Capítulo 6

Vuelta atrás

En diversas ocasiones, un problema de optimización sólo se puede resolver *enumerando* todas las posibles soluciones y eligiendo la mejor. El esquema de "vuelta atrás" proporciona una forma elegante y metódica de recorrer virtualmente todo el espacio de soluciones para encontrar la solución óptima. Además, el esquema proporciona algunas ventajas adicionales que permiten acelerar el proceso si la instancia del problema lo permite.

6.1. Esquema algorítmico

En este capítulo se irá explicando el esquema algorítmico de vuelta atrás de manera incremental, empleando el problema de la mochila como hilo conductor, con el objetivo de dar forma inmediatamente a los conceptos relacionados con este esquema. Al final de esta sección se proporcionará un esquema genérico para situar cada componente del mismo.

6.1.1. El problema de la mochila (general)

La versión del problema de la mochila que se resolverá en este capítulo es la versión general, en el cual solo se permite una unidad de cada objeto (*0-1 knapsack problem*). A diferencia de la versión vista en el Capítulo 4, en esta ocasión los pesos pueden ser continuos, lo que impide (o desaconseja) la utilización de programación dinámica. Además, no se pueden fraccionar, como ocurría en el Capítulo 5.

Formalmente, se dispone de una cantidad de peso máximo W y una serie de objetos $i \in \{1, 2, ..., n\}$. Cada objeto añade peso $w_i \in \mathbb{R}$ y valor $v_i \in \mathbb{R}$ a la mochila. El objetivo es maximizar el valor de la mochila sin superar el

peso máximo de la misma. Es decir:

$$\max_{\mathbf{x}} \sum_{i=1}^{n} v_i x_i \,;\, \sum_{i=0}^{n} w_i x_i < W; \mathbf{x} \in \{0,1\}^n \tag{6.1}$$

Codificación

El primer paso para resolver un problema mediante "vuelta atrás" es encontrar una codificación que permita generar todas las soluciones posibles. Normalmente, una codificación adecuada es un vector que pertenezca al espacio de soluciones posible. Es decir, si el problema que se aborda son decisiones sobre un conjunto de n elementos, un vector $\mathbf{x} \in \mathbb{V}^n$ sería adecuado.[1]

En el caso de la mochila, podemos aprovechar la definición del problema donde se incluye un vector $\mathbf{x}$ binario que indica, para cada objeto, si se añade a la mochila o no.

Recorrido: expansión y validez

El paso clave en "vuelta atrás" es generar todas las soluciones posibles metódicamente. El espacio de soluciones se suele enumerar de manera recursiva, formando una estructura de árbol (*árbol de expansión* o *árbol de recorrido*). Aunque hay diversas formas de realizar la expansión, en esta asignatura utilizaremos el recorrido más sencillo e intuitivo: en profundidad.

El algoritmo que tendríamos para este recorrido es el siguiente:

```
función recorrido(W, Wmax, V, Vcurrent, i):
    si i = |W|
        devuelve

    recorrido(W, Wmax-W[i], V, Vcurrent+V[i], i+1)
    recorrido(W, Wmax, V, Vcurrent, i+1)
```

La Fig. 6.1 representa el recorrido realizado por la función propuesta para una instancia del problema de la mochila con 3 objetos. Este árbol contiene las soluciones parciales que contienen los nodos intermedios y las soluciones completas (llamados *nodos hoja*).

El esquema algorítmico de "vuelta atrás" recibe su nombre debido a la forma de enumerar y recorrer las soluciones. Este recorrido tiene forma de árbol, de manera que explorar las soluciones a partir de cierta solución

[1] Aquí $\mathbb{V}$ representa los posibles valores de cada decisión.

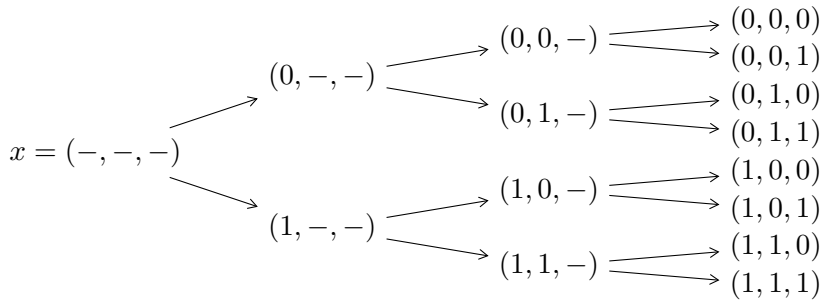


Figura 6.1: Ejemplo de árbol de recorrido para el problema de la mochila con $n = 3$. En cada nodo del árbol se muestra el valor de $\mathbf{x}$.

parcial crea nuevas ramas para este recorrido y para continuar por una rama distinta al evaluar una solución final (nodo hoja) es necesario retroceder a una solución parcial anterior, es decir, *volver atrás* en el recorrido.

Una vez se tiene el recorrido, es importante distinguir entre soluciones factibles (válidas) y no factibles. Esto se puede realizar como una comprobación en los nodos hoja. La condición de factibilidad es independiente del esquema y viene dada específicamente por el problema a resolver.

En el problema de la mochila es necesario descartar (no son factibles) las soluciones cuyo peso supera la capacidad de la mochila. A continuación se muestra nuestro algoritmo con esta comprobación adicional:

```
función recorrido(W, Wmax, V, Vcurrent, i):
    si i = |W|
        si Wmax < 0
            devuelve NO_FACTIBLE
        sino
            devuelve FACTIBLE

    recorrido(W, Wmax-W[i], V, Vcurrent+V[i], i+1)
    recorrido(W, Wmax, V, Vcurrent, i+1)
```

Si el ejemplo de recorrido de la Fig. 6.1 tenemos $W = 30, \mathbf{w} = \{25, 10, 5\}$ nos quedaría el árbol de la Fig. 6.2 con las soluciones no factibles marcadas en rojo.

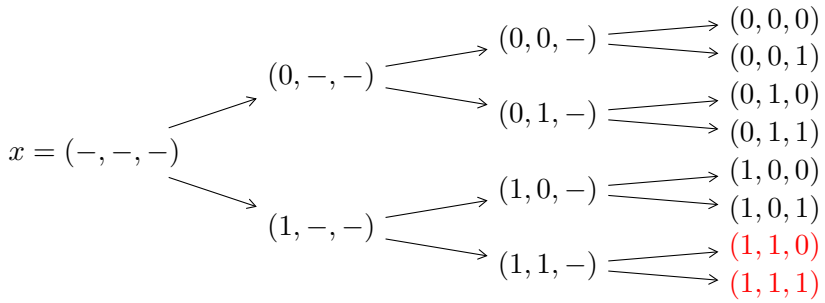


Figura 6.2: Ejemplo de árbol de recorrido para el problema de la mochila con $n = 3, W = 30, \mathbf{w} = \{25, 10, 5\}$. En cada nodo del árbol se muestra el valor de $\mathbf{x}$. En rojo se indican los nodos con soluciones no factibles (no válidas).

Mejor solución

Una vez sabemos que el algoritmo visita todas las soluciones factibles, ya podemos compararlas y guardarnos la mejor posible (optimización). A continuación se muestra el algoritmo de recorrido modificado para guardar la mejor solución. Por simplicidad, asumimos la existencia de un elemento global ajeno a la función llamado "best" y que guardará el mejor valor encontrado hasta el instante actual de la búsqueda. Como la solución que buscamos se guarda en este elemento "best", no es necesario que el algoritmo devuelva el valor.

```
función mejor_solución(W, Wmax, V, Vcurrent, i):
    si i = |W|
        si Wmax < 0
            devuelve
        sino
            si Vcurrent > best
                best := Vcurrent
            devuelve

    mejor_solución(W, Wmax-W[i], V, Vcurrent+V[i], i+1)
    mejor_solución(W, Wmax, V, Vcurrent, i+1)
```

En ocasiones puede ser interesante también buscar la secuencia de decisiones que dan lugar a la solución óptima. Esto puede ser tan sencillo como llevar un registro de las decisiones tomadas en cada llamada y guardar este registro al actualizar la mejor solución:

```
función mejores_decisiones(W, Wmax, V, Vcurrent, i, x):
    si i = |W|
        si Wmax < 0
                devuelve
            sino
                si Vcurrent > best
                    best := Vcurrent
                    bestX := x
                devuelve

    x[i] := 1
    mejores_decisiones(W, Wmax-W[i], V, Vcurrent+V[i], i+1, x)

    x[i] := 0
    mejores_decisiones(W, Wmax, V, Vcurrent, i+1, x)
```

Podas

Con el algoritmo actual podemos asegurar que encontramos la solución óptima, ya que visitamos todas las soluciones posibles y diferenciamos entre las que son factibles y las que no lo son. No obstante, el tiempo de ejecución de este esquema puede llegar a ser muy elevado ya que su complejidad es exponencial respecto a n.

Para visualizar mejor por qué esto es un problema, para obtener una solución al problema de la mochila con 50 objetos usando el algoritmo actual, asumiendo que cada nodo visitado tan solo consume 0.001 segundos, habría que esperar **35.000 años** para obtener la solución óptima. Por esto mismo, evaluar **todas** las soluciones posibles resulta **muy ineficiente**.

En la mayoría de aplicaciones prácticas el tiempo consumido para encontrar la solución óptima es muy importante, ya que los algoritmos se suelen ejecutar con bastante frecuencia. Por este motivo es útil acotar la búsqueda empleando información adicional sobre el problema de forma que se evita visitar caminos del árbol de recorrido que no tengan potencial para mejorar la solución actual. A esta técnica se la denomina "poda" porque consiste en *cortar* ramas del árbol de recorrido.

Validez Sabiendo que, dada una solución parcial, ninguna solución que la contenga va a ser factible, ¿tiene sentido seguir expandiendo a partir de esa solución? Por ejemplo, si mi mochila puede llevar 5 kg y los objetos que

estoy probando a meter acumulan ya más de 5 kg, ¿tiene sentido probar a meter algún otro objeto más?

El primer tipo de poda consiste en restringir la expansión del árbol, evitando explorar caminos que sabemos con antelación que no van a conducir a ninguna solución válida. Normalmente este tipo de poda se puede realizar comprobando las restricciones del problema antes de visitar cada nodo.

En el problema de la mochila esto se puede hacer comprobando si el peso del objeto que se intenta meter en la mochila ya supera el peso restante de la misma. Si no es el caso, se consideran las dos expansiones: cogerlo y no cogerlo; sin embargo, si el objeto no cabe solo se considera la opción de no cogerlo.

A continuación se muestra cómo se puede realizar esta comprobación en el recorrido propuesto, concretamente no se añaden objetos cuando su peso supera el peso disponible:

```
función recorrido_factible(W, Wmax, V, Vcurrent, i):
    si i = |W|
        si Vcurrent > best
            best := Vcurrent
        devuelve

    si Wmax-W[i] >= 0
        recorrido_factible(W, Wmax-W[i], V, Vcurrent+V[i], i
   +1)
    recorrido_factible(W, Wmax, V, Vcurrent, i+1)
```

El árbol de recorrido de la Fig. 6.2 quedaría como se indica en la Fig. 6.3 si se realiza la comprobación de validez durante la expansión, es decir, si se aplica la poda por validez.

Cota optimista Una vez sabemos el valor de una solución factible, ¿de qué nos sirve considerar una solución peor? Por ejemplo, si sabemos que en nuestra mochila podemos reunir un valor total de 8, ¿para qué deberíamos considerar una solución de valor 4 si sabemos que será descartada? Suponiendo que podemos saber qué valor se puede llegar a obtener al expandir un nodo, ¿interesaría expandirlo si ese valor no supera el de la mejor solución encontrada hasta el momento?

El segundo tipo de poda consiste en estimar el mejor valor que se podría alcanzar a partir de una solución parcial. A este valor se llama cota optimista. La cota optimista de un nodo debe ser **igual o mejor** que la mejor solución que se pueda encontrar recorriendo las ramas que salen de dicho nodo ya que

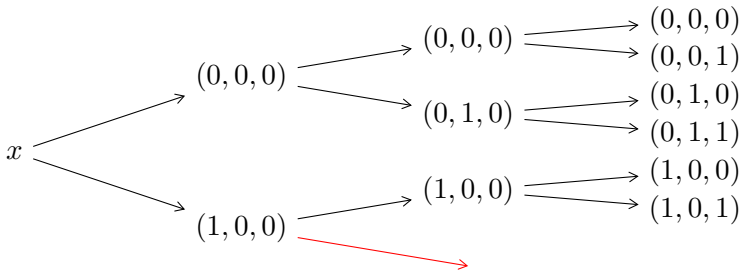


Figura 6.3: Ejemplo de árbol de recorrido para el problema de la mochila con $n = 3, W = 30, \mathbf{w} = \{25, 10, 5\}$ expandiendo únicamente las ramas factibles. En cada nodo del árbol se muestra el valor de $\mathbf{x}$. En rojo se indican los nodos y/o aristas que se han podado.

se empleará para descartar ramificaciones del recorrido. Es decir, si la cota optimista de un nodo es peor o igual que la mejor solución actual, no vale la pena expandir dicho nodo. Si esta propiedad no se cumple, se podrían estar ignorando soluciones potencialmente óptimas.

Para asegurar esta propiedad, normalmente se calculan este tipo de cotas relajando restricciones del problema. A continuación se comentan dos posibles cotas para el problema de la mochila:

- Asumir que se pueden coger todos los objetos restantes, ignorando el límite de peso.
- Asumir que se pueden fragmentar los objetos restantes. Entonces coger, de entre los objetos restantes, los que tengan mayor valor por unidad de peso primero hasta llenar la mochila.

La primera cota relaja la restricción del límite de peso, dando un valor que no puede llegar a ser peor que la mejor solución posible ya que si cupieran todos los objetos restantes, la cota optimista tendría el mismo valor que esta solución, que sería la mejor de su rama. Demostrar que esta cota es optimista resulta trivial ya que cualquier solución que contenga al menos los mismos objetos que otra (objetos específicos, no cantidad de objetos) siempre tendrá el mismo valor o mayor que esta.

La segunda cota relaja la restricción de no fraccionar objetos. Al trabajar con objetos fraccionados, se puede tratar el problema como si se dispusiese de objetos, que son fragmentos de los originales, cuyo peso es una unidad infinitesimal y su valor es proporcional al del objeto al que pertenecen. Al tener

todos los fragmentos de objeto el mismo peso, escoger los de mayor valor primero resulta óptimo. Esto se puede demostrar fácilmente: Asumiendo que tenemos 3 objetos cualesquiera con pesos W_1, W_2, W_3 y valores V_1, V_2, V_3, siempre que se cumpla que el tercer objeto tiene mayor valor por unidad de peso que los otros dos ($\frac{V_1}{W_1} < \frac{V_3}{W_3}$ y $\frac{V_2}{W_2} < \frac{V_3}{W_3}$), ocupar una parte de la mochila W_m con cualquiera de los dos primeros objetos proporcionará menor valor que ocuparla con el tercero ya que la desigualdad se sigue cumpliendo al multiplicar ambos lados por el mismo factor ($W_m \frac{V_1}{W_1} < W_m \frac{V_3}{W_3}$ y $W_m \frac{V_2}{W_2} < W_m \frac{V_3}{W_3}$). Además, cualquier combinación lineal de los dos primeros objetos para rellenar el seguirá aportando menos valor a la mochila que introducir el tercer objeto: $W_{m1} \frac{V_1}{W_1} + W_{m2} \frac{V_2}{W_2} < W_m \frac{V_3}{W_3}$ para $W_{m1} + W_{m2} = W_m$.

Al conocer una cota optimista del valor que se puede obtener expandiendo determinadas ramas, es posible evitar expandir el árbol de recorrido en determinadas situaciones para ahorrar recursos (temporales y espaciales). Normalmente se suele comparar la cota optimista con el mejor valor encontrado hasta el momento ya que no interesa encontrar soluciones que empeoren la actual, solo aquellas que puedan mejorarla. Cuanto más ajustada sea la cota (más cercana al valor real) más recursos ahorrará. A continuación se muestran las cotas mencionadas en esta sección en el mismo orden:

```
función optimista_peso(V, Vcurrent, i):
    devuelve Vcurrent + sum(V[i:])
```

```
función optimista_fracción(W, Wmax, V, Vcurrent, i):
    ordenar W[i:] y V[i:] de mayor a menor según V[j]/W[j] de
    j=i a j=|W|

    mientras que i <= |W| y Wmax > 0:
        si Wmax >= W[i]
            Vcurrent := Vcurrent+V[i]
            Wmax := Wmax-W[i]
        sino
            Vcurrent := Vcurrent + Wmax * V[i]/W[i]
            Wmax := 0

    devuelve Vcurrent
```

El algoritmo que resuelve el problema de la mochila quedaría como se muestra a continuación:

```
función mochila_con_poda(W, Wmax, V, Vcurrent, i):
    si i = |W|
```

```
        si Vcurrent > best
            best := Vcurrent
        devuelve

    si optimista_fracción(W, Wmax, V, Vcurrent, i) <= best
        devuelve

    si Wmax-W[i] >= 0
        mejor_solución(W, Wmax-W[i], V, Vcurrent+V[i], i+1)
    mejor_solución(W, Wmax, V, Vcurrent, i+1)
```

Cota pesimista La poda por cota optimista no empieza a tener efecto hasta que no se conoce el valor de una solución factible al problema. Pero, ¿es realmente necesario explorar el espacio de soluciones hasta encontrar una solución factible para empezar a utilizar la cota optimista? Es posible aprovechar la cota optimista desde el inicio de la búsqueda?

A pesar del beneficio de emplear una cota optimista, por muy ajustada que sea, no empieza a aplicarse hasta que se encuentra una solución lo suficientemente buena, como ya se ha comentado arriba. Por este motivo, es también bastante común iniciar la búsqueda con una solución válida conocida. A esto se llama cota pesimista ya que "como muy mal, la solución que encuentre tiene que ser esta". Esta solución debe ser válida, por lo que no se debe encontrar relajando las restricciones del problema. Al igual que con la cota optimista, cuanto más se acerque la cota pesimista a la solución óptima, más recursos ahorrará durante el recorrido ya que permitirá podar más nodos.

Una posible cota pesimista puede ser coger los objetos en orden de mayor valor por unidad de peso a menor, sin fragmentarlos:

```
función pesimista_greedy(W, Wmax, V, Vcurrent, i):
    ordenar W[i:] y V[i:] de mayor a menor según V[j]/W[j] de
    j=i a j=|W|

    mientras que i <= |W| y Wmax > 0:
        si Wmax >= W[i]
            Vcurrent := Vcurrent+V[i]
            Wmax := Wmax-W[i]
```

Hay cotas pesimistas que se pueden calcular durante la búsqueda, pero normalmente el coste que añaden a cada nodo no compensa el tiempo de

búsqueda que ahorran. Esta cota, por ejemplo, se podría calcular en cada nodo, pero añadiría un coste bastante elevado, de manera que es mejor usar esta cota para obtener una solución inicial para adelantar el efecto de la cota optimista. Los parámetros iniciales de esta función serán en este caso *Wmax=Wmax*, *Vcurrent=0* e *i=1*.

Es posible también calcular varias cotas pesimistas y quedarse la de mayor valor, cuando el espacio de búsqueda es potencialmente muy elevado, emplear operaciones más complejas antes de empezar la búsqueda para reducir más el tiempo que esta pueda consumir se vuelve más interesante.

Una posible cota pesimista es generar varias soluciones **válidas** aleatorias y quedarse con la de mejor valor. No obstante, esta cota no asegura una solución inicial suficientemente buena, por lo que no suele emplearse si se dispone de otra cota pesimista que aproveche las propiedades del problema.

Otros

A parte de las estrategias ya comentadas, es posible emplear conocimiento más específico del problema para acortar el tiempo de búsqueda y reducir el número de nodos visitados. Dos estrategias bastante conocidas son los pre-cálculos y el orden de expansión o de recorrido.

Cálculos previos Los cálculos previos (o pre-cálculos) consisten en adelantar el cálculo de todo aquello que no cambie durante la búsqueda. Muchas cotas optimistas es posible calcularlas antes de iniciar la búsqueda ya que no dependen del recorrido actual, tan solo del recorrido restante.

Al realizar estos cálculos antes de iniciar la búsqueda, se puede evitar la repetición de determinados cálculos, potencialmente ahorrando una gran cantidad de recursos.

Orden de recorrido Alterar el orden de recorrido es otra técnica bastante usada. En el problema de la mochila es posible ahorrar tiempo de cómputo si se evalúan los objetos en distinto orden. Por ejemplo, se podrían recorrer en orden decreciente según el valor por unidad de peso de los objetos. Este orden también podría ahorrar tiempo a la hora de calcular la cota optimista "optimista_fracción" propuesta anteriormente y la cota pesimista voraz propuesta.

Poda			O	Nodos		Tiempo	
V	O	P		Total	Hojas	Pesimista	Búsqueda
✗	✗	✗	✗	67108863	33554432	0.00e+00	1.58e+01
✓	✗	✗	✗	65359897	32110044	0.00e+00	1.57e+01
✗	✓	✗	✗	557	132	0.00e+00	1.36e-03
✓	✓	✗	✗	354	21	0.00e+00	1.30e-03
✗	✓	✓	✗	175	10	8.76e-06	6.60e-04
✓	✓	✓	✗	171	6	9.12e-06	6.61e-04
✗	✓	✗	✓	221	80	0.00e+00	1.52e-04
✓	✓	✗	✓	81	3	0.00e+00	1.01e-04
✗	✓	✓	✓	221	80	2.86e-06	1.60e-04
✓	✓	✓	✓	81	3	2.74e-06	1.01e-04

Cuadro 6.1: Tabla comparativa de la eficiencia empírica usando (✓) o sin usar (✗) cada una de las componentes del esquema algorítmico de vuelta atrás. Las componentes son: Poda (V-validez, O-optimista, P-pesimista) y Orden. La cota optimista considerada en esta tabla es "optimista_fracción".

Eficiencia

En esta sección se muestra una comparación empírica de la eficiencia del algoritmo empleando distintas versiones. La tabla 6.1 muestra distintas métricas como el número de nodos visitados y el tiempo de ejecución (media de 4 ejecuciones) de cada una de los componentes del algoritmo.

Como se puede comprobar, la cota optimista es extremadamente importante ya que permite reducir en gran cantidad el número de nodos visitados. La cota pesimista es bastante importante también, pero no se puede aplicar sin una cota optimista. Por otro lado, expandir únicamente los nodos que cumplen las restricciones del problema puede acortar bastante la búsqueda si se emplea un orden de recorrido apropiado. No obstante, este orden de recorrido puede llegar a anular el efecto de algunos componentes, por ejemplo evaluar los objetos en orden descendente de valor por unidad de peso anula el efecto de la cota pesimista ya que la primera solución probada es igual que la cota pesimista.

6.1.2. Esquema general

A continuación se muestra una versión general de este esquema algorítmico con los distintos mecanismos explicados anteriormente. En este pseudocódigo se pueden ver 4 partes: gestión de hojas, poda con almacén, poda

con cota optimista y expansión. La cota pesimista se calcularía antes de iniciar la búsqueda.

```
función vuelta_atrás(...):
    si es hoja
        si es mejor que best
            best := solución actual
        devuelve

    si este nodo está en el almacén
        si el valor almacenado es mejor o igual que el actual
            return
    almacenar el valor actual asociado al nodo actual

    si cota optimista es peor o igual que best
        devuelve

    para cada expansión posible:
        si es válida
            vuelta_atrás(...)
```

6.2. Camino óptimo en un laberinto

En esta sección se propone otro problema de ejemplo y una solución al mismo, para afianzar los conceptos explicados en este capítulo.

Supongamos que queremos resolver un laberinto de tamaño $n \times m$, empezando en el nodo (Sx, Sy) y acabando en el nodo (Ex, Ey). Tan solo se permite el desplazamiento vertical y horizontal, y no se pueden atravesar paredes. El coste de atravesar las distintas casillas (entrar en ellas) puede ser distinto para cada una, este coste se puede consultar en la matriz $L_{n \times m}$. Las paredes tienen coste -1 y ninguna casilla puede tener coste 0, además, el coste de la casilla inicial se puede ignorar al inicio del recorrido, pero no las siguientes visitas a esta casilla. El objetivo es buscar el camino de coste mínimo. Para resolver este problema (y los demás) con este esquema algorítmico se recomienda seguir la misma lógica que en la sección anterior, empezando por construir una búsqueda exhaustiva que visite todas las soluciones o las soluciones válidas y se guarde la mejor, a continuación diseñar una o varias cotas optimistas, elaborar alguna cota pesimista y finalmente evaluar la posibilidad de emplear un almacén o cambiar el orden de recorrido. También

se recomienda realizar los cálculos previos que se puedan justificar (ahorren más cómputo total del que necesitan para ser calculados al principio).

La codificación de la solución de este problema puede ser una matriz de $n \times m$ que contenga 1 en las casillas que pertenezcan al camino de la solución y 0 en las demás. Otra posible codificación es una lista con las posiciones (x, y) en el orden en que aparecen en la solución. Finalmente, otra posible codificación es una lista de direcciones (norte, sur, este, oeste) que conformen el camino hacia la casilla final desde la casilla inicial. Por simplicidad se ha decidido emplear la primera opción, ya que permite comprobar en tiempo constante si se ha visitado ya una casilla durante el recorrido.

A continuación se muestra la función que realiza la búsqueda:

```
función laberinto(L, x, y, c):
    si (x = Ex) y (y = Ey)
        si c < best
            best := c
        devuelve

    si (almacén[x][y] != -1) y (almacén[x][y] <= c)
            devuelve

    almacén[x][y] := c

    para cada par (vx, vy) en [(1, 0), (0, 1), (-1, 0), (0,
   -1)]:
        nx, ny := x+vx, y+vy

        si no 0 <= nx < |L|: salta esta iteración
        si no 0 <= ny < |L[nx]|: salta esta iteración
        si L[nx][ny] = -1: salta esta iteración
        si visitado[nx][ny]: salta esta iteración

        nc := c+L[nx][ny]

        si nc+optimista[nx][ny] < best
            visitado[nx][ny] := True
            laberinto(L, nx, ny, nc)
            visitado[nx][ny] := False
```

Donde *visitado* es una matriz de valores booleanos que indican si ya se ha pasado por una casilla durante el recorrido, para evitar bucles y porque si

ya has pasado por una casilla, volver a pasar por la misma tan solo aumenta el coste sin aportar progreso hacia la casilla objetivo.

Almacén contiene el coste mínimo de llegar a cada casilla desde la casilla inicial. Si ya se ha visitado anteriormente una casilla, el coste total de alcanzar la solución es el coste mínimo desde esa casilla hasta la final sumado al coste de alcanzar la misma. La primera parte es la misma para ambas visitas, por lo que si se ha alcanzado la casilla anteriormente con un coste menor, el coste actual no puede ser mejor que el que se alcanzó a raíz de la anterior visita por lo que se puede detener la expansión del nodo en cuestión.

La cota optimista es el coste de alcanzar la solución en el número mínimo de movimientos. Para cada movimiento en horizontal se suma el coste mínimo de la columna que se visita (sin contar las paredes, que cuestan -1) y cada movimiento en vertical suma el coste mínimo de la fila que se visita (también sin contar las paredes). De esta forma, se asegura que la cota es optimista ya que si el camino óptimo contiene un movimiento que aumenta la distancia euclídea entre la casilla actual y el destino, debe contener otro movimiento que lo contrarreste (que lo acerque al destino), la cota optimista considera el mínimo de esos dos, o incluso un valor menor y tan solo una vez. La restricción que relaja esta cota es la de que la solución debe ser un camino válido (esta cota considera la posibilidad de moverse a casillas no contiguas). Un ejemplo de esta cota en un laberinto de 3 filas y 3 columnas, empezando en la esquina superior izquierda y acabando en la esquina inferior derecha, en el que los costes sean los siguientes:

1	10	1
10	10	1
10	1	10

El valor de la cota optimista sería 4: derecha (coste 1 por la casilla (3, 2)), derecha (coste 1 por la casilla (1, 3)), abajo (coste 1 por la casilla (2, 3)), abajo (coste 1 por la casilla (3, 2)). Si en una de las columnas o filas todos los valores fueran 10, atravesar dicha fila o columna siempre tendrá coste 10, por lo que la cota no se alejaría tanto como una cota trivial como puede ser el coste mínimo multiplicado por la distancia euclídea entre cada casilla y la casilla final.

La cota pesimista es el algoritmo de Dijkstra. Este algoritmo resuelve el problema de encontrar el camino de coste mínimo entre dos nodos de un grafo. Como este problema se puede resolver planteándolo como una búsqueda de camino de menor coste, esta cota elimina completamente la necesidad de realizar la búsqueda. No obstante, el tiempo total de ejecución

puede llegar a ser más elevado en determinadas ocasiones ya que el algoritmo de Dijkstra tiene complejidad $O(|V|^2)$ donde $|V| = n \cdot m$ es el número de vértices del grafo (casillas del laberinto), por lo que la complejidad de esta cota sería $O(n^2 \cdot m^2)$.

Es importante destacar que en este ejemplo, la cota pesimista puede llegar a consumir bastante tiempo. A menudo hay que valorar la posibilidad de emplear distintas estrategias en función del tamaño del problema. En la Sección **??** se comenta en más detalle en esta cuestión y se ponen ejemplos para clarificar esta idea.

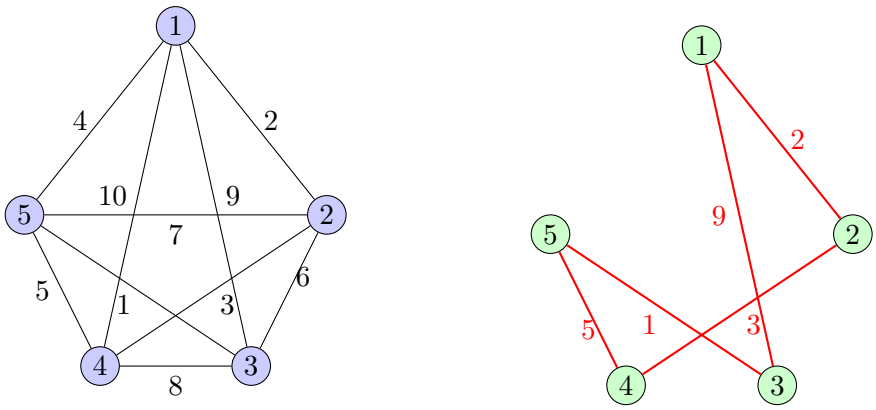
Por otro lado, también es importante tener en cuenta que un correcto análisis previo del problema a resolver es imprescindible. Este problema se puede resolver empleando el algoritmo de Dijkstra, un algoritmo voraz empleado para encontrar caminos de coste mínimo. A menudo hay problemas que se pueden resolver con esquemas algorítmicos más sencillos y es importante identificar cuándo y si es más eficiente emplear dichos esquemas antes de aplicar algoritmos de vuelta atrás.

6.3. El viajante de comercio

El problema del viajante de comercio (Travelling Salesman Problem, o TSP) trata de resolver la siguiente situación: Un comerciante quiere visitar una serie de ciudades y volver a la ciudad inicial. La intención del comerciante es visitar cada ciudad únicamente una vez. El coste de desplazamiento entre cada par de ciudades puede ser distinto. El objetivo del comerciante es minimizar el coste total (suma de los costes de desplazamiento) del camino que cumple las condiciones ya comentadas.

Este problema se suele modelar con grafos. Cada ciudad es un vértice y el camino entre cada par de ciudades es una arista. Dependiendo de qué versión del problema se trate de resolver, el coste del camino entre dos ciudades puede ser el mismo en ambas direcciones (grafo no dirigido) o ser distinto (grafo dirigido). Además, también es posible que no haya un camino entre todos los pares de ciudades, pero siempre debe haber al menos un camino de entrada y uno de salida a cada ciudad. Para no complicar mucho las cotas propuestas, se considerará la versión modelada con un grafo no dirigido. Un ejemplo del problema formalizado con grafos y su solución se presenta en la Fig. 6.4.

La solución de este problema se puede codificar como una lista que contenga las ciudades visitadas en orden cronológico. Esta lista debe empezar y acabar por la ciudad inicial y no tener otras ocurrencias de la misma, el



(a) Grafo original (b) Solución.

Figura 6.4: Ejemplo visual del problema del viajante de comercio. El nodo de partida es (1).

resto de ciudades deben aparecer exactamente una vez y para cada par de ciudades contiguas (c_i, c_{i+1}), debe existir un camino que lleve de la ciudad c_i a la ciudad c_{i+1}. El coste de la solución es la suma de los costes de cada par de ciudades (c_i, c_{i+1}). En el ejemplo de la Fig. 6.4, el recorrido de mínimo coste es el formado por $1 \to 2 \to 4 \to 5 \to 3 \to 1$, con peso 20.

Una propuesta de solución a este problema mediante vuelta atrás es la siguiente:

```
función viajante(C, actual, coste, restantes):
    si restantes = 0
        si coste < best
            best := coste
        devolver

    para cada destino aún por visitar:
        si existe camino de ciudad actual a ciudad destino
            si restantes = 1 o destino != inicio
                marcar destino como visitado
                nuevoCoste := coste + C[actual][destino]
                si nuevoCoste + optimista < best
                    viajante(C, destino, nuevoCoste, restantes
   -1)
                marcar destino como no visitado
```

En este caso, C es una matriz cuadrada que contiene el coste de desplazarse de una ciudad origen a una ciudad destino para cada entrada de la matriz $C[origen][destino]$. Durante la expansión tan solo se permite visitar la ciudad inicial si no quedan ciudades por visitar.

Una posible cota optimista puede ser la suma del coste mínimo de salir de cada ciudad aún por visitar, sin contar la que se está a punto de visitar, para la cual se usa el coste de desplazarse a dicha cuidad desde la ciudad actual. Otra posible cota optimista podría ser calcular el árbol de expansión de coste mínimo antes de iniciar la búsqueda, utilizando el algoritmo de Kruskal (ver Capítulo 5).

Por otro lado, una posible cota pesimista para este problema podría ser calculada mediante el siguiente proceso: empezar con un grafo vacío. Para cada ciudad que tenga únicamente dos ciudades conectadas, añadir al grafo estas aristas y los nodos que conectan. Estos nodos y aristas deben pertenecer al camino óptimo obligatoriamente ya que el camino debe contener todos los nodos y cada ciudad debe ser visitada una única vez, por lo que si un nodo solo tiene una arista en este grafo, el camino final contendrá dos veces el otro nodo que conecta dicha arista. En este instante se pueden tener una o varias componentes conexas. En estas componentes conexas deben haber uno o dos nodos que solo tengan una arista, estos nodos se llamarán "punta" de la componente conexa a la que pertenecen. Para añadir el resto de las ciudades se procede de la siguiente manera: de entre las aristas que no pertenecen aún al grafo y conecten un nodo punta con otro de otra componente conexa o con una ciudad que aún no está en el grafo, se añade la de menor coste. Cuando no queden ciudades por añadir, se añaden las aristas de menor coste que unan dos componentes conexas a través de sus puntas, cuando solo quede una componente conexa, se añadirá la arista de menor coste que una sus puntas. Si en algún momento del proceso ninguna arista se puede añadir, no existe solución.

Con las estructuras de datos apropiadas se puede calcular esta cota pesimista sin acarrear una complejidad computacional exponencial. Por ejemplo, si en todo momento se conserva una lista que contenga los nodos punta y se conozca a qué componente conexa pertenece cada uno, es posible mantener una lista ordenada con las aristas que conectan estos nodos con otros pero aún no pertenecen al grafo.

6.4. Consideraciones adicionales

Es importante tener en cuenta algunas consideraciones al diseñar algoritmos con el esquema de vuelta atrás. La primera de las consideraciones es que el recorrido, y por tanto la eficiencia, puede depender bastante de la codificación que se haya empleado. El número de nodos hoja de la búsqueda exhaustiva inicial se puede calcular, por lo que es recomendable asegurar que las expansiones realizadas cubran todas las soluciones posibles empleando tallas pequeñas ya que si algún detalle de la expansión produce como consecuencia la discriminación de soluciones válidas de manera no intencionada, es posible que se descarte la solución óptima y este hecho se pase por alto. Es necesario por tanto, que se puedan recorrer todas las soluciones válidas de manera estructurada y ordenada, evitando visitar soluciones repetidas o dejar soluciones prometedoras sin visitar.

En cuanto a la complejidad, los algoritmos de vuelta atrás suelen tener una cota superior de complejidad exponencial ($O(x^n)$) que no se puede reducir de ninguna forma ya que a fin de cuentas el espacio de soluciones se expande exponencialmente en función de la talla del problema. No obstante, con las cotas apropiadas, la complejidad promedio puede llegar a disminuir en gran cantidad. Por este motivo es importante evaluar adecuadamente los algoritmos implementados. Además, como ya se ha comentado en capítulos anteriores, las cotas de complejidad son límites asintóticos, por lo que es posible que para ciertas tallas, un algoritmo cuya cota de complejidad sea mayor resuelva el problema en menos tiempo.