuar con la exploración. El resto de la traza, que detallamos a continuación, se ilustra gráficamente en las figuras 9.3 y 9.4.

Ahora hay dos estados con idéntica puntuación. Hemos de escoger uno y parece más conveniente ramificar el que está más próximo a ser una solución completa, ya que producirá más rápidamente nuevas soluciones factibles. Seleccionamos entonces el estado

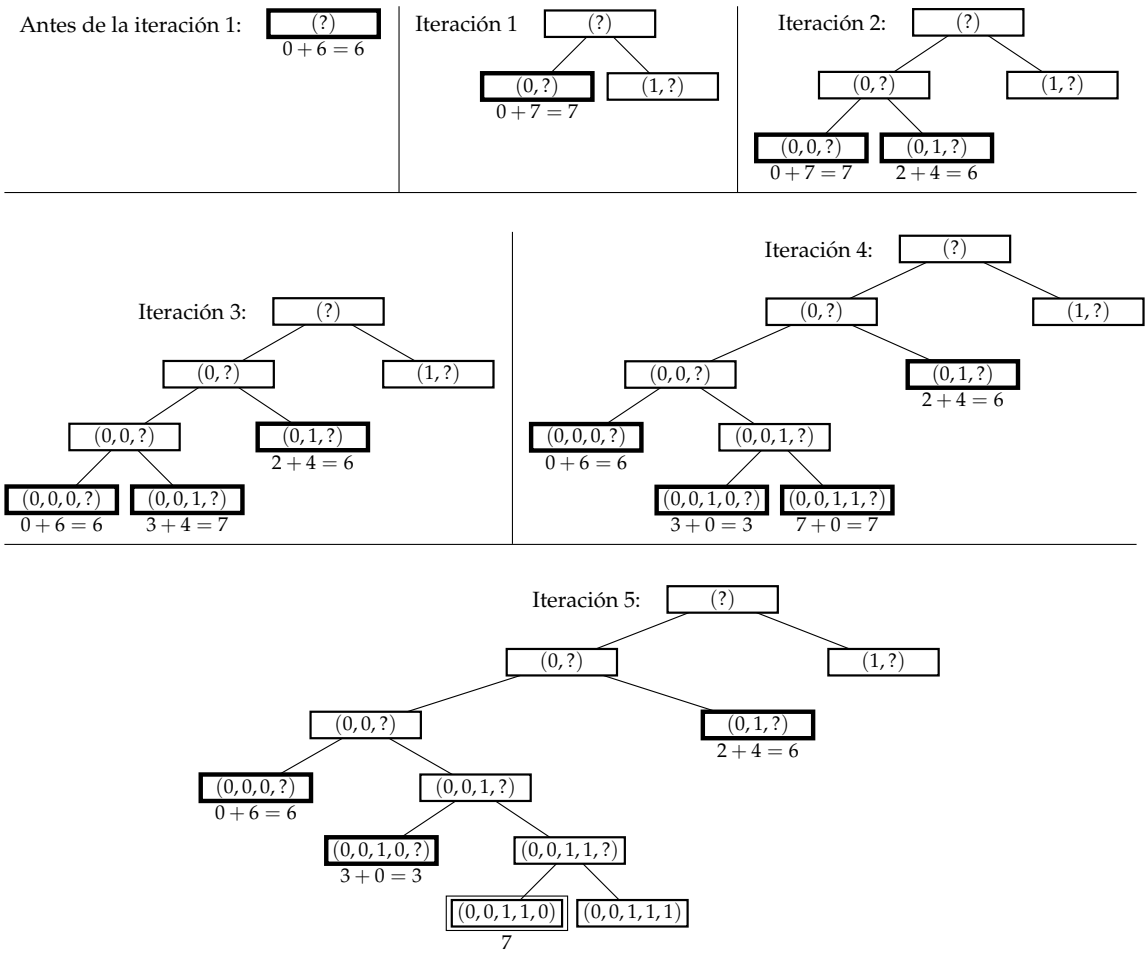


Figura 9.2: Estados explorados en el árbol en cada iteración hasta el instante en que se conoce una solución factible. Resaltamos con borde grueso los estados activos (elementos del conjunto $A^{(i)}$, donde i es la iteración actual) y con doble recuadro los estados unitarios visitados y que corresponden a soluciones factibles. La hojas del árbol con recuadro simple son estados generados y descartados por no factibles.

$(0,0,0,?)$. Su ramificación deja así el conjunto de estados activos:

$$A^{(6)} = \{\overbrace{(0,0,0,0,?)}^{0+2=2}, \overbrace{(0,0,0,1,?)}^{4+2=6}, \overbrace{(0,0,1,0,?)}^{3+0=3}, \overbrace{(0,1,?)}^{2+4=6}\}, \qquad \hat{x} = \overbrace{(0,0,1,1,0)}^{f(\hat{x})=7}.$$

Seleccionamos el estado $(0,0,0,1,?)$, que, ramificado, nos conduce a dos soluciones factibles, $(0,0,0,1,0)$ y $(0,0,0,1,1)$. El valor de la primera es 4 y el de la segunda, 6. No hemos mejorado, pues, el valor de la mejor solución factible vista hasta el momento, por lo que podemos desecharlas:

$$A^{(7)} = \{\overbrace{(0,0,0,0,?)}^{0+2=2}, \overbrace{(0,0,1,0,?)}^{3+0=3}, \overbrace{(0,1,?)}^{2+4=6}\}, \qquad \hat{x} = \overbrace{(0,0,1,1,0)}^{f(\hat{x})=7}.$$

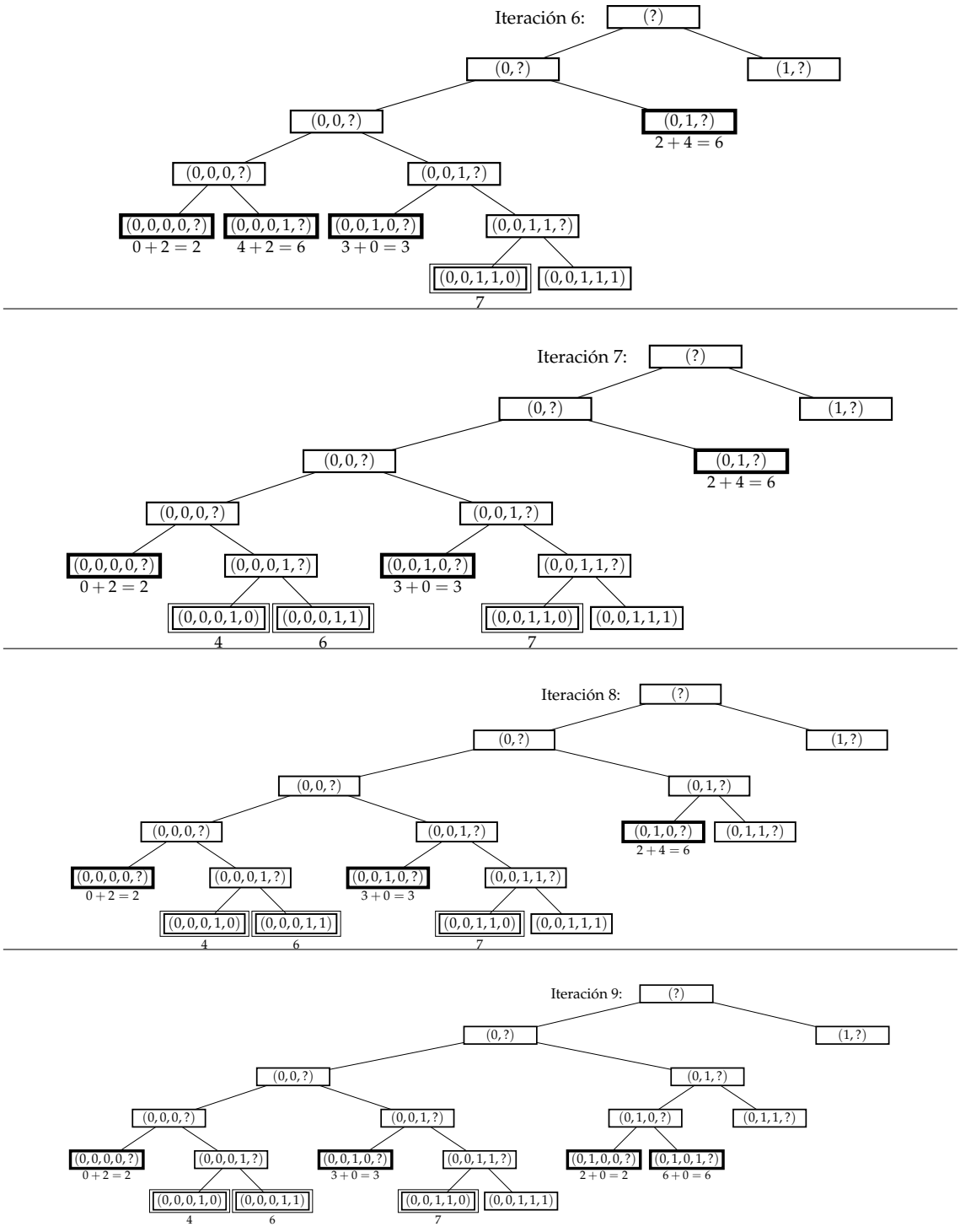


Figura 9.3: Continuación de la figura 9.2. Sigue en la figura 9.4.

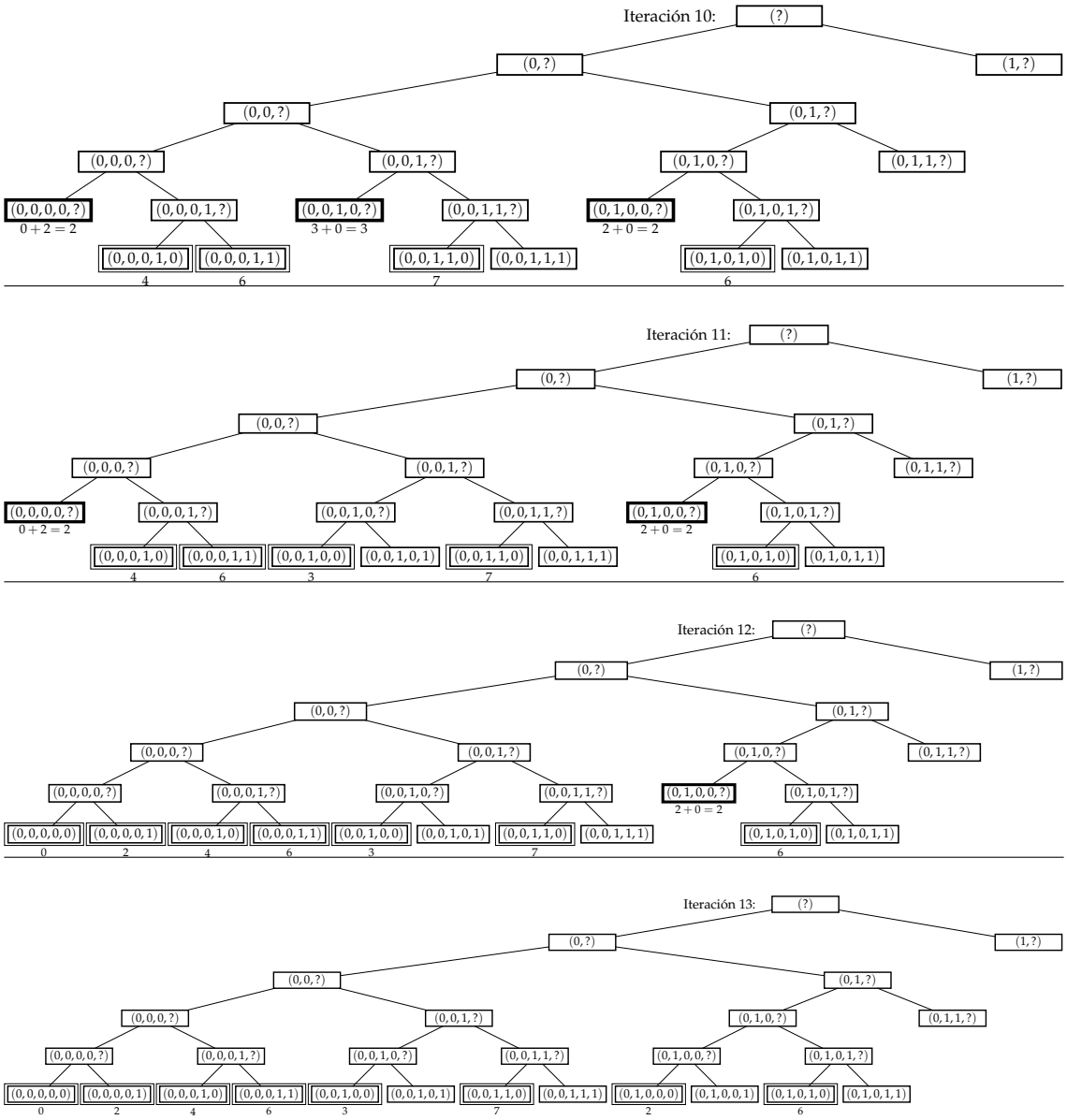


Figura 9.4: Continuación de la figura 9.3.

Seleccionamos ahora el estado $(0, 1, ?)$, que se ramifica en dos estados de los que sólo uno es prometedor:

$$A^{(8)} = \{\overbrace{(0,0,0,0,?)}^{0+2=2}, \overbrace{(0,0,1,0,?)}^{3+0=3}, \overbrace{(0,1,0,?)}^{2+4=6}\}, \qquad \hat{x} = \overbrace{(0,0,1,1,0)}^{f(\hat{x})=7}.$$

Seleccionamos el estado $(0,1,0,?)$ y generamos dos estados:

$$A^{(9)} = \{\overbrace{(0,0,0,0,?)}^{0+2=2}, \overbrace{(0,0,1,0,?)}^{3+0=3}, \overbrace{(0,1,0,0,?)}^{2+0=2}, \overbrace{(0,1,0,1,?)}^{6+0=6}\}, \qquad \hat{x} = \overbrace{(0,0,1,1,0)}^{f(\hat{x})=7}.$$

Seleccionamos el estado $(0,1,0,1,?)$, que produce dos soluciones completas, sólo una de las cuales es factible: $(0,1,0,1,0)$. El valor de esta solución es 6, que no mejora el de la mejor solución factible vista hasta el momento:

$$A^{(10)} = \{\overbrace{(0,0,0,0,?)}^{0+2=2}, \overbrace{(0,0,1,0,?)}^{3+0=3}, \overbrace{(0,1,0,0,?)}^{2+0=2}\}, \qquad \hat{x} = \overbrace{(0,0,1,1,0)}^{f(\hat{x})=7}.$$

Seleccionamos el más prometedor, que genera dos soluciones completas, sólo una de ella factible: $(0,0,1,0,0)$, con valor 3. Dicho valor no es mayor que el de la función objetivo sobre la mejor solución vista hasta el momento:

$$A^{(11)} = \{\overbrace{(0,0,0,0,?)}^{0+2=2}, \overbrace{(0,1,0,0,?)}^{2+0=2}\}, \qquad \hat{x} = \overbrace{(0,0,1,1,0)}^{f(\hat{x})=7}.$$

Seleccionamos uno cualquiera de los dos estados y lo ramificamos. El primero de los dos, por ejemplo, produce dos soluciones factibles: $(0,0,0,0,0)$ y $(0,0,0,0,1)$. El valor de la función objetivo sobre cada uno es 0 y 2, respectivamente. Ninguno mejora, pues, el valor de la mejor solución factible ya vista. Seguimos:

$$A^{(12)} = \{\overbrace{(0,1,0,0,?)}^{2+0=2}\}, \qquad \hat{x} = \overbrace{(0,0,1,1,0)}^{f(\hat{x})=7}.$$

Ramificamos ahora el único estado activo y generamos una única solución factible, $(0,1,0,0,0)$, cuyo valor no es mayor que el de la mejor solución factible conocida.

Ahora el conjunto A está vacío, así que el algoritmo termina su ejecución concluyendo que la solución óptima es $\hat{x} = (0,0,1,1,0)$, con $f(\hat{x}) = 7$.

Podemos hacer un comentario acerca de la aproximación seguida: el algoritmo no ha recorrido el árbol «a ciegas», es decir, siguiendo una estrategia que no tiene en cuenta la información que proporcionan los estados visitados hasta el momento. De hecho, se ha orientado rápidamente hacia la solución factible óptima, que ha encontrado muy pronto. Pero el no poder asegurar que ésa sea la solución óptima ha obligado a proseguir con la búsqueda y explorar un gran número de estados, los mismos que hubiésemos visitado en un recorrido por primero en profundidad y que constituyen la práctica totalidad del árbol. No basta, pues, con orientar la búsqueda: hemos de evitar la exploración de tan gran porción del árbol de estados. Cambiemos ahora la función puntuación en favor de una que supondrá un avance en esta línea.

9.1.4. Una función optimista para la puntuación de los estados

Entre las muchas funciones de puntuación que podemos definir hay una familia de ellas que siguen un criterio de aplicación general: la puntuación de acuerdo con la solución de

una «**relajación del problema**». Entendemos por relajar un problema la eliminación de una o más de las restricciones que debe satisfacer una solución factible. Si, por ejemplo, asumimos que los objetos seleccionados pueden exceder la capacidad de carga de la mochila, estamos relajando el problema de la mochila. La solución del problema relajado es trivial: el máximo beneficio se obtiene cargando todos los objetos en la mochila.

Puntuemos la parte desconocida de los estados con la solución del problema relajado y veamos qué ocurre (las figuras 9.5 y 9.6 ilustran gráficamente la traza). Partimos del conjunto de estados activos con un sólo estado: el inicial, que es $(?)$. Si resolvemos el problema relajado y cargamos todos los objetos el beneficio es 21. De nuevo anotaremos sobre cada estado de A el valor de su puntuación desglosado en «parte conocida» y «parte desconocida»:

$$A^{(0)} = \{ \overbrace{(?)}^{0+21=21} \}.$$

Sólo podemos seleccionar un estado. Al ramificarlo sólo generamos un estado factible: $(0,?)$.

$$A^{(1)} = \{ \overbrace{(0,?)}^{0+11=11} \}.$$

Seleccionamos el estado $(0,?)$ y, al ramificarlo, generamos dos nuevos estados: $(0,0,?)$ y $(0,1,?)$. El estado $(0,0,?)$ puede completarse, en el problema relajado, para formar la solución $(0,0,1,1,1)$, que ofrece un valor 9. El estado $(0,1,?)$ se puede completar, sin tener en cuenta la capacidad de la mochila, para formar la solución $(0,1,1,1,1)$, con la que se obtiene un beneficio de valor 11:

$$A^{(2)} = \{\overbrace{(0,0,?)}^{0+9=9}, \overbrace{(0,1,?)}^{2+9=11}\}.$$

Seleccionamos el estado más prometedor $(0,1,?)$ y pasamos a este nuevo conjunto de estados activos:

$$A^{(3)} = \{\overbrace{(0,0,?)}^{0+9=9}, \overbrace{(0,1,0,?)}^{2+6=8}\}.$$

El estado $(0,1,1,?)$, fruto de ramificar $(0,1,?)$ no ingresa en A por no contener solución factible alguna (el peso de los dos objetos que cargan en la parte conocida suma 11). Seleccionamos y ramificamos ahora el estado $(0,0,?)$:

$$A^{(4)} = \{\overbrace{(0,0,0,?)}^{0+6=6}, \overbrace{(0,0,1,?)}^{3+6=9}, \overbrace{(0,1,0,?)}^{2+6=8}\}.$$

Seleccionamos y ramificamos ahora el estado más prometedor, $(0,0,1,?)$:

$$A^{(5)} = \{\overbrace{(0,0,0,?)}^{0+6=6}, \overbrace{(0,0,1,0,?)}^{3+2=5}, \overbrace{(0,0,1,1,?)}^{7+2=9}, \overbrace{(0,1,0,?)}^{2+6=8}\}.$$

Seleccionamos y ramificamos ahora el estado $(0,0,1,1,?)$. Obtenemos dos soluciones completas: los estados $(0,0,1,1,0)$ y $(0,0,1,1,1)$. La solución $(0,0,1,1,1)$ se descarta por

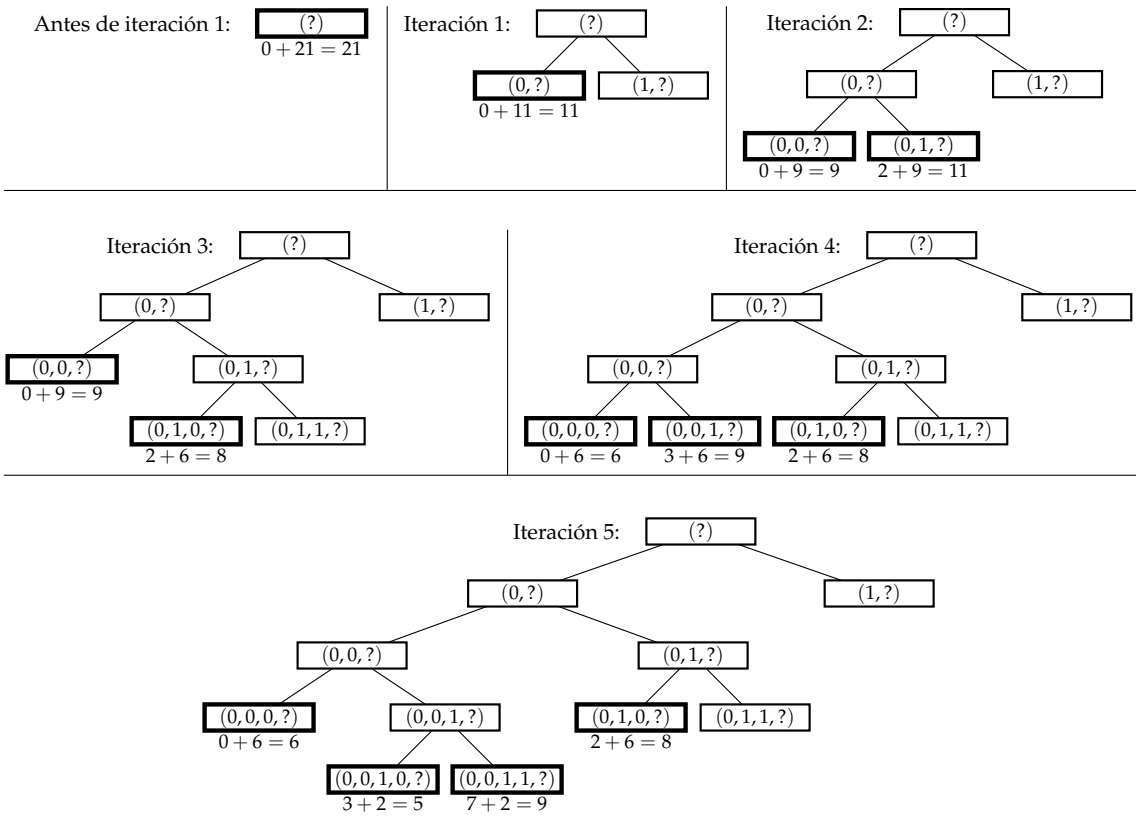


Figura 9.5: Primeros pasos en la exploración del árbol de estados asistidos por una función de puntuación basada en una relajación del problema: se puede exceder la capacidad de carga de la mochila. Sigue en la figura 9.6.

no ser factible; la otra es factible y el valor de la función objetivo sobre ella es 7:

$$A^{(6)} = \{\overbrace{(0,0,0,?)}^{0+6=6}, \overbrace{(0,0,1,0,?)}^{3+2=5}, \overbrace{(0,1,0,?)}^{2+6=8}\}, \qquad \hat{x} = \overbrace{(0,0,1,1,0)}^{f(\hat{x})=7}.$$

Conviene que nos formulemos nuevamente la pregunta, ¿puede alguno de los estados activos conducirnos a una solución factible de mayor valor? No todos. Veamos por qué.

La función de puntuación que hemos definido es «optimista»: proporciona una puntuación que siempre es mayor o igual que el máximo valor que podemos obtener con una solución factible contenida en el estado, pues considera que podemos cargar en la mochila todos los objetos no considerados todavía. La función de puntuación es una **cota superior** del valor de la función objetivo f en el conjunto de soluciones que representa el estado.

El estado $(0,0,0,?)$, por ejemplo, tiene una puntuación 6, obtenida suponiendo que los objetos cuarto y quinto se cargan en la mochila. Si ya hemos visto una solución factible de valor 7, ¿tiene sentido considerar activo el estado $(0,0,0,?)$? La respuesta es no, pues no puede conducirnos a una solución factible mejor que la que ya conocemos. Lo mismo podemos decir del estado $(0,0,1,0,?)$, cuya estimación optimista del valor de cualquiera

de sus soluciones factibles es 5. Ambos estados puede eliminarse de A sin peligro de afectar a la corrección del cálculo que estamos efectuando: aunque contienen soluciones factibles, no son prometedores en el sentido de que su exploración pudiera mejorar el mejor valor conocido de la función objetivo.

El estado $(0, 1, 0, ?)$ tiene puntuación 8, un valor mayor que 7, por lo que *es posible* que contenga una solución factible de valor superior a 7. Es necesario seguir explorando el conjunto de estados activos, pero eliminando antes los estados que no son prometedores:

$$A^{(6)} = \{\overbrace{(0, 1, 0, ?)}^{2+6=8}\}.$$

Seleccionamos el único estado de A y lo ramificamos, así que en principio éste es el valor de $A^{(7)}$:

$$\{\overbrace{(0, 1, 0, 0, ?)}^{2+2=4}, \overbrace{(0, 1, 0, 1, ?)}^{6+2=8}\}.$$

Pero lo cierto es que el estado $(0, 1, 0, 0, ?)$ no tiene por qué ingresar en A: el valor de su cota optimista, 4, es menor que 7, el valor de la función objetivo sobre la mejor solución vista hasta el momento. Así queda A:

$$A^{(7)} = \{\overbrace{(0, 1, 0, 1, ?)}^{6+2=8}\}.$$

Seleccionamos y ramificamos el estado $(0, 1, 0, 1, ?)$ y obtenemos dos soluciones completas: $(0, 1, 0, 1, 0)$ y $(0, 1, 0, 1, 1)$. La primera solución es factible, pero la función objetivo evaluada sobre él proporciona beneficio 6, que no mejora el valor de la solución factible que ya conocíamos: $(0, 0, 1, 1, 0)$. La segunda solución completa no es factible, así que se descarta. El conjunto A ha quedado vacío. El algoritmo puede finalizar con la conclusión de que la solución de máximo beneficio es $(0, 0, 1, 1, 0)$, con valor 7.

Nótese que hemos encontrado la solución factible óptima explorando una porción mucho menor del árbol de estados que cuando efectuamos la búsqueda con la otra función de puntuación de estados. (Compárese el árbol de la figura 9.6 con el de la figura 9.3.)

No presentaremos ahora un algoritmo para resolver el problema de la mochila, pues aún no hemos expuesto todos los elementos que pueden conformar un algoritmo de ramificación y poda: iremos presentando diferentes versiones conforme desarrollemos diferentes esquemas.

9.2. Esquema de ramificación y poda

En general, notaremos por X el conjunto de **soluciones factibles**. Los elementos de X suelen ser elementos que satisfacen ciertas restricciones y que pertenecen a un conjunto más amplio, X'. Los elementos de X' son las **soluciones** y los de $X' - X$ son las **soluciones no factibles**. Buscamos un elemento de X que haga óptimo (mínimo o máximo) el valor de cierta **función objetivo** $f : X \to \mathbb{R}$:

$$\hat{x} = \arg \operatorname*{opt}_{x \in X} f(x).$$

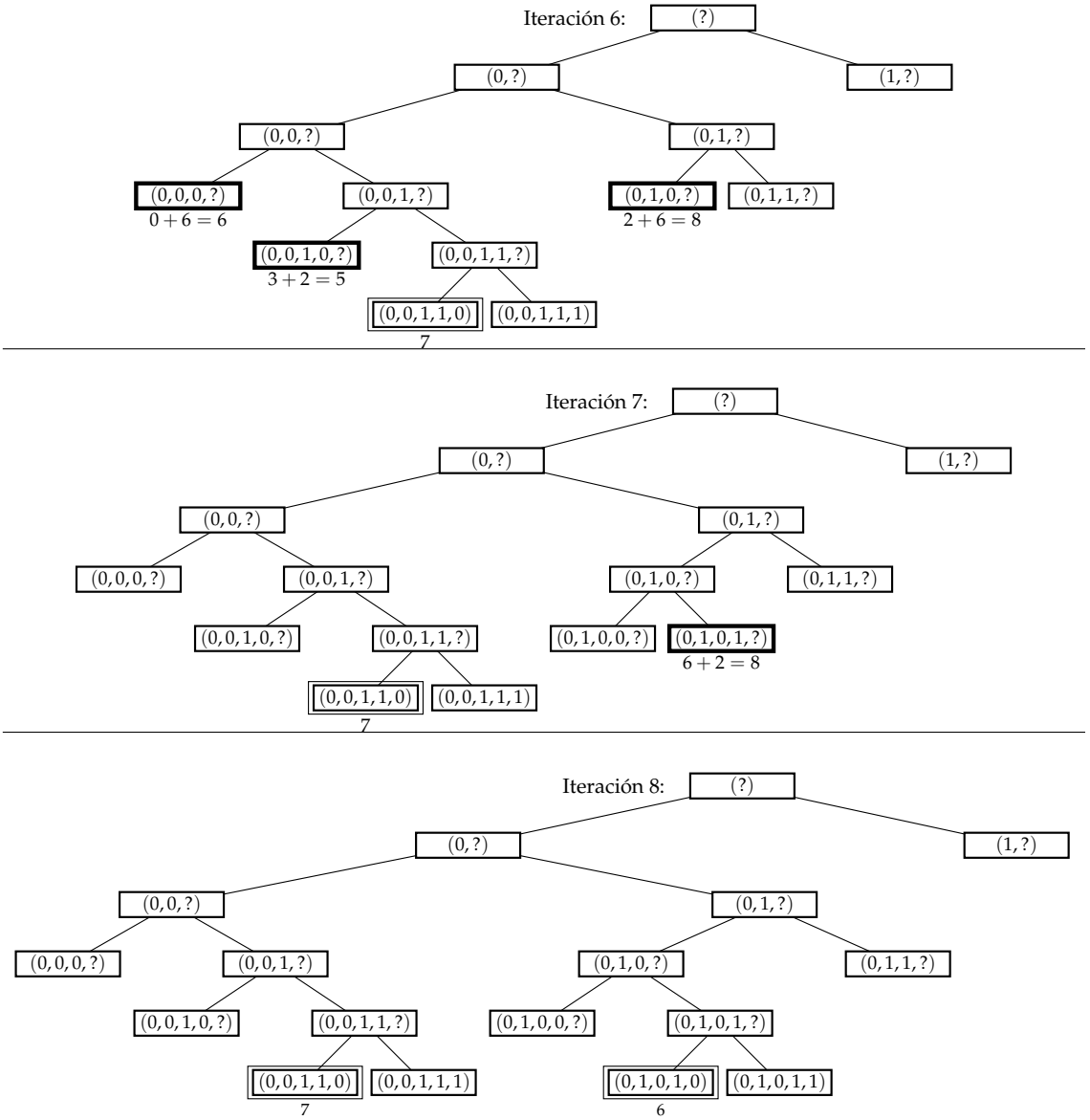


Figura 9.6: Continuación de la figura 9.5. Compárese la parte explorada del árbol de estados con la que exploramos usando otra función de puntuación y que se muestra en la figura 9.4.

Supondremos en lo sucesivo que $X \neq \emptyset$ y, por tanto, existe siempre una solución óptima.

9.2.1. Conceptos básicos y notación

Al diseñar un algoritmo de ramificación y poda empezamos por definir el concepto de **estado**, que es una representación de un conjunto de soluciones (factibles o no). Un estado representa un elemento de $\mathcal{P}(X')$. Usaremos los símbolos s, s', s'', etc. para denotar

estados y los símbolos x, x', etc. para denotar soluciones factibles. La talla de un estado es el número de soluciones que contiene, por lo que $|s| = 1$ indica que el estado representa un conjunto con una sola solución. Las soluciones factibles de un estado s son $s \cap X$.

Es frecuente que los estados representen conjuntos de soluciones que comparten un prefijo. Usamos una notación como ésta para indicar un estado con un prefijo de talla k común a todas sus soluciones:

$$(x_1, x_2, \ldots, x_k, ?).$$

Reservamos las letras mayúsculas para los conjuntos de estados, que son conjuntos de subconjuntos de X', es decir, elementos de $\mathcal{P}(\mathcal{P}(X'))$. En particular, usaremos la letra A para denotar el **conjunto de estados activos**. El conjunto A presenta un contenido diferente tras cada iteración del algoritmo. Denotaremos con $A^{(0)}$ su valor inicial y con $A^{(i)}$ su valor tras la i-ésima iteración.

Los estados han de permitir diseñar cómodamente un proceso de **ramificación**. Dicho proceso se aplica sobre un estado (subconjunto de X') y devuelve un conjunto de estados (conjunto de subconjuntos de X'). (En el ejemplo de la mochila discreta siempre se obtenían dos estados al ramificar uno, pero en general puede haber un número arbitrario.) Representamos el proceso de ramificación por medio de la función

$$\mathit{branch} : \mathcal{P}(X') \to \mathcal{P}(\mathcal{P}(X')).$$

Hemos de seguir un **criterio de selección** del estado activo que se ramifica en cada iteración. Hay tres criterios de selección: primero en profundidad, primero en anchura y primero el mejor (aunque veremos que las dos primeras pueden reducirse a casos particulares del tercero). Cualquiera que sea el método escogido lo representaremos por medio de una función *select* que escoge un estado de A y, por tanto, tiene por perfil

$$\mathit{select} : \mathcal{P}(\mathcal{P}(X')) \to \mathcal{P}(X').$$

Finalmente, el proceso de **poda** puede ser descrito por una función de perfil

$$\mathit{prune} : \mathcal{P}(\mathcal{P}(X')) \to \mathcal{P}(\mathcal{P}(X')).$$

La función objetivo sólo puede aplicarse sobre soluciones factibles, pero un estado terminal no es un elemento, sino un conjunto con un solo elemento. No obstante, abusaremos de la notación y cuando $s = \{x\}$, asumiremos que $f(s)$ se define como $f(x)$.

9.2.2. Un esquema básico

Con estos componentes básicos podemos presentar los algoritmos de ramificación y poda como los que se obtienen por instanciación del esquema de la figura **??**.

bbscheme.py

```
class BranchAndBoundBasicScheme:
    def is_complete(self, s):
        """Determina si un estado es completo."""

    def select(self, A):
        """Selecciona el elemento más prometedor de A."""

    def branch(self, s):
        """Ramifica el estado s."""

    def prune(self, A, x):
        """Poda el contenido de s con la información que proporciona el estado x."""

    def solve(self):
        A = [self.X]
        while not (len(A) == 1 and self.is_complete(A[0])):
            s = self.select(A)
            A.remove(s)
            A = self.prune(A + list(self.branch(s)))
        return A.pop()
```

Es un esquema deliberadamente sencillo, pero cuya corrección sólo se garantiza bajo la observación de determinadas propiedades (que presentamos en el siguiente apartado) de cada una de las tres funciones fundamentales, *select*, *branch* y *prune*. Inicialmente, A contiene un único estado activo que representa toda posible solución (factible o no). Con cada iteración se selecciona un estado activo y se ramifica. Se extrae el estado seleccionado del conjunto de estados activos y se insertan en A los estados resultantes de la ramificación. El conjunto A sufre entonces una poda que elimina aquellos estados que, a la luz de la información disponible hasta el momento, se consideran no prometedores.

Una de la condiciones que exigiremos es un tanto peculiar, así que vale la pena detenernos a comentarla: tras la poda, en A puede haber a lo sumo un estado de talla unitaria. Dicho estado contiene la mejor solución factible vista hasta el momento. Cuando A sólo tiene un elemento y dicho elemento es unitario, no hay nada más que ramificar, así que el algoritmo puede detenerse y devolver la solución contenida en dicho estado como solución factible óptima.

Este esquema simplificado nos resulta de ayuda para demostrar sobre él la terminación y corrección del método bajo la única condición de que X' sea de cardinalidad finita. Refinamientos posteriores incorporarán mejoras que toman en cuenta cuestiones relativas a la eficiencia computacional. (Por ahora resulta difícil comparar este esquema con la solución dada al problema de la mochila en la sección anterior, pues éste se deriva de un refinamiento del esquema.)

Terminación y corrección del esquema básico

Si las funciones del esquema observan ciertas condiciones y existe al menos una solución factible, el procedimiento encuentra la solución factible óptima. Abordaremos la

demostración de terminación y corrección del esquema algorítmico tras presentar dichas condiciones. Y antes de presentarlas, definimos las funciones $f' : \mathcal{P}(X') \to \mathbb{R}$ y $f'' : \mathcal{P}(\mathcal{P}(X')) \to \mathbb{R}$ como sigue

$$f'(s) = \operatorname*{opt}_{x \in s \cap X} f(x), \qquad f''(B) = \operatorname*{opt}_{s \in B} f'(s) = \operatorname*{opt}_{s \in B} \operatorname*{opt}_{x \in s \cap X} f(x).$$

La función f' proporciona el valor óptimo de la función objetivo sobre las soluciones factibles de un estado. La función f'' hace lo propio sobre las soluciones factibles contenidas en cualquiera de los miembros de un conjunto de estados.

Condiciones exigibles a la función de ramificación. Exigimos dos condiciones a la función de ramificación: (1) que devuelva subconjuntos propios y no nulos de aquél que ramifica; y (2) que toda solución factible de un estado esté incluida en alguno de los estados resultantes de la ramificación, que no deben contener soluciones factibles que no se encontraban en el estado original:

Condición B1 Para todo $i \geq 0$ y para todo estado $s \in A^{(i)}$ tal que $|s| > 1$, y para todo $s' \in \mathit{branch}(s)$, se observa $s' \subset s$ y $s' \neq \emptyset$.

Condición B2 Para todo $i \geq 0$ y para todo estado $s \in A^{(i)}$ tal que $|s| > 1$, se cumple $\bigcup_{s' \in \mathit{branch}(s)} s' = s$.

Condiciones exigibles a la función de selección. La función de selección debe devolver un elemento del conjunto de estados activos cuya talla sea mayor que uno.

Condición S1 Para todo $A^{(i)}$, con $i \geq 0$, se observa $\mathit{select}(A^{(i)}) \in A^{(i)}$.

Condición S2 Para todo $A^{(i)}$, con $i \geq 0$, se cumple $|\mathit{select}(A^{(i)})| > 1$.

Condiciones exigibles a la función de poda. La función de poda elimina estados del conjunto de estados activos, pero debe preservar el estado que contenga la solución óptima. En el esquema que estamos presentando, puede haber a lo sumo un estado unitario que contenga una solución factible: la mejor vista hasta el momento. (En un refinamiento posterior eliminaremos esta restricción y almacenaremos esta solución en una variable.)

Condición P1 Para todo $i \geq 0$ y $A^{(i)}$, se cumple $\mathit{prune}(A^{(i)}) \subseteq A^{(i)}$.

Condición P2 Para todo $i \geq 0$ y $A^{(i)}$, se cumple $|\{s \in \mathit{prune}(A^{(i)}) : |s| = 1\}| \leq 1$.

Condición P3 Para todo $i \geq 0$ y $A^{(i)}$, se cumple que $f''(A^{(i)}) = f''(\mathit{prune}(A^{(i)}))$.

Teorema 9.1 (Terminación) *Si la talla de X' es finita, entonces el esquema algorítmico de ramificación y poda termina.*

Demostración. En el peor de los casos, el algoritmo no poda ningún estado de talla mayor que uno. En tal caso, los estados son eliminados de A por una de dos razones:

- son seleccionados (Condición S1), ramificados y eliminados de A,

- o conviven dos o más de talla unitaria en A de los que sólo uno puede sobrevivir a la poda (por la Condición P2).

Por S2, si s es seleccionado, entonces $|s| > 1$. Por la Condición B1, los diferentes conjuntos B obtenidos a resultas de ramificar s son de talla menor que $|s|$. El conjunto de estados activos A puede crecer en cardinalidad conforme evoluciona el algoritmo, pero se observa

$$\max_{s \in A^{i+1}} |s| \leq \max_{s \in A^i} |s|$$

siendo $A^{(i)}$ el valor de A en la i-ésima iteración y A^{i+1}, su valor en la siguiente.

Cada estado seleccionado irá, pues, dividiéndose en nuevos estados de talla menor con cada iteración. Llegará un instante en que se alcancen estados de talla unitaria. Por la Condición P2, si hay más de un estado de talla unitaria, sólo uno sobrevive a la poda. Finalmente, llegará a haber un único estado en A de talla unitaria, por lo que el algoritmo alcanzará la condición de terminación. □

Trataremos de demostrar ahora la corrección del procedimiento:

Lema 9.1 (Persistencia del óptimo en A) *Sean s^i, B^i y $A^{(i)}$ los valores de las variables s, B y A del esquema al finalizar la i-ésima iteración, y sea $A^{(0)} = \{X'\}$. Sea t el número de iteraciones que requiere el esquema algorítmico para terminar (t es infinito si no termina). Al menos una de las soluciones factibles óptimas de X incluidas en algún estado de $A^{(i)}$ está incluida en algún estado de A^{i+1}; es decir,*

$$f''(A^{i+1}) = f''(A^i).$$

Demostración.

$$\begin{aligned}
f''(A^{i+1}) &= f''(prune((A^{(i)} - \{s^{i+1}\}) \cup B^{i+1})) \\
&= f''((A^{(i)} - \{s^{i+1}\}) \cup B^{i+1}) && \text{(por P3)} \\
&= f''((A^{(i)} - \{s^{i+1}\}) \cup branch(s^{i+1})) \\
&= f''((A^{(i)} - \{s^{i+1}\}) \cup \{s^{i+1}\}) && \text{(por B2)} \\
&= f''(A^{(i)}) && \text{(por S1)}
\end{aligned}$$

□

Ya podemos demostrar que el esquema algorítmico encuentra la solución:

Teorema 9.2 *Para todo i entre 0 y t se observa $f''(A^i) = opt_{x \in X} f(x)$.*

Demostración. Por inducción sobre i, el número de iteración, y considerando que $A^{(0)} = \{X'\}$ y que $X \subseteq X'$, aplicamos el lema de persistencia del óptimo. □

Corolario 9.1 *Si el esquema algorítmico termina, entonces devuelve un elemento $\hat{x}$ tal que*

$$\hat{x} = \arg \operatorname*{opt}_{x \in X} f(x).$$

Demostración. Si el esquema algorítmico termina en un número de iteraciones t, el conjunto A^t contiene un único estado que es de talla unitaria:

$$A^t = \{\{\hat{x}\}\}.$$

Por el teorema anterior tenemos

$$f''(\{\{\hat{x}\}\}) = \operatorname*{opt}_{x \in X} f(x).$$

Y por la definición de f'',

$$f''(\{\{\hat{x}\}\}) = f(\hat{x}) = \operatorname*{opt}_{x \in X} f(x),$$

de donde se sigue que $\hat{x}$ es un elemento óptimo de X. Al ser $\hat{x}$ devuelto por el esquema algorítmico, éste encuentra un elemento óptimo de X. □

Aplicación del esquema a la mochila discreta

Empezamos por puntuar los estados con una función que considera el valor de los objetos que ya se ha decidido insertar en la mochila y suma el valor de todos los objetos sobre los que se ha adoptado ya una decisión. La poda se limita conservar todo estado con dos o más soluciones y un solo estado con una sola solución, el que presenta mejor valor de la función objetivo:

bb_knapsack.py

```
1  from bbscheme import BranchAndBoundBasicScheme
2  from utils import argmax, max
3  from offsetarray import OffsetArray
4
5  class KnapsackBasicMixin:
6      def __init__(self, W, v, w):
7          self.W, self.v, self.w, self.N = W, v, w, len(v)
8          self.X = OffsetArray([])
9
10     def select(self, A):
11         return argmax((s for s in A if not self.is_complete(s)),
12                       lambda s: sum(s[i]*self.v[i] for i in xrange(1,len(s)+1)))
13
14     def branch(self, s):
15         if not self.is_complete(s):
16             yield s + [0]
17             if sum(s[i]*self.w[i] for i in xrange(1,len(s)+1)) + self.w[len(s)+1] <= self.W:
18                 yield s + [1]
19
20     def prune(self, A):
21         survivors = [s for s in A if not self.is_complete(s)]
22         s' = argmax((s for s in A if self.is_complete(s)),
```

```
23                      lambda s: sum(s[i]*self.v[i] for i in xrange(1,len(s)+1)))
24          if s' != None: survivors.append(s')
25          return survivors
26
27      def is_complete(self, s):
28          return len(s) == self.N
29
30  class Knapsack1(KnapsackBasicMixin, BranchAndBoundBasicScheme):
31      pass
```

```
1  from bb_knapsack import Knapsack1
2  from offsetarray import OffsetArray
3
4  W, v, w = 10, OffsetArray([10, 2, 3, 4, 2]), OffsetArray([12, 5, 6, 2, 6])
5  print Knapsack1(W, v, w).solve()
```

```
[0, 0, 1, 1, 0]
```

¿Es un algoritmo correcto? Hemos de comprobar que se satisfacen las condiciones B1, B2, S1, S2, P1, P2 y P3. La primera prescribe que la función de ramificación produzca subconjuntos propios y no vacíos sobre el los estados incompletos, y así ocurre con nuestra función de ramificación. La Condición B2 indica que la reunión de los hijos de toda ramificación dé como resultado al padre, como también resulta fácil comprobar en nuestro algoritmos. La función de selección escoge el estado incompleto de A con mayor valor de la función de puntuación; así pues, devuelve un estado de A (Condición S1) y el estado no es completo (Condición S2). La función de poda devuelve un subconjunto de A (Condición P1) y en dicho subconjunto hay, a lo sumo, un estado completo (Condición P2). La Condición P3 exige que al menos un estado que contenga una solución óptima sobreviva a la poda. Como solo podamos estados unitarios y el estado unitario con la solución de mayor beneficio no se poda nunca, nuestro algoritmo observa la Condición P3.

9.2.3. Esquema con almacenamiento explícito de la mejor solución factible vista hasta el momento

El esquema anterior puede refinarse evitando insertar estados de talla unitaria en A, pues de entre todos ellos sólo nos interesa uno: el óptimo. Para ello mantiene la mejor solución hallada hasta el momento en una variable x y no considera la inclusión en A de estados que representan una sola solución: en el mismo momento en que son generados se comprueba si contiene una solución mejor que cualquier otra vista hasta el momento:

`bbscheme.py` *(cont.)*

```
23  from utils import infinity
24
25  class BranchAndBoundKeepingBestSolutionScheme:
26      def is_complete(self, s):
27          """Determina si un estado es completo."""
```

```
28
29      def is_factible(self, s):
30          """Determina si un estado es factible."""
31
32      def worst_value(self):
33          """Pésimo"""
34
35      def select(self, A):
36          """Selecciona el elemento más prometedor de A."""
37
38      def branch(self, s):
39          """Ramifica el estado s."""
40
41      def prune(self, A, x):
42          """Poda el contenido de s con la información que proporciona el estado x."""
43
44      def f(self, x):
45          """Función objetivo"""
46
47      def opt(self, a, b):
48          """Función de optimización (mín o máx)."""
49
50      def solve(self):
51          if self.is_complete(self.X): return self.X
52          A = [self.X]
53          x, fx = None, self.worst_value()
54          while len(A) != 0:
55              s = self.select(A)
56              A.remove(s)
57              for s' in self.branch(s):
58                  if self.is_complete(s'):
59                      if self.is_factible(s') and self.opt(fx, self.f(s')) != fx:
60                          x, fx = s', self.f(s')
61                  else:
62                      A.append(s')
63              A = self.prune(A)
64          return x
```

Nótese que hemos modificado la función de poda: ahora recibe, además de un conjunto de estados, la mejor solución vista hasta el momento. *Con el nuevo esquema, las Condiciones P2 y S2 se satisfacen siempre*, pues ya no ingresan en A conjuntos de talla unitaria.

La Condición P3 debe reformularse ahora, pues cuando se enunció suponíamos que x, la mejor solución factible vista hasta el momento, formaba parte de A:

Condición P3′ Una vez se conoce alguna solución factible y x es la mejor solución factible vista hasta el momento, se cumple que $f''(A^{(i)} \cup \{\{x\}\}) = f''(prune(A^{(i)}) \cup \{\{x\}\})$, para todo $i \geq 0$.

Aplicación del esquema a la mochila discreta

Es necesario explicitar ahora la función objetivo y la optimización deseada (máx), así como un valor pésimo, esto es, un elemento neutro para la maximización:

bb_knapsack.py *(cont.)*

```
34 from bbscheme import BranchAndBoundKeepingBestSolutionScheme
35
36 class KnapsackKeepingBestSolutionMixin(KnapsackBasicMixin):
37     def worst_value(self):
38         return -1
39
40     def is_factible(self, s):
41         return sum(s[i]*self.w[i] for i in xrange(1,len(s)+1)) <= self.W
42
43     def f(self, x):
44         return sum(x[i]*self.v[i] for i in xrange(1,len(x)+1))
45
46     def opt(self, a, b):
47         return max(a, b)
48
49 class Knapsack2(KnapsackKeepingBestSolutionMixin, BranchAndBoundKeepingBestSolutionScheme):
50     pass
```

```
1 from bb_knapsack import Knapsack2
2 from offsetarray import OffsetArray
3
4 W, v, w = 10, OffsetArray([10, 2, 3, 4, 2]), OffsetArray([12, 5, 6, 2, 6])
5 print Knapsack2(W, v, w).solve()
```

```
[0, 0, 1, 1, 0]
```

La Condición P3′ se observa en el algoritmo presentado: la poda es inefectiva en tanto que sólo elimina estados completos, y estos ya no ingresan en A.

9.2.4. Esquema con cola de prioridad para el conjunto de estados activos

El criterio de selección afecta a la estructura de datos con que hemos de representar A. Consideremos tres criterios: primero en profundidad, primero en anchura y primero el mejor. Los tres criterios de selección satisfacen la Condición S1. Los dos primeros criterios conducen a **estrategias de búsqueda ciega** o **no informadas**, mientras que la última permite diseñar una **estrategia de búsqueda heurística** o **informada**, pues se usa información propia del problema para orientar la búsqueda. Cada estrategia presenta ciertas ventajas:

- Primero en profundidad: conduce rápidamente a estados que representan conjuntos de talla unitaria y, por tanto, encuentra rápidamente soluciones factibles, pero que no tienen porqué estar «próximas» a la solución buscada.
- Primero en anchura: conduce muy lentamente a estados de talla unitaria pero, al ser toda solución factible de talla finita, presenta la ventaja de encontrar con seguridad la solución óptima si el factor de ramaje es finito, aunque X sea de talla infinita.
- Primero el mejor: sus prestaciones dependen de la calidad de la estimación del valor de la función objetivo sobre la mejor solución factible de un estado. Si ésta es buena, se dirige rápidamente hacia una solución factible cuyo valor de la función objetivo está próximo al óptimo.

Cualquiera de los criterios satisface la Condición S1. El último criterio es el más interesante de todos, pues suele conducir a los procedimientos más eficientes. La determinación de qué estado de A es el mejor en cada instante nos obliga a estimar por medio de cierta **función de puntuación** de estados qué valor de la función objetivo es óptimo en cada estado $s \in A$. Notaremos con $score : \mathcal{P}(X') \to \mathbb{R}$ a dicha función. En cualquier caso, los dos primeros criterios de selección (primero en profundidad y primero en anchura) pueden verse como casos particulares de la selección por primero el mejor. Si puntuamos los estados con un número que indique su profundidad en el árbol y el instante en que se ha generado el estado, la selección del estado de menor puntuación conduce a una búsqueda en anchura. Con una ligera variante de esta función de puntuación, la búsqueda por primero el de mayor puntuación es equivalente a una búsqueda por primero en profundidad. Nos ahorraremos, pues, hablar en el futuro de los diferentes criterios de selección suponiendo siempre una búsqueda por primero el mejor. Este esquema hace uso de una cola de prioridad para efectuar una búsqueda por primero el mejor:

`bbscheme.py` *(cont.)*

```
66 from utils import *
67
68 class BranchAndBoundBestFirstScheme:
69     def is_complete(self, s):
70         """Determina si un estado es completo."""
71
72     def is_factible(self, s):
73         """Determina si un estado es factible."""
74
75     def worst_value(self):
76         """Pésimo"""
77
78     def branch(self, s):
79         """Ramifica el estado s."""
80
81     def prune(self, A, x):
82         """Poda el contenido de s con la información que proporciona el estado x."""
83
84     def f(self, x):
```

```
85          """Función objetivo"""
86
87      def score(self, s):
88          """Puntuación que indica cuán prometedor es un estado."""
89
90      def opt(self, a, b):
91          """Función de optimización (mín o máx)."""
92
93      def argopt(self, A):
94          """Argumento que proporciona el óptimo de f sobre los elementos de A."""
95
96      def solve(self):
97          if self.is_complete(self.X): return self.X
98          A = [self.X]
99          x, fx = None, self.worst_value()
100         while len(A) != 0:
101             s = self.argopt(A)
102             A.remove(s)
103             for s' in self.branch(s):
104                 if self.is_complete(s'):
105                     if self.is_factible(s') and self.opt(fx, self.f(s')) != fx:
106                         x, fx = s', self.f(s')
107                 else:
108                     A.append(s')
109             A = self.prune(A)
110         return x
```

Esta estrategia de búsqueda requiere el empleo de una estructura de datos que, en principio, asegure eficiencia en tres operaciones: inserción de estados puntuados, extracción del estado con mejor puntuación y eliminación de los estados no prometedores. En cada iteración del algoritmo se extrae el estado más prometedor, se realizan b inserciones en A y se podan p estados. La tabla 9.1 muestra la complejidad con que estas operaciones se pueden efectuar empleando diferentes estructuras de datos (ordenación con respecto a la función de puntuación).

	extracción	b inserciones	p borrados
Lista	$O(\|A\|)$	$O(b)$	$O(\|A\|)$
Lista ordenada	$O(1)$	$O(b \lg b + \|A\|)$	$O(\|A\|)$
Heap	$O(\lg \|A\|)$	$O(b \lg \|A\|)$	$O(\|A\|)$

Tabla 9.1: Coste de una extracción, b inserciones y p borrados en función de la estructura de datos con que se implementa A, el conjunto de estados activos.

A la vista de la tabla, resulta evidente que el heap ofrece un coste aceptable tanto para inserciones como para la extracción del óptimo. El borrado de p elementos es una operación costosa en cualquiera de las estructuras. Hasta que no encontremos una solución satisfactoria para la ejecución eficiente de esta operación, no podemos inclinarnos definitivamente por ninguna estructura de datos (al menos no en términos asintóticos).

Aplicación del esquema a la mochila discreta

Hemos de definir una función de puntuación que permita decidir si un estado nos parece más prometedor que otro. Tomaremos como función de puntuación una que, dado un estado, suma al valor de los objetos que ya se ha decidido cargar en la mochila el de una solución voraz para el conjunto de objetos sobre los que no se ha tomado una decisión:

bb_knapsack.py *(cont.)*

```
52 from bbscheme import BranchAndBoundBestFirstScheme
53
54 class KnapsackBestFirstMixin(KnapsackKeepingBestSolutionMixin):
55     def score(self, s):
56         value = sum(s[i]*self.v[i] for i in xrange(1,len(s)+1))
57         W = self.W - sum(s[i]*self.w[i] for i in xrange(1,len(s)+1))
58         for i in xrange(len(s)+1, self.N+1):
59             if self.w[i] <= W:
60                 value += self.v[i]
61                 W -= self.w[i]
62         return value
63
64     def argopt(self, A):
65         return argmax(A, self.score)
66
67 class Knapsack3(KnapsackBestFirstMixin, BranchAndBoundBestFirstScheme):
68     pass
```

```
1 from bb_knapsack import Knapsack3
2 from offsetarray import OffsetArray
3
4 W, v, w = 10, OffsetArray([10, 2, 3, 4, 2]), OffsetArray([12, 5, 6, 2, 6])
5 print Knapsack3(W, v, w).solve()
```

```
[0, 0, 1, 1, 0]
```

9.2.5. Esquema con poda por cota optimista

Hay una familia de funciones de poda que conduce a un nuevo refinamiento del esquema: las funciones de **poda por cota optimista**. Podríamos decir que en ellas radica una idea fundamental en ramificación y poda: la eliminación de aquellos estados de los que sabemos que, aunque pueden contener soluciones factibles, éstas no pueden mejorar el valor de la función objetivo sobre la mejor solución vista hasta el momento. Para ello necesitamos una función que proporcione una estimación optimista del valor de la función objetivo sobre las soluciones factibles de un estado.

La cota optimista es una función $optimistic : \mathcal{P}(X') \to \mathbb{R}$ que satisface:

$$\operatorname{opt}\left(optimistic(s),\ \operatorname*{opt}_{x \in s \cap X} f(x)\right) = optimistic(s).$$

Quizá se entienda mejor esta condición si la especializamos para resolver, por una parte, problemas de minimización y, por otra, de maximización. En problemas de minimización, *optimistic* debe satisfacer

$$optimistic(s) \leq \min_{x \in (s \cap X)} f(x).$$

Y en problemas de maximización,

$$optimistic(s) \geq \max_{x \in (s \cap X)} f(x).$$

Si la cota optimista sobre un estado es «peor» que el valor de la función objetivo sobre una solución factible cualquiera, el estado se considera no prometedor. Así pues, un estado activo s puede ser descartado si, dada una solución factible cualquiera, x, tenemos

$$\text{opt}(optimistic(s), f(x)) = f(x).$$

Como la notación puede resultar un tanto confusa, vale la pena indicar que esta expresión, cuando minimizamos, se lee «$optimistic(s) \leq f(x)$», y cuando maximizamos, «$optimistic(s) \geq f(x)$».

En el siguiente esquema se efectúa una poda por cota optimista:

bbscheme.py *(cont.)*

```
112 class BranchAndBoundWithOptimisticBoundScheme:
113     def is_complete(self, s):
114         """Determina si un estado es completo."""
115 
116     def is_factible(self, s):
117         """Determina si un estado es factible."""
118 
119     def worst_value(self):
120         """Pésimo"""
121 
122     def branch(self, s):
123         """Ramifica el estado s."""
124 
125     def prune(self, A, x):
126         """Poda el contenido de s con la información que proporciona el estado x."""
127 
128     def f(self, x):
129         """Función objetivo"""
130 
131     def score(self, s):
132         """Puntuación que indica cuán prometedor es un estado."""
133 
134     def opt(self, a, b):
135         """Función de optimización (mín o máx)."""
136 
137     def optimistic(self, s):
```

```
138         """Cota optimista del valor óptimo de la función objetivo sobre elementos de s."""
139
140     def priority_queue(self, iterable=[]):
141         """Estructura de datos para la cola de prioridad. Se inicializa con el iterable."""
142
143     def solve(self):
144         if self.is_complete(self.X): return self.X
145         A = self.priority_queue([(self.score(self.X), self.X)])
146         x, fx = None, self.worst_value()
147         while len(A) != 0:
148             (score_s, s) = A.extract_opt()
149             for s' in self.branch(s):
150                 if self.is_complete(s'):
151                     if self.is_factible(s') and self.opt(fx, self.f(s')) != fx:
152                         x, fx = s', self.f(s')
153                         A = self.priority_queue([(score_s'', s'') for (score_s'', s'') in A \
154                                                  if self.opt(self.optimistic(s''), fx) != fx])
155                 else:
156                     if self.opt(self.optimistic(s'), fx) != fx:
157                         A.insert((self.score(s'), s'))
158         return x
```

La poda aparece explícitamente en la línea 94 e implícitamente en las líneas 97–98. La poda explícita se ejecuta cuando se encuentra una solución factible x mejor que cualquier otra vista hasta el momento y considera no prometedores los estados de A con una estimación optimista igual o peor que el valor de $f(x)$. La poda implícita de las líneas 97–98 prescinde de insertar en A nuevos estados de los que se sabe que no son prometedores.

Recordemos que, para garantizar la corrección de un algoritmo basado en el esquema original, la función de poda debe satisfacer las condiciones P1, P2 y P3′. La condición P2 estaba garantizada en el refinamiento anterior, del que éste es un refinamiento adicional. Nos queda por estudiar si se observan o no las condiciones P1 y P3′:

- Las podas por cota optimista satisfacen la Condición P1. La Condición P1 dice que la poda de A resulta en un subconjunto de A, extremo evidente si consideramos la línea 15 del esquema de la figura **??**, que elimina algunos elementos de A. La poda implícita de las líneas 17–18 considera la no inclusión de algunos elementos en A, así que A acaba siendo un subconjunto del conjunto de estados activos que hubiéramos formado de no mediar la poda por cota optimista.

- La Condición P3′ también se satisface. Si $f''(A^{(i)} \cup \{\{x\}\}) = f(x)$, entonces x es una solución factible óptima y la eliminación de estados de $A^{(i)}$ de la línea 15 o la no inclusión de estados de las líneas 17–18 no pueden hacer que $f''(prune(A^{(i)}) \cup \{\{x\}\}) \neq f(x)$.

  En caso contrario, es decir, si $f''(A^{(i)} \cup \{\{x\}\}) \neq f(x)$, x no es la solución factible óptima y algún estado s de $A^{(i)}$ contiene una solución factible óptima $\hat{x}$. Su puntuación, $optimistic(s)$, es mejor que $f(\hat{x})$, que a su vez es mejor que $f(x)$. Así pues, el es-

tado s no puede podarse en la línea 15 o será insertado en la línea 18 y $f''(prune(A^{(i)}) \cup \{\{x\}\})$ coincide con $f''(A^{(i)} \cup \{\{x\}\})$.

La función de cota optimista constituye, pues, un criterio de poda válido. En cada problema concreto es tarea del diseñador del algoritmo encontrar tal función cota y, por lo general, se requiere un conocimiento bastante profundo del problema a resolver. No obstante, más adelante presentaremos algunas líneas de diseño de cotas optimistas que pueden ser de ayuda en la práctica.

Aplicación del esquema a la mochila discreta

Podemos diseñar una cota optimista si al valor de los objetos que ya hemos decidido insertar en la mochila le sumamos el valor de todos aquellos sobre los que no hemos tomado una decisión. Es evidente que se trata de una cota optimista, pues no es posible obtener más beneficio que el que se obtiene si metemos en la mochila todos los objetos:

bb_knapsack.py *(cont.)*

```
70 from bbscheme import BranchAndBoundWithOptimisticBoundScheme
71 from heap import MaxHeap
72 from offsetarray import OffsetArray
73
74 class KnapsackWithOptimisticBoundMixin(KnapsackBestFirstMixin):
75     def priority_queue(self, iterable=[]):
76         return MaxHeap(iterable)
77
78     def optimistic(self, s):
79         return sum(s[i]*self.v[i] for i in xrange(1,len(s)+1)) + \
80                sum(self.v[i] for i in xrange(len(s)+1,self.N+1))
81
82 class Knapsack4(KnapsackWithOptimisticBoundMixin, BranchAndBoundWithOptimisticBoundScheme):
83     pass
```

```
1 from bb_knapsack import Knapsack4
2 from offsetarray import OffsetArray
3
4 W, v, w = 10, OffsetArray([10, 2, 3, 4, 2]), OffsetArray([12, 5, 6, 2, 6])
5 print Knapsack4(W, v, w).solve()
```

```
[0, 0, 1, 1, 0]
```

Tiene interés que estudiemos empíricamente cómo afecta a la eficiencia de la búsqueda la poda por cota optimista. Estudiaremos la talla máxima A, el número de estados que se insertan A, el de estados que se extraen de A (lo que indica el número de iteraciones efectuadas) y el número de estados que se generan por ramificación del seleccionado. La tabla 9.2 recoge el resultado de un experimento en el que hemos generado instancias aleatorias del problema de la mochila discreta para valores de N de 3 a 10. Cada valor es

N	máx $\|A\|$		inserciones		generados		extracciones	
3	2.4	2.4	4.8	4.1	10.1	7.5	5.8	4.4
4	3.3	3.1	10.8	6.4	20.8	10.7	11.8	6.3
5	4.7	3.9	20.7	8.6	39.8	13.4	21.7	7.8
6	8.1	4.8	41.8	13.6	75.0	20.3	42.8	12.4
7	10.9	5.4	84.3	21.1	155.5	31.8	85.3	19.9
8	16.8	6.1	135.3	27.8	250.4	40.1	136.3	25.7
9	39.6	8.9	399.5	50.9	748.5	76.4	400.5	46.8
10	68.0	10.3	722.8	88.9	1334.5	137.8	723.8	85.4

Tabla 9.2: Comparación experimental de (el promedio de) la talla máxima de A, inserciones en A, número de estados generados y extracciones de A para diferentes valores de N y 20 instancias aleatorias. En cada columna el número de la izquierda corresponde al algoritmo sin poda por cota optimista y el de la derecha, al algoritmo con cota optimista.

el promedio del resultado obtenido sobre 20 instancias aleatorias y cada objeto tenía un peso aleatorio entre 1 y $2N$ y un valor aleatorio entre 1 y $10N$.

Se puede observar en la tabla que la introducción de la poda por cota optimista supone una importante reducción del espacio necesario y del esfuerzo computacional. Y la reducción de esfuerzo es tanto más importante cuanto mayor es el valor de N.

9.2.6. Esquema con función de puntuación y criterio de poda optimistas

Si se utiliza un criterio de selección por primero el mejor necesitamos, en un principio, definir dos funciones: una función de puntuación *score* para efectuar la selección por primero el mejor y otra *optimistic* podar los estados no prometedores.

Ambas deben de ser buenas aproximaciones del valor óptimo de f aplicado sobre las soluciones factibles de un estado por lo que, en lugar de tratar de obtener dos funciones diferentes, puede resultar conveniente concentrar esfuerzos y tratar de definir *score* y *optimistic* como una *única función*. Hacerlo tiene una consecuencia positiva inmediata: permite soslayar el problema de la eliminación explícita de estados no prometedores, que es una operación muy costosa. Esta poda explícita obliga a recorrer todos los estados de A, eliminar los que ya no son prometedores y reconstruir la cola de prioridad. Es un proceso costoso que puede realizarse gran número de veces e incidir muy negativamente en el coste del algoritmo.

Lo cierto es que podemos prescindir de este paso y no efectuar poda alguna si cambiamos la condición de terminación del algoritmo. Si finalizamos cuando A es el conjunto vacío o cuando su mejor elemento debería ser podado (presenta un valor de la cota optimista peor o igual que el valor de f sobre la mejor solución conocida), estaremos efectuando una **poda implícita**:

`bbscheme.py` *(cont.)*

```
160 class BranchAndBoundScheme:
161     def is_complete(self, s):
162         """Determina si un estado es completo."""
163 
164     def is_factible(self, s):
165         """Determina si un estado es factible."""
166 
```

```
167     def worst_value(self):
168         """Pésimo"""
169 
170     def branch(self, s):
171         """Ramifica el estado s."""
172 
173     def f(self, x):
174         """Función objetivo"""
175 
176     def score(self, s):
177         """Puntuación que indica cuán prometedor es un estado."""
178 
179     def opt(self, a, b):
180         """Función de optimización (mín o máx)."""
181 
182     def optimistic(self, s):
183         """Cota optimista del valor óptimo de la función objetivo sobre elementos de s."""
184 
185     def priority_queue(self, iterable=[]):
186         """Estructura de datos para la cola de prioridad. Se inicializa con el iterable."""
187 
188     def solve(self):
189         if self.is_complete(self.X): return self.X
190         A = self.priority_queue([(self.optimistic(self.X), self.X)])
191         x, fx = None, self.worst_value()
192         while len(A) != 0 and self.opt(A.opt()[0], fx) != fx:
193             (score_s, s) = A.extract_opt()
194             for s' in self.branch(s):
195                 if self.is_complete(s'):
196                     if self.is_factible(s') and self.opt(fx, self.f(s')) != fx:
197                         x, fx = s', self.f(s')
198                 else:
199                     optimistic_s' = self.optimistic(s')
200                     if self.opt(optimistic_s', fx) != fx:
201                         A.insert((optimistic_s', s'))
202         return x
```

La poda implícita supone una significativa aceleración en el tiempo de ejecución, aunque acarree un mayor consumo de memoria. Una consecuencia beneficiosa es que permite decantarnos definitivamente y sin reparos por el max-heap como estructura de datos adecuada para la cola de prioiridad A.

Aplicación a la mochila discreta

Poco hay que decir o implementar, pues ya hemos definido una cota optimista y basta con usarla ahora como función de puntuación para la selección por primero el mejor:

bb_knapsack.py *(cont.)*

```
85 from bbscheme import BranchAndBoundScheme
86
87 class Knapsack5(KnapsackWithOptimisticBoundMixin, BranchAndBoundScheme):
88     pass
```

```
1 from bb_knapsack import Knapsack5
2 from offsetarray import OffsetArray
3
4 W, v, w = 10, OffsetArray([10, 2, 3, 4, 2]), OffsetArray([12, 5, 6, 2, 6])
5 print Knapsack5(W, v, w).solve()
```

```
[0, 0, 1, 1, 0]
```

En la tabla 9.3 se recoge el resultado de un estudio de la eficiencia comparada del algoritmo que usa una función de puntuación diferente de la usada en la cota con el úñtimo que hemos presentado (ambas coinciden y, además, hay poda implícita). Se puede observar que la ocupación espacial crece significativamente, pero es lo que cabía esperar. A cambio de este aumento de la ocupación espacial tenemos una mejora de la complejidad temporal en tanto que no es necesario recorrer la cola de prioridad para eliminar elementos y reconstruirla cada cierto tiempo. Se ha reducido, además, el número de iteraciones del algoritmo (que coincide con el número de extracciones de elementos de A) y el número de estados generados. Aumenta, en cambio, el número de estados que acaban ingresando en A.

N	máx $\|A\|$		inserciones		generados		extracciones	
3	2.4	2.4	4.1	4.1	7.5	7.3	4.4	4.3
4	3.1	3.3	6.4	6.8	10.7	10.7	6.3	6.3
5	3.9	4.0	8.6	9.0	13.4	12.8	7.8	7.5
6	4.8	6.2	13.6	15.0	20.3	19.9	12.4	12.2
7	5.4	7.4	21.1	21.8	31.8	30.0	19.9	18.9
8	6.1	11.3	27.8	31.7	40.1	38.4	25.7	24.8
9	8.9	16.6	50.9	50.0	76.4	63.9	46.8	39.9
10	10.3	34.4	88.9	106.8	137.8	126.8	85.4	79.5
11	10.9	36.3	112.4	123.0	164.3	146.3	109.1	98.5
12	11.1	71.5	180.3	222.9	277.8	260.4	178.7	167.3
13	17.7	83.3	255.2	302.6	389.4	369.0	250.8	237.3
14	20.3	191.2	622.8	697.4	936.4	875.5	618.6	580.3
15	22.3	170.3	510.8	601.9	766.8	731.4	504.9	482.6
16	38.4	358.5	1097.1	1243.7	1661.1	1524.3	1087.9	995.8
17	28.3	320.4	1087.9	1242.5	1581.2	1497.5	1082.0	1026.3
18	33.1	454.8	1438.5	1685.6	2118.4	2002.7	1425.0	1350.2
19	62.9	870.6	2315.5	2890.8	3721.6	3558.3	2300.9	2200.9
20	94.0	1547.6	6412.3	6832.4	9627.6	9209.1	6393.8	6138.3

Tabla 9.3: En cada columna se muestra a la izquierda el resultado para el algoritmo con función de puntuación diferente de la cota optimista y a la derecha, el resultado para el algoritmo en el que ambas coinciden y hay poda implícita.

9.2.7. Diseño de cotas optimistas

Por sus ventajas, nos hemos inclinado por usar una cota optimista como función de puntuación de estados para la búsqueda pro primero el mejor y como puntuación que permite efectuar una poda eficiente de estados no prometedores. El diseño de cotas optimistas resulta, pues, crucial en la construcción de algoritmos de ramificación y poda, por lo que vale la pena que nos detengamos a efectuar algunas consideraciones al respecto.

En primer lugar conviene que nos planteemos qué es una buena cota optimista o, al menos, que tengamos algún criterio que permita comparar entre sí dos cotas optimistas diferentes. Pues bien, es fácil decir cuál es la mejor cota optimista posible: una que proporcione el valor óptimo de f sobre las soluciones factibles del estado s, es decir,

$$\operatorname*{opt}_{x \in (s \cap X)} f(x).$$

Una cota optimista como esta permite que el algoritmo siempre seleccione, de entre todos los estados activos, uno que contiene una solución óptima. Si el espacio de búsqueda es finito y el árbol de búsqueda está balanceado, daremos con una solución óptima en apenas $O(\lg |X'|)$ iteraciones. Así pues, la búsqueda se orienta muy rápidamente hacia la solución óptima, lo que es indudablemente ventajoso.

Y aún hay más: tan pronto se alcanza la solución óptima, el algoritmo finaliza porque todos los estados activos son podados en ese instante. Esto es así porque cualquier estado activo presenta como cota optimista un valor peor o igual que el de la solución óptima. (Lo cierto es que no se elimina explícitamente, pero el elemento que ocupa la cima del heap presenta una puntuación (cota optimista) igual o peor que el valor de f sobre la solución factible óptima.) El problema estriba en cómo calcular esa cota optimista eficientemente. La respuesta es desalentadora: por regla general, no podemos. Al calcularla para ayudarnos a resolver el problema, estaríamos resolviendo efectivamente el mismo problema.

En cualquier caso vale la pena que advirtamos que esa cota optimista ideal tiene una propiedad interesante: es la cota optimista *menos optimista* de cuantas hay. Es menos optimista en tanto que el valor que proporciona es el de la solución óptima del estado sobre el que se evalúa. Podríamos decir que es «realista» más que auténticamente optimista. Pero hay aquí un principio interesante: dadas dos cotas optimistas, resulta mejor aquella que sea *menos optimista* de las dos. No siempre podremos comparar dos cotas optimistas en términos de cuál de las dos es menos optimista. Pero cuando sea posible efectuar esta comparación, tenemos ahí un criterio para elegir la que conducirá a un algoritmo más eficiente. Si una cota se aproxima más a la ideal que otra, decimos de la primera que está **más informada**. Y decimos de la ideal que está «perfectamente informada». Cuanto más informada está una cota, mejor. La cota optimista menos informada es aquella que proporciona la misma puntuación sobre todos los estados. Una cota optimista no informada trivial es aquella que proporciona el valor $+\infty$ si estamos resolviendo un problema maximización y $-\infty$ si el problema es de minimización.

Hay un principio de diseño de cotas optimistas que vale la pena destacar ahora. Cuando estamos explorando un estado de la forma $(s_1, s_2, \ldots, s_k, ?)$ estamos formando una solución factible paso a paso. En cada paso decidimos que valor dar a un nuevo elemento

de la tupla. Podemos decir que hay una parte conocida y una parte desconocida en la solución factible que buscamos al ir ramificando un estado. Los elementos de la parte desconocida suelen definir una instancia del problema que estamos resolviendo. Ciertas funciones objetivo son separables y su valor sobre cada solución factible contenida en un estado puede verse como la combinación de una cantidad en cuyo cálculo sólo participan elementos de la parte conocida y otra parte en el que participan elementos de la parte desconocida. Ha ocurrido así con el problema de la mochila discreta. Al calcular la cota optimista de un estado $(s_1, s_2, \ldots, s_k, ?)$ hemos considerado que podíamos sumar al valor de los objetos ya cargados en la mochila la suma de $v_{k+1}, v_{k+2}, \ldots, v_N$. Así pues, hay dos partes en la cota optimista, una que asociamos a la parte conocida de toda solución contenida en el estado y otra que asociamos a la parte desconocida:

$$optimistic((s_1, s_2, \ldots, s_k, ?)) = \overbrace{\sum_{1 \le i \le k} s_i v_i}^{\text{parte conocida}} + \overbrace{\sum_{k < i \le N} v_i}^{\text{parte desconocida}} \quad .$$

Esta división en parte conocida y desconocida puede (y suele) ser de gran ayuda a la hora de diseñar cotas optimistas: la parte conocida resulta de cálculo trivial y la parte desconocida puede formularse en términos de resolución de una **relajación del problema** sobre los elementos de la parte desconocida. Un problema relajado es aquel que obtenemos eliminando restricciones exigibles a toda solución factible. Toda solución del problema con restricciones es una solución del problema relajado, pero lo contrario no es cierto (véase la figura 9.7).

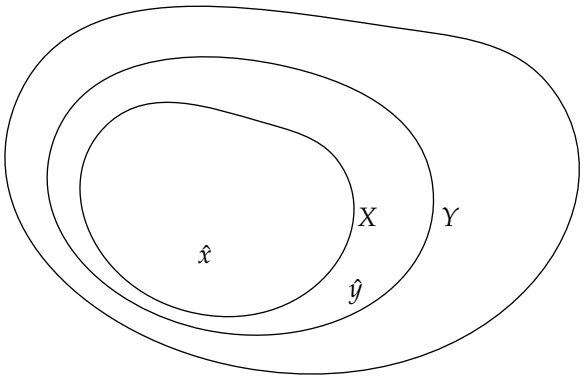


Figura 9.7: Relaciones de inclusión entre el conjunto de soluciones, X', el conjunto de soluciones factibles, X, y el conjunto de soluciones factibles, Y, para una relajación del problema. La solución óptima en Y, que denotamos con $\hat{y}$, presenta mejor valor o igual que la mejor solución $\hat{x}$ en X'.

Así, la solución óptima del problema relajado es una estimación optimista de la solución del problema original con restricciones. En minimización,

$$X \subseteq Y \quad \Rightarrow \quad \min_{x \in Y} f(x) \le \min_{x \in X} f(x),$$

y en maximización,

$$X \subseteq Y \quad \Rightarrow \quad \max_{x \in Y} f(x) \ge \max_{x \in X} f(x).$$

El respeto a la capacidad de carga de la mochila no es la única restricción que podemos suprimir al diseñar una cota optimista para el problema de la mochila. Podemos

plantearnos, por ejemplo, suprimir la restricción de «integridad» de los objetos que se pueden cargar en la mochila, es decir, podemos considerar la inclusión de partes de un objeto. Nos encontramos en tal caso ante el problema de la mochila con fraccionamiento, problema para el que sabemos encontrar una solución óptima eficientemente siguiendo una estrategia voraz:

bb_knapsack.py *(cont.)*

```
90  from bbscheme import BranchAndBoundScheme
91  from fractionalknapsack import fractional_knapsack
92  from offsetarray import OffsetArray
93
94  class KnapsackWithOptimisticBoundMixin2(KnapsackWithOptimisticBoundMixin):
95      def optimistic(self, s):
96          v = OffsetArray([self.v[i] for i in xrange(len(s)+1, self.N+1)])
97          w = OffsetArray([self.w[i] for i in xrange(len(s)+1, self.N+1)])
98          W = self.W - sum(s[i]*self.w[i] for i in xrange(1,len(s)+1))
99          x = fractional_knapsack(w, v, W)
100         return sum(s[i]*self.v[i] for i in xrange(1,len(s)+1)) + \
101                 sum(x[i]*self.v[i+len(s)] for i in xrange(1,len(x)+1))
102
103 class Knapsack6(KnapsackWithOptimisticBoundMixin2, BranchAndBoundScheme):
104     pass
```

```
1  from bb_knapsack import Knapsack6
2  from offsetarray import OffsetArray
3
4  W, v, w = 10, OffsetArray([10, 2, 3, 4, 2]), OffsetArray([12, 5, 6, 2, 6])
5  print Knapsack6(W, v, w).solve()
```

```
[0, 0, 1, 1, 0]
```

Ahora que tenemos dos cotas diferentes, ¿cuál está más informada? Notemos que el valor de la segunda cota (solución del problema de la mochila con fraccionamiento) siempre es menor o igual que el de la primera cota (suma del valor de todos los objetos). Así pues, podemos afirmar que la segunda cota está mejor informada que la primera y es, por tanto, mejor en el sentido de orientar mejor la búsqueda y efectuar una poda más efectiva.

Esta conjetura debe plasmarse, en la práctica, en una mejora apreciable de la eficiencia computacional. La tabla 9.4 permite comparar los resultados obtenidos al ejecutar el algoritmo con la cota optimista basada en introducir todos los elementos en la mochila y la basada en la resolución del problema de la mochila con fraccionamiento. La mejora es sencillamente espectacular.

Pero el grado de información de una cota, esto es, su mayor o menor optimismo no es el único criterio para la comparación de cotas. También hemos de tener en cuenta la eficiencia computacional. Hemos de diseñar una función *optimistic* que sea computacionalmente eficiente, pues se evalúa sobre todo estado incompleto generado en el proceso

N	máx $\|A\|$		inserciones		generados		extracciones	
3	2.4	2.4	4.1	3.6	7.3	6.5	4.3	3.5
4	3.3	3.5	6.8	6.0	10.7	8.8	6.3	4.7
5	4.0	4.0	9.0	7.8	12.8	11.1	7.5	6.0
6	6.2	5.4	15.0	10.8	19.9	13.8	12.2	7.7
7	7.4	5.8	21.8	12.1	30.0	15.5	18.9	8.6
8	11.3	7.0	31.7	15.1	38.4	18.3	24.8	10.2
9	16.6	9.3	50.0	19.3	63.9	23.7	39.9	12.4
10	34.4	11.4	106.8	23.6	126.8	27.9	79.5	14.4
11	36.3	11.4	123.0	27.1	146.3	32.5	98.5	18.3
12	71.5	12.7	222.9	32.3	260.4	37.5	167.3	21.9
13	83.3	18.3	302.6	43.4	369.0	50.5	237.3	27.5
14	191.2	20.6	697.4	56.1	875.5	70.3	580.3	39.2
15	170.3	20.0	601.9	46.5	731.4	53.8	482.6	29.4
16	358.5	19.8	1243.7	47.4	1524.3	57.0	995.8	30.6
17	320.4	21.9	1242.5	56.4	1497.5	66.8	1026.3	37.3
18	454.8	25.3	1685.6	58.3	2002.7	66.6	1350.2	35.9
19	870.6	26.9	2890.8	60.4	3558.3	67.8	2200.9	35.8
20	1547.6	27.4	6832.4	72.2	9209.1	88.8	6138.3	47.7

Tabla 9.4: En cada columna se muestra a la izquierda el resultado para el algoritmo con la cota optimista basada en la inclusión de todos los objetos y a la derecha, el resultado para el algoritmo con cota optimista basada en la resolución del problema de la mochila con fraccionamiento.

de búsqueda. Ya hemos visto que la cota optimista ideal es rechazable porque, en general, su cálculo resulta computacionalmente prohibitivo.

Si comparamos en términos de eficiencia las dos cotas consideradas para la mochila discreta comprobaremos que, en principio, ambas requieren la ejecución de $O(N)$ pasos. Pero la primera de la cotas es más eficiente a poco que efectuemos su cálculo con un poco de cuidado. Es posible calcularla **incrementalmente**, es decir, podemos calcular la cota de $(s_1, s_2, \ldots, s_k, ?)$ cuando obtenemos ese estado por ramificación de su padre, $(s_1, s_2, \ldots, s_{k-1}, ?)$:

$$\mathit{optimistic}(s_1, s_2, \ldots, s_k) = \mathit{optimistic}(s_1, s_2, \ldots, s_{k-1}) + s_k v_k - v_k.$$

Así pues, la primera cota puede calcularse en tiempo $O(1)$. No ocurre lo mismo con la segunda.

Recordemos, pues, que *la eficiencia de un algoritmo de ramificación y poda depende críticamente de lo informada que esté la cota y del coste computacional de su cálculo*. Y como ambos factores suelen ser antagónicos, se ha de adoptar una solución de compromiso. Muchas veces este compromiso sólo puede adoptarse tras un estudio empírico de la eficiencia de los algoritmos resultantes.

Por la naturaleza del cálculo de las funciones de puntuación y cota optimista es frecuente que éstas produzcan estimaciones tanto más precisas cuanto «más cerca» está un estado de una solución factible, es decir, cuanto menor es el número de soluciones que contiene. Es natural, pues la información disponible aumenta en el camino de la raíz a las hojas. Este mayor grado de precisión se traduce, en el caso de la función de cota optimista, en que los estados obtenidos de la ramificación presentan un valor de cota optimista «más realista» que el de su padre. Es decir, en problemas de minimización, su valor es no decreciente si vamos de raíz a hojas, y en problemas de maximización, no creciente:

- Monotonía no decreciente de *optimistic* (frecuente en problemas de minimización):

$$s' \in branch(s) \Rightarrow optimistic(s') \geq optimistic(s).$$

- Monotonía no creciente de *optimistic* (frecuente en problemas de maximización):

$$s' \in branch(s) \Rightarrow optimistic(s') \leq optimistic(s).$$

Usaremos la expresión **monótonamente menos optimista** para acoger ambos casos bajo un mismo término.

Si estamos puntuando estados para su selección por primero el mejor, interesa justo lo contrario: que la puntuación de los estados resultantes de la ramificación les haga más prometedores que al propio padre. De ese modo se consigue acelerar la generación de soluciones factibles y, por tanto, acelerar la reducción del espacio de búsqueda que supone la poda por cota optimista.

Como hemos dicho, es frecuente que se use la misma función de acotación para puntuar y para acotar de modo optimista el valor de la función objetivo en un estado. En tal caso, nos encontramos ante dos intereses antagónicos para la función *optimistic*: lo que beneficia a la función de puntuación perjudica a la poda y viceversa. No obstante, hay un efecto beneficioso en el hecho de que la función de puntuación sea «monótonamente menos optimista»: la puntuación del mejor estado es cada vez menos optimista (o igual) y, por tanto, más próxima al valor óptimo de la función objetivo. En espacios de búsqueda de cardinalidad infinita se puede aprovechar esta propiedad para demostrar la terminación de un método de ramificación y poda.

9.2.8. Algunas consideraciones adicionales sobre la eficiencia de la estrategia de ramificación y poda

El procedimiento de búsqueda exhaustiva es, en gran cantidad de problemas, extremadamente costoso, ya que la talla del espacio de búsqueda es exponencial con parámetros que miden la talla del problema. Los algoritmos de ramificación y poda sirven como estrategia de reducción de complejidad en algunos de dichos problemas: cuando se poda una rama del árbol de estados, se está eliminando un número de soluciones que es (potencialmente) exponencial. Pero eliminar un número exponencial de elementos de un conjunto de talla exponencial *no tiene por qué reducir la talla del conjunto a una cantidad polinómica de elementos*. Así pues, el propio algoritmo de ramificación y poda puede resultar un mecanismo de resolución no aplicable en la práctica. Hay que decir, no obstante, que la talla de numerosos problemas de interés práctico puede ser suficientemente reducida como para que la ramificación y poda resuelva en un tiempo razonable el problema sin que ocurra lo mismo con la búsqueda exhaustiva.

Es posible determinar la complejidad computacional de cada una de las iteraciones del bucle principal del esquema. En general, sin embargo, no es posible determinar la complejidad del proceso completo, que depende enormemente de la instancia del problema a resolver y de la «calidad» de la función de puntuación y cota optimista.

En una iteración dada, el coste temporal vendrá dado por el proceso de selección, el proceso de ramificación y el cálculo de la cota sobre cada uno de los nuevos estados. También se puede realizar un estudio del coste de la poda explícita, si es que ésta tiene lugar.

Al igual que ocurre con la complejidad temporal, no siempre es posible estimar o acotar de forma ajustada el número de estados que se generarán, por lo que no se puede determinar la complejidad espacial del proceso completo. En ese caso, deberemos contentarnos con hacer un estudio del almacenamiento requerido para un estado.

Ya hemos hecho varias apreciaciones acerca de factores que afectan a la complejidad computacional. Hemos hablado, por ejemplo, del beneficio que supone la incrementalidad del cálculo de las funciones de puntuación y/o cota optimista. Consideremos algunos aspectos adicionales.

Ramificación sin solapamientos

Nada se dice acerca de la posibilidad de que una misma solución aparezca en dos o más estados obtenidos a resultas de ramificar. Es preferible, no obstante, diseñar una función de ramificación que efectúe una *partición* del conjunto original; es decir, tal que los estados obtenidos representen conjuntos disjuntos dos a dos.

Condición B3 **(opcional)** Para todo $s = select(A^{(i)})$, con $i \geq 0$, y para todo par de estados s' y s'' pertenecientes a $branch(s)$, se cumple que $s' \cap s'' = \emptyset$.

En caso de no ser así, tendremos que resolver subproblemas idénticos en diferentes instantes, lo cual resulta en un procedimiento ineficiente (recuérdese que la técnica de programación dinámica reducía la complejidad de procedimientos recursivos —similares a la exploración de un árbol que propone ramificación y poda— cuando lograba que cada subproblema se resolviera una sola vez, dejando la solución accesible en un almacén).

Ramificación equilibrada

Interesa, además, que la partición está bien balanceada, es decir, que la cardinalidad de cada uno de los estados resultantes de la ramificación sea similar.

En el algoritmo diseñado para el problema de la mochila discreta, por ejemplo, la ramificación particiona el conjunto de soluciones de un estado en dos conjuntos, cada uno con la mitad de las soluciones contenidas en el estado padre.

Coincidencia de la cota optimista con la función objetivo aplicada sobre la solución de los estados unitarios

Otra propiedad deseable en *optimistic* es que coincida con el valor de f cuando se aplica sobre estados representando conjuntos de talla unitaria. Es decir, conviene que

$$optimistic(s') = f(s'),$$

para todo $s' \in branch(s)$ tal que $|s'| = 1$ y $s \in A^{(i)}$ para todo $i \geq 0$. De esta forma se ahorra el cómputo expreso de la función objetivo sobre las soluciones.

Las cotas optimistas que hemos diseñado para el problema de la mochila, por ejemplo, satisfacen esta propiedad.

9.2.9. Representación compacta de los estados: árbol de prefijos

Hemos representado los estados con tuplas (implementadas con listas Python). Esta representación hace que cada estado consuma memoria proporcional al número de elementos de la tupla y que el proceso de ramificación requiera tiempo proporcional a la talla de la tupla por cada uno de los hijos. Pero hay una relación entre estados padre e hijo que podemos explotar en aras de una mayor eficiencia espacial y temporal: un estado hijo de talla k comparte sus $k-1$ primeros componentes con su padre. Bastaría con almacenar el k-ésimo componente y un puntero al padre para, en principio, representar la misma información. Esta representación alternativa, aplicada sobre los estados generados hasta un momento dado recibe el nombre de **árbol de prefijos**. La figura 9.8 muestra un árbol de prefijos para un ejemplo ficticio de conjunto de estados.

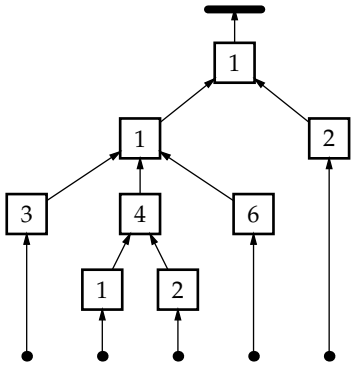


Figura 9.8: Árbol de prefijos con el que representar el conjunto de estados activos $\{(1,1,3,?),(1,1,4,1,?),(1,1,4,2,?),(1,1,6,?),(1,2,?)\}$.

Cada estado consume únicamente el espacio de memoria de un puntero (el puntero al padre) y la celda que indica el valor de su último elemento. El conjunto A es un almacén de punteros a celdas de este tipo. La implementación en un lenguaje como Python puede resultar extremadamente sencilla, pues cada estado puede representarse con una tupla de dos elementos: (una referencia a) la tupla que representa al estado padre y el nuevo elemento. El estado $(2,4,1,1)$ se representa con la tupla de tuplas `(((1,), 1), 4), 2)`. Esta función de ramificación, por ejemplo, añade los valores 0, 1, 2 y 3 al estado que se le suministra para generar cuatro nuevos estados:

```
1  def branch(s):
2      for i in xrange(4):
3          yield (s, i)
```

En lenguajes como C o C++, con reserva y liberación explícita de memoria, la gestión del árbol de prefijos puede resultar un tanto farragosa, tanto por la necesidad de definir registros adecuados como por la liberación de memoria cuando un estado se descarta, pues se deben liberar algunos registros del estado, pero no todos. Una implementación con contadores de referencias puede resultar adecuada.

Esta representación tan sencilla de los estados presenta algunos inconvenientes: operaciones necesarias, como el cálculo de la talla de la tupla o el término de la cota optimista que corresponde a la parte conocida de la tupla, requieren recorridos de la tupla hasta la raíz que resultan costosas. Conviene por ello incluir información adicional a los estados que permita acelerar estos cálculos. El cambio es sencillo: basta con que cada nodo del árbol de prefijos sea un registro u objeto de cierta clase con los datos necesarios.

Para facilitar el desarrollo de programas con este tipo de implementación, ofrecemos ahora una clase que implementa estados como árboles de prefijos. Se pueden definir clases con estados enriquecidos con información adicional heredando de ésta:

bbscheme.py *(cont.)*

```
204 class State:
205     def __init__(self, parent=None, sk=None):
206         self.parent = parent
207         self.sk = sk
208         self.len = parent.len + 1  if parent != None else (0 if sk==None else 1)
209 
210     def __len__(self): return self.len
211 
212     def __gt__(self, right): return self.score > right.score
213     def __lt__(self, right): return self.score < right.score
214 
215     def list(self):
216         def _list(self, l):
217             if self.parent != None: l = _list(self.parent, l)
218             if self.sk != None: l.append(self.sk)
219             return l
220         return _list(self, [])
221 
222     def __str__(self): return str(self.list())
223     __repr__ = __str__
224 
225     def set(self, **kwargs):
226         self.__dict__.update(kwargs)
227         return self
```

Cada celda mantiene información sobre su padre y la longitud de la tupla que representa. Si el padre no es *None*, mantiene además el último elemento de la tupla que

representa. Calcular la longitud de la tupla (método *__len__* es una operación ejecutable en tiempo $O(1)$ porque la talla del estado se almacena explícitamente. Los métodos *__gt__* y *__lt__* permiten efectuar comparaciones entre estados basándose en el valor del atributo *score*, cuyo valor debe estar definido. Estos operadores hacen posible el almacenamiento de los estados en colas de prioridad basándose en el valor de *score*. El método *list* devuelve una lista con los valores de la tupla y resulta útil, en combinación con *__str__* (y *__repr__*), para mostrar las tuplas con el aspecto visual de una lista. El método *set*, finalmente, permite crear atributos con un valor dado muy cómodamente: si *x* es un estado, *x*.*set*(*score*=*valor*), por ejemplo, crea un atributo *score* (si no existía) y le asigna el valor *valor*. Como el método *set* devuelve la instancia sobre la que se invoca, resulta cómodo para efectuar asignaciones donde se espera una expresión.

Hay un problema de diferente naturaleza con la representación de los estados en problemas con áboles de estados con un elevado factor de ramaje. Al ramificar un estado se genera una gran cantidad de nuevos estados que deben representarse en memoria e insertarse en el conjunto de estados activos. De todos ellos, uno resulta el más prometedor, así que cuando se seleccione uno de esos estados en una iteración posterior, se seleccionará primero ése. Puede resultar conveniente, pues, diseñar la representación de modo que la ramificación siempre genere dos elementos: el estado más prometedor y un «paquete» que represente al resto. La puntuación con que el paquete ingresa en A es la de su estado más prometedor. La implementación de los estados se complica considerablemente, pero la eficiencia computacional puede mejorar significativamente al evitar numerosas inserciones en A, cuyo número pasa de ser b por iteración a sólo 2. No nos ocuparemos de esta técnica, pero el lector puede tener que recurrir a ella si el número de estados generados estalla.

Aplicación a la mochila discreta

Conviene enriquecer los estados con información relativa al peso y valor de los objetos que ya hemos cargado en la mochila:

bb_knapsack.py *(cont.)*

```
106 from bbscheme import BranchAndBoundScheme, State
107
108 class KnapsackState(State):
109     def __init__(self, parent=None, sk=None, v=None, w=None):
110         State.__init__(self, parent, sk)
111         self.value = parent.value + sk * v[self.len] if parent != None else 0
112         self.weight = parent.weight + sk * w[self.len] if parent != None else 0
113
114 class Knapsack(KnapsackWithOptimisticBoundMixin2, BranchAndBoundScheme):
115     def __init__(self, W, v, w):
116         self.W, self.v, self.w, self.N = W, v, w, len(v)
117         self.total_value = OffsetArray([0]*(self.N+1))
118         for i in xrange(self.N, 0, -1):
119             self.total_value[i] = self.total_value[i+1] + self.w[i]
120         self.X = KnapsackState()
```

```
121
122     def is_factible(self, x):
123         return x.weight <= self.W
124
125     def f(self, x):
126         return x.value
127
128     def optimistic(self, s):
129         v = OffsetArray([self.v[i] for i in xrange(len(s)+1, self.N+1)])
130         w = OffsetArray([self.w[i] for i in xrange(len(s)+1, self.N+1)])
131         W = self.W - s.weight
132         x = fractional_knapsack(w, v, W)
133         return s.value + sum(x[i]*self.v[i+len(s)] for i in xrange(1,len(x)+1))
134
135     def branch(self, s):
136         if not self.is_complete(s):
137             yield KnapsackState(s, 0, self.v, self.w)
138             if s.weight + self.w[len(s)+1] <= self.W:
139                 yield KnapsackState(s, 1, self.v, self.w)
```

```
1 from bb_knapsack import Knapsack, KnapsackState
2 from offsetarray import OffsetArray
3
4 W, v, w = 10, OffsetArray([10, 2, 3, 4, 2]), OffsetArray([12, 5, 6, 2, 6])
5 print Knapsack(W, v, w).solve()
```

```
[0, 0, 1, 1, 0]
```

Este algoritmo no es más eficiente que el último presentado en el sentido en el sentido de requerir un conjunto de estados activos de menor talla o en el de efectuar menos iteraciones o generar menos estados. Lo es en el sentido de que cada estado activo sólo requiere, per se, memoria de tamaño constante, pues comparte la memoria ocupada por el prefijo que tiene en común con su padre con el propio estado padre. Por otra parte, actualizar cierta información propia del estado, como la suma del valor y del peso de los objetos cargados en la mochila, requiere tiempo $O(1)$, en lugar de $O(N)$, pues suele bastar con sumar una cantidad a la información correspondiente en el estado padre.

9.3. Algunas variantes del esquema

9.3.1. Esquema con aplicación temprana de poda basada en la inicialización con una solución factible cualquiera

Un problema práctico con el que nos enfrentamos es la posibilidad de que el algoritmo tarde mucho en alcanzar una solución factible, independientemente de que sea o no la óptima. Esta tardanza supone la no disponibilidad de un valor que permita efectuar podas sobre el conjunto de estados activos y que, hasta ese instante, todo estado factible

fruto de la ramificación de un estado seleccionado ingrese en A. Se trata de un proceso capaz de generar un número de inserciones que es función exponencial del número de extracciones: $O(b^n)$, donde b es el factor de ramaje del árbol de estados y n es el número de estados seleccionados y ramificados.

Hay una forma de paliar este problema: inicializar la variable x (donde se guarda la mejor solución factible vista hasta el momento) con una solución factible cualquiera.

bbscheme.py *(cont.)*

```
229 class BranchAndBoundWithInitializationScheme(BranchAndBoundScheme):
230     def arbitrary_solution(self):
231         """Proporciona un elemento cualquiera de X."""
232 
233     def solve(self):
234         if self.is_complete(self.X): return self.X
235         A = self.priority_queue([(self.score(self.X), self.X)])
236         x = self.arbitrary_solution()
237         fx = self.f(x)
238         while len(A) != 0 and self.opt(A.opt()[0], fx) != fx:
239             (score_s, s) = A.extract_opt()
240             for s' in self.branch(s):
241                 if self.is_complete(s'):
242                     if self.is_factible(s') and self.opt(fx, self.f(s')) != fx:
243                         x, fx = s', self.f(s')
244                 else:
245                     optimistic_s' = self.optimistic(s')
246                     if self.opt(optimistic_s', fx) != fx:
247                         A.insert((optimistic_s', s'))
248         return x
```

Aunque x puede ser una solución cualquiera, conviene que sea una aproximación de la solución factible aunque, eso sí, de cálculo rápido. La estrategia voraz, por ejemplo, suele producir buenas inicializaciones eficientemente.

Aplicación a la mochila discreta

En el problema de la mochila discreta, por ejemplo, podemos inicializar x con una solución voraz, es decir, introduciendo en la mochila los objetos uno a uno mientras quepan. Eso sí, con la precaución de tratar de cargar primero los objetos con mejor ratio valor/peso.

bb_knapsack.py *(cont.)*

```
141 from bbscheme import BranchAndBoundWithInitializationScheme
142 from heap import MaxHeap
143 from offsetarray import OffsetArray
144 
145 class Knapsack7(KnapsackWithOptimisticBoundMixin2, BranchAndBoundWithInitializationScheme):
146     def arbitrary_solution(self):
147         W = self.W
148         ratios = sorted(((self.v[i]/float(self.w[i]), i) for i in xrange(1, self.N+1)), reverse=True)
```

```
149         x = OffsetArray([0]*self.N)
150         for (r, i) in ratios:
151             if self.w[i] <= W:
152                 x[i] = 1
153                 W -= self.w[i]
154         return x
```

```
1 from bb_knapsack import Knapsack7
2 from offsetarray import OffsetArray
3
4 W, v, w = 10, OffsetArray([10, 2, 3, 4, 2]), OffsetArray([12, 5, 6, 2, 6])
5 print Knapsack7(W, v, w).solve()
```

```
[0, 0, 1, 1, 0]
```

Un nuevo experimento permite comparar la eficiencia de la introducción de poda temprana en la tabla 9.5. Se puede apreciar que las mejoras en eficiencia afectan principalmente al máximo tamaño de A y al número de inserciones efectuadas.

N	máx $\|A\|$		inserciones		generados		extracciones	
3	2.4	1.2	3.6	2.0	6.5	5.0	3.5	2.8
4	3.5	1.4	6.0	1.9	8.8	4.6	4.7	2.3
5	4.0	1.5	7.8	3.6	11.1	7.8	6.0	4.3
6	5.4	1.9	10.8	5.0	13.8	9.8	7.7	5.3
7	5.8	2.0	12.1	6.8	15.5	12.4	8.6	7.0
8	7.0	1.9	15.1	6.7	18.3	12.2	10.2	7.0
9	9.3	1.9	19.3	8.7	23.7	17.6	12.4	9.3
10	11.4	3.1	23.6	12.3	27.9	22.9	14.4	11.8
11	11.4	3.8	27.1	16.6	32.5	28.4	18.3	15.9
12	12.7	3.8	32.3	19.4	37.5	31.4	21.9	18.6
13	18.3	4.5	43.4	27.3	50.5	48.3	27.5	26.1
14	20.6	5.2	56.1	37.6	70.3	66.8	39.2	37.4
15	20.0	3.9	46.5	26.8	53.8	48.3	29.4	26.4
16	19.8	4.0	47.4	25.4	57.0	46.6	30.6	24.8
17	21.9	4.7	56.4	35.8	66.8	62.5	37.3	34.7
18	25.3	4.4	58.3	32.6	66.6	58.8	35.9	31.6
19	26.9	4.0	60.4	29.8	67.8	54.6	35.8	29.1
20	27.4	4.3	72.2	44.5	88.8	83.0	47.7	44.3

Tabla 9.5: En cada columna el valor de la izquierda muestra el resultado para el algoritmo sin poda temprana y el de la derecha, el resultado para el algoritmo con poda temprana.

9.3.2. Esquema con cota pesimista

Ya hemos considerado la conveniencia de hacer podas efectivas tan pronto sea posible (inicializando, por ejemplo, el valor de x con una solución factible cualquiera). Hay otro modo de anticipar las podas: haciendo uso de una cota pesimista del valor de la mejor solución factible contenida a partir de un estado, es decir, una función *pessimistic* :

$\mathcal{P}(X') \to \mathbb{R}$ tal que

$$\operatorname{opt}\left(\mathit{pessimistic}(s),\ \operatorname*{opt}_{x \in s \cap X} f(x)\right) = \operatorname*{opt}_{x \in s \cap X} f(x).$$

La idea es sencilla: no podemos considerar prometedor un estado que presenta una cota optimista peor que la mejor cota pesimista vista hasta el momento. Nótese que de este modo no es necesario alcanzar una solución factible para empezar a podar.

La clave de que la cota pesimista sea efectivamente útil radica en la eficiencia con que podamos llevar a cabo su cálculo. Podemos integrarla en el proceso de búsqueda como se indica en este esquema:

bbscheme.py *(cont.)*

```
250 class BranchAndBoundWithPessimisticBound(BranchAndBoundScheme):
251     def pessimistic(self, s):
252         """Cota pesimista del valor óptimo de la función objetivo sobre elementos de s."""
253 
254     def solve(self):
255         A = self.priority_queue([(self.optimistic(self.X), self.X)])
256         x, fx = None, self.worst_value()
257         worst = self.pessimistic(self.X)
258         while len(A) != 0 and self.opt(A.opt()[0], fx) != fx:
259             (score_s, s) = A.extract_opt()
260             for s' in self.branch(s):
261                 if self.is_complete(s'):
262                     if self.is_factible(s') and self.opt(fx, self.f(s')) != fx:
263                         x, fx = s', self.f(s')
264                         worst = self.opt(worst, fx)
265                 else:
266                     optimistic' = self.optimistic(s')
267                     if self.opt(optimistic', worst) == optimistic' and self.opt(optimistic', fx) != fx:
268                         worst = self.opt(worst, self.pessimistic(s'))
269                         A.insert((optimistic', s'))
270         return x
```

En aras de la simplicidad, el esquema presentado no inicializa x con una solución arbitraria, pero esa inicialización resulta perfectamente compatible con el uso de la cota pesimista. No obstante, muchas veces la cota pesimista sobre X y la inicialización son equivalentes, ya que ambas se basan en el cálculo de una solución voraz.

Aplicación a la mochila discreta

En el problema de la mochila discreta hay un par de cotas pesimistas sencillas: no meter en la mochila objeto alguno de los que quedan (con lo que el beneficio obtenido es nulo) o el valor que proporciona una solución voraz de la parte desconocida. El uso de un cota pesimista puede ahorrar la inicialización de x con una solución factible arbitraria, pues la cota pesimista permite podar desde el principio.

bb_knapsack.py *(cont.)*

```
156 from bbscheme import BranchAndBoundWithPessimisticBound
157
158 class Knapsack8(KnapsackWithOptimisticBoundMixin2, BranchAndBoundWithPessimisticBound):
159     def pessimistic(self, s):
160         W = self.W - sum(s[i]*self.w[i] for i in xrange(1,len(s)+1))
161         value = sum(s[i]*self.v[i] for i in xrange(1,len(s)+1))
162         ratios = sorted(((self.v[i]/float(self.w[i]), i) for i in xrange(len(s)+1, self.N+1)), reverse=True)
163         for (r, i) in ratios:
164             if self.w[i] <= W:
165                 value += self.v[i]
166                 W -= self.w[i]
167         return value
```

```
1 from bb_knapsack import Knapsack8
2 from offsetarray import OffsetArray
3
4 W, v, w = 10, OffsetArray([10, 2, 3, 4, 2]), OffsetArray([12, 5, 6, 2, 6])
5 print Knapsack8(W, v, w).solve()
```

```
[0, 0, 1, 1, 0]
```

La tabla **??** muestra el resultado de un experimento que compara el uso de poda temprana con el uso de cota pesimista. Se observa un mayor reducción de la talla máxima de A al usa cota pesimista, pero un ligero empeoramiento en el tiempo de ejecución. Es obvio que una combinación de los dos principios comparados aquí conduciría a un algoritmo más eficiente en todos los indicadores.

N	máx $\|A\|$		inserciones		generados		extracciones	
3	1.2	1.4	2.0	2.6	5.0	6.5	2.8	3.5
4	1.4	1.4	1.9	3.8	4.6	8.8	2.3	4.7
5	1.5	1.5	3.6	5.0	7.8	11.1	4.3	6.0
6	1.9	1.9	5.0	6.8	9.8	13.8	5.3	7.7
7	2.0	1.9	6.8	7.8	12.4	15.5	7.0	8.6
8	1.9	2.0	6.7	9.5	12.2	18.3	7.0	10.2
9	1.9	2.1	8.7	11.6	17.6	23.7	9.3	12.4
10	3.1	2.5	12.3	13.8	22.9	27.9	11.8	14.4
11	3.8	3.4	16.6	18.1	28.4	32.5	15.9	18.3
12	3.8	3.0	19.4	21.1	31.4	37.5	18.6	21.9
13	4.5	3.3	27.3	26.9	48.3	50.5	26.1	27.5
14	5.2	4.6	37.6	38.5	66.8	70.3	37.4	39.2
15	3.9	3.1	26.8	28.6	48.3	53.8	26.4	29.4
16	4.0	3.4	25.4	29.9	46.6	57.0	24.8	30.6
17	4.7	3.8	35.8	36.6	62.5	66.8	34.7	37.3
18	4.4	3.9	32.6	35.6	58.8	66.6	31.6	35.9
19	4.0	3.5	29.8	35.3	54.6	67.8	29.1	35.8
20	4.3	3.8	44.5	47.0	83.0	88.8	44.3	47.7

Tabla 9.6: En cada columna se muestra a mano izquierda el resultado para el algoritmo con la cota optimista basada en la mochila con fraccionamiento y a mano derecha el resultado para el algoritmo con cota pesimista basada en el algoritmo voraz.

9.3.3. Esquema con aceleración del cómputo con sacrificio controlado de la optimalidad

La elevada complejidad computacional de la ramificación y poda se debe, en muchos casos, al hecho de que deseamos calcular con exactitud la solución óptima. Si nos conformamos con una aproximación a ella (algo razonable en muchos problemas de interés práctico) podemos obtener un resultado mucho más rápidamente. Basta para ello con definir una tolerancia, $tolerance : \mathbb{R} \to \mathbb{R}$, que nos permita sacrificar controladamente la optimalidad del resultado. Decimos entonces que obtenemos una solución **subóptima**.

Centremos el discurso en problemas de maximización, pues resulta farragoso hacerlo en términos generales. Hasta el momento hemos podado aquellos estados s tales que $optimistic(s) \leq f(x)$, siendo x la mejor solución factible conocida en cada instante. Si podamos aquellos estados s tales que

$$optimistic(s) \leq f(x) + tolerance(f(x))$$

obtenemos una solución subóptima, x_0, cuya «distancia» a la óptima, $\hat{x}$, se encuentra acotada:

$$f(x_0) \leq f(\hat{x}) \leq f(x_0) + tolerance(f(x_0)).$$

Normalmente, se utilizan como funciones de tolerancia las que suman un error permisible absoluto y las que suman un error permisible relativo, es decir, funciones de la forma:

- $tolerance(v) = E_a$, para $E_a \geq 0$: el **error absoluto** cometido es menor o igual que E_a.
- $tolerance(v) = E_r v$, $E_r \geq 0$: el **error relativo** cometido es inferior o igual al $100E_r$ %.

El esquema queda así:

bbscheme.py *(cont.)*

```
272 class ApproximateBranchAndBoundScheme(BranchAndBoundScheme):
273     def tolerance(self, fx):
274         """Tolerancia con la que una solución se considera aceptable."""
275 
276     def solve(self):
277         A = self.priority_queue([(self.optimistic(self.X), self.X)])
278         x, fx, approx_fx = None, self.worst_value(), self.worst_value()
279         while len(A) != 0 and self.opt(A.opt()[0], approx_fx) != approx_fx:
280             (score_s, s) = A.extract_opt()
281             for s' in self.branch(s):
282                 if self.is_complete(s'):
283                     if self.is_factible(s') and self.opt(fx, self.f(s')) != fx:
284                         x, fx = s', self.f(s')
285                         approx_fx = fx + self.tolerance(fx)
286                 else:
287                     optimistic_s' = self.optimistic(s')
288                     if self.opt(optimistic_s', approx_fx) != approx_fx:
289                         A.insert((optimistic_s', s'))
290         return x
```

Si hemos de demostrar la corrección de un algoritmo derivado del esquema para un conjunto X' de cardinalidad finita, hemos de indicar que sólo restará por comprobar las Condiciones B1 y B2.

Si X' no es de cardinalidad finita, aún es posible que el algoritmo de ramificación y poda sea de aplicación, pero se requiere una demostración *ad hoc*. Lo dicho en el último apartado sobre la monotonía menos optimista de la función de puntuación y cota optimista puede resultar de interés para este objetivo. Más adelante veremos un ejemplo.

Aplicación a la mochila discreta

Este algoritmo proporciona una solución factible con la garantía de que el beneficio de la solución óptima es, a lo sumo, un 10 % superior:

bb_knapsack.py *(cont.)*

```
169 from bbscheme import ApproximateBranchAndBoundScheme
170
171 class Knapsack9(KnapsackWithOptimisticBoundMixin2, ApproximateBranchAndBoundScheme):
172     def tolerance(self, fx):
173         return 0.1 * fx
```

```
1 from bb_knapsack import Knapsack9
2 from offsetarray import OffsetArray
3
4 W, v, w = 10, OffsetArray([10, 2, 3, 4, 2]), OffsetArray([12, 5, 6, 2, 6])
5 print Knapsack9(W, v, w).solve()
```

```
[0, 0, 1, 1, 0]
```

Hemos ejecutado el experimento comparativo entre este algoritmo aproximado y el último con el resultado que se muestra en la tabla 9.7. La reducción del coste espacial y temporal son drásticas. La solución que proporcionó el algoritmo aproximado con una tolerancia del 10 % coincidió con la exacta el 81.9 % de las veces.

9.3.4. Esquema de ramificación y poda para problemas resolubles por programación dinámica

Algunos problemas resolubles por programación dinámica se abordan también por ramificación y poda. El objetivo es diseñar algoritmos más eficientes en la práctica, aunque presenten un coste asintótico para el peor de los casos igual al de programación dinámica.

Cada estado presenta ciertas decisiones ya adoptadas y otras que están pendientes de adopción. La decisiones que aún no se han adoptado definen una instancia diferente de un problema de la misma naturaleza que el original. La esencia de la programación dinámica es la memorización de resultados para instancias del problema ya resueltas. Los estados que definen instancias equivalentes es sus respectivas partes desconocidas

N	máx $\|A\|$		inserciones		generados		extracciones	
3	2.4	2.4	3.6	3.5	6.5	6.1	3.5	3.4
4	3.5	3.5	6.0	6.0	8.8	8.8	4.7	4.7
5	4.0	4.0	7.8	7.8	11.1	10.8	6.0	5.8
6	5.4	5.4	10.8	10.7	13.8	13.4	7.7	7.4
7	5.8	5.8	12.1	11.9	15.5	14.8	8.6	8.2
8	7.0	7.0	15.1	14.8	18.3	17.6	10.2	9.8
9	9.3	9.3	19.3	17.9	23.7	20.9	12.4	10.8
10	11.4	11.4	23.6	22.5	27.9	25.9	14.4	13.4
11	11.4	11.1	27.1	24.4	32.5	27.6	18.3	15.4
12	12.7	11.9	32.3	28.5	37.5	31.9	21.9	18.6
13	18.3	16.4	43.4	35.0	50.5	37.9	27.5	20.6
14	20.6	18.9	56.1	41.5	70.3	44.7	39.2	24.6
15	20.0	19.1	46.5	40.1	53.8	42.8	29.4	23.1
16	19.8	19.4	47.4	39.2	57.0	41.6	30.6	21.9
17	21.9	21.2	56.4	47.6	66.8	51.6	37.3	28.6
18	25.3	23.6	58.3	52.0	66.6	57.5	35.9	30.6
19	26.9	25.4	60.4	52.8	67.8	56.3	35.8	29.5
20	27.4	24.6	72.2	52.9	88.8	57.0	47.7	30.4

Tabla 9.7: En cada columna se muestra a mano izquierda el resultado para el algoritmo exacto y a mano derecha, el resultado para el algoritmo aproximado.

son equivalentes desde el punto de vista de la programación dinámica. Imaginemos una instancia del problema en la que $W = 7$, $N = 5$, $w_1 = 2$, $w_2 = 3$, $w_3 = 2$, $w_4 = 1$, $w_5 = 4$, $v_1 = 1$, $v_2 = 2$, $v_3 = 3$, $v_4 = 4$ y $v_5 = 5$; los estados $(1, 0, 0, ?)$ y $(0, 0, 1, ?)$ son equivalentes en el sentido de que ambos proponen, en sus partes desconocidas, la resolución de una misma instancia del problema: el cálculo de la carga óptima para una mochila con capacidad $W = 5$. Sea cual sea esa carga óptima, los dos estados se completan óptimamente con ella. Y la compleción del segundo estado proporcionará siempre mayor valor total que la del primero.

Bajo ciertas condiciones, si dos estados proponen instancias idénticas del problema en su parte desconocida, son estados equivalentes. Dados dos estados equivalentes, sólo tiene interés uno de ellos: el que proporciona mayor valor en su parte conocida.

La técnica de memorización puede jugar un papel interesante a la hora de evitar cálculos innecesarios en ramificación y poda. Podemos gestionar una tabla que asocie representantes de «estados equivalentes» a valores de la función objetivo sobre la parte conocida. Cada vez que se genera un estado que ya tiene entrada en la tabla, se estudia si mejora o no el valor de la tabla. Si no lo mejora, se poda. Si lo mejora, se tiene en cuenta.

`bbscheme.py` *(cont.)*

```
292 class BranchAndBoundWithMemoizationScheme(BranchAndBoundScheme):
293     def equivalence_id(self, s):
294         """Devuelve un identificador único de la instancia del problema que se forma con
295  la parte desconocida."""
296 
297     def g(self, s):
298         """Devuelve la aportación de f de la 'parte conocida' en el estado s."""
299 
300     def solve(self):
```

```
301         A = self.priority_queue([(self.score(self.X), self.X)])
302         mem = { self.equivalence_id(self.X): self.g(self.X) }
303         x, fx = None, self.worst_value()
304         while len(A) != 0 and self.opt(A.opt()[0], fx) != fx:
305             (score_s, s) = A.extract_opt()
306             for s' in self.branch(s):
307                 if self.is_complete(s'):
308                     if self.is_factible(s') and self.opt(fx, self.f(s')) != fx:
309                         x, fx = s', self.f(s')
310                 else:
311                     eq = self.equivalence_id(s')
312                     g_eq = mem.get(eq, self.worst_value())
313                     g_s' = self.g(s')
314                     if self.opt(g_eq, g_s') != g_eq:
315                         mem[eq] = g_s'
316                         optimistic_s' = self.optimistic(s')
317                         if self.opt(optimistic_s', fx) != fx:
318                             A.insert((optimistic_s', s'))
319         return x
```

Aplicación a la mochila discreta

bb_knapsack.py *(cont.)*

```
175 from bbscheme import BranchAndBoundWithMemoizationScheme
176 from prioritydict import MaxPriorityDict
177
178 class KnapsackWithMemoization(KnapsackWithOptimisticBoundMixin2, BranchAndBoundWithMemoizationScheme):
179     def equivalence_id(self, s):
180         return (self.N - len(s), self.W-sum(s[i]*self.w[i] for i in xrange(1, len(s)+1)))
181
182     def g(self, s):
183         return sum(s[i]*self.v[i] for i in xrange(1, len(s)+1))
```

```
1 from bb_knapsack import KnapsackWithMemoization
2 from offsetarray import OffsetArray
3
4 W, v, w = 10, OffsetArray([10, 2, 3, 4, 2]), OffsetArray([12, 5, 6, 2, 6])
5 print KnapsackWithMemoization(W, v, w).solve()
```

```
[0, 0, 1, 1, 0]
```

EJERCICIOS

9-1 Otra mejora posible pasa por detener la ramificación cuando la capacidad de la mochila es inferior a la del objeto de menos peso de los no considerados hasta el momento. Implementa esta mejora y compara experimentalmente el nuevo algoritmo con el anteriormente presentado.

9-2 Aún hay un refinamiento adicional de fácil implementación. El algoritmo voraz considera la inclusión de los objetos de mayor a menor ratio beneficio/peso. Parece interesante considerar

este orden de consideración de objetos en el propio proceso de ramificación de estados: de ese modo, consideramos primero estados que ofrecen un elevado beneficio a cambio de una baja ocupación de la mochila. Implementa este refinamiento (junto al del ejercicio anterior) y evalúa experimentalmente la ganancia en eficiencia.

9-3 Puede que haya más de una forma de cargar la mochila de modo que se obtenga el máximo beneficio. Modifica el algoritmo para que proporcione todas las selecciones de objetos de valor óptimo. ¿Puedes generalizar el cambio efectuado y presentar un esquema de ramificación y poda que proporciona todas las soluciones óptimas?

9-4 Modifica el algoritmo de ramificación y poda para que genere todas las soluciones factibles ordenadas de menor a mayor beneficio.

9.4. El problema del ensamblaje

La construcción de un artefacto requiere ensamblar M piezas que identificamos con números entre 0 y $M-1$. El coste de ensamblar la pieza i depende del número de piezas ya ensambladas. Los costes, que son valores positivos, se nos proporcionan en una matriz c. La celda $c_{i,j}$ es el coste de ensamblar la pieza i cuando ya se han ensamblado j piezas. Deseamos calcular el orden de ensamblado de menor coste total, entendido éste como la suma del coste de ensamblaje de cada una de las piezas.

Este problema es el conocido como problema del emparejamiento bipartito (en inglés, bipartite matching). Un grafo es bipartito si el conjunto de vértices V puede dividirse en dos conjuntos, V_1 y V_2, tales que si $(u,v) \in E$, entonces $u \in V_1$ y $v \in V_2$. El problema del emparejamiento bipartito consiste en dado un grafo bipartito y ponderado, encontrar un subconjunto de E tal que cada vértice de V_1 esté en correspondencia con uno y solo uno de V_2 y la suma de los pesos de las aristas del subconjunto sea mínima. El denominado algoritmo húngaro encuentra una solución en tiempo $O(|V|^3)$.

Podemos representar las soluciones con tuplas de la forma $(x_0, x_1, \ldots, x_{M-1})$, donde x_i es el número de piezas montadas en el momento en que se decide montar la pieza que identificamos con el índice i. El espacio de búsqueda es el conjunto de permutaciones de los enteros $\{0, 1, \ldots, M-1\}$:

$$X = \{(x_0, x_1, \ldots, x_{M-1}) \in [0..M-1]^M \mid x_i \neq x_j, 0 \leq i < j < M\},$$

y la función objetivo, que deseamos minimizar, es

$$f((x_0, x_1, \ldots, x_{M-1})) = \sum_{0 \leq i < M} c_{i,x_i}.$$

9.4.1. Estados y función de ramificación

Podemos representar el conjunto de elementos de X que comparten un prefijo de k elementos con la tupla $(x_0, x_1, \ldots, x_{k-1}, ?)$ con $k < M$. Al ramificar un estado de esa forma obtenemos el siguiente conjunto de $M-k$ nuevos estados:

$$branch((x_0, x_1, \ldots, x_{k-1}, ?)) = \{(x_0, x_1, \ldots, x_{k-1}, x_k, ?) \mid 0 \leq x_k < M; \quad x_k \neq x_i, 0 \leq i < k\}.$$

9.4.2. Una cota inferior trivial

Necesitamos diseñar ahora un estimación optimista de la mejor solución que podemos alcanzar a partir de un estado, es decir, una cota inferior al valor de la función objetivo sobre la soluciones factibles de un estado.

Se puede diseñar una cota inferior trivial si sólo consideramos el coste de las piezas cuyo orden de ensamblado ya se ha decidido, es decir, teniendo en cuenta sólo la parte conocida de las soluciones de un estado.

$$optimistic((x_0, x_1, \ldots, x_{k-1}, ?)) = \sum_{0 \leq i < k} c_{i,x_i}.$$

Es una cota optimista que supone que se podrán ensamblar las piezas restantes sin coste alguno.

Aprovechamos para presentar una implementación eficiente en la representación de los estados. Empezamos por definir una clase *AssemblyState* que hereda de *State* y enriquece a esta última clase con un atributo que mantiene la puntuación y un método que proporciona un conjunto con los enteros entre 0 y $M - 1$ que forman parte del estado. Este método resultará útil para ramificar un estado eficientemente:

assembly.py

```
1  from bbscheme import State
2
3  class AssemblyState(State):
4      def __init__(self, parent=None, sk=None, c=None):
5          State.__init__(self, parent, sk)
6          self.score = parent.score + c[len(self)-1][sk] if parent != None else 0
7
8      def numbers(self):
9          if self.parent == None: return set([])
10         nums = self.parent.numbers()
11         nums.add(self.sk)
12         return nums
```

Disponer del atributo *score* hace que el cálculo de la cota, conociendo el valor *score* en el padre, sea una operación $O(1)$. Nótese que el valor de f sobre un estado completo coincide con el valor almacenado en *score*.

assembly.py *(cont.)*

```
15 from bbscheme import BranchAndBoundScheme
16 from heap import MinHeap
17 from utils import infinity
18
19 class Assembly(BranchAndBoundScheme):
20     def __init__(self, c):
21         self.c = c
22         self.M = len(c)
23         self.X = AssemblyState()
24
```

```
25    def is_complete(self, s):
26        return len(s) == self.M
27
28    def is_factible(self, s):
29        return True
30
31    def worst_value(self):
32        return infinity
33
34    def branch(self, s):
35        numbers = s.numbers()
36        for i in xrange(self.M):
37            if i not in numbers:
38                yield AssemblyState(s, i, self.c)
39
40    def f(self, x):
41        return x.score
42
43    def optimistic(self, s):
44        return s.score
45
46    def opt(self, a, b):
47        return min(a, b)
48
49    def priority_queue(self, iterable=[]):
50        return MinHeap(iterable)
```

test_assembly.py

```
1  from assembly import Assembly
2
3  c = [[3, 1, 2, 1], [3, 3, 4, 2], [2, 1, 1, 1], [4, 2, 4, 3]]
4  x = Assembly(c).solve()
5  print 'Las piezas se ensamblan en el orden', x, 'con coste', x.score
```

```
Las piezas se ensamblan en el orden [3, 0, 2, 1] con coste 7
```

9.4.3. Una cota más informada

Podemos mejorar la cota si tenemos en cuenta el menor coste con que es posible ensamblar cada una de las piezas por montar y sumamos estas cantidades al coste de la parte conocida.

$$optimistic((x_0, x_1, \ldots, x_{k-1}, ?)) = \sum_{0 \le i < k} c_{i,x_i} + \sum_{k \le i < M} \min_{0 \le j < M} c_{i,j}.$$

Esta cota resulta computable eficientemente si precalculamos $minc_k = \min_{0 \le j < M} c_{k,j}$ para todo valor de i. El cálculo de la cota sobre los estados resultantes de ramificar otro es muy sencillo:

$$optimistic((x_0, x_1, \ldots, x_{k-1}, x_k, ?)) = optimistic((x_0, x_1, \ldots, x_{k-1}, ?)) + c_{k,x_k} - minc_k.$$

Este valor se puede calcular eficientemente si se almacena el valor de la cota optimista en el propio estado.

El siguiente algoritmo de ramificación y poda usa *optimistic* como cota inferior y como función de puntuación de estados.

assembly.py *(cont.)*

```
52 class AssemblyState2(AssemblyState):
53     def __init__(self, parent=None, sk=None, c=None, minc=None):
54         State.__init__(self, parent, sk)
55         self.score = parent.score + c[len(self)-1][sk] - minc[sk] \
56             if parent != None else sum(minc)
57 
58 class Assembly2(Assembly):
59     def __init__(self, c):
60         Assembly.__init__(self,c)
61         self.minc = [min(row) for row in c]
62         self.X = AssemblyState2(minc=self.minc)
63 
64     def branch(self, s):
65         numbers = s.numbers()
66         for i in xrange(self.M):
67             if i not in numbers:
68                 yield AssemblyState2(s, i, self.c, self.minc)
```

test_assembly2.py

```
1 from assembly import Assembly2
2 
3 c = [[3, 1, 2, 1], [3, 3, 4, 2], [2, 1, 1, 1], [4, 2, 4, 3]]
4 x = Assembly2(c).solve()
5 print 'Las piezas se ensamblan en el orden', x, 'con coste', x.score
```

```
Las piezas se ensamblan en el orden [3, 0, 2, 1] con coste 7
```

9.4.4. Inicialización con una solución voraz

Podemos mejorar el algoritmo si partimos de una solución inicial arbitraria —por ejemplo, la permutación $(0, 1, \ldots, M-1)$—, pues así la poda será efectiva desde del principio:

assembly.py *(cont.)*

```
70 from bbscheme import BranchAndBoundWithInitializationScheme
71 
72 class Assembly3(Assembly, BranchAndBoundWithInitializationScheme):
73   def arbitrary_solution(self):
74       print "**"
75       return range(self.M)
```

```
from assembly import Assembly3

c = [[3, 1, 2, 1], [3, 3, 4, 2], [2, 1, 1, 1], [4, 2, 4, 3]]
x = Assembly3(c).solve()
print 'Las piezas se ensamblan en el orden', x, 'con coste', x.score
```

```
Las piezas se ensamblan en el orden [3, 0, 2, 1] con coste 7
```

EJERCICIOS

9-5 La inicialización puede ser más efectiva si considera una solución construida siguiendo una estrategia voraz. Modifica el programa para inicializar x con una solución «razonablemente» buena. Haz una traza de las dos variantes del algoritmo con la instancia del ejemplo.

9.4.5. Una cota más informada

Es posible ajustar más la cota inferior. Esta cota, por ejemplo, puntúa la parte desconocida sin tener en cuenta los instantes de tiempo en que ya se está montando una pieza:

$$optimistic((x_0, x_1, \ldots, x_{k-1}, ?)) = \sum_{0 \le i < k} c_{i,x_i} + \sum_{k \le i < M} \min_{\substack{0 \le j < M: \\ j \ne x_i, 0 \le i < k}} c_{i,j}.$$

El problema con esta cota es que no podemos efectuar un precálculo al estilo del que efectuamos antes, por lo que resulta computacionalmente costosa. Con esta cota, el cálculo asociado a la generación de un estado requiere tiempo $O(M^2)$.

EJERCICIOS

9-6 Desarrolla un algoritmo de ramificación y poda que resuelva el mismo problema, pero modelando las soluciones de modo que x_i, en una tupla $(x_0, x_1, \ldots, x_{M-1})$, represente que en el instante i se monta la pieza x_i. Define la función de ramificación y cotas apropiadas. ¿Es más eficiente el cálculo de una cota equivalente a la última propuesta en el desarrollo de la solución anterior?

9-7 Se desea asignar n tareas a n agentes, de forma que cada agente realice exactamente una tarea. Si al agente a se le asigna la tarea t, entonces el coste de realizar esa tarea es $c_{a,t}$. El problema consiste en encontrar una asignación de tareas a agentes que minimice el coste total de ejecutar las n tareas.

a) Describe brevemente los componentes (estructura de un estado y las funciones de ramificación y cota optimista) de un algoritmo de ramificación y poda que resuelva el problema propuesto.

b) Haz una traza de la ejecución del algoritmo de ramificación y poda para una instancia en la que $n = 5$ y la matriz c viene dada por:

		t				
		1	2	3	4	5
	1	13	19	12	40	10
	2	12	15	16	23	21
a	3	11	17	19	21	12
	4	12	14	18	22	26
	5	10	24	12	23	21

Donde las filas son los agentes y las columnas las tareas. Por ejemplo $c_{2,3} = 13$ indica que el coste de realizar la tarea 3 por el agente 2 es 13.

c) ¿Qué modificaciones habría que hacer para que el algoritmo de ramificación y poda para devuelva *todas* las soluciones óptimas?

9-8 Una agencia gestiona la ficha de n proveedores y n clientes. Cada proveedor ha consultado el expediente de todos los clientes y ha manifestado interés por suministrarles servicios. El interés de cada proveedor por atender a cada cliente han sido cuantificado mediante una puntuación de 0 a 10: $p_interes[i,j]$ es la puntuación otorgada al interés del proveedor i por atender al cliente j. También los clientes han expresado su interés por ser atendidos por cada uno de los proveedores. En $c_interes[i,j]$ se ha almacenado el interés (también de 0 a 10) manifestado por el cliente i en ser atendido por el proveedor j.

La agencia desea asignar un único proveedor a cada cliente y un único cliente a cada proveedor de modo que la «satisfacción» obtenida por ellos sea máxima. Entendemos por «satisfacción» del par formado por el proveedor i y el cliente j la suma de sus intereses mutuos, es decir, $p_interes[i,j] + c_interes[j,i]$.

Escribe un algoritmo de ramificación y poda que asigne óptimamente a cada proveedor un único cliente y viceversa.

9-9 Dada una matriz M de K filas por K columnas, se dice que una K-tupla de posiciones de M,

$$t = ((i_1, j_1), (i_2, j_2), ..., (i_K, j_K))$$

es una «Configuración Libre de inter-Ataque Torre (CLAT)» si $i_m \neq i_n$ y $j_m \neq j_n$, para todo par m y n tal que $m \neq n$ y $1 \leq m, n \leq K$.

Sea $S(t) = \sum_{1 \leq k \leq K} M_{i_k, j_k}$ la suma de las K componentes indicadas por la tupla t y sea T_M el conjunto de las CLAT de M.

Por ejemplo, sea M esta matriz

1	22	23	19	18
2	9	3	15	20
14	4	5	8	17
12	10	7	6	24
11	13	21	20	25

La tupla $t = ((1,1),(2,3),(3,2),(4,4),(5,5))$ es una CLAT con $S(t) = 39$. Otra CLAT mejor es $t' = ((1,2),(2,4),(3,1),(4,5),(5,3))$, con $S(t') = 96$.

a) Desarrolla un algoritmo de ramificación y poda que encuentre una k-tupla $\hat{t}$ de M cuya suma de componentes sea máxima, es decir,

$$\hat{t} = \arg \max_{t \in T_M} S(t).$$

Para el desarrollo de una cota superior puedes hacer uso del hecho de que

$$\text{mín} \left(\sum_{1 \leq i \leq k} \max_{1 \leq j \leq k} M_{ij}, \sum_{1 \leq j \leq k} \max_{1 \leq i \leq k} M_{ij} \right).$$

es mayor o igual que el valor de S sobre cualquier CLAT.

b) Representa el árbol de estados correspondiente a la ejecución del algoritmo para la instancia del ejemplo, indicando en cada nodo su estado y el respectivo valor de la cota. Explica brevemente los cálculos realizados.

c) Refina el algoritmo para que considere la inicialización de la mejor solución vista hasta el momento con una solución factible cualquiera.

d) ¿Puedes diseñar cotas optimistas más informadas?

. . . . . .

9.5. El problema del viajante de comercio

Dado un digrafo $G = (V, E)$ ponderado por una función $d : E \to \mathbb{R}^{\geq 0}$, deseamos encontrar un ciclo hamiltoniano de peso mínimo. Recordemos que un ciclo hamiltoniano es un camino que visita todos los vértices una sola vez, excepto el de partida, que se visita dos veces. (Sin pérdida de generalidad, supondremos que el vértice inicial es un vértice arbitrario u, pues cualquier vértice del grafo puede considerarse inicio y final del ciclo.)

El conjunto de soluciones factibles es

$$X = \left\{ (v_0, v_1, \ldots, v_{|V|}) \in V^{|V|+1} \mid v_0 = v_{|V|}; \bigcup_{0 \leq i < |V|} \{v_i\} = V; (v_i, v_{i+1}) \in E, 0 \leq i < |V| \right\}.$$

Nuestra función objetivo es

$$f((v_0, v_1, \ldots, v_{|V|})) = \sum_{0 \leq i < |V|} d(v_i, v_{i+1}),$$

pues deseamos calcular

$$(\hat{v}_0, \hat{v}_1, \ldots, \hat{v}_{|V|}) = \arg \min_{(v_0, v_1, \ldots, v_{|V|}) \in X} \sum_{0 \leq i < n} d(v_i, v_{i+1}).$$

La figura 9.9 muestra un grafo ponderado y su ciclo hamiltoniano más corto. Usaremos esta implementación del grafo en nuestras pruebas:

traveling_salesman_graph.py

```
from graph import Graph, WeightingFunction

d = WeightingFunction({(0,1): 0, (0,2): 15, (0,3): 2, (1,3): 3, (1,4): 13, (2,3): 11,
                       (2,5): 4, (3,4): 5, (3,5): 8, (3,6): 12, (4,7): 9, (5,6): 16,
                       (5,8):10, (6,7): 17, (6,8): 1, (6,9): 6, (7,9): 14, (8,9): 7},
                       symmetrical=True)
G = Graph(E=d.keys())
```

9.5.1. Estados y ramificación

En los problemas en que se desea calcular un camino en un grafo es frecuente modelar los estados como prefijos de un camino completo. Representaremos un estado con una secuencia de vértices no repetidos. Un estado representará una solución completa cuando

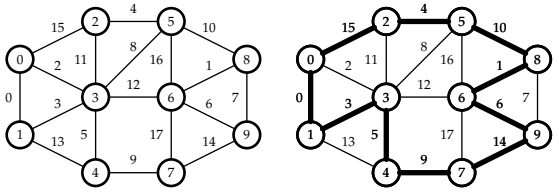


Figura 9.9: (a) Grafo sobre el que se propone la búsqueda del ciclo hamiltoniano más corto. (b) Ciclo hamiltoniano de coste mínimo.

su talla coincida con el número de vértices y será factible si hay una arista entre el último vértice y el primero.

La ramificación de un estado $(v_0, v_1, \ldots, v_k, ?)$ será el conjunto de estados de la forma $(v_0, v_1, \ldots, v_k, v_{k+1}, ?)$, donde (v_k, v_{k+1}) es una arista del grafo y el vértice v_{k+1} es diferente de cualquiera de los que aparecen en el prefijo de camino.

9.5.2. Cotas

Vamos a presentar numerosos algoritmos de ramificación y poda, cada uno con una cota optimista diferente. Todas ellas suman una cantidad al peso de la parte conocida de un estado. Con objeto de facilitar el seguimiento de la exposición, mostramos ahora una lista de las diferentes cotas agrupadas en familias.

- Cota trivial:

  $optimistic_1(v_0, v_1, \ldots, v_k, ?)$**:** Empezaremos con una cota trivial: el coste del camino recorrido hasta el momento. Es un cota optimista, muy poco informada es decir, demasiado optimista: supone que el ciclo se completa con un camino que recorre una distancia nula.

- Cotas basadas en la selección de la mejores aristas con las que completar el ciclo:

  $optimistic_2(v_0, v_1, \ldots, v_k, ?)$**:** Esta una cota que supone que las $|V| - k + 1$ aristas necesarias para completar el ciclo son las $|V| - k + 1$ de menos coste. Es una cota mejor informada que la anterior (al menos supone que completar el camino supone recorrer una distancia de depende, de algún modo, de los arcos). Esta cota puede calcularse incrementalmente.

  $optimistic_3(v_0, v_1, \ldots, v_k, ?)$**:** Mejora a la anterior cota suponiendo que dichas aristas son las $|V| - k + 1$ de menor coste de entre las que no han sido ya utilizadas. Es una cota mejor informada que la anterior: se puede demostrar que $optimistic_2 x(s)$ es mayor o igual que $optimistic_1(s)$ para todo estado. Esta cota puede calcularse incrementalmente.

  $optimistic_4(v_0, v_1, \ldots, v_k, ?)$**:** Una cota aún más informada considera sólo las aristas de mínimo coste que parten de estados no visitados, más la de mínimo que parte del último estado. Se trata de un cota de cálculo incremental.

  $optimistic_5(v_0, v_1, \ldots, v_k, ?)$**:** Esta cota considera las aristas de mínimo coste que parten y llegan a vértices no visitados. Pese a ser una cota mejor informada que

$optimistic_4$ presenta una desventaja: su coste computacional es elevado por no permitir un preproceso que sí permiten las cotas anteriores.

- Cotas basadas en el cálculo del camino más corto con el que cerrar un ciclo:

  $optimistic_6(v_0, v_1, \ldots, v_k, ?)$**:** El ciclo debe completarse con un camino de v_k a v_0. Podemos considerar como estimación optimista de la mejor forma de completar el ciclo, el peso del camino más corto de v_k a v_0. Esta cota, tras un preproceso, puede calcularse incrementalmente.

  $optimistic_7(v_0, v_1, \ldots, v_k, ?)$**:** La anterior cota puede mejorarse si se considera que el camino no debe tocar vértices ya visitados en la parte conocida. Esta cota, mejor informada que la anterior, resulta computacionalmente mucho más costosa.

- Cotas basadas en el cálculo del árbol de recubrimiento de coste mínimo:

  $optimistic_8(v_0, v_1, \ldots, v_k, ?)$**:** La parte desconocida de un estado es un camino, que es un caso particular de árbol de recubrimiento para los vértices no visitados, incluyendo a v_0 y v_k.

9.5.3. Una cota trivial: el coste del camino recorrido

Nos resta la parte más difícil: el diseño de una función de puntuación y una cota inferior que ayuden a orientar la búsqueda y a reducir significativamente el espacio de búsqueda. Las diferentes soluciones que propondremos a continuación utilizan una sola función para ambos cometidos y aparecen expuestas por orden creciente de «calidad» (y complejidad).

La primera función que consideramos es trivial: estima que es posible completar un camino incompleto con coste cero.

$$optimistic_1((v_0, v_1, \ldots, v_k, ?)) = \sum_{0 \le i < k} d(v_i, v_{i+1}).$$

Aunque esta cota se puede calcular incrementalmente en tiempo $O(1)$, es bastante inefectiva a la hora de podar el espacio de búsqueda. El algoritmo explorará todo camino cuyo peso sea menor que el del ciclo hamiltoniano de coste mínimo. El número de dichos caminos crece, probablemente, exponencialmente con la talla del grafo.

travelingsalesman.py

```
1 from bbscheme import BranchAndBoundScheme, State
2 from utils import infinity
3 from heap import MinHeap
4
5 class SalesmanState(State):
6     def __init__(self, parent=None, sk=None, d=None):
7         State.__init__(self, parent, sk)
8         self.score = parent.score + d(parent.sk, sk) if parent != None else 0
```

```
9
10      def visited(self):
11          if self.parent == None: return set([self.sk])
12          nodes = self.parent.visited()
13          nodes.add(self.sk)
14          return nodes
15
16  class TravelingSalesman1(BranchAndBoundScheme):
17      def __init__(self, G, d):
18          self.G = G
19          self.d = d
20          self.first_vertex = list(G.V)[0]
21          self.X = SalesmanState(sk=self.first_vertex)
22
23      def is_complete(self, s):
24          return len(s) == len(self.G.V)
25
26      def is_factible(self, s):
27          return (s.sk, self.first_vertex) in self.G.E
28
29      def worst_value(self):
30          return infinity
31
32      def branch(self, s):
33          visited = s.visited()
34          for v in self.G.succs(s.sk):
35              if v not in visited:
36                  yield SalesmanState(s, v, self.d)
37
38      def f(self, x):
39          return x.score
40
41      def optimistic(self, s):
42          return s.score
43
44      def opt(self, a, b):
45          return min(a, b)
46
47      def priority_queue(self, iterable=[]):
48          return MinHeap(iterable)
49
50      def solve(self):
51          path = BranchAndBoundScheme.solve(self).list()
52          return path + [path[0]]
```

test_salesman1.py

```
1  from travelingsalesman import TravelingSalesman1
2  from traveling_salesman_graph import G, d
3
```

```
4 path = TravelingSalesman1(G, d).solve()
5 print 'Ciclo', path, 'con peso', sum(d(path[i],path[i+1]) for i in xrange(len(path)-1))
```

```
Ciclo [0, 1, 3, 4, 7, 9, 6, 8, 5, 2, 0] con peso 67
```

El grafo sobre el que hemos probado el programa y su ciclo hamiltoniano óptimo se muestran en la figura 9.9.

9.5.4. Una cota mejor informada: compleción del ciclo con las aristas de mínimo coste

Todos los caminos representados por un estado de la forma $(v_0, v_1, \ldots, v_k, ?)$ deben completarse añadiendo $|V| - k$ aristas. En el mejor de los casos, estas aristas serán las de menor peso en el grafo. Podemos suponer que se escogen éstas y diseñar así una función optimista:

$$optimistic_2((v_0, v_1, \ldots, v_k, ?)) = \sum_{0 \le i < k} d(v_i, v_{i+1}) + \sum_{1 \le j \le |V|-k} \min_{(u,v) \in E}^{j} d(u, v),$$

donde mín^j es el j-ésimo menor elemento de un conjunto.

No hace falta ordenar las $|E|$ aristas si sólo deseamos disponer de las $|V|$ mejores. Podemos, por ejemplo, seleccionar las $|V|$ mejores en tiempo esperado $O(\lg |E|)$ (pero $O(|E|)$ para el peor caso) y ordenar sólo los $|V|$ mejores.

Esta cota se basa en una relajación del problema: las $|V| - k$ aristas que faltan no tienen por qué formar un ciclo; es más, ni siquiera tienen que formar un camino; y más aún, ni siquiera tienen que escogerse de entre las que aún o han sido usadas.

El cálculo de $optimistic_2$ puede realizarse en tiempo constante al generar un estado a partir de otro en el proceso de ramificación. Basta para ello con acceder a un vector si tenemos almacenado el valor de $\text{mín}^j_{(u,v) \in E} d(u, v)$ para todo j:

$$optimistic_2((v_0, v_1, \ldots, v_k, ?)) = optimistic_2((v_0, v_1, \ldots, v_{k-1}, ?)) + d(v_{k-1}, v_k) - \min_{(u,v) \in E}^{k} d(u, v).$$

travelingsalesman.py *(cont.)*

```
55 class SalesmanState2(SalesmanState):
56     def __init__(self, parent=None, sk=None, d=None, weights=None):
57         State.__init__(self, parent, sk)
58         self.score = parent.score - weights[len(self)-1] + d(parent.sk,sk) if parent != None else sum(weights)
59
60 class TravelingSalesman2(TravelingSalesman1):
61     def __init__(self, G, d):
62         self.G = G
63         self.d = d
64         self.first_vertex = list(G.V)[0]
```

```
65          self.weights = sorted(d.values())[:len(G.V)]
66          self.X = SalesmanState2(sk=self.first_vertex, weights=self.weights)
67
68      def branch(self, s):
69          visited = s.visited()
70          for v in self.G.succs(s.sk):
71              if v not in visited:
72                  yield SalesmanState2(s, v, self.d, self.weights)
```

test_salesman2.py

```
1 from travelingsalesman import TravelingSalesman2
2 from traveling_salesman_graph import G, d
3
4 path = TravelingSalesman2(G, d).solve()
5 print "Ciclo", path, "con peso", sum(d(path[i],path[i+1]) for i in xrange(len(path)-1))
```

```
Ciclo [0, 1, 3, 4, 7, 9, 6, 8, 5, 2, 0] con peso 67
```

La cota $optimistic_2$ está mejor informada que $optimistic_1$: se puede demostrar que $optimistic_2(s) \geq optimistic_1(s)$ para todo estado, pues, evidentemente,

$$\begin{aligned} optimistic_2((v_0, v_1, \ldots, v_k, ?)) &= \sum_{0 \leq i < k} d(v_i, v_{i+1}) + \sum_{1 \leq j \leq |V|-k} \min_{(u,v) \in E}^{j} d(u,v) \\ &\geq \sum_{0 \leq i < k} d(v_i, v_{i+1}) \\ &= optimistic_1((v_0, v_1, \ldots, v_k, ?)). \end{aligned}$$

Pese a ser una cota mejor informada que su antecesora, todo estado con el mismo número de vértices conocidos completa su puntuación con el mismo valor, así que cuando se comparan dos estados con igual longitud de sus partes conocidas, se escoge el de menor distancia en la parte conocida. El algoritmo que usa $optimistic_2$ presenta un comportamiento próximo al del que usa $optimistic_1$.

9.5.5. Una cota más informada: compleción del ciclo con las aristas de mínimo coste no utilizadas

Este aspecto indeseable puede paliarse si no consideramos las aristas ya utilizadas por la parte conocida del ciclo que estamos formando.

$$optimistic_3((v_0, v_1, \ldots, v_k, ?)) = \sum_{0 \leq i < k} d(v_i, v_{i+1}) + \sum_{1 \leq j \leq |V|-k} \min_{(u,v) \in E - \{(v_i, v_{i+1}) | 0 \leq i < k\})}^{j} d(u,v).$$

Esta cota se basa en una relajación del problema menos severa que la de $optimistic_2$: mantenemos una restricción del problema original: las aristas que seleccionamos no pueden haber sido elegidas previamente.

La gestión de estados para el cálculo de esta cota es relativamente más compleja:

travelingsalesman.py *(cont.)*

```
74 class SalesmanState3(SalesmanState2):
75     def __init__(self, parent=None, sk=None, d=None, weights=None):
76         State.__init__(self, parent, sk)
77         self.known = parent.known + d(parent.sk,sk) if parent != None else 0
78         edges = self.edges()
79         D, i = 0.0, 0
80         for (u,v) in weights:
81             if (u,v) not in edges:
82                 D += weights[u,v]
83                 i += 1
84             if i == len(weights) - len(self): break
85         self.score = self.known + D
86 
87     def edges(self):
88         if self.parent == None: return set([])
89         edges = self.parent.edges()
90         edges.add( (self.parent.sk, self.sk) )
91         return edges
92 
93 class TravelingSalesman3(TravelingSalesman2):
94     def __init__(self, G, d):
95         self.G = G
96         self.d = d
97         self.first_vertex = list(G.V)[0]
98         aux = sorted((d[k], k) for k in d)[:len(G.V)]
99         self.weights = dict((k, v) for (v, k) in aux)
100        self.X = SalesmanState3(sk=self.first_vertex, weights=self.weights)
101
102    def branch(self, s):
103        visited = s.visited()
104        for v in self.G.succs(s.sk):
105            if v not in visited:
106                yield SalesmanState3(s, v, self.d, self.weights)
```

test_salesman3.py

```
1 from travelingsalesman import TravelingSalesman3
2 from traveling_salesman_graph import G, d
3 
4 path = TravelingSalesman3(G, d).solve()
5 print "Ciclo", path, "con peso", sum(d(path[i],path[i+1]) for i in xrange(len(path)-1))
```

```
Ciclo [0, 1, 3, 4, 7, 9, 6, 8, 5, 2, 0] con peso 67
```

$optimistic_3$ es una cota mejor informada que $optimistic_2$. Para demostrar que, para todo estado s, $optimistic_3(s) \geq optimistic_2(s)$, basta con darse cuenta de que la suma de los k menores valores de un conjunto es menor o igual que la suma de los k menores valores de un subconjunto suyo. No dedicaremos más tiempo a estudiar esta cota, pues es manifiestamente mejorable.

9.5.6. Una cota más informada: compleción del ciclo con las aristas de mínimo coste que parten de vértices no visitados

Sea cual sea el ciclo óptimo que podamos formar ramificando un estado $(v_1, v_2, \ldots, v_k, ?)$, sabemos que el resto de vértices formará un camino que empieza en v_k, pasa por todo vértice diferente de los ya visitados (una sola vez) y finaliza en v_1. Todo par de vértices contiguos, u y v, debe ser un elemento de E. Podemos relajar el problema eliminando esta restricción, es decir, escogiendo la arista de menor coste que parte de cada uno de los vértices no visitados y la de menor coste que parte del último vértice visitado.

$$optimistic_4((v_0, v_1, \ldots, v_k, ?)) = \sum_{0 \leq i < k} d(v_i, v_{i+1}) + \sum_{v \in V - \{v_0, v_1, \ldots, v_{k-1}\}} \min_{(v,w) \in E} d(v, w).$$

$optimistic_4$ es una cota inferior de f para cualquier estado, pues toda solución del problema original es una solución del problema relajado. Y es una cota mejor informada que $optimistic_3$, pues seleccionar una arista que parte de un estado hace que el resto de aristas que parten del mismo estado sean no elegibles, lo que no ocurría con $optimistic_3$.

Podemos precalcular el valor de la arista de menor peso que parte de cada vértice:

$$m_v = \min_{(v,w) \in E} d(v, w).$$

Esta cota es monótona no decreciente en la dirección de la raíz a las hojas, su valor coincide con la función objetivo en las soluciones factibles y su cómputo es incremental y realizable en tiempo $O(1)$:

$$optimistic_4((v_1, v_2, \ldots, v_k, v_{k+1}, ?)) = optimistic((v_1, v_2, \ldots, v_k, ?)) + d(v_k, v_{k+1}) - m_k.$$

travelingsalesman.py *(cont.)*

```
108 class SalesmanState4(SalesmanState2):
109     def __init__(self, parent=None, sk=None, d=None, best_departure=None):
110         State.__init__(self, parent, sk)
111         self.known = parent.known + d(parent.sk,sk) if parent != None else 0
112         self.score = self.known - best_departure[sk] if parent != None else sum(best_departure.values())
113 
114 class TravelingSalesman4(TravelingSalesman3):
115     def __init__(self, G, d):
116         self.G = G
117         self.d = d
118         self.first_vertex = list(G.V)[0]
119         self.best_departure = dict((u, min(d(u,v) for v in G.succs(u))) for u in G.V)
120         self.X = SalesmanState4(sk=self.first_vertex, best_departure=self.best_departure)
121 
122     def branch(self, s):
123         visited = s.visited()
```

```
124         for v in self.G.succs(s.sk):
125             if v not in visited:
126                 yield SalesmanState4(s, v, self.d, self.best_departure)
```

test_salesman4.py

```
1 from travelingsalesman import TravelingSalesman4
2 from traveling_salesman_graph import G, d
3
4 path = TravelingSalesman4(G, d).solve()
5 print "Ciclo", path, "con peso", sum(d(path[i],path[i+1]) for i in xrange(len(path)-1))
```

```
Ciclo [0, 1, 3, 4, 7, 9, 6, 8, 5, 2, 0] con peso 67
```

9.5.7. Cota basada en la compleción del ciclo con las aristas de mínimo coste que parten de y llegan a vértices no visitados

Podemos refinar la cota anterior seleccionando las aristas de peso mínimo que parten de y llegan a vértices no visitados, exceptuando los vértices inicial (que puede ser punto de llegada) y último del estado (que puede ser punto de partida).

$$optimistic_5((v_0, v_1, \ldots, v_k, ?)) = \sum_{0 \leq i < k} d(v_i, v_{i+1}) + \sum_{v \in V - \{v_0, v_1, \ldots, v_{k-1}\}} \min_{\substack{(v,w) \in E: \\ w \in V - \{v_0, v_1, \ldots, v_{k-1}\}}} d(v, w).$$

Ciertamente, la función de puntuación estará mejor informada que $optimistic_4$ y, por tanto, la poda será más efectiva. No obstante, esta ventaja puede quedar mitigada por el mayor coste de actualización de la cota inferior: por una parte no es posible efectuar un cálculo incremental y, por otra, ni siquiera podemos precalcular el valor de la mejor arista que parte de cada uno de los vértices no visitados. Tener que calcula la cota "de cero" para cada estado supone un coste temporal $O(|E|)$ por estado generado.

travelingsalesman.py *(cont.)*

```
128 from utils import min
129
130 class SalesmanState5(SalesmanState2):
131     def __init__(self, parent=None, sk=None, first_vertex=None, G=None, d=None):
132         State.__init__(self, parent, sk)
133         self.known = parent.known + d(parent.sk,sk) if parent != None else 0
134         unvisited = set(G.V) - self.visited()
135         departures = unvisited.union([sk])
136         arrivals = unvisited.union([first_vertex])
137         unknown = sum(min((d(u,v) for v in G.succs(u) if v in arrivals), ifempty=infinity) for u in departure
138         self.score = self.known + unknown
139
140 class TravelingSalesman5(TravelingSalesman4):
141     def __init__(self, G, d):
```

```
142         self.G = G
143         self.d = d
144         self.first_vertex = list(G.V)[0]
145         self.X = SalesmanState5(sk=self.first_vertex, G=G, d=d)
146 
147     def branch(self, s):
148         visited = s.visited()
149         for v in self.G.succs(s.sk):
150             if v not in visited:
151                 yield SalesmanState5(s, v, self.first_vertex, self.G, self.d)
```

test_salesman5.py

```
1 from travelingsalesman import TravelingSalesman5
2 from traveling_salesman_graph import G, d
3 
4 path = TravelingSalesman5(G, d).solve()
5 print "Ciclo", path, "con peso", sum(d(path[i],path[i+1]) for i in xrange(len(path)-1))
```

```
Ciclo [0, 2, 5, 8, 6, 9, 7, 4, 3, 1, 0] con peso 67
```

9.5.8. Dos cotas basadas en el coste del camino más corto que cierra el ciclo

Hemos relajado el problema seleccionando las $|V| - k$ aristas de menor peso de vértices no visitados. Alternativamente, podemos relajar el problema eliminando la restricción de que se pase por todos los vértices no visitados, pero manteniendo la obligación de escoger las aristas que conforman un camino que cierre el ciclo: el camino más corto desde el último vértice visitado y el vértice de partida. Si $D(u, v)$ es la distancia del camino más corto de u a v en G tenemos:

$$optimistic_6((v_0, v_1, \ldots, v_k, ?)) = \sum_{0 \leq i < k} d(v_i, v_{i+1}) + D(v_k, v_0).$$

Podemos calcular $D(v_0, u)$ para todo u en el digrafo invertido (que coincide con el grafo original si el grafo no está dirigido) para conocer el valor de $D(u, v_0)$ en el digrafo original. El cálculo de $D(v_0, u)$, para todo u, puede realizarse en una primera fase con el algoritmo de Dijkstra en tiempo $O(|E| + |V| \lg |V|)$.

Una vez efectuado el preproceso y almacenada en una tabla la distancia de v a u para todo v, actualizar la puntuación de un estado a partir de la puntuación de su estado padre es una operación ejecutable en tiempo constante:

$$optimistic_6((v_0, v_1, \ldots, v_k, ?)) = optimistic_6((v_0, v_1, \ldots, v_{k-1}, ?)) + d(v_{k-1}, v_k) - D(v_{k-1}, s) + D(v_k, s).$$

travelingsalesman.py *(cont.)*

```
153 from shortestpaths import all_shortest_paths_from_source
154 from graph import Digraph, Graph, WeightingFunction
155
156 class SalesmanState6(SalesmanState2):
157     def __init__(self, parent=None, sk=None, d=None, D=None):
158         State.__init__(self, parent, sk)
159         self.known = parent.known + d(parent.sk,sk) if parent != None else 0
160         self.score = self.known + (D[parent.sk] - D[sk] if parent != None else 0)
161
162 class TravelingSalesman6(TravelingSalesman4):
163     def __init__(self, G, d):
164         self.G = G
165         self.d = d
166         self.first_vertex = list(G.V)[0]
167         Ginv = Digraph(E=[(u,v) for (v,u) in d])
168         dinv = WeightingFunction(((u,v),d(u,v)) for (v,u) in d)
169         aux, self.D = all_shortest_paths_from_source(G, d, self.first_vertex, infinity=infinity)
170         self.X = SalesmanState6(sk=self.first_vertex, d=d, D=self.D)
171
172     def branch(self, s):
173         visited = s.visited()
174         for v in self.G.succs(s.sk):
175             if v not in visited:
176                 yield SalesmanState6(s, v, self.d, self.D)
```

test_salesman6.py

```
1 from travelingsalesman import TravelingSalesman6
2 from traveling_salesman_graph import G, d
3
4 path = TravelingSalesman6(G, d).solve()
5 print "Ciclo", path, "con peso", sum(d(path[i],path[i+1]) for i in xrange(len(path)-1))
```

```
Ciclo [0, 1, 3, 4, 7, 9, 6, 8, 5, 2, 0] con peso 67
```

Esta aproximación puntúa del mismo modo todo prefijo finalizado en el mismo vértice, lo que no resulta conveniente. Si modificamos el algoritmo de Dijkstra para que no tenga en cuenta los vértices ya visitados a costa, eso sí, de un elevado coste computacional en el cálculo de cada cota optimista: la generación de un hijo requiere ejecutar el algoritmo de Dijkstra sobre un subgrafo de G que se obtiene eliminando todos los vértices del camino visitado excepto el primero y el último (y sus correspondientes aristas). Sea $D(V', u, v)$ la distancia del camino más corto de u a v en el grafo de G inducido por el subconjunto de vértices $V' \subseteq V$:

$$optimistic_7((v_0, v_1, \ldots, v_k, ?)) = \sum_{0 \leq i < k} d(v_i, v_{i+1}) + D(V - \{v_0, \ldots, v_k\}, v_k, v_0).$$

Para evitar construir explícitamente un grafo para cada estado, hemos preferido ofrecer una versión del algoritmo de Dijkstra que evita tocar aquellos vértices que ya forman parte del ciclo:

travelingsalesman.py *(cont.)*

```
178 from prioritydict import MinPriorityDict
179
180 def distance_avoiding_vertices(G, d, forbidden, s, t):
181     D = dict((v, infinity) for v in G.V)
182     D[s] = 0
183     frontier = MinPriorityDict(D, len(G.V))
184     added = set(forbidden)
185     while len(frontier) > 0:
186         v, dmin = frontier.extract_min()
187         added.add(v)
188         if v == t: break
189         for w in G.succs(v):
190             if w not in added and D[v] + d(v,w) < D[w]:
191                 frontier[w] = D[w] = D[v] + d(v,w)
192     return D.get(t, infinity)
193
194 class SalesmanState7(SalesmanState2):
195     def __init__(self, parent=None, sk=None, first_vertex=None, G=None, d=None):
196         State.__init__(self, parent, sk)
197         self.known = parent.known + d(parent.sk,sk) if parent != None else 0
198         forbidden = self.visited() - set([first_vertex, sk])
199         distance = distance_avoiding_vertices(G, d, forbidden, sk, first_vertex)
200         self.score = self.known + distance
201
202 class TravelingSalesman7(TravelingSalesman4):
203     def __init__(self, G, d):
204         self.G = G
205         self.d = d
206         self.first_vertex = list(G.V)[0]
207         self.X = SalesmanState7(sk=self.first_vertex, first_vertex=self.first_vertex, G=G, d=d)
208
209     def branch(self, s):
210         visited = s.visited()
211         for v in self.G.succs(s.sk):
212             if v not in visited:
213                 yield SalesmanState7(s, v, self.first_vertex, self.G, self.d)
```

test_salesman7.py

```
1 from travelingsalesman import TravelingSalesman7
2 from traveling_salesman_graph import G, d
3
4 path = TravelingSalesman7(G, d).solve()
5 print "Ciclo", path, "con peso", sum(d(path[i],path[i+1]) for i in xrange(len(path)-1))
```

```
Ciclo [0, 2, 5, 8, 6, 9, 7, 4, 3, 1, 0] con peso 67
```

9.5.9. Una cota diferente: el árbol de recubrimiento de coste mínimo

Hay una relajación del problema que conduce a una aproximación diferente. Todo camino que visita todos los vértices una sola vez es un árbol de recubrimiento (si bien calificable de "degenerado" en tanto que todos los nodos internos tienen factor de ramaje uno). Podemos obtener una cota inferior para el coste del resto del camino si calculamos el árbol de expansión mínimo para el conjunto de vértices no visitados.

Nuevamente, ofrecemos un versión del algoritmo de Prim para calcular el peso del MST óptimo que evita un conjunto de vértices:

travelingsalesman.py *(cont.)*

```
215 from prioritydict import MinPriorityDict
216
217 def MST_weight_avoiding_vertices(G, d, forbidden, u):
218     weight = 0
219     added = set(forbidden)
220     in_frontier_from = MinPriorityDict(dict((v,(d(u,v),u)) for v in G.succs(u) if v not in added))
221     while len(in_frontier_from) > 0:
222         (v, (dv, u)) = in_frontier_from.extract_min()
223         added.add(v)
224         weight += d(u,v)
225         for w in G.succs(v):
226             if w not in added and \
227                     (w not in in_frontier_from or d(v, w) < d(in_frontier_from[w][1], w)):
228                 in_frontier_from[w] = (d(v,w), v)
229     return weight
230
231 class SalesmanState8(SalesmanState2):
232     def __init__(self, parent=None, sk=None, first_vertex=None, G=None, d=None):
233         State.__init__(self, parent, sk)
234         self.known = parent.known + d(parent.sk,sk) if parent != None else 0
235         forbidden = self.visited() - set([first_vertex, sk])
236         distance = MST_weight_avoiding_vertices(G, d, forbidden, sk)
237         self.score = self.known + distance
238
239 class TravelingSalesman8(TravelingSalesman4):
240     def __init__(self, G, d):
241         self.G = G
242         self.d = d
243         self.first_vertex = list(G.V)[0]
244         self.X = SalesmanState8(sk=self.first_vertex, first_vertex=self.first_vertex, G=G, d=d)
245
246     def branch(self, s):
247         visited = s.visited()
```

```
248         for v in self.G.succs(s.sk):
249             if v not in visited:
250                 yield SalesmanState8(s, v, self.first_vertex, self.G, self.d)
```

test_salesman8.py

```
1 from travelingsalesman import TravelingSalesman8
2 from traveling_salesman_graph import G, d
3
4 path = TravelingSalesman8(G, d).solve()
5 print "Ciclo", path, "con peso", sum(d(path[i],path[i+1]) for i in xrange(len(path)-1))
```

```
Ciclo [0, 2, 5, 8, 6, 9, 7, 4, 3, 1, 0] con peso 67
```

Nótese que todos lo árboles de recubrimiento para el mismo conjunto de vértices no visitados es el mismo. Se puede estar tentado, pues, de almacenar el peso de dicho árbol en un tabla indexada por los vértices no visitados. Es tabla podría tener una cantidad de celdas exponencialmente creciente con $|V|$.

EJERCICIOS

9-10 Define los estados como registros con la información necesaria para abaratar al máximo el coste temporal y espacial del algoritmo diseñado.

9-11 Podemos considerar equivalentes a aquellos estados que han visitado ya los mismos vértices: de todos ellos, Sólo interesa el estado con menor valor de la cota. Modifica el algoritmo para que explote esta propiedad. ¿Qué ventajas e inconvenientes presenta la técnica?

9-12 Realiza un estudio experimental que compare las diferentes aproximaciones al problema del viajante de comercio que hemos presentado. El estudio debe considerar tanto los estados generados, los estados insertados y extraídos de A y el tamaño máximo de A como el coste temporal promedio de cada iteración de los programas.

9-13 Diseña un algoritmo de ramificación y poda que solucione el problema del viajante de comercio y que use un cota optimista basada en el hecho de que

$$\frac{1}{2}\sum_{v\in V}\min_{(u,v)\in E} d(u,v) + \min_{(v,w)\in E} d(v,w)$$

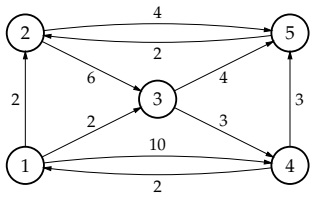
es menor o igual que el peso de cualquier ciclo hamiltoniano.

¿Qué ventajas e inconvenientes encuentras a esta cota?

9-14 Sea $G = (V, E)$ un grafo dirigido ponderado por una función $p : E \to \mathbb{R}^+$, y sean s y t dos vértices del grafo. Hemos diseñado un algoritmo de ramificación y poda que encuentra el camino simple (no contiene vértices repetidos) de *mayor* peso que va desde s a t. El algoritmo emplea una función cota optimista que se basa en que al ir de un vértice v a otro cualquiera w visitando únicamente vértices de un conjunto $V' \subset V$ tal que $v, w \notin V'$, el camino simple de mayor peso que va desde v a w es siempre de longitud no mayor que

$$\sum_{u\in V'\cup\{w\}} \max_{u'\in V'\cup\{w\}} p(u,u')$$

Suponiendo que el algoritmo emplea la estrategia de primero el mejor, desarrolla una traza para la instancia que se da en el siguiente grafo al buscar el camino simple de mayor peso que va del vértice 1 al 5.



9-15 Deseamos encontrar un camino euleriano de mínimo coste en un grafo $G = (V, E)$ dirigido, ponderado por la función $d : E \to \mathbb{R}^+$. Recuerda que un camino euleriano es aquel que visita todas las aristas del grafo al menos una vez.

a) Desarrolla un algoritmo de ramificación y poda.

b) En caso de que exista solución, ¿cómo solucionarías el problema del espacio de búsqueda infinito que podría plantearse para grafos con ciclos, en relación con la estrategia de búsqueda utilizada?

c) En caso de que no exista solución, ¿cómo actuaría el algoritmo en caso de que existan ciclos y de que no existan ciclos?

9-16 Diseña un algoritmo de ramificación y poda que, dado un grafo no dirigido $G = (V, E)$ ponderado por una función $d : E \to \mathbb{R}$, encuentre una partición de V en dos subconjuntos V_1 y V_2 tales que $V = V_1 \cup V_2$ y $V_1 \cap V_2 = \emptyset$ de modo que $\sum_{v \in V_1, w \in V_2} d(v, w)$ sea máxima.

9-17 En la sección **??** estudiamos el problema del viajante de comercio euclídeo. Diseña un algoritmo de ramificación y poda que haga uso de él y que explote su solución voraz de dos modos:

- para obtener un solución inicial que permita anticipar la poda
- y para obtener un cota optimista en la compleción del ciclo.

9.6. El camino más corto en un grafo euclídeo

Un grafo euclídeo es un grafo dirigido $G = (V, E)$ y ponderado cuyos vértices son puntos de $\mathbb{R}^n$ (por simplicidad supondremos que $n = 2$) y en el que el peso de la arista que une dos vértices u y v es su distancia euclídea, $d(u, v)$. Deseamos encontrar el camino más corto de un vértice s a otro t.

Ya hemos resuelto el problema de diferentes modos: si el grafo es acíclico, podemos encontrar el camino óptimo en tiempo $O(|E|)$, y si contiene ciclos, el algoritmo de Dijkstra efectúa el cálculo en $O(|E| \log |V|)$ pasos. Desarrollaremos un algoritmo de ramificación y poda que puede considerarse una extensión del algoritmo de Dijkstra y que conduce a un algoritmo muy eficiente en la práctica. Conviene recordar que en $\mathbb{R}^n$ se satisface la desigualdad triangular: dados tres puntos, a, b y c, se cumple $d(a, c) \leq d(a, b) + d(b, c)$. Haremos uso de esta propiedad para diseñar una cota optimista.

9.6.1. Estados, función objetivo y ramificación

Buscamos un camino, es decir, una secuencia de vértices $(v_1, v_2, \ldots, v_m)$. El primer vértice de la secuencia debe ser u y el último debe ser t. Además, todo par de vértices consecutivos en el camino debe ser un elemento de E. El conjunto X es el conjunto de todos los caminos entre u y t. Si el grafo es acíclico, X es de talla finita; en caso contrario, la talla es infinita, pero podemos limitarnos a considerar los caminos con no más de $|V|$ vértices.

Un estado unitario se representará con la secuencia de vértices que forman su único camino. Un estado con más de un camino se representará con una secuencia de vértices que determina el prefijo común de todos los caminos que describe el estado y una marca final «?», que indica la incompletitud del camino descrito. El estado inicial, que describe la integridad del espacio de búsqueda, es $X = \{(s, ?)\}$.

La función objetivo es

$$f((v_1, v_2, \ldots, v_m)) = \sum_{1 \leq i < m} d(v_i, v_{i+1}).$$

La función de ramificación toma un estado y devuelve un conjunto de estados

$$\mathit{branch}((v_1, v_2, \ldots, v_k, ?)) = \{(v_1, v_2, \ldots, v_k, v_{k+1}, ?) \mid (v_k, v_{k+1}) \in E\}.$$

Nótese que cualquier secuencia de vértices cuyo último vértice sea t es un estado unitario: aunque t tenga sucesores, no tiene sentido ramificar extendiendo la secuencia de vértices, pues cualquier camino que formemos de ese modo recorrerá una distancia igual o mayor que el camino que finaliza en t, cuando estamos interesados en un camino de mínima distancia.

9.6.2. Un primer algoritmo basado en una cota inferior trivial

Nuestro objetivo ahora es diseñar una función que sea cota inferior del valor de la función objetivo sobre el camino más corto entre u y t. Utilizaremos la función tanto para tener una cota optimista de la función objetivo sobre un conjunto de soluciones, como para puntuar los estados para su selección por primero el mejor.

Hay una cota optimista trivial formada por la suma del peso de las aristas del prefijo común de todos los caminos de un estado:

$$\mathit{optimistic}_1((v_1, v_2, \ldots, v_k, ?)) = \sum_{1 \leq i < k} d(v_i, v_{i+1}).$$

Esta función renuncia a efectuar una estimación de la distancia más corta con que podemos llegar del último vértice de un camino incompleto a t. Es una cota inferior de la mejor forma de completar el camino, pues cualquier prolongación de un camino supone recorrer una distancia mayor o igual que cero, pero no está muy informada: sólo cuenta con la información sobre la parte conocida de los caminos de un estado.

El algoritmo de ramificación y poda que presentamos a continuación soluciona el problema con una búsqueda por primero el mejor basada en la cota inferior recién presentada. Hemos incluido en el programa algunas variables que cuentan el máximo número

de estados activos en A, el número de extracciones del conjunto de estados activos y el número de inserciones con objeto de estimar la eficiencia de nuestra solución en comparación a los refinamientos que iremos proponiendo.

euclidean.py

```
from bbscheme import State, BranchAndBoundScheme
from heap import MinHeap
from utils import infinity

class EuclideanState(State):
    def __init__(self, parent=None, sk=None, d=None):
        State.__init__(self, parent, sk)
        self.score = parent.score + d(parent.sk, sk) if parent != None else 0

class EuclideanShortestPathMixin:
    def __init__(self, G, d, s, t):
        self.G = G
        self.d = d
        self.s = s
        self.t = t
        self.X = EuclideanState(sk=self.s, d=d)

    def is_complete(self, s):
        return s.sk == self.t

    def is_factible(self, s):
        return True

    def worst_value(self):
        return infinity

    def branch(self, s):
        for v in self.G.succs(s.sk):
            yield EuclideanState(s, v, self.d)

    def f(self, x):
        return x.score

    def optimistic(self, s):
        return s.score

    def opt(self, a, b):
        return min(a, b)

    def priority_queue(self, iterable=[]):
        return MinHeap(iterable)

class EuclideanShortestPath(EuclideanShortestPathMixin, BranchAndBoundScheme):
    pass
```

test_euclidean.py

```
1 from euclidean import EuclideanShortestPath
2 from iberia import iberia, km
3
4 path = EuclideanShortestPath(iberia, km, 'Ciudad Real', 'Soria').solve().list()
5 print 'Camino', ', '.join(path), 'con distancia', sum(km(path[i],path[i+1]) for i in xrange(len(path)-1))
```

```
Camino Ciudad Real, Toledo, Madrid, Venturada, Riaza, San Esteban de Gormaz, B
urgo de Osma, Soria con distancia 325.168
```

Tiene interés conocer un par de datos sobre la eficiencia del algoritmo: el número de iteraciones efectuadas para resolver esta instancia concreta del problema es de 125144 y el número de estados generados es de 482259. Son números enormes. ¿Por qué se seleccionan y generan tantos estados? Muy sencillo: el procedimiento genera *todo* camino con distancia menor o igual que la distancia del camino más corto entre u y t (y algunos más). Se puede ir del vértice origen a cualquiera de los alcanzados por muchos caminos diferentes con distancia inferior a la que hay entre u y t. ¿Cómo evitarlo?

En la figura 9.10 se muestra el resultado del cálculo que hemos realizado al probar el programa sobre el grafo euclídeo del mapa **iberia**. Hemos marcado con un círculo los vértices alcanzados en el proceso de búsqueda, es decir, tales que ha sido considerado algún camino cuyo último vértice es él. Los vértices con doble círculo son aquellos para los que algún camino cuyo último vértice es él se ha ramificado, o sea, se consideró en algún instante «el camino más prometedor». Puede observarse cómo la búsqueda se expande alrededor del vértice inicial y visita todos los vértices que se encuentra a distancia suya igual o superior que el vértice destino.

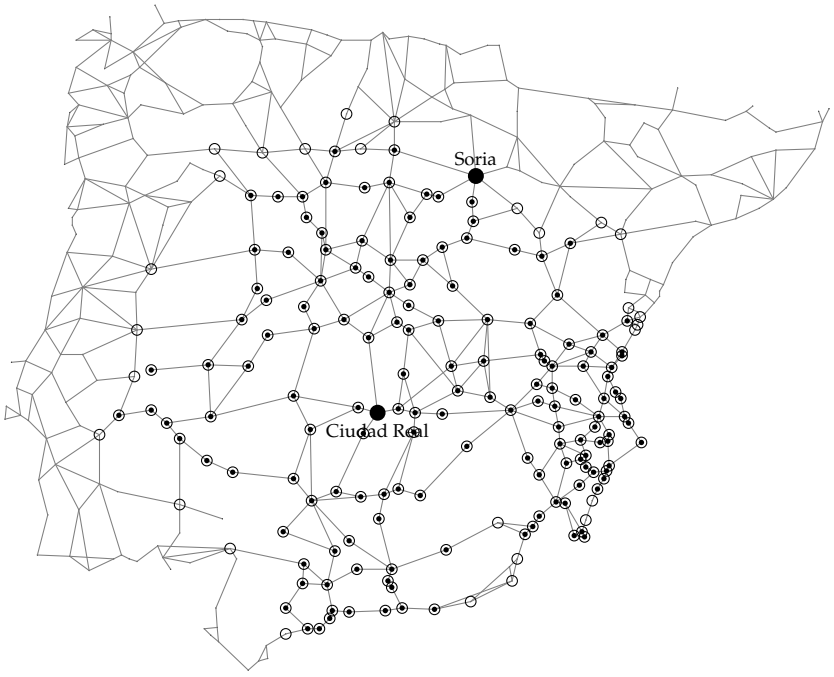


Figura 9.10: Resultado del cálculo del camino más corto entre Ciudad Real y Soria usando una cota optimista que sólo aprovecha información de la parte conocida de un estado. Las ciudades marcadas con un punto grueso y rodeadas con un círculo son vértices finales de estados ramificados durante la búsqueda. Las ciudades con un punto fino y rodeadas con un círculo han sido «alcanzadas» por algún estado, pero no ha sido preciso ramificarlas.

EJERCICIOS

9-18 Modifica el proceso de ramificación para que no se generen ciclos. ¿Resulta eficiente el nuevo proceso de ramificación si lo comparamos con el original?

9.6.3. Un refinamiento: memorización de la mejor forma de llegar a cada vértice

El algoritmo recuerda mucho al que Dijkstra propuso para resolver el mismo problema, aunque no es idéntico a éste. El método de ramificación y poda puede considerar prometedores dos caminos que finalizan en un mismo vértice, cuando en realidad sólo debería considerar que uno de ellos es prometedor: el más corto de ambos. Si incorporamos este criterio a nuestro método, habremos eliminado el problema de los ciclos: cuando se ramifica un camino y se acaba llegando a un vértice ya visitado, formando un bucle, es imposible mejorar el valor memorizado para ese vértice, con lo que el nuevo estado se descarta. Pero no sólo hemos logrado la reducción de esfuerzo derivada de la poda de estados con ciclos: esta técnica también elimina caminos que llegan a un vértice por un trayecto algo más largo que mediante un trayecto alternativo conocido. Esta técnica de memorización encuentra aplicación en este problema, aunque el algoritmo de Dijkstra no se considere un algoritmo de programación dinámica.

He aquí el nuevo algoritmo, que incorpora, además, una implementación compacta de los estados con un árbol de prefijos:

euclidean.py *(cont.)*

```
47 from bbscheme import State, BranchAndBoundWithMemoizationScheme
48 from heap import MinHeap
49 from utils import infinity
50
51 class EuclideanShortestPathMixin2(EuclideanShortestPathMixin):
52     def g(self, s):
53         return s.score
54
55     def equivalence_id(self, s):
56         return s.sk
57
58 class EuclideanShortestPath2(EuclideanShortestPathMixin2,BranchAndBoundWithMemoizationScheme):
59     pass
```

test_euclidean2.py

```
1 from euclidean import EuclideanShortestPath2
2 from iberia import iberia, km
3
4 path = EuclideanShortestPath2(iberia, km, 'Ciudad Real', 'Soria').solve().list()
5 print 'Camino', ', '.join(path), 'con distancia', sum(km(path[i],path[i+1]) for i in xrange(len(path)-
```

```
Camino Ciudad Real, Toledo, Madrid, Venturada, Riaza, San Esteban de Gormaz, B
urgo de Osma, Soria con distancia 325.168
```

Con este algoritmo, el número de iteraciones es de 166 y se generan 548 estados. La reducción de coste es espectacular. El algoritmo presentado es, en el fondo, una versión del algoritmo de Dijkstra: partimos del vértice inicial y vamos alcanzamos sus sucesores. Cada vértice se puntúa con el peso del arco correspondiente. A continuación escogemos el vértice mejor puntuado y alcanzamos a cada uno de sus sucesores, asignándole la puntuación que se obtiene de añadir a su puntuación original el peso de la arista con el que lo alcanzamos. Si un vértice ya había sido puntuado, sólo se modificaba la puntuación si el nuevo valor calculado era mejor. El proceso finaliza cuando se escoge, para su ramificación, el vértice final del camino.

9.6.4. Una cota inferior más informada: adición de la distancia del último vértice al destino

Todo camino formado añadiendo vértices al camino $(v_1, v_2, \ldots, v_k)$ tendrá una longitud igual o superior a la de éste, pues su longitud es la del prefijo más una cantidad mayor o igual que cero:

$$\mathit{optimistic}_1((v_1, v_2, \ldots, v_k, ?)) = \sum_{1 \leq i < k} d(v_i, v_{i+1}) \leq \min_{(v_1, v_2, \ldots, v_k, v_{k+1}, \ldots, v_m) \in X} \sum_{1 \leq i < m} d(v_i, v_{i+1}).$$

Es posible efectuar una estimación más informada de la mejor forma de completar un camino: la distancia euclídea entre el último vértice de un estado y t es siempre menor o igual que la longitud recorrida por cualquier trayecto entre dichos puntos. Por tanto, la función

$$\mathit{optimistic}_2((v_1, v_2, \ldots, v_k, ?)) = \left(\sum_{1 \leq i < k} d(v_i, v_{i+1}) \right) + d(v_k, t)$$

es también una cota inferior de la distancia del camino más corto entre u y t que empieza visitando los vértices v_1, v_2, …, v_k. Nótese que no tiene por qué haber carretera entre v_k y t. La desigualdad triangular es suficiente garantía para que esta función sea cota optimista.

Esta función cuenta con estas propiedades destacables:

- Se puede calcular incrementalmente:

  $$\mathit{optimistic}_2((v_1, v_2, \ldots, v_k, ?)) = \mathit{optimistic}_2((v_1, v_2, \ldots, v_{k-1}, ?)) - d(v_{k-1}, t) + d(v_{k-1}, v_k) + d(v_k, t).$$

  Calcular el valor de $\mathit{optimistic}_2$ para cada descendiente de un estado es una operación que requiere tiempo $O(1)$.

- Es monótona no decreciente en la dirección de las hojas. Por la desigualdad triangular sabemos que $d(v_{k-1}, t) \leq d(v_{k-1}, v_k) + d(v_k, t)$, así que $d(v_{k-1}, v_k) + d(v_k, t) - d(v_{k-1}, t)$ es una cantidad no negativa. Las estimaciones optimistas mejoran (son más próximas al verdadero valor) conforme avanzamos de la raíz a las hojas y la puntuación del mejor elemento de A no decrece el progresar el cálculo.

- El valor de la función objetivo coincide con el de la única solución factible contenida en un estado unitario:

$$optimistic_2((v_1, v_2, \ldots, v_m)) = \left(\sum_{1 \leq i < m} d(v_i, v_{i+1})\right) + d(v_m, t) = \left(\sum_{1 \leq i < m} d(v_i, v_{i+1})\right) + 0 = f((v_1, v_2,$$

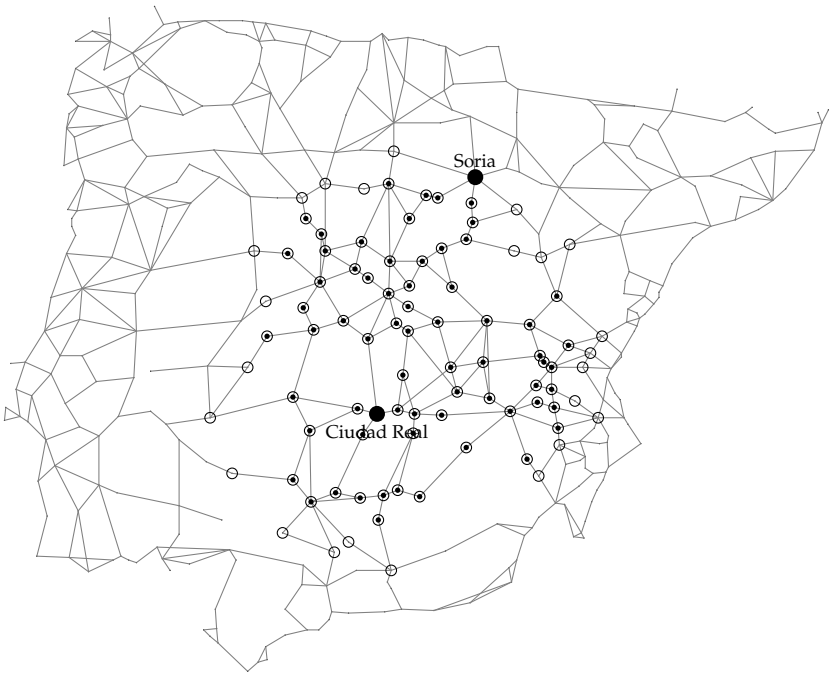


Figura 9.11: Resultado del cálculo del camino más corto entre Ciudad Real y Soria aplicando el algoritmo de ramificación y poda con la cota informada basada en la desigualdad triangular.

euclidean.py *(cont.)*

```
61 from bbscheme import State, BranchAndBoundWithMemoizationScheme
62 from heap import MinHeap
63 from utils import infinity
64
65 class EuclideanState3(State):
66     def __init__(self, parent=None, sk=None, d=None, straight=None):
67         State.__init__(self, parent, sk)
68         self.known = parent.known + d(parent.sk, sk) if parent != None else 0
69         self.score = self.known + straight[sk] if parent != None else straight[sk]
70
71 class EuclideanShortestPathMixin3(EuclideanShortestPathMixin):
72     def __init__(self, G, d, s, t):
73         self.G = G
74         self.d = d
75         self.s = s
76         self.t = t
77         self.straight = dict((c, distance(coords[c], coords[t])) for c in coords)
78         self.X = EuclideanState3(sk=self.s, d=d, straight=self.straight)
```

```
79
80      def branch(self, s):
81          for v in self.G.succs(s.sk):
82              yield EuclideanState3(s, v, self.d, self.straight)
83
84      def g(self, s):
85          return s.known
86
87  class EuclideanShortestPath3(EuclideanShortestPathMixin2,BranchAndBoundWithMemoizationScheme):
88      pass
```

test_euclidean3.py

```
1  from euclidean import EuclideanShortestPath3
2  from iberia import iberia, km
3
4  path = EuclideanShortestPath3(iberia, km, 'Ciudad Real', 'Soria').solve().list()
5  print 'Camino', ', '.join(path), 'con distancia', sum(km(path[i],path[i+1]) for i in xrange(len(path)-1))
```

```
Camino Ciudad Real, Toledo, Madrid, Venturada, Riaza, San Esteban de Gormaz, B
urgo de Osma, Soria con distancia 325.168
```

El tamaño máximo de A pasa de 70 a sólo 19.

EJERCICIOS

9-19 Disponemos de una tabla con la distancia kilométrica por carretera entre cualquier par de capitales en el mapa, pero no de todas las ciudades. ¿Puedes utilizar esta información para acelerar de algún modo el cálculo en el algoritmo de ramificación y poda? ¿Cómo? ¿Y si dispones de la distancia por carretera entre cualquier par de ciudades? ¿Puedes acotar el coste computacional de un algoritmo que haga uso de esa información?

9-20 Nos dan un grafo en el que cada vértice describe la posición de una ciudad con su latitud y longitud. Cada posición es un par (a, b) donde a es la latitud y b la longitud (en radianes, es decir, no en grados y segundos y usando valores negativos de la longitud para posiciones del hemisferio sur). La «distancia de gran círculo» permite calcular la distancia entre dos ciudades teniendo en cuenta la curvatura de la Tierra, aunque considerando que es una esfera perfecta. Dada la posición de dos ciudades, (a_1, b_1) y (a_2, b_2), su distancia de gran círculo es

$$r \arccos(\cos(a_1)\cos(b_1)\cos(a_2)\cos(b_2) + \cos(a_1)\sin(b_1)\cos(a_2)\sin(b_2) + \sin(a_1)\sin(a_2)),$$

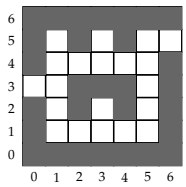
donde r es el radio de la Tierra, que (suponiendo que sea una esfera perfecta) es de aproximadamente 3 963.1 kilómetros.

¿Es utilizable el algoritmo de ramificación y poda anterior (con la pertinente modificación de la función de cálculo de distancia entre un par de ciudades)?

9-21 Sea $G = (V, E)$ un grafo multietapa con K etapas ponderado por una función $d : E \to \mathbb{R}^{\geq 0}$. Desarrolla un algoritmo de ramificación y poda que proporcione el camino de coste mínimo entre el nodo inicial y final. Calcula el coste temporal de la función cota que se haya propuesto y demostrar que, efectivamente, es una cota.

9-22 Deseamos encontrar el camino más corto en un laberinto que nos viene descrito mediante una matriz de $n \times m$ valores booleanos, la fila y columna de la entrada y la fila y columna de la salida. Una casilla es un muro si su celda correspondiente en la matriz es el valor «cierto», y es pasillo en caso contrario.

He aquí un ejemplo de laberinto con entrada en $(3,0)$ y salida en $(5,6)$ en el que hemos pintado de negro los muros.



El camino más corto es la secuencia de casillas $(3,0)$, $(3,1)$, $(4,1)$, $(4,2)$, $(4,3)$, $(4,4)$, $(4,5)$, $(5,5)$ y $(6,5)$.

Te pedimos:

a) Que diseñes un algoritmo de ramificación y poda que encuentre, para cualquier laberinto, el camino más corto entre su entrada y salida, si éste existe.

b) Que indiques el comportamiento del algoritmo cuando no existe un camino entre la entrada y la salida.

c) Que lo modifiques para que encuentre todos los caminos de longitud igual que la longitud del más corto.

d) Que compares la eficiencia del algoritmo con el algoritmo de Dijkstra aplicado al grafo y con una versión que encuentre la salida por programación dinámica.

9-23 Nuestra nave espacial no tripulada ha aterrizado en un punto de Marte al que denotamos con las coordenadas $(0,0)$. Disponemos de un mapa cartográfico que describe el área inscrita en el rectángulo de coordenadas $(-n,-m)$, $(-n,m)$, $(n,-m)$ y (n,m). Dado que el mapa se puede considerar una matriz de $(2m+1)\times(2n+1)$, denominaremos «celda» a cada uno de los cuadrados de un metro de lado que podemos identificar por sus coordenadas.

El Robot de Exploración Marciana (REM) ha de tomar muestras del planeta. REM puede cargar su batería en la nave espacial, lo que le dota de una cantidad de energía E. Queremos desplazar al robot desde la nave espacial hasta un punto (x_t,y_t) y regresar a la nave. El objetivo es tomar una muestra del terreno en ese punto, pero REM aprovecha el viaje y toma una muestra en cada celda que visita *por primera vez*.

Podemos controlar el robot con tres órdenes:

- `FORWARD`: Avanza un metro.
- `RIGHT`: Gira 90 grados a la derecha.
- `LEFT`: Gira 90 grados a la izquierda.

REM pesa cinco kilogramos y cada muestra de terreno pesa diez gramos. Notaremos con p el peso de REM en kilogramos. El consumo energético de REM depende de la carga que lleve:

- Cada vez que REM se desplaza una casilla hacia adelante consume $p/5$ unidades de energía.
- Cada vez que REM gira 90 grados, consume $p/10$ unidades de energía.
- Cada vez que REM asciende a metros, consume $ap/2$ unidades de energía (que se suman a las que consume por el hecho de desplazarse).

Diseña un algoritmo de ramificación y poda que encuentre el camino de ida y vuelta que menor energía consume. En cualquier caso, el camino no puede consumir más de E unidades de energía. Si un camino tal existe, el algoritmo nos lo debe devolver como secuencia de órdenes con la que programar el robot. (Ten en cuenta que el mejor camino de ida no tiene porqué ser el mejor camino de vuelta.)

9.7. El problema del empaquetado en cajas

El problema del empaquetado en cajas fue abordado en el apartado **??** del capítulo dedicado a los algoritmos voraces. Vimos que los algoritmos voraces no ofrecían garantía de encontrar la solución óptima, $\hat{x}$, pero uno de ellos devolvía una solución x cuyo valor de la función objetivo era menor o igual que $4 + \frac{11}{9} f(\hat{x})$. Usaremos este algoritmo aproximado para estimar una primera solución. El mismo algoritmo servirá para diseñar una cota optimista.

Recordemos el planteamiento del problema. Deseamos cargar un conjunto de N objetos, cada uno con un peso w_i, para $1 \leq i < n$, en el menor número posible de cajas, cada una de ellas con una capacidad de carga C.

Una solución puede expresarse con una n-tupla $(x_1, x_2, \ldots, x_N)$ cuyos elementos son números naturales indicando el contenedor en el que se carga cada objeto. Una solución factible debe respetar la restricción de que la suma de pesos cargados en un contenedor sea menor o igual que C. (Asumimos que $w_i \leq C$ para todo i.) Naturalmente, el máximo número de contenedores es N.

Por otra parte, supondremos que los valores del conjunto $\{x_1, x_2, \ldots, x_N\}$ forman el rango completo de enteros $[1..M]$, para algún valor de $M \geq 1$, pues no tiene sentido que haya contenedores sin objetos asignados. O sea, X es el conjunto de N-tuplas de enteros estrictamente positivos tales que:

- $\sum_{1 \leq i \leq n : x_i = j} w_i \leq C$, para todo $j \in \{x_1, x_2, \ldots, x_N\}$,
- y existe un $M \in \mathbb{N}^{>0}$ tal que $\{x_1, x_2, \ldots, x_N\} = [1..M]$.

La función objetivo, cuyo valor deseamos minimizar, es

$$f((x_1, x_2, \ldots, x_N)) = \max_{1 \leq i \leq n} x_i.$$

9.7.1. Estados y ramificación

Usaremos nuevamente una representación de los conjuntos de soluciones con prefijos comunes basada en dicho prefijo. El estado $(x_0, x_1, \ldots, x_{k-1}, ?)$ representa el conjunto de soluciones cuyos k primeros objetos ya han sido asignados a contenedores.

Al ramificar un estado $(x_0, x_1, \ldots, x_{k-1}, ?)$ hemos de decidir si el objeto de índice k ocupa uno de los contenedores que ya contienen objetos o si usa un nuevo contenedor:

$$branch((x_0, x_1, \ldots, x_{k-1}, ?)) = \left\{ (x_0, x_1, \ldots, x_{k-1}, x_k, ?) \mid x_k \in [0.. \max_{0 \leq i < k} x_k] \right\}.$$

El número de estados generados al ramificar otro está acotado por el número de contenedores ya usados.

No tiene sentido generar estados que no pueden contener soluciones factibles porque los elementos ya dispuestos en contenedores rebasan su capacidad. Basta con que

efectuemos una comprobación con el último objeto asignado a un contenedor:

$$branch((x_0, x_1, \ldots, x_{k-1}, ?)) = \left\{ (x_0, x_1, \ldots, x_{k-1}, x_k, ?) \mid x_k \in [0.. \max_{0 \le i < k} x_k]; \sum_{0 \le i < k : x_i = x_k} w_k \le C \right\}.$$

9.7.2. Una cota sencilla

Es posible calcular una cota inferior al número en cajas necesarias para empaquetar n objetos calcular la suma de todos los pesos, dividiendo el resultado por C y redondeándolo al alza. Esta cota se puede estimar para los últimos i objetos, con i variando entre 1 y N. El número de cajas necesarias para completar la solución $(x_1, x_2, \ldots, x_k, ?)$ es el máximo entre $\max_{1 \le i \le k} x_i$ y $\lceil (\sum_{k < i \le N} w_i)/C \rceil$.

bb_bin_packing.py

```
1  from heap import MinHeap
2  from greedy_bin_packing import first_fit_decreasing_bin_packing
3  from math import ceil
4  from offsetarray import OffsetArray
5  from bbstate import State
6
7  class Packing(State):
8    def weight_in_pack(self, i, w):
9      if self.sk != i: weight = 0
10     else: weight = w[self.len]
11     if self.parent: return self.parent.weight_in_pack(i, w) + weight
12     else: return weight
13
14 def bb_bin_packing(w, C):
15   bins = OffsetArray([ None ] * len(w))
16   bins[len(w)] = w[len(w)]
17   for i in reversed(xrange(1, len(w))): bins[i] = bins[i+1] + w[i]
18   for i in xrange(1, len(w)+1): bins[i] = int(ceil(bins[i]/float(C)))
19
20   greedysol = first_fit_decreasing_bin_packing(w, C)
21   x = Packing()
22   for i in xrange(1, len(w)+1): x = Packing(x, greedysol[i])
23   x.set(score=max(greedysol))
24
25   A = MinHeap( [Packing(Packing(), 1).set(M=1, score=bins[2])] )
26   while len(A) > 0 and A.min().score < x.score:
27     s = A.extract_min()
28     for sk in xrange(1, s.M+2):
29       if s.weight_in_pack(sk, w) + w[len(s)+1] > C: continue
30       s' = Packing(s, sk).set(M=max(s.M, sk))
31       if len(s') == len(w):
32         if s'.M < x.score:
33           x = s'.set(score=s'.M)
34       else:
```

```
35          s'.score = max(s'.M, bins[len(s')+1])
36          if s'.score < x.score:
37            A.insert( s' )
38      return OffsetArray(x.list())
```

test_bb_bin_packing.py

```
1  from greedy_bin_packing import first_fit_decreasing_bin_packing, show_solution
2  from bb_bin_packing import bb_bin_packing
3  from offsetarray import OffsetArray
4
5  w = OffsetArray([52, 18, 36, 21, 88, 15, 26, 29, 86, 7])
6  print 'Solución voraz:'
7  show_solution(first_fit_decreasing_bin_packing(w, 100), w)
8  print 'Solución óptima:'
9  show_solution(bb_bin_packing(w, 100), w)
```

```
Solución voraz:
 1: 88 7
 2: 86
 3: 52 36
 4: 18 21 26 29
 5: 15
Solución óptima:
 1: 52 18 29
 2: 36 21 15 26
 3: 88 7
 4: 86
```

EJERCICIOS

9-24 Cabe esperar un mejor comportamiento del algoritmo si se suministran los pesos ordenados de mayor a menor. Implementa una nueva versión del algoritmo que efectúe la ordenación y compara experimentalmente el número de estados generados, insertados y extraídos y el tamaño máximo de la cola de prioridad.

9-25 Trabajamos para una empresa textil. Cada rollo de tella mide m metros de ancho y n metros de largo. Nos llegan P pedidos de rectángulo de tela de dimensión $a_i \times b_i$, para i entre 1 y P.

Diseña un algoritmo de ramificación y poda que nos diga cómo cortar todos los rectángulos de modo que se consuma el menor número posible de rollos.

9.8. El problema de las múltiples mochilas

Disponemos de M mochilas en las que queremos cargar un conjunto de objetos. Hay N objetos, cada uno con un peso w_j y un valor v_j, para $1 \leq j \leq N$. Cada mochila soporta una carga máxima c_i, para $1 \leq i \leq M$. Deseamos conocer la selección de objetos que hemos de cargar en cada mochila de modo que el beneficio sea máximo y ninguna mochila vea superada su capacidad de carga.

9.8.1. Modelado

Una solución factible puede expresarse con una N tupla en el que la componente i-ésima es el índice de la mochila en el que se carga la caja, o el valor 0 en el caso de que el objeto no se cargue en mochila alguna. El espacio de soluciones factibles es

$$X = \left\{ (x_1, x_2, \ldots, x_N) \in [0..M]^N \;\middle|\; \sum_{1 \leq j \leq N : x_j = i} w_j \leq c_i, 1 \leq i \leq M \right\}.$$

Deseamos maximizar el valor total de la carga de las mochilas. Nuestra función objetivo es

$$f((x_1, x_2, \ldots, x_N)) = \sum_{1 \leq j \leq N : x_j \neq 0} v_j.$$

Buscamos

$$(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N) = \arg \max_{(x_1, x_2, \ldots, x_N) \in X} \sum_{1 \leq j \leq N : x_j \neq 0} v_j.$$

9.8.2. Ramificación y cota optimista

Las soluciones son secuencias formadas por N elementos. Los estados no unitarios serán, pues, secuencias con menos de N elementos, es decir, tuplas de la forma $(x_1, x_2, \ldots, x_k, ?)$ con $k < N$. La función de ramificación generará, a partir de un estado de la forma $(x_1, x_2, \ldots, x_k, ?)$, el conjunto de estados de la forma $(x_1, x_2, \ldots, x_k, x_{k+1}, ?)$ para x_{k+1} en $[0..M]$, siempre que la suma de pesos de objetos asignados a cada mochila no exceda de su capacidad.

Si mantenemos en cada estado la carga acumulada en cada mochila (o la capacidad de carga disponible), la ramificación es un proceso realizable en tiempo $O(M)$, es decir, $O(1)$ por estado.

Se puede diseñar una cota optimista (superior) suponiendo que las mochilas no tienen limitación alguna de carga. En tal caso, el máximo beneficio se obtiene cargando todos los objetos sobre los que no se ha adoptado decisión alguna. El beneficio obtenido así es mayor o igual que el que obtendremos si cargamos un subconjunto de los objetos.

$$\mathit{optimistic}((x_1, x_2, \ldots, x_k, ?)) = \sum_{1 \leq j \leq k : x_j \neq 0} v_j + \sum_{k+1 \leq j \leq N} v_j.$$

Esta cota puede calcularse en tiempo constante por estado si se realiza incrementalmente.

La cota puede ajustarse más si resolvemos el problema de la mochila continua sobre la capacidad de carga disponible en todas las mochilas.

EJERCICIOS

9-26 Implementa sendos programas de ramificación y poda inspirados en las dos cotas propuestas.

9.9. Valor extremo en un paseo aleatorio

Un **paseo aleatorio**

Paseo aletorio: Random walk.

> El estudio de paseos aleatorios presenta interés en física. Son modelos simplificados del denominado movimiento browniano, el movimiento aleatorio de las moléculas en líquidos y gases. También se los conoce en la literatura como «paseos de borracho».

es el resultado de avanzar paso a paso tomando una decisión arbitraria en cada momento. El paso aleatorio más sencillo se construye sobre una línea con estas reglas:

- se parte en la posición 0,
- la distancia dada en cada paso es constante,
- antes de dar cada paso se decide aleatoriamente si ir al norte o al sur.

La figura 9.12 muestra algunos paseos aleatorios con N pasos. La figura muestra la posición del paseante a lo largo del tiempo.

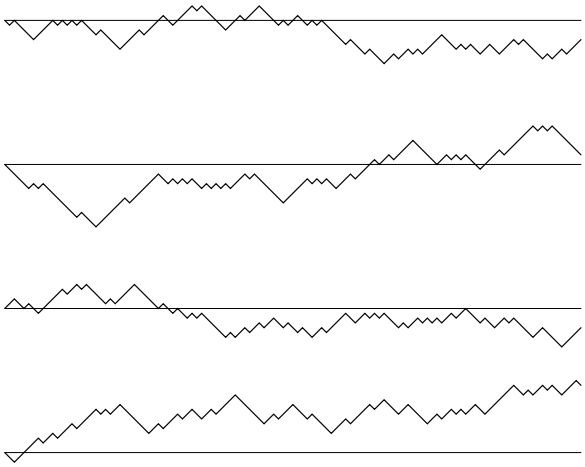


Figura 9.12: Cuatro paseos aleatorios. El eje horizontal es el tiempo.

Dado un paseo aletorio, deseamos conocer el punto más septentrional de cuantos visita el paseante. El paseo se proporciona con una secuencia de valores enteros, $w = (w_0, w_1, \ldots, w_{n-1})$. El primer valor es 0 (el punto de partida). Valores positivos indican posiciones al norte del punto de partida y valores negativos, posiciones al sur de dicho punto. El punto más septentrional es el máximo valor de la lista.

9.9.1. Modelado

Este problema presenta interés porque la representación de los estados no se forma con secuencias de valores, como hemos hecho hasta el momento.

En principio, cualquier punto del vector puede ser el punto más septentrional. Representaremos con el par (i, j) el conjunto de puntos con índices $\{i, i+1, i+2, \ldots, j-2, j-$

1}. El conjunto de todos los puntos es $(0, n)$. Un estado de la forma $(i, i+1)$ es un estado unitario.

Ramificaremos un estado (i, j) dividiéndolo en dos nuevos estados: $(i, (i+j)/2)$ e $((i+j)/2, j)$.

9.9.2. Una cota superior

¿Cómo acotar superiormente la posición más septentrional en un estado (i, j) sin recorrer el rango de puntos (que sería tanto como resolver el problema)? Es posible efectuar una estimación si conocemos w_i: podemos suponer que los $j - i$ pasos dados se dan en dirección al norte. El valor de $w_i + (j - i)$ es, pues, una cota superior. Podemos hacer lo mismo con w_j, pero suponiendo que los últimos $j - i$ pasos se dieron al sur. El valor de $w_i + (j - i)$ también es, pues, una cota superior. De ambas, nos interesa la menor:

$$\mathit{optimistic}((i,j)) = \min(w_i + (j-i), w_{j-1} + (j-i)) = \min(w_i, w_{j-1}) + (j-i).$$

La figura 9.13 ilustra la idea. A la vista de la figura, es evidente que la cota es excesivamente optimista. Podemos ajustar más la cota si consideramos como máximo valor la altura a la que se cruzan las dos línea grises.

$$\mathit{optimistic}((i,j)) = \frac{w_i + w_{j-1} + j - i}{2}.$$

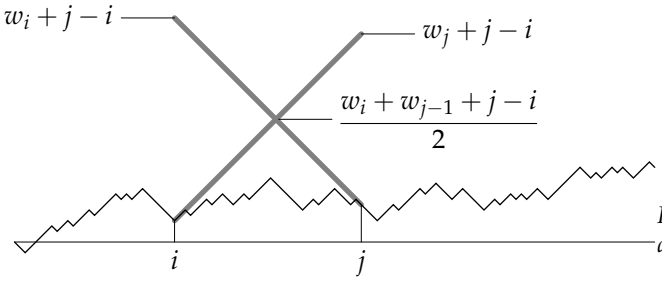


Figura 9.13: Representación gráfica de la cota superior definida.

9.9.3. Un algoritmo de ramificación y poda

Ya tenemos los elementos necesario para diseñar un algoritmo de ramificación y poda.

extreme_random_walk.py

```
1 from heap import MaxHeap
2
3 def upper_random_walk(w):
4   A = MaxHeap([((w[0]+w[-1]+len(w))/2, [0, len(w)])])
5   x = 0
6   max_value = w[0]
7   mems = 2
8   while len(A) > 0 and A.max()[0] > max_value:
```

```
    (score, (i,j)) = A.extract_max()
    for (ii, jj) in (i, (i+j)/2), ((i+j)/2, j):
      if ii == jj-1:
        if w[ii] > max_value:
          x = ii
          max_value = w[ii]
      else:
        mems += 2
        upper_bound = (w[ii]+w[jj-1]+jj-ii) / 2
        if upper_bound > w[ii]:
          A.insert( (upper_bound, (ii, jj)) )
  return x, max_value, mems
```

test_upper_random_walk.py

```
from extreme_random_walk import upper_random_walk
from random import randrange, seed

seed(1)
N = 100000
walk = []
h = 0
walk.append(0)
for i in xrange(1, N):
  if randrange(2): h += 1
  else:            h -= 1
  walk.append(h)

print 'El máximo es', max(walk),'en instante de índice', walk.index(max(walk))
print 'Solución por ramificación y poda:',
idx, maxval, mems = upper_random_walk(walk)
print 'valor máximo', maxval,'en instante de índice', idx
print mems, 'accesos a posiciones distintas del vector (talla %d).' % len(walk)
```

```
El máximo es 251 en instante de índice 69459
Solución por ramificación y poda: valor máximo 251 en instante de índice 69459

1956 accesos a posiciones distintas del vector (talla 100000).
```

EJERCICIOS

9-27 Modifica el programa para que encuentre el valor extremo de un paseo aleatorio en el que es posible ir al norte o al sur con pasos de una o dos unidades en cada instante.

9-28 Modifica el programa original para paseos aleatorios en los que no es posible dar más de 5 pasos seguidos en la misma dirección.

9-29 Diseña un algoritmo de ramificación y poda que nos indique la posición en que se encuentra un valor v en un vector que describe un paseo aleatorio. A continuación, modifica el algoritmo para que devuelva todas las posiciones en que aparece el valor v.

9.10. Otras técnicas de búsqueda

La búsqueda de soluciones en conjuntos de gran talla presenta un enorme interés en informática. Campos como la Inteligencia Artificial se han ocupado profusamente de este tema. El resultado es un elenco de métodos de exploración que guardan relación con ramificación y poda. Entre ellos podemos destacar la búsqueda A^* (que se lee «A estrella»). Es un proceso de ramificación de estados similar al que efectuamos en ramificación y poda, pero que gestiona dos «listas»: una lista de estados abiertos (equivalente a nuestra lista de estados activos) y otra de estados cerrados. Los estados cerrados son aquellos que ya han sido explorados. Un ventaja del esquema de A^* es que permite explorar de forma cómoda espacios en los que los estados pueden repetirse en el proceso de ramificación, es decir, no se cumple la restricción de que un estado se ramifique en subconjuntos suyos. Más que de árbol de estados, hablamos entonces de grafo de estados. A^* es una técnica de exploración de grafos. También se basa en el uso de una función de puntuación para orientar heurísticamente la búsqueda.

Los algoritmos de ramificación y poda o A^* no son de aplicación inmediata a problemas de juego con dos jugadores e información completa (como el ajedrez o las damas, juegos en los que no interviene el azar). Hay variantes aplicables en esos casos como la búsqueda minimax o la búsqueda alfa-beta. En este tipo de problemas no buscamos una solución factible, sino decidir cuál será próxima jugada. No obstante, hay un árbol de configuraciones del tablero (estados) cuya raíz es la configuración actual y que hemos de explorar. Más que buscar hojas, se decide explorar el árbol (o una parte de él) hasta cierto nivel de profundidad.

En la búsqueda minimax, por ejemplo, una función evalúa las configuraciones de tablero devolviendo valores positivos cuando un estado es ventajoso para el primer jugador y negativo cuando lo es para el segundo. Como los jugadores alternan su jugada y uno trata de maximizar el valor de la función mientras el otro trata de minimizarla, no interesa necesariamente escoger la jugada que nos conduzca a un estado beneficioso, sino una que impida al otro jugador llegar a estados que nos resultan perjudiciales.

EJERCICIOS

9-30 Dada una cadena $a = a_1 a_2 \ldots a_n$ de dígitos decimales, es decir, con $a_i \in [0..9]$ para i entre 0 y n, una k-segmentación de a, $S_{a,k}$, es una secuencia de k subcadenas contiguas de A. El valor de $S_{a,k}$ se denota con $V(S_{a,k})$ y es la suma de los valores de las subcadenas especificadas por $S_{a,k}$.

Por ejemplo, dada una cadena $a = 9300248$, la tupla $(93, 0024, 8)$ es una 3-segmentación y su valor es $93 + 24 + 8 = 125$.

Haciendo uso de la propiedad $V(S_{a,k}) \geq K \cdot 10^{\lfloor n/K \rfloor - 1} \min_{1 \leq i \leq n}(a_i)$, válida para toda k-segmentación $S_{a,k}$, desarrolla un algoritmo de ramificación y poda que, dada una cadena de dígitos decimales, obtenga una k-segmentación de valor mínimo de a. Haz una traza con esos valores.

9-31 Dado $N \in \mathbb{Z}^{>0}$, una K-factorización de N es una K-tupla de enteros positivos $(n_1, n_2, \ldots, n_K)$ tal que $N = n_1 \cdot n_2 \cdot \ldots \cdot n_K$.

Por ejemplo, $N = 9$ y $K = 3$, $n_1 = 1, n_2 = 3, n_3 = 3, 9 = 1 \cdot 3 \cdot 3$, es una posible 3-factorización de 9.

Escribe un algoritmo de ramificación y poda que encuentre una K-factorización de N tal que $\sum_{1 \leq i \leq K} n_i$ sea mínimo, asumiendo como cierta la siguiente propiedad sobre las $K-$factorizaciones:

si $n_i \in \mathbb{Z}^{>0}, 1 \leq i \leq K$ es una $K-$factorización de N, entonces se verifica que

$$\sum_{1 \leq i \leq K} n_i \geq K \sqrt[K]{N}.$$

9-32 Un sistema está formado por n módulos conectados en serie. Si uno sólo de los módulos falla, el sistema completo falla. Si la probabilidad de que un el módulo i-ésimo falle es p_i, la probabilidad de que el sistema entero falle es $1 - \prod_{1 \leq i \leq n}(1 - p_i)$.

Podemos construir cada módulo con hasta m_i copias de un mismo tipo de pieza dispuestas en paralelo. El módulo falla sólo cuando fallan todas sus piezas. Si la probabilidad de que falle una pieza de las que podemos montar en el módulo i es q_i, la probabilidad de que falle un módulo con k piezas en paralelo es $\prod_{1 \leq i \leq k} q_i$. El coste de una pieza para el módulo i es c_i.

Desarrolla un algoritmo de ramificación y poda que diga cuántas piezas hemos de poner en paralelo en cada módulo para minimizar la probabilidad de un fallo sin que el sistema supere un coste máximo C.

9-33 Desarrolla un algoritmo de ramificación y poda que, dado un conjunto finito C, y una función $S : C \in \mathbb{Z}^+$, encuentre la partición de C en k subconjuntos disjuntos C_1, C_2 ,..., C_k que minimice

$$\sum_{1 \leq i \leq k} \left(\sum_{a \in C_i} S(a) \right)^2.$$

Puedes utilizar esta propiedad:

$$\sum_{1 \leq i \leq k} \left(\sum_{a \in C_i} S(a) \right)^2 \geq |C| \left(\min_{a \in C} S(a) \right)^2.$$

9-34 Los organizadores de un partido de un fútbol han reservado la grada A para la afición del equipo visitante y la grada B para la afición del equipo local. Las aficiones de cada equipo están organizadas en peñas, cada una de ellas con un número de socios determinado. Únicamente se venden entradas a las peñas y sólo en el caso de que haya sitio para todos sus socios. Es decir, si se decide vender entradas a una peña, se venden entradas para todos sus socios. Por supuesto, no se pueden vender entradas de la grada A a peñas del equipo local o viceversa. Se desea maximizar la venta de entradas total sabiendo, para cada una de las aficiones, el número de peñas y el número de socios de cada una de ellas. Los datos son los siguientes:

- El número de entradas en la grada A es N_A, y el de la B, N_B.
- El número de peñas de la afición del equipo visitante es P_A, y el del equipo local, P_B.
- El número de socios de cada peña de la afición del equipo visitante se denota con SA_i, para $1 \leq i \leq P_A$. De forma análoga, el número de socios de cada peña del equipo local es SB_i, para $1 \leq i \leq P_B$.

Además, se sabe que no hay entradas suficientes para todas las peñas del equipo visitante y del local, es decir, $\sum_{1 \leq i \leq P_A} SA_i > N_A$ y $\sum_{1 \leq i \leq P_B} SB_i > N_B$.

Diseña un algoritmo de ramificación y poda que resuelva el problema. Presta atención a

a) La estructuras de datos más conveniente para representar estados y soluciones factibles. Calcula su coste espacial.

b) La función cota optimista, cuyo coste temporal debes estudiar.

c) El coste temporal de una iteración completa del algoritmo en los siguientes casos:

1) Cuando los estados que se obtienen al ramificar el estado seleccionado no son soluciones factibles.
2) Cuando se encuentra una solución que mejora la actual.

Escribe un algoritmo de ramificación y poda que resuelva el problema.

9-35 Se desea abonar M campos con un máximo de N sacos de abono. La utilización de i sacos en el campo j permite obtener una cosecha cuyo valor es v_{ij}.

Desarrolla un algoritmo de ramificación y poda que determine cuántos sacos hay que utilizar en cada campo para obtener una cosecha de valor máximo.

Recuerda que este problema admite solución por programación dinámica. ¿Puedes aprovechar este hecho de algún modo para aumentar la eficiencia del algoritmo de ramificación y poda?

9-36 Sean $\alpha, \beta : \mathbb{N} \to \mathbb{N}$, dos funciones definidas como $\alpha(i) = 3 \cdot i$, y $\beta(i) = \lfloor i/2 \rfloor$.

a) Escribir un algoritmo de ramificación y poda que determine, dados $m, n \in \mathbb{N}$ con $m > n$, cúal es la mínima secuencia S de aplicaciones de α y β que transforme m en n.

   Ejemplo: para $m = 15$ y $n = 4$, la mínima secuencia es $S = \beta\alpha\beta\beta$, la cual, en efecto, verifica: $S(15) = 4$, es decir, $\beta(\alpha(\beta(\beta(15)))) = 4$.

   Como función de cota inferior para un estado puede utilizarse esta propiedad:

$$g(s) = |s| + \lfloor \lg(\lfloor s(m)/n \rfloor) \rfloor,$$

   donde s es una solución parcial (un nodo en el árbol de ramificación y poda) o composición de funciones α y β, y $|s|$ es el número de funciones que forman la composición.

   En el ejemplo: $g(\alpha\beta\beta) = 3 + \lfloor \lg(\lfloor \alpha\beta\beta(15)/4 \rfloor) \rfloor = 3 + \lfloor \lg(\lfloor 9/4 \rfloor) \rfloor = 4$

b) Demuestra que la función g así definida es una cota inferior del número de aplicaciones de α y β requeridas para transformar m en n.

c) Discute cuál sería el comportamiento del algoritmo obtenido para aquellas instancias en las que no exista transformación posible de m a n.