Apuntes de Algorítmica

Andrés Marzal
María José Castro
Pablo Aibar

Borrador
6 de febrero de 2007

Capítulo 7
ALGORITMOS VORACES

El esquema algorítmico de resolución **voraz** o **avariciosa** encuentra aplicación en problemas de optimización: la búsqueda de la solución factible que hace óptimo (mínimo o máximo) el valor de cierta función objetivo. Las soluciones (factibles o no) de los problemas abordables con esta estrategia se pueden expresar mediante conjuntos (o secuencias) de ciertos componentes elementales. La estrategia voraz es muy sencilla: se trata de obtener un óptimo global tomando decisiones locales, es decir, en cada momento añade a la solución en curso un componente elemental con la esperanza de que forme parte de la solución óptima. En muchos problemas, esta esperanza se materializa en una realidad y se consigue escoger siempre el componente elemental que acabará formando parte de una solución factible que hace extremo el valor de la función objetivo.

Podemos entender mejor el proceso si comparamos el procedimiento voraz con la búsqueda con retroceso: en cada instante, se seleccionan modos de extender o ramificar el estado activo, esto es, de avanzar hacia una solución completa; pero en la estrategia voraz no se consideran varias alternativas que pueden descartarse si conducen a una vía muerta («retrocediendo»): una vez se ha tomado una decisión, ésta no se revisa nunca. Podemos imaginar un proceso de exploración del árbol que siempre escoge la rama adecuada y procede descendiendo un nivel en cada paso hasta llegar a un nodo hoja.

La estrategia voraz no es útil para resolver cualquier problema de optimización en el que hay que tomar decisiones locales. Pero incluso cuando no encuentra la solución óptima, puede constituir una buena estrategia para calcular aproximaciones a ésta con un bajo coste computacional. Un algoritmo que siempre encuentra una solución factible (aunque no necesariamente la óptima) y tal que podemos acotar la diferencia absoluta o relativa entre el valor de la función objetivo para la solución encontrada y el valor óptimo de la función objetivo, es un **algoritmo de aproximación**. Un algoritmo que en ocasiones no halla una solución factible o para el que no podemos establecer cotas absolutas o relativas entre el valor óptimo de la función objetivo y el de dicha función aplicada a la solución hallada es un **algoritmo heurístico** o una **heurística**.

Empezaremos por abordar el problema del desglose óptimo de una cantidad de di-

nero con el menor número posible de monedas y billetes. Veremos que, en ciertos casos, una estrategia voraz conduce a la solución óptima (y en otros no). Presentaremos a continuación un esquema algorítmico para la estrategia voraz y pasaremos a resolver varios problemas aplicando los principios expuestos en el esquema. El capítulo finaliza con la presentación de algunos problemas que no se resuelven correctamente con un algoritmo voraz, pero para los que la estrategia voraz proporciona aproximaciones aceptables.

7.1. Desglose óptimo de una cantidad de dinero

Dado un sistema monetario con billetes y monedas de diferentes valores deseamos desglosar una cantidad determinada de dinero en el menor número de billetes y monedas (en aras de la brevedad, a partir de ahora usaremos el término moneda para referirnos indistintamente a monedas propiamente dichas y a billetes). Supondremos que no hay limitación en la cantidad de monedas disponibles de cada valor.

Si imponemos la restricción de que hay limitación en la cantidad de monedas disponibles de cada valor, el problema resulta más interesante, pues se asemeja al que plantean máquinas expendedoras y cajeros automáticos. Estudiaremos este problema cuando consideremos la técnica algorítmica de Programación Dinámica.

Consideremos una instancia particular del problema. Contamos con monedas de 1, 2, 5 y 10 € y deseamos entregar 11 € con la menor cantidad de monedas. Tenemos varias posibilidades. La tabla 7.1 muestra todos los desgloses posibles.

Monedas				
1 €	**2 €**	**5 €**	**10 €**	**Total**
11	0	0	0	11
9	1	0	0	10
7	2	0	0	9
6	0	1	0	7
5	3	0	0	8
4	1	1	0	6
3	4	0	0	7
2	2	1	0	5
1	5	0	0	6
1	0	2	0	3
1	0	0	1	2
0	3	1	0	4

Tabla 7.1: Soluciones al desglose de 11 € en un sistema monetario con monedas de 1, 2, 5 y 10 €.

Cada una de estas posibilidades constituye una **solución factible**. ¿Cómo representar cada una de las soluciones factibles? Hay varias opciones:

- Una consiste en un conjunto de pares de la forma `(`*valor*`,` *cantidad*`)`, donde *valor* es el valor de una moneda y *cantidad* es el número de monedas de ese valor que usamos.

- Alternativamente, si suponemos que los valores de las monedas se suministran en un orden cualquiera, $(v_1, v_2, \ldots, v_N) \in \mathbb{N}^N$, podemos expresar cada solución factible como una N-tupla $(x_1, x_2, \ldots, x_N) \in (\mathbb{Z}^{\geq 0})^N$ donde x_i, para $1 \leq i \leq N$, es el número de monedas de valor v_i.

En un sistema con monedas de valor $v_1 = 1$, $v_2 = 2$, $v_3 = 5$ y $v_4 = 10$, la solución factible «una moneda de valor 1, ninguna de valor 2, dos de valor 5 y ninguna de valor 10» se expresaría con la primera representación así: $\{(5,2),(1,1)\}$. La misma solución, con la segunda representación, se expresaría así: $(1,0,2,0)$. Usaremos esta segunda representación.

Si $Q \in \mathbb{N}$ es la cantidad que deseamos desglosar y $(v_1, v_2, \ldots, v_N) \in \mathbb{N}^N$ contiene los valores de las monedas, el conjunto de soluciones factibles es:

$$X = \left\{ (x_1, x_2, \ldots, x_N) \in (\mathbb{Z}^{\geq 0})^N \Big| \sum_{1 \leq i \leq N} x_i v_i = Q \right\}.$$

El conjunto de la soluciones factibles X es subconjunto de otro, $X' = (\mathbb{Z}^{\geq 0})^N$, al que denominamos «conjunto de soluciones» y que forma el dominio de la denominada función objetivo, $f : X' \to \mathbb{R}$. Dicha función indica cuantas monedas hay en una solución:

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

De entre todos los elementos de X nos interesa el que proporciona un valor mínimo de la función objetivo, la solución óptima:

$$(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N) = \arg \min_{(x_1, x_2, \ldots, x_N) \in X} f((x_1, x_2, \ldots, x_N)).$$

Un ejemplo ayudará a entender qué es cada objeto: un sistema que sólo contiene monedas de valores $v_1 = 1$, $v_2 = 2$, $v_3 = 5$ y $v_4 = 10$ con el que deseamos desglosar la cantidad $Q = 11$. El conjunto de soluciones factibles es

$$\begin{aligned} X = \{&(11,0,0,0),(9,1,0,0),(7,2,0,0),(6,0,1,0),(5,3,0,0),(4,1,1,0),\\ &(3,4,0,0),(2,2,1,0),(1,5,0,0),(1,0,2,0),(1,0,0,1),(0,3,1,0)\}. \end{aligned}$$

El valor de $f(x)$ se obtiene sumando los componentes: $f((3,4,0,0)) = 3+4+0+0 = 7$. La solución óptima para el problema planteado es $(1,0,0,1)$, pues presenta el valor más bajo de f.

En Python, representaremos una solución con una lista cuyo primer elemento tiene índice 1. Nos vendrá bien disponer de una función para mostrar por pantalla una solución (factible o no):

greedychange.py

```
def show_change(x, v, Q):
    print 'Desglose de la cantidad %d con monedas de valores %s:' % (Q, v),
    print ', '.join('%d moneda%s de valor %d' % (xi, 's' if xi>1 else '', vi) \
                    for (xi,vi) in zip(x,v) if xi > 0) + '.'
    q = sum(xi*vi for (xi,vi) in zip(x, v))
    print 'Se proporciona la cantidad %d con %d monedas.' % (q, sum(x)),
    if q != Q: print 'No se proporciona la cantidad solicitada.'
    else:      print 'Desglose exacto.'
```

7.1.1. Fuerza bruta

Un algoritmo basado en la fuerza bruta consideraría todas las soluciones factibles (es decir, todos y cada uno de los elementos de X), calcularía el valor de f para cada una de ellas y escogería la que contiene menor número de monedas. El número de soluciones factibles puede crecer exponencialmente con el número de monedas, así que no resulta permisible explorarlas todas para escoger la mejor. La búsqueda con retroceso aceleraría la búsqueda descartando estados no prometedores, pero el algoritmo resultante seguiría necesitando tiempo exponencial para el peor de los casos.

EJERCICIOS

7-1 Diseña un algoritmo que resuelva, por fuerza bruta, el problema del desglose de una cantidad de dinero. Los datos del problema son la cantidad de dinero que deseamos desglosar y una lista con el tipo de monedas válidas en el sistema monetario (por ejemplo, la lista de valores `[1, 2, 5, 10, 20, 50, 100, 200, 500]` describe el sistema europeo sin contar fracciones de euro).

7-2 Resuelve el ejercicio anterior con un algoritmo de búsqueda con retroceso.

7-3 ¿Cuántas soluciones factibles hay al tratar de desglosar 1 000 € con monedas de 1, 2 y 5 €? ¿Es computacionalmente razonable encontrar la menor cantidad de monedas por fuerza bruta?

7.1.2. Una aproximación voraz naíf

Una estrategia voraz inmediata consistiría en recorrer en un orden cualquiera el conjunto de valores de monedas e ir seleccionando el mayor número posible de monedas de cada uno de dichos valores que no supera la cantidad que resta por desglosar:

greedychange.py *(cont.)*

```
def naive_greedy_change(v, Q):
    x = []
    for vi in v:
        x.append(Q/vi)
        Q = Q % vi
    return x
```

Esta estrategia no siempre conduce a una solución óptima, como se puede comprobar con esta ejecución de ejemplo:

```
1 from greedychange import naive_greedy_change, show_change
2
3 v = [1, 2, 5, 10]
4 x = naive_greedy_change(v, 11)
5 show_change(x, v, 11)
```

```
Desglose de la cantidad 11 con monedas de valores [1, 2, 5, 10]: 11 monedas de
 valor 1.
Se proporciona la cantidad 11 con 11 monedas. Desglose exacto.
```

Proporcionar cambio con once monedas de valor unitario no es la mejor solución (de hecho, es la peor).

7.1.3. Una solución voraz más elaborada

Una estrategia voraz alternativa consiste en ordenar primero las monedas por valor decreciente e ir seleccionando la mayor cantidad de monedas de cada uno de los tipos:

greedychange.py *(cont.)*

```
18 def greedy_change(v, Q):
19     x = []
20     for vi in sorted(v, reverse=True):
21         x.append(Q/vi)
22         Q = Q % vi
23     return x
```

Téngase en cuenta que la función devuelve la solución con una lista x «paralela» a la lista v, que ahora está ordenada de mayor a menor valor. Probemos el método con un subconjunto del sistema monetario del euro:

```
1 from greedychange import greedy_change, show_change
2
3 v = [1, 2, 5, 10]
4 x = greedy_change(v, 11)
5 show_change(x, sorted(v, reverse=True), 11)
```

```
Desglose de la cantidad 11 con monedas de valores [10, 5, 2, 1]: 1 moneda de v
alor 10, 1 moneda de valor 1.
Se proporciona la cantidad 11 con 2 monedas. Desglose exacto.
```

Para este sistema particular y cantidad a desglosar, el algoritmo ha encontrado la solución óptima. ¿Ocurre esto siempre?

7.1.4. Corrección

Debe tenerse en cuenta que la solución propuesta por el algoritmo no es la óptima para cualquier instancia del problema:

- A veces no hay solución al problema porque, en algunos sistemas monetarios, no es posible descomponer ciertas cantidades de dinero:

```
1 from greedychange import greedy_change, show_change
2
3 x = greedy_change([3, 7], 11)
4 show_change(x, sorted([3, 7], reverse=True), 11)
```

```
Desglose de la cantidad 11 con monedas de valores [7, 3]: 1 moneda de va
lor 7, 1 moneda de valor 3.
Se proporciona la cantidad 10 con 2 monedas. No se proporciona la cantid
ad solicitada.
```

El programa no ha conseguido desglosar los 11 €. No es posible hacerlo con monedas de 3 y 7 €.

- A veces no encuentra la solución óptima, aun cuando la hay:

```
1 from greedychange import greedy_change, show_change
2
3 x = greedy_change([1, 4, 5], 12)
4 show_change(x, sorted([1, 4, 5], reverse=True), 12)
```

```
Desglose de la cantidad 12 con monedas de valores [5, 4, 1]: 2 monedas d
e valor 5, 2 monedas de valor 1.
Se proporciona la cantidad 12 con 4 monedas. Desglose exacto.
```

La solución que usa 3 monedas de 4 € es mejor que la propuesta por el programa.

- Y a veces no sólo no encuentra la solución óptima, sino que ni siquiera encuentra una solución factible, aun cuando la hay:

```
1 from greedychange import greedy_change, show_change
2
3 x = greedy_change([2, 3, 4], 5)
4 show_change(x, sorted([2, 3, 4], reverse=True), 5)
```

```
Desglose de la cantidad 5 con monedas de valores [4, 3, 2]: 1 moneda de
valor 4.
Se proporciona la cantidad 4 con 1 monedas. No se proporciona la cantida
d solicitada.
```

Aunque existe una solución factible (una moneda de 3 € y una moneda de 2 €), el programa no la ha encontrado. El algoritmo propuesto es, pues, un algoritmo heurístico.

No obstante los casos presentados, esta estrategia funciona y proporciona siempre la solución óptima en ciertos sistemas monetarios:

- Sistemas en los que las monedas tienen valores que son potencias de determinada base c e incluye a c^0. Por ejemplo, en un sistema con monedas de 2^0, 2^1, 2^2, 2^3, 2^4 unidades podemos proporcionar un desglose óptimo de cualquier cantidad entera.

- Sistemas con una moneda de valor unitario y tales que el valor de cada moneda, considerados en orden de valor creciente, es múltiplo del valor anterior. (Nótese que el sistema anterior es un caso particular de éste.)


Recuerda que estamos suponiendo que disponemos de una cantidad ilimitada de monedas de cada valor. De no ser así, el problema es diferente y estos comentarios no son válidos.

Veamos la demostración de corrección en un sistema concreto para el que la estrategia voraz devuelve la solución de menor número de monedas. Consideremos, por ejemplo, un sistema con monedas de valores 2^0, 2^1 y 2^2. Dividiremos el estudio en diferentes casos según el valor de Q:

- $Q = 0$: la única solución es devolver 0 monedas de cada valor, que es lo que devuelve el algoritmo voraz.
- $Q = 1$: la única solución es devolver 1 moneda de valor 2^0, que es lo que devuelve el algoritmo voraz.
- $2 \leq Q < 4$: la solución voraz contiene una sola moneda de valor 2^1. Esto nos deja un «subproblema» consistente en devolver $Q - 2^1$ monedas en un sistema con monedas de valor 2^0, «subproblema» que se resuelve óptimamente siguiendo la estrategia voraz.

  Nos hemos de preguntar si no sería mejor no usar moneda alguna de valor 2^1. Si no entregamos una moneda de valor 2, hemos de entregar, en su lugar, dos de valor 1. O sea, es mejor usar una moneda de valor 2^1.

  Pero aún queda por responder una pregunta: de entre todas la soluciones al «subproblema», ¿cuál nos interesa? Obviamente, la óptima, la que usa el menor número de monedas. Y ésa es, precisamente, la que hemos visto que calcula el algoritmo cuando Q vale 0 o 1.
- $Q \geq 4$: la solución voraz contiene $m = \lfloor Q/2^2 \rfloor$ monedas de valor 2^2. El «subproblema» consistente en devolver $Q - m2^2$ monedas en un sistema con monedas de valores 2^0 y 2^1 se puede resolver óptimamente siguiendo la misma estrategia voraz. Cada moneda de valor 2^2 que eliminemos de la solución voraz obliga a introducir, como mínimo, dos monedas de valor inferior. Así pues, cualquier solución diferente requiere usar más monedas.


Aunque el algoritmo voraz es iterativo, no cuesta mucho plantearlo en términos de recursión y entender así que la solución de un problema no trivial pasa por la solución recursiva de un subproblema.

Hemos demostrado que si el algoritmo devuelve una solución factible, ésta es la óptima. Pero una cosa es que devuelva una solución óptima cuando encuentra una solución factible y otra diferente es demostrar que siempre encuentra una solución factible. Faltaría, pues, responder a una pregunta para asegurar que el algoritmo es correcto: ¿puede desglosarse cualquier cantidad Q con el sistema de monedas descrito?

7.1.5. Análisis de coste

Es evidente que el coste del algoritmo es $O(N \lg N)$, siendo N el número de valores diferentes de monedas: hemos de ordenar una lista con N valores como primer paso y el resto de acciones requiere tiempo $O(N)$.

EJERCICIOS

7-4 Demuestra que el desglose óptimo de una cantidad en todo sistema cuyas monedas son de valor potencia de determinada base c y que incluye a c^0 es susceptible de resolverse con el algoritmo voraz.

7-5 Demuestra que la estrategia seguida no proporciona una solución óptima si sólo disponemos de monedas de valores 1, 3 y 4.

7-6 Con un sistema monetario con monedas de valores 1, 5, 10, 25 y 100, el algoritmo siempre encuentra la solución óptima. Sin embargo, si nos quedamos sin monedas de 5 unidades, no necesariamente encuentra la solución óptima. Encuentra un contraejemplo para demostrar tal afirmación.

7-7 Demuestra si el algoritmo *greedy_change* encuentra o no la solución óptima para sistemas monetarios donde cada tipo de moneda vale al menos el doble que la moneda de valor inmediatamente inferior y la serie contiene una moneda de valor unitario.

7-8 El sistema monetario inglés, antes de la normalización decimal, constaba de medias coronas (30 peniques), florines (24 peniques), chelines (12 peniques), monedas de 6 peniques, 3 peniques y 1 penique. Demuestra que con este sistema monetario, el algoritmo no necesariamente genera una solución óptima.

7.2. La estrategia voraz

La estrategia voraz pretende encontrar una solución factible a un problema en el que las soluciones son subconjuntos de un conjunto D. En el ejemplo que hemos estudiado, una solución es una selección de monedas de un conjunto de monedas. El proceso de construcción de la solución es, en cierto sentido, similar al que seguíamos en la búsqueda con retroceso: representamos subconjuntos de D con «estados» y, en cada paso, refinamos el estado actual para acercarnos a la solución. En la búsqueda con retroceso considerábamos, de acuerdo con un criterio de ramificación, los «hijos» de un estado, lo que generaba un árbol de estados que procedíamos a explorar por primero en profundidad (descartando, eso sí, los estados no prometedores). La estrategia voraz sólo genera uno de dichos hijos y, naturalmente, sólo explora éste. No hay vuelta atrás: del potencial árbol de estados sólo se explora una rama de la raíz a una hoja. Y el criterio de ramificación y selección no considera información global: trata de generar un estado prometedor seleccionando un elemento de D que añade al estado actual sin considerar más información que la proporcionada por el resultado de las decisiones ya tomadas y por los elementos de D que aún no han sido seleccionados.

7.2.1. Esquema algorítmico

El esquema algorítmico de la estrategia voraz es un método general por el que se obtiene un subconjunto de D que se pretende que constituya una solución factible óptima:

greedyscheme.py

```
class GreedyScheme:
    def is_not_complete_and_is_extensible(self, x, D):
        """Devuelve falso si el estado x es completo o si D no contiene
            elementos compatibles con x, es decir, si x no puede extenderse
            con elementos de D."""

    def is_promising(self, x):
        """Recibe un estado. Devuelve cierto si cabe la posibilidad de que el conjunto
            de soluciones que representa x contenga al menos una solución factible."""

    def greedy_choice(self, x, D):
        """Selecciona un elemento de D (posiblemente a partir de información
            contenida en x)."""

    def greedy_strategy(self, D):
        x = set()
        while self.is_not_complete_and_is_extensible(x, D):
            d = self.greedy_choice(x, D)
            D.remove(d)
            if self.is_promising(x.union(set([d]))):
                x.add(d)
        return x

    def solve(self, D):
        return self.greedy_strategy(D)
```

Si no se imponen ciertas restricciones, no se garantiza la devolución de una solución factible y óptima o, ni siquiera, de una solución factible. Hay una característica importante de la estrategia voraz: el carácter irreversible de las decisiones tomadas. En cada instante se selecciona un elemento de D y si conduce a una solución factible, se añade a la solución en curso x. Esa selección, que constituye una decisión tomada localmente, es decir, sin considerar una solución factible completa, no es reconsiderada nunca. Es una de las razones en las que se basa la eficiencia y sencillez de la estrategia voraz.

7.2.2. El algoritmo de desglose de monedas reformulado en términos del esquema

Comprobemos que el esquema voraz puede instanciarse en el algoritmo que calcula el desglose de monedas. Nos interesa por el momento considerar que las soluciones factibles se expresan como conjuntos de pares «valor de moneda, cantidad de monedas». Empecemos interpretando cada una de las funciones del esquema en términos de nuestro problema:

- El método *greedy_choice* escoge un elemento de D: el que describe el uso del mayor número de monedas de mayor valor.

- El método *is_not_complete_and_is_extensible* recibe un conjunto de pares «valor de moneda, cantidad de monedas» x y, a partir del valor que deseamos cambiar, Q, devuelve «cierto» si y sólo si la suma del producto de los dos elementos de cada par es igual a la cantidad que deseamos desglosar.

- El método *is_promising* recibe un conjunto de pares «valor de moneda, cantidad de monedas» x, que es una solución posiblemente incompleta, y considerando el valor que deseamos cambiar, Q, devuelve «cierto» si x es subconjunto de algún elemento de X, es decir, si la suma del producto de todo par de valores es menor o igual que Q.

He aquí una implementación en la que aparece explícitamente cada uno de los elementos del esquema:

greedychange2.py

```
1  from greedyscheme import GreedyScheme
2
3  class GreedyChange(GreedyScheme):
4      def __init__(self, Q):
5          self.Q = Q
6
7      def is_not_complete_and_is_extensible(self, x, D):
8          return len(D) > 0 and sum( vi*xi for (vi,xi) in x ) < self.Q
9
10     def is_promising(self, x):
11         return sum( vi*xi for (vi,xi) in x ) <= self.Q
12
13     def greedy_choice(self, x, D):
14         return max(D)
```

Podemos comprobar que suministra la solución correcta, aunque codificada de un modo diferente al usado en pruebas anteriores:

```
1  from greedychange2 import GreedyChange
2
3  Q = 11
4  v = [1, 2, 5, 10]
5  D = set( (vi, j) for vi in v for j in xrange(1, Q/vi+1) )
6  print 'D: %s\nSolución: %s' % (sorted(D), GreedyChange(Q).solve(D))
```

```
D: [(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (1
, 10), (1, 11), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (5, 1), (5, 2), (10, 1
)]
Solución: set([(1, 1), (10, 1)])
```

7.2.3. Propiedades que debe observar la solución de un problema abordable mediante la estrategia voraz

Los algoritmos voraces se basan en la selección del elemento de D que más conviene en cada paso. La elección de dicho elemento se denomina «**elección avariciosa**». Para que un problema sea abordable mediante la estrategia voraz, sus soluciones deben observar dos propiedades:

a) La **propiedad de la elección avariciosa**: existe una solución óptima del problema que incluye la elección avariciosa.

b) La **propiedad de la subestructura óptima**: Tras hacer una elección avariciosa, podemos encontrar una solución óptima combinándola con la solución óptima al subproblema restante. Esta propiedad conduce a una visión recursiva en el diseño de un método resolutivo.

En el caso del problema del cambio de monedas hemos basado en estas propiedades la demostración de corrección para juegos de monedas de valores 2^0, 2^1 y 2^2 para $Q \geq 4$:

- Hemos demostrado que la elección avariciosa ($m = \lfloor Q/2^2 \rfloor$) forma parte de la solución óptima: si no la incluimos, hemos de usar más monedas.
- Y hemos visto que una vez tomamos la elección avariciosa, $m = \lfloor Q/2^2 \rfloor$, hemos de desglosar $Q - m2^2$ con monedas de valores 2^1 y 2^0. La solución de esta nueva instancia debe ser óptima, es decir, debe hacer uso del menor número de monedas para combinarse adecuadamente con la solución de la instancia original: si hay una solución de la nueva instancia que usa más monedas que otra, la segunda siempre será más conveniente que la primera, ya que a las monedas que «consume» esa solución hemos de sumar las que se han de dar para resolver la instancia original.

Por otra parte, vale la pena reseñar que el esquema voraz suele conducir a soluciones muy eficientes y relativamente sencillas de implementar. No obstante, demostrar la corrección del algoritmo no es, por regla general, trivial.

7.2.4. Refinamientos

Ahora que hemos identificado los diferentes elementos del algoritmo, podemos implementar una versión con menor sobrecarga en llamadas a funciones y que efectúe menos cálculos repetidos (como los sumatorios que tienen lugar en *is_complete* e *is_factible*).

greedychange3.py

```
1 def greedy_change(D, Q):
2     x = set()
3     total = 0
4     while len(D) > 0 and total < Q:
5         (vi,xi) = max(D)
6         D.remove((vi,xi))
```

```
7         if total + vi*xi <= Q:
8             x.add((vi,xi))
9             total += vi*xi
10    return x
```

```
1  from greedychange3 import greedy_change
2
3  Q = 11
4  v = [1, 2, 5, 10]
5  D = set( (vi, j) for vi in v for j in xrange(1, Q/vi+1) if vi * j <= Q)
6  print 'D:', D, '\nSolución:', greedy_change(D, Q)
```

```
D: set([(1, 2), (1, 3), (5, 1), (2, 2), (1, 4), (2, 4), (1, 5), (1, 8), (1, 6)
, (1, 9), (10, 1), (1, 7), (1, 10), (2, 3), (2, 5), (1, 11), (5, 2), (1, 1), (
2, 1)])
Solución: set([(1, 1), (10, 1)])
```

Hemos construido el conjunto D explícitamente, cuando resulta innecesario. Veamos por qué. Cada elemento de D es un par «valor de moneda, cantidad de moneda». Se consideran ordenadamente, de mayor a menor valor de la moneda y, a igual valor de moneda, se escoge en primer lugar el de mayor cantidad de monedas. Una vez se ha seleccionado y añadido a la solución un par, cualquier otro par de D con el mismo valor de moneda acaba siendo descartado, pues la solución se forma poniendo la máxima cantidad posible de monedas de cada valor. Para evitar construir D explícitamente podemos recorrer los valores de moneda en orden decreciente y considerar únicamente la máxima cantidad de monedas de cada valor con que podemos avanzar hacia la solución final.

greedychange4.py

```
1  def greedy_change(D, Q):
2      D = sorted(D, reverse=True)
3      x = set()
4      total = 0
5      i = 0
6      while i <= len(D) and total < Q:
7          xi = (Q-total) / D[i]
8          if xi > 0:
9              x.add((D[i],xi))
10             total += D[i]*xi
11         i += 1
12     return x
```

```
1  from greedychange4 import greedy_change
2
3  Q = 11
4  v = [1, 2, 5, 10]
5  print 'Solución:', greedy_change(v, Q)
```

```
Solución: set([(1, 1), (10, 1)])
```

Por otra parte, el conjunto x puede representarse con una tupla de cantidades si suponemos un determinado orden en las monedas (como hicimos en el apartado anterior, al plantear el problema). De paso, refinaremos el algoritmo sustituyendo el bucle **while** del método de selección por un bucle **for**.

greedychange5.py

```
def greedy_change(v, Q):
    v = sorted(v, reverse=True)
    x = [0] * len(v)
    total = 0
    for (i, vi) in enumerate(v):
        x[i] = (Q-total) / vi
        total += vi * x[i]
        if total >= Q: break
    return x
```

```
from greedychange5 import greedy_change
from greedychange import show_change
from offsetarray import OffsetArray

Q = 11
v = OffsetArray([1, 2, 5, 10])
x = greedy_change(v, Q)
show_change(x, OffsetArray(sorted(v, reverse=True)), Q)
```

```
Desglose de la cantidad 11 con monedas de valores [10, 5, 2, 1]: 1 moneda de v
alor 10, 1 moneda de valor 1.
Se proporciona la cantidad 11 con 2 monedas. Desglose exacto.
```

Y si finalmente suprimimos la variable *total* y recorremos directamente los elementos de v, obtenemos una versión que coincide con la presentada en el apartado 7.1.

```
def naive_greedy_change(v, Q):
    x = []
    for vi in v:
        x.append(Q/vi)
        Q = Q % vi
    return x
```

EJERCICIOS

7-9 Queremos comparar la «calidad» de dos sistemas monetarios diferentes: el del euro (considerando sólo las monedas de valor entero) y un sistema imaginario en el que hay monedas de valores 1, 5, 25, 100, 500. Mediremos la calidad como el número medio de monedas necesario para desglosar cualquier cantidad Q comprendida entre 1 y 1 000. Diseña un programa que efectúe el cálculo y nos indique cuál es de mayor calidad.

7-10 Implementa un programa que devuelva el desglose óptimo para el sistema monetario europeo al completo, esto es, considerando también las monedas fraccionarias (de 0.50, 0.20, 0.10, 0.05, 0.02 y 0.01 euros). Adapta el algoritmo para que trabaje con valores no enteros.

7.3. El problema de la mochila con fraccionamiento

Tenemos una mochila con capacidad para cargar W unidades de peso y N productos que podemos cargar en ella. Cada producto tiene un peso $w_i \in \mathbb{R}^{\geq 0}$ y un valor $v_i \in \mathbb{R}^{\geq 0}$ por unidad, para $1 \leq i \leq N$. Podemos fraccionar los productos y guardar en la mochila sólo parte de cada uno. Si r es un real entre 0 y 1, el beneficio que aporta guardar r partes del producto i es $r \cdot v_i$. ¿Cómo cargar la mochila de forma que el beneficio sea máximo sin exceder el límite de carga?

No pienses literalmente en términos de una mochila. El problema surge, por ejemplo, en la carga de productos para su transporte en camiones, barcos, etc.

7.3.1. Formalización

Formalicemos el problema en términos de optimización, pues buscamos una carga de la mochila que proporcione máximo valor. Una solución factible es una lista de valores entre 0 y 1, cada uno de los cuales indica la cantidad de un producto que cargamos en la mochila, tal que no represente una carga superior a la capacidad de la mochila:

$$X = \left\{ (x_1, x_2, \ldots, x_N) \in [0,1]^N \Big| \sum_{1 \leq i \leq N} x_i w_i \leq W \right\}.$$

No buscamos una solución factible cualquiera: buscamos una de las que hagan máximo el beneficio obtenido por la carga que representa. Nuestra función objetivo es, pues,

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

y buscamos

$$(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N) = \arg \max_{(x_1, x_2, \ldots, x_N) \in X} f((x_1, x_2, \ldots, x_N)).$$

Estudiar un ejemplo concreto puede resultar de ayuda. Supongamos que en la mochila caben $W = 20$ kg de peso, y que tenemos tres productos ($N = 3$) con pesos $w_1 = 18$ kg, $w_2 = 15$ kg y $w_3 = 10$ kg. El valor de cada uno de los productos es $v_1 = 25$ €, $v_2 = 24$ € y $v_3 = 15$ €. Una carga factible de la mochila sería $(1/2, 1/3, 1/4)$. Su peso es de 16.5 kg y el beneficio que nos reporta, de 24.25 €. Naturalmente, no es la única solución factible. Ésta otra, por ejemplo, $(1, 2/15, 0)$, supone cargar la mochila con un peso de 20 kg y obtener un beneficio de 28.20 €. No podemos plantearnos siquiera enumerar explícitamente el conjunto de soluciones factibles para buscar entre ellas, una por una, la solución óptima, pues su número es infinito.

7.3.2. Una aproximación voraz: «mientras quepa»

Pensemos ahora en términos de una estrategia voraz. Nos interesa ir formando una solución considerando cada producto por separado. Cuando consideremos un producto determinado, trataremos de aumentar el beneficio que nos reporta su carga en la mochila. En principio parece que el mayor beneficio que podemos obtener de la carga de un producto se alcanza poniendo «cuanto más, mejor». La limitación vendrá impuesta por la carga que aun soporte la mochila (pues ya habremos introducido otros productos). Finalizaremos tan pronto esté cargada la mochila hasta su límite de peso o hayamos considerado todos los productos.

Las soluciones pueden representarse eficientemente con un vector x indexado por el número de producto. Nos vendrá bien disponer de una función que muestre una solución por pantalla:

fractionalknapsack.py

```
from offsetarray import OffsetArray

def show_fractional_knapsack(x, v, w):
    print 'Valor total: %s.' % sum(x[i]*v[i] for i in xrange(1, len(x)+1)),
    print 'Carga total: %s.' % sum(x[i]*w[i] for i in xrange(1, len(x)+1))
    print 'Detalle de la carga: %s.' \
          % ', '.join('%.1f%% del producto %d' % (100*x[i], i) \
                      for i in xrange(1, len(x)+1) if x[i] > 0)
```

Este primer algoritmo voraz considera los objetos en el orden que nos son dados y mete en la mochila la mayor cantidad posible de cada uno de ellos.

fractionalknapsack.py *(cont.)*

```
def suboptimal_fractional_knapsack(w, v, W):
    x = OffsetArray([0]*len(w))
    for i in xrange(1, len(w)+1):
        x[i] = min(1, W/float(w[i]))
        W -= x[i] * w[i]
    return x
```

Desgraciadamente, no siempre da con la solución óptima. He aquí un ejemplo de ejecución de la función:

```
from fractionalknapsack import suboptimal_fractional_knapsack, \
                               show_fractional_knapsack
from offsetarray import OffsetArray

W, v, w = 50, OffsetArray([60, 30, 40, 20, 75]), OffsetArray([40, 30, 20, 10, 50])
x = suboptimal_fractional_knapsack(w, v, W)
show_fractional_knapsack(x, v, w)
```

```
Valor total: 70.0. Carga total: 50.0.
Detalle de la carga: 100.0% del producto 1, 33.3% del producto 2.
```

El lector puede comprobar que hay una solución mejor: si metemos la totalidad de los objetos 3 y 4 y el 40 % del objeto 5, la mochila también se carga completamente y obtenemos 90 como valor total de la carga.

7.3.3. Otra aproximación voraz: «mientras quepa», pero con criterio de ordenación

Considerar los productos en un orden arbitrario no parece conveniente: imaginemos un producto de peso enorme que nos reporta un beneficio ridículo. Si consideramos ese producto en primer lugar, cargaremos parte de la mochila con un beneficio menor que el que obtendríamos con otros productos. ¿Hay alguna forma de ordenar los productos de modo que, cuando los consideremos uno tras otro, aseguremos un beneficio máximo en cada carga local? Hay tres criterios de ordenación que, a primera vista, parecen convenientes:

a) De mayor a menor valor.

b) De menor a mayor peso.

c) De mayor a menor relación valor/peso.

La última de las opciones conduce a un algoritmo voraz que proporciona la solución óptima:

fractionalknapsack.py *(cont.)*

```
18 def fractional_knapsack(w, v, W):
19     x = OffsetArray([0] * len(w))
20     v = OffsetArray(sorted([(v[i]/float(w[i]), i) for i in xrange(1,len(w)+1)],
21                            reverse=True))
22     for (ratio, i) in v:
23         x[i] = min(1, W/float(w[i]))
24         W -= x[i] * w[i]
25     return x
```

Probemos el programa con la misma instancia con la que probamos nuestro primer algoritmo voraz y comprobaremos que proporciona la solución óptima:

```
1 from fractionalknapsack import fractional_knapsack, show_fractional_knapsack
2 from offsetarray import OffsetArray
3
4 W = 50
5 v, w = OffsetArray([60, 30, 40, 20, 75]), OffsetArray([40, 30, 20, 10, 50])
6 q = fractional_knapsack(w, v, W)
7 show_fractional_knapsack(q, v, w)
```

```
Valor total: 90.0. Carga total: 50.0.
Detalle de la carga: 100.0% del producto 3, 100.0% del producto 4, 40.0% del p
roducto 5.
```

7.3.4. Corrección del algoritmo

Una técnica interesante de demostración de la corrección del algoritmo es la denominada «reducción—a la diferencia». Consiste en considerar que toda solución diferente de la que devuelve el algoritmo voraz es igual o peor desde el punto de vista del valor de la función objetivo. Las soluciones se suelen expresar como secuencias y se considera una diferente de la que devuelve el algoritmo, pero que comparte con ella un prefijo. Entonces se estudia por qué el primer elemento en el que difiere la solución alternativa de la que devuelve el algoritmo debe modificarse si se desea alcanzar un valor óptimo de la función objetivo. Apliquemos la técnica al algoritmo *fractional_knapsack*.

Para simplificar, y sin pérdida de generalidad, supondremos que la lista de productos está ordenada por ratio valor/peso decreciente, o sea, $v_1/w_1 \geq v_2/w_2 \geq \cdots v_N/w_N$. Demostraremos que, dada una solución factible $(x_1, x_2, \ldots, x_N)$, su valor $\sum_{1 \leq i \leq N} x_i v_i$ es menor o igual que el de la solución devuelta por el algoritmo voraz, $(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N)$.

Fijémonos en el primer índice i tal que $x_i \neq \hat{x}_i$. Está claro que $x_i < \hat{x}_i$, ya que el algoritmo voraz selecciona la mayor cantidad posible del producto i. Esto significa que estamos dejando una mayor capacidad de carga libre para el resto de productos. Sea $a = (\hat{x}_i - x_i)w_i$ dicha capacidad de carga. ¿Cuál es el mayor beneficio que podemos obtener con esa capacidad de carga disponible? El mayor beneficio posible nos lo ofrece el producto $i+1$, del que podemos añadir (en el mejor de los casos) una cantidad q'_{i+1} tal que $a = q'_{i+1}w_{i+1}$. El beneficio adicional sería entonces $q'_{i+1}v_{i+1}$; pero ese valor siempre será inferior o igual a $(\hat{x}_i - x_i)v_i$, ya que

$$\frac{v_i}{w_i} \geq \frac{v_{i+1}}{w_{i+1}} \Rightarrow a\frac{v_i}{w_i} \geq a\frac{v_{i+1}}{w_{i+1}} \Rightarrow (\hat{x}_i - x_i)w_i\frac{v_i}{w_i} \geq q'_{i+1}w_{i+1}\frac{v_{i+1}}{w_{i+1}} \Rightarrow (\hat{x}_i - x_i)v_i \geq q'_{i+1}v_{i+1}.$$

Así queda demostrado que en cada paso conviene cargar la mayor cantidad posible del producto de mayor ratio valor/peso.

EJERCICIOS

7-11 Demuestra que si los productos se seleccionan por orden de mayor a menor valor, no siempre se encuentra una solución óptima aplicando una estrategia voraz.

7-12 Demuestra que si los productos se seleccionan por orden de menor a mayor peso, no siempre se encuentra una solución óptima aplicando una estrategia voraz.

7-13 ¿Funciona correctamente el algoritmo si no podemos fraccionar los objetos, es decir, si sólo podemos incluir o no incluir completamente cada objeto?

7.3.5. Coste computacional

El algoritmo empieza ordenando los ratios valor/peso. El coste temporal de la ordenación es $O(N \lg N)$ para N objetos. Recorre a continuación los N elementos ordenados y efectúa una serie de cálculos que requieren tiempo $O(1)$ sobre cada uno de ellos. El coste temporal total es, pues, $O(N \lg N)$.

EJERCICIOS

7-14 ¿Funciona correctamente el algoritmo si puedes poner tanta cantidad como desees de cualquiera de los productos (es decir, x_i puede ser mayor que 1)? ¿Sigue funcionando un algoritmo voraz? ¿Puedes diseñar una versión más eficiente (y sencilla)?

7-15 Supongamos que la solución óptima selecciona completa o parcialmente un número de productos k y que no incluye nada de los restantes $N - k$. Diseña e implementa un algoritmo que solucione el problema en tiempo $O(N + k \lg N)$, que es una cota asintótica mejor que $O(N \lg N)$. (Pista: Utiliza un max-heap.)

7.4. Selección de actividades

Queremos obtener el mayor beneficio posible alquilando una sala para la realización de una serie de actividades. Cada actividad ocupa un intervalo de tiempo con una hora de inicio s y una hora de terminación t. Sólo puede llevarse a cabo una actividad en cada instante. No obstante, si una tarea acaba en un instante y otra empieza en ese mismo instante, no hay conflicto alguno. El beneficio que nos reporta cualquier actividad es el mismo, así que el mayor beneficio global se obtiene efectuando el mayor número posible de alquileres. Dado un conjunto de actividades descritas por sus instantes inicial y final,

$$C = \{(s_1, t_1), (s_2, t_2), \ldots, (s_N, t_N)\},$$

tales que $s_i < t_i$, para $1 \leq i \leq N$, queremos seleccionar el mayor subconjunto de actividades sin solapamiento.

Este problema se engloba en la familia de los denominados «problema de planificación de tareas» (en inglés, *task scheduling*).

7.4.1. Formalización

Empezaremos por formalizar el problema y modelarlo en términos de la estrategia voraz. Una solución es un subconjunto de actividades, es decir, un subconjunto de C. Buscamos una solución que maximice el beneficio, es decir, de talla máxima. El conjunto X, que describe las soluciones factibles está formado por los subconjuntos de C tales que si (s_i, t_i) y (s_j, t_j) son elementos del mismo subconjunto, entonces $s_i < t_i \leq s_j < t_j$ o $s_j < t_j \leq s_i < t_i$, es decir, que no se solapan. La función objetivo, cuyo valor queremos maximizar, es la cardinalidad del conjunto que suministramos como argumento: $f(x) = |x|$. Queremos conocer, por tanto,

$$\hat{x} = \arg \max_{x \in X} |x|.$$

7.4.2. Una solución voraz

Una aplicación inmediata del esquema algorítmico voraz selecciona al azar una actividad cualquiera y, si no se solapa con las ya seleccionadas, la añade a la solución en curso. Es evidente que esta estrategia siempre proporciona una solución factible. Pero un contraejemplo basta para demostrar que ésta no tiene porqué ser óptima. Supongamos el conjunto de tareas $C = \{(1,5), (1,2), (3,6), (4,7), (6,7)\}$ (véase la figura 7.1). Si empezamos seleccionando el elemento $(1,5)$ sólo podremos seleccionar, además, la tarea $(6,7)$. Sin embargo, la solución $\{(1,2), (3,6), (6,7)\}$ proporciona mayor beneficio.

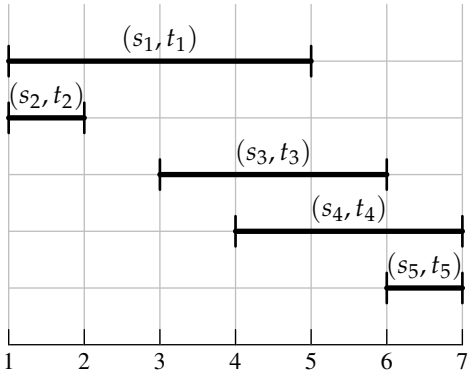


Figura 7.1: Representación gráfica de una instancia del problema de la selección de actividades.

¿En qué orden podemos seleccionar cada actividad de modo que maximicemos el beneficio? Podemos pensar en diversos criterios:

- De mayor a menor duración.
- De menor a mayor duración.
- De menor a mayor instante de inicio.
- De menor a mayor instante de finalización.

La estrategia que proporciona la solución óptima es la cuarta, como demostraremos a continuación.

EJERCICIOS

7-16 Demuestra que los otros tres criterios conducen a soluciones no óptimas.

activities_selection.py

```
1  def activities_selection1(C):
2      x = set()
3      for (s, t) in sorted(C, key=lambda (s,t): t):
4          if not any(s' <= s < t' or s' < t <= t' for (s', t') in x):
5              x.add( (s, t) )
6      return x
```

En el programa, *sorted* recibe una lista de pares y una «clave» (*key*) de ordenación. Dicha clave es una función anónima que extrae el segundo elemento de una tupla. El resultado es una lista ordenada por valor creciente del segundo elemento de cada par.

```
1  from activities_selection import activities_selection1
2
3  x = activities_selection1([(1,5), (1,2), (3,6), (4,7), (6,7)])
4  print 'Actividades seleccionadas:',
5  for (s, t) in sorted(x): print (s, t),
6  print
```

```
Actividades seleccionadas: (1, 2) (3, 6) (6, 7)
```

7.4.3. Corrección

Otra estrategia de demostración para algoritmos voraces pasa por encontrar una subestructura óptima, algo que haremos frecuentemente en los problemas de «programación dinámica». La idea es considerar qué conviene hacer ante la primera decisión que hemos de tomar. Una vez tomada la decisión, tendremos que resolver un problema de naturaleza similar pero de menor talla: un «subproblema». Tendremos que demostrar que la solución óptima del subproblema que hemos de resolver tras tomar la decisión voraz es mejor o igual que la del que hemos de resolver si hubiésemos tomado una decisión distinta y que, la solución óptima del subproblema, combinada con la decisión local, conduce a una solución óptima globalmente.

Apliquemos esta idea a nuestro algoritmo. Supongamos, sin pérdida de generalidad, que las tareas se nos proporcionan ordenadas de menor a mayor por el instante de finalización, es decir, que $t_i \leq t_{i+1}$, para $1 \leq i < N$. En aras de la simplicidad supondremos inicialmente que no hay dos tareas que finalizan en el mismo instante, es decir, que $t_i < t_{i+1}$, para $1 \leq i < N$.

Nuestra primera pregunta es, ¿hemos de seleccionar la tarea (s_1, t_1)? Consideramos las dos posibilidades:

- Si la seleccionamos, estamos generando un subproblema definido por un nuevo conjunto de actividades $C' = \{(s_i, t_i) \in C \mid s_i \geq t_1\}$ que queremos resolver maximizando el número de tareas seleccionadas. Sea m' el mayor número de tareas que podemos seleccionar de C'.

  Seleccionar (s_1, t_1) proporciona una solución con $1 + m'$ tareas.

- Si no seleccionamos la primera tarea, acabaremos seleccionando en primer lugar alguna otra, (s_k, t_k) tal que $t_k > t_1$. Esto nos dejará un subproblema definido por $C'' = \{(s_i, t_i) \in C \mid s_i \geq t_k\}$. Sea m'' el mayor número de tareas que podemos seleccionar de C''.

  Seleccionar (s_k, t_k), para $k > 1$, proporciona una solución con $1 + m''$ tareas.

Como $C'' \subseteq C'$, tenemos que, necesariamente, $m'' \leq m'$. Así pues, el número de tareas seleccionadas con la primera decisión es mayor o igual que el de tareas seleccionadas con la segunda opción. Conclusión: la primera tarea debe seleccionarse. (La figura 7.2 ilustra, con un ejemplo concreto, este razonamiento.)

Como esto nos plantea un nuevo problema de la misma naturaleza, podemos aplicar este principio recursivamente al nuevo problema. Esto conduce a un procedimiento iterativo de selección de tareas por instante de finalización creciente.

Habíamos dejado pendiente el caso en que dos tareas finalicen en el mismo instante t_1, es decir, cuando podemos escoger una cualquiera de ellas en primer lugar. Es indiferente escoger una u otra, pues ambas suponen la inclusión de una actividad en la solución y plantean la resolución del mismo subproblema.

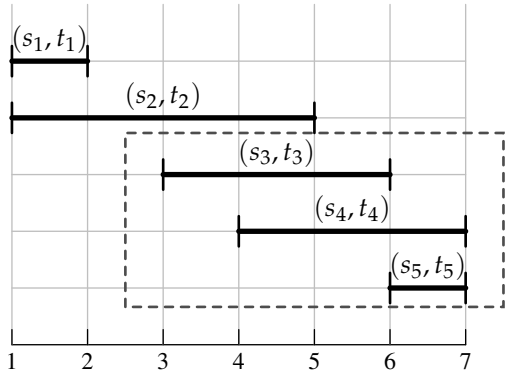


Figura 7.2: Instancia del problema de la selección de actividades considerada en la figura 7.1, pero con las tareas ordenadas de menor a mayor instante de finalización. El marco de trazo discontinuo muestra el subproblema que resulta de seleccionar la primera tarea como parte de la solución. El subproblema que resulta de seleccionar cualquier otra en primer lugar es un subconjunto de este subproblema.

7.4.4. Análisis de coste (y refinamiento)

Es fácil ver que este algoritmo es $\Theta(N^2)$. Ordenar la lista de los n eventos requiere tiempo $O(N \lg N)$. Cada uno de los N elementos de la lista se compara con, en el peor de los casos, todos los demás. Esta segunda parte del algoritmo requiere, pues, tiempo $\Theta(N^2)$. Es posible, no obstante, reducir el coste temporal a $O(N \lg N)$. ¿Cómo? No es necesario comprobar si la actividad que estamos considerando con cada iteración del bucle **for** solapa con cualquier actividad ya seleccionada: basta con comprobar si hay solapamiento con la última que hemos seleccionado.

activities_selection.py *(cont.)*

```
9  def activities_selection(C):
10     x = set()
11     t' = min(s for (s,t) in C)
12     for (s, t) in sorted(C, key=lambda (s, t): t):
13         if t' <= s:
14             x.add( (s, t) )
15             t' = t
16     return x
```

```
1  from activities_selection import activities_selection
2
3  x = activities_selection([(1,5), (1,2), (3,6), (4,7), (6,7)])
4  print 'Actividades seleccionadas:',
5  for (s, t) in sorted(x): print (s, t),
6  print
```

```
Actividades seleccionadas: (1, 2) (3, 6) (6, 7)
```

El coste del bucle interior ha pasado a ser $\Theta(N)$, pero la ordenación del vector sigue siendo $O(N \lg N)$, término que domina el coste del algoritmo.

. . . . . EJERCICIOS . . . . .

7-17 Queremos programar una serie de actividades. Cada actividad tiene una hora de inicio, una de finalización y un número de asistentes. Queremos maximizar el número de actividades

realizadas. Si dos programaciones de actividades suponen realizar el mismo número de actividades, nos interesa aquella a la que acuda mayor número de asistentes. ¿Devolverá la solución óptima un algoritmo voraz?

7-18 Nos han contratado para hacer críticas de estrenos cinematográficos. Cada crítica enviada a la redacción se paga con una cantidad fija, así que cada viernes tratamos de ver el mayor número posible de estrenos. En cada línea de un fichero de texto tenemos indicado el título de una película, su duración en minutos y los cines y horas en los que se proyecta. Este fichero muestra un ejemplo:

```
Paycheck|119|Rex 19:30 22:30
Pegado a ti|118|Ábaco 16:00 19:00 22:00 16:50 19:40 22:20
Tesoro del Amazonas|105|Ábaco 16:10 18:20 20:30 22:40|Rex 19:30 22:30
Titus|162|Ábaco 16:05 19:15 22:35
```

Tardamos 10 minutos en ir de un cine a otro, así que si formamos un conjunto de actividades con pares que contienen la hora de inicio y la de finalización de la película más 10 minutos para cada sesión de cada película, ¿devolverá la solución óptima un algoritmo voraz como el diseñado para la selección de actividades? ¿Por qué? Si la respuesta es negativa, ¿podrías diseñar una algoritmo basado en alguno de los esquemas algorítmicos estudiados en capítulos anteriores? Si es así, hazlo.

7-19 Nos han contratado para hacer críticas de estrenos cinematográficos en un festival de cine con un único pase por película. Cada crítica enviada se paga con una cantidad fija, así que pretendemos ver el mayor número posible de películas. En cada línea de un fichero de texto tenemos indicado el título de una película, su duración en minutos y la hora en la que se inicia la proyección. Este fichero muestra un ejemplo:

```
Paycheck|119|19:30
Pegado a ti|118|16:00
Tesoro del Amazonas|105|22:40
Titus|162|19:15
```

Diseña un programa que planifique óptimamente nuestra jornada de trabajo. ¿Qué habría que hacer para que, caso de existir dos o más soluciones óptimas posibles, el algoritmo nos ofrezca la que nos permite acabar lo más temprano posible?

7.5. El repostaje

Un camionero debe realizar la ruta entre dos ciudades con un camión que le permite, con el depósito lleno, recorrer n kilómetros. El camionero parte con el depósito lleno y dispone de una hoja de ruta que indica la distancia entre cualquier gasolinera y la siguiente. Desea pararse a repostar el menor número de veces posible para reducir al máximo el tiempo de viaje. Supondremos que siempre hay una distancia menor entre dos gasolineras que el número de kilómetros n que el camión puede recorrer sin repostar; de lo contrario, el problema no presenta solución.

El problema queda descrito por el número n de kilómetros que pueden recorrerse sin repostar, el número de estaciones de servicio, M, y un vector d con la distancia que separa cada estación de la siguiente. El valor de $d[i]$ es la distancia entre los puntos i e $i+1$ de la

ruta, teniendo en cuenta que el punto 0 es el de partida y el punto $M+1$ es el de llegada. La figura 7.3 ilustra gráficamente un problema con 6 gasolineras.

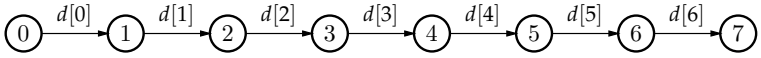


Figura 7.3: Representación del grafo asociado a una ruta con 6 estaciones de servicio.

Una estrategia voraz para determinar en qué gasolineras debe parar consistirá en recorrer el mayor número posible de kilómetros sin repostar, esto es, tratar de ir desde cada gasolinera en donde pare a repostar a la más lejana posible, y así hasta llegar a su destino. El siguiente algoritmo nos devuelve un vector con el punto de partida, las gasolineras en que hemos de parar a repostar y el punto de llegada:

gas.py

```
1  def gas_stations(M, d, n):
2      stop = [0]
3      j = 1
4      km = n
5      for i in xrange(M+1):
6          if d[i] >= km:
7              stop.append(i)
8              km = n
9          km -= d[i]
10     stop.append(M+1)
11     return stop
```

```
1  from gas import gas_stations
2
3  print gas_stations(6, [65, 23, 45, 62, 12, 56, 26], 150)
```

```
[0, 3, 6, 7]
```

Como es habitual con la estrategia voraz, hemos diseñado un algoritmo muy rápido: lineal con el número de gasolineras, es decir, $\Theta(M)$.

Hemos de demostrar que proporciona la solución óptima para cualquier instancia del problema. Sean $x = \{x_0, x_1, x_2, \ldots, x_k\}$ los puntos de parada devueltos por el algoritmo, y sea $y = \{y_0, y_1, y_2, \ldots, y_{k'}\}$ otro posible conjunto solución. Si N es el número total de kilómetros entre la ciudad origen y la ciudad destino, sea D_i la distancia recorrida en la ruta x hasta la i-ésima gasolinera:

$$D_i = \sum_{0 \le j < i} d[j].$$

En la ciudad destino se tiene que $D_{M+1} = N$. Hemos de demostrar que $k \le k'$, ya que lo que se desea minimizar es el número de paradas a realizar. Para ello, basta con demostrar que $x_j \ge y_j$ para todo j. En primer lugar, como ambas soluciones son distintas, sea j el primer índice tal que $x_j \ne y_j$. Sin perder generalidad, supondremos que $j = 1$, puesto que $x_0 = y_0 = 0$ (el camión parte de la ciudad origen).

- Por el funcionamiento del algoritmo, si $x_1 \neq y_1$, entonces $x_1 > y_1$, pues x_1 es la gasolinera más alejada a donde se podía llegar en la primera ruta sin repostar.

- Además, también se tiene que $x_2 \geq y_2$, pues x_2 es la gasolinera más alejada a donde se podía llegar sin repostar desde x_1. Para demostrar este hecho, supongamos, para llegar a una contradicción, que y_2 es estrictamente mayor que x_2. Si es así, entre y_1 e y_2 no puede haber más de n kilómetros: $D_{y_2} - D_{y_1} \leq n$. Como teníamos que $x_1 > y_1$ y, por tanto, $D_{x_1} > D_{y_1}$, también se cumplirá que $D_{y_2} - D_{x_1} \leq n$. Pero entonces el algoritmo no habría escogido a x_2 como siguiente gasolinera a x_1 en la solución, habría escogido la gasolinera y_2, pues el algoritmo elige la gasolinera más alejada siempre que no sobrepase la distancia n. Por tanto, no es posible que $y_2 > x_2$.

Repitiendo el proceso, vamos obteniendo en cada paso que $x_j \geq y_j$ para todo valor de j hasta llegar a la ciudad destino, lo que demuestra la hipótesis.

7.6. Código Huffman

La compresión de información es un campo de estudio de enorme interés y que presenta numerosas aplicaciones prácticas: aprovechamiento de memoria secundaria, menor consumo de ancho de banda en comunicaciones, etc. El código Huffman es un código binario de longitud variable que trata de representar los datos más frecuentes con el menor número de bits posible, así que puede considerarse una técnica de compresión de información. El código Huffman permite transmitir/almacenar mensajes con una reducción de la carga de bits que oscila entre el 20 % y el 90 %.

El fundamento del código Huffman es sencillo y un par de ejemplos permiten captar su idea básica.

- Supongamos que el sistema de control de un dispositivo nos informa constantemente de su estado con uno de tres valores: «alto», «medio» o «bajo». El sistema está en estado «medio» el 80 % del tiempo, en «alto» un 15 % y en «bajo» un 5 %. Podemos codificar cada estado con dos bits: `00` para «alto», `01` para «medio» y `10` para «bajo». Un mensaje con n datos de estado requerirá $2n$ bits. Pero esta otra codificación requiere menos bits: `0` codifica «medio»; `10`, «alto»; y `11`, «bajo». Así, un mensaje con n datos requiere un promedio de $(0.8 \cdot 1 + 0.15 \cdot 2 + 0.05 \cdot 2)n = 1.2n$ bits.

- En español hay 25 letras. Podemos codificar cualquier texto sin signos de puntuación (ni diferencias entre mayúsculas y minúsculas) con 5 bits por letra. Pero si tenemos en cuenta la frecuencia de aparición de las letras en español, su número puede reducirse a un promedio de 3.97 bits por carácter. La letra «e», por ejemplo, es la más común de todas (su frecuencia relativa ronda el 16.75 %), así que conviene representarla con menor número de bits que la letra «z», mucho más infrecuente (apenas un 0.15 %).

El código Morse se diseñó siguiendo un principio similar. La letra «e», la más frecuente en inglés, por ejemplo, se representa con un simple punto; letras de uso infrecuente requieren cuatro signos, como la «x», que se codifica con «`-..-`». Ojo con el código Morse, pues no es un código binario, sino ternario: además del punto y la raya hace uso del silencio. El silencio es necesario porque no se trata de un código prefijo.

Antes de presentar esta técnica de codificación hemos de introducir algunos conceptos. El primero es el concepto de **alfabeto**. Un alfabeto Σ es un conjunto finito de símbolos, como el conjunto de letras minúsculas del español o el conjunto de las palabras ortográficamente válidas en español, por poner un par de ejemplos. El segundo concepto es el de **código prefijo**. Un código binario que codifica un alfabeto es prefijo si ningún símbolo del alfabeto está codificado con una secuencia de bits que es un prefijo de la secuencia de bits que codifica a cualquier otro símbolo. La tabla 7.2 muestra un código prefijo para el conjunto de símbolos $\{a, b, c, d, e\}$.

a	*b*	*c*	*d*	*e*
`101`	`0`	`1001`	`1101`	`1100`

Tabla 7.2: Un ejemplo de código prefijo para un alfabeto de cinco símbolos. Ninguna de la secuencias de bits es un prefijo de otra.

Una ventaja de los códigos prefijo es la eficiencia con que permiten descodificar una secuencia de bits con un mensaje: basta con ir leyendo la secuencia de izquierda a derecha hasta detectar una secuencia de la tabla. Como ninguna es prefijo de otra, no hay ambigüedad posible. La secuencia `01011101` se divide en subsecuencias individuales como `0·101·1101` y codifica el mensaje *bad*. La detección de los prefijos se puede realizar eficientemente si el código se representa con un árbol binario. La figura 7.4 muestra el árbol binario asociado al código prefijo de la tabla 7.2. Descodificar consiste en leer la secuencia de bits y, partiendo de la raíz del árbol, seleccionar el hijo izquierdo cuando se lee un cero y el hijo derecho cuando se lee un uno hasta llegar a un nodo hoja. El nodo hoja alcanzado está etiquetado con el símbolo que corresponde a los bits leídos. Tras alcanzar un nodo hoja se parte nuevamente de la raíz del árbol y se siguen leyendo bits.

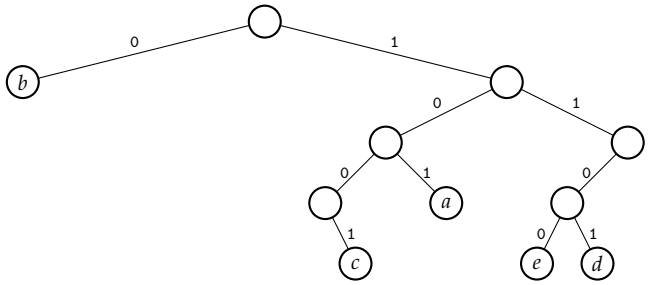


Figura 7.4: Árbol binario para la descodificación eficiente del código prefijo de la tabla 7.2.

Como se puede observar, los símbolos de un código prefijo no se codifican con secuencias de bits de idéntica longitud. Podemos aprovechar esta propiedad para codificar la información de forma óptima etiquetando convenientemente las hojas del árbol con símbolos del alfabeto. La idea es bien sencilla: codificar con las secuencias de bits más cortas los símbolos que aparecen con mayor frecuencia.

Esta técnica de compresión fue presentada por David A. Huffman en el trabajo «A method for the construction of minimum-redundancy codes», publicado en los *Proceedings of the IRE*, 40 (9), pp. 1098–1101, 1952. Huffman la ideó en 1951, cuando era estudiante del MIT. Su profesor, Robert M. Fano, asignó como tarea resolver el problema de hallar la codificación binaria más eficiente y Huffman dio con una solución que mejoraba una propuesta anterior conocida como codificación de Shannon-Fano.

Existe un estándar para comunicación entre máquinas de fax basado una variante del código de Huffman («Modified Huffman» o MH) para transmitir imágenes en blanco y negro. Cada línea se codifica con una serie de números que indican el número de píxeles de un mismo color (sólo hay dos colores, blanco y negro, y se empieza por el número de píxeles blancos). Esta secuencia de números contiene, normalmente, muchos valores repetidos, por lo que es susceptible de comprimirse eficientemente con el método de Huffman.

El código Huffman es una técnica de compresión sin pérdidas: es posible codificar un mensaje y obtener exactamente el mismo al descodificar. Los compresores con pérdidas no permiten recuperar el mensaje original, pero sí una aproximación razonable al mismo. Qué se entiende por razonable depende de la aplicación. MP3 o JPEG son métodos de compresión con pérdidas para audio e imagen respectivamente. El mensaje descodificado, aunque diferente del original, puede resultar perceptualmente indistinguible para un humano.

Hay otro algoritmo voraz y muy popular para compresión sin pérdida de información: el algoritmo de Lempel-Ziv (y su variante de Lempel-Ziv-Welch). El algoritmo de Lempel-Ziv es capaz de construir el código durante la lectura de los datos, sin necesidad de tabla de frecuencias para los símbolos. Otra peculiaridad de Lempel-Ziv es que no asume una talla fija en los símbolos: si le conviene, puede considerar que dos o más caracteres deben codificarse como un solo símbolo.

Sea Σ el alfabeto y sea T un árbol binario con $|\Sigma|$ hojas etiquetadas cada una con un símbolo de Σ. Para todo $a \in \Sigma$ denotaremos con $freq(a)$ la frecuencia (relativa o absoluta) del símbolo a. (Si se considera la frecuencia relativa en tanto por uno, $freq(a)$ es la probabilidad asociada a la aparición del carácter a.) Con $d_T(a)$ denotaremos la profundidad de la hoja de T etiquetada con el símbolo a, considerando que la raíz del árbol tiene profundidad 0. El número de bits esperados en la codificación de los símbolos del alfabeto con un código binario es:

$$bits(T) = \sum_{a \in \Sigma} d_T(a) \cdot \frac{freq(a)}{\sum_{b \in \Sigma} freq(b)}.$$

Ésta es la función objetivo: de entre todos los códigos prefijo nos interesa uno que proporcione el menor valor de $bits(T)$.

El árbol que hace mínimo el valor de *bits* hace mínimo, también, el valor de esta otra función, algo más sencilla:

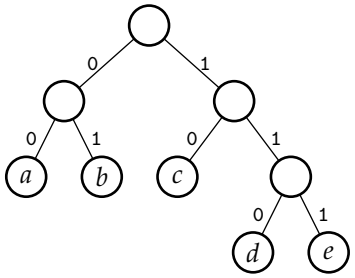
$$f(T) = \sum_{a \in \Sigma} d_T(a) \cdot freq(a).$$

La razón es obvia: el denominado $\sum_{b \in \Sigma} freq(b)$ es un valor constante. Usaremos f, pues, como función objetivo. Un código diseñado a partir de un árbol que minimice $f(T)$ recibe el nombre de código Huffman. En la figura 7.5 se muestra un código prefijo óptimo

para un alfabeto de 5 símbolos de aparición equiprobable y su árbol de descodificación eficiente.

a	*b*	*c*	*d*	*e*
00	01	10	110	111

(a)



(b)

Figura 7.5: (a) Un ejemplo de código prefijo óptimo para un alfabeto de cinco símbolos (para una aparición equiprobable de los 5 símbolos). El promedio de bits por símbolo es de 2.4, cuando el promedio para el código prefijo de la tabla 7.2 es de 3.2. Un código de tamaño fijo hubiese empleado 3 bits por símbolo. (b) Árbol binario para la descodificación eficiente del código prefijo óptimo.

7.6.1. Formalización

El conjunto de soluciones factibles es el conjunto de árboles binarios con $|\Sigma|$ hojas, cada una de las cuales está etiquetada con un símbolo distinto del alfabeto. De entre todos ellos nos interesa el que hace mínimo el valor de $f(T) = \sum_{a \in \Sigma} d_T(a) \cdot freq(a)$.

La entrada del problema es una tabla que asocia a cada símbolo del alfabeto que se quiere codificar su frecuencia de aparición en los mensajes que deseamos codificar. Podemos considerar que la tabla es un conjunto de pares $(frecuencia, símbolo)$.

7.6.2. Una solución voraz

Ilustraremos el proceso ideado por D. A. Huffman con un ejemplo: el diseño de un código óptimo para un alfabeto de 5 símbolos con las frecuencias relativas indicadas en la tabla 7.3.

Símbolo	*a*	*b*	*c*	*d*	*e*
Frecuencia (%)	30	25	15	20	10

Tabla 7.3: Tabla de frecuencias relativas (en tanto por cien) de un alfabeto de cinco símbolos.

La figura 7.6 muestra el proceso paso a paso. Con el conjunto de pares formamos un conjunto de árboles de un solo nodo etiquetado con un par (frecuencia, símbolo) (figura 7.6 (a)). Ordenamos el conjunto de árboles por orden de frecuencia creciente (figura 7.6 (b)). Seleccionamos los dos elementos menos frecuentes y formamos con ellos un nuevo árbol cuya raíz se etiqueta con la suma de frecuencias de sus dos hijos. El nuevo árbol se dispone en la posición que le corresponde por su valor de frecuencia (figura 7.6 (c)). El procedimiento se repite hasta que el conjunto de árboles tiene un solo elemento: la solución (figura 7.6 (f)). El árbol está etiquetado con pares (frecuencia, símbolo). La componente frecuencial es superflua a la hora de obtener el código y puede eliminarse. No obstante, nosotros la mantendremos en la solución.

Representaremos los árboles como listas de listas. Vamos a presentar ahora una primera versión (ineficiente) que sólo devuelve el árbol. Esta versión representa el bosque

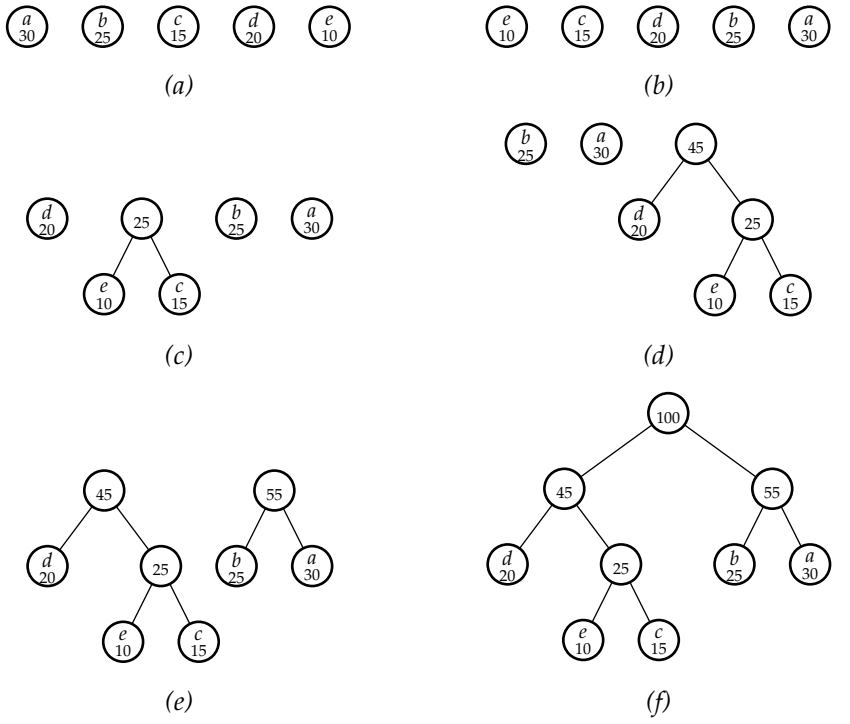


Figura 7.6: Traza del proceso de construcción del árbol sobre el que se diseña el código Huffman para el alfabeto de la tabla 7.3.

con una lista T desordenada. Con cada iteración se buscan y eliminan de T los dos elementos de menor frecuencia. Con ellos se forma el nuevo árbol que ingresa en T:

huffman_code.py

```
1  def huffman_tree1(freq):
2      T = [ [(freq[symbol], symbol)] for symbol in freq ]
3      while len(T) > 1:
4          left_tree = min(T)
5          T.remove(left_tree)
6          right_tree = min(T)
7          T.remove(right_tree)
8          new_tree = [ (left_tree[0][0]+right_tree[0][0], ), left_tree, right_tree ]
9          T.append( new_tree )
10     return T[0]
```

```
1  from huffman_code import huffman_tree1
2
3  print huffman_tree1({'a':30, 'b':25, 'c':15, 'd':20, 'e':10})
```

```
[(100,), [(45,), [(20, 'd')], [(25,), [(10, 'e')], [(15, 'c')]]], [(55,), [(25
, 'b')], [(30, 'a')]]]
```

Efectivamente, hemos obtenido el árbol de la figura 7.6 (f). Nos queda obtener el código Huffman a partir del árbol:

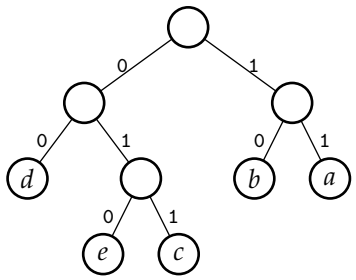
huffman_code.py *(cont.)*

```
13 def huffman_code(tree, code=None, prefix=''):
14     if code == None: code = {}
15     if len(tree[0]) > 1:
16         code[tree[0][1]] = prefix
17     else:
18         huffman_code(tree[1], code, prefix + '0')
19         huffman_code(tree[2], code, prefix + '1')
20     return code
```

```
1 from huffman_code import huffman_tree1, huffman_code
2
3 print huffman_code(huffman_tree1(freq = {'a':30, 'b':25, 'c':15, 'd':20, 'e':10}))
```

```
{'a': '11', 'c': '011', 'b': '10', 'e': '010', 'd': '00'}
```

La figura 7.7 muestra el árbol con las ramas etiquetadas con bits y el código Huffman para el programa del ejemplo.



(a)

a	*b*	*c*	*d*	*e*
11	10	011	00	010

(b)

Figura 7.7: (a) Etiquetado de las aristas del árbol óptimo de la figura 7.6. (b) Código Huffman.

7.6.3. Corrección del algoritmo

Empezaremos por extender el dominio de *freq* a los nodos de los árboles: $freq(x)$, siendo x el nodo de un árbol, es la suma de $freq(a)$ para todo símbolo a que etiqueta una hoja de x. Sea F_k el bosque formado tras la iteración k-ésima del bucle y sea F_0 el bosque inicial (un árbol por símbolo). Diremos que un bosque es consistente con un árbol T si es posible construir éste mediante uniones de árboles del bosque como las realizadas por el algoritmo. Vamos a demostrar que el algoritmo propuesto siempre gestiona un bosque compatible con $\hat{T}$, el árbol óptimo. Como el último bosque considerado es un árbol, coincidirá con $\hat{T}$.

Está claro que F_0 es consistente con $\hat{T}$, pues con él se puede formar cualquier árbol binario cuyas hojas sean símbolos del alfabeto. Si el algoritmo no construyó el árbol óptimo, en alguna iteración, digamos la $k+1$, construyó un bosque que no es consistente con $\hat{T}$ a partir de un bosque, F_k, que sí lo era.

Sean x e y los nodos que seleccionó el algoritmo para que tuvieran un padre común en la iteración $k+1$. Nótese que x e y no pueden ser hermanos en $\hat{T}$, pues en tal caso F_{k+1}

sería consistente con $\hat{T}$. Sean p_x y p_y los padres de x e y en $\hat{T}$ y sean h_x y h_y sus hermanos, como se muestra en la figura 7.8 (a). Sin pérdida de generalidad, supongamos que p_x está tan lejos de la raíz o más que p_y y que $\delta \geq 0$ es la diferencia entre sus alturas (si no es así, basta con intercambiar x e y).

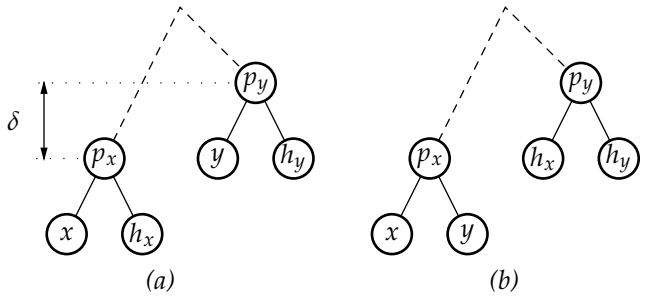


Figura 7.8: Diagrama para la demostración de corrección del algoritmo de cálculo del árbol óptimo. (a) Parte del árbol óptimo $\hat{T}$. (b) Árbol que suponemos (incorrectamente) que forma el algoritmo en la iteración $k+1$.

Podemos construir un árbol T' a partir de éste: intercambiamos h_x e y, como se muestra en la figura 7.8 (b). Como x e y son hermanos en el nuevo árbol, el bosque F_{k+1} es consistente con él. En el nuevo árbol, y está δ niveles más lejos de la raíz que en el óptimo, así que

$$f(T') - f(\hat{T}) = \delta freq(y) - \delta freq(h_x) = \delta(freq(y) - freq(h_x)).$$

Por el algoritmo sabemos que x e y eran los nodos con menor frecuencia en F_k, así que $freq(h_x) \geq freq(y)$. Y como δ es positivo, tenemos:

$$f(T') \leq f(\hat{T}).$$

O sea, T' es de coste menor o igual que el del óptimo. Si es de coste menor, esta afirmación constituye una contradicción. Si el coste es idéntico, T' es un árbol óptimo y, por tanto, no se ha perdido la compatibilidad del bosque con un árbol óptimo. Conclusión: el algoritmo siempre trabaja con un bosque compatible con $\hat{T}$. Al final, el bosque es el propio $\hat{T}$.

7.6.4. Análisis de complejidad (y un refinamiento del algoritmo de construcción del árbol)

El programa se ejecuta en tiempo $\Theta(n^2)$, siendo n la talla del alfabeto: se producen $\Theta(n)$ iteraciones del bucle y cada iteración es $\Theta(n)$, pues hemos de buscar los dos elementos de menor peso, borrarlos y añadir a T un nuevo elemento. Mantener la lista ordenada hubiese requerido el mismo esfuerzo computacional.

La estructura de datos escogida para T (una simple lista) no es la más apropiada: extraer los dos elementos de menor frecuencia e insertar uno nuevo requiere ejecutar $\Theta(n)$ pasos. Una cola de prioridad implementada con un min-heap permite realizar estas operaciones en tiempo $O(\lg n)$. Ésa es la clave que conduce a un algoritmo $O(n \lg n)$:

huffman_code.py *(cont.)*

```
22 from heap import MinHeap
23
24 def huffman_tree(freq):
25     T = MinHeap([ [(freq[symbol], symbol)] for symbol in freq ])
26     while len(T) > 1:
27         left_tree = T.extract_min()
28         right_tree = T.extract_min()
29         T.insert( [ (left_tree[0][0]+right_tree[0][0], ), left_tree, right_tree ] )
30     return T.min()
```

EJERCICIOS

7-20 Diseña:

a) Una función que, a partir de un diccionario con el código Huffman de un alfabeto y una cadena, devuelva la secuencia de bits que la codifica (codificación).

b) Una función que, a partir de un diccionario con el código Huffman de un alfabeto y una secuencia de bits, devuelva la cadena correspondiente (descodificación).

La codificación/descodificación debe efectuarse en tiempo $O(m)$ siendo m el número de bits del mensaje codificado. (Es posible que necesites efectuar un preproceso para convertir el diccionario en una estructura de datos adecuada. En tal caso, el preproceso deberá efectuarse en tiempo proporcional al número total de bits del código Huffman.)

7-21 La siguiente tabla muestra la frecuencia de aparición (en tanto por cien) de cada una de las letras del español.

letras_espanyol.py

```
1 freq = {'A': 11.96, 'B': 0.92, 'C': 2.92, 'D': 6.87, 'E': 16.78, 'F': 0.53, 'G': 0.74,
2         'H': 0.90,  'I': 4.15, 'J': 0.31, 'K': 0.01, 'L': 8.37,  'M': 2.12, 'N': 7.01,
3         'Ñ': 0.30,  'O': 8.69, 'P': 2.78, 'Q': 1.53, 'R': 4.94,  'S': 7.88, 'T': 3.31,
4         'U': 4.80,  'V': 0.40, 'W': 0.01, 'X': 0.07, 'Y': 1.54,  'Z': 0.16 }
```

Encuentra un código Huffman para la transmisión de mensajes escritos en español usando sólo letras mayúsculas. ¿Qué longitud cabe esperar en un mensaje de n letras?

(Observa que no consideramos letras acentuadas o espacios en blanco ni diferenciamos entre minúsculas o mayúsculas, por lo que el texto que codifiquemos no las puede contener. Por otra parte, para obtener una buena compresión, el texto que deseamos codificar debe presentar caracteres con frecuencias relativas similares a las de la tabla. Si diseñamos un código Huffman para textos en español, su comportamiento será mejor al codificar texto en español que texto en inglés.)

7-22 Completa la tabla de frecuencia de apariciones de letras del español para tener en cuenta que mayúsculas y minúsculas (acentuadas o no) presentan diferentes frecuencias de aparición en la lengua escrita. Obtén tú mismo la frecuencia de aparición de letras mayúsculas y minúsculas (acentuadas o no) a partir de texto en castellano. (Internet es una fuente de recursos textuales genial. Recurre a ella.) Usa tu tabla de frecuencias para obtener un código Huffman y úsalo para codificar textos.

¿Qué cantidad de bits por símbolo usas, en promedio, para transmitir un texto? Calcula el valor promedio que se deduce de la tabla y calcula empíricamente el valor promedio al codificar texto real.

7-23 Repite el experimento anterior para el idioma inglés, cuya tabla de frecuencias te adjuntamos en el siguiente fichero:

letras_ingles.py

```
freq = {'A': 8.167, 'B': 1.492, 'C': 2.782, 'D': 4.253, 'E': 12.702, 'F': 2.228,
        'G': 2.015, 'H': 6.094, 'I': 6.966, 'J': 0.153, 'K': 0.772,  'L': 4.025,
        'M': 2.406, 'N': 6.749, 'O': 7.507, 'P': 1.929, 'Q': 0.095,  'R': 5.987,
        'S': 6.327, 'T': 9.056, 'U': 2.758, 'V': 0.978, 'W': 2.360,  'X': 0.150,
        'Y': 1.974, 'Z': 0.074 }
```

Usa el código Huffman para el inglés con texto en español (ojo con la eñe). ¿Que cantidad de bits usas, en promedio, para codificar un texto de n letras?

7-24 Nos han contratado en un archivo del censo de principios de siglo. Los datos de cada persona vienen en una ficha de papel y agrupados por ciudades. Cada grupo de fichas está ordenado alfabéticamente. Nos piden formar un único grupo con todas las fichas ordenadas alfabéticamente. Nuestro proceso consiste en tomar cada vez dos grupos de fichas y fundirlas en un solo grupo (del mismo modo que hacíamos en la operación *merge* de *mergesort*). ¿En qué orden hemos de ir tomando los grupos de fichas para minimizar el tiempo necesario para obtener un solo grupo?

7-25 Diseña un algoritmo de compresión basado en la confección de códigos de Huffman ternarios, es decir, usando los valores 0, 1 y 2.

7-26 Transmitir un texto con el código Huffman requiere que el emisor lea el texto para generar el árbol, transmita el árbol y, después, transmita el texto codificado. Existen técnicas para ir construyendo un árbol de codificación durante la primera lectura del texto, que puede simultanearse con la transmisión del mismo. No es necesario transmitir explícitamente el árbol: el receptor puede ir construyéndolo sobre la marcha. Encontrarás en la bibliografía algoritmos de codificación que siguen esta estrategia. Implementa uno de ellos y compara la tasa de compresión conseguida por el código Huffman «estándar» y el adaptativo.

7.7. Árbol de recubrimiento mínimo

En la sección 4.2 presentamos el concepto de árbol de recubrimiento en un grafo no dirigido y un algoritmo para encontrar uno cualquiera. Un **árbol de recubrimiento mínimo** o **MST** (acrónimo del inglés «Minimum Spanning Tree») de un grafo no dirigido $G = (V, E)$ y ponderado por una función $d : E \to \mathbb{R}^{\geq 0}$ es aquel árbol de recubrimiento $T \subseteq E$ cuya suma de pesos

$$D(T) = \sum_{(u,v)\in T} d(u,v)$$

es menor o igual que la de cualquier otro árbol de recubrimiento. La figura 7.9 (a) muestra un grafo no dirigido y ponderado. La figura 7.9 (b) muestra, en trazo grueso, un MST cuyo coste es de 45.

Nótese, pues, que sería más apropiado referirnos al árbol de recubrimiento *con coste* mínimo, pero en la literatura se le conoce como «árbol de recubrimiento mínimo».

Encontrar el MST presenta interés en el diseño de redes de transporte o comunicaciones: permite interconectar todos los vértices de un grafo con el menor coste posible (menor número de kilómetros asfaltados en el caso de carreteras o menor número

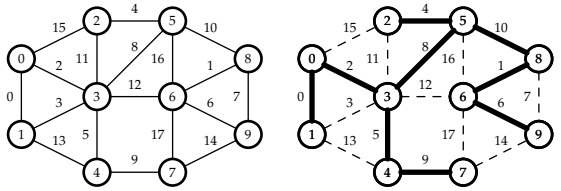


Figura 7.9: (a) Grafo no dirigido y ponderado. (b) Árbol de recubrimiento mínimo (MST) sobre el grafo (aristas en trazo continuo).

de kilómetros de cable en el caso de redes para la transmisión de datos o voz). La figura 7.10 (a) muestra una representación simplificada de un mapa de carreteras de la península ibérica. Si tuviésemos que diseñar una red de comunicaciones por cable que permita comunicar cualquier par de ciudades y el cable tuviera que discurrir siempre junta a una carretera, el árbol de recubrimiento mínimo que se muestra en la figura 7.10 (b) nos indica dónde tirar cable para que la cantidad total empleada sea mínima.

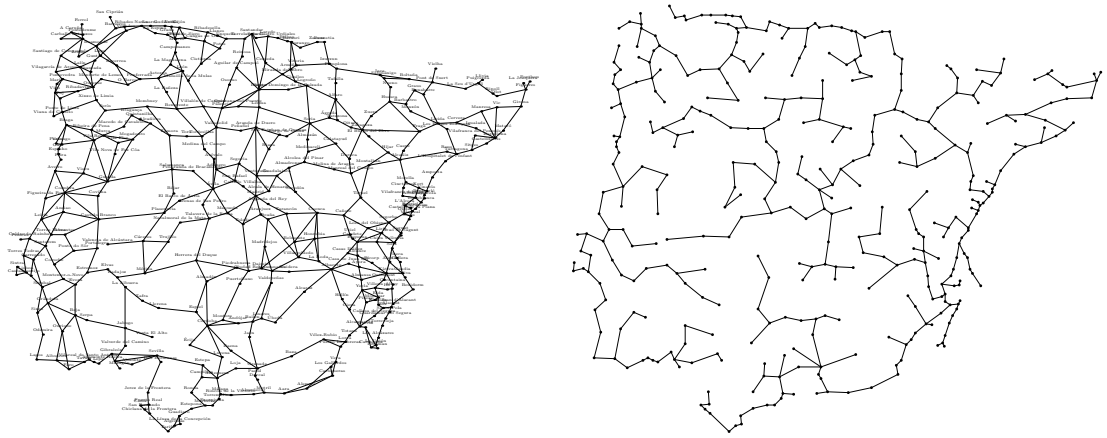


Figura 7.10: (a) Mapa de carreteras de la península ibérica (cuyos datos están disponibles en el fichero `iberia.py`). (b) Árbol de recubrimiento mínimo.

Vamos a estudiar dos algoritmos voraces para el cálculo del MST: el algoritmo de Kruskal y el algoritmo de Prim.

7.7.1. El algoritmo de Kruskal

El algoritmo de Kruskal empieza por crear un conjunto de aristas vacío, T, con el que representa un bosque que, al final, contendrá un único árbol: el MST. Inicialmente el bosque está formado por $|V|$ árboles de un solo vértice (y ninguna arista): uno por cada vértice del grafo. A continuación, el algoritmo considera todas y cada una de las aristas del grafo en orden de menor a mayor peso. Si la arista considerada, (u, v), no forma un ciclo en el grafo $G' = (V, T)$, se añade a T; en caso contrario, se descarta. Si ingresa (u, v) en T, se está formando un nuevo árbol a partir de otros dos, ya que u y v eran vértices de árboles distintos. La figura 7.11 muestra una traza del algoritmo descrito sobre un grafo.

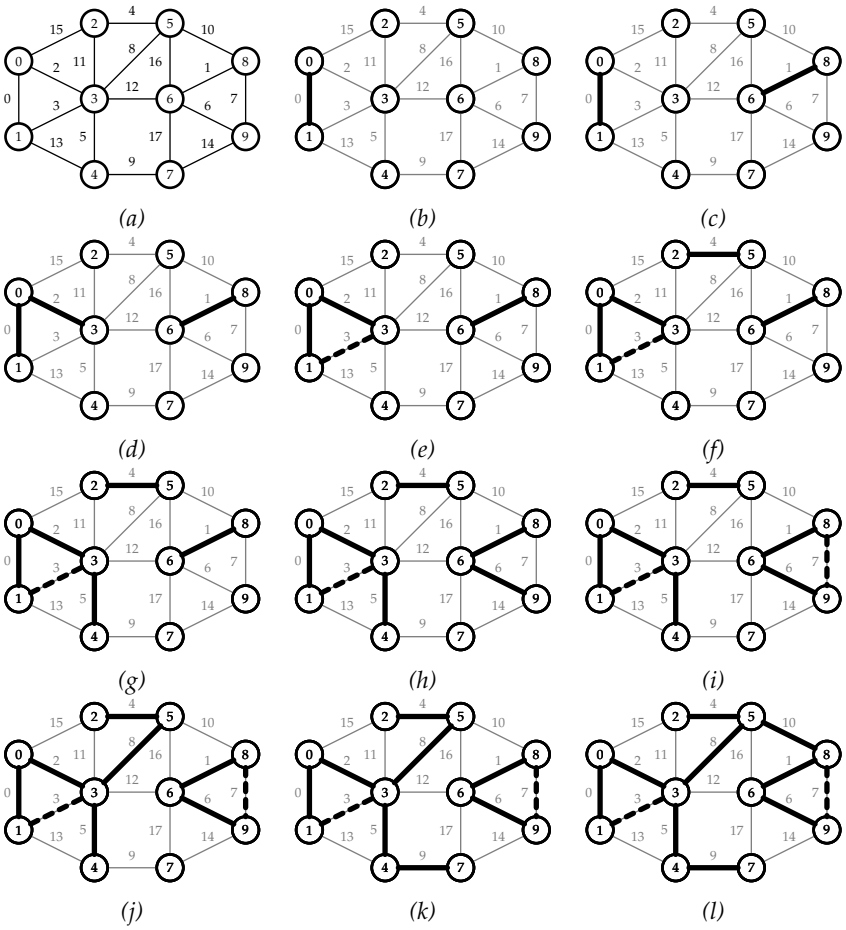


Figura 7.11: Traza del algoritmo de Kruskal. Las aristas de trazo continuo y grueso forman parte de T. Las aristas de trazo grueso y discontinuo han sido consideradas y descartadas por crear ciclos.

La detección de ciclos en T tiene un gran impacto en la eficiencia del algoritmo, pues se efectúa hasta $|E|$ veces. Afortunadamente, un MFSET permite representar los conjuntos de vértices en cada árbol de T y resolver con eficiencia preguntas de pertenencia y fusiones. Para saber si una arista (u, v) crea un ciclo, preguntaremos por el conjunto al que pertenecen u y v: la arista no creará un ciclo si y sólo si son diferentes. Y cuando ingrese la arista en T, fundiremos los conjuntos a los que pertenecen u y v. Podremos efectuar hasta $|E|$ preguntas de pertenencia y hasta $|V| - 1$ fusiones en tiempo amortizado $O(|E|)$.

kruskal.py

```
1 from mfset import MFset
2
3 def Kruskal(G, d):
4   forest = MFset(G.V)
```

```
5   MST = set([])
6   for (weight, (u,v)) in sorted([(d(u,v), (u,v)) for (u,v) in G.E]):
7     if forest.find(u) != forest.find(v):
8       MST.add((u,v))
9       forest.merge(u, v)
10      if len(MST) == len(G.V)-1: break
11  return MST
```

```
1  from graph import Graph, WeightingFunction
2  from kruskal import Kruskal
3
4  d = WeightingFunction({(0,1): 0, (0,2): 15, (0,3): 2, (1,3): 3, (1,4): 13, (2,3): 11,
5                         (2,5): 4, (3,4): 5, (3,5): 8, (3,6): 12, (4,7): 9, (5,6): 16,
6                         (5,8):10, (6,7): 17, (6,8): 1, (6,9): 6, (7,9): 14, (8,9): 7},
7                         symmetrical=True)
8  MST = Kruskal(Graph(E=d.keys()), d)
9  print 'MST: %s con peso %s.' % (MST, sum (d(u,v) for (u,v) in MST))
```

```
MST: set([(0, 1), (6, 9), (6, 8), (4, 7), (2, 5), (3, 4), (0, 3), (5, 8), (3,
5)]) con peso 45.
```

Corrección del algoritmo

Es evidente que el algoritmo acaba encontrando un árbol de recubrimiento si el grafo es conexo, pues considera todas las aristas y va uniendo los árboles del bosque (que son disjuntos) dos a dos hasta formar un sólo árbol. Que el árbol encontrado sea de coste mínimo no resulta evidente. Supondremos de momento que sólo hay un MST (y no dos o más con idéntico peso).

Lema 7.1 *Sea $G = (V, E)$ un grafo no dirigido y ponderado por una función $d : E \to \mathbb{R}$, sea X un subconjunto de V y sea $e = (u, v)$ la arista de menor peso que conecta X con $V - X$. La arista e es parte del MST.*

Demostración. Supongamos que T es un árbol de recubrimiento que no contiene a e. Vamos a demostrar que T no es el MST. En consecuencia, e deberá formar parte del MST. La arista e es un par (u, v) donde $u \in X$ y $v \in V - X$. Como T es un árbol de recubrimiento, ya conectaba X con $V - X$. Sea $e' = (u', v')$ la arista que efectuaba esta conexión. Si añadimos a T la arista e, se producirá un ciclo. El árbol $T' = (T \cup e) - e'$ elimina el ciclo, conecta a todos los vértices y es, por tanto, un árbol de recubrimiento. El peso $D(T')$ es menor que el de $D(T)$, pues $d(u, v) < d(u'v')$. Así pues, T no es el MST. □

En el caso de que pudiera haber más de un MST, hubiésemos tenido que considerar la posibilidad de que $D(T') = D(T)$, lo que implicaría $d(u, v) = d(u', v')$. En tal caso, tanto T como T' serían MST y hubiera dado igual escoger e o e', pues ambos forman parte de un MST.

Corolario 7.1 *El algoritmo de Kruskal calcula el MST.*

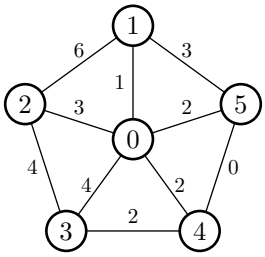
Demostración. Con cada iteración, el algoritmo escoge la arista e que conecta dos regiones disjuntas del grafo. Una de ellas es un subconjunto del MST (en realidad, las dos) y la arista seleccionada es la de menor peso de cuantas le conectan con vértices no perteneciente a ese subconjunto. Por el lema 7.1, esta arista es parte del MST. □

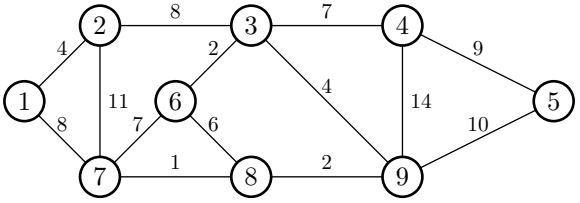
Análisis de complejidad

La complejidad temporal es $O(|V| + |E| \lg |E|)$. Ordenar el vector de aristas por orden creciente de peso es $O(|E| \lg |E|)$. El algoritmo efectúa un máximo de $|E|$ iteraciones y un máximo de $|E|$ consultas de pertenencia y de $|V| - 1$ fusiones de conjuntos sobre el MFSET. Como ya hemos visto, el coste temporal global de estas operaciones es $O(|E|)$.

EJERCICIOS

7-27 Realiza una traza del algoritmo de Kruskal para los grafos de las siguientes figuras:





7-28 Utilizando el algoritmo de Kruskal, ¿cómo se podría determinar si un grafo es conexo? ¿cómo se podrían obtener los vértices que forman cada una de las componentes conexas del grafo?

7-29 Supongamos que hemos obtenido un MST para un grafo G no dirigido, conexo y ponderado. Si a continuación se añadiera un nuevo vértice y la arista o aristas que lo conectan al resto de los vértices, ¿de qué forma se podría actualizar rápidamente el MST, sin necesidad de ejecutar nuevamente el algoritmo?

7-30 Deseamos montar un sistema de comunicaciones por cable que comunique entre sí a todas las ciudades de la península ibérica y queremos minimizar la cantidad de cable instalado.

a) Diseña un programa que indique qué pares de ciudades hemos de conectar entre sí suponiendo que es posible comunicar dos ciudades con una cantidad de cable igual a la distancia que las separa en línea recta, haya o no carretera que las una.

b) Diseña un programa que indique qué pares de ciudades hemos de conectar entre sí suponiendo que es posible comunicar dos ciudades con una cantidad de cable igual a la distancia que las separa en línea recta, pero sólo si hay carretera entre ellas.

En ambos casos, representa gráficamente el resultado.

7-31 El algoritmo de Kruskal resulta costoso sobre grafos euclídeos en el plano en los que cada vértice está conectado con todos los demás. Existe un preproceso, llamado triangulación de Delaunay, que permite reducir sensiblemente el coste del algoritmo. Averigua qué es la triangulación de Delaunay y por qué puede mejorar la eficiencia asintótica del algoritmo de Kruskal sobre grafos euclídeos.

7-32 Hay otro algoritmo voraz para el cálculo del árbol de recubrimiento mínimo: el algoritmo de Boruvka. Es similar al algoritmo de Kruskal, pero en cada paso selecciona *todas* las aristas de coste mínimo con las que cada uno de los árboles del bosque puede unirse a otro árbol. Todas las aristas seleccionadas ingresan simultáneamente en T. Implementa el algoritmo de Boruvka, demuestra su corrección y analiza su complejidad computacional.

7.7.2. El algoritmo de Prim

El algoritmo de Prim se basa en la técnica de «crecimiento del árbol» que presentamos en la sección 4.2 al abordar el problema del cálculo de un árbol de recubrimiento cualquiera. De hecho, puede verse como una modificación del algoritmo *spanning_tree* que presentamos allí. Recordemos que dicho algoritmo considera que, en cada instante, todo vértice pertenece a uno de tres conjuntos de vértices: el de los que forman parte del recubrimiento en construcción; el de los «vértices frontera», es decir, los que no forman parte del primero conjunto pero están unidos por alguna arista («arista frontera») al primer conjunto; y el formado por el resto de vértices. El algoritmo de crecimiento del árbol escoge en cada paso una arista frontera cualquiera y la integra en el árbol en construcción (con lo que éste pasa a tener una arista y un vértice más) hasta no poder escoger una arista más. El algoritmo de Prim hace eso mismo, pero no escoge una arista frontera cualquiera, sino la que más le conviene en ese instante: la que presenta menor peso de entre todas las elegibles. El algoritmo termina cuando todos los vértices forman parte del árbol. La figura 7.12 muestra gráficamente una traza de este método.

Podemos observar que un vértice frontera puede estar unido a varios del árbol con diferentes aristas frontera, pero como sólo nos interesa la de menor coste podemos quedarnos con una sola arista frontera por cada vértice frontera: la de menor peso. He aquí una primera versión del algoritmo de Prim:

prim.py

```
1  from utils import argmin
2
3  def Prim(G, d):
4      MST = set()
5      u = list(G.V)[0]
6      added = set([u])
7      in_frontier_from = dict( (v,u) for v in G.succs(u) )
8      while len(in_frontier_from) > 0:
9          (v, u) = argmin(in_frontier_from.items(), d)
10         del in_frontier_from[v]
11         added.add(v)
12         MST.add( (u,v) )
13         for w in G.succs(v):
14             if w not in added and \
15                 (w not in in_frontier_from or d(v, w) < d(in_frontier_from[w], w)):
16                 in_frontier_from[w] = v
17     return MST
```

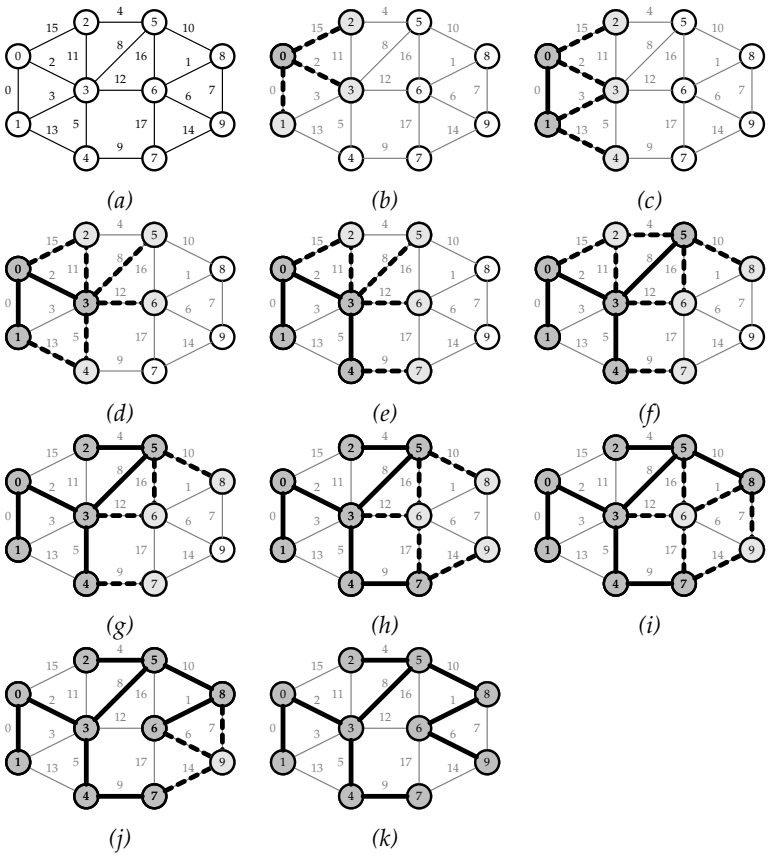


Figura 7.12: Traza del algoritmo de Prim. Los vértices de color gris oscuro pertenecen al conjunto de nodos que ya forman parte del MST. Los vértices de color gris claro son vértices frontera. Las aristas que forman parte del MST se muestran en trazo continuo y grueso y las aristas frontera en trazo discontinuo y grueso.

```
from graph import Graph, WeightingFunction
from prim1 import Prim

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)
MST = Prim(Graph(E=d.keys()), d)
print 'MST: %s con peso %s.' % (MST, sum (d(u,v) for (u,v) in MST))
```

```
MST: set([(0, 1), (4, 7), (3, 4), (6, 9), (8, 6), (0, 3), (5, 2), (5, 8), (3,
5)]) con peso 45.
```

Corrección

En tanto que concreción del método de selección en el algoritmo de crecimiento del árbol, está claro que el algoritmo descrito encuentra siempre un árbol de recubrimiento (si el grafo es conexo). Pero, ¿es el árbol de recubrimiento mínimo?

Corolario 7.2 *El algoritmo de Prim calcula el MST.*

Demostración. Con cada iteración el algoritmo escoge la arista e que conecta dos regiones disjuntas del grafo (los nodos que ya forman parte del árbol y el resto) y la arista seleccionada es la de menor peso de cuantas conectan un vértice de X con un vértice de $X - T$. Por el lema 7.1, esta arista es parte del MST. □

Análisis de coste computacional (y un refinamiento)

El coste computacional del algoritmo es $O(|V|^2)$: se realizan hasta $|V| - 1$ iteraciones del bucle **while** (con cada una se añade un vértice de V a *added*) y en cada iteración se consideran los $O(|V|)$ vértices de la frontera para seleccionar una arista frontera. Observamos, pues, que la operación crítica es la selección del mejor elemento en un conjunto de aristas frontera que contiene una arista por cada vértice frontera (la de menor peso que conecta a ese vértice frontera con algún vértice del árbol en construcción). Podemos implementar el conjunto de aristas/vértices frontera con un diccionario de prioridad: en cada iteración extraemos el vértice con arista frontera de menor peso, insertamos nuevos vértices frontera (con sus respectivas aristas) y modificamos el peso de otros vértices frontera en función de si cambia la arista frontera que lo conecta con el árbol:

prim.py *(cont.)*

```
20 from prioritydict import MinPriorityDict
21
22 def Prim_with_prioritydict(G, d):
23     MST = set()
24     u = list(G.V)[0]
25     added = set([u])
26     in_frontier_from = MinPriorityDict(dict((v,(d(u,v),u)) for v in G.succs(u)), len(G.V))
27     while len(in_frontier_from) > 0:
28         (v, (dv, u)) = in_frontier_from.extract_min()
29         added.add(v)
30         MST.add( (u,v) )
31         for w in G.succs(v):
32             if w not in added and \
33                     (w not in in_frontier_from or d(v, w) < d(frontier[w][1], w)):
34                 frontier[w] = (d(v,w), v)
35     return MST
```

Analicemos la complejidad computacional de esta nueva versión. Antes de entrar en el bucle se inicializa un diccionario de prioridad con $O(|V|)$ elementos. El bucle realiza $|V| - 1$ iteraciones. Con cada una se extrae un elemento del diccionario de prioridad (coste temporal $O(\lg |V|)$) y se insertan o actualizan $O(|V|)$ elementos. Parece, pues, que efectuemos $O(|V|^2 \lg |V|)$ operaciones. Un estudio más detallado permite observar que el número de inserciones y actualizaciones está acotado por $|E|$, así que la complejidad computacional temporal de la nueva versión es $O(|E| \lg |V|)$.

Tenemos, pues, dos variantes del algoritmo de Prim, una que requiere tiempo $O(|V|^2)$ y otra que requiere tiempo $O(|E| \lg |V|)$. En grafos dispersos, la primera versión es ineficiente, pero en grafos densos, es decir, cuando $|E|$ está próximo a $|V|^2/2$, resulta mejor.

El algoritmo de Prim fue realmente descubierto por Vojtěch Janík en 1930. Robert C. Prim lo redescubrió en 1957, cuando trabaja en los Bell Laboratories de At&T. Lo publicó ese mismo año en la revista *Bell System Technical Jorunal* bajo el título «Shortest connection networks and some generalisations». En 1959 Edsger Dijkstra descubrió nuevamente el algoritmo. Hay quien denomina a este algoritmo el «algoritmo DJP» por las iniciales de sus tres descubridores independientes.

7.7.3. ¿Kruskal o Prim?

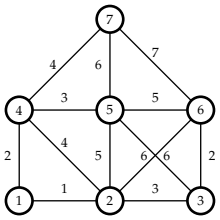
El algoritmo de Prim tiene dos versiones, una directa que se ejecuta en tiempo $O(|V|^2)$ y otra que usa un diccionario de prioridad y se ejecuta en tiempo $O(|E| \lg |V|)$. El algoritmo de Kruskal, por su parte, requiere tiempo $O(|E| \lg |E|)$.

Cuando el grafo es denso, la primera versión del algoritmo de Prim es más eficiente. Cuando el grafo es disperso, la complejidad computacional de Prim y Kruskal es comparable, pues $O(|E| \lg |E|) = O(|E| \lg |V|^2) = O(|E| \lg |V|)$.

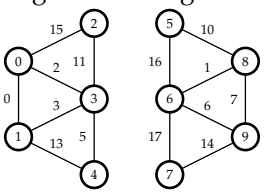
Y el algoritmo de Prim puede ejecutarse en tiempo $O(|E| + |V| \lg |V|)$ si se usa un «Heap de Fibonacci» en lugar del diccionario de prioridad. Hay que decir, no obstante, que si se dispone de las aristas ya ordenadas por valor creciente del peso, el coste del algoritmo de Kruskal se reduce, en la práctica, a $O(|E|)$.

EJERCICIOS

7-33 Haz sendas trazas de los algoritmo de Kruskal y Prim para este grafo:



7-34 ¿Qué ocurre cuando se suministra un grafo no conexo como entrada a los algoritmos de Kruskal y Prim? Haz una traza sobre el grafo de la figura.



Modifica los algoritmos, si procede, para que encuentre el bosque de recubrimiento de coste mínimo cuando se suministra un grafo no conexo.

7.8. El camino más corto de un vértice a cualquier otro en un digrafo ponderado positivo

Dado un digrafo $G = (V, E)$ cuyos arcos están ponderados por una función $d : E \to \mathbb{R}^{\geq 0}$ y dado un vértice s al que denominamos vértice inicial o fuente, queremos encontrar, para todo vértice v, el camino más corto de s a t, es decir, un camino tal que ningún otro presente menor distancia.

Este es un problema importante y con infinidad de aplicaciones prácticas. En un caso particular, cuando el digrafo es *ponderado positivo*, admite una estrategia resolutiva voraz que presentamos en esta sección. Muchos grafos que modelan objetos del mundo real son ponderados positivos, por lo que este algoritmo es aplicable en ellos.

Los caminos más cortos de s a cualquier vértice cumplen la propiedad de la subestructura óptima. Supongamos que el camino más corto de s a v pasa por un vértice u. Es evidente que el prefijo de ese camino que va de s a u es un camino más corto de s a u: si hubiera otro de menor distancia recorrida, el camino más corto de s a v se podría mejorar sustituyendo su prefijo hasta u por este otro camino. (Usamos la expresión «*un* camino más corto» en lugar de «*el* camino más corto» porque podría haber varios caminos con igual suma de pesos y ser ésta mínima.) Es interesante observar que en un digrafo ponderado *estrictamente* positivo un camino más corto de s a cualquier vértice v no puede contener ciclos: recorrer el ciclo supondrá recorrer una distancia para ir de un vértice a sí mismo y siempre será mejor no recorrerla. Si hay aristas con peso nulo, un camino más corto de s a un vértice puede tener un ciclo, pero el mismo camino sin ese ciclo es también un camino más corto. Estas dos propiedades (subestructura óptima y ausencia de ciclos) hacen que podamos construir un árbol de recubrimiento (óptimo en cierto sentido) con los caminos más cortos de s a cualquier vértice alcanzable desde s. En la figura 7.13 (a) se muestran este árbol de recubrimiento para el mapa de carreteras de la península ibérica tomando como ciudad de partida Madrid y en 7.13 (b) el que se obtiene con Castellón de la Plana como vértice inicial.

Estas observaciones son interesantes porque indican que es posible encontrar un árbol de recubrimiento (dirigido y con raíz en s) en el que la rama de s a cualquier vértice v es el camino más corto de s a v. Un árbol de recubrimiento con estas características recibirá el nombre de **árbol de caminos más cortos** con origen o raíz en s.

7.8.1. El algoritmo de Dijkstra: una primera versión

Estas reflexiones conducen a una variante del método de crecimiento del árbol: iremos construyendo el árbol de recubrimiento añadiendo en cada iteración una arista que conecta un vértice frontera al árbol ya construido. Lo importante es la definición del criterio por el que escogemos una arista (y su vértice frontera) y no otra. El algoritmo que presentamos mantiene un diccionario D que asocia a cada vértice v del grafo un peso: el del camino más corto de s a v conocido *hasta el momento*. En cada iteración de crecimiento del árbol, ingresa en el mismo el vértice frontera con menor valor de D.

(a) *(b)*

Figura 7.13: (a) Árbol de recubrimiento con los caminos más cortos de Madrid a todas las demás ciudades del mapa ibérico. (b) Ídem para Castellón de la Plana como punto de partida.

A continuación presentamos una implementación directa de esta idea que computa la distancia que recorre el camino más corto de s a cualquier otro vértice:

shortestpaths.py

```
1  def all_shortest_paths_from_source1(G, d, s, infinity=3.4e+38):
2      ST = set()
3      D = dict((v, infinity) for v in G.V)
4      D[s] = 0
5      for v in G.succs(s): D[v] = d(s,v)
6      added = set([s])
7      in_frontier_from = dict( (v, s) for v in G.succs(s) )
8      while len(in_frontier_from) > 0:
9          dmin, u, v = min( (D[v], u, v) for (v, u) in in_frontier_from.items() )
10         del in_frontier_from[v]
11         added.add(v)
12         ST.add( (u, v) )
13         for w in G.succs(v):
14             if w not in added and D[v] + d(v,w) < D[w]:
15                 D[w] = D[v] + d(v,w)
16                 in_frontier_from[w] = v
17     return ST
```

Pongamos a prueba nuestra implementación sobre el grafo de la figura 7.14.

```
1  from shortestpaths import all_shortest_paths_from_source1
2  from mallorca import Mallorca, km
3
4  ST = all_shortest_paths_from_source1(Mallorca, km, 'Andratx')
5  for (u,v) in ST: print '(%s, %s)' % (u,v),
```

```
(Inca, Alcúdia) (Llucmajor, Campos del Port) (Calvià, Palma de Mallorca) (Artà
```

```
, Capdepera) (Alcúdia, Pollença) (Palma de Mallorca, Llucmajor) (Marratxí, Inc
a) (Palma de Mallorca, Marratxí) (Andratx, Calvià) (Manacor, Artà) (Campos del
 Port, Santanyí) (Inca, Manacor) (Andratx, Sóller)
```

La figura 7.14 (b) muestra el árbol de caminos más cortos.

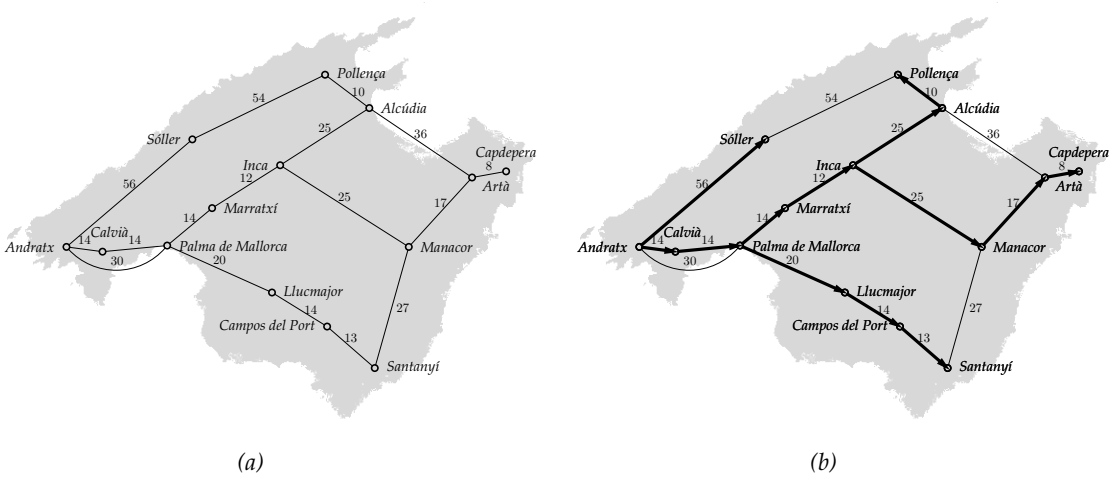


Figura 7.14: (a) Grafo ponderado que modela un mapa de carreteras entre algunas de las principales ciudades de la isla de Mallorca. (El fichero `mallorca.py` *contiene la definición del grafo en Python. La zona noroeste de la isla es muy accidentada, lo que hace que las carreteras no sigan trayectorias rectas, aunque la idealización de la representación gráfica sugiera lo contrario.) (b) En trazo grueso, aristas del árbol de caminos más cortos con origen en Andratx.*

El algoritmo que hemos presentado (y los refinamiento que presentaremos más adelante) recibe el nombre de algoritmo de Dijkstra. (En realidad, se conoce por «algoritmo de Dijkstra» la variante que permite conocer el camino más corto de s a un vértice determinado, t. Ese resultado es un subproducto del algoritmo que hemos presentado.) Las figuras 7.15, 7.16 y 7.17 muestran una traza del algoritmo para el grafo de Mallorca y la ciudad de Andratx como punto de partida.

Edsger W. Dijkstra (1930–2002) es uno de los pioneros de la algorítmica. El conocido como «algoritmo de Dijkstra» se publicó en el artículo «A note on two problems in connection with graphs», *Numerical Mathematics*, núm. 1, pp. 269–271, 1959.

7.8.2. Corrección

Teorema 7.1 *El algoritmo de Dijsktra calcula el camino más corto entre s y cualquier vértice v.*

Demostración. Por inducción sobre la talla de *added*. Comprobaremos que, para todo $u \in$ *added*, $D[u]$ es la distancia del camino más corto de s y u y que $D[v]$, para todo $v \in V -$ *added*, es la distancia del camino más corto de s a v que, antes de llegar a v, únicamente visita vértices de *added*.

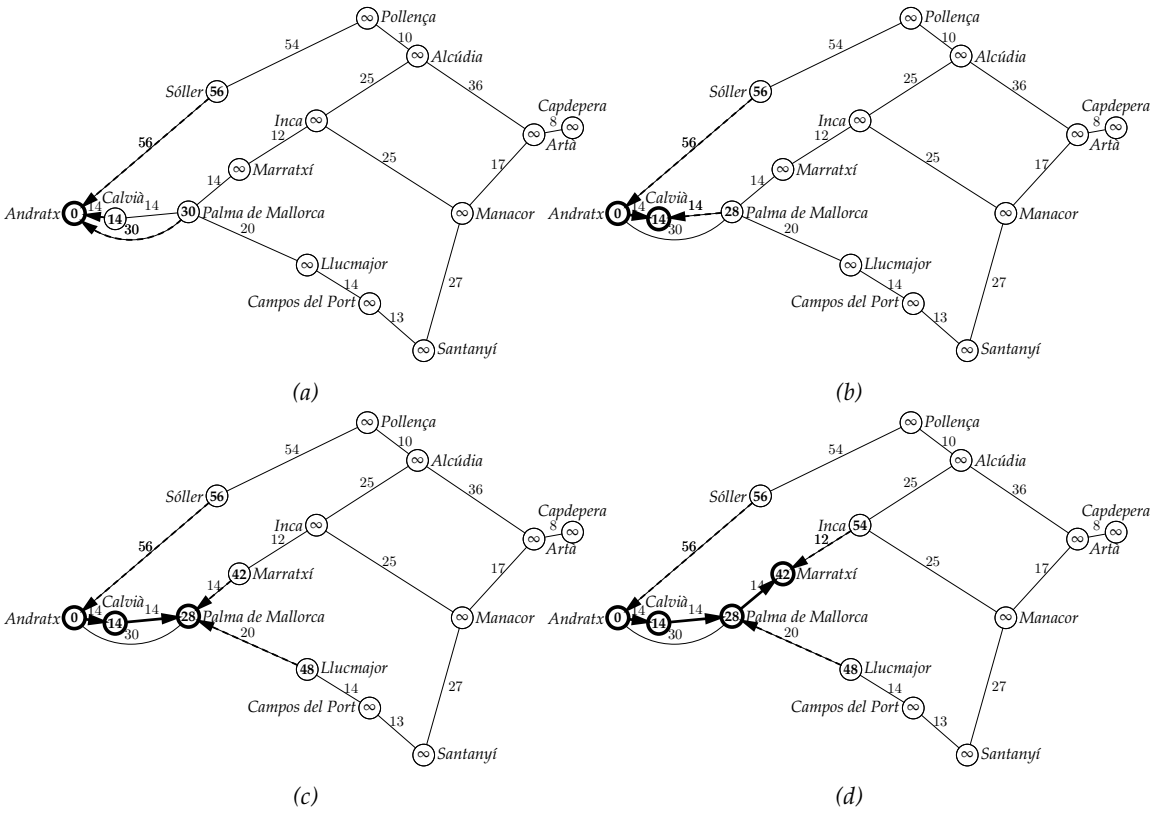


Figura 7.15: Traza del algoritmo de Dijkstra sobre el grafo de Mallorca para el cálculo de los caminos más cortos desde Andratx a cualquier otra ciudad. Los vértices de added y las aristas de ST se muestran en trazo grueso. Aunque el grafo del ejemplo es no dirigido, los arcos se almacenan con dirección (pues se va de un punto a otro siguiendo una dirección determinada). En trazo discontinuo, la información almacenada en el diccionario in_frontier_from: la clave es el destino de la flecha y su valor, el origen de la misma. En el interior de cada nodo se muestra el valor de D. La figura (a) corresponde al instante previo a ejecutarse el bucle **while**. *Cada una de las restantes figuras refleja el estado al final de una iteración de dicho bucle.*

Base de inducción. Cuando $|added| = 1$, su único vértice es s y $D[s]$ vale 0, que es la distancia del camino más corto de s a s.

Paso de inducción. Supongamos que $D[u]$ es la distancia del camino más corto de s a u para todo elemento u de *added* en un instante dado, es decir, para una talla determinada de *added*. Sea v el vértice de $V - added$ seleccionado en la línea 9 del algoritmo. Supongamos que $D[v]$ *no* es la distancia del camino más corto de s a v, es decir, que existe un camino p más corto. Dicho camino debe tener al menos un vértice diferente de v que no esté en *added*. Sea v' el primer vértice de p que no está en *added*. La figura 7.18 ilustra esta situación.

Supongamos que no hay aristas de peso nulo. Por fuerza, para que el camino presente una distancia menor que $D[v]$, la distancia de s a v' ha de ser menor que $D[v]$, ya que el tramo de p que va de v' a v añade una distancia que no puede ser negativa (no hay aristas de peso negativo). Como el prefijo de p que llega a v' está formado por vértices

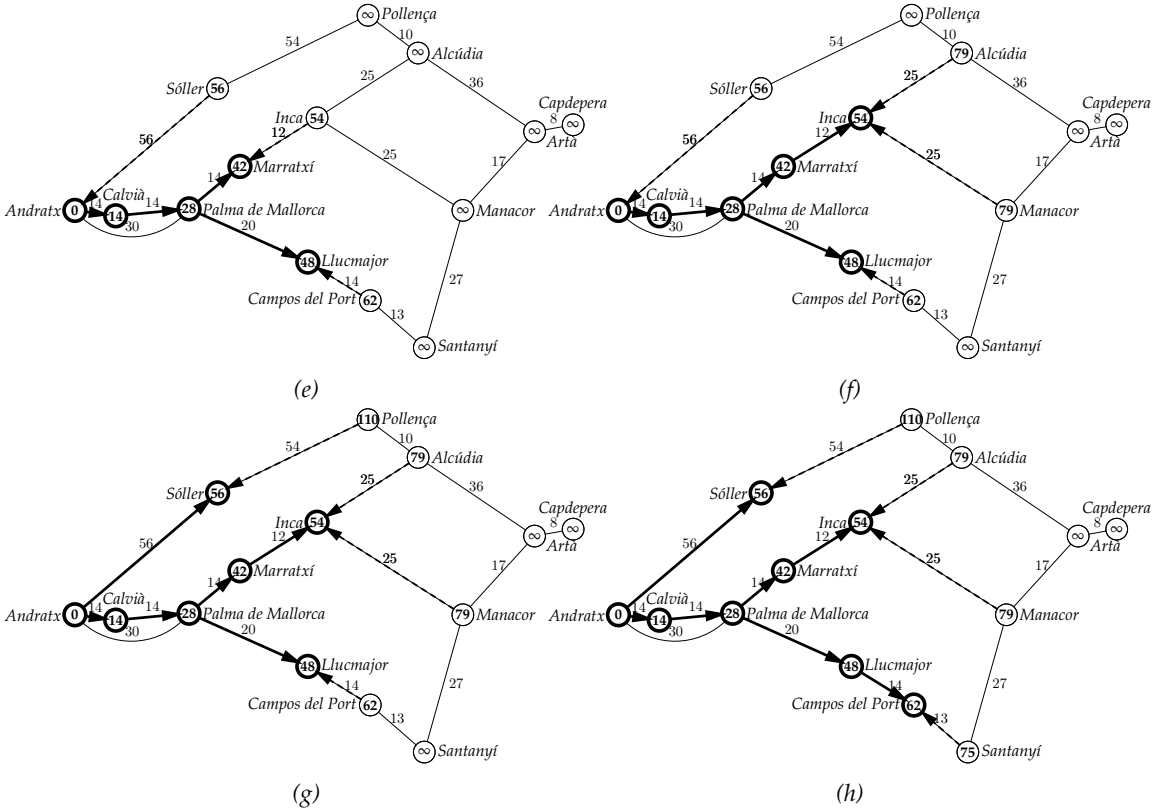


Figura 7.16: Continuación de la figura anterior.

de *added*, esto equivale a suponer que $D[v'] < D[v]$. Pero eso es una contradicción: de ser así, en la iteración actual no habríamos seleccionado el vértice v, sino el vértice v'. La conclusión es que el camino p no existe.

Si hubiera aristas de peso nulo, la única posibilidad de que p existiera es que la porción que conecta v' con v se formara con una secuencia de aristas de peso nulo. En tal caso, $D[v']$ sería igual a $D[v]$ y, por tanto, $D[v]$ correspondería a la distancia del camino más corto entre s y v, como queríamos demostrar.

Nótese que una vez ingresa en *added* un vértice v, su valor $D[v]$ no se modifica nunca más, así que mantiene la distancia mínima entre s y v.

El algoritmo finaliza cuando todos los vértices ingresan en *added*, así que $D[v]$, para todo $v \in V$, es la distancia del camino más corto de s a v. Si en ST hay un arco (u, v), se observa $D[v] = D[u] + d(u, v)$, así que el conjunto de arcos en ST representa el conjunto de caminos más cortos con origen en s. □

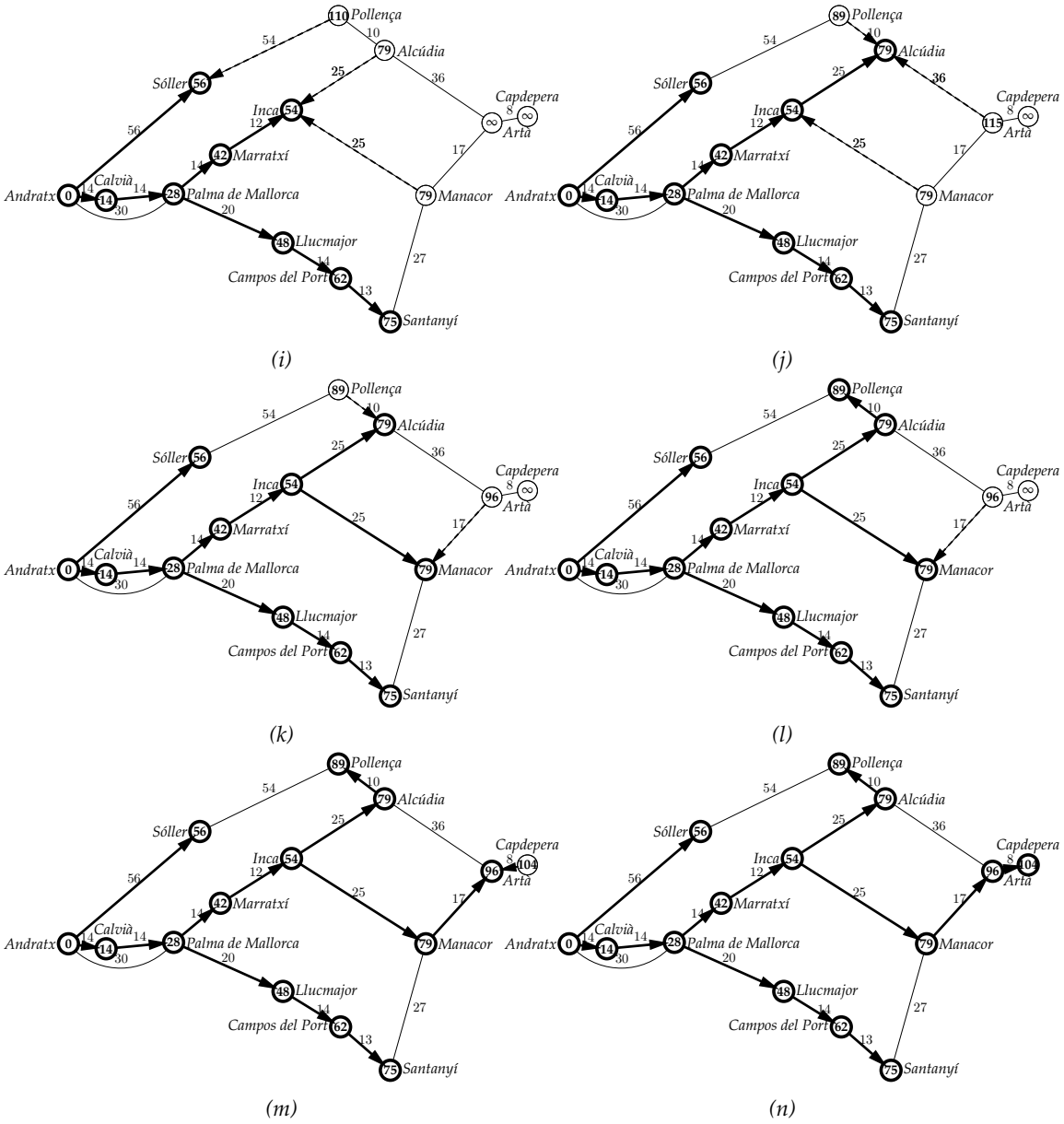


Figura 7.17: Continuación de la figura anterior.

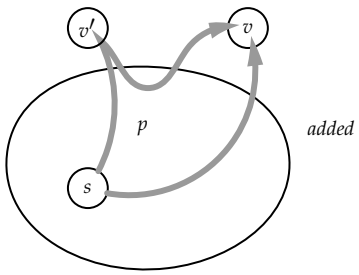


Figura 7.18: El camino p no puede ser más corto que el camino que conecta s con v pasando únicamente por vértices del conjunto added.

7.8.3. Un refinamiento: el árbol de caminos más cortos expresado con punteros hacia atrás

Devolver el árbol de caminos más cortos como conjunto de aristas soluciona el problema, pero dificulta conocer el camino más corto entre s y un vértice v cualquiera. Si deseamos conocer el camino más corto de s a v y empezamos a construirlo a partir del conjunto de aristas y el nodo s, tendremos que efectuar una búsqueda con retroceso: s puede estar conectado a varios vértices en el árbol de caminos más cortos, y no sabemos cuál conducirá a v. Es más sencillo proceder del revés: sólo hay un arco que conecte v con su predecesor en el camino de s a v, así que no hay alternativas. Y el predecesor sólo tiene un vértice que le antecede en dicho camino. Y así sucesivamente hasta llegar a s.

Ya que, por eficiencia, conviene recorrer del revés el camino que se desea explicitar, hay una representación del árbol más adecuada: la que usa punteros al padre en cada nodo del árbol. Con ella, basta con seguir la rama de ancestros de v hasta s para recuperar el camino más corto de s a v, si bien lo obtendremos en orden inverso al que recorre el camino. Los punteros al padre se conocen en la literatura por «punteros hacia atrás», ya que expresan el camino en orden inverso, apuntando en cada vértice de qué vértice proviene el camino óptimo. La figura 7.19 muestra esta representación para el árbol de caminos más cortos de Mallorca que tiene origen en Andratx.

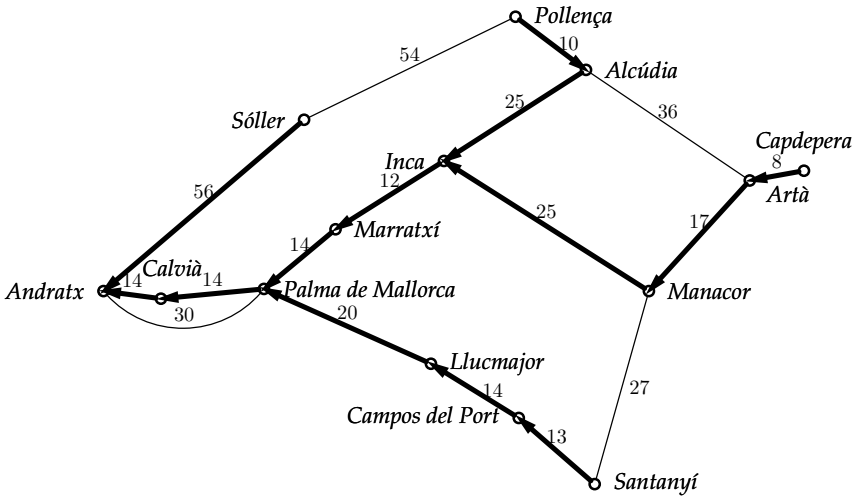


Figura 7.19: Árbol de caminos más cortos en el grafo de Mallorca con origen en Andratx representado con punteros al padre, esto es, manteniendo para cada vértice un puntero al vértice que le antecede en el camino más corto desde el vértice origen a él.

No resulta difícil refinar el algoritmo anterior para que devuelva los punteros hacia atrás en lugar del conjunto de aristas *ST*. De hecho, la variable *in_frontier_from* ya almacena punteros hacia atrás. El único problema es que, al usarlos para representar la frontera, se eliminan algunos de sus componentes durante el cálculo. Haremos estos cambios:

- representaremos los vértices de la frontera con un conjunto, y no con un diccionario (variable *frontier*),
- sustituiremos *ST* por un diccionario de punteros hacia atrás (variable *backpointer*): a cada vértice se asociará el vértice que le precede en el camino óptimo desde el origen hasta él.

shortestpaths.py *(cont.)*

```
20 def all_shortest_paths_from_source2(G, d, s, infinity=3.4e+38):
21     D = dict((v, infinity) for v in G.V)
22     D[s] = 0
23     backpointer = dict( (v, None) for v in G.V )
24     for v in G.succs(s):
25         D[v] = d(s,v)
26         backpointer[v] = s
27     added = set([s])
28     frontier = set( v for v in G.succs(s) )
29     while len(frontier) > 0:
30         dmin, v = min( (D[v], v) for v in frontier )
31         frontier.remove(v)
32         added.add(v)
33         for w in G.succs(v):
34             if w not in added and D[v] + d(v,w) < D[w]:
35                 D[w] = D[v] + d(v,w)
36                 backpointer[w] = v
37                 frontier.add(w)
38     return backpointer
```

```
1 from shortestpaths import all_shortest_paths_from_source2
2 from mallorca import Mallorca, km
3
4 backpointer = all_shortest_paths_from_source2(Mallorca, km, 'Andratx')
5 for city in sorted(backpointer):
6     print '%18s <- %-18s' % (city, backpointer[city])
```

```
          Alcúdia <- Inca
          Andratx <- None
             Artà <- Manacor
           Calvià <- Andratx
  Campos del Port <- Llucmajor
        Capdepera <- Artà
             Inca <- Marratxí
        Llucmajor <- Palma de Mallorca
          Manacor <- Inca
         Marratxí <- Palma de Mallorca
Palma de Mallorca <- Calvià
         Pollença <- Alcúdia
         Santanyí <- Campos del Port
           Sóller <- Andratx
```

Si queremos conocer el camino más corto de un ciudad a otra, podemos aplicar un algoritmo sencillo:

shortestpaths.py *(cont.)*

```
40 def backtrace(backpointer, target):
41     v = target
42     path = []
43     while v != None:
```

```
44          path.append(v)
45          v = backpointer[v]
46      path.reverse()
47      return path
```

```
1  from shortestpaths import all_shortest_paths_from_source2, backtrace
2  from mallorca import Mallorca, km
3
4  backpointer = all_shortest_paths_from_source2(Mallorca, km, 'Andratx')
5  print ', '.join(backtrace(backpointer, 'Manacor'))
```

```
Andratx, Calvià, Palma de Mallorca, Marratxí, Inca, Manacor
```

7.8.4. Análisis y nuevo refinamiento

Suponiendo que todos los vértices son alcanzables desde s, el algoritmo ejecuta un total de $|V| - 1$ iteraciones, ya que el conjunto *added* se inicializa con un vértice, en cada iteración ingresa uno nuevo en *added* y han de ingresar todos para que el algoritmo finalice. En cada iteración se selecciona un vértice de entre todos los que presenta la frontera. El número de vértices en la frontera es $O(|V|)$. A continuación se visitan los sucesores del vértice seleccionado, que también pueden ser hasta $|V|$. El coste del algoritmo es, pues, $O(|V|^2)$.

Es posible reducir el coste temporal si utilizamos una estructura de datos adecuada para representar la frontera (como hicimos con el algoritmo de Prim). Efectuamos las siguientes operaciones sobre dicho conjunto:

- Inserción de un vértice.
- Extracción del vértice v con menor valor de $D[v]$.

En principio, una cola de prioridad parece apropiada. Pero hemos de tener en cuenta que el valor de $D[v]$ para un vértice de la frontera puede modificarse en el intervalo de tiempo que transcurre entre su inserción y extracción. Por ello, un diccionario de prioridad es más apropiado. Esta nueva versión explota esta estructura de datos (y devuelve también el diccionario con la distancia más corta de s a cualquier otro vértice a la vez que simplifica la inicialización):

shortestpaths.py *(cont.)*

```
49  from prioritydict import MinPriorityDict
50
51  def all_shortest_paths_from_source(G, d, s, infinity=3.4e+38):
52      D = dict((v, infinity) for v in G.V)
53      D[s] = 0
54      backpointer = dict((v, None) for v in G.V)
55      frontier = MinPriorityDict(D, len(G.V))
56      added = set()
```

```
57      while len(frontier) > 0:
58          v, dmin = frontier.extract_min()
59          added.add(v)
60          for w in G.succs(v):
61              if w not in added and D[v] + d(v,w) < D[w]:
62                  frontier[w] = D[w] = D[v] + d(v,w)
63                  backpointer[w] = v
64      return backpointer, D
```

```
1  from shortestpaths import all_shortest_paths_from_source, backtrace
2  from mallorca import Mallorca, km
3
4  backpointer, D = all_shortest_paths_from_source(Mallorca, km, 'Andratx')
5  print ', '.join(backtrace(backpointer, 'Manacor'))
6  print 'Distancia recorrida:', D['Manacor']
```

```
Andratx, Calvià, Palma de Mallorca, Marratxí, Inca, Manacor
Distancia recorrida: 79
```

El algoritmo sigue efectuando $|V| - 1$ iteraciones, pero las operaciones que realiza y su coste respectivo es:

- Se extrae el vértice v con menor valor de $D[v]$, pero esta operación ahora requiere tiempo $O(\lg |V|)$. El coste global de estas extracciones es $O(|V| \lg |V|)$.
- Se recorren los $O(|V|)$ sucesores de V. Como cada vértice sólo se extrae una vez de la frontera y sus sucesores, por tanto, sólo se recorren una vez, el coste temporal total de estos recorridos es $O(|E|)$.
- Si uno de los sucesores w ha de actualizar el valor de $D[w]$, hemos de actualizar su ubicación en la cola de prioridad con coste $O(\lg |V|)$. El coste global de estas actualizaciones es $O(|E| \lg |V|)$.

Así pues, el coste temporal global es $O(|E| \lg |V|)$.

Nótese que si el grafo es denso, esta modificación del algoritmo es más ineficiente. En grafos con $|E|$ sensiblemente inferior a $|V|^2$, la versión del algoritmo que usa colas de prioridad resulta preferible.

Existe una implementación aún más eficiente que se basa en la implementación de las colas de prioridad con Heaps de Fibonacci. Su coste temporal es $O(|E| + |V| \lg |V|)$.

7.8.5. El camino más corto entre un par de vértices en un grafo ponderado positivo

Frecuentemente no necesitamos conocer todos los caminos más cortos de s a cualquier otro vértice: sólo deseamos conocer el camino más corto de s a un vértice destino t. El algoritmo de Dijkstra puede modificarse trivialmente para efectuar este cálculo: basta con detener las iteraciones cuando el vértice t ingresa en el árbol de recubrimiento.

shortestpaths.py *(cont.)*

```
66 def shortest_path(G, d, s, t, infinity=3.4e+38):
67     D = dict((v, infinity) for v in G.V)
68     D[s] = 0
69     backpointer = dict((v, None) for v in G.V)
70     frontier = MinPriorityDict(D, len(G.V))
71     added = set()
72     while len(frontier) > 0:
73         v, dmin = frontier.extract_min()
74         added.add(v)
75         if v == t: break
76         for w in G.succs(v):
77             if w not in added and D[v] + d(v,w) < D[w]:
78                 frontier[w] = D[w] = D[v] + d(v,w)
79                 backpointer[w] = v
80     return backtrace(backpointer, t), D[t]
```

```
1 from shortestpaths import shortest_path
2 from mallorca import Mallorca, km
3
4 path, distance = shortest_path(Mallorca, km, 'Andratx', 'Manacor')
5 print ', '.join(path)
6 print 'Distancia recorrida:', distance
```

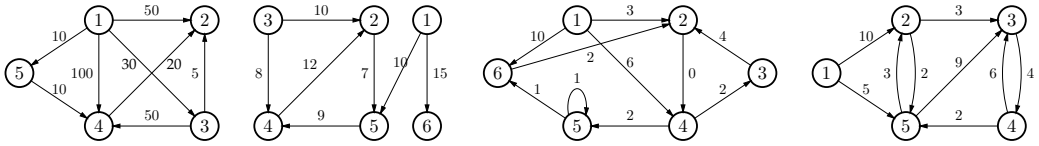
```
Andratx, Calvià, Palma de Mallorca, Marratxí, Inca, Manacor
Distancia recorrida: 79
```

El coste de este nuevo algoritmo para el peor de los casos es igual al del anterior: en el peor caso, el vértice destino ingresa en *added* tras haberlo hecho todos los demás.

EJERCICIOS

7-35 Un digrafo ponderado en el que todos los arcos pesan 1 es un digrafo ponderado positivo. Podemos usar, pues, el algoritmo de Dijkstra para calcular el camino más corto entre un par de vértices. ¿Es lo mejor que podemos hacer?

7-36 Haz una traza del algoritmo de Dijkstra para los siguientes grafos, tomando como vértice origen el 1.



7-37 Muestra un ejemplo sencillo de un grafo dirigido con pesos negativos en las aristas para el que el algoritmo de Dijkstra produce una solución errónea.

7-38 Deseamos encontrar un camino de coste mínimo entre la esquina inferior izquierda y la esquina superior derecha de una matriz en la que el valor de la celda de índices (i, j) representa la altura del terreno en esas coordenadas. Desde una casilla podemos desplazarnos únicamente hacia el norte y hacia el este. Buscamos:

a) el camino de menor altura total (suma de alturas de cada casilla visitada);

b) el camino de altura media mínima.

Diseña un algoritmo que resuelva estos problemas y analiza su complejidad computacional en función n y m, los números de filas y columnas de la matriz.

7-39 Si en lugar de la suma de pesos de aristas consideramos el producto de sus pesos para calcular el peso de un camino, ¿es suficiente con exigir que el peso de las aristas sea positivo para poder aplicar el algoritmo de Dijkstra?

7-40 Una empresa debe repartir paquetes entre cualquier par de las ciudades de un mapa modelado con un digrafo ponderado por la distancia entre ciudades conectadas directamente por carretera. Sus camiones no pueden viajar a más de 80 kilómetros por hora y la ley prohíbe que un camionero circule más de dos horas seguidas. Teniendo en cuenta que sólo podemos estacionar el camión en las ciudades señaladas en el mapa, queremos diseñar un programa que calcule el camino más corto entre un par de ciudades cualquiera, que nos indique el menor número de paradas que debe efectuar el camionero para no violar la ley y en qué ciudades debe efectuar las paradas. Analiza el coste temporal de tu algoritmo.

7-41 Tenemos una cinta magnética de longitud L y n ficheros $f_1, f_2, \ldots, f_n$ de longitudes $l_1, l_2, \ldots, l_n$ (los n ficheros caben en la cinta). Deseamos acceder eficientemente a los ficheros de la cinta y para ello resulta conveniente almacenarlos en un orden determinado. La cinta siempre está al principio antes de efectuar cualquier acceso, así que, si el orden de almacenamiento se expresa con los enteros $i_1, i_2, \ldots, i_n$, donde i_k es el índice del fichero que disponemos en la k-ésima posición, el tiempo necesario para acceder al fichero i_j será $t_j = \sum_{1 \leq k \leq j} l_{i_k}$.

Se pide:

a) Si suponemos que la probabilidad de acceder a cualquier fichero es la misma, el tiempo medio de acceso es $T = \frac{1}{n} \sum_{1 \leq j \leq n} t_j$. ¿Cuál es el orden de almacenamiento que proporciona menor tiempo medio de acceso?

b) Si al fichero f_i se accede con probabilidad p_i, donde $\sum_{1 \leq i \leq n} p_i = 1$, el tiempo medio de acceso es $T = \sum_{1 \leq j \leq n} t_j p_j$. ¿Cuál es el orden de almacenamiento que proporciona menor tiempo medio de acceso?

7-42 En un juego de cartas de fantasía cada carta está marcada con un valor de ataque y un valor de defensa. El oponente efectúa un ataque disponiendo sobre la mesa n cartas con valores de ataque $a_1, a_2, \ldots, a_n$. Podemos bloquear cada carta atacante con una carta defensora. Para que el bloqueo sea efectivo, el valor de defensa de la carta defensora ha de ser mayor o igual que el valor de ataque de la carta atacante. Cada carta atacante no bloqueada nos hace perder un punto. Disponemos en nuestra mano de m cartas con valores de defensa $d_1, d_2, \ldots, d_m$. Diseña, si es posible, una estrategia voraz que permita resistir un ataque con la menor pérdida posible de puntos. Analiza su coste temporal.

7-43 Se tiene un disco con capacidad para T kilobytes y N ficheros que se desea almacenar en el disco. El tamaño total de los ficheros puede exceder la capacidad T. Disponemos de un vector t con el tamaño de cada uno de los N ficheros, a los que identificamos con un índice entre 0 y $N-1$.

a) Deseamos maximizar la cantidad total de ficheros almacenados en disco. Diseña un algoritmo voraz que resuelva el problema, demuestra su corrección y analiza su complejidad.

b) Con el mismo objetivo del apartado anterior, un estudiante propone este algoritmo voraz: «Ordenar los ficheros por orden *decreciente* de tamaño e ir almacenando ficheros mientras quede espacio en el disco; si un determinado fichero no cabe, se descarta y se pasa al siguiente.» Discute el coste temporal del algoritmo propuesto y da un contraejemplo para comprobar que este algoritmo no siempre obtiene la solución óptima.

c) Se desea diseñar un algoritmo voraz que decida qué ficheros almacenar en el disco para conseguir la máxima ocupación posible del disco. ¿Devuelve el algoritmo voraz la solución correcta? Calcula el coste temporal del algoritmo.

7-44 Un sistema informático con p procesadores debe ejecutar n tareas, cada una con un tiempo de ejecución t_i, para i entre 1 y n. Queremos minimizar el tiempo medio de espera hasta la compleción de cada tarea. Diseña un algoritmo voraz que resuelva el problema, demuestra su corrección y analiza su complejidad.

7-45 Dada una serie de puntos en la recta, $x_1, x_2, \ldots, x_n$, tales que $x_i < x_{i+1}$, para i entre 1 y $n-1$, queremos calcular el menor número de intervalos cerrados de longitud a que contienen a todos los puntos. Diseña un algoritmo voraz que resuelva el problema, demuestra su corrección y analiza su complejidad.

Por ejemplo, la serie de puntos 1, 2.5, 3.5, 7 y 9 queda comprendida en dos intervalos de longitud 3: $[1,4]$ y $[7,10]$.

7.9. La estrategia voraz como heurística o técnica de aproximación

Como dijimos en la sección 2.7, ciertos problemas son intrínsecamente difíciles. De algunos se sabe que no existe algoritmo resolutivo que requiera tiempo polinómico con la talla de sus instancias y de otros se presume que ocurre esto mismo (y, en cualquier caso, hoy no se conoce algoritmo eficiente alguno). Dado que no resulta práctico intentar obtener la solución exacta para las instancias de estos problemas, es interesante obtener algoritmos que proporcionen soluciones aproximadas. La estrategia voraz puede proporcionar soluciones subóptimas que constituyen buenas aproximaciones a la óptima. Un algoritmo que proporciona una solución aproximada a las instancias de un problema de optimización se denomina **algoritmo de aproximación**. En los algoritmos de aproximación es posible acotar la calidad de la solución encontrada, esto es, la diferencia o el ratio entre ella y la solución óptima. Cuando no podemos proporcionar estas cotas o cuando el algoritmo ni tan siquiera garantiza que se encuentre una solución factible, hablamos de un **algoritmo heurístico**.

7.9.1. El problema del empaquetado en cajas

Deseamos cargar un conjunto de n objetos, cada uno con un peso $w_i \in \mathbb{R}^{\geq 0}$, para $1 \leq i \leq n$, en el menor número posible de contenedores, cada uno de ellos con una capacidad de carga C (suponemos que $w_i \leq C$, para todo i). El problema del empaquetado en

cajas, que así se llama, encuentra numerosas aplicaciones: carga de contenedores, corte de listones de madera en trozos de diferentes longitudes, almacenamiento de archivos en el menor número de cintas, asignación de procesos con estimación de tiempo de ejecución al menor número posible de procesadores, etc.

Este problema se conoce en la literatura en inglés por «*bin packing*». Se trata de un problema que aparece en numerosos campos de aplicación, por lo que presenta gran interés. Existen generalizaciones bidimensionales y tridimensionales del problema. La generalización bidimensional propone encontrar el menor número de planchas rectangulares necesarias para, mediante su recorte, proporcionar n tablas rectangulares. La generalización tridimensional propone el cálculo del menor número de paralelepípedos (remolques de camión) en el que podemos empaquetar n paralelepípedos (cajas).

Una solución puede expresarse con una n-tupla $(x_1, x_2, \ldots, x_n) \in \mathbb{N}^n$ en la que cada número indica 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. 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 tal que $\bigcup_{1 \leq i \leq n} \{x_i\} = [1..M]$.

La función objetivo, cuyo valor deseamos minimizar, es

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

Un ejemplo ayudará a entender el problema. Supóngase que disponemos de listones de madera de 10 metros y nos hacen un pedido de listones con las siguientes medidas: 1, 2, 8, 7, 8 y 3 metros. Podemos cortar tres listones de madera de 10 metros para satisfacer el pedido, como muestra la figura 7.20. La solución del ejemplo se puede expresar con la tupla $(3, 1, 1, 2, 3, 2)$. El número de listones usados es 3.

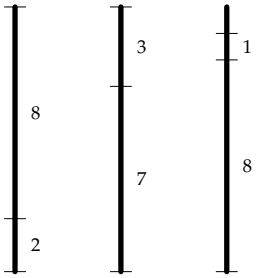


Figura 7.20: Solución al problema del empaquetado en cajas con $C = 10$ y $w_1 = 1$, $w_2 = 2$, $w_3 = 8$, $w_4 = 7$, $w_5 = 8$ y $w_6 = 3$. La solución encontrada utiliza 3 cajas.

Vamos a presentar varios algoritmos voraces para el problema. Ninguno de ellos da garantías de encontrar la solución óptima, pero los dos últimos son algoritmos de aproximación.

«Mientras quepa»

Una solución voraz inmediata consiste en recorrer los objetos en un orden arbitrario y mantener un contenedor «abierto». Mientras quepan objetos en él, se insertan, y cuando no quepan, se abre un nuevo contenedor:

greedy_bin_packing.py

```
from offsetarray import OffsetArray

def next_fit_bin_packing(w, C):
    x = OffsetArray([])
    cap_left = C
    bin = 1
    for i in xrange(1, len(w)+1):
        if cap_left < w[i]:
            cap_left = C
            bin += 1
        cap_left -= w[i]
        x.append(bin)
    return x

def show_solution(x, w):
    for i in xrange(1, max(x)+1):
        print '%2d:' % i,
        for j in xrange(1, len(x)+1):
            if x[j] == i: print w[j],
        print
```

```
from greedy_bin_packing import next_fit_bin_packing, show_solution
from offsetarray import OffsetArray

w = OffsetArray([ 1, 2, 8, 7, 8, 3])
show_solution(next_fit_bin_packing(w, 10), w)
```

```
 1: 1 2
 2: 8
 3: 7
 4: 8
 5: 3
```

EJERCICIOS

7-46 Analiza el coste temporal del algoritmo *next_fit_bin_packing*.

«En el primero que quepa»

Una estrategia alternativa consiste en meter cada objeto en el primer contenedor en el que quepa, si cabe en alguno. Si no cabe en ninguno, se crea un nuevo contenedor y se inserta en él el objeto. Esta técnica se conoce como «primero en el que quepa»:

greedy_bin_packing.py *(cont.)*

```
23 def first_fit_bin_packing(w, C):
24     x = OffsetArray([None]*len(w))
25     free = OffsetArray([])
26     for i in xrange(1, len(w)+1):
27         for j in xrange(1, len(free)+1):
28             if free[j] >= w[i]:
29                 x[i] = j
30                 free[j] -= w[i]
31                 break
32         if x[i] == None:
33             free.append(C-w[i])
34             x[i] = len(free)
35     return x
```

```
1 from greedy_bin_packing import first_fit_bin_packing, show_solution
2 from offsetarray import OffsetArray
3
4 w = OffsetArray([1, 2, 8, 7, 8, 3])
5 show_solution(first_fit_bin_packing(w, 10), w)
```

```
1: 1 2 7
2: 8
3: 8
4: 3
```

Es fácil ver que se trata de un algoritmo de aproximación y que consume un número de contenedores que es, como mucho, el doble del mínimo. Veamos por qué. Si el algoritmo usa M contenedores, al menos $M-1$ están más que medio llenos: si hubiese dos que no llegan a estar medio llenos, los objetos de uno de ellos hubiesen podido almacenarse en el otro. Tenemos, pues, que

$$\sum_{1\leq i\leq n} w_i > \frac{C(M-1)}{2}.$$

O sea,

$$M < 1 + 2\frac{\sum_{1\leq i\leq n} w_i}{C}.$$

Por otra parte, el valor de $(\sum_{1\leq i\leq n} w_i)/C$ es una cota inferior sobre el número mínimo de contenedores necesarios, al que denotamos con $f(\hat{x})$:

$$\frac{\sum_{1\leq i\leq n} w_i}{C} \leq f(\hat{x}).$$

Se puede deducir, entonces, que

$$M < 1 + \frac{2\sum_{1\leq i\leq n} w_i}{C} \leq 1 + 2f(\hat{x}).$$

Y como M es entero,

$$M \leq 2f(\hat{x}).$$

Hay un resultado más fuerte que éste. Se puede demostrar que la solución encontrada por este algoritmo es menor o igual que $2 + \frac{17}{10}f(\hat{x})$.

EJERCICIOS

7-47 Analiza el coste temporal del algoritmo.

7-48 Otro algoritmo de aproximación con la misma cota se puede denominar «en el que mejor quepa» (*best fit*): consiste en meter cada objeto en el contenedor que deje menor espacio libre tras su inclusión. Implementa este algoritmo y analiza su coste temporal. Demuestra, con un contrajemplo, que ese método no encuentra la solución óptima.

«De mayor a menor peso, en el primero que quepa»

Resulta mejor la consideración de los objetos ordenados de mayor a menor peso:

greedy_bin_packing.py *(cont.)*

```
37 def first_fit_decreasing_bin_packing(w, C):
38     x = OffsetArray([None] * len(w))
39     W = OffsetArray(sorted([(w[i], i) for i in xrange(1, len(w)+1)], reverse=True))
40     w' = OffsetArray([W[i][0] for i in xrange(1, len(W)+1)])
41     x = first_fit_bin_packing(w', C)
42     x' = OffsetArray([None] * len(x))
43     for i in xrange(1, len(x)+1): x'[W[i][1]] = x[i]
44     return x'
```

```
1 from greedy_bin_packing import first_fit_decreasing_bin_packing, show_solution
2 from offsetarray import OffsetArray
3
4 w = OffsetArray([ 1, 2, 8, 7, 8, 3])
5 show_solution(first_fit_decreasing_bin_packing(w, 10), w)
```

```
1: 2 8
2: 1 8
3: 7 3
```

Se puede demostrar que la solución encontrada por este algoritmo es menor o igual que $4 + \frac{11}{9}f(\hat{x})$.

El problema puede extenderse a 2 y 3 dimensiones. En la versión bidimensional se propone obtener una serie de rectángulos de madera, cada uno con una anchura y altura, recortando planchas rectangulares de dimensión fija y conocida. El objetivo es reducir al máximo el número de planchas rectangulares utilizadas. En la versión tridimensional se propone cargar una serie de contenedores (cuya altura, anchura y profundidad nos indican) con un conjunto de paquetes (paralelepípedos, cada uno con su altura, anchura y profundidad). El objetivo es cargar todos los paquetes en el menor número posible de contenedores.

EJERCICIOS

7-49 Encuentra un ejemplo en el que esta estrategia voraz no dé con la solución óptima.

7-50 Analiza el coste temporal del algoritmo.

7-51 Otro algoritmo de aproximación con la misma cota se puede denominar «de mayor a menor peso, en el que mejor quepa» (*best fit decreasing*): consiste en considerar los objetos por orden de peso decreciente y meter cada objeto en el contenedor que deje menor espacio libre tras su inclusión. Implementa este algoritmo, analiza su coste temporal y demuestra que no encuentra la solución óptima.

7.9.2. Coloreado de un grafo

Queremos asignar colores a los vértices de un grafo no dirigido de modo tal que dos vértices adyacentes no tengan asignado el mismo color y que el número de colores empleado sea mínimo.

Este problema tiene una aplicación curiosa: al imprimir mapas en color se desea reducir el número de tintas, pues encarece la impresión, a la vez que evitar que dos países vecinos compartan el mismo color. Se puede modelar el problema representando cada país con un vértice y uniendo con una arista países que comparten línea fronteriza. Desde mediados del siglo XIX se pensaba que 4 colores eran suficientes para colorear cualquier mapa. Appel y Haken demostraron en 1976 que es así. Su demostración se basa en reducir todos los grafos posibles a unos 1 500 casos y resolver cada uno de ellos con ayuda de un computador, tarea que requirió 1 200 horas de cálculo. El «teorema de los cuatro colores» es el primer teorema demostrado con la ayuda de un computador.

Se trata de un problema para el que cualquier algoritmo que proporciona la solución óptima requiere tiempo exponencial. Se puede seguir una estrategia voraz para *aproximar* eficientemente la solución. Una aproximación voraz posible consiste en, reiteradamente, tomar un color no utilizado aún y asignarlo al mayor número posible de vértices satisfaciendo la restricción de que dos vértices adyacentes no compartan color:

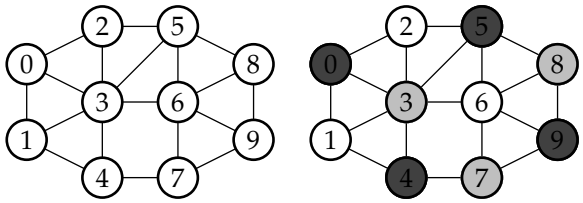
graph_colors.py

```
1  def graph_colors(G):
2      V = set(G.V)
3      partition = set()
4      while len(V) > 0:
5          last_color = set()
6          for v in V:
7              if all(v not in G.succs(u) for u in last_color):
8                  last_color.add(v)
9          V = V.difference(last_color)
10         partition.add(frozenset(last_color))
11     return partition
```

```
1  from graph import Graph
2  from graph_colors import graph_colors
3
4  G = Graph(E=[(0,1), (0,2), (0,3), (1,3), (1,4), (2,3), (2,5), (3,4), (3,5),
5               (3,6), (4,7), (5,6), (5,8), (6,7), (6,8), (6,9), (7,9), (8,9)])
6  colors = graph_colors(G)
7  for i, s in enumerate(colors):
8      print 'Color %d:' % (i+1),
9      for v in s: print v,
10     print
```

```
Color 1: 0 9 4 5
Color 2: 1 2 6
Color 3: 8 3 7
```

La figura 7.21 ilustra el coloreado de grafo propuesto por el algoritmo en esta ejecución.



(a) *(b)*

Figura 7.21: (a) Grafo no dirigido. (b) Coloreado de los vértices encontrado por el algoritmo voraz.

EJERCICIOS

7-52 Calcula el coste del algoritmo propuesto.

7-53 Propón un ejemplo de grafo en el que el algoritmo presentado no encuentre una solución óptima.

El algoritmo propuesto no siempre encuentra una solución óptima, aunque podría encontrarla. Cabe advertir que en todo grafo existe al menos una ordenación de los vértices que garantiza que la aplicación de esta estrategia encuentre la solución óptima. Si nos proporcionan una solución óptima, cualquier orden de recorrido que considere consecutivamente todos los vértices del mismo color presenta esta propiedad. La solución encontrada no tiene por qué coincidir con la que nos proporcionaron, pero es igualmente óptima. ¡La dificultad estriba en que conocer ese orden pasa por haber resuelto el problema! Enumerar todas las ordenaciones posibles de n vértices está fuera de lugar: hay $n!$.

Pero no se trata de un algoritmo de aproximación, sino heurístico, aunque siempre encuentra una solución factible. Si suministramos un grafo con n vértices puede encontrar una solución que requiere un número de colores proporcional a n, cuando una cantidad constante puede ser suficiente. Consideremos un grafo como el de la figura 7.22: si visi-

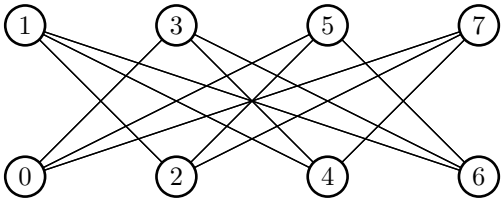


Figura 7.22: Grafo en el que el algoritmo voraz de coloreado puede necesitar un número de colores proporcional al de vértices.

tamos los vértices pares primero e impares después, coloreamos el grafo con dos colores; si los visitamos en el orden natural tendremos que usar $n/2$ colores.

7.9.3. El problema del viajante de comercio

¡viajante de comercio

Deseamos conocer el recorrido de longitud mínima que partiendo y finalizando en un vértice cualquiera, visita todos los vértices de un grafo una sola vez. Un recorrido así recibe el nombre de ciclo hamiltoniano del grafo. El problema se conoce en la literatura con el nombre de **el problema del viajante de comercio** o por TSP, las siglas de su traducción inglesa: «traveling salesman problem».

Se trata de un problema de mucho interés práctico: ¿qué ruta debe seguir el camión de la basura para pasar por todos los puntos de recogida en el menor tiempo posible?, ¿cómo interconectamos varios ordenadores en una red en anillo con el menor consumo de cable?, ¿cómo organiza su ruta un viajante de comercio que debe visitar una serie de establecimientos repartidos por el país?

Desgraciadamente, los algoritmos que encuentran una solución exacta para cualquier grafo se ejecutan en tiempo que crece exponencialmente con el número de vértices. Este es un problema NP y no se conoce ningún algoritmo capaz de proporcionar la solución óptima en tiempo polinómico.

Podemos diseñar algoritmos voraces para tratar de encontrar una solución y, si es posible, que ésta constituya una aproximación a la óptima.

- Nótese que en un ciclo hamiltoniano todos los vértices tienen dos aristas: la que le conecta al vértice anterior y la que le conecta al vértice siguiente. Una idea sencilla consiste en seleccionar reiteradamente la arista de menor peso de entre las que no violan esta propiedad de los ciclos hamiltonianos.

- Una idea alternativa consiste en empezar en un vértice cualquiera y escoger como siguiente vértice un sucesor no considerado hasta el momento. Interesa considerar el sucesor cuya arista de conexión con el último vértice visitado presente menor peso.

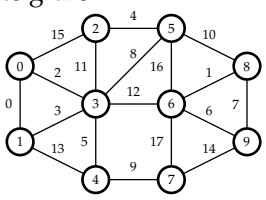
Las dos técnicas son meras heurísticas: no ofrecen garantías de encontrar una solución factible.

EJERCICIOS

7-54 Analiza el coste temporal de los algoritmos basados en las heurísticas descritas.

7-55 Encuentra contraejemplos para ambas heurísticas.

7-56 Aplica ambos algoritmos a este grafo:



¿Existe un ciclo hamiltoniano en el grafo? ¿Encuentra cada uno de los algoritmos un ciclo hamiltoniano? ¿Es, en cada caso, el ciclo hamiltoniano de coste mínimo?

7-57 Aplica los algoritmos al mapa `iberia` para tratar de encontrar una ruta de coste mínimo que visite todas las ciudades. Representa gráficamente el resultado. ¿Se obtienen ciclos hamiltonianos? ¿Existe un ciclo hamiltoniano en el mapa?

7.9.4. El problema del viajante de comercio euclídeo

¡viajante de comercio—euclídeo

Si bien el problema del viajante de comercio no es abordable con un algoritmo voraz, sí hay casos particulares en los que la estrategia voraz conduce a buenos algoritmos de aproximación. Consideremos el caso del **problema del viajante de comercio euclídeo**, esto es, el problema del viajante de comercio en grafos *no dirigidos*, *completos* y *euclídeos*.

Una primera idea consiste en partir de un vértice cualquiera e ir formando un camino considerando el vértice más próximo que no ha sido visitado. Una vez visitados todos los vértices, cerramos el ciclo volviendo a visitar el vértice de partida. Este algoritmo voraz se conoce como «el vecino más próximo».

Esta implementación obtiene la solución del vecino más próximo si se suministra una lista de puntos a la función:

tsp.py

```
1  from math import sqrt
2  from utils import argmin
3
4  def d(a, b):
5      return sqrt(sum((a[i]-b[i])**2 for i in xrange(len(a))))
6
7  def nearest_neighbor_tsp(points):
8      path = [ points[0] ]
9      unvisited = set(points[1:])
10     while len(unvisited) > 0:
11         nn = argmin(unvisited, lambda (x,y): d(path[-1], (x,y)))
12         unvisited.remove(nn)
13         path.append(nn)
14     path.append(points[0])
15     return path
```

Probemos el programa con el grafo euclídeo de la figura 7.23 (a). La figura 7.23 (b) ilustra el recorrido encontrado por el algoritmo del vecino más próximo.

```
1 from tsp import nearest_neighbor_tsp, d
2
3 path = nearest_neighbor_tsp([(6,4), (0,6), (1,0), (2,6), (3,1), (4,1), (4,4)])
4 print path
5 print 'Distancia recorrida:', sum(d(path[i], path[i+1]) for i in xrange(len(path)-1))
```

```
[(6, 4), (4, 4), (2, 6), (0, 6), (3, 1), (4, 1), (1, 0), (6, 4)]
Distancia recorrida: 23.2247809172
```

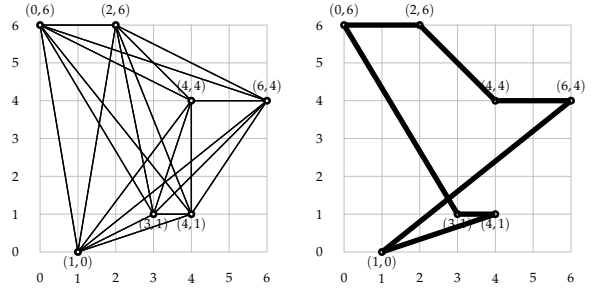


Figura 7.23: (a) Un grafo no dirigido, completo y euclídeo. (b) Ciclo hamiltoniano obtenido con el método del vecino más próximo partiendo del vértice $(6,4)$.

Hay una técnica voraz alternativa que resulta interesante. Podemos advertir que un ciclo hamiltoniano tiene $|V|$ aristas y que la eliminación de una cualquiera de ellas nos proporciona un árbol de recubrimiento. Si D es la distancia recorrida por el ciclo hamiltoniano, el coste del árbol resultante de eliminar una arista es menor o igual que D. Así pues, el árbol de recubrimiento mínimo presenta una suma de distancias D' y sabemos que $D' \leq D$.

La figura 7.24 (a) muestra el MST del grafo no dirigido, completo y euclídeo de la figura 7.23 (a). El árbol de recubrimiento puede explorarse por primero en profundidad considerando que un vértice cualquiera es la raíz, por ejemplo, el vértice $(6,4)$. Si usamos cada arista del MST dos veces tenemos un recorrido que visita cada vértice al menos una vez: $(6,4)$, $(4,4)$, $(2,6)$, $(0,6)$, $(2,6)$, $(4,4)$, $(4,1)$, $(3,1)$, $(1,0)$, $(3,1)$, $(4,1)$, $(4,4)$ y $(6,4)$. La figura 7.23 (a) muestra, en trazo discontinuo, este recorrido. Se trata de un ciclo, pero no de un ciclo hamiltoniano: visita algunos vértices en más de una ocasión. El coste de este camino es $2D'$, pues cada arista del MST se recorre dos veces.

Consideremos ahora el recorrido en preorden del MST y conectemos el último vértice de dicho recorrido con la raíz. En el árbol de la figura 7.23 (a), dicho recorrido produce la secuencia de vértices $(6,4)$, $(4,4)$, $(2,6)$, $(0,6)$, $(4,1)$, $(3,1)$ y $(1,0)$. (Las figuras 7.24 (b) a (h) muestran una traza, paso a paso, de este método al aplicarse a este grafo.) El coste de este ciclo hamiltoniano es necesariamente menor que $2D'$: ciertas secuencias de dos o más aristas consecutivas en el recorrido original se han sustituido por una arista «directa» entre sus vértices inicial y final, que es más corta que el rodeo dado por la secuencia eliminada. Decimos que la arista directa es un atajo. Nótese que el algoritmo escoge, en cada paso del recorrido en preorden, el atajo que ve en cada instante, sin considerar la posibilidad de que haya otros más beneficiosos globalmente. El ciclo hamiltoniano

encontrado no es el óptimo: el que muestra la figura 7.25, por ejemplo, recorre menor distancia.

El algoritmo voraz basado en Kruskal es un algoritmo de aproximación: siempre encuentra una solución factible y, además, ésta presenta un ciclo que recorre una distancia que nunca es más de dos veces la que recorre el óptimo. Veamos por qué. La solución que encuentra este algoritmo es menor o igual que $2D'$, es decir, dos veces el coste del árbol de recubrimiento mínimo. Y como D, el coste del ciclo hamiltoniano óptimo, es mayor o igual que D', podemos afirmar que el coste del ciclo devuelto recorre una distancia que es, como mucho, el doble que la distancia recorrida por el ciclo hamiltoniano óptimo.

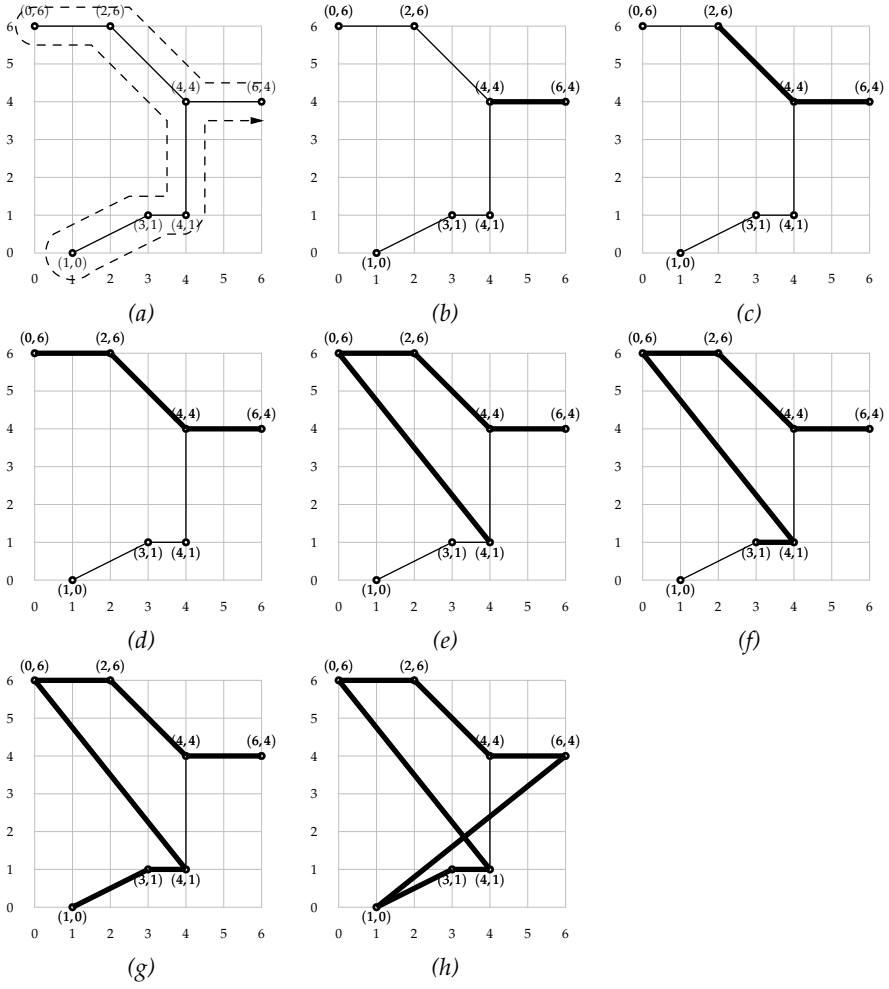


Figura 7.24: Traza de ejecución algoritmo de aproximación. En (a) se muestra el MST del grafo de la figura 7.23 (a) y, en trazo discontinuo, un recorrido por primero en profundidad de dicho MST. De (b) a (h) se muestra, paso a paso, el proceso de construcción del ciclo hamiltoniano: se visitan los vértices en el orden de recorrido por primero y profundidad y se une cada vértice visitado por última vez con el último vértice del ciclo en construcción.

He aquí una implementación del algoritmo propuesto:

`tsp.py` *(cont.)*

```
18 from graph import Graph
19 from kruskal import Kruskal
20 from graph_traversal import depth_first_traversal
21
```

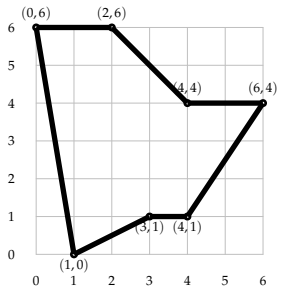


Figura 7.25: Ciclo hamiltoniano óptimo para el grafo completo y euclídeo de la figura 7.23 (a).

```
22 def mst_tsp(points):
23   G = Graph(E=[(a, b) for a in points for b in points if a!=b])
24   MST = Graph(V=points, E=Kruskal(G, d))
25   path = [ v for v in depth_first_traversal(MST, points[0]) ] + [points[0]]
26   return path
```

La siguiente ejecución de prueba sobre el grafo de la figura 7.23 (a) proporciona el ciclo hamiltoniano con que hemos ilustrado el funcionamiento del método.

```
1 from tsp import mst_tsp, d
2
3 path = mst_tsp([(6,4), (0,6), (1,0), (2,6), (3,1), (4,1), (4,4)])
4 print path
5 print 'Distancia recorrida:', sum(d(path[i], path[i+1]) for i in xrange(len(path)-1))
```

```
[(6, 4), (4, 4), (2, 6), (0, 6), (4, 1), (3, 1), (1, 0), (6, 4)]
Distancia recorrida: 22.8707435771
```

. . . . . EJERCICIOS . . . . .

7-58 ¿Qué algoritmo conviene escoger parea calcular el MST de un grafo como el que sirve de entrada en este problema, Prim o Kruskal?

Analiza el coste temporal del algoritmo de aproximación propuesto.

7-59 ¿Funciona el algoritmo si en lugar de usar la distancia euclídea usamos cualquier función de ponderación positiva que satisfaga la desigualdad triangular, esto es, con una función de ponderación métrica?

7-60 Una compañía de telefonía inalámbrica puede usar n puntos de enlace para establecer una comunicación. Conocemos las coordenadas (x, y) de cada punto de enlace y sabemos que un punto de enlace puede comunicarse con cualquier otro punto de enlace situado a k o menos kilómetros de distancia.

a) Si quisiéramos representar mediante un grafo la red de puntos de enlace capaces de comunicarse entre sí, ¿qué estructura de datos usarías, en función de qué y cuánto tiempo y espacio costaría obtener dicha representación?
b) ¿Cómo podríamos averiguar si es posible efectuar una comunicación entre *todo par* de puntos de enlace? ¿Con qué coste computacional?
c) Cuando un usuario efectúa una llamada, hemos de enviar la señal desde un punto de enlace A a otro B a través, posiblemente, de otros puntos de enlace. Usar una conexión entre dos puntos de enlace que pueden comunicarse directamente le cuesta a la compañía telefónica tantos céntimos como kilómetros separan los puntos de enlace. ¿Con qué algoritmo y con qué coste computacional es posible efectuar la conexión de coste mínimo?

d) Cuando la compañía envía un SMS publicitario debe difundirlo desde un punto de enlace A a todos los demás puntos de enlace. ¿Qué coste económico le supone? ¿Cómo puede saber la compañía de qué modo deben saltar los SMS de un punto de enlace a otro? ¿Qué coste computacional requiere dicho cálculo?

7-61 Dado un conjunto de puntos en la recta real $S = \{x_1, x_2, x_3, \ldots, x_n\}$, una 1-cobertura de S es un conjunto de intervalos cerrados de longitud 1 que incluyen a todos los puntos de S. Un ejemplo te ayudará a entender el concepto. Sea $S = \{1, 1.3, 1.4, 2.2, 3, 7, 7.2\}$, el siguiente conjunto de intervalos es una 1-cobertura: $C = \{[0,1], [1,2], [2.2, 3.2], [7,8]\}$. La talla de una 1-cobertura es el número de intervalos. Una 1-cobertura óptima es una con la menor talla posible. C no es una 1-cobertura óptima, pues contiene 4 intervalos y $C' = \{[1,2], [2,3], [7,8]\}$, por ejemplo, sólo contiene 3.

Diseña un algoritmo voraz que encuentre la 1-cobertura óptima, demuestra que tu algoritmo es correcto y analiza su coste computacional.

7-62 La señora Amanita tiene una lucrativa afición: la recolección de setas. Conoce muy bien el bosque que hay cerca de su casa y todos los años recoge las setas que crecen en él. A lo largo de los años ha apuntado, con ayuda de un GPS, las coordenadas de cada uno de los n puntos en los que hay setas.

La señora Amanita siempre camina a la misma velocidad (unos cinco kilómetros a la hora) y este año quiere recolectar todas las setas del bosque en el menor tiempo posible. Ha ideado el siguiente sistema: partiendo de su casa, va al lugar más cercano en el que hay setas y las recoge; a continuación, va al siguiente punto más cercano; y así sucesivamente.

La estrategia seguida es, evidentemente, voraz.

a) ¿Qué coste computacional presenta su estrategia? (Exprésala en función de n.)

b) ¿Es una estrategia que garantiza una solución óptima? Es decir, ¿crees que así recolectará todas las setas en el menor tiempo posible?

El señor Bolet compra setas a la señora Amanita y las vende por bares y restaurantes. Hay p variedades de seta que identificamos con números entre 1 y p. El precio de compra de la variedad i, para i entre 1 y p, es de c_i euros el kilo y su precio de venta es de v_i euros el kilo, donde $v_i > c_i$. La señora Amanita dispone de k_i kilos de la variedad i-ésima. La camioneta del señor Bolet puede cargar hasta K kilos de setas.

c) ¿Qué estrategia debe seguir el señor Bolet para maximizar su beneficio con la carga que le cabe en la camioneta? Diseña una estrategia voraz. (Explica con la mayor claridad posible el algoritmo diseñado.)

d) ¿Constituye tu estrategia un algoritmo óptimo? ¿Qué coste computacional tiene?

El señor Champiñón, al igual que la señora Amanita, recoge sus propias setas. En el bosque del señor Champiñón hay n puntos de recogida. En el punto i-ésimo hay k_i kilogramos de setas de una variedad que el señor Champiñón vende a v_i euros el kilo. Cada vez que el señor Champiñón va a un punto de recogida, se hace con todas las setas del lugar y vuelve a su casa para descargar.

e) ¿Qué debe hacer si quiere ganar la mayor cantidad posible de dinero y sólo puede permitirse hacer m viajes? Diseña una estrategia voraz.

f) ¿Le garantiza tu estrategia que recolectará la mayor cantidad posible de setas?

g) ¿Qué coste computacional tiene la estrategia del señor Champiñón? (Trata de ajustar el cálculo del coste temporal tanto como te sea posible y recurre a la estructura o estructuras de datos que consideres más apropiadas.)

7-63 En un departamento universitario hay P profesores y una lista con $2P$ asignaturas. Cada profesor debe impartir dos asignaturas. Al confeccionar el plan de organización docente se ha consultado a los profesores acerca de qué asignaturas prefieren dar. Cada profesor ha puntuado de 0 a 10 su interés por cada asignatura: la puntuación que el profesor p otorga a la asignatura a se ha almacenado en la celda $Q[p, a]$ de la matriz Q, donde p y a son números enteros en el rango $[0..P-1]$ y $[0..2P-1]$, respectivamente.

Deseamos asignar profesores a asignaturas de modo que la satisfacción global (suma de puntuaciones) sea máxima. Para resolver el problema hemos diseñado tres estrategias voraces:

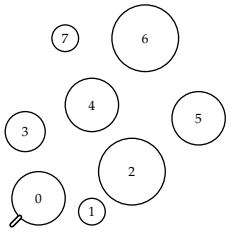
a) Se asignan al primer profesor las dos primeras asignaturas de la lista; al segundo, las dos siguientes; y así sucesivamente.

b) Se considera el primer profesor y se le asignan las dos asignaturas que prefiere; a continuación, se asignan al segundo profesor sus dos asignaturas favoritas de entre las no asignadas al primero; y así sucesivamente.

c) Se ordenan los profesores por valor decreciente de la suma de puntos asignados a sus dos asignaturas favoritas y se usa el mismo método del apartado anterior, pero considerando a los profesores en el orden indicado.

Debes demostrar si cada uno de los algoritmos propuestos es válido o no e indicar su respectivo coste temporal.

7-64 Nos han contratado como informáticos en un prestigioso despacho de arquitectura especializado en jardines y paisajística. Inexplicablemente, tienen éxito diseñando unos parques con n estanques de agua de aspecto sencillo: todos los estanques son cilindros de idéntica altura, pero de radio arbitrario. Los estanques están numerados de 0 a $n-1$, y, respectivamente, tienen radios $r_0, r_1, \ldots, r_{n-1}$ y centros en las coordenadas $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$ (tanto radios como coordenadas se expresan en metros). Esta figura muestra un ejemplo de parque con 8 estanques visto desde arriba.

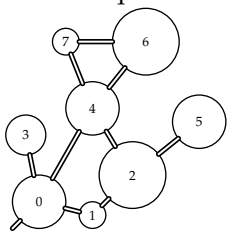


El estanque 0 está conectado a una fuente (representado con un tubo). De ella mana el agua con la que hemos de llenar todos los estanques.

Por el principio de los vasos comunicantes, el agua fluye entre dos estanques hasta igualar su nivel si éstos están conectados por un tubo en su parte inferior. La cantidad de tubo necesario para unirlos depende de las coordenadas de los dos estanques y de sus respectivos radios. Este diagrama ilustra esta idea y muestra, además, que el sistema de tuberías es subterráneo (la línea gris indica el nivel del suelo):

Dadps dps estanques cualesquiera, *a* y *b*, la tubería que los une parte del punto del estanque *a* más próximo al estanque *b*, desciende un metro por debajo del estanque *a*, va en línea recta al punto más próximo del estanque *b* y asciende un metro hasta el estanque *b*.

Nuestros jefes nos han pedido un programa que nos diga cómo hemos de interconectar los estanques entre sí para asegurarnos de que el agua llega a todos ellos *gastando la menor cantidad posible de metros de tubería*. No eres el primero en intentarlo. Aquí tienes el tipo de interconexiones que calcula el programa del último informático que contrataron:



Es fácil comprobar que no se trata de la solución óptima: podríamos suprimir el tubo que une los estanques 6 y 7 y el agua seguiría fluyendo desde el estanque 0 a todos los demás.

Diseña un algoritmo que resuelva el problema de la interconexión óptima de los estanques. Define previamente una función que devuelva, para todo par de estanques, los metros de tubería necesarios para conectarlos directamente. Finalmente, presenta un análisis de los costes temporal y espacial del algoritmo en función del número de estanques.