CAPÍTULO

23 Estructura de partición

Este capítulo describe una estructura de datos eficiente para resolver el problema de la equivalencia: la estructura de partición. Esta estructura se implementa de forma sencilla; cada rutina requiere sólo unas líneas de código. Dicha implementación también es extremadamente rápida, requiriendo un tiempo constante en promedio por operación. Esta estructura de datos también resulta interesante desde un punto de vista teórico, ya que su análisis es muy difícil; la forma funcional del coste en el caso peor no se parece a ninguna de las discutidas hasta ahora en este libro.

En este capítulo veremos:

- Dos aplicaciones simples de la estructura de partición.
- Cómo podemos implementar la estructura de partición con un esfuerzo mínimo.
- Cómo mejorar la velocidad de las estructuras de partición utilizando dos observaciones simples.
- Un análisis del tiempo de ejecución de una implementación rápida de la estructura de partición.

23.1 Relaciones de equivalencia

Una *relación* está definida sobre un conjunto indicando si cada par de elementos están o no relacionados. Una *relación de equivalencia* es reflexiva, simétrica y transitiva.

Una *relación* R sobre un conjunto S se define como un conjunto de pares de elementos (a, b) con $a, b \in S$. Si $(a, b) \in R$ escribiremos $a\ R\ b$ y diremos que a está relacionado con b. Una *relación de equivalencia* es una relación R que satisface las siguientes tres propiedades:

1. *Reflexiva*: $a\ R\ a$ es cierto para todo $a \in S$.
2. *Simétrica*: $a\ R\ b$ si y sólo si $b\ R\ a$.
3. *Transitiva*: $a\ R\ b$ y $b\ R\ c$ implican que $a\ R\ c$.

La conectividad eléctrica, donde todas las conexiones se realizan mediante cables eléctricos, es una relación de equivalencia. Claramente, la relación es reflexiva, ya que cualquier componente está conectada a sí misma. Si a está conectado eléctricamente a b, entonces b está conectado a a, por lo que la relación es simétrica. Finalmente, si a está conectado a b y a su vez b está conectado a c, entonces a estará conectado a c.

De la misma manera, la conectividad a través de una red bidireccional forma clases de equivalencia de componentes conectadas. Sin embargo, si las conexiones de la red fueran dirigidas (es decir, una conexión de v a w no implica una de w a v), entonces no tendríamos una relación de equivalencia porque fallaría la propiedad de simetría. Un ejemplo es una relación en la cual la ciudad a está relacionada con la ciudad b si es posible viajar de a a b mediante una carretera. Esta relación es una relación de equivalencia si las carreteras son de doble sentido.

23.2 Equivalencias dinámicas y dos aplicaciones

Dada una relación de equivalencia, que denotaremos por el símbolo $\sim$, el problema inmediato que surge es decidir, dados a y b cualesquiera, si $a \sim b$. Si la relación se almacena como un vector bidimensional de valores booleanos, esto puede decidirse en tiempo constante. El problema es que las relaciones generalmente se definen de forma implícita, en vez de explícitamente.

Por ejemplo, si consideramos una relación de equivalencia definida sobre el conjunto con cinco elementos $\{a_1, a_2, a_3, a_4, a_5\}$, hay en total 25 pares de elementos, cada uno de los cuales puede, en principio, pertenecer a la relación o no. Sin embargo, de la información $a_1 \sim a_2$, $a_3 \sim a_4$, $a_1 \sim a_5$ y $a_4 \sim a_2$ se sigue que todos los elementos han de estar relacionados entre sí. Nos gustaría ser capaces de inferir rápidamente algo como esto.

La *clase de equivalencia* de un elemento x en el conjunto S es el subconjunto de S que contiene a todos los elementos que están relacionados con x. Las clases de equivalencia forman una estructura de partición. Las operaciones básicas necesarias para manipular las estructuras de partición son `unir` y `buscar`.

La *clase de equivalencia* de un elemento $x \in S$ es el subconjunto de S que contiene a todos los elementos relacionados con x. Observemos que las clases de equivalencia forman una partición de S: todo miembro de S aparece exactamente en una clase de equivalencia. Para decidir si $a \sim b$, sólo necesitamos comprobar si a y b están en la misma clase de equivalencia. Esto proporciona la estrategia para resolver el problema de la equivalencia.

Consideremos un tipo partición cuyos elementos se construirán a partir de la partición trivial que viene dada por una colección de N conjuntos, cada uno con un elemento. Esta partición inicial corresponde a la relación de equivalencia trivial. Cada conjunto tiene un elemento distinto, por lo que $S_i \cap S_j = \varnothing$ obteniéndose, por tanto, una partición.

Hay dos operaciones permitidas. La primera es `buscar`, que devuelve el nombre del conjunto (es decir, la clase de equivalencia) que contiene a un elemento dado. La segunda operación añade pares a la relación representada. Si queremos añadir el par (a, b) a la relación, vemos primero si a y b ya están relacionados. Esto se consigue llamando a `buscar` con a y b, comprobando si están en la misma clase de equivalencia. Si no es así, llamamos a `unir`. Esta operación mezcla las clases de equivalencia que contienen a a y a b, en una nueva clase de equivalencia. Desde el punto de vista conjuntista, el efecto es la creación de un nuevo conjunto $S_k = S_i \cup S_j$, destruyendo los originales, con lo que se preserva el hecho de que todos los conjuntos de la partición sean disjuntos. La estructura de datos que contiene estas operaciones se denomina en ocasiones *estructura de partición unir/buscar*. El término *algoritmo* `unir/buscar` se refiere al proceso de peticiones de unión y búsqueda utilizando una estructura de partición.

El algoritmo es *dinámico* en el sentido de que, durante su ejecución, los conjuntos de la partición pueden cambiar vía la operación `unir`. El algoritmo debe

también ir produciendo sus respuestas en tiempo de ejecución, en el sentido de que cuando se realiza una búsqueda, debe mostrarse la respuesta antes de proseguir. Otra posibilidad sería permitir que la secuencia completa de resultados se mostrara tras la ejecución de todas las llamadas a `buscar` y `unir`. En tal caso, la respuesta que proporcionará cada `buscar` debería ser consistente con las uniones realizadas antes de la búsqueda, aunque el algoritmo procese todas las peticiones antes de producir la primera respuesta. Esta diferencia es parecida a la existente entre un examen escrito (donde generalmente sólo se tiene que dar la respuesta antes de un tiempo límite) y un examen oral (donde hay que dar la respuesta a cada pregunta antes de conocer la siguiente).

En el algoritmo cada respuesta debe darse antes de ver la siguiente petición.

Observemos que no se realiza ninguna operación comparando los valores de los elementos sino que sólo se necesita su localización. Por esta razón, podemos asumir que todos los elementos han sido numerados de forma secuencial partiendo de 0 y que la numeración puede determinarse fácilmente utilizando un esquema hash.

Los elementos del conjunto se numeran secuencialmente comenzando en 0.

Antes de describir cómo implementar las operaciones `buscar` y `unir`, esta sección proporciona dos aplicaciones de la estructura de datos.

23.2.1 Aplicación #1: árboles de recubrimiento mínimo

Un *árbol de recubrimiento* de un grafo no dirigido G es un árbol formado a partir de las aristas del grafo que conecta todos los vértices de G. Observe que, al contrario de lo que sucedía en los grafos del Capítulo 14, ahora una arista (u, v) de G es idéntica a la arista (v, u). El coste de un árbol de recubrimiento es la suma de los costes de las aristas en el árbol; el problema del *árbol de recubrimiento mínimo* consiste en buscar el árbol de recubrimiento de menor coste. Como mostraremos en breve, la conectividad de un grafo puede comprobarse como parte del cálculo del árbol de recubrimiento mínimo.

El *árbol de recubrimiento mínimo* es un subgrafo conexo de G que abarca a todos los vértices, con un coste total mínimo.

En la Figura 23.1, el grafo de la derecha es un árbol de recubrimiento mínimo del grafo de la izquierda (en este caso es el único, pero esto no tiene por qué ser cierto en general, en particular si el grafo tiene muchas aristas con el mismo coste). Observe que el número de aristas del árbol de recubrimiento mínimo es $|V| - 1$. El árbol de recubrimiento mínimo es un *árbol* porque es acíclico; es *de recubrimiento* porque cubre todos los vértices; y es *mínimo*, por razones obvias. Supongamos que tenemos que conectar varias ciudades mediante carreteras, minimizando el coste total de la construcción, con la restricción de que sólo se puede cambiar de carretera en una ciudad (en otras palabras, no se permiten cruces adicionales). En tal caso necesitamos resolver un problema de árbol de recubrimiento mínimo, en el que cada vértice es una ciudad y cada arista representa el coste de construir una carretera entre las ciudades que conecta.

Añadir cruces en lugares arbitrarios daría lugar al *problema del árbol de Steiner*, el cual es mucho más difícil de resolver. Por otra parte, entre otros resultados, se puede demostrar que si el coste de una conexión es proporcional a la distancia euclídea, el árbol de recubrimiento mínimo es, como mucho, un 15 por cierto más costoso que el árbol mínimo de Steiner. Esto significa que el árbol de recubrimiento mínimo, que es fácil de calcular, proporciona una buena aproximación al árbol mínimo de Steiner, que es difícil de calcular.

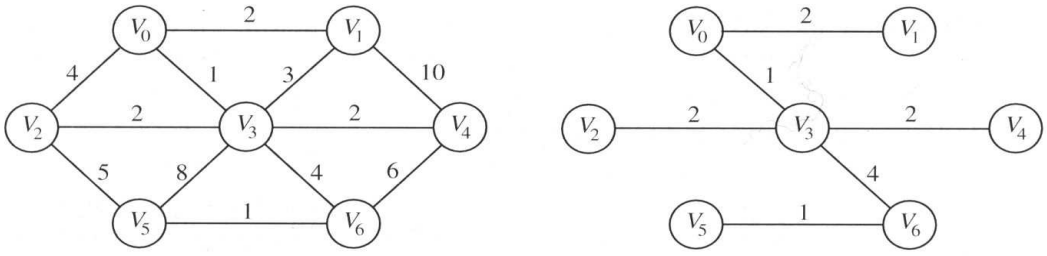

Figura 23.1 Un grafo G (a la izquierda) y su árbol de recubrimiento mínimo.

El *algoritmo de Kruskal* para el cálculo del árbol de recubrimiento mínimo selecciona aristas en orden creciente de costes. Se añade una arista al árbol si con ello no se crea un ciclo.

Un algoritmo sencillo, comúnmente denominado *algoritmo de Kruskal*, consiste en seleccionar las aristas en orden creciente de coste, incorporando una arista si con ello no se introduce un ciclo. Formalmente, el algoritmo de Kruskal mantiene un bosque, esto es, una colección de árboles. Inicialmente, hay $|V|$ árboles cada uno con un único nodo. Al añadir una arista se unen dos árboles en uno. Cuando el algoritmo termina, hay sólo un árbol, y éste es el árbol de recubrimiento mínimo[1]. Contando el número de aristas seleccionadas, podemos determinar cuándo debe darse por concluido el algoritmo.

La Figura 23.2 muestra la acción del algoritmo de Kruskal sobre el grafo de la Figura 23.1. Las cinco primeras aristas son todas incorporadas porque no crean ciclos. Las dos aristas siguientes (v_1, v_3) (de coste 3) y (v_0, v_2) (de coste 4), son rechazadas porque crearían un ciclo en el grafo. La arista considerada a continuación es incorporada. Ya que es la sexta arista y el grafo tiene siete vértices, podemos dar por concluida la aplicación del algoritmo.

Se puede ordenar las aristas o utilizar una cola de prioridad.

Para generar el orden en que serán examinadas las aristas sería suficiente con ordenarlas. Podemos hacerlo con un coste $|E| \log |E|$, tras lo que iríamos recorriendo el vector ordenado de aristas. De forma alternativa, se podría construir una cola de prioridad con las $|E|$ aristas y obtener repetidamente las aristas llamando a `borrarMin`. Aunque el coste en el caso peor no cambia, utilizar una cola de prioridad es a veces mejor porque cuando los grafos son aleatorios el algoritmo de Kruskal tiende a utilizar sólo una pequeña fracción del total de aristas. Por supuesto, en el caso peor, siempre es posible que haga falta probar con todas las aristas. Por ejemplo, si añadieramos un vértice extra v_8 y una arista (v_5, v_8) de coste 100, tendríamos que examinar todas las aristas. En este caso, utilizar directamente quicksort sería más rápido. En consecuencia, la elección entre la cola de prioridad y una ordenación inicial debe hacerse en base a una especulación sobre cuántas aristas serán examinadas a la postre.

La comprobación de la creación o no de ciclos se hace utilizando una estructura de partición.

Más interesante es la cuestión de cómo decidimos si una arista (u, v) debe aceptarse o rechazarse. Claramente, añadir la arista (u, v) provocaría un ciclo si (y sólo si) u y v estuvieran ya conectados en el bosque de recubrimiento actual. Por tanto, mantendremos las componentes conexas del bosque de recubrimiento en una estructura de partición. Inicialmente, cada vértice está en una clase distinta. Si u y v están en la misma componente, la arista se rechaza porque u y v ya están conectados. En otro caso, se selecciona la arista y se realiza una operación de `unir` con los dos conjuntos disjuntos que contienen a u y a v. Esto es exactamente lo que necesitamos hacer porque una vez se ha añadido la arista (u, v) al bosque de

[1] Si el grafo no es conexo, el algoritmo terminará con más de un árbol: cada árbol representará un árbol de recubrimiento mínimo de cada componente conexa del grafo.

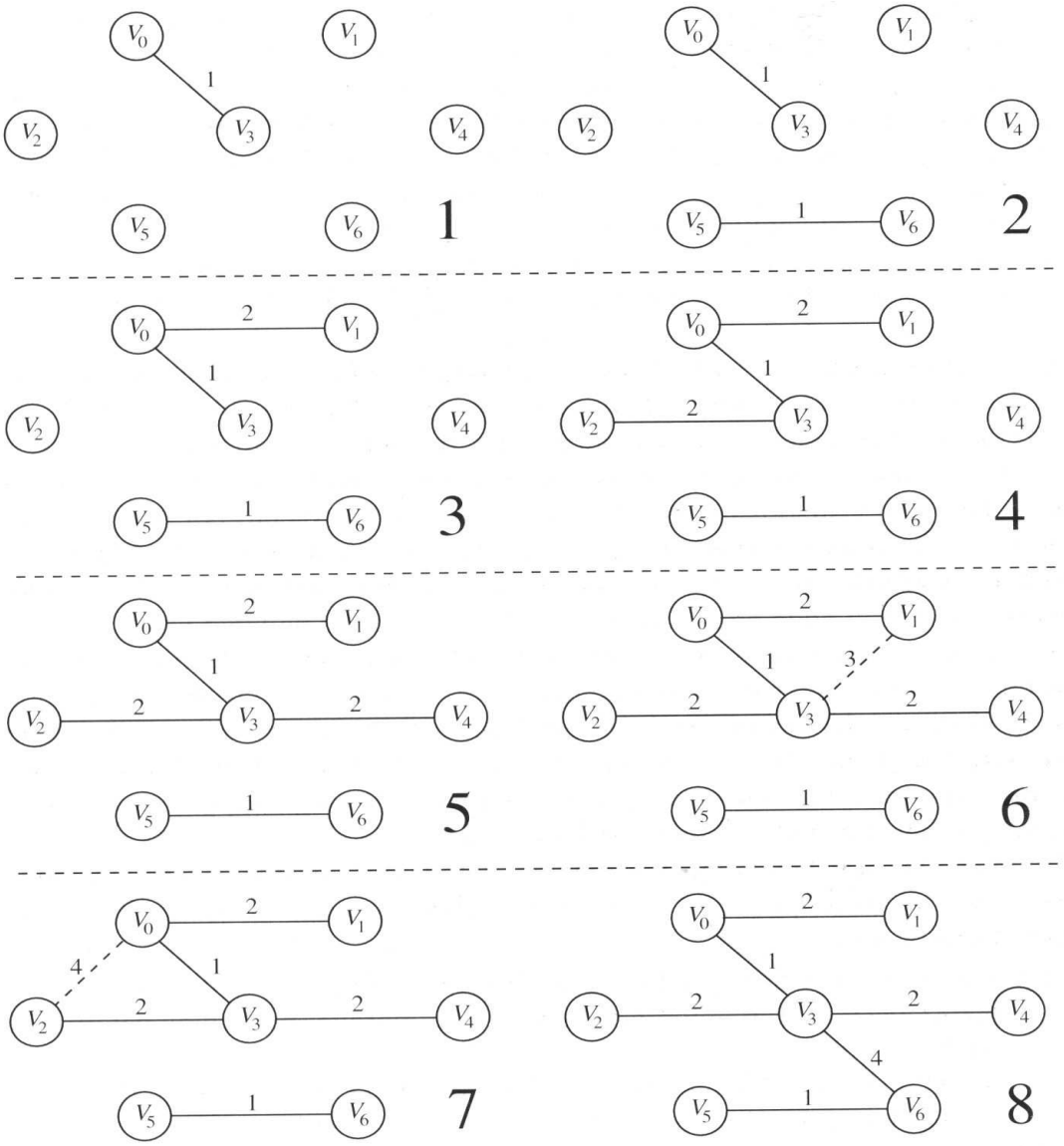

Figura 23.2 El algoritmo de Kruskal después de considerar cada arista; los pasos se presentan de arriba abajo y de izquierda a derecha, siguiendo la numeración.

recubrimiento, si w estaba conectado a u y x lo estaba a v, x y w quedan conectados y por tanto pasan a pertenecer al mismo conjunto.

23.2.2 Aplicación #2: el problema del antecesor común más próximo

Otro ejemplo de aplicación de la estructura de partición es el problema del antecesor común más próximo (ACP).

PROBLEMA DEL ANTECESOR COMÚN MÁS PRÓXIMO

Dado un árbol y una lista de pares de nodos en el árbol, buscar el antecesor común más próximo de cada par de nodos.

ACP es importante en los algoritmos sobre grafos y en biología computacional.

Como ejemplo, la Figura 23.3 muestra un árbol con una lista de pares que contiene cinco peticiones. Para el par de nodos u y z, el nodo C es el antecesor de ambos más cercano. (A y B son también antecesores, pero no son los más cercanos.) Disponemos de la secuencia completa de peticiones antes de tener que proporcionar la primera respuesta. Éste es un problema importante en la teoría de grafos que también tiene aplicaciones en el campo de la biología computacional (donde el árbol representa la evolución).

Para resolver el problema se puede utilizar un recorrido en postorden.

El algoritmo funciona realizando un recorrido en postorden del árbol. Cuando estamos a punto de acabar de procesar un nodo, examinamos la lista de pares para ver si hay algún calculo de antecesor que realizar. Si u es el nodo actual, (u, v) está en la lista de peticiones y ya hemos terminado la llamada recursiva con v, entonces tenemos información suficiente para determinar $ACP(u, v)$.

La *sujeción* de un nodo v visitado (pero no necesariamente marcado) es el nodo en el camino de acceso actual que está más cerca de v.

La Figura 23.4 ayuda a comprender cómo funciona el algoritmo. Estamos a punto de terminar la llamada recursiva correspondiente a D. Todos los nodos sombreados han sido visitados por una llamada recursiva, y excepto por los nodos en el camino a D, todas las demás llamadas recursivas ya han terminado. Marcamos un nodo después de que su llamada recursiva haya terminado. Si v está marcado, $ACP(D, v)$ es algún nodo en el camino a D. La *sujeción* de un nodo v visitado (pero no necesariamente marcado) se define como el nodo en el camino de acceso actual que está más cerca de v. En la Figura 23.4, la sujeción de p es A, la sujeción de q es B y r no está sujeto, pues aún tiene que ser visitado; podemos considerar que la sujeción de r es el mismo r la primera vez que se visita. Como muestran los dibujos, cada nodo en el camino de acceso actual es una sujeción (al menos de él mismo). Es más, los nodos visitados forman clases de equivalencia: dos nodos están relacionados si tienen la misma sujeción, y podemos considerar que cada nodo visitado está en su propia clase. Supongamos ahora que (D, v) está en la lista de pares. Tenemos tres casos:

1. v no está marcado, por lo que no tenemos información de cómo calcular $ACP(D, v)$. Sin embargo, cuando v se marque, podremos determinar $ACP(v, D)$.
2. v está marcado pero no está en el subárbol de D, por lo que $ACP(v, D)$ es la sujeción de v.

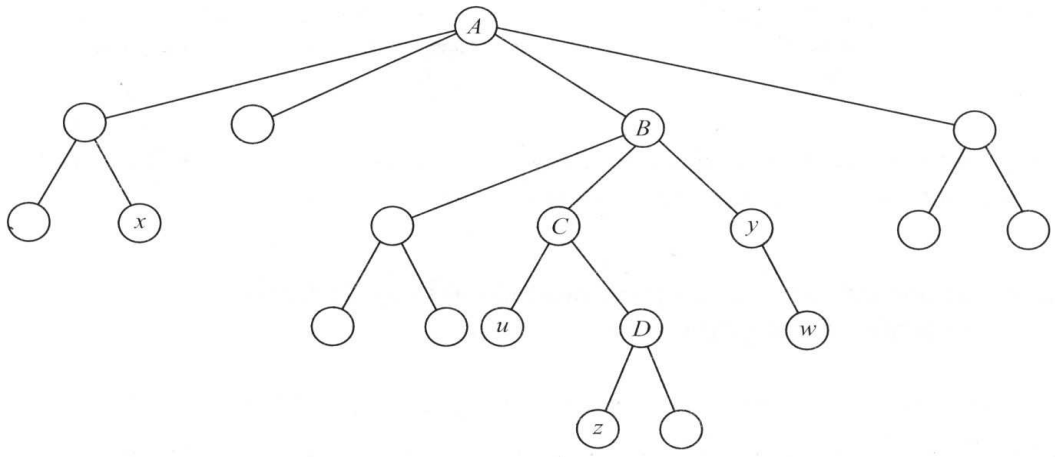

Figura 23.3 El antecesor común más próximo de cada petición en la secuencia de pares (x, y), (u, z), (w, x) (z, w) y (w, y) es A, C, A, B e y, respectivamente.

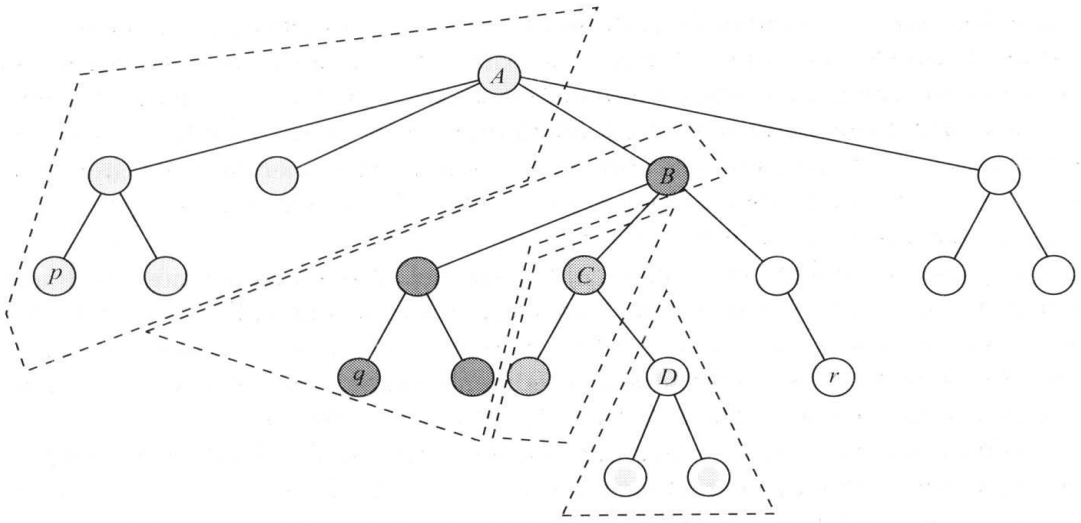

Figura 23.4 Los conjuntos justo antes de terminar la llamada recursiva correspondiente a *D*; *D* se marca como visitado y *ACP(D, v)* es la sujeción de *v* al camino actual.

3. *v* está en el subárbol de *D*, por lo que $ACP(v, D) = D$. Observe que éste no es un caso especial pues la sujeción de *v* es *D*.

Para mantener los conjuntos de nodos con sujeción común. Se utiliza el algoritmo `unir/buscar`.

Todo lo que falta es asegurarnos de que en todo instante podemos determinar la sujeción de un nodo visitado. Esto se logra fácilmente utilizando el algoritmo `unir/buscar`. Después de terminar una llamada recursiva, llamamos a `unir`. Por ejemplo, después de que termina la llamada recursiva con *D* en la Figura 23.4, se cambia a *C* la sujeción de todos los nodos en *D*. La nueva situación se muestra en la Figura 23.5. En consecuencia, necesitamos unir las dos clases de equivalencia

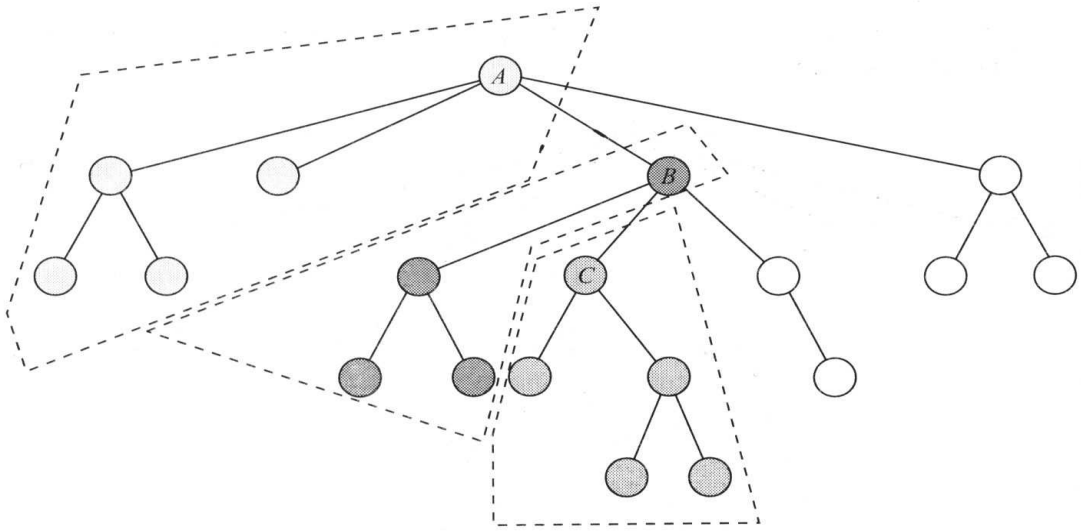

Figura 23.5 Después de que termina la llamada correspondiente a *D*, mezclamos el conjunto con sujeción en *D* con el conjunto con sujeción en *C* y calculamos *ACP(C, v)* para todos los nodos *v* que están marcados, antes de completar la llamada recursiva correspondiente a *C*.

en una. En cualquier momento, podemos obtener la sujeción de un vértice v llamando a `buscar` sobre una estructura de partición. Ya que `buscar` devuelve el número de un conjunto, podemos utilizar un vector `sujecion` para almacenar el nodo sujeción correspondiente a cada conjunto.

El pseudocódigo es compacto.

La Figura 23.6 muestra una implementación en pseudocódigo del algoritmo ACP. Como hemos mencionado anteriormente, la operación `buscar` generalmente asume que los elementos del conjunto son 0, 1, ..., $N - 1$, por lo que en un paso de preprocesamiento que calcula el tamaño del árbol almacenamos el número en preorden de cada nodo del árbol. Aunque un enfoque orientado a objetos podría intentar incorporar una traducción dentro de `buscar`, utilizando quizás un diccionario, en este caso ello sería computacionalmente ineficiente. Asumimos también que tenemos un vector de listas para almacenar las peticiones ACP. La lista i almacena las peticiones correspondientes al nodo i. Teniendo cuidado de estos detalles, el código es extraordinariamente breve.

Cuando se visita por primera vez un nodo u, se genera la sujeción a sí mismo en la línea 18 de la Figura 23.6. Entonces se procesan recursivamente sus hijos v, mediante la llamada en la línea 23. Después de que acaba cada llamada recursiva,

```
1  // Algoritmo del antecesor común más cercano
2  //
3  // Precondiciones (y objetos globales):
4  // 1. La estructura de partición está inicializada
5  // 2. Todos los nodos están inicialmente sin marcar
6  // 3. Los números en preorden ya están asignados en el campo num
7  // 4. Cada nodo puede almacenar su estado de marcaje
8  // 5. La lista de pares está disponible globalmente
9
10 Particion s = new Particion( tamanyoArbol );
11 Nodo [ ] sujecion = new Nodo[ tamanyoArbol ];
12
13 // main realiza la llamada acp( raiz )
14 // después de las inicializaciones necesarias
15
16 void acp( Nodo u )
17 {
18    sujecion[ s.buscar( u.num ) ]=u;
19
20    // Hace las llamadas en postorden
21    for( cada hijo v de u )
22    {
23       acp( v );
24       s.unir( s.buscar( u.num ), s.buscar( v.num ) );
25       sujecion( s.buscar( u.num ) ]=u;
26       }
27
28       // Hace los cálculos acp para los pares que involucren a u
29       u.marcado = true;
30       for( cada v tal que acp( u, v ) ha sido solicitado )
31           if( v.marcado )
32              System.out.println("acp( " + u + ", " + v +
33                                 " ) es " + sujecion[ s.buscar( v.num ) ] );
34 }
```

Figura 23.6 Pseudocódigo del problema ACP.

el subárbol se combina con la clase de equivalencia actual de u y nos aseguramos de actualizar la sujeción, en las líneas 24 y 25. Cuando todos los hijos han sido visitados recursivamente, podemos marcar u como procesado en la línea 29 y terminar comprobando todas las peticiones ACP que involucren a u en las líneas de la 30 a la 33[2].

23.3 El algoritmo de búsqueda rápida

Esta sección presenta los preparativos para la implementación eficiente de la estructura de partición. Hay dos estrategias básicas para resolver el problema unir/buscar. Una consigue que la instrucción `buscar` pueda ejecutarse en tiempo constante en el caso peor, y la otra logra lo mismo para la operación `unir`. Recientemente se ha demostrado que ambas cosas no pueden lograrse simultáneamente en tiempo constante (incluso amortizado) en el caso peor.

El primer enfoque se denomina *algoritmo de búsqueda rápida*. Para que la operación `buscar` sea rápida, podríamos mantener en un vector el nombre de la clase de equivalencia de cada elemento. `buscar` se reduce entonces a un simple acceso en tiempo constante. Supongamos que queremos realizar `unir`(a, b). Supongamos, también, que a está en la clase de equivalencia i y b está en j. Podemos recorrer el vector, intercambiando todas las i por j. Desgraciadamente, este recorrido cuesta un tiempo lineal. Por tanto una secuencia de $N - 1$ operaciones `unir` (el máximo número posible, ya que tras ellas todos los elementos estarían en la misma clase) tardaría un tiempo cuadrático. En el caso típico, en el cual el número de búsquedas es subcuadrático, esto es claramente inaceptable.

Una idea consiste en mantener todos los elementos que están en la misma clase de equivalencia en una lista enlazada. Esto ahorra tiempo al actualizar, pues no tenemos que buscar en el vector completo. Sin embargo, esto no reduce por sí mismo el tiempo de ejecución asintótico, ya que todavía es posible realizar $\Theta(N^2)$ actualizaciones de las clases de equivalencia durante el transcurso del algoritmo.

Si mantenemos también el tamaño de las clases de equivalencia (y cuando se realiza una unión, cambiamos el nombre de la clase pequeña por el de la mayor) entonces el tiempo total invertido en N llamadas a unir es $O(N \log N)$. Esto se debe a que cada elemento puede cambiar de clase de equivalencia como mucho $\log N$ veces, ya que cada vez que cambia de clase, su nueva clase de equivalencia es al menos el doble de grande de lo que era la antigua (por lo que se puede aplicar el principio de duplicaciones sucesivas).

El argumento por el cual una clase de equivalencia puede cambiar como mucho $\log N$ veces por elemento, se utilizará también en el algoritmo de unión rápida. La búsqueda rápida es un algoritmo sencillo, pero la unión rápida es mejor.

Esta estrategia asegura que cualquier secuencia de a lo sumo M búsquedas y $N - 1$ uniones tardará, como mucho, $O(M + N \log N)$. Si M es lineal, ésta es todavía una solución costosa. También es un poco confusa, ya que debemos mantener listas ordenadas. La próxima sección examina una solución al problema unir/buscar que hace más fácil `unir`, pero más difícil `buscar`. Este enfoque alternativo es el algoritmo de unión rápida. Aun así, el tiempo de ejecución de cualquier secuencia de como mucho M búsquedas y $N - 1$ uniones será sólo un poco más que $O(M + N)$, y sólo se usará un vector de enteros.

[2] Estrictamente hablando, u debería ser marcado en la última instrucción, pero haciéndolo antes tratamos las molestas peticiones $ACP(u, u)$.

Figura 23.7 Un bosque correspondiente a ocho elementos, inicialmente en conjuntos diferentes.

23.4 El algoritmo de unión rápida

Recordemos que el problema unir/buscar no requiere que la operación `buscar` devuelva un nombre específico, sólo que las búsquedas de dos elementos devuelvan la misma respuesta si y sólo si están en el mismo conjunto. Una idea puede ser utilizar un árbol para representar un conjunto, ya que cada elemento en el árbol tiene la misma raíz, por lo que la raíz puede usarse como nombre del conjunto.

Se representa un árbol con un vector de enteros que indican el padre de cada nodo. El nombre del conjunto de cada nodo en un árbol es la raíz del árbol.

Cada conjunto se representa por un árbol, con lo que en suma tenemos un *bosque*. El nombre de cada conjunto viene dado por el nodo en la raíz. Nuestros árboles no son necesariamente binarios, pero su representación es sencilla porque la única información necesaria es el padre de cada nodo. Por tanto, sólo necesitamos un vector de enteros: cada entrada `p[i]` en el vector representa el padre del elemento `i`, y para indicar que estamos en la raíz podemos utilizar -1 como padre. La Figura 23.7 muestra un bosque de árboles triviales y el vector que lo representa.

`unir` requiere un tiempo constante.

Para realizar la unión de dos conjuntos, mezclamos los dos árboles haciendo que la raíz de uno de ellos apunte a la raíz del otro. Debería estar claro que esta operación tiene un coste constante. Las Figuras 23.8, 23.9 y 23.10 representa el bosque después de `unir`(4,5), `unir`(6,7) y `unir`(4,6), donde hemos adoptado el convenio de que la nueva raíz después de `unir`(x,y) es x.

El `coste` de buscar depende de la profundidad del nodo accedido y en general podría ser lineal.

La búsqueda del elemento x se realiza devolviendo la raíz del árbol que contiene x. El tiempo necesario para realizar esta operación es proporcional al número de nodos en el camino de x a la raíz. La estrategia de unión descrita anteriormente nos permite crear un árbol en el que todo nodo esté en el camino a x, obteniéndose

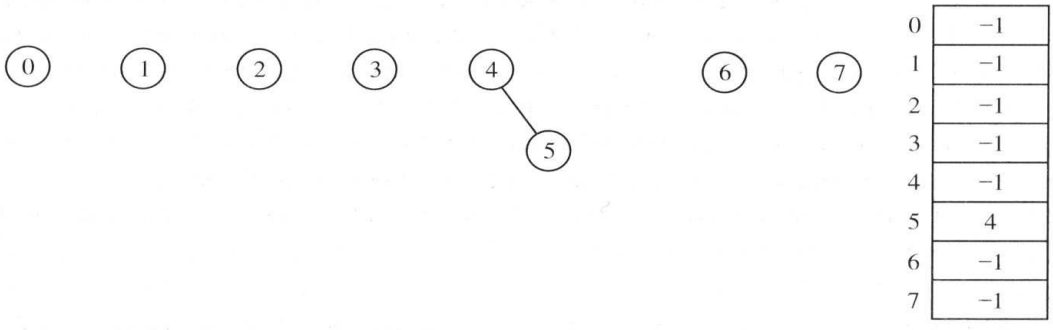

Figura 23.8 El bosque después de la unión de los árboles con raíces 4 y 5.

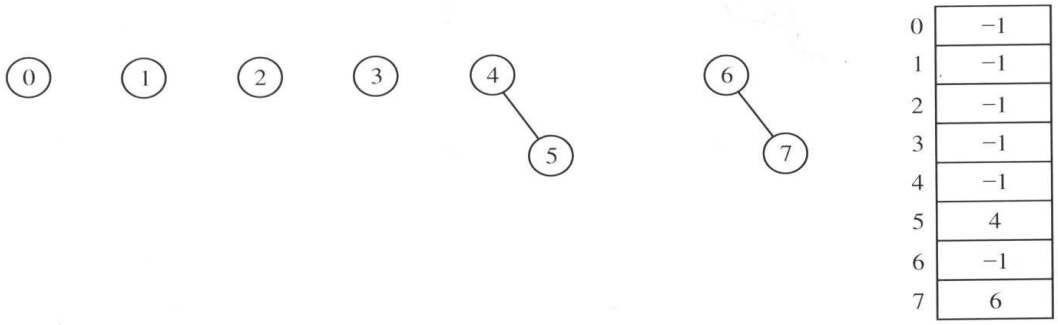

Figura 23.9 El bosque después de la unión de los árboles con raíces 6 y 7.

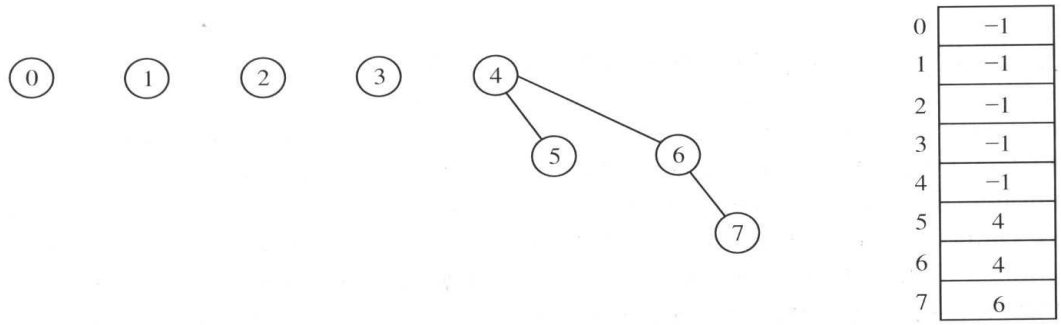

Figura 23.10 El bosque después de la unión de los árboles con raíces 4 y 6.

un tiempo de ejecución en el caso peor $\Theta(N)$ para cada operación `buscar`. Típicamente (como muestran las dos aplicaciones previas), el tiempo de ejecución se calcula para una secuencia de M instrucciones entremezcladas. En el caso peor, M operaciones consecutivas podrían tardar un tiempo $\Theta(MN)$.

En general, un tiempo de ejecución cuadrático para una secuencia de operaciones no es aceptable. Afortunadamente, hay varias formas sencillas de asegurar que este tiempo de ejecución no se dé nunca.

23.4.1 Algoritmos de unión inteligentes

Las uniones anteriores se hicieron de una forma bastante arbitraria, haciendo que el segundo árbol pasara a ser un subárbol del primero. Una mejora sencilla es exigir que siempre el árbol más pequeño pase a ser subárbol del árbol mayor, rompiendo los empates con un criterio cualquiera; este método se denomina *unión por tamaño*. Las tres uniones de la sección anterior corresponden a sendos empates, por lo que podemos considerar que se realizaron por tamaño. Si la siguiente operación fuera `unir`(3,4), se formaría el árbol de la Figura 23.11. Si no hubiéramos utilizado la heurística del tamaño, se habría generado un bosque más profundo (tres nodos, en lugar de uno sólo, verían incrementada su profundidad en un nivel).

La *unión por tamaño* garantiza búsquedas logarítmicas.

Podemos demostrar que si las uniones se hacen por tamaño, la profundidad de cualquier nodo nunca es mejor que log N. Para ver esto, observamos que inicialmente todo nodo está a profundidad 0. Cuando su profundidad aumenta como resultado de una unión, se coloca en un árbol cuyo tamaño es al menos el doble del anterior. En consecuencia su profundidad puede incrementarse como mucho

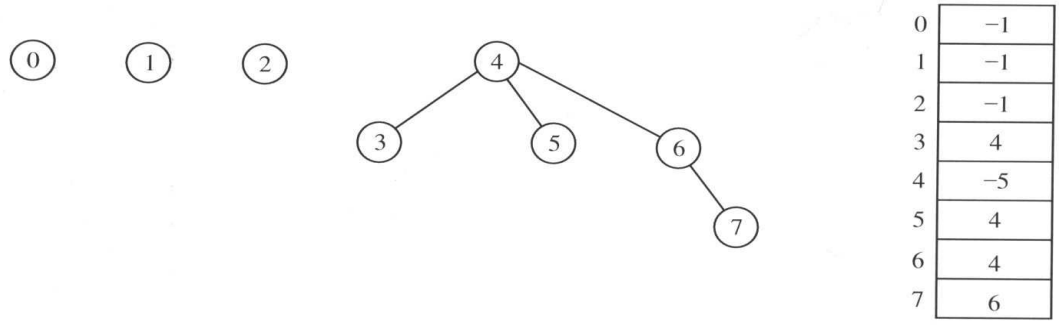

Figura 23.11 El bosque generado por uniones por tamaño, con los tamaños representados por medio de sus inversos negativos.

log N veces. (Utilizamos este mismo argumento en el algoritmo de búsqueda rápida de la Sección 23.3.) Esto implica que el tiempo de ejecución de una operación `buscar` es $O(\log N)$ y que una secuencia de M operaciones tarda como mucho $O(M \log N)$. El árbol de la Figura 23.12 muestra el peor caso posible después de 15 uniones, que se obtiene cuando todas las uniones son entre árboles de igual tamaño. Dicho caso corresponde a los denominados *árboles binomiales*. Los árboles binomiales tienen otras aplicaciones en estructuras de datos avanzadas.

En vez de guardar siempre −1 en las raíces, guardamos el inverso negativo del tamaño del correspondiente árbol.

Para implementar esta estrategia, necesitamos mantener información sobre el tamaño de cada árbol. Ya que sólo estamos utilizando un vector, podemos hacer que la entrada del vector correspondiente a la raíz contenga el inverso *negativo* del tamaño del árbol, como muestra la Figura 23.11. En consecuencia, la representación inicial será un vector con todas las entradas iguales a −1. Cuando se realiza una unión, comparamos los tamaños; el nuevo tamaño es la suma de los viejos. Por tanto, la unión por tamaño no es en absoluto complicada y no requiere ningún espacio extra. También es rápida en promedio. Esto se debe a que cuando se realizan uniones arbitrarias, habitualmente se mezclan conjuntos muy pequeños (usualmente de un solo elemento) con conjuntos grandes. El análisis matemático de esto es bastante complejo; la bibliografía al final del capítulo proporciona algunos punteros a la literatura.

La *unión por altura* también garantiza búsquedas logarítmicas.

Una implementación que también garantiza una profundidad logarítmica es la *unión por altura*. Mantenemos la altura de los árboles en vez de su tamaño y realizamos las uniones haciendo que el árbol menos profundo pase a ser subárbol del más profundo. Éste es un algoritmo sencillo, ya que la altura del árbol sólo aumenta en una unidad cuando los dos árboles unidos tienen la misma altura. En consecuencia,

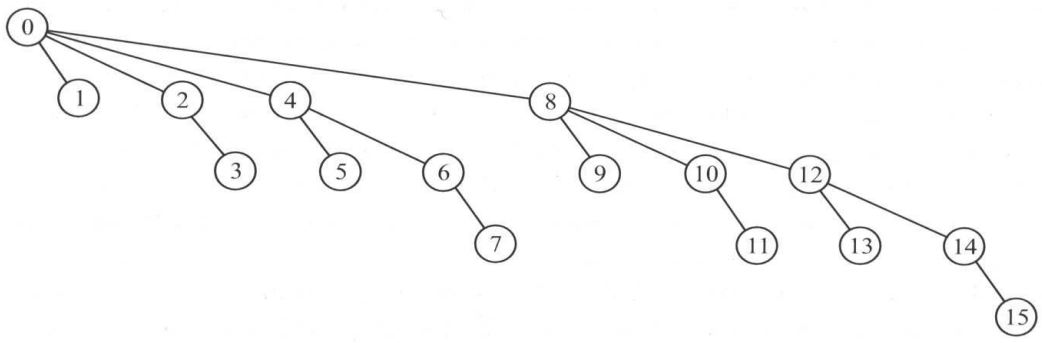

Figura 23.12 Árbol del caso peor con $N = 16$.

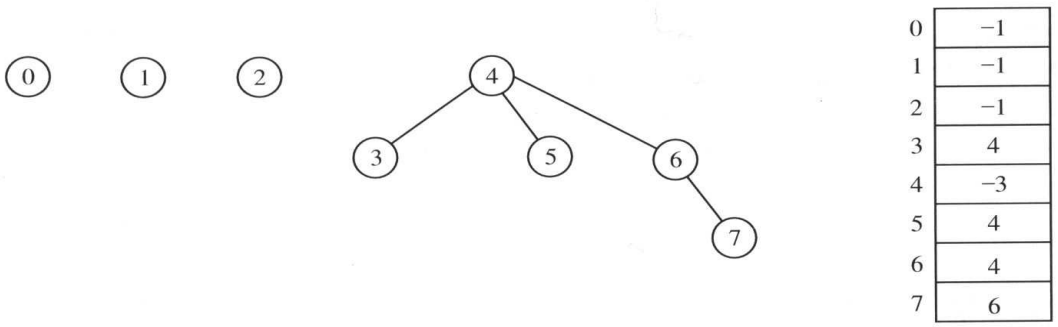

Figura 23.13 Un bosque generado por la unión por altura, con ésta codificada por medio de valores negativos.

la unión por altura es una variante trivial de la unión por tamaño. Ya que las alturas empiezan en 0, almacenamos el inverso negativo del número de nodos en el camino más profundo, en vez de la altura. Esto se muestra en la Figura 23.13.

23.4.2 Compresión de caminos

El algoritmo unir/buscar, tal y como se ha descrito hasta el momento, es bastante aceptable en la mayoría de las ocasiones. Es muy simple y lineal en promedio para una secuencia de M instrucciones. Sin embargo, el caso peor todavía es poco atractivo. Esto es así, porque no es obvio que una secuencia de operaciones `unir` de una aplicación particular (como el problema ACP) sea aleatoria (de hecho, para árboles concretos, se aleja mucho de serlo. En consecuencia, resulta deseable que intentemos cambiar las cosas de alguna forma, de manera que podamos garantizar una cota mejor para el caso peor de una secuencia de M operaciones. Parece probable que no haya más mejoras posibles del algoritmo unir, ya que el caso peor se alcanza cuando se mezclan árboles de igual tamaño. La única forma de acelerar el algoritmo, sin alterar completamente la estructura de datos, es hacer algo inteligente en la operación `buscar`.

La *compresión de caminos* hace que cada nodo accedido sea un hijo de la raíz hasta que se produzca otra unión.

Esta operación inteligente es la *compresión de caminos*. Claramente, después de realizar una búsqueda del elemento x, tendría sentido cambiar el padre de x tomando como tal la raíz del árbol. De esta forma, una segunda búsqueda de x, o de cualquier elemento en el subárbol de x, sería más rápida. Sin embargo, no hay necesidad de detenernos ahí. Podríamos también cambiar el padre de todos los nodos en el camino de acceso hasta x. El efecto de la compresión de caminos es que se cambia el padre de *todos* los nodos en el camino desde x a la raíz. La Figura 23.14 muestra el efecto de la compresión de caminos después de `buscar`(14), en el árbol del caso peor de la Figura 23.12. Con el cambio extra de dos padres que hemos realizado, los nodos 12 y 13 están ahora una posición más cerca de la raíz y los nodos 14 y 15 están dos posiciones más cerca. Por tanto los accesos futuros se beneficiarán (eso esperamos) del trabajo extra de la compresión de caminos. Observe que las uniones subsiguientes harán que los nodos levantados vuelvan a ser llevados a mayor profundidad.

La compresión de caminos garantiza un coste amortizado logarítmico por operación.

Cuando las uniones se producen de forma arbitraria, la compresión de caminos es una buena idea, porque hay gran número de nodos profundos; éstos son acercados a la raíz por la compresión de caminos. Se ha demostrado en este caso que,

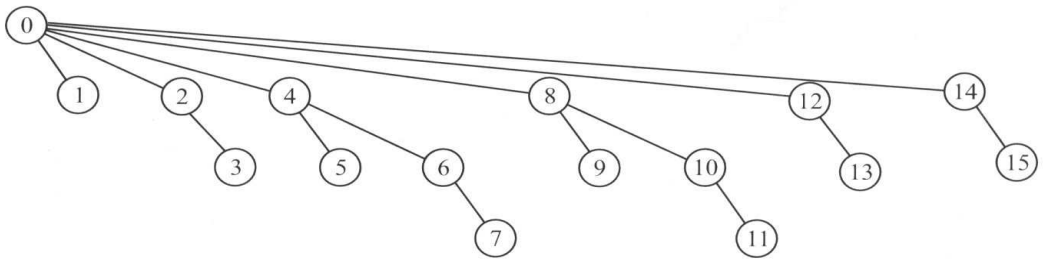

Figura 23.14 Compresión de caminos que resulta al ejecutar `buscar(14)` en el árbol de la Figura 23.12.

con la compresión de caminos, una secuencia de M operaciones requiere como mucho un tiempo $O(M \log N)$, por lo que la compresión de caminos por sí sola garantiza un coste amortizado logarítmico para la operación `buscar`.

La compresión de caminos y una unión inteligente garantizan esencialmente un coste amortizado constante por operación (es decir, una secuencia larga puede ejecutarse casi en un tiempo lineal).

La compresión de caminos es perfectamente compatible con la unión por tamaño, por lo que ambas rutinas pueden implementarse a la vez. Sin embargo, la compresión de caminos no es completamente compatible con la unión por alturas, porque la compresión de caminos puede cambiar la altura de los árboles. No está claro cómo calcularla de nuevo eficientemente, por lo que no intentamos seguir este camino. En consecuencia, los valores almacenados en cada árbol se convierten en estimaciones de la altura (las cuales en lo sucesivo denominaremos *rangos*), pero esto no es un problema. El algoritmo resultante recibe el nombre de *unión por rango*. Como veremos en la Sección 23.6, la combinación de una unión inteligente y la compresión de caminos garantiza un tiempo de ejecución casi lineal para una secuencia de M operaciones.

```
1  package EstructurasDatos;
2
3  // Clase EstrPartición
4  //
5  // CONSTRUCCIÓN: indicando el número inicial de conjuntos
6  //
7  // ******************OPERACIONES PÚBLICAS*********************
8  // void unir( raiz1, raiz2 ) --> Mezcla los dos conjuntos
9  // int buscar( x )           --> Devuelve el conjunto que contiene a x
10 // ************************ERRORES*****************************
11 // Sin comprobación de errores
12
13 /**
14  * Estructura de partición, utilizando unión por rango
15  * y compresión de caminos.
16  */
17 public class EstrParticion
18 {
19     public EstrParticion( int numElementos )
20       { /* Figura 23.16 */ }
21     public void unir( int raiz1, int raiz2 )
22       { /* Figura 23.16 */ }
23     public int buscar( int x )
24       { /* Figura 23.16 */ }
25
26     private int [ ] vector;
27 }
```

Figura 23.15 Esqueleto de la clase partición.

23.5 Implementación en Java

La estructura de partición es relativamente simple de implementar.

El esqueleto de la clase de la estructura de partición se muestra en la Figura 23.15; la implementación se muestra en la Figura 23.16. Se ha omitido la comprobación de errores, para evitar oscurecer los detalles del algoritmo. Por supuesto, un programa robusto debería comprobar estos errores. Sin la comprobación de errores, el algoritmo completo es sorprendentemente breve.

En la rutina, la unión se realiza sobre las raíces de los árboles. Algunas veces la operación se implementa pasándole dos elementos cualesquiera y teniendo que utilizar `buscar` para determinar las raíces.

```
1  /**
2   * Constructor de la estructura de partición.
3   * @param numElementos el número inicial de conjuntos.
4   */
5  public EstrParticion( int numElementos )
6  {
7      vector = new int [ numElementos ];
8      for( int i = 0; i < vector.length; i++ )
9          vector[ i ] = -1;
10 }
11
12 /**
13  * unir une dos conjuntos disjuntos utilizando la altura.
14  * Asumimos que raiz1 y raiz2 son distintos.
15  * @param raiz1 la raíz del primer conjunto.
16  * @param raiz2 la raíz del segundo conjunto.
17  */
18 public void unir( int raiz1, int raiz2 )
19 {
20     if( vector[ raiz2 ] < vector[ raiz1 ] ) // raiz2 más profunda
21         vector[ raiz1 ] = raiz2; // raiz2 nueva raíz
22     else
23     {
24         if( vector[ raiz1 ] == vector[ raiz2 ] )
25             vector[ raiz1 ]--;    // Actualizar altura
26         vector[ raiz2 ] = raiz1; // raiz1 nueva raíz
27     }
28 }
29
30 /**
31  * Búsqueda con compresión de caminos.
32  * Se omite la detección de errores.
33  * @param x el elemento buscado.
34  * @return el conjunto que contiene a x.
35  */
36 public int buscar( int x )
37 {
38     if( vector[ x ] < 0 )
39         return x;
40     else
41         return vector[ x ] = buscar( vector[ x ] );
42 }
```

Figura 23.16 Implementación de la estructura de partición sin detección de errores.

El procedimiento interesante es `buscar`. Una vez la búsqueda se ha realizado recursivamente, se asigna a `vector[x]` la raíz que después es devuelta. Ya que esto ocurre de forma recursiva, todos los nodos en el camino pasarán a apuntar (como padre) a la raíz original.

23.6 Caso peor de la unión por rango y la compresión de caminos

Cuando se utilizan ambas heurísticas, el algoritmo es casi lineal en el caso peor. Concretamente, el tiempo necesario en el caso peor para procesar una secuencia de como mucho $N - 1$ operaciones de unión y M de búsqueda es $\Theta(M\alpha(M, N))$ (supuesto $M \geq N$), donde a(M, N) es la función inversa de la *función de Ackermann*, que se define de la siguiente manera[3]:

$$\begin{aligned} A(1, j) &= 2^j && j \geq 1 \\ A(i, 1) &= A(i - 1, 2) && i \geq 2 \\ A(i, j) &= A(i - 1, A(i, j - 1)) && i, j \geq 2 \end{aligned}$$

A partir de esto definimos

$$\alpha(M, N) = \min \{i \geq 1 \mid (A(i, \lfloor M/N \rfloor) > \log N)\}.$$

La *función de Ackermann* crece muy rápidamente, y su inverso toma muy difícilmente valores mayores que 4.

Podríamos querer calcular algunos valores, pero para todos los propósitos prácticos, $\alpha(M, N) \leq 4$, que es lo único importante aquí. Por ejemplo, para cualquier $j > 1$, tenemos

$$\begin{aligned} A(2, j) &= A(1, A(2, j - 1)) \\ &= 2^{A(2, j - 1)} \\ &= 2^{2^{2^{\cdots}}} \end{aligned}$$

donde el número de veces que aparece el 2 en el exponente es j. $F(N) = A(2, N)$ se denomina habitualmente la función de Ackermann de una sola variable. El inverso de la función de Ackermann de una sola variable, representado a veces como log* N, es el número de veces que se tiene que aplicar el logaritmo a N hasta que $N \leq 1$. Por tanto, log*65.536 = 4 porque log log log log 65.536 = 1. log*$2^{65.536} = 5$. Sin embargo, tenga en cuenta que $2^{65.536}$ tiene más de 20.000 dígitos. $\alpha(M, N)$ crece aún más lentamente que log*N. Por ejemplo, $A(3, 1) = A(2, 2) = 2^{2^2} = 16$. Por tanto, para $N < 2^{16}$, $\alpha(M, N) \leq 3$. Es más, ya que $A(4, 1) = A(3, 2) = A(3, A(2, 1)) = A(2, 16)$, el cual es 2 elevado a una pila de dieciséis 2, en la práctica, $\alpha(M, N) \leq 4$. Sin embargo, $\alpha(M, N)$ no es constante cuando M es algo mayor que N, por lo que el tiempo de ejecución no es estrictamente lineal[4].

[3] La función de Ackermann se define frecuentemente como $A(i, j) = j + 1$, para $j \geq 1$. La forma utilizada en este texto crece aún más rápidamente; por tanto, el inverso crece más lentamente.

[4] Observe, sin embargo, que si $M = N \log^* N$, $\alpha(M, N)$ es como mucho 2. Por tanto, siempre que M sea sólo algo más que lineal, el tiempo de ejecución sí será lineal en M.

En el resto de esta sección, probaremos un resultado algo más débil. Mostraremos que cualquier secuencia de $M = \Omega(N)$ operaciones de unión o búsqueda requieren un tiempo total $O(M \log^* N)$. La misma cota sigue siendo cierta si utilizamos la unión por tamaño en vez de la unión por rango. Este análisis es, probablemente, el más complicado del libro y uno de los primeros análisis realmente complejos jamás realizados para un algoritmo trivial de implementar. Extendiendo esta técnica, podríamos probar también la cota más fuerte afirmada previamente.

23.6.1 Análisis del algoritmo unir/buscar

En esta sección establecemos una cota bastante ajustada del tiempo de ejecución de una secuencia de $M = \Omega(N)$ operaciones de unión y búsqueda. Las operaciones `unir` y `buscar` pueden producirse en cualquier orden, pero `unir` se hace por rango y `buscar` se hace con compresión de caminos.

Empezamos con algunos teoremas que tienen que ver con el número de nodos de rango r. Intuitivamente, debido a la regla de la unión por rango, hay muchos más nodos de rango pequeño que de rango grande. En particular, puede haber como mucho un nodo de rango $\log N$. Lo que queremos hacer es calcular una cota tan precisa como sea posible del número de nodos de cierto rango r. Ya que los rangos sólo cambian cuando se realiza la operación `unir` (y sólo cuando los dos árboles tienen el mismo rango), podemos hacer el cálculo ignorando la compresión de caminos. Esto se hace en el Teorema 23.1

Teorema 23.1

En ausencia de la compresión de caminos, cuando una secuencia de instrucciones `unir` *está siendo ejecutada, un nodo de rango r debe tener al menos* 2^r *descendientes, incluyéndole a él mismo.*

Demostración

La demostración es por inducción. La base, cuando $r = 0$, *es claramente cierta. Sea T el árbol de rango r con el menor número de descendientes y sea x la raíz de T. Supongamos que la última unión que involucró a x fue entre* T_1 *y* T_2. *Supongamos que la raíz de* T_1 *era x. Si* T_1 *tuviera rango r, entonces* T_1 *sería un árbol de rango r con menos descendientes que T. Esto contradice la hipótesis de que T tiene el menor número de descendientes posible. Por tanto, el rango de* T_1 *es como mucho* $r - 1$. *El rango de* T_2 *es como mucho el rango de* T_1, *pues la unión se hace por rango. Ya que T tiene rango r y el rango sólo podría incrementarse debido a* T_2, *se deduce que el rango de* T_2 *es* $r - 1$. *En consecuencia, el rango de* T_1 *es también* $r - 1$. *Por la hipótesis de inducción, cada uno de estos árboles tiene como mucho* 2^{r-1} *descendientes, lo que da un total de* 2^r, *lo que demuestra el teorema.*

El Teorema 23.1 afirma que si no se hace compresión de caminos, cualquier nodo de rango r tiene al menos 2^r descendientes. La compresión de caminos puede cambiar esto, ya que elimina descendientes de un nodo. Sin embargo, cuando se realizan uniones, aun con compresión de caminos, utilizamos los rangos, que son estimaciones de la altura. Estos rangos se comportan como si no hubiera compresión de caminos. Por tanto, cuando se está acotando el número de nodos de rango r, la compresión de caminos puede ser ignorada. Esto se hace en el Teorema 23.2.

Teorema 23.2 *El número de nodos de rango r es como mucho $N/2^r$.*

Demostración *Sin la compresión de caminos, cada nodo de rango r es la raíz de un subárbol de al menos 2^r nodos. Ningún otro nodo en el árbol puede tener rango r. Por tanto todos los subárboles de rango r son disjuntos. En consecuencia, hay como mucho $N/2^r$ subárboles disjuntos y por tanto $N/2^r$ nodos de rango r.*

El Teorema 23.3 puede parecer bastante obvio, pero es crucial en el análisis.

Teorema 23.3 *En cualquier punto del algoritmo unir/buscar, los rangos de los nodos de un camino desde una hoja a la raíz crecen de forma monótona.*

Demostración *El teorema es obvio si no hay compresión de caminos. Si después de la compresión de caminos, un nodo v es descendiente de otro w, claramente v tiene que ser descendiente de w cuando sólo se realizan las operaciones de unión. Por tanto, el rango de v es estrictamente menor que el rango de w.*

No hay muchos nodos de rango elevado, y los rangos crecen en cada camino hacia la raíz.

Presentemos ahora un resumen de nuestros resultados preliminares. El Teorema 23.2 nos indica a cuántos nodos se les puede asignar rango r. Ya que los rangos sólo se asignan mediante uniones, las cuales no tienen idea de la compresión de caminos, el Teorema 23.2 es válido en cualquier paso del algoritmo unir/buscar, aun en medio de compresiones. El Teorema 23.2 es ajustado, en el sentido de que para cada rango r es posible llegar a tener $N/2^r$ nodos. Sin embargo, es ligeramente holgado, ya que lo que no es posible es alcanzar simultáneamente todas las cotas de los distintos rangos. Mientras que el Teorema 23.2 describe el número de nodos de rango r, el Teorema 23.3 nos indica su distribución. Como era de esperar, el rango de los nodos crece estrictamente a lo largo de cada camino desde una hoja a la raíz.

Utilizamos monedas para contabilizar los costes como alternativa a la función de potencial. El número total de monedas representará el tiempo total.

Ahora estamos preparados para demostrar el teorema principal. El plan básico es el siguiente: una búsqueda de cualquier nodo v requiere un tiempo proporcional al número de nodos en el camino de v a la raíz. Durante cada búsqueda cargaremos una unidad de coste por cada nodo en el camino de v a la raíz. Para ayudar a contabilizar los cargos, colocaremos una moneda imaginaria en cada nodo del camino. Éste es un truco de contabilidad que no es parte del programa. Representa una alternativa, aunque similar en el fondo, al uso de la función de potencial en el análisis amortizado de los árboles de ensanchamiento y los montículos sesgados. Cuando termina el algoritmo *recogeremos* todas las monedas depositadas para determinar el tiempo total invertido en su ejecución.

Utilizamos euros y dólares. Los dólares cuentan las primeras veces que comprimimos; los euros tienen en cuenta las siguientes compresiones o cuando no hay compresión.

Como truco de contabilidad adicional, depositaremos dos tipos de monedas diferentes, digamos euros y dólares. Mostraremos que durante la ejecución del algoritmo, sólo podremos depositar cierto número de euros durante cada búsqueda (independientemente de cuántos nodos haya). Mostraremos también que sólo podemos depositar cierta cantidad de dólares en cada nodo (independientemente del número de búsquedas). Sumando estos dos totales obtendremos una cota del número total de monedas que pueden ser depositadas.

Ahora esbozamos con más detalle nuestro método de contabilidad. Dividiremos los nodos por su rango. Luego dividiremos los rangos en grupos de rangos. En cada búsqueda, depositaremos euros en un fondo común general y algunos dólares en nodos específicos. Para calcular el número total de dólares depositados, comenzamos por calcular lo depositado en cada nodo. Sumando lo depositado en cada nodo de rango r, obtendremos el depósito total por rango. Luego, sumaremos todos los depósitos de cada rango r en un mismo grupo g, obteniendo el total depositado en cada grupo de rangos g. Finalmente, sumaremos todos los depósitos en cada grupo de rangos g para obtener el número total de dólares depositados. Sumando esto al número de euros en el fondo común obtenemos la respuesta final.

Los rangos se reparten en grupos. Los grupos adecuados quedan determinados al final de la demostración. El grupo 0 contiene sólo el rango 0.

Como hemos mencionado anteriormente, repartiremos los rangos en grupos. Cada rango r irá al grupo $G(r)$, siendo G determinada más adelante (para equilibrar los cargos en euros y doláres). El rango mayor en cada grupo de rangos se denotará $F(g)$, de modo que $F = G^{-1}$ es una *inversa* de G. El número de rangos en un grupo de rangos, $g > 0$, es por tanto $F(g) - F(g-1)$. Claramente, $G(N)$ es una cota superior muy relajada del mayor grupo de rangos. Como ejemplo, supongamos que dividimos los rangos como en la Figura 23.17. En este caso, $G(r) = \lceil \sqrt{r} \rceil$. El mayor rango en el grupo g es $F(g) = g^2$. Observemos también que el grupo g contiene los rangos del $F(g-1) + 1$ al $F(g)$, inclusive. Esta fórmula no se puede aplicar al grupo de rangos 0, por lo que por conveniencia haremos que el grupo de rangos 0 sólo contenga a los elementos de rango 0. Observe que los grupos se forman con rangos consecutivos.

Como mencionamos anteriormente en el capítulo, cada instrucción `unir` requiere un tiempo constante, con tal de que cada raíz mantenga su rango. Por tanto, las uniones son esencialmente gratis, en lo que se refiere a esta demostración.

Cuando se comprime un nodo, su nuevo padre tendrá un rango mayor que su viejo padre.

Cada búsqueda requiere un tiempo proporcional al número de nodos en el camino desde el nodo que representa al elemento i accedido hasta la raíz. Por tanto, depositaremos un euro en cada vértice del camino. Sin embargo, si esto fuera todo lo que hiciésemos, no podríamos esperar mucho de la cota, ya que no estamos sacando ventaja de la compresión de caminos. En consecuencia, debemos utilizar en nuestro análisis alguna propiedad de la compresión de caminos. La observación clave es que, como resultado de la compresión de caminos, un nodo obtiene un nuevo padre que tiene un mayor rango que el viejo.

Reglas para los depósitos de euros y dólares.

Para incorporar este hecho en la demostración, utilizaremos el siguiente método imaginativo de contabilidad: para cada nodo v en el camino desde el nodo i accedido a la raíz, depositamos una moneda de uno de los dos tipos posibles:

Grupo	Rango
0	0
1	1
2	2,3,4
3	de 5 a 9
4	de 10 a 16
i	de $(i-1)^2 + 1$ a i^2

Figura 23.17 Posible división de los rangos en grupos.

1. Si v es la raíz, o si el padre de v es la raíz o si el padre de v está en un grupo de rangos diferente al de v, cargamos una unidad en concepto de aplicación de esta regla, depositando un euro en el fondo común.
2. En otro caso, depositamos un dólar en el nodo.

El Teorema 23.4 afirma que la contabilidad es acertada.

Teorema 23.4 *Para cualquier operación de búsqueda, el número total de monedas depositadas, en el fondo común o en los nodos, es exactamente igual al número de nodos accedido durante la búsqueda.*

Demostración *Obvia.*

Los cargos en euros están limitados por el número de grupos diferentes. Los cargos en dólares están limitados por el tamaño de los grupos. Ocasionalmente necesitamos equilibrar estos costes.

Por tanto, sólo necesitamos sumar todos los euros depositados bajo la regla 1 con todos los dólares depositados bajo la regla 2. Antes de seguir con la demostración, esbocemos las ideas intuitivas. Los dólares son depositados en un nodo cuando es comprimido y su padre está en el mismo grupo de rangos que el nodo. Ya que después de cada compresión de caminos el nodo obtiene un padre de grado mayor y dado que el tamaño de un grupo de rangos es finito, ocasionalmente el nodo obtendrá un padre que no está en el mismo grupo. En consecuencia, el número de dólares que pueden colocarse en un nodo está limitado. Este número es aproximadamente igual al tamaño del grupo de rangos del nodo. Por otro lado, los cargos en euros también están limitados, esencialmente por el número de grupos de rangos. En consecuencia, queremos elegir los grupos de rangos de forma que sean pequeños (para limitar los cargos en dólares), pero que no haya muchos (para limitar los cargos en euros). Ahora estamos preparados para razonar con detalle por medio de una serie de teoremas, del 23.5 al 23.10.

Teorema 23.5 *Durante todo el algoritmo, los depósitos totales de euros bajo la regla 1 totalizan $M(G(N)+2)$.*

Demostración *En cada búsqueda, se depositan como mucho dos euros, debido a la raíz y a su hijo. Por el Teorema 23.3, los vértices en el camino hacia la raíz crecen monótonamente en rango, y por tanto, el grupo de rangos nunca decrece al ascender por el camino. Ya que como mucho hay $G(N)$ grupos de rangos (además del grupo 0), sólo otros $G(N)$ vértices pueden ser utilizados para generar un depósito por aplicación de la regla 1 dentro de una misma búsqueda. Por tanto, en cada búsqueda pueden colocarse a lo sumo $G(N)+2$ euros en el fondo común. En consecuencia, a lo largo de una secuencia de M búsquedas como mucho $M(G(N)+2)$ euros pueden ser depositados por aplicación de la regla 1.*

Teorema 23.6 *En cada nodo individual en el grupo de rangos g, el número total de euros depositados es $F(g)$.*

Si en aplicación de la regla 2 se deposita un dólar en un vértice v, v será movido por la compresión de caminos y pasará a tener un padre de rango mayor que su padre antiguo. Ya que el rango mayor de su grupo es $F(g)$, tenemos la garantía de que después de depositar $F(g)$ monedas, el padre de v no estará más en el grupo de rangos de v.

Demostración

La cota del Teorema 23.6 puede mejorarse utilizando el tamaño del grupo de rangos en vez de su mayor miembro. Sin embargo, esto no mejoraría la cota obtenida para el algoritmo unir/buscar.

El número de nodos, $N(g)$ en el grupo de rangos $g > 0$ es como mucho $N/2^{F(g-1)}$.

Teorema 23.7

Por el Teorema 23.2, hay como mucho $N/2^r$ nodos de rango r. Al sumar sobre el grupo de rangos g, obtenemos

Demostración

$$
\begin{aligned}
N(g) &\leqslant \sum_{r=F(g-1)+1}^{F(g)} \frac{N}{2^r} \\
&\leqslant \sum_{r=F(g-1)+1}^{\infty} \frac{N}{2^r} \\
&\leqslant N \sum_{r=F(g-1)+1}^{\infty} \frac{1}{2^r} \\
&\leqslant \frac{N}{2^{F(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} \\
&\leqslant \frac{2N}{2^{F(g-1)+1}} \\
&\leqslant \frac{N}{2^{F(g-1)}}.
\end{aligned}
$$

El máximo número de dólares depositados en todos los vértices en el grupo de rangos g es como mucho $NF(g)/2^{F(g-1)}$.

Teorema 23.8

El resultado se obtiene mediante una simple multiplicación de las cantidades obtenidas en los Teoremas 23.6 y 23.7.

Demostración

La cantidad total depositada por aplicación de la regla 2 es, a lo sumo, de $N\sum_{g=1}^{G(N)} F(g)/2^{F(g-1)}$ dólares.

Teorema 23.9

Ya que el grupo 0 contiene sólo elementos de rango 0, no puede contribuir a los cargos por aplicación de la regla 2 (no puede tener un padre en el mismo grupo de rangos). Entonces, la cota se obtiene sumando las contribuciones de los otros grupos de rangos.

Demostración

Grupo	Rango
0	0
1	1
2	2
3	3,4
4	de 5 a 16
5	de 17 a 65.536
6	de 65.537 a $2^{65.536}$
7	Rangos realmente grandes

Figura 23.18 División concreta de los rangos en grupos utilizada en la demostración.

Ahora podemos especificar los grupos de rangos para minimizar la cota. Nuestra elección no es siempre la óptima, pero sí está cerca.

Tenemos, pues, los depósitos bajo las reglas 1 y 2. El total es

$$M(G(N) + 2) + N\Sigma_{g=1}^{G(N)} F(g)/2^{F(g-1)}. \qquad \textbf{(23.1)}$$

Todavía no hemos especificado $G(N)$ o su inversa $F(N)$. Obviamente, somos libres de elegir virtualmente cualquier cosa, pero parece razonable elegir $G(N)$ de forma que su participación en la Ecuación 23.1 nos conduzca a minimizar ésta. Sin embargo, si $G(N)$ es demasiado pequeña, $F(N)$ será grande, perjudicando la cota obtenida. Aparentemente una buena elección es que $F(i)$ sea la función definida recursivamente como $F(0) = 0$ y $F(i) = 2^{F(i-1)}$. Esto nos conduce a $G(N) = 1 + \lfloor \log^* N \rfloor$. La Figura 23.18 muestra cómo se clasifican los rangos. Observe que el grupo 0 contiene sólo el rango 0, lo que requeríamos en la demostración del Teorema 23.9. F es muy parecida a la función de Ackermann de una variable, diferenciándose sólo en el caso básico. Con esta elección de F y G, podemos completar el análisis.

Teorema 23.10 *El tiempo de ejecución del algoritmo unir/buscar con $M = \Omega(N)$ operaciones de* `buscar` es $O(M \log^* N)$.

Demostración *Sustituimos las definiciones de F y G en la Ecuación 23.1. El número total de euros es $O(MG(N)) = O(M \log^* N)$. Ya que $F(g) = 2^{F(g-1)}$, el número total de dólares es $NG(N) = O(N \log^* N)$. Del hecho de que $M = \Omega(N)$, se deduce la cota.*

Observe que tenemos más euros que dólares. La función $\alpha(M, N)$ equilibra las cosas; ésta es la razón por la que se genera una buena cota.

Resumen

En este capítulo hemos estudiado una estructura de datos muy sencilla para mantener una estructura de partición. Cuando se ejecuta la operación `unir`, no importa, en lo que se refiere a la corrección, qué conjunto mantiene su nombre. Una valiosa

lección que deberíamos aprender aquí es que es muy importante considerar alternativas cuando un paso particular no está totalmente especificado. El paso de unión es flexible. Sacando ventaja de este hecho, podemos conseguir un algoritmo más eficiente.

La comprensión de caminos fue históricamente una de las primeras formas de auto-ajustamiento, que nosotros hemos visto ya en otros lugares (árboles de ensanchamiento, montículos sesgados). Su uso aquí es extremadamente interesante desde un punto de vista teórico, ya que fue uno de los primeros ejemplos de un algoritmo sencillo con un análisis en el caso peor no tan simple.

Elementos del juego

algoritmo de búsqueda rápida Implementación de unir/buscar donde `buscar` es una operación constante en tiempo.

algoritmo de Kruskal Algoritmo que selecciona aristas en orden creciente de costes y añade una arista al árbol si con ello no se crea un ciclo.

algoritmo de unión rápida Implementación de unir/buscar donde `unir` es una operación constante en tiempo.

algoritmo unir/buscar Algoritmo que se ejecuta procesando operaciones `unir` y `buscar` utilizando una estructura de partición.

árbol de recubrimiento Árbol formado por aristas de un grafo no dirigido que conectan todos los vértices.

árbol de recubrimiento mínimo Subgrafo conexo de G que cubre todos los vértices con coste total mínimo. Es un problema fundamental en la teoría de grafos.

bosque Colección de árboles.

clase de equivalencia La clase de equivalencia de un elemento x en un conjunto S es el subconjunto de S que contiene todos los elementos relacionados con x.

compresión de caminos Hace que cada nodo accedido pase a ser hijo de la raíz hasta que ocurra otra unión.

estructura de partición Estructura de datos utilizada para manipular conjuntos disjuntos.

función de Ackermann Función que crece muy rápidamente. Su inversa está acotada en la práctica por 4.

operaciones de la estructura de partición Operaciones básicas para manipular la estructura de partición. Son `unir` y `buscar`.

problema del antecesor común más próximo (ACP) Dados un árbol y una lista de pares de nodos del árbol, encontrar el antecesor común más próximo de cada par de nodos. ACP es importante en ciertos algoritmos de grafos y en biología computacional.

rango En los algoritmos de la estructura de partición, la estimación de la altura de un nodo.

relación Definida sobre un conjunto nos dice si cada par de elementos están o no relacionados.

relación de equivalencia Relación reflexiva, simétrica y transitiva.

unión por altura Hace que el árbol de menor altura sea hijo del árbol con mayor altura durante la unión.

unión por rango Unión según la altura cuando se hace compresión de caminos.
unión por tamaño Hace que el árbol más pequeño sea hijo del árbol mayor durante la unión.

Errores comunes

1. `unir` asume que sus parámetros son raíces de árboles. Se producirían estragos si en efecto no lo fueran. Una implementación más cuidadosa realizaría dicha comprobación.

En Internet

La clase partición está disponible en Internet en el directorio **DataStructures**. El nombre del fichero es el siguiente:

DisjSet.java Traducido como `EstrParticion.java`, contiene la implementación de la clase de la estructura de partición.

Ejercicios

Cuestiones breves

23.1. Muestre el resultado de la siguiente secuencia de instrucciones: `unir`(1,2), `unir`(3,4), `unir`(3,5), `unir`(1,7), `unir`(3,6), `unir`(8,9), `unir`(1,8), `unir`(3,10), `unir`(3,11), `unir`(3,12), `unir`(3,13), `unir`(14,15), `unir`(16,17), `unir`(14,16), `unir`(1,3), `unir`(1,14), cuando se utilizan las siguientes operaciones de unión:

a) Arbitraria.
b) Por altura.
c) Por tamaño.

23.2. Para cada uno de los árboles del ejercicio anterior, realice una búsqueda con compresión de caminos del nodo más profundo.
23.3. Calcule el árbol de recubrimiento mínimo del grafo de la Figura 23.19.
23.4. Muestre el funcionamiento del algoritmo ACP con los datos de la Figura 23.3.

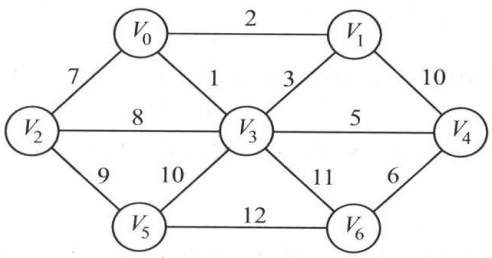

Figura 23.19 Un grafo *G* para el Ejercicio 23.3.

Problemas teóricos

23.5. Demuestre que el algoritmo de Kruskal es correcto. ¿Ha asumido que los costes de las aristas son no negativos?

23.6. Muestre que si la unión se realiza por altura, la profundidad de cualquier árbol es logarítmica.

23.7. Muestre que si todas las uniones preceden a las búsquedas, el algoritmo de la estructura de partición con compresión de caminos es lineal, aun si las uniones se realizan arbitrariamente. Observe que el algoritmo no cambia; sólo cambia su rendimiento.

23.8. Suponga que quiere añadir una nueva operación, `eliminar(x)`, que elimina `x` de su conjunto actual y lo coloca aislado. Muestre cómo modificar el algoritmo unir/buscar, de forma que el tiempo de ejecución de una secuencia de M operaciones siga siendo $O(M\alpha(M,N))$.

23.9. Demuestre que si las uniones se hacen por tamaño y se realiza la compresión de caminos, el tiempo de ejecución en el caso peor es aún $O(M \log^* N)$.

23.10. Suponga que implementa una compresión de caminos parcial en `buscar(i)` cambiando cada padre de los nodos en el camino desde `i` a la raíz por su abuelo (donde ello tenga sentido). Esto se denomina *división del camino por la mitad*. Demuestre que si se hace esto y se utiliza cualquier unión heurística, el tiempo de ejecución en el caso peor sigue siendo $O(M \log^* N)$.

Problemas prácticos

23.11. Implemente de forma no recursiva la operación `buscar`. ¿Hay una diferencia notable en el tiempo de ejecución?

23.12. Suponga que quiere añadir una operación extra, `desunir`, que deshace la última unión aún no deshecha. Una forma de hacer esto es utilizar la unión por rango, pero con un `buscar` sin compresión de caminos, y utilizar una pila para almacenar el estado antiguo anterior a la unión. Una `desunión` puede implementarse desapilando de la pila el estado antiguo.

a) ¿Por qué no podemos utilizar la compresión de caminos?
b) Implemente el algoritmo unir/buscar/desunir.

Prácticas de programación

23.13. Añada comprobación de errores a la implementación de la estructura de partición de la Figura 23.16.

23.14. Escriba un programa que determine los efectos de la compresión de caminos y de diversas estrategias de unión. Su programa debería procesar una larga secuencia de operaciones de equivalencia utilizando todas las estrategias estudiadas (incluyendo la división del camino por la mitad del Ejercicio 23.10).

23.15. Implemente el algoritmo de Kruskal.

23.16. Una alternativa al algoritmo del árbol de recubrimiento mínimo se debe a Prim [12]. Trabaja expandiendo un solo árbol en sucesivos pasos. Empie-

za eligiendo cualquier nodo como la raíz. Al comienzo de un paso, algunos nodos son parte del árbol y el resto no lo son. En cada paso, añadimos la arista de coste mínimo que conecta un nodo del árbol con uno de fuera. Una implementación del algoritmo de Prim es esencialmente idéntica al algoritmo de caminos mínimos de Dijkstra visto en la Sección 14.3, con la siguiente regla de actualización: $d_w = \min(d_w, c_{v,w})$ (en vez de $d_w = \min(d_w, d_v + c_{v,w})$). Además, ya que el grafo no es dirigido, cada arista aparece en dos listas de adyacencia. Implemente el algoritmo de Prim y compare su rendimiento con el de Kruskal.

23.17. Escriba un programa para resolver el problema ACP para árboles binarios. Compruebe su eficiencia construyendo un árbol binario aleatorio con 10.000 elementos y realizando 10.000 preguntas sobre el antecesor.

Bibliografía

La representación de los conjuntos de una partición mediante árboles fue propuesta en [8]. [1] atribuye la compresión de caminos a McIlroy y Morris y contiene varias aplicaciones de la estructura de partición. El algoritmo de Kruskal aparece en [11], mientras que la alternativa estudiada en el Ejercicio 23.16 es de [12]. El algoritmo ACP se estudia en [2]. Se pueden encontrar otras aplicaciones en [15].

La cota $O(M \log^* N)$ para el problema unir/buscar es de [9]. Tarjan [13] obtuvo la cota $O(M\alpha(M, N))$ y demostró que la misma es ajustada. Es más, la cota es intrínseca al problema y no puede ser mejorada por un algoritmo alternativo [16]. Otras estrategias para la compresión de caminos y la unión consiguen las mismas cotas; véase [16] para más detalles. Si la secuencia de uniones se conoce de antemano, es posible resolver el problema en tiempo lineal [7]. Este resultado puede utilizarse para probar que el problema ACP puede resolverse en tiempo lineal.

Diversos resultados sobre la complejidad en promedio del problema unir/buscar aparecen en [6], [10], [17] y [4]. Otros resultados que acotan el tiempo de ejecución de cada operación por separado (en oposición a la consideración de secuencias enteras) aparecen en [5].

1. A. V. Aho, J. E. Hopcroft, y J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass. (1974).

2. A. V. Aho, J. E. Hopcroft, y J. D. Ullman, «On Finding Lowest Common Ancestors in Trees», *SIAM Journal on Computing* **5** (1976), 115-132.

3. L. Banachowski, «A Complement to Tarjan's Result about the Lower Bound on the Set Union Problem», *Information Processing Letters* **11** (1980), 59-65.

4. B. Bollobas y I. Simon, «Probabilistic Analysis of Disjoint Set Union Algorithms», *SIAM Journal on Computing* **22** (1993), 1053-1086.

5. N. Blum, «On the Single-operation Worst-case Time Complexity of the Disjoint Set Union Problem», *SIAM Journal on Computing* **15** (1986), 1021-1024.

6. J. Doyle y R. L. Rivest, «Linear Expected Time of a Simple Union Find Algorithm», *Information Processing Letters* **5** (1976), 146-148.

7. H. N. Gabow y R. E. Tarjan, «A Linear-time Algorithm for a Special Case of Disjoint Set Union», *Journal of Computer and System Sciences* **30** (1985), 209-221.

8. B. A. Galler y M. J. Fischer, «An Improved Equivalence Algorithm», *Communications of the ACM* **7** (1964), 301-303.

9. J. E. Hopcroft y J. D. Ullman, «Set Merging Algorithms», *SIAM Journal on Computing* **2** (1973), 294-303.

10. D. E. Knuth y A. Schonage, «The Expected Linearity of a Simple Equivalence Algorithm», *Theoretical Computer Science* **6** (1978), 281-315.

11. J. B. Kruskal, Jr., «On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem», *Proceedings of the American Mathematical Society* **7** (1956), 48-50.

12. R. C. Prim, «Shortest Connection Networks and Some Generalizations», *Bell System Technical Journal* **36** (1957), 1389-1401.

13. R. E. Tarjan, «Efficiency of a Good but Not Linear Set Union Algorithm», *Journal of the ACM* **22** (1975), 215-225.

14. R. E. Tarjan, «A Class of Algorithms Which Require Nonlinear Time to Maintain Disjoint Sets», *Journal of Computer and System Sciences* **18** (1979), 110-127.

15. R. E. Tarjan, «Applications of Path Compression on Balanced Trees», *Journal of the ACM* **26** (1979), 690-715.

16. R. E. Tarjan y J. van Leeuwen, «Worst Case Analysis of Set Union Algorithms», *Journal of the ACM* **31** (1984), 245-281.

17. A. C. Yao, «On the Average Behavior of Set Merging Algorithms», *Proceedings of the Eighth Annual ACM Symposium on the Theory of Computation* (1976), 192-195.