UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

ANALISIS AMORTIZADO DE ALGORITMOS PARA EL PROBLEMA DE MANTENER CONJUNTOS DISJUNTOS

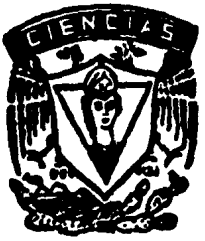
T E S I S

QUE PARA OBTENER EL TITULO DE

M A T E M A T I C O

P R E S E N T A:

CRIEL MERINO LOPEZ



MEXICO, D. F. 1994

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

CIUDAD UNIVERSITARIA

VNIVERSIDAD NACIONAL
AVFNOMA DE
MEXICO

FACULTAD DE CIENCIAS
División de Estudios
Profesionales
Exp. Núm. 55

M. EN C. VIRGINIA ABRIN BATULE
Jefe de la División de Estudios Profesionales
Universidad Nacional Autónoma de México.
P r e s e n t e .

Por medio de la presente, nos permitimos informar a Usted, que habiendo revisado el trabajo de tesis que realizó el pasante
Merino López Criel
con número de cuenta *8738426-8* con el título:
Análisis Amortizado de Algoritmos para el Problema de Mantener Conjuntos Disjuntos

Consideramos que reune ya los méritos necesarios para que pueda continuar el trámite de su Examen Profesional para obtener el título de *Matemático*.

GRADO NOMBRE Y APELLIDOS COMPLETOS	FIRMA
Dr. Sergio Rajsbaum Gorodezky Director de Tesis	
Dra. Hortensia Galeana Sánchez	
Dr. David Rosenblueth Laguette	
Dr. Javier Bracho Carpizo Suplente	
Dr. Ibadón Sánchez Arroyo Suplente	

Ciudad Universitaria, D.F., a 23 de *Mayo* de 199 4

Contenido

Lista de Figuras

Introducción

El problema de mantener conjuntos disjuntos (*union-find problem*), llamado también problema de equivalencia (*equivalence problem*), consiste en dar una estructura de datos para representar la partición en un universo con n elementos; ésta resulta de una serie de afirmaciones de la forma "$x \equiv y$". Para esto, la estructura debe poder realizar, en cualquier instante, una de las siguientes operaciones: CREA-ELEMENTO(x, l), para crear en el universo un conjunto con nombre l y único elemento x; ENCUENTRA-ELEMENTO(x) para determinar la clase de un elemento x; y UNE(x, y) para unir las clases de los elementos x y y.

Varios algoritmos para estas operaciones y que usan la estructura de datos de [12] han sido desarrollados en [4, 8, 12, 19, 23, 24, 36] y una colección de 26 algoritmos son descritos y analizados en [45]. Muchos de los resultados importantes son en parte debidos a Robert Ende Tarjan [38, 42, 43, 45, 50]. Las soluciones del problema tienen aplicación en teoría de gráficas [28, 40, 22, 37, 18, 30], sistemas operativos [17, 35, 16, 31, 13] y otras ramas de la computación [2, 32].

La intención de esta tesis es exponer dos de los resultados más importantes sobre el problema. Primero, el análisis de una solución, la dada en [45], que obtiene una complejidad amortizada de $O(m\alpha(m+n, n)+n)$, donde α es la función inversa de Ackermann. O sea, lo anterior es el tiempo máximo de cualquier secuencia con m operaciones de ENCUENTRA-ELEMENTO y n elementos iniciales en el universo. Y segundo, la demostración de que la anterior complejidad es la mejor que se puede esperar para cualquier solución, bajo ciertas restricciones técnicas [42].

Los dos primeros capítulos sirven para introducir la notación y las herramientas que usamos en el resto del trabajo y en su mayor parte provienen de [10, 43, 2]. El conocimiento de la notación-O es fundamental para entender éste y cualquier trabajo sobre complejidad de algoritmos. En el capítulo 2 describimos el análisis amortizado que tiene un uso muy difundido para analizar las estructuras de datos; las definiciones y ejemplos son de [10]. En el capítulo 3 exponemos ampliamente el problema de mantener

conjuntos disjuntos y analizamos una de las soluciones más conocidas, la dada por Tarjan; esto en su mayor parte es de [43, 45]. En el capítulo 4 describimos la máquina de apuntador y presentamos una implantación de la solución del capítulo 3; todas las ideas son de [42] y sólo su implantación ha sufrido nuestros cambios. En el capítulo 5 mostramos uno de los resultados importantes de este problema, la cota inferior de $\Omega(m\alpha(m+n,n)+n)$ para la complejidad de los algoritmos que solucionan el problema. La prueba es la de Tarjan dada en [42] con las correcciones de Banachowski [5]. En las conclusiones pusimos una lista de aplicaciones que usan nuestra solución, o sea, su funcionamiento se basa en el algoritmo del capítulo 3; además, mencionamos algunos resultados sobre la complejidad de tiempo en el peor caso y el caso promedio.

Capítulo 1

Conceptos Preliminares

Durante el presente trabajo vamos a realizar análisis de algoritmos, para lo cual es necesario saber qué son los algoritmos y la forma de describirlos, qué es lo que analizamos de ellos y sobre cual modelo de computación los pensamos implantados. En las siguientes tres secciones describimos esto.

1.1 Modelo de computación

Históricamente, el primer modelo de computación fue la **máquina de Turing** [46]; pero para nuestro propósito modelos más adecuados son la **máquina de acceso aleatorio** o **RAM** (*random-access machine*) [9] y la **máquina de apuntador** (*pointer machine*) [34].

Un RAM consiste de un **programa**, una colección finita de **registros**, cada uno de los cuales puede guardar un solo entero o un número real, y una **memoria** formada por un arreglo con N **palabras**, cada palabra tiene una **dirección** única entre 1 y N pudiendo contener un entero o número real. En cada paso, la máquina, puede realizar una operación lógica o aritmética sobre el contenido de algunos registros específicos; traer hasta un registro el contenido de una palabra cuya dirección está en otro registro; o guardar el contenido de un registro en una palabra cuya dirección está en algún registro.

En la máquina de apuntador, la memoria consiste en una colección de **nodos**, cada uno dividido en un número fijo de **campos**. Un campo puede contener un número o un apuntador a un nodo. Para usar la información, la máquina debe tener un conjunto de registros con apuntadores a nodos en la memoria. Operaciones sobre el contenido de los registros, leer los campos de un nodo y la creación o destrucción de nodos toman tiempo constante.

La máquina de apuntador nos facilita el estudio de cotas inferiores para la complejidad de los algoritmos que estudiamos aquí, por lo que hablaremos más sobre este modelo en el capítulo 4.

Los modelos que utilizamos comparten dos propiedades: son **secuenciales**, o sea, ejecutan un paso o instrucción en cada **unidad de tiempo**; y son **determinísticos**, esto es, la conducta futura de la máquina es únicamente determinada por su configuración presente.

1.2 Algoritmos

Informalmente, un algoritmo es una sucesión finita de pasos que se pueden ejecutar en un tiempo finito y que reciben una entrada para obtener una salida. Una vez establecido el modelo, podemos pensar cada paso del algoritmo como un conjunto finito y fijo de instrucciones elementales del modelo. Expresamos los algoritmos en un código que describe en cada línea un paso; esta codificación es llamada seudocódigo.

Los algoritmos pueden ser evaluados por gran variedad de criterios; en los que estamos interesados son el crecimiento en tiempo y espacio requeridos para que un algoritmo resuelva casos grandes de un problema, ya implantado en un modelo. Para esto asociamos a cada caso válido del problema un número entero, llamado el **tamaño de la entrada**, lo cual es una medida para la cantidad de entrada de datos y depende de cada caso particular. Por ejemplo, el tamaño de la entrada en un problema de gráficas puede ser el número de aristas de la gráfica.

Para determinar el tiempo y espacio de una implantación del algoritmo en un modelo, debemos conocer el tiempo requerido para ejecutar cada línea de seudocódigo en el modelo y el espacio usado por cada registro. Utilizamos el criterio del **costo uniforme** en ambos casos, o sea, cada línea requiere una unidad de tiempo y cada registro requiere una unidad de espacio. Otro criterio, a veces más realista, toma en cuenta el tamaño de cada palabra en memoria, y es llamado **costo logarítmico**; éste carga, para cada operación, un tiempo proporcional al número de bits necesarios para representar el operando.

El tiempo requerido por un algoritmo, o **tiempo de corrida**, en una entrada particular es entonces, el número de líneas de seudocódigo ejecutadas. Cada línea de seudocódigo puede requerir de un tiempo diferente para su ejecución en nuestros modelos, pero asumimos que cada ejecución de la i-ésima línea toma un tiempo C_i, donde C_i es constante y existe una constante real $C > 0$, independiente del tamaño de la entrada, tal que $C_i \leq C$ para toda $i \geq 1$.

El tiempo de corrida expresado como una función del tamaño de la

entrada será el máximo tiempo de corrida tomado sobre todas las entradas del tamaño dado, y la llamamos **complejidad del tiempo de corrida** del algoritmo, o más familiarmente **costo en tiempo**; también es conocido como **complejidad del peor de los casos**. Si como función se toma el tiempo de corrida de la entrada más probable, se le llama **complejidad del caso promedio**.

El comportamiento límite de la complejidad de tiempo cuando el tamaño crece lo llamamos **complejidad asintótica del tiempo**. Análogas definiciones pueden ser hechas para **complejidad de espacio** y **complejidad asintótica de espacio**.

Usamos la siguiente notación para la complejidad asintótica.

Definición 1.1 *Sean* $F, G : \mathcal{N} \times \ldots \times \mathcal{N} \to \Re$, *funciones no negativas.*

Decimos que $F(n, m, \ldots) = O(G(n, m, \ldots))$, $F(n, m, \ldots)$ *es del orden de* $G(n, m, \ldots)$, *si*

$$\exists\, C \in \Re \text{ y } \exists\, (n_0, m_0, \ldots) \in \mathcal{N} \times \ldots \times \mathcal{N} \cdot \ni \cdot \forall n \geq n_0, m \geq m_0, \ldots, \text{ y}$$

$$F(n, m, \ldots) \leq CG(n, m, \ldots).$$

Escribimos $F(n, m, \ldots) = \Omega(G(n, m, \ldots))$ *si*

$$G(n, m, \ldots) = O(F(n, m, \ldots))$$

y $F(n, m, \ldots) = \Theta(G(n, m, \ldots))$ *si*

$$F(n, m, \ldots) = O(G(n, m, \ldots)) \text{ y } F(n, m, \ldots) = \Omega(G(n, m, \ldots)).$$

1.3 Seudocódigo

Usamos las siguientes convenciones para el seudocódigo que describe nuestros algoritmos:

1. La identación indica estructura de bloque.
2. Las estructuras de control **while**, **for**, **return** e **if-then-else** tienen la misma interpretación que en Pascal.
3. Los comentarios se encierran entre los símbolos /* y */.
4. Usamos ← para denotar asignación.
5. Las variables son locales a los procedimientos donde se dan. No usamos variables globales sin indicación explícita.

6. Los elementos de un arreglo son referidos especificando el nombre del arreglo seguidos por el índice entre corchetes. Por ejemplo, $A[i]$ indica el i-ésimo elemento en el arreglo A. La notación ".." es usada para denotar un rango de valores dentro de un arreglo. Entonces $A[1 \,..\, j]$ indica el subarreglo de A consistente de los elementos $A[1],\ldots, A[j]$.

7. Los elementos compuestos son organizados dentro de **objetos**, los cuales están formados por **atributos** o **campos**. Se tiene acceso a un campo particular mediante su nombre seguido por el nombre del objeto entre paréntesis cuadrados. Por ejemplo, en una estructura de datos de árbol, un objeto v de tipo nodo tienen un atributo llamado *padre*, su padre en el árbol, y puede referirse el padre de un nodo v como $padre[v]$.

   A una variable que represente un arreglo u objeto la tratamos como un apuntador a los datos representados.

   Un apuntador puede no referir ningún objeto en absoluto. En este caso le damos el valor especial de NIL.

8. Los parámetros son pasados a un procedimiento por valor: el procedimiento que llama recibe su propia copia de los parámetros, y si asigna un valor a un parámetro el cambio no es visto fuera del procedimiento. Cuando pasamos objetos como parámetros, el apuntador a los datos que representa el objeto es copiado, pero los campos del objeto no.

Nuestro siguiente paso es dar una forma de analizar los algoritmos, en especial los que operan sobre estructuras de datos.

Capítulo 2

Análisis Amortizado

En este capítulo exponemos el concepto de análisis amortizado; describimos tres de las técnicas para realizarlo, las dos primeras serán usadas en el capítulo siguiente y la tercera la ponemos para comparar; al final del capítulo desarrollamos dos ejemplos.

2.1 Descripción

Veamos en qué consiste el análisis amortizado y qué lo motiva.

Las estructuras de datos tienen ciertas operaciones que actúan sobre ellas. Para muchas de sus aplicaciones se realiza una secuencia de operaciones, más que sólo una operación, y estamos interesados en el tiempo total de la secuencia, más que en el tiempo de cada operación individual.

Al hacer un análisis del peor caso sumamos el tiempo máximo de cada operación, lo cual resulta sumamente pesimista ya que se están ignorando los efectos correlacionados de las operaciones en la estructura. Por otra parte, un análisis usando el caso promedio puede ser inexacto dado que las suposiciones probabilísticas necesarias para realizarlo a veces son falsas.

En tal situación podemos promediar un tiempo que acote al de cualquier secuencia con el mismo número de operaciones sobre el total de operaciones. A este análisis le llamamos **amortizado**, y a su resultado **complejidad amortizada** o **costo amortizado**.

El análisis amortizado proporciona un modo más preciso, que las dos aproximaciones anteriores, para medir el tiempo que gasta cualquier tipo de operación en una estructura de datos, y cualquier secuencia de operaciones; además puede ser usado para mostrar que el costo promedio de una operación es pequeño, si uno promedia sobre una secuencia de operacio-

nes, aunque ella sola pueda ser cara. Sugiere también nuevas estructuras de datos y algoritmos, para sus operaciones, que sean eficientes en este sentido.

Por ejemplo, para estructuras de datos de baja complejidad amortizada el costo en tiempo para operaciones sucesivas puede variar considerablemente, pero de tal modo que el costo total, promediado sobre el total de operaciones, sea pequeño, esto es, los costos se van compensando. Por ello, para llevar a cabo un análisis amortizado necesitamos métodos que proporcionen cotas para los tiempos totales de las secuencias y sus variaciones durante el tiempo. Consideramos ahora tres formas de hacer esto.

2.2 Método del rasero

En el **método del rasero** (*aggregate method*) [10] se busca una función $T(n)$, $T : \mathcal{N} \to \Re$, que acote el tiempo total empleado para cualquier secuencia de n operaciones en la estructura, o sea, si σ es una secuencia tal y $A(\sigma)$ es el tiempo que se emplea en realizarla, entonces $A(\sigma) \leq T(n)$; y el **costo amortizado** por operación es entonces $T(n)/n$. Hacemos notar que este costo amortizado se aplica a cada operación aun cuando haya varios tipos de operación en la secuencia; ésta también es la razón para el nombre que le dimos en español.

2.3 Método del saldo

Para el **método del saldo** (*accounting method*) [10, 44] pensemos por un momento que nuestra computadora opera con monedas; insertando una moneda hacemos que la máquina funcione por un instante constante de tiempo. Para cada operación asignamos un cierto número de monedas, lo cual es el **costo amortizado** de la operación. Cuando éste sea mayor que su costo real asignamos la diferencia a objetos específicos en la estructura de datos, como crédito. Éste servirá después para ayudar a pagar por operaciones cuyo costo amortizado sea menor que su costo real. Nuestra meta es mostrar que todas las operaciones pueden ser realizadas en la máquina con el costo asignado, asumiendo que al comenzar no hay crédito en la estructura. Se permiten préstamos, sin embargo cualquier deuda deberá finalmente ser pagada con el crédito en la estructura.

Si podemos probar que no necesitamos préstamos para pagar por las operaciones, entonces el costo real de tiempo en cualquier segmento inicial de la secuencia está acotado por la suma de los correspondientes costos amortizados. Si necesitamos préstamos, pero tales son pagados al final de

la secuencia, también tendremos una cota para el tiempo empleado, aunque en medio de la secuencia el tiempo consumido exceda la suma de los costos amortizados por el monto de la deuda. En ambos casos habremos obtenido el costo amortizado para las secuencias con n operaciones.

Es de notar que nos interesa tal costo amortizado para secuencias con n operaciones, y $n \geq N$ para alguna $N \in \mathcal{N}$ fija, entonces no importa que tal costo no funcione para secuencias con pocas operaciones.

2.4 Método del potencial

Para el **método del potencial** (*potential method*) [10] se empieza con una estructura de datos D_0; denotamos D_i a la estructura resultante de aplicar la i-ésima operación de alguna secuencia con n operaciones a la estructura D_{i-1}, y por C_i al costo real de la i-ésima operación, para cada $i \geq 1$. Se propone una función

$$\Phi : \{D_0, D_1, \ldots, D_n, \ldots\} \rightarrow \Re$$

que llamamos de **potencial**, la cual asocia a cada estructura D_i el número real $\Phi(D_i)$, el potencial asociado a la estructura de datos. Con esto, definimos a ser el **costo amortizado** C'_i de la i-ésima operación como:

$$C'_i = C_i + \Phi(D_i) - \Phi(D_{i-1}),$$

o sea, el costo real más el incremento o decremento del potencial en la estructura. De esto se sigue que el costo amortizado de las n operaciones es:

$$\begin{aligned} \sum_{i=1}^{n} C'_i &= \sum_{i=1}^{n} (C_i + \Phi(D_i) + \Phi(D_{i+1})) \\ &= \sum_{i=1}^{n} C_i + \Phi(D_n) - \Phi(D_0). \end{aligned}$$

Como en los anteriores métodos se requiere que este costo amortizado sea una cota superior para el costo real, lo cual se logra tomando Φ tal que $\Phi(D_n) \geq \Phi(D_0)$ para toda $n \geq 1$.

Notemos adicionalmente que dada una función de potencial Φ, podemos definir una nueva función Φ' como:

$$\Phi' : \{D_0, D_1, \ldots, D_n, \ldots\} \rightarrow \Re$$

donde $\Phi'(D_i) = \Phi(D_i) - \Phi(D_0)$ y

$$\begin{aligned} C_i + \Phi'(D_i) - \Phi'(D_{i-1}) &= C_i + \Phi(D_i) - \Phi(D_0) - \Phi(D_{i-1}) + \Phi(D_0) \\ &= C_i + \Phi(D_i) - \Phi(D_{i-1}) = C'_i \end{aligned}$$

esto es, da el mismo costo amortizado que Φ; con esta función es suficiente ver que $\Phi'(D_i) \geq 0$, para toda $i \geq 1$.

Intuitivamente la diferencia $\Phi(D_i) - \Phi(D_{i-1})$ refleja, cuando es positiva, que sobrecargamos el costo C'_i y tal sobrecarga la guardamos como energía potencial; cuando es negativo estamos descargado el costo C'_i, y para pagar la operación usamos la energía en la estructura con su consecuente disminución.

El método del rasero es el más simple de los tres, pero puede ser difícil de utilizar en estructuras con operaciones fuertemente relacionadas entre sí. El método del saldo y el método del potencial son enteramente equivalentes y la elección de uno u otro dependerá de cuál se ajusta más a las dificultades del problema. Además tiene la ventaja de que se pueden ajustar a las abstracciones que se hagan del problema. En todo caso se pueden complementar unos con otros en el análisis de una estructura de datos.

A continuación presentamos dos ejemplos que ilustran el uso de los tres métodos.

2.5 Operaciones de pila

En nuestro primer ejemplo implantamos una pila usando un arreglo $S[1..l]$; el arreglo tiene el atributo $top[S]$ que indica el objeto insertado más recientemente.

```
PUSH(S, x)
/* Pone el objeto x dentro de la pila S */
1 if PILA-LLENA(S)
2    then error "overflow"
3    else top[S] ← top[S] + 1
4         S[top[S]] ← x
```

```
POP(S)
/* Saca el objeto del tope de la pila S */
1 if PILA-VACÍA(S)
2    then error "underflow"
3    else top[S] ← top[S]-1
4         return S[top[S]+1]
```

Dado que estas operaciones corren en tiempo $O(1)$, consideramos su costo como 1 y el costo total de una secuencia de n operaciones, entre PUSH(S, x) y POP(S) es, por lo tanto, $\Theta(n)$.

A esta estructura añadimos una operación llamada MULTIPOP(S, k), la cual remueve los k objetos en el tope de la pila, o bien, saca el contenido total de ésta si contiene menos de k objetos.

```
MULTIPOP(S, k)
/* Remueve min(|S|, k) objetos de la pila S */
1 while not PILA-VACÍA(S) and k ≠ 0
2       do POP(S)
3          k ← k − 1
```

MULTIPOP se basa en la operación de POP, que tiene un costo de 1; al realizar MULTIPOP en una pila con $|S|$ objetos, el **while** se satisface en total min($|S|$, k) veces y éste es el número de operaciones POP usados en la llamada, por lo que su costo es proporcional a min($|S|$, k).

Analicemos ahora una secuencia de n operaciones POP, PUSH y MULTIPOP, en una pila S inicialmente vacía. Basta observar que cada objeto que sea sacado, tuvo que haber sido metido en la pila, por lo que el número de veces que es llamado POP(S) en una pila no vacía, incluyendo las llamadas hechas por un MULTIPOP(S, k), es a lo más tantas como los PUSH(S, x) que se hicieron, lo cual no sobrepasa n; de lo cual concluimos que cualquier secuencia de n operaciones toma un tiempo $O(n)$, $\forall n \in \mathcal{N}$. El costo amortizado de cada una de las tres operaciones es, según el método del rasero, $O(1)$.

Para el método del saldo asignamos los siguientes costos amortizados:

```
PUSH(S, x)      2;
POP(S)          0;
MULTIPOP(S, k)  0;
```

notemos que el costo amortizado de MULTIPOP(S, k) es constante mientras que su costo real es variable. Estos costos, obtenidos del razonamiento dado arriba, intentan distribuir el costo variable de MULTIPOP sobre las operaciones de PUSH.

Utilizamos una moneda para representar una unidad de costo y pensamos en la estructura como una pila de charolas, entonces la operación de PUSH recibe 2 monedas y las otras operaciones no reciben monedas. Mostramos ahora cómo pagar por una secuencia de n operaciones de pila usando el costo amortizado. Comenzamos con una pila vacía; cuando se pone una charola en la pila, usamos una moneda para pagar por la operación de PUSH y dejamos la moneda sobrante como crédito encima de la

charola. En cualquier momento todas las charolas en la pila tendrán una moneda como crédito.

Cuando ejecutamos una operación de POP pagamos su costo usando el crédito guardado en la pila, o sea, de la charola que saquemos, tomamos la moneda que contiene y pagamos el costo de la operación.

Para pagar el costo de una operación de MULTIPOP, al sacar la primera charola tomamos la moneda de crédito que contiene y pagamos el costo de esa operación de POP; así actuamos con todas las charolas que tengamos que sacar.

Dado que cada charola tiene una moneda y la pila siempre tiene un número no negativo de éstas, tenemos que el crédito siempre es no negativo. Entonces para cualquier secuencia de n operaciones en la estructura, la suma de los costos amortizados de cada operación es $O(n)$ y funciona como una cota superior para el costo real de la secuencia.

2.6 Incrementando un contador binario

Como segundo ejemplo tenemos la implantación de un contador binario de k bits usando un arreglo $A[0..k-1]$ de bits que tiene el atributo *length*, con el valor de la longitud del arreglo. Un número binario x que está en el contador tiene su bit de orden más bajo en $A[0]$, el más alto en $A[k-1]$ y es tal que $x = \sum_{i=0}^{k-1} A[i]2^i$; inicialmente $x = 0$ y para incrementar en 1 usamos el siguiente procedimiento:

```
INCREMENTA(A)
/* Incrementa en 1 el contador */
1 i ← 0
2 while i < length[A] and A[i] = 1
3       do A[i] ← 0
4          i ← i + 1
5 if i < length[A] then
6       A[i] ← 1
```

Al comienzo de cada iteración, debemos añadir un 1 a la posición i; si $A[i] = 1$ lo hacemos cambiando a cero el bit i-ésimo y llevando un acarreo de 1; de otro modo el ciclo termina, entonces, si $i < length[A]$ sabemos que $A[i] = 0$ y sumamos el uno del acarreo en la posición i prendiendo el bit. Es claro que el costo de INCREMENTA es proporcional al número de bits cambiados.

De nuevo usamos en el método del saldo una moneda para representar cada unidad de costo, que en este caso será el de cambiar un bit.

Asignamos 2 monedas a la operación de prender un bit y 0 para apagar un bit. Cuando prendemos un bit, usamos una moneda para pagar el

costo de la operación y colocamos la que sobra en el bit como crédito. En cualquier momento, todo bit prendido tendrá una moneda de crédito; para pagar por la operación de apagar un bit, usamos la moneda de crédito que éste contiene.

Podemos así determinar el costo amortizado de la operación de INCREMENTA. El costo de apagar los bits dentro del **while** se paga con las monedas contenidas en los bits que son apagados. A lo más un bit es prendido en la línea 6, y el costo amortizado es de 2. El número de bits prendidos es siempre no negativo, por lo que el crédito es no negativo. Entonces, para n operaciones de INCREMENTA, el costo total de la secuencia es $O(n)$.

En el método del potencial, definimos a ser la función de potencial, $\Phi(D_i) = b_i$, el número de bits prendidos en el contador; como inicialmente el contador empieza en 0, todos los bits están apagados y $\Phi(D_0) = 0$; además el número de bits prendidos es no negativo, por lo que $\Phi(D_i) \geq 0$, para toda $i \geq 1$. Calculamos el costo amortizado de una operación de INCREMENTA en el contador. Suponiendo que la operación apagó t_i bits, el costo de la operación es a lo más $t_i + 1$, por el bit que posiblemente prendemos al final; y el número de bits que permanecen prendidos es entonces, a lo más, $b_{i-1} - t_i + 1$; por lo que

$$C_i' = C_i + b_i - b_{i-1} \leq (t_i + 1) + (b_{i-1} - t_i + 1) - b_{i-1} = 2,$$

de lo cual concluimos que en n operaciones el costo es $O(n)$.

El método da la flexibilidad de analizar un contador que no empiece inicialmente en 0. Supongamos que el contador empieza con b bits prendidos y después de las n operaciones hay b_n bits prendidos; de la definición de costo amortizado obtenemos:

$$\begin{aligned} \sum_{i=1}^{n} C_i &= \sum_{i=1}^{n} C_i' - \Phi(D_n) + \Phi(D_0) \\ &= \sum_{i=1}^{n} C_i' - b_n - b. \end{aligned}$$

Como C_i' es independiente del valor específico $\Phi(D_0)$, tenemos que $\sum_{i=1}^{n} C_i'$ es $O(n)$; y como $b - b_n \leq b$, $\forall n \geq 1$, el costo de n operaciones está acotado por una función $O(n)$ más el factor b; si n es $\Omega(b)$ la función es $O(n)$, o sea, si realizamos suficientes operaciones en la estructura el costo se mantiene como $O(n)$.

FALTA PAGINA

No. 12

Capítulo 3

Conjuntos Disjuntos

Introducimos ahora el problema que da origen a esta tesis. Damos primero una estructura de datos, con sus operaciones implantadas en dos formas, que lo resuelven, para finalmente determinar su complejidad en tiempo mediante un análisis amortizado.

3.1 El problema

El problema de mantener conjuntos disjuntos consiste en poder determinar en cualquier momento, a qué conjunto pertenece un elemento cualquiera y unir los conjuntos de dos elementos dados, esto sobre una colección de conjuntos disjuntos. Inicialmente hay un universo $\{ x_1, x_2, \ldots, x_n \}$ de n elementos y se realiza una sucesión cualquiera de las siguientes operaciones:

CREA-CONJUNTO(x, l) Crea un nuevo conjunto con etiqueta l conteniendo como único elemento a x, el cual no está previamente en ningún conjunto. Supondremos que se ejecuta una operación por cada elemento del universo.

ENCUENTRA-ELEMENTO(x) Determina el conjunto que contiene al elemento x y tiene como salida su nombre.

UNE(x, y) Crea un nuevo conjunto que es la unión de los que contienen a los elementos x y y, la etiqueta es la del viejo conjunto conteniendo al elemento x, y destruye los dos conjuntos anteriores. La operación se realiza sólo si los elementos x y y están inicialmente en conjuntos diferentes.

Nuestro planteamiento puede ser puesto en términos generales especificando las entradas válidas (**instancias del problema**), las salidas correctas (**soluciones**) y la relación que guardan estas dos. La entrada será una colección finita de conjuntos, cada uno de los cuales tiene un nombre distinto y un solo elemento; una operación de CREA-CONJUNTO por cada uno de los conjuntos anteriores; y una sucesión de operaciones UNE(x, y) y ENCUENTRA-ELEMENTO(x). La salida será una sucesión de nombres de conjuntos. Cada nombre corresponde a la respuesta correcta de una operación ENCUENTRA-ELEMENTO. Esto nos permite hacer la siguiente

Definición 3.1 *Por PCD(m, n, t) denotamos una instancia del problema de mantener conjuntos disjuntos con n elementos iniciales, m operaciones de* ENCUENTRA-ELEMENTO *y un total de t operaciones. Además el número de operaciones* UNE *es a lo más $n - 1$.*

Por las restricciones hechas para CREA-CONJUNTO, los conjuntos existentes en cualquier momento son disjuntos y definen una partición de los elementos en clases de equivalencia [12].

Hacemos una clara distinción entre sucesión de operaciones dada en **línea** (*on line*) y fuera de **línea** (*off line*).

Definición 3.2 *La ejecución en línea de una sucesión de operaciones σ requiere que las operaciones en σ sean ejecutadas de izquierda a derecha, ejecutando la i-ésima operación sin tener conocimiento de las siguientes operaciones. La ejecución fuera de línea de σ permite que σ sea recorrida antes que cualquier respuesta sea producida.*

Claramente cualquier algoritmo para una sucesión de operaciones dada en línea puede ser usado para una dada fuera de línea pero al revés no es necesariamente cierto. Durante este trabajo estaremos interesados sólo en algoritmos para sucesiones de operaciones dadas en línea. Sólo en el capítulo final haremos algunos comentarios sobre algoritmos para el caso fuera de línea.

3.2 Solución

Procedemos a dar una primera solución. Con cada conjunto distinguimos un elemento arbitrario pero único, llamado **elemento canónico**, el cual sirve para representar al conjunto. Cada elemento es representado por un objeto de tipo **nodo** con los siguientes atributos. El atributo *símbolo* tiene un símbolo para el elemento representado y el atributo *etiqueta* contiene el

nombre del conjunto donde está este elemento, este nombre es actualizado sólo para los elementos canónicos.

Para llevar a cabo ENCUENTRA-ELEMENTO y UNE usamos dos operaciones de bajo nivel que manipulan elementos canónicos:

BUSCA(x) Regresa el elemento canónico del conjunto que contiene al elemento x.

JUNTA(x, y) Combina los conjuntos cuyos elementos canónicos son x y y en un solo conjunto y hace a x, o bien a y, el elemento canónico del nuevo conjunto. La etiqueta del nuevo conjunto es la etiqueta del viejo conjunto que contiene al elemento x. Esta operación requiere que $x \neq y$.

Los procedimientos ENCUENTRA-ELEMENTO y UNE se implantan como sigue:

```
UNE(x, y)
1 e ← BUSCA(x)
2 f ← BUSCA(y)
3 if e ≠ f
4    then JUNTA(e, f)

ENCUENTRA-ELEMENTO (x)
1 return etiqueta[BUSCA(x)]
```

Ahora usamos la estructura de datos propuesta por Galler y Fischer [19]. Representamos cada conjunto mediante un árbol con raíz. Los vértices del árbol son los elementos del conjunto y el elemento canónico es la raíz del árbol. Cada elemento x tiene un apuntador a su **padre** en el árbol, guardado en un atributo nuevo llamado *padre*, y la raíz apunta a sí misma. Ver figura 3.1. De ahora en adelante no hacemos distinción entre elementos de un conjunto, vértices en un árbol y objetos tipo **nodo** que representen a los elementos.

Para realizar CREA-CONJUNTO(x) hacemos *padre*[x] igual a x. Para BUSCA(x) seguimos los apuntadores hacia los padres, desde x hasta la raíz del árbol, y regresamos la raíz, lo cual asocia una trayectoria a cada operación BUSCA. Para JUNTA(x, y) hacemos que *padre*[y] sea x y éste será el elemento canónico del nuevo conjunto.

Este primer algoritmo no es eficiente, pues cuando el árbol es una trayectoria de n elementos, requiere de $O(n)$ unidades de tiempo para cada operación BUSCA(x). Añadimos dos tipos de mejoras para disminuir el tiempo total de ejecución.

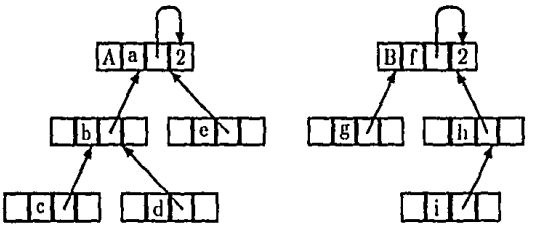


Figura 3.1: La estructura de datos utilizada en la representación para la partición del universo. Los conjuntos son $A = \{a, b, c, d, e\}$ y $B = \{f, g, h, i\}$.

La primera idea es mantener los árboles poco profundos después de cada operación JUNTA usando un nuevo atributo, para la estructura de nodo, llamado *rango*. Galler y Fischer [19] propusieron **unión por pesos** (*linking by size*). Mantenemos en el atributo *rango*, del elemento canónico, el tamaño del árbol y hacemos que la raíz del árbol más chico apunte a la raíz del árbol más grande. Una variante es la de **unión por altura** (*linking by rank*) inventada por Tarjan [45]. Mantenemos una cota superior para la altura, en el atributo *rango* de los elementos canónicos, y hacemos que la raíz del árbol menos alto apunte a la raíz del árbol más alto. Ambas versiones tiene efecto parecido pero **unión por altura** es preferible ya que requiere menos recursos de memoria.

La segunda idea es cambiar la estructura del árbol durante una operación BUSCA(x) acercando los vértices a la raíz. McIlroy y Morris [2] (Knuth se lo atribuye a Tritter [8]) proponen la mejora de **compresión por trayectoria** (*path compression*). Después de localizar la raíz r del árbol que contiene a x hacemos que cualquier vértice en la trayectoria de x a r apunte directamente a la raíz; esto incrementa el tiempo de la operación en un factor constante pero ahorra suficiente tiempo a las posteriores operaciones como para pagarse por sí mismo.

En conclusión, la estructura de nodo tiene 4 tributos: *símbolo*, *etiqueta*, *rango* y *padre*. El valor *símbolo*[x] tiene un símbolo para el elemento x. El valor *padre*[x] tiene el padre del elemento x en el árbol. Los valores *etiqueta*[x] y *rango*[x] están actualizados sólo para elementos canónicos; el primero es el nombre del conjunto donde está x y el segundo es un valor asociado al tamaño del conjunto de x. Los siguientes programas usan los métodos anteriores.

CREA-CONJUNTO(x, l)

1 *padre*[x] $\leftarrow x$

2 $rango[x] \leftarrow 0$
3 $etiqueta[x] \leftarrow l$
4 $símbolo[x] \leftarrow x$

JUNTA(x, y)

/* Versión para unión por altura. En el atributo *rango* guardamos un entero que servirá para mantener una cota superior a la altura del árbol. La implantación para unión por pesos es análoga, sólo que en $rango[x]$ se mantiene el peso del árbol.

- Si $rango[x] > rango[y]$ hacemos y hijo de x y x es el elemento canónico.
- Si $rango[x] < rango[y]$ hacemos x hijo de y y y es el elemento canónico.
- Si $rango[x] = rango[y]$ hacemos x hijo de y, incrementamos $rango[y]$ en 1 y y es el elemento canónico. La etiqueta del conjunto es la de x.

*/

1 **if** $rango[x] > rango[y]$
2 **then** $padre[y] \leftarrow x$
3 **else** $padre[x] \leftarrow y$
4 $etiqueta[y] \leftarrow etiqueta[x]$
5 **if** $rango[x] = rango[y]$
6 **then** $rango[y] \leftarrow rango[x]+1$

BUSCA(x)

/* Utilizamos un procedimiento recursivo. Si x es la raíz, entonces regresamos $x = padre[x]$, lo cual termina la recursión. De otro modo llamamos a la función recursivamente con parámetro $padre[x]$, lo cual regresa un apuntador a la raíz y es puesto en $padre[x]$, que se regresa como resultado */

1 **if** $x \neq padre[x]$
2 **then** $padre[x] \leftarrow$ BUSCA($padre[x]$)
3 **return** $padre[x]$

BUSCA(x)

/* Versión no recursiva. En la primera iteración se encuentra la raíz del árbol. En la segunda se actualiza el atributo *padre* en todos los vértices de la trayectoria para que apunten a la raíz */

```
1 raíz ← x
2 viejo ← x
3 while padre[raíz] ≠ raíz
4       do raíz ← padre[raíz]
5 while padre[viejo] ≠ viejo
6       do nuevo ← padre[viejo]
7       padre[viejo] ← raíz
8       viejo ← nuevo
9 return raíz
```

Van Leeuwen y Van Der Weide [49, 47] propusieron otros dos modos para acotar las trayectorias durante una operación BUSCA(x). **Dividir a lo largo** (*splitting*) y **dividir en dos** (*halving*). En el primero, durante la operación hacemos que cada vértice en la trayectoria hacia la raíz apunte a un vértice que esté dos lugares adelante (excepto para el último y el penúltimo). Esto rompe la trayectoria en dos que tienen la mitad de la longitud de la original. En la segunda, hacemos alternadamente que un vértice apunte al padre de su padre, o bien, no lo cambiamos. Ambas requieren de un solo recorrido de la trayectoria en cada operación BUSCA, a diferencia de **compresión de trayectoria** que requiere dos. Usamos sólo **dividir a lo largo** en nuestro trabajo.

BUSCA(x)

/* Durante el recorrido de la trayectoria desde x hasta la raíz del árbol hacemos el padre de un vértice ha ser el padre de su padre */

```
1 ancestro ← padre[x]
2 while ancestro ≠ padre[ancestro]
3       do padre[x] ← padre[padre[x]]
4       x ← ancestro
5       ancestro ← padre[x]
6 return ancestro
```

3.3 Análisis

Hicimos ya un análisis de la complejidad para la primera solución, ahora damos el análisis para nuestra solución con las mejoras anteriores.

Como las operaciones ENCUENTRA-ELEMENTO y UNE las construimos a partir de las básicas BUSCA y JUNTA, basta que consideremos sucesiones de este tipo de operaciones para determinar el costo de tiempo en nuestra solución. Hasta el final de este capítulo usamos por extensión $PCD(m,n,t)$

para una instancia del problema pero con operaciones CREA-CONJUNTO, BUSCA y JUNTA.

Para unión por pesos y compresión de trayectorias, Fischer [12] encontró una complejidad de $O(t \log(\log(n)))$. Hopcroft y Ullman [23] probaron una complejidad de $O(t \log^*(n))$ para unión por altura y compresión por trayectoria, donde definimos para $j \geq 1$

$$\log^i(j) = \begin{cases} j & \text{para } i = 0, \\ \log(\log^{(i-1)}(j)) & \text{para } i > 0 \text{ y} \log^{(i-1)}(j) > 0, \\ \text{indefinido} & \text{para } i > 0 \text{ y} \log^{(i-1)}(j) \leq 0 \text{ o} \\ & \log^{(i-1)}(j) \text{ es indefinido;} \end{cases}$$

$$\log^*(j) = \min\{i |\ \log^{(i)}(j) \leq 1\}.$$

Tarjan y Van Leeuwen [45] obtienen una complejidad de $O(m\alpha(m + n, n) + n)$ para unión por altura o por pesos y compresión de trayectoria o dividir a lo largo, donde $\alpha(i, j)$ es la función inversa de la **función de Ackermann**.

Definición 3.3 *Definimos la función de Ackermann $A(i, j)$ para $i, j \in \mathcal{N}$ por*

$$\begin{array}{lll} A(1, j) = & 2^j & \text{para } j \geq 1 \\ A(i, 1) = & A(i-1, 2) & \text{para } i \geq 2 \\ A(i, j) = & A(i-1, A(i, j-1)) & \text{para } i, j \geq 2. \end{array}$$

Definición 3.4 *Definimos la función inversa de Ackermann, $\alpha(i, j)$ para $i \geq j \in \mathcal{N}$ por*

$$\alpha(i, j) = \min\{l \geq 1 |\ A(l, \left\lfloor \frac{i}{j} \right\rfloor) > \log(j)\}.$$

Las definiciones de la función de Ackermann y su inversa se han ido ajustando durante el tiempo y surgen a partir de los requerimientos de la prueba [10], nosotros usamos la dada en [43]. Una característica sobresaliente de la función de Ackermann [1, 43] es su acelerado crecimiento en ambas entradas. La función $\alpha(i, j)$ no es realmente una función inversa de $A(i, j)$ pero captura la esencia de una función inversa, crece tan despacio como la función de Ackermann lo hace rápido.

Si fijamos el valor de n, cuando m crece, la función $\alpha(m + n, n)$ es monótonamente creciente. Para ver esta propiedad basta notar que $\lfloor \frac{m+n}{n} \rfloor$ es monótonamente creciente cuando m crece; entonces, como n es fijo, el valor mínimo necesario para que $A(i, \lfloor \frac{m+n}{n} \rfloor)$ sea mayor que $\log(n)$ es monótonamente decreciente. Esta propiedad puede ser vista intuitivamente en la estructura dada, con la mejora de compresión por trayectoria,

como sigue: para n elementos iniciales, cuando el número m de operaciones ENCUENTRA-ELEMENTO se incrementa, esperamos que en promedio la longitud de sus trayectorias decrezca. Si, como aseguramos, realizar m operaciones de ENCUENTRA-ELEMENTO toma un tiempo de $O(m\alpha(m+n,n))$, entonces el tiempo promedio por operación es de $\alpha(m+n,n)$.

Es de notar que $\alpha(m+n,n) \leq 4$ para todo propósito práctico. Dado que la función de Ackermann es estrictamente creciente en cada entrada, ver Apéndice, $A(i, \lfloor \frac{m+n}{n} \rfloor) \geq A(i,1)$ para todo $i \geq 1$. En particular $A(4, \lfloor \frac{m+n}{n} \rfloor) \geq A(4,1)$. Pero

$$\begin{aligned} A(4,1) &= A(3,2) \\ &= 2^{2^{\cdot^{\cdot^{2}}}\}16}, \end{aligned}$$

lo cual es más grande que el número estimado de átomos en el universo observable (aproximadamente 10^{80}). Entonces sólo para valores imprácticos de n, $A(4,1) \leq \log(n)$. Otras propiedades son enunciadas en el Apéndice.

Todo el resto del capítulo lo dedicamos a la prueba de la cota dada por Tarjan y van Leeuwen en [45]; los pocos cambios hechos los señalamos explícitamente.

Primero, para una trayectoria en la operación BUSCA decimos que se hace un **acortamiento de trayectoria** si cada vértice en la trayectoria, excepto el último y el penúltimo, apunta a un vértice que está a una distancia de al menos dos hacia adelante. Compresión por trayectoria es localmente el mejor tipo de acortamiento, mientras que dividir a lo largo es localmente el peor.

Medimos los cambios de apuntadores causados por el acortamiento con respecto a un bosque inicial, llamado **bosque de referencia**, el cual es producido por una secuencia de operaciones ignorando las operaciones BUSCA. Durante esta parte definimos el **rango** de un elemento, *rango*(x), como su altura en el bosque de referencia y *tamaño*(x) como el peso del subárbol que genera; si unión por altura es usada para JUNTA, entonces el rango de un vértice es el último valor asignado a *rango*[x].

Las siguientes propiedades se mantienen en cualquier secuencia de operaciones usando una forma de acortamiento de trayectoria.

Lema 3.5 *Sea B un bosque de referencia de la sucesión de operaciones σ. Si x es un elemento de B, entonces el valor padre[x] es un ancestro de x en B durante la ejecución de σ. Además, si $x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_k$ es la trayectoria de una operación* BUSCA *en σ, entonces rango(x_0) < rango(x_1) < ... < rango(x_k).*

Demostración Se prueba usando inducción sobre el número de operaciones JUNTA. □

Lema 3.6 *Sea x cualquier vértice en un bosque de referencia. Si usamos unión por altura o por pesos, entonces tamaño$(x) \geq 2^{rango(x)}$.*

Demostración Procedemos por inducción sobre el número de operaciones JUNTA. Para cero operaciones se cumple el enunciado claramente. Supongamos, como hipótesis inductiva, el resultado para k operaciones. Para $k+1$ operaciones el enunciado se mantiene mientras no se altere el valor de $rango(x)$ en la última operación. La única forma en que cambie $rango(x)$ es que JUNTA haga a un vértice w, con $rango(w) = rango(x)$, hijo de x, pero entonces

$$\begin{aligned} tamaño'(x) &= tamaño(x) + tamaño(w) \\ &\geq 2^{rango(x)} + 2^{rango(w)} \\ &\geq 2^{rango(x)+1} \\ &\geq 2^{rango'(x)}, \end{aligned}$$

donde $rango'(x)$ y $tamaño'(x)$ denotan los valores después de realizar la operación JUNTA. □

Corolario 3.7 *En un bosque de referencia, usando unión por altura o por pesos, el número de vértices de rango i es a lo más $n/2^i$.*

Demostración Por el Lema 3.5, los rangos de los vértices son estrictamente crecientes a lo largo de cualquier trayectoria en el bosque, entonces los vértices con rango i no están relacionados en el bosque de referencia y los descendientes de cualesquiera dos son conjuntos disjuntos, por el Lema 3.6 cada vértice con rango i tiene al menos 2^i descendientes, por lo tanto hay a lo más $n/2^i$ vértices de rango i. □

Corolario 3.8 *En un bosque de referencia, usando unión por altura o por pesos, la distancia de cualquier vértice x a la raíz no excede $log(n)$, y $rango(x)$ está acotado por $log(n)$.*

Demostración Consecuencia del Corolario 3.7, pues $n/2^{\log(n)+1} < 1$. □

Procedemos a acotar el total de vértices en las trayectorias de las operaciones BUSCA. Para esto usamos la técnica de **partición múltiple** (*multiple partition*) de Tarjan [38], que usa el incremento de los rangos de los padres a lo largo del tiempo para agrupar nodos y poder contabilizar sus gastos.

Definimos una partición de los enteros desde 0 hasta el máximo rango de un vértice. Hay una partición por cada **nivel** i en $[0, \ldots, \kappa]$. La clase j-ésima, llamada **bloque de la partición**, para el nivel i es definida como

$$bloque(i,j) = [B(i,j), B(i,j+1) - 1] \text{ para } j \in [0, l_i - 1],$$

donde las cotas de los intervalos, $B(i,j)$; el número de bloques en el nivel i, l_i; y el número de niveles, κ; son parámetros ha ser determinados después. Denotamos por H al máximo rango de un vértice.

Para que nuestra definición tenga sentido requerimos de una función $B(i,j)$, con $i \in [0,\kappa]$ y $j \in [0,l_i]$, las siguientes propiedades:

1. $B(0,j) = j$ para $j \in [0,l_0]$.
2. $B(i,0) = 0$ para $i \in [1,\kappa]$.
3. $B(i,j) < B(i,j+1)$ para $i \in [1,\kappa], j \in [0,l_i - 1]$.
4. $B(i,l_i) > H$ para $i \in [0,\kappa]$.
5. $l_\kappa = 1$.

Primero, la propiedad 3 implica que los bloques de partición en el nivel i son intervalos disjuntos no vacíos. Segundo, las propiedades 2 y 4 implican que cualquier entero en $[1,H]$ está en algún bloque de partición en el nivel i. Tercero, la propiedad 1 implica que cada bloque de partición en el nivel 0 consiste en un solo entero. Finalmente, la propiedad 5 implica que la partición del nivel κ consta de un solo bloque.

En cada nivel los bloques particionan a los vértices por rangos. Para $i \in [1,\kappa], j \in [0,l_i - 1]$, definimos $n_{i,j}$ como el número de vértices con rango en $bloque(i,j) - bloque(i-1,0)$.

Corolario 3.9 *Para cualquier* $i \in [0,\kappa]$,

$$\sum_{j=0}^{l_i-1} n_{i,j} \leq n.$$

Demostración En cada nivel i hay n vértices. □

Intentamos que los bloques sean cada vez más amplios conforme los niveles aumentan. Como una medida para este aumento denotamos por $b_{i,j}$, para $i \in [1,\kappa], j \in [0,l_i - 1]$, al número de bloques en el nivel $i-1$ cuya intersección con $bloque(i,j)$ es no vacío.

Definimos el nivel que le corresponde al vértice x, $nivel(x)$, como

$$\min\{i |\ rango(x), rango(padre[x]) \in bloque(i,j), \quad \text{p. a. } j \in [0,l_i - 1]\}.$$

La posibilidad de tal definición se sigue de la propiedad 4. Además, cada $nivel(x) \in [1,\kappa]$, a menos que x sea la raíz de un árbol en el bosque de referencia, en cuyo caso $nivel(x) = 0$. Aunque el rango de un vértice es fijo, su nivel puede incrementarse pero no decrecer.

Usamos el método del saldo para acotar el número de vértices en las trayectorias de las operaciones BUSCA en un $PCD(m,n,t)$, que es proporcional a su costo de tiempo. El crédito asignado por operación es repartido entre los vértices en la trayectoria y la operación misma, tal que dependa de los niveles justo antes de realizar la operación. El crédito asignado a un vértice es guardado en los niveles que serán especificados después.

Crédito por operación Asignamos un crédito de 3κ para cada operación BUSCA.

Crédito por vértice Sea x un vértice en la trayectoria de BUSCA, diferente del primero y del último. Sea I el valor máximo de la función *nivel* para los vértice precediendo a x en la trayectoria. Si

$$\min\{I, nivel(padre[x])\} \geq nivel(x)$$

entonces asignamos

$$\min\{I, nivel(padre[x])\} - nivel(x) + 1$$

a x.

Esta cantidad es guardada en el intervalo

$$[nivel(x), \min\{I, nivel(padre[x])\}].$$

Ver figura 3.2.

Notemos que la regla de crédito por vértice no asigna crédito al penúltimo vértice de la trayectoria. Veamos como usar este crédito para pagar el costo de cada operación BUSCA, después sumamos el total de créditos y obtenemos una cota superior al costo real de las operaciones BUSCA.

Teorema 3.10 *La cantidad de crédito para una operación* BUSCA *en un* $PCD(m,n,t)$ *es al menos el número de vértices en su trayectoria.*

Demostración Sea $x = x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_h$ la trayectoria de la operación. Tomamos

$$P = \{i \in [0, h-1] |\ nivel(x_i) \leq nivel(padre[x_i])\} \quad \text{y}$$

$$N = \{i \in [0, h-1] |\ nivel(x_i) > nivel(padre[x_i])\};$$

claramente $P \cup N \cup \{h\} = [0,h]$ y $P \cap N = \emptyset$.

Notemos lo siguiente:

$$\sum_{i=0}^{h-1} nivel(x_{i+1}) - nivel(x_i) = nivel(x_h) - nivel(x_0) \geq -nivel(x_0).$$

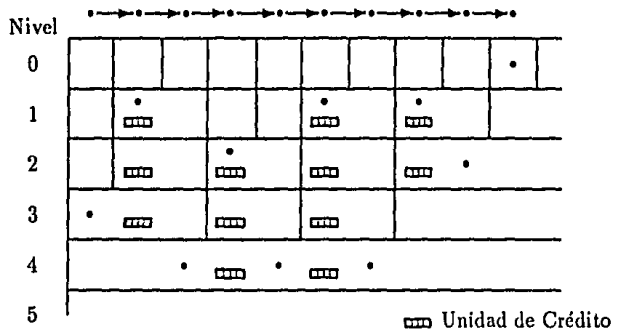


Figura 3.2: Ejemplo de la asignación de crédito por vértice en una trayectoria para la partición del dibujo. Además, se muestra el modo de guardar este crédito.

Despejando los término en P obtenemos

$$\begin{aligned}
&\sum_{i\in P}(nivel(x_{i+1}) - nivel(x_i)) \\
&\geq \; -nivel(x_0) + \sum_{i\in N}(nivel(x_i) - nivel(x_{i+1})) \\
&\geq \; -nivel(x_0) + |N|.
\end{aligned}$$

Sumando $|P|$ en ambos lados

$$\begin{aligned}
\sum_{i\in P}(nivel(x_{i+1}) - nivel(x_i) + 1) \;&\geq\; -nivel(x_0) + |N| + |P| \\
&=\; h - nivel(x_0).
\end{aligned}$$

Sea $y_0, \ldots, y_g$ la subsucesión máxima de la sucesión de vértices formada con los x_i tales que

$$nivel(x_i) > nivel(x_j)\ \forall j < i.$$

De la definición $y_0 = x_0$ y $g \leq \kappa - 1$, pues κ es el nivel máximo.

Procedemos a determinar la cantidad de créditos dados a la operación. Para que un vértice x_i reciba crédito por vértice es necesario que $i \in P$. Tenemos dos casos:

- Si $x_{i+1} \notin \{y_0, \ldots, y_g\}$, entonces existe y_j que está antes que x_i en la trayectoria y cuyo nivel es mayor al de x_{i+1}; en consecuencia, x_i recibe un crédito de

$$nivel(x_{i+1}) - nivel(xi) + 1.$$

- Si $x_{i+1} = y_j$, p. a. $j \in [1, \kappa]$ y $x_i \neq y_{j-1}$, entonces x_i recibe $nivel(y_{j-1}) - nivel(x_i) + 1$ de crédito, o sea, al menos

$$nivel(y_{j-1}) - nivel(x_i) = [nivel(x_{i+1}) - nivel(x_i) + 1] - [nivel(y_j) - nivel(y_{j-1}) + 1]$$

tal cantidad se mantiene aún si $x_i = y_{j-1}$, pues x_i no recibe crédito.

Como el anterior análisis considera todos los vértices $U = \{x_i | \ i \in P\}$, sumamos las cantidades obtenidas para cada vértice en U, algunas de las cuales son 0, mas 3κ, del crédito por operación, para obtener un total de al menos

$$\sum_{i \in P}(nivel(x_{i+1}) - nivel(x_i) + 1) - \sum_{j=1}^{g}(nivel(y_j) - nivel(y_{j-1}) + 1) + 3\kappa$$

$$\begin{aligned} &\geq h - nivel(x_0) - nivel(y_g) + nivel(y_0) - g + 3\kappa \\ &\geq h - \kappa - (\kappa - 1) + 3\kappa > h. \end{aligned}$$

□

Lema 3.11 *El total de créditos por vértice para todas las operaciones* BUSCA *en un PCD(m, n, t) es a lo más*

$$\sum_{i=1}^{\kappa}\sum_{j=0}^{l_i - 1} b_{i,j} n_{i,j}.$$

Demostración A cada vértice x que recibe crédito por vértice le haremos una marca por cada unidad recibida, las marcas son de diferentes tipos: ponemos una marca de tipo i si su nivel es exactamente i antes de una operación BUSCA. Veremos que si $nivel(x) = i$ y el rango de x está en $bloque(i, j)$, entonces no ponemos más de $b_{i,j}$ marcas de tipo i. Además, de estos vértices marcados, cuyos rangos estén en $bloque(i, j)$, no marcamos más que a $n_{i,j}$. La idea de marcar los vértices es nuestra.

Consideremos un vértice x que recibe crédito por vértice y su nivel es i. Antes de la operación BUSCA, $1 \leq nivel(x) \leq i$ y después, $nivel(x) \geq i$.

Supongamos que x recibe un crédito de $q+1$, de la regla para crédito por vértice se sigue que $nivel(padre[x]) \geq q+i$ y existe x', en la trayectoria de la operación, que está antes que x y $nivel(x')-nivel(x) \geq q$. Como $nivel(padre[x]) \geq i$, $rango(padre[x])$ y $rango(padre[padre[x]])$ están en diferentes bloques de nivel $i-1$ antes de la operación y después de ella $rango(padre[x])$ y $rango(x)$ estarán separados por al menos un bloque más de nivel $i-1$. Esto no puede suceder más de $b_{i,j}-1$ veces antes de que se incremente $nivel(x)$. Ponemos en cada uno de estos casos una marca de tipo i a x. Si $q>0$, entonces $nivel(x)$ pasa a ser al menos $i+q$, y en este caso hacemos una marca adicional de tipo i, lo cual haría tener a x a lo más $b_{i,j}$ marcas de tipo i; podemos biyectar las q unidades de crédito con marcas de tipo $i,\ldots,i+q-1$, que se hubieran desperdiciado. Además, $nivel(x') \geq i+q$, entonces $rango(x) \notin bloque(i-1,0),\ldots, bloque(i+q-1,0)$, por lo que las marcas que hicimos son cuando $rango(x) \in bloque(i+l,j_l)-bloque(i+l-1,0)$, para $l \in [0,\kappa]$. Concluimos que no marcamos más de $n_{i,j}$ vértices cuando su nivel es i.

Sumando todas las marcas obtenemos el resultado. □

Corolario 3.12 *El total de vértices recorridos por las trayectorias de m operaciones* BUSCA *en un $PCD(m,n,t)$ es a lo más*

$$3m\kappa + \sum_{i=1}^{\kappa}\sum_{j=0}^{l_i-1} b_{i,j}n_{i,j}.$$

Demostración Por el Lema 3.10 la cantidad total de crédito acota el total de vértices. El Lema 3.11 da una cota para el total de crédito por vértice y $3m\kappa$ es el total de crédito por operación. □

Lema 3.13 *Si utilizamos unión por altura o por pesos para la operación* JUNTA *en un $PCD(m,n,t)$ tenemos que*

$$n_{i,j} \leq \frac{n}{2^{\max\{B(i,j),B(i-1,1)\}-1}} \quad \text{para } i \in [1,\kappa], j \in [0,l_i-1].$$

Demostración Sabemos por el Corolario 3.7 que hay a lo más $\frac{n}{2^l}$ vértices con rango l; dado que $n_{i,j}$ son todos los vértices cuyos rangos están en $bloque(i,j)-$bloque$(i-1,0)$, o sea, en el intervalo $[B(i,j),\ B(i,j+1)-1]-[0,\ B(i-1,1)-1]$ obtenemos

$$\begin{aligned} n_{i,j} &\leq \sum_{l=\max\{B(i,j),B(i-1,1)\}}^{B(i,j+1)-1} \frac{n}{2^l} \\ &\leq \sum_{l=\max\{B(i,j),B(i-1,1)\}}^{\infty} \frac{n}{2^l} \end{aligned}$$

$$= \frac{n}{2^{\max\{B(i,j),B(i-1,1)\}-1}}.$$

□

Lema 3.14 *Si usamos unión por altura o unión por pesos y cualquier forma de acortamiento de trayectoria, entonces el total de vértices en las trayectorias que siguen las operaciones* BUSCA *en un PCD(m,n,t) es a lo más*

$$3m\alpha(m+n,n)+11n+4m.$$

Demostración Aquí el rango máximo es de $\log(n)$, por el Corolario 3.8. Tomemos $\kappa = \alpha(m+n,n)+1$; $l_i = \min\{j|\ A(i,j) > \log(n)\}$, para $i \in [1,\alpha(m+n,n)]$ y $l_\kappa = 1$; definimos también

$$\begin{aligned} B(i,j) &= A(i,j), \text{ si } i \in [1,\alpha(m+n,n)], j \in [1,l_i] \text{ y} \\ B(\kappa,1) &= \lfloor \log(n) \rfloor + 1. \end{aligned}$$

Las propiedades 3 y 4 se siguen de las propiedades para la función de Ackerman en el Apéndice.

Tenemos que acotar la suma

$$\sum_{i=1}^{\alpha(m+n,n)+1} \sum_{j=0}^{l_i-1} b_{i,j} n_{i,j},$$

para lo cual estimamos los valores de $b_{i,j}$:

(i) $b_{i,0} \leq 2$, para $i \in [1,\alpha(m+n,n)]$.

$$\begin{aligned} \mathit{bloque}(i,0) &= [B(i,0), B(i,1)-1] \\ &= [0, A(i,1)-1] \\ &= [0, A(i-1,2)-1] \\ &= [0, B(i-1,2)-1] \\ &= [0, B(i-1,1)-1] \cup [B(i-1,1), B(i-1,2)-1]. \end{aligned}$$

(ii) $b_{\alpha(m+n,n)+1,0} \leq \lfloor \frac{m+n}{n} \rfloor$.

$$A(\alpha(m+n,n), \lfloor \frac{m+n}{n} \rfloor) > \log(n)$$

por definición. De donde

$$l_{\alpha(m+n,n)} \leq \lfloor \frac{m+n}{n} \rfloor.$$

(iii) $b_{i,j} \leq A(i,j)$, $i \in [1, \alpha(m+n,n)]$, $j \in [1, l_i - 1]$.

Si $i = 1$, entonces

$$\begin{aligned} bloque(1,j) &= [B(1,j), B(1,j+1) - 1] \\ &= [A(1,j), A(1,j+1) - 1] \\ &= [2^j, 2^{j+1} - 1] \end{aligned}$$

y $b_{1,j} \leq 2^{j+1} - 1 - 2^j \leq 2^j = A(1,j)$.

Si $i > 1$, entonces

$$\begin{aligned} bloque(i,j) &= [B(i,j), B(i,j+1) - 1] \\ &= [A(i,j), A(i,j+1) - 1] \\ &\subseteq [0, A(i,j+1) - 1] \\ &= [0, A(i-1, A(i,j)) - 1] \\ &= [0, B(i-1, A(i,j)) - 1]. \end{aligned}$$

Ahora procedemos a acotar las siguientes sumas parciales:
Primero,

$$\sum_{i=1}^{\alpha(m+n,n)+1} b_{i,0} n_{i,0} \leq 3n + m.$$

Si usamos $|bloque(i,j)|$ para el número de vértices con rango en $bloque(i,j)$, entonces $n_{i,0} = |bloque(i,0)| - |bloque(i-1,0)|$, para $i > 0$, y

$$\sum_{i=1}^{\alpha(m+n,n)} n_{i,0} = |bloque(\alpha(m+n,n),0)| - |bloque(0,0)| \leq n;$$

por lo que

$$\begin{aligned} \sum_{i=1}^{\alpha(m+n,n)+1} b_{i,0} n_{i,0} &= \sum_{i=1}^{\alpha(m+n,n)} b_{i,0} n_{i,0} + b_{\alpha(m+n,n)+1,0} n_{\alpha(m+n,n)+1,0} \\ &\leq 2 \sum_{i=1}^{\alpha(m+n,n)} n_{i,0} + \lfloor \frac{m+n}{n} \rfloor n \quad \text{de (i) y (ii)} \\ &\leq 2n + m + n. \end{aligned}$$

Esto es una mejora a la dada en [45] de $5n + m$.

Segundo,

$$\sum_{i=1}^{\alpha(m+n,n)} \sum_{j=1}^{l_i - 1} b_{i,j} n_{i,j} \leq \sum_{i=1}^{\alpha(m+n,n)} n \left(\frac{A(i,1)+1}{2^{A(i,1)-2}} \right).$$

Usamos el Lema 3.13 con $\kappa = \alpha(m+n,n)$ y obtenemos

$$n_{i,j} \leq \frac{n}{2^{\max\{B(i,j),B(i-1,1)\}-1}} = \frac{n}{2^{A(i,j)-1}}.$$

Entonces

$$\begin{aligned}
\sum_{j=1}^{l_i-1} b_{i,j} n_{i,j} &\leq \sum_{j=1}^{l_i-1} A(i,j) \frac{n}{2^{A(i,j)-1}} \\
&= n \sum_{j=1}^{l_i-1} \frac{A(i,j)}{2^{A(i,j)-1}} \\
&\leq n \sum_{j=A(i,1)}^{\infty} \frac{j}{2^{j-1}} \\
&= n \left(\frac{A(i,1)+1}{2^{A(i,1)-2}} \right).
\end{aligned}$$

Concluimos que

$$\sum_{i=1}^{\alpha(m+n,n)} \sum_{j=1}^{l_i-1} b_{i,j} n_{i,j} \leq \sum_{i=1}^{\alpha(m+n,n)} n \left(\frac{A(i,1)+1}{2^{A(i,1)-2}} \right).$$

Como $A(i,1) \geq 2, i \geq 1$, podemos aumentar sumandos y obtener la siguiente serie que acota a la anterior suma

$$\begin{aligned}
n \sum_{h=2}^{\infty} \frac{h+1}{2^{h-2}} &= n \sum_{h=2}^{\infty} \left(\frac{1}{2^{h-2}} + 2 \frac{h}{2^{h-1}} \right) \\
&\leq n(2 + 2 \times 3) \\
&= 8n.
\end{aligned}$$

Por lo tanto

$$\begin{aligned}
&\sum_{i=1}^{\alpha(m+n,n)+1} \sum_{j=0}^{l_i-1} b_{i,j} n_{i,j} \\
&= \sum_{i=1}^{\alpha(m+n,n)+1} b_{i,0} n_{i,0} + \sum_{i=1}^{\alpha(m+n,n)} \sum_{j=1}^{l_i-1} b_{i,j} n_{i,j} \\
&\leq 11n + m
\end{aligned}$$

y del Corolario 3.12 se sigue el resultado. □

Finalmente, tenemos el resultado principal del capítulo.

Teorema 3.15 *Para el problema de mantener conjuntos disjuntos existe un algoritmo que tiene una complejidad asintótica de tiempo*

$$O(n + m\alpha(m + n, n)).$$

Demostración Consideremos el algoritmo descrito usando la operación JUNTA con unión por altura o unión por pesos y la operación de BUSCA con compresión de trayectoria o dividir a lo largo. Cada operaciones de CREA-CONJUNTO y JUNTA toman tiempo constante y su complejidad de tiempo total es $O(n)$. La operación de BUSCA está implantada con una forma de acortamiento de trayectoria, y su complejidad de tiempo es proporcional al total de vértices recorridos, del Lema 3.14 obtenemos una complejidad en tiempo para las operaciones de BUSCA de $O(n + m\alpha(m + n, n))$. □

En los dos siguientes capítulos veremos que para los algoritmos llamados separables esto es lo mejor que podemos obtener.

Capítulo 4

Máquina de Apuntador

Ahora nos hacemos una pregunta difícil de contestar. ¿No hay un modo de implantar las operaciones para resolver el problema de mantener conjuntos disjuntos, cuya complejidad asintótica sea una función lineal en el número de operaciones?. Claro que una pregunta tal debe hacerse en el marco de referencia de un modelo de computación. Si el modelo es una **máquina de apuntador**, entonces la respuesta es *no*; aún más, podemos decir que $m\alpha(m+n,n)+n$ es la menor complejidad asintótica en tiempo de un $PCD(m,n,t)$. Para esto necesitamos analizar más a detalle la máquina de apuntador.

Las descripciones del modelo de computación y la solución del problema de conjuntos disjuntos en una máquina de apuntador provienen de las dos primeras secciones en [42]. Además, tomamos la idea de representación de un entero por una lista y el Lema 4.1 de [42]. Añadimos el algoritmo que incrementa un contador binario de [10], explicado en el capítulo 2, para incrementar un número en 1.

4.1 El modelo

Una máquina de apuntador consiste en una **memoria** y un número finito de **registros**. Los registros son de dos tipos: de **datos** y de **apuntador**. La memoria consiste en un número finito, pero expandible, de **celdas** (*records*). Cada celda consiste en un número finito de **campos**, cada uno de los cuales es campo de datos, o bien, campo de apuntador. Cada campo tiene un nombre de identificación. Todas la celdas son idénticas en estructura, o sea, ellas constan de los mismos campos. Ver figura 4.1.

Una máquina de apuntador manipula datos y apuntadores. Un apunta-

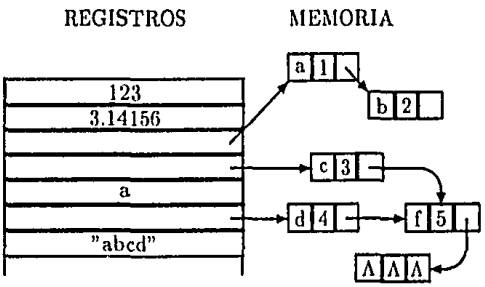


Figura 4.1: Visualización de una máquina de apuntador.

dor especifica una celda en particular, o bien, tiene el valor NIL ($\emptyset$). Cada registro de apuntador y campo de apuntador puede guardar un apuntador. Los datos pueden ser de cualquier tipo (enteros, valores lógicos, cadenas, reales, etc.). Cada registro de datos y campo de datos puede guardar un dato.

Un programa para una máquina de apuntador consiste en una sucesión finita de instrucciones de los ocho tipos siguientes y la última siempre será **halt**.

Denotamos a un registro de apuntador por r, un registro de datos por s y un registro de cualquier tipo por t; cada n denota el nombre de un campo en una celda.

$r \leftarrow \emptyset$ Coloca el valor NIL en el registro r.

$t_1 \leftarrow t_2$ Coloca el contenido del registro t_2 en el registro t_1.

$t \leftarrow n[r]$ Coloca el contenido del campo n de la celda especificada por el registro r en el registro t.

$n[r] \leftarrow t$ Coloca el contenido del registro t en el campo n de la celda especificada por el registro r.

$s_1 \leftarrow s_2 \theta_3$ Coloca el resultado de aplicar la operación θ a s_1 y s_2 en s_1.

create r Crea una nueva celda, o sea, no estaba especificada por ningún apuntador, y coloca un apuntador a ésta en r. Todos los campos inicialmente contienen un valor especial llamado *INDEFINIDO*(Λ).

halt Cesa la ejecución del programa.

if *condición* **then go to** *i* Si la condición es verdadera, entonces el control se transfiere a la instrucción *i*. Si es falsa no lo hace.

Las asignaciones sólo se realizan entre objetos del mismo tipo y cada condición es de uno de los siguientes tipos:

TRUE Siempre verdadera.

$t_1 = t_2$ Es verdadera si los contenidos de t_1 y t_2 son iguales.

$P(s_1, s_2)$ Es verdadera si el contenido de s_1 y s_2 satisface el predicado P sobre datos.

Usamos el término **símbolo** para referirnos a datos sobre los cuales sólo la verificación de igualdad es una operación permitida. Una **máquina de apuntador pura** es una máquina de apuntador que no usa datos. La máquina de apuntador que aparece en [24] sólo usa símbolos como datos.

En la máquina de apuntador, la consulta a memoria es sólo por referencia explícita; ningún calculo sobre apuntadores es posible. Aparentemente, es menos poderosa que el modelo RAM con costo uniforme, pues pierde la capacidad de usar aritmética de direcciones para propósitos como el de manejar **tablas hash** (*hash table*) [25] o realizar un **ordenamiento radix** (*radix sort*) [2]. Sin embargo, es suficientemente poderosa para simular lenguajes de procesamiento en listas como LISP [42].

4.2 Una solución

Una solución **máquina apuntador** para el problema de conjuntos disjuntos consiste en una máquina de apuntador y programas que lleven a cabo las operaciones ENCUNTRA-ELEMENTO y UNE. Inicialmente una colección de celdas de memoria representa los conjuntos de entrada. Hacemos las siguientes suposiciones:

1. Cada conjunto y cada elemento tiene asociado un símbolo distintivo.

2. Ninguna celda en la colección contiene el símbolo de otro conjunto o elemento fuera del suyo.

3. Ninguna celda en la colección de conjuntos de entrada contiene un apuntador a otra celda fuera de la colección.

4. Antes que el programa ENCUENTRA-ELEMENTO(x) sea ejecutado, un apuntador a la celda conteniendo el símbolo para x es colocado en el registro r, designado para entrada, y Λ es colocado en todos los demás. El programa para con el símbolo del conjunto que contiene a x en el registro s_0, designado para salida.

5. Antes que el programa UNE(x, w) sea ejecutado, apuntadores a celdas conteniendo los símbolos para x y w son colocados en los registros r_1 y r_2, designados para la entrada, y Λ en todos los demás; al terminar no se produce salida.

La restricción 1 impide codificar todos los elementos de un conjunto en un solo dato y mover este dato con un costo de uno; mientras que las restricciones 2, 3, 4 y 5 implican que la máquina, cuando realiza ENCUENTRA-ELEMENTO(x) o UNE(x, w), tiene acceso sólo a celdas que representen a x, o x y w. Se sigue, por inducción sobre el número de operaciones, que las restricciones 2 y 3 se mantienen para los conjuntos existentes en cualquier instante, o sea, el contenido de la memoria tiene una partición en colecciones de celdas y cada colección corresponde a un conjunto. Sin las restricciones 2 y 3 cualquier problema de conjuntos disjuntos puede ser resuelto en tiempo lineal.

La sucesión de pasos ejecutados por una solución maquina apuntador de un $PCD(m, n, t)$, la cual denotamos como $SPCD(m, n, t)$, mide el tiempo total de ejecución en la máquina de apuntador. Además, dada una $SPCD(m, n, t)$, se puede particionar en subsucesiones de pasos que corresponden a las operaciones ENCUNTRA-ELEMENTO y UNE del $PCD(m, n, t)$.

Los algoritmos del capítulo 3 tienen una implantación inmediata en la máquina de apuntador, observando que la instrucción **while** se logra con las instrucciones:

M **if** $\neg$ *condición*
  then go to $N + 2$
  ⋮
  /* cuerpo del while*/
  ⋮
N **if** *condición*
  then go to $M + 2$

4.3 Análisis

Procedemos a realizar el análisis de complejidad en tiempo para nuestra solución.

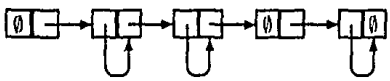


Figura 4.2: Representación de $26 = 10110_2$ como una lista.

El algoritmo requiere que las celdas contengan campos de datos enteros y que la máquina incremente en 1 y compare enteros. Veamos que podemos implantar la operación básica de JUNTA en una máquina de apuntador pura, en modo que el tiempo total de las operaciones JUNTA sea de $O(n)$ para un $PCD(m, n, t)$.

Cada entero no negativo es representado en modo binario por una lista de celdas. Cada celda contiene los campos *valor* y *siguiente*. El campo *valor* es un cero o un uno y *siguiente* tiene la siguiente celda en la lista. El cero se representa con NIL y el uno con un apuntador diferente de NIL (puede apuntar a la celda donde está). La lista empieza con el bit de orden más bajo y termina con el bit de orden más alto. Ver figura 4.2.

Para incrementar en 1 el contador del campo *rango* de la estructura de nodo, representado como una lista, usamos el algoritmo del capítulo 2.

INCREMENTA(r)

/* Donde r es un apuntador al comienzo de la lista que representa al número que se va incrementar.*/

```
1 while r ≠ ∅ and valor[r] ≠ ∅
2       do valor[r] ← ∅
3       anterior ← r
4       r ← siguiente[r]
5 if r = ∅
6   then create(nuevo)
7         siguiente[anterior] ← nuevo
8         valor[nuevo] ← nuevo
9         siguiente[nuevo] ← ∅
```

Como vimos en el capítulo 2, este algoritmo tiene una complejidad amortizada de tiempo $O(k)$, donde k es el número de operaciones INCREMENTA.

Ahora consideremos los n contadores y notemos que cada vez que se ejecuta JUNTA, uno de ellos desaparece; nuestro peor caso es que en cada operación de JUNTA tengamos que usar INCREMENTA y haya $n-1$ operaciones JUNTA. Como la complejidad de tiempo para un contador en particular es la cantidad de veces que fue incrementado, y para todos los contadores

la suma de estas cantidades no pasa de $n-1$, la complejidad de tiempo en total es $O(n)$.

Para realizar las comparaciones entre valores enteros, representados como una lista, usamos el siguiente algoritmo.

COMPARA(r_1, r_2)

/* r_1 y r_2 son apuntadores al inicio de las listas que representan los números a y b a ser comparados. Si $a \leq b$ en el registro M queda el valor $\emptyset$. Si $a > b$ en el registro M queda valor diferente de $\emptyset$.*/

```
1 M ← ∅
2 while r1 ≠ ∅ and r2 ≠ ∅ and valor[r1] = valor[r2]
3       do r1 ← siguiente[r1]
4       r2 ← siguiente[r2]
/* Si termino aquí, entonces a = b y M = ∅.*/
5 if r1 ≠ r2
6   then M = r1 halt
/* Si r1 ≠ ∅ y r2 = ∅, entonces a > b y M ≠ ∅.
Si r1 = ∅ y r2 ≠ ∅, entonces a < b y M= ∅.*/
7 while r1 ≠ ∅ and r2 ≠ ∅
8       do if valor[r1] ≠ valor[r2]
9       then M = valor[r1]
        /* Si valor[r1] ≠ ∅ y valor[r2] = ∅, hasta aquí a > b y M≠ ∅.
        Si valor[r1] = ∅ y valor[r2] ≠ ∅, hasta aquí a < b y M= ∅.*/
10 if r1 ≠ r2
11   then M = r1 halt
```

Un algoritmo análogo se puede hacer para ver si ambas listas son iguales, o bien, usar la instrucción COMPARA(r_1, r_2) = COMPARA(r_2, r_1), lo cual ocurre si y sólo si los números representados son iguales.

La complejidad de los algoritmos para comparar es proporcional a la longitud de la lista más chica. Basta acotar la suma de las longitudes de los enteros binarios menores en cada par comparado para encontrar la complejidad de todas las comparaciones. Sea $f(n)$ una cota para el peor caso de ese total en un $PCD(m, n, t)$.

$f(1) = 0$ pues sólo hay un entero y no hay comparaciones,

$$f(n) = \max\{\lceil \log(k) \rceil + 1 + f(k) + f(n-k) \mid 1 \leq k \leq n/2\},\ n \geq 1,$$

que es el más grande de sumar el peor caso para k elementos, mas el peor caso de los $n-k$ restantes y después tener que unir un conjunto de cada uno, donde se comparan dos enteros menores o iguales que k y $n-k$, entonces el tamaño del número binario menor es a lo más $\log(k)+1$.

Lema 4.1

$$f(n) \leq 2n - \log(n) - 2.$$

Demostración Por inducción sobre n.

Para $n = 1$,

$$f(1) = 0 \leq 2 - \log(1) - 2 = 0.$$

Sea $n \geq 2$ y supongamos válido el lema para todos los valores menores a n. Sea $1 \leq k_0 \leq n/2$ tal que

$$f(n) = \lceil \log(k_0) \rceil + 1 + f(k_0) + f(n - k_0).$$

Por H.I.

$$\begin{aligned} & f(n) \\ & \leq \log(k_0) + 1 + (2k_0 - \log(k_0) - 2) + (2(n - k_0) - log(n - k_0) - 2) \\ & \leq 2n - (\log(n - k_0) + 1) - 2 \end{aligned}$$

y como $\log(n - k_0) + 1 \geq \log(n)$ pues $n/2 \geq k_0$

$$f(n) \leq 2n - \log(n) - 2.$$

□

Del Lema 4.1 y el resultado enunciado antes de éste, concluimos que el total de las operaciones JUNTA, que usa los algoritmos expuesto, tiene una complejidad de tiempo $O(n)$.

Finalizamos el capítulo con el siguiente resultado.

Teorema 4.2 *Existe una máquina de apuntador pura la cual resuelve cualquier PCD(m, n, t) con una complejidad de tiempo*

$$O(m\alpha(m + n, n) + n).$$

Demostración Se sigue del análisis en el capítulo 3 y notando que podemos identificar cada símbolo para elemento y conjunto con un apuntador hacia la celda misma que lo contiene. □

FALTA PAGINA

No. 38

Capítulo 5

Una Cota Inferior

Veamos ahora que para cualquier par de algoritmos usados en las operaciones de ENCUENTRA-ELEMENTO y UNE, que tengan las características 1-5 del capítulo 4, la complejidad de tiempo de un $PCD(m,n,t)$ es $\Omega(m\alpha(m+n,n)+n)$.

En la primera sección, establecemos una relación entre las soluciones del problema de mantener conjuntos disjuntos y las soluciones para un problema de construcción de gráficas. En la segunda sección, mostramos una cota inferior para el tiempo de las construcciones de las gráficas. En la sección final, juntamos esto para poder obtener una demostración de la afirmación anterior.

5.1 Las soluciones

El primer paso es darle una forma normal a las posibles soluciones $SPCD(m, n, t)$ para poder relacionar con cada una, la construcción de una gráfica.

Notemos que para solucionar un $PCD(m,n,t)$ basta una máquina de apuntador y una sucesión de instrucciones, las cuales se puedan partir en subsucesiones correspondientes a las operaciones ENCUENTRA-ELEMENTO y UNE y las realicen. Generalizamos nuestra $SPCD(m,n,t)$ a ser todo este tipo de soluciones. Supondremos además que cualquier $SPCD(m,n,t)$ puede llevarse a cabo sin necesidad de instrucciones condicionales ya que éstas pueden ser removidas sin alterar la ejecución.

Teorema 5.1 *Sean $m,n \in \mathcal{N}$ y S_1 un $SPCD(m,n,t)$ con $t \geq m$, entonces existe un $SPCD(m,n,t)$, S_2, que soluciona también el problema y tiene las siguientes propiedades:*

1. *la longitud de la sucesión S_2 es menor o igual a $2(m+n+|S_1|)$.*
2. *La sucesión S_2 maneja sólo símbolos para conjuntos y elementos.*
3. *La sucesión S_2 representa cada conjunto de entrada mediante una sola celda y no tiene instrucciones* **create**.
4. *La sucesión S_2 busca un símbolo en memoria sólo como la última instrucción de una operación* ENCUENTRA-ELEMENTO *y no lo hace para* UNE.

Demostración Ver [12]. □

A partir del Teorema 5.1 podemos pensar a cada $SPCD(m,n,t)$ como la construcción de una gráfica. Inicialmente hay n vértices, uno por cada celda inicial. Cada que ponemos o seguimos apuntadores entre celdas, o sea, establecemos relaciones, ponemos alguna arista en la gráfica. UNE(x,y) establece una relación entre x y y por lo que debe haber una arista entre ellos. ENCUENTRA-ELEMENTO(x) recorre apuntadores hasta encontrar el nombre del conjunto, digamos y; las aristas que ponemos en la gráfica establecen la relación de todos los vértices que pasamos con y. Necesitamos de una representación de la memoria para reglamentar los movimientos de apuntadores; para lo cual usamos una gráfica que represente las operaciones de UNE.

Definición 5.2 *El* **bosque de unión** *de una sucesión de operaciones* UNE *es una gráfica donde los vértices son los conjuntos iniciales y las aristas son los pares (x,y), tales que* UNE(x,y) *ocurren en la sucesión.*

Si pusiéramos las flechas $x \rightarrow y$ para cada UNE(x,y), de la restricción para UNE tenemos que cada componente conexa en la digráfica es un árbol dirigido, entonces cada componente conexa en nuestro bosque de unión es un árbol. Elegimos como raíz al vértice que sea nombre del conjunto después de llevar a cabo todas las operaciones. Notemos que cualquier bosque es posible como bosque de unión.

Definición 5.3 *Sea p un entero no negativo. Una* **p-solución por aristas** *para un $PCD(m,n,t)$ con bosque de unión T consiste en:*

1. *Una representación de la colección de conjuntos iniciales por medio de una gráfica $G=(V,E)$, donde los vértices son los conjuntos iniciales y el conjunto de aristas es*

$$\{(x,y) |\ x \stackrel{+}{\rightarrow} y \text{ y } d(x) \geq p \text{ en } T\}.$$

A estas aristas les llamamos **gratuitas**.

2. *Una sucesión de instrucciones liga(x, y), donde* $x, y \in V$ *y* $x \xrightarrow{+} y$ *en* T. *La operación liga(x, y) inserta en la gráfica* G *todas las aristas* (w, z) *tales que*

$$x \xrightarrow{*} w \xrightarrow{+} z \xrightarrow{*} y$$

en T.

Además, la sucesión debe cumplir con las siguientes propiedades:

(a) La sucesión puede ser dividida en subsucesiones contiguas, y cada una representa una operación ENCUENTRA-ELEMENTO *o* UNE.

(b) Sea ENCUENTRA-ELEMENTO*(y) con respuesta x. Si* $x \neq y$, *entonces al terminar la subsucesión asociada con esta operación,* (x, y) *es una arista de la gráfica* G *y cada operación liga(w, z) en la subsucesión satisface*

(i) $w = x$.

(ii) $\lambda_G(w, z) = 2$ *antes de realizar liga(w, z).*

(c) Sea UNE*(x, y). En la subsucesión asociada a esta operación hay una instrucción liga(x, y) y para cualquier otra instrucción liga(w, z) en ella, las condiciones* (i) *y* (ii) *se cumplen.*

A las 0-soluciones por aristas, o sea, que no tiene aristas gratuitas, las llamamos simplemente soluciones por aristas. La complejidad asociada a la p-solución por aristas S, es la longitud de su sucesión que denotamos por $|S|$. El símbolo $u \xrightarrow{+} v$ indica que v es descendiente propio de u en el árbol T y $u \xrightarrow{*} v$, que v es descendiente impropio.

La definición de p-solución por aristas es una generalización de Banachowski [5] a la dada por Tarjan en [42], para tener una descripción precisa de cuales son las aristas gratuitas que permitimos; además refleja nuestro anterior razonamiento y la forma en que opera una máquina de apuntador. Notemos que en el inciso 1 de la definición permitimos que haya aristas cuya construcción no contamos para la complejidad; y el inciso 2 permite al algoritmo establecer muchas relaciones a un bajo costo. Debe quedar claro que el bosque de unión no tiene que ver con los árboles que consideren algunos algoritmos como representación de los conjuntos.

El siguiente teorema nos relaciona las $SPCD(m, n, t)$ con las p-soluciones por aristas.

Teorema 5.4 *Para cualquier* $SPCD(m, n, t)$ *con* k *pasos hay una solución por aristas cuya sucesión de operaciones tiene una longitud que no excede a* $4m + 5n + 4k$. *En particular esta cota se cumple para cualquier p-solución por aristas.*

Demostración Ver [42]. □

5.2 Analizando las soluciones

Veamos ahora que las p-soluciones por aristas para ciertos $PCD(m,n,t)$ no pueden tener una longitud pequeña, para lo cual usamos como bosque de unión un árbol binario completo, lo cual facilita el manejo de los conjuntos; además, representa el peor caso, en el sentido que las operaciones UNE son hechas de forma distribuida.

Para el manejo de la profundidad de los árboles consideramos la siguiente función.

Definición 5.5 *Para $i,j \in \mathcal{N}$, definimos la función $C(i,j)$ como*

$$\begin{array}{lll} C(1,j) = & 1 & \text{para } j \geq 1 \\ C(i,1) = & C(i-1,2)+1 & \text{para } i \geq 2 \\ C(i,j) = & C(i,j-1)+C(i-1,2^{C(i,j-1)}) & \text{para } i,j \geq 2. \end{array}$$

Lema 5.6 *Para $i,j \geq 1$*

$$C(i,j)+1 \leq A(i,4j).$$

Demostración Ver Apéndice. □

Teorema 5.7 *Sea $k,s \geq 1$. Sea T un árbol binario completo con profundidad $\delta > C(k,s)$ tal que contiene un conjunto de vértices*

$$V = \{v_i \mid 1 \leq i \leq s2^{C(k,s)}\}$$

no relacionados dos a dos, cada uno a profundidad estrictamente mayor que $C(k,s)$ y exactamente s vértices de V están en cada subárbol de T con raíz a profundidad $C(k,s)$. Entonces, para $n = 2^{\delta+1}-1$ y $m = s2^{C(k,s)}$, existe un $PCD(m,n,t)$ tal que:

1. *El bosque de unión es T.*
2. *Los vértices de V son los únicos sobre los que se realizan las operaciones* ENCUENTRA-ELEMENTO.
3. *La respuesta de cada operación* ENCUENTRA-ELEMENTO *es un vértice a profundidad estrictamente menor que $C(k,s)$.*
4. *Cualquier $C(k,s)$-solución por aristas tiene complejidad al menos de km.*

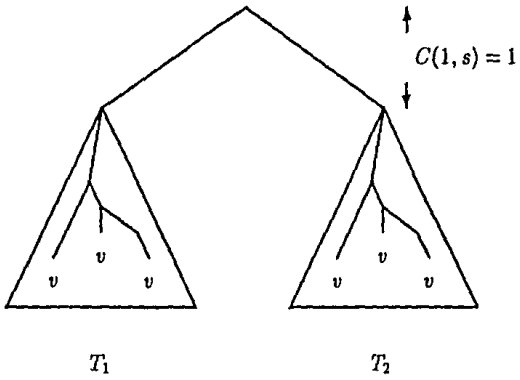


Figura 5.1: Árbol para el caso $k = 1$. T_1 y T_2 son arboles binarios completos. Cada v denota un vértice V; todos los vértices v están a una distancia de al menos 2 de la raíz.

Demostración. la posibilidad de la construcción en 1 se sigue de la definición de bosque de unión y de la libertad para elegir las operaciones se sigue el inciso 2.

Para los incisos 3 y 4 procedemos por doble inducción sobre k y s.

Si $k = 1$ y $s \geq 1$ entonces $C(k, s)$=1. Sean T y V que cumplen las condiciones del teorema. Ver figura 5.1. Formamos un $PCD(m, n, m+2n-1)$ con las $n-1$ operaciones UNE necesarias para tener a T como bosque de unión y al final las operaciones de ENCUENTRA-ELEMENTO para cada vértice en V. Cualquier $C(k, s)$-solución por aristas realiza al menos una instrucción *liga* para resolver cada operación ENCUENTRA-ELEMENTO, ya que los vértices no están relacionados, o sea, al menos km instrucciones.

Consideremos la siguiente construcción dada en [5] para los casos faltantes. Sean T y V como en el teorema; elegimos en cada subárbol con raíz a profundidad $C(k, s)$ un $w_i \in V$, $1 \leq i \leq 2^{C(k,s)}$, y formamos

$$\begin{aligned} V_1 &= \{w_i | \ 1 \leq i \leq 2^{C(k,s)}\}, \\ V_2 &= V - V_1 \text{ y} \\ V' &= \{u_i \in V(T) | \ d(u_i) = C(k, s)\} \end{aligned}$$

numerados tal que $u_i \stackrel{+}{\rightarrow} w_i$. Notemos que V_2 puede ser vacío. Sea $P_1 = PCD(m', n, m'+2n-1)$ que satisface el teorema para k', s', T y V'. Para el inciso 4, consideramos f-soluciones por arista con $f \leq C(k, s)$. Formemos

un $PCD(m', n, m' + 2n - 1) = P_2$ de P_1 sustituyendo cada ENCUENTRA-ELEMENTO(u_i) por ENCUENTRA-ELEMENTO(w_i). Sea S_2 una h-solución por aristas de P_2, con $h = f$ o $h = f + 1$. Ahora, construimos una f-solución por aristas S_1 a partir de S_2, considerando la gráfica con aristas gratuitas adecuada y sustituyendo:

> La operación $liga(x, z)$, $\forall x, z \in V(T)$, tal que $u_i \xrightarrow{+} z$, para alguna $i \in [1, 2^{C(k,s)}]$, y z no es descendiente de ningún vértice en V_2, por la operación $liga(x, u_i)$.

Además, quitamos de S_1 todas las operaciones que no produzcan nuevas aristas.

Lema 5.8 *La Sucesión S_1 es una f-solución por aristas para P_1 y*

$$|S_2| \geq |S_1| + 2^{C(k,s)} \geq k'm' + 2^{C(k,s)}.$$

La demostración de este lema se pospone para el final.

Continuamos con la demostración. Supongamos como H.I. el resultado para $k - 1$ y $s \geq 1$, con $k \geq 2$, demostraremos el caso $k \geq 2$ y $s = 1$. De la definición $C(k, 1) = C(k - 1, 2)+1$. Hagamos la construcción anterior aplicando H.I al tomar $k' = k - 1$, $s' = 2$ y $f + 1 = h = C(k, s)$, veamos que P_2 satisface el teorema. Ver figura 5.2. ENCUENTRA-ELEMENTO(w_i) tendrá la misma respuesta que ENCUENTRA-ELEMENTO(u_i) que es un vértice a profundidad estrictamente menor a $C(k - 1, 2) < C(k, 1)$. Para cualquier $C(k, s)$-solución por aristas S_2,

$$\begin{aligned} |S_2| \geq k'm' + 2^{C(k,s)} &= (k-1)2 \times 2^{C(k-1,2)} + 2^{C(k,1)} \\ &= (k-1)2^{C(k,1)} + 2^{C(k,1)} \\ &= km. \end{aligned}$$

Supongamos ahora el resultado para $k - 1$ y $s \geq 1$, también para k y $s - 1$, en ambos casos $k, s \geq 2$; demostraremos el caso $k, s \geq 2$. De la definición $C(k, s) = C(k, s - 1)+ D$, donde

$$D = C(k - 1, 2^{C(k,s-1)}).$$

Hagamos la primera parte de la construcción para el Lema 5.8. Consideremos los árboles

$$T_j,\ 1 \leq j \leq 2^D,$$

con raíz a profundidad D en T. Apliquemos la H.I. con k, $s - 1$, T_j, $V_2 \cap V(T_j)$, $n_j = 2^{d(T_j)+1}$ y $m_j = (s - 1)2^{C(k,s-1)}$. Obtenemos para cada j un $PCD(m_j, n_j, m_j + 2n_j - 1) = q_j$ que satisface los incisos 1-4.

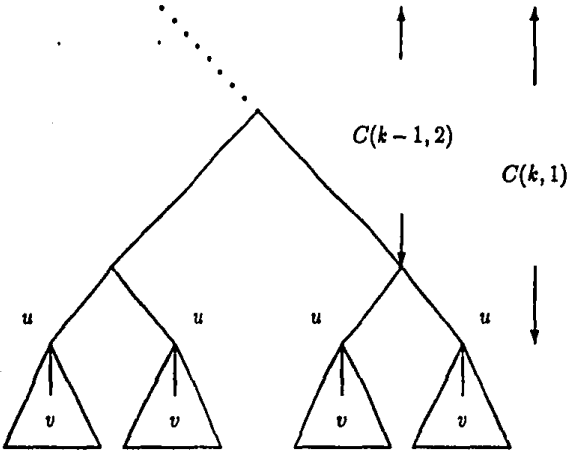


Figura 5.2: Rama del árbol T en el caso $s = 1$. Cada v denota un vértice de V_1; cada u denota un vértice de V'.

Ver figura 5.3. Continuamos la construcción original aplicando H.I. con $k' = k-1, s' = 2^{C(k,s-1)}, T$ y V'. Obtenemos el $PCD(m', n, m'+2n-1)$, P_1, que satisface los incisos 1-4. Podemos asumir que los árboles T_j son construidos al comienzo de P_1, por las aristas gratuitas que permitimos. Después, formamos P_1' de P_1 quitando las operaciones que construyen los árboles T_j. Finalmente construimos P_2 de P_1' cambiando cada ENCUENTRA-ELEMENTO(u_i) por ENCUENTRA-ELEMENTO(w_i). Veamos que el $PCD(m, n, m+2n-1)$, $P = q_1 q_2 \cdots q_{2^D} P_2$, satisface el teorema.

Por la observación anterior, P tiene a T como bosque de unión y se realizan las operaciones ENCUENTRA-ELEMENTO sobre todos los elementos de V. Para cada $v \in V_2$, la respuesta de ENCUENTRA-ELEMENTO(v) es un vértice a profundidad menor a $C(k, s-1)$ en T_j, que es un vértice a profundidad menor que $C(k,s)$ en T. Para $w_i \in V_1$, la respuesta será igual a ENCUENTRA-ELEMENTO(u_i) que es un vértice a profundidad menor que $C(k-1, 2^{C(k,s-1)}) < C(k,s)$. Sea S_2 una $C(k,s)$-solución por aristas para P, siguiendo la construcción para el Lema 5.8 obtenemos S_1, que es una $C(k,s)$-solución por aristas para $q_1 q_2 \cdots q_{2^D} P_1'$, tal que $|S_2| \geq |S_1| + 2^{C(k,s)}$. La solución S_1 se puede dividir en $C(k, s-1)$-soluciones por aristas para q_j, $1 \leq j \leq 2^D$, digamos r_j y en una D-solución por aristas para P_1, digamos

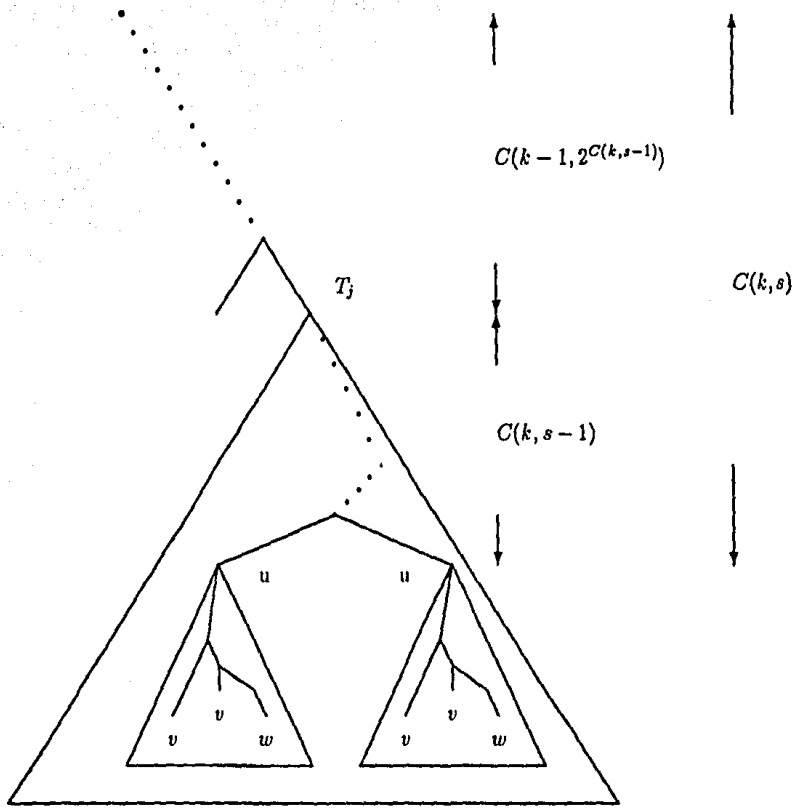


Figura 5.3: Rama del árbol T en el caso general. Cada v denota un vértice de V_2, cada w denota un vértice de V_1, cada u denota un vértice de V'. Todas las operaciones de ENCUENTRA-ELEMENTO en vértices de V_2 ocurren en T_j, las operaciones de ENCUENTRA-ELEMENTO en vértices de V_1 se realizan en el árbol T.

S_1'. Finalmente obtenemos que

$$\begin{aligned} |S_2'| &\geq \sum_{j=1}^{2^D} |r_j| + |S_1'| + 2^{C(k,s)} \\ &\geq \sum_{j=1}^{2^D} km_j + (k-1)2^{C(k,s-1)}2^D + 2^{C(k,s)} \\ &= k(s-1)2^{C(k,s-1)}2^D + (k-1)2^{C(k,s)} + 2^{C(k,s)} \\ &= ks2^{C(k,s)} = km. \end{aligned}$$

□

Demostración del Lema 5.8 La siguiente propiedad se mantiene durante la ejecución paralela de S_1 y S_2 en gráficas separadas, $G(S_1)$ y $G(S_2)$, donde inicialmente $G(S_1)$ contiene todas las aristas

$$\{(x,y) |\ x \xrightarrow{+} y \text{ y } \ d(x) \geq f\}$$

y $G(S_2)$ contiene

$$\{(x,y) |\ x \xrightarrow{+} y \text{ y } \ d(x) \geq h\}.$$

Propiedad 5.9 *Si en $G(S_2)$ se pone la arista (x,z), entonces en $G(S_1)$ está la arista*

1. *(x,u_i), si $x \xrightarrow{+} u_i \xrightarrow{+} z$ y z no es descendiente de un vértice en V_2.*

2. *(x,z), en otro caso.*

Lo cual se sigue por inducción sobre el número de operaciones *liga* en S_1.

Como S_2 soluciona las operaciones ENCUENTRA-ELEMENTO(v), para $v \in V$, entonces S_1 tambien las soluciona, dado las aristas gratuitas que permitimos. Al final de una operación ENCUENTRA-ELEMENTO(w_i), G_2 tiene la arista (x,w_i), entonces del inciso 1 se sigue que al final de ENCUENTRA-ELEMENTO(u_i) se tiene la arista (x,u_i). Durante la operación UNE(a,b), S_2 pone la arista (a,b), con $d(a), d(b) \leq h$, entonces S_1 pone la arista (a,b), o bien, ya la tenía. Las propiedades 2b(i) y 2c(i) son claras, pues no se cambia el primer parámetro de una operación *liga*. Falta ver que si *liga*(x,z) se ejecuta en S_1, entonces $\lambda_{G(S_1)}(x,z) = 2$.

Supongamos que S_2 realiza *liga*(x,z), entonces $\lambda_{G(S_2)}(x,z) = 2$ y existe un vértice t tal que están las aristas (x,t) y (t,z) en $G(S_2)$. Tenemos los siguientes casos.

(i) Si $d(z) \leq d(u_i)$, entonces (x,t) y (t,z) están en $G(S_1)$ por 5.9 y S_1 puede ejecutar *liga*(x,z).

Para los otros caos tenemos que $u_i \xrightarrow{+} z$, p. a. i.

(ii) Si existe $v \in V_2$ tal que z sea su descendiente, entonces S_1 tiene que ejecutar *liga*(x,z). Las aristas (t,z) y (u_i,z) están en $G(S_2)$, la primera por 5.9 y la segunda es gratuita. Por 5.9 está la arista (x,u_i), o bien, (x,t) en $G(S_1)$, pero en ambos casos se puede realizar la operación.

(iii) Como último caso, S_1 tiene que ejecutar *liga*(x,u_i). Como z no es descendiente de algún v en V_2, tampoco lo es t. Tenemos las siguientes situaciones:

Primero,

$$x \xrightarrow{*} u_i \xrightarrow{*} t \xrightarrow{+} z,$$

entonces (x,u_i) está en $G(S_1)$ lo cual no es posible pues quitamos las operaciones *liga* que no introducían nuevas aristas.

Segundo,

$$x \xrightarrow{+} t \xrightarrow{+} u_i \xrightarrow{+} z,$$

y por 5.9 tenemos las aristas (x,t) y (t,u_i).

Concluimos que S_1 es una f-solución para P_1.

Finalmente veamos que $|S_2| \geq |S_1| + 2^{C(k,s)}$. Consideremos la primera instrucción *liga* (x,z) en S_2 con

$$x \xrightarrow{+} u_1 \xrightarrow{+} w_1 \xrightarrow{*} z,$$

esto ocurre pues se resuelve la operación ENCUENTRA-ELEMENTO(w_1). Debe existir un vértice t tal que (x,t) y (t,z) están en $G(S_2)$. Si $d(t) < d(u_1)$, entonces se habría realizado una operación del mismo tipo antes de *liga*(x,z) lo cual es contradictorio. Tenemos que $u_1 \xrightarrow{*} t$ y está la arista (x,u_1) en $G(S_1)$, por lo que quitamos *liga*(x,u_1). Procedemos así con los demás elementos de V', por lo tanto

$$|S_2| \geq |S_1| + |V'| = |S_1| + 2^{C(k,s)}.$$

□

Corolario 5.10 *Sea $k,s \geq 1$ y T un árbol binario completo de profundidad $C(k,s)$. Entonces, existe un PCD$(m,n,m+2n-1)$, con $m = s2^{C(k,s)}$ y $n = 2^{C(k,s)+1} - 1$, cuyo bosque de unión es T y requiere al menos $(k-1)m$ operaciones liga para cualquier solución por aristas.*

Demostración Sea $l \geq 1$ tal que $2^l \geq s$. Sea T' el árbol binario completo formado mediante reemplazar cada hoja de T por un árbol binario completo de altura l. Sea

$$V = \{v_i | \, 1 \leq i \leq m\}$$

cualquier conjunto de vértices que satisface las hipótesis del Teorema 5.7 para k, s y T'. Obtenemos un $PCD(m,\ n',\ m + 2n' - 1) = P'$, donde $n' = 2^{d(T')+1} - 1$, cuyas $C(k,s)$-soluciones por aristas tiene una complejidad de al menos km; podemos asumir que las operaciones UNE que forman los subárboles de T' con raíz a profundidad $C(k,s)$ suceden al inicio del problema, por las aristas gratuitas que permitimos.

Elegimos un vértice u_i a profundidad $C(k,s)$ en T' tal que $u_i \xrightarrow{+} v_i$, cada vértice se repite s veces. Después formamos P de P' borrando las operaciones UNE antes expuestas y reemplazando cada ENCUENTRA-ELEMENTO(v_i) por ENCUENTRA-ELEMENTO(u_i). Entonces, P define un $PCD(m, n, m + 2n - 1)$ que tiene como bosque de unión a T y realiza m operaciones ENCUENTRA-ELEMENTO. Ahora, tomamos una solución por aristas de P, digamos S y formamos S' poniendo después de cada operación $liga(x, u_i)$ en S, una operación $liga(x, v_i)$; esto define una $C(k,s)$-solución de P' y $|S'| \geq km$. Por lo tanto

$$|S| \geq |S'| - m \geq (k-1)m.$$

□

Concluimos esta sección con el resultado más importante en [42]. Hacemos notar que en [42] se usa otra definición para la función de Ackermann pero el hecho de usar el Lema 5.6 (4.1 en [42]) hace que no varie la demostración sino en las constantes.

Teorema 5.11 *Existe una constante real positiva c tal que, para todo $m \geq n \geq 1$, hay un $PCD(m,\ n,\ m + 2n - 1)$ cuya solución por una máquina de apuntador requiere al menos $c\alpha(m,n)$ pasos.*

Demostración Sea $s = \lfloor \frac{m}{n} \rfloor, n \geq 3$, Elegimos

$$k = \max\{i | \, 2^{C(i,s)+1} - 1 \leq n\}.$$

Partimos los n elementos en conjuntos $A_1, \ldots, A_l$ de tamaño $2^{C(k,s)+1} - 1$, mas el conjunto A_{l+1} con los restantes, entonces $|A_{l+1}| \leq \frac{n}{2}$ y $l = \lfloor \frac{n}{2^{C(k,s)+1}-1} \rfloor$.

Cada conjunto A_i, $1 \leq i \leq l$, define un $PCD(m_i, n_i, m_i + 2n_i - 1)$ que satisface el Corolario 5.10. Concatenamos estos problemas y tenemos

$ls2^{C(k,s)}$ operaciones de ENCUENTRA-ELEMENTO, o sea, a lo más

$$\begin{aligned} \lfloor \frac{n}{2^{C(k,s)+1}-1} \rfloor \lfloor \frac{m}{n} \rfloor 2^{C(k,s)} &\leq \lfloor \frac{n}{2^{C(k,s)}} \rfloor \lfloor \frac{m}{n} \rfloor 2^{C(k,s)} \\ &\leq m. \end{aligned}$$

Añadimos las operaciones necesarias para completar el $PCD(m,n,m+2n-1)$. Cualquier solución por aristas para este problema requiere al menos

$$\begin{aligned} l(k-1)s2^{C(k,s)} &= \lfloor \frac{n}{2^{C(k,s)+1}-1} \rfloor (k-1) \lfloor \frac{m}{n} \rfloor 2^{C(k,s)} \\ &> \frac{n}{2^{C(k,s)+2}-1}(k-1) \lfloor \frac{m}{n} \rfloor 2^{C(k,s)} \\ &\geq (k-1)\frac{m}{2}\frac{1}{4} \\ &= \frac{(k-1)m}{8} \text{ operaciones } \textit{liga}. \end{aligned}$$

Entonces, por el Teorema 5.4, cualquier $SPCD(m,n,m+2n-1)$ tiene al menos

$$\begin{aligned} \frac{(k-1)m}{32} - m - \frac{5n}{4} &= \frac{(k-33)m}{32} - \frac{5n}{4} \\ &\geq \frac{(k-33)m}{32} - \frac{5m}{4} \\ &\geq \frac{(k-73)m}{32} \text{ pasos.} \end{aligned}$$

Si $\alpha(m,n) \geq 4$, entonces $k \geq \alpha(m,n)-3$, pues

$$\begin{aligned} C(\alpha(m,n)-3, \lfloor \frac{m}{n} \rfloor)+1 &\leq A(\alpha(m,n)-3, 4\lfloor \frac{m}{n} \rfloor) \text{ Lema 5.6} \\ &\leq A(\alpha(m,n)-1, \lfloor \frac{m}{n} \rfloor) \text{ Apéndice} \\ &\leq \log(n). \end{aligned}$$

Entonces

$$\frac{(k-73)m}{32} \geq \frac{(\alpha(m,n)-76)m}{32} \geq \frac{\alpha(m,n)m}{64},$$

si $\alpha(m,n) \geq 156$; pero si $\alpha(m,n) < 156$ o $n < 3$, entonces cualquier $SPCD(m,n,m+2n-1)$ requiere $m \geq \frac{\alpha(m,n)m}{156}$ pasos. Eligiendo $c = \frac{1}{156}$ obtenemos el resultado. □

5.3 La cota inferior

Damos ahora una cota inferior general para la complejidad de las soluciones del problema en una máquina de apuntador; ésta se puede extender para la clase de algoritmos, llamados **separables**, que solucionan el problema y cumplen con las siguientes características:

1. Aplican el método del elemento canónico.

2. Empiezan con una estructura de lista L que representa los conjuntos definidos por las operaciones CREA-CONJUNTO. Cada nodo es un elemento y puede contener un número arbitrario de nodos auxiliares.

3. Cada nodo contiene un número arbitrario de apuntadores a otros nodos. Cada nodo puede ser *usado* o *no-usado*, y esta etiqueta cambia en cada operación.

4. El algoritmo tiene dos tipos de pasos:

   (i) Seguir un apuntador $x \rightarrow y$ de un nodo *usado* x a un nodo *no-usado* y. Esto hace a y ser un nodo *usado*.

   (ii) Poner un apuntador de un nodo *usado* x a otro *usado* y.

5. Las operaciones CREA-CONJUNTO(x), ENCUENTRA-ELEMENTO(x) y UNE(x, y) hacen que x sea un nodo *usado*, y también y en el último caso.

   El algoritmo realiza una secuencia arbitraria de pasos que causan, para ENCUENTRA-ELEMENTO(x) tener el representante canónico del conjunto donde está x, como nodo *usado*; para UNE(x, y), la estructura de lista L refleja la unión de los conjuntos. Al final de cada operación todos los nodos son *no-usados*.

6. La estructura de lista L puede ser dividida en n componentes, tal que cada elemento está en una componente diferente y no hay apuntadores de una componente a otra. Como no hay memoria global, esta propiedad, llamada **separabilidad**, se preserva después de cada operación.

Estas características son abstracciones de los requisitos 1-5 del capítulo 4 que pedimos para nuestras soluciones máquina apuntador, y los resultados anteriores son válidos para tales algoritmos. En particular nuestras soluciones máquina apuntador son algoritmos separables.

Teorema 5.12 *Cualquier algoritmo separable que implante las operaciones del problema de conjuntos disjuntos tiene una complejidad de tiempo*

$$f(n,m) = \Omega(m\alpha(m+n,n)+n),$$

para n conjuntos iniciales y m operaciones de ENCUENTRA-ELEMENTO.

Demostración Hemos probado lo anterior para la cota $m\alpha(m,n)$, si $m \geq n$. Asumiendo que cualquier elemento está involucrado en al menos una operación, tenemos un tiempo de $\Omega(n)$; un elemento que no interviene en las operaciones lo podemos quitar.

Si $m \geq n$, entonces $1 \leq \alpha(m+n,n) \leq \alpha(m,n)$ y el resultado se sigue del Teorema 5.10.

Si $m < n$ y $m\alpha(n,n) < n$, como $\alpha(n,n) \geq \alpha(m+n,n)$, entonces

$$\Omega(m\alpha(m+n,n)+n) = n$$

y como dijimos $f(m,n) = \Omega(n)$, por lo tanto

$$f(n,m) = \Omega(m\alpha(m+n,n)+n).$$

Si $m < n$, pero $m\alpha(n,n) > n$, obtenemos del Teorema 5.10, considerando solo m elementos,

$$f(m,n) = \Omega(m\alpha(m,m)).$$

Como

$$n > \sqrt{n} > \log(n) > \alpha(n,n),$$

entonces $m > \sqrt{n}$, o sea, $m^2 > n$. De al definición de α y el hecho que $\log(m) < \log(n)$, obtenemos

$$\begin{aligned} \alpha(m,m) &\leq \alpha(n,n) \\ &\leq \alpha(m^2,m^2) \\ &\leq \alpha(m,m)+1. \end{aligned}$$

La última desigualdad se sigue de que:

$$\begin{aligned} A(\alpha(m,m)+1,1) &\geq 2A(\alpha(m,m),1) \quad \text{Apéndice} \\ &> 2\log(m) = \log(m^2). \end{aligned}$$

Por lo tanto

$$\alpha(m,m) \geq \alpha(n,n)-1 \geq \alpha(m+n,n)-1$$

y

$$\Omega(m\alpha(m+n,n)) = m\alpha(m,m);$$

junto con el hecho de que $f(n,m) = \Omega(n)$ y $f(n,m) = \Omega(m\alpha(m,m))$ concluimos

$$f(n,m) = \Omega(m\alpha(m+n,n)+n).$$

□

FALTA PAGINA

No. 54

Capítulo 6

Conclusiones

La intención de este capítulo es dar una breve discusión de los resultados expuestos en esta tesis, presentar algunas aplicaciones que tiene el algoritmo analizado en nuestro trabajo y mencionar otras investigaciones que sobre el problema y sus soluciones se han venido realizando.

6.1 Discusión de resultados

Hemos hecho un análisis de tipo amortizado para la solución del problema de mantener conjuntos disjuntos dada en [43, 45]. De éste obtuvimos que su complejidad asintótica de tiempo es $O(m\alpha(m+n,n)+n)$, en donde α es la función inversa de Ackermann y la instancia del problema tiene n conjuntos iniciales y realiza m operaciones de ENCUENTRA-ELEMENTO.

Una vez analizado el algoritmo tratamos de determinar su complejidad exacta de tiempo, para lo cual intentamos ver que existía una sucesión de instancias del problema que requerían un tiempo de ejecución $\Omega(m\alpha(m+n,n)+n)$, y así obtener una complejidad de $\Theta(m\alpha(m+n,n)+n)$.

Con esto en mente estudiamos la máquina de apuntador e implantamos nuestro algoritmo en ella. Después sacamos las características del algoritmo y usamos el hecho de que el modelo obtiene información sólo por medio de apuntadores para poder asociar a cada solución de una instancia la construcción de una gráfica. Logramos definir construcciones que requerían un número de $\Omega(m\alpha(m,n))$ instrucciones fundamentales y así mostrar que los algoritmos con las características del nuestro, la clase de los algoritmos separables, requiere un tiempo de $\Omega(m\alpha(m+n,n)+n)$.

Concluimos al final de este proceso que nuestro algoritmo, al tener un tiempo de $\Theta(m\alpha(m+n,n)+n)$, o sea, $\Omega(m\alpha(m+n,n)+n)$ y $O(m\alpha(m+$

$n, n) + n)$, es óptimo en tiempo dentro de la clase de los separables.

Para obtener la prueba de la cota inferior fue necesario unificar los resultados del trabajo en [45] y en [5]. Ya que el primero contenía una prueba con errores y el segundo mencionaba las correcciones necesarias. Además, como estos dos artículos y el análisis amortizado del algoritmo se habían publicado con definiciones diferentes de la función de Ackermann, elegimos una definición, la que se da en [43, 45] y es ahora su definición habitual, y adecuamos ambas pruebas a esta definición.

6.2 Aplicaciones

Damos una lista de aplicaciones que usan nuestro nuestro algoritmo para su funcionamiento. Todas se mencionan en [18].

1. Distribución del trabajo en dos procesadores. La entrada consiste en una colección de n tareas que requieren una unidad de tiempo de procesador y un orden parcial dado mediante una gráfica acíclica dirigida, dag (*directed acyclic graph*) con n vértices y m aristas. El objetivo es distribuir las tareas en dos procesadores para minimizar el tiempo de ejecución. El algoritmo de Gabow [17] tiene un tiempo de $O(m + n\alpha(n, n))$.

2. Distribución del trabajo en multiprocesadores. Hay dos aplicaciones relacionadas. La primera es calcular una distribución del trabajo en varios procesadores a partir de una lista de prioridad. La entrada es una colección de n tareas que requieren una unidad de tiempo de procesador con un orden parcial dado por un dag con m aristas, una lista de prioridad que da un orden total a las tareas y un número de procesadores $p \leq n$. El objetivo es distribuir las tareas en los procesadores tal que la próxima tarea a comenzar es la primera tarea disponible en la lista de prioridad. El algoritmo de Sethi [35] tiene un tiempo de $O(m + n\alpha(n, n))$.

   La segunda aplicación es optimizar la distribución de las tareas cuyas resticciones están dadas por un orden de intervalos (*interval order*), o sea, un orden parcial $P = (V, A)$, donde V es un conjunto de intervalos cerrados en la recta real y $(u, v) \in A \Leftrightarrow (x \in u, y \in v \Rightarrow x < y)$. La entrada es una colección de n tareas, un orden de intervalos con m resticciones y un número de procesadores $p \leq n$. El objeto es distribuir las tareas para minimizar el tiempo de ejecución. Papadimitriou y Yannakakis dan un algoritmo para encontrar una lista de prioridad de tiempo $O(m + n)$ [16, 31]. Combinando esto con el algoritmo de arriba se obtiene un algoritmo de tiempo $O(m + n\alpha(n, n))$.

3. El problema del mínimo dado fuera de línea [2]. El objetivo es mantener un conjunto de enteros en el rango $[1 \ldots n]$ bajo dos operaciones:

   - INSERTA(i). El cual añade el elemento i al conjunto.
   - EXTRAE-MIN. El cual borra y regresa el elemento mínimo.

   Si cada entero es añadido una sola vez y la secuencia total de operaciones es dada fuera de línea, en [2] se encuentra un algoritmo que resuelve el problema en tiempo $O(n\alpha(n,n))$.

4. Apareamiento en gráficas convexas y distribución del trabajo con tiempos de espera y espera máxima. Hay dos problemas que están muy relacionados. En el primero, el objetivo es encontrar el máximo apareamiento en una gráfica convexa, bipartita de n vértices. El algoritmo de Lipski y Preparata [28] tiene un tiempo de $O(n\alpha(n,n))$.

   En el segundo problema, la entrada es una colección de tareas que requieren una unidad de tiempo de procesador con un tiempo entero de espera y uno de espera máxima, además de un número $p \leq n$ de procesadores. Frederickson [13] da un algoritmo que involucra el problema fuera de línea del mínimo, por lo que el tiempo de ejecución para su algoritmo es $O(n\alpha(n,n))$.

5. Dos árboles generadores dirigidos. Dada una gráfica de flujo con n vértices y m aristas, el objetivo es encontrar dos árboles generadores dirigidos con el menor número de aristas en común posible. El algoritmo de Tarjan en [40] tiene un tiempo de $O(m\alpha(m,n))$.

6. Reducibilidad de gráficas de flujo. Hopcroft y Ullman [22] han construido un algoritmo para determinar si una gráfica de flujo, con n vértices y m aristas, es reducible o no, de acuerdo a la definición de Hecht y Ullman en [21], con una complejidad de tiempo $O(m \log(m))$.

   Tarjan [37] da un algoritmo más veloz que el de Hopcroft y Ullman. El algoritmo verifica la caracterización estructural de Hecht y Ullman [21]. Su método usa DFS (*depth-first search*) para recorrer la estructura de la gráfica de flujo y nuestro algoritmo para verificar la reducibilidad usando la información encontrada. La complejidad del algoritmo depende de la complejidad de nuestro algoritmo por lo que se obtiene una complejidad de $O(m\alpha(m,n))$ (en las gráficas de flujo $n = O(m)$).

7. Apareamiento de cardinalidad máxima en gráficas no bipartitas. El algoritmo de Gabow [15] tiene una complejidad de $O(nm\alpha(m,n))$, donde n es el número de vértices y m el de aristas, y se asume $n =$

$O(m)$. Un algoritmo más veloz que usa nuestro algoritmo ha sido dado por Micali y Vazirani en [30], y tiene una complejidad de $O(\sqrt{n}m)$.

8. Equivalencia de autómatas finitos. Sean dos autómatas finitos determinísticos $M_1 = (Q_1, A, \delta_1, q_0, F_1)$ y $M_2 = (Q_2, A, \delta_2, p_0, F_2)$. El problema consiste en determinar si M_1 y M_2 aceptan el mismo lenguaje, o sea, si son equivalentes. En [2] se da un algoritmo que realiza a lo más $n-1 = |Q_1|+|Q_2|$ operaciones de UNE y a lo más $m = n \times |A|$ operaciones de ENCUENTRA-ELEMENTO y tiene una complejidad de $O(m + m\alpha(m, n))$.

Es de notar que para las aplicaciones 1-7, el tiempo se puede mejorar para ser lineal, al poder aplicarse la restricción del problema de tener la sucesión de operaciones UNE fuera de línea y usar el algoritmo dado en [18].

6.3 Líneas de desarrollo del problema

La investigación sobre soluciones del problema ha continuado por diferentes caminos y no se detiene en [45] y lo expuesto en esta tesis.

Para que se tenga un panorama de las diversas líneas de desarrollo del problema, se hará a continuación una breve descripción de ellas.

Durante algún tiempo quedó la pregunta abierta de si existía un algoritmo con tiempo lineal para el problema. Esto debido a que los algoritmos separables no usan todo el poder del modelo RAM, pues tienen la restricción de separabilidad la cual no permite que un elemento de una clase tenga acceso a otro elemento de una clase diferente. Además, tienen restricciones en el modo en que los datos son representados y manipulados.

Pero en [14], Fredman y Saks muestran que cualquier implantación en el modelo CPROBE(log(n)) requiere un tiempo de $\Omega(m\alpha(m, n))$ para ejecutar m operaciones de ENCUENTRA-ELEMENTO y $n-1$ UNE con n elementos iniciales. En el modelo cell probe con palabras de tamaño b, CPROBE(b), la complejidad de tiempo en una secuencia de operaciones se define como el número de palabras que fueron consultadas. El número b de bits es un parámetro al modelo y en nuestro caso $b = \log(n)$. Este modelo es al menos tan poderosos como el modelo RAM, pues permite las mismas instrucciones, representación de datos y el direccionamiento indirecto, además éstas pueden costar menos que en el modelo RAM. Sin embargo, Fredman y Saks dicen no saber si el resultado se mantiene para el modelo CPROBE(POLYlog(n)), donde aquí $b \leq \log^t(n)$ para alguna t.

Un algoritmo con tiempo lineal para el modelo RAM, en el caso especial donde las operaciones UNE son conocidas fuera de línea, lo describen Harold

N. Gabow y Robert E. Tarjan en [18].

Todos los análisis hasta ahora han sido amortizados. El análisis para el peor caso fue considerado por N. Blum [6] y prueba una complejidad de $\Theta(\log(n)/\log\log(n))$; para la cota inferior de la complejidad asume la propiedad de separabilidad de los algoritmos. Fredman y Saks muestran en [14] que el algoritmo dado por Blum en [6] es óptimo en el modelo CPROBE(POLYlog(n)).

El análisis en el caso promedio fue considerado por Doyle y Rivest [11], Yao [51], Knuth y Schönhage [26], Yao [52] y Bollobás y Simon [7]. Muestran que bajo varias suposiciones probabilísticas el tiempo de corrida esperado es lineal, aun si sólo una de la mejoras es usada.

Manilla y Ukkonen [29] considera una variante del problema mediante permitir dar marcha atrás (*backtracking*) sobre las últimas operaciones de unión, esto es posible mediante una operación de desunir (*Deunion*). Presentan dos métodos que permiten realizar m ENCUENTRA-ELEMENTO, k UNE y k DES-UNE con una complejidad de $O((m+k)\log(n)/\log\log(n))$ y $O(k+m\log(n))$ respectivamente [50]. Una generalización más donde se le asocia un peso a cada operación de *Une* es considerada por Gambosi en [20].

La prueba de Fredman y Sacks nos dice que el algoritmo dado es óptimo aún en el modelo RAM con palabras de tamaño $\log(n)$, un tamaño razonable para nuestro problema. El resultado de Gabow y Tarjan cubre en parte el caso cuando las operaciones son dadas fuera de línea, diferente a nuestra suposición de operaciones dadas en línea. Los otros resultados cubren dos tipos de análisis que no tratamos aquí, el del peor caso y el del caso promedio.

FALTA PAGINA

No. 60

Apéndice

Enunciamos algunas de las propiedades de la función de Ackermann. Su demostración se puede realizar por inducción sobre la definición y siguiendo el orden en que se enuncian.

Definición 6.1 *Usamos la siguiente función auxiliar.*

$$\begin{aligned} g(0) &= 2 \\ g(i) &= 2^{g(i-1)} \quad \forall i \geq 1. \end{aligned}$$

Teorema 6.2

1. $i < 2^i < g(i), \quad \forall i \geq 0$. *En particular* $2i < g(i), \quad \forall i \geq 0$.
2. $A(2,i) = g(i), \quad \forall i \geq 1$. *En particular* $A(2,i+1) = 2^{A(2,i)}, \quad \forall i \geq 1$.
3. $x < A(i,x), \quad \forall i,x \geq 1$.
4. $A(i,x) < A(i,x+1), \quad \forall i,x \geq 1$.
5. $A(i,x) < A(i+1,x), \quad \forall i,x \geq 1$.
6. $A(i,2^{x+1}) < A(i+1,x+1), \quad \forall i,x \geq 1$. *En particular* $A(i,2^x) < A(i+1,x+1)$.
7. $A(i,j) \leq \frac{1}{2}A(i,j+1), \quad \forall i,j \geq 1$.
8. $C(2,j) = j+1, \quad \forall j \geq 1$.
9. $A(x,4j) \leq A(x+2,j), \quad \forall i,s \geq 2$.
10. $2j \leq A(i,j-1), \quad \forall i,j \geq 2$.
11. $A(x,1) \leq \frac{1}{2}A(x+1,1), \quad \forall x \geq 1$.

12. Si $x \geq y \geq 1$, entonces $\alpha(x,z) \leq \alpha(y,z)$. □

Demostración Lema 5.6

$$C(i,j) < A(i,4j), \quad \forall i,j \geq 1.$$

La demostración se realiza por doble inducción sobre i,j.
Probemos el caso $i = 1,2$.

$$\begin{aligned} C(1,j) &= 1 < A(1,p), \quad \forall p \geq 1. \\ C(2,j) &= j+1 \\ &< g(j) \\ &\leq A(2,4j). \end{aligned}$$

Probemos el caso $i \geq 3$ y $j = 1$. Tenemos que

$$C(i,1) = C(i-1,2)+1 < 1 + A(i-1,8),$$

pero $A(i-1,8) < A(i-1,A(i,3))$, entonces

$$C(i,1) < A(i-1,A(i,3)) = A(i,4).$$

Probemos el caso $i \geq 3, j \geq 2$. De la definición,

$$C(i,j) = C(i,j-1) + C(i-1,2^{C(i,j-1)}),$$

el resultado se sigue de que

$$\begin{aligned} C(i,j-1) &< A(i,4j-4) \\ &\leq \frac{1}{2}A(i,4j-3) \\ &\leq \frac{1}{2}A(i,4j) \text{ y} \\ C(i-1,2^{C(i,j-1)}) &< A(i-1,2^{C(i,j-1)+2}) \\ &\leq A(i-1,2^{A(i,4j-4)+1}) \\ &\leq A(i-1,2^{A(i,4j-3)}) \\ &\leq A(i-1,A(i-2,2^{A(i,4j-3)})) \\ &\leq A(i-1,A(i-1,A(i,4j-3))) \\ &= A(i-1,A(i,4j-2)) \\ &= A(i,4j-1) \\ &\leq \frac{1}{2}A(i,4j). \end{aligned}$$

□

Índice de Términos

Listamos los términos usados para esta tesis.

- $\mathcal{N}$ denota los números naturales.
- $\Re$ denota los números reales.
- $[a, b]$ denota los números naturales a, $a+1, \ldots, b$.
- x, y, ... variables usadas principalmente para denotar elementos de un conjunto.
- u, v, ... variables para denotar vértices de una gráfica.
- A, B, ... variables para denotar conjuntos.
- V denota un conjunto de vértices.
- G denota una gráfica.
- T denota un árbol.
- σ y S denotan una sucesión de operaciones.
- $V(G)$ denota los vértices de la gráfica G.
- $G(S)$ gráfica que asociamos a una sucesión de operaciones según el capítulo 5.
- $u \rightarrow v$ flecha de u a v en una gráfica.
- $\lambda_G(u, v)$ distancia entre u y v en la gráfica G.
- Para u y v vértices de un árbol T con raíz r :
  - $d(u)$ la distancia de u a r.
  - $d(T)$ la máxima distancia de r a un nodo.

- $u \xrightarrow{+} v$ v es descendiente propio de u en el árbol T.
- $u \xrightarrow{*} v$ v es descendiente impropio de u en el árbol T.

- log denota la función $\log_2$.
- $\lfloor x \rfloor$ el máximo entero menor o igual que x, $x \in \Re$.
- $\lceil x \rceil$ el mínimo entero mayor o igual que x, $x \in \Re$.
- $A(m,n)$ función de Ackermann. Ver capítulo 3
- $\alpha(m,n)$ función inversa de Ackermann. Ver capítulo 3
- $PCD(m,n,t)$ abreviación para problema de conjuntos disjuntos con m operaciones ENCUENTRA-ELEMENTO, n elementos iniciales y t operaciones totales.
- $SPCD(m,n,t)$ abreviación para solución de un $PCD(m,n,t)$.

Los nombres de subrutinas se ponen en MAYÚSCULAS; los nombres de campos se ponen en *itálicas*; las palabras reservadas del seudocódigo se ponen en **negrillas** y las constantes, como NIL, en MAYÚSCULAS; para los nombres de funciones usamos *itálicas*; usamos **negrillas** para la primera aparición de términos introducidos en el trabajo y el término en inglés en *itálicas*.

No traducimos las palabras como **while** o TRUE por ser de uso difundido. Traducimos las palabras como *memory* (memoria), *field* (campo), *register* (registro) por ser usadas con el mismo significado en español. Los términos técnicos relacionados al trabajo (máquina de apuntador, complejidad amortizada, método de el saldo) los traducimos y ponemos a la derecha de su primera aparición su nombra en inglés.

Bibliografía

[1] W. Ackermann. Zum hilbertshen aufbau der reellen zahlen. *Math. Ann.*, 99:118–133, 1928.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. On finding lowest common ancestors in trees. *Society for Industrial and Applied Mathematics Journal Computing*, 5:115–132, 1976.

[4] B. W. Arden, B. A. Galler, and R. M. Graham. An algorithm for equivalence declarations. *Comm. the Association for Computing Machinery*, 4(7):310–314, 1961.

[5] Lech Banachowski. A complement to Tarjan's result about the lower bound on the complexity of the set union problem. *Information Processing Letters*, 11(2):59–65, 1980.

[6] Norbert Blum. On the single-operation worst-case time complexity of the disjoint set union problem. *Society for Industrial and Applied Mathematics Journal Computing*, 15(4):1021–1024, 1986.

[7] Béla Bollobás and I. Simon. On the expected behavoir of disjoint set union algorithms. In *Proc. Seventeenth Annual the Association for Computing Machinery Symposium on the Theory of Computing*, pages 224–231, 1985.

[8] V. Chvátal, D. A. Klarner, and Donald E. Knuth. Selected combinatorial research problems. Technical Report STAN-CS-72-292, Comput. Sci. Dep., Stanford U., Stanford, California, 1972.

[9] S. A. Cook and R. A. Reckhow. Time-bounded random access machines. *Journal of Computer and System Science*, 7:354–375, 1973.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald R. Rivest. *Introduction to Algorithms.* The MIT Electrical engineering and computer science series. The MIT Press, 1990.

[11] I. Doyle and Ronald L. Rivest. Linear expected time of a simple union-find algorithm. *Information Processing Letters*, 5:146–148, 1976.

[12] Michael J. Fischer. Efficiency of equivalence algorithms. In R. E. Miller and J. W Thatcher, editors, *Complexity of Computer Computation*, pages 153–168, New York, 1972. Prenum Press.

[13] G. N. Frederickson. Scheduling unit-time tasks with integer release times and deadlines. *Information Processing Letters*, 5:171–173, 1983.

[14] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dinamic data structures. In *Proc. of the Twenty-first Annual the Association for Computing Machinery, Symposium on theory of Computing*, 1989.

[15] Harold N. Gabow. An efficient implementation of Edmonds' algorithm for maximum matching on graphs. *Journal Association Computing*, 23:221–234, 1976.

[16] Harold N. Gabow. A linear-time recognition algorithm for interval dags. *Information Processing Letters*, 12:20–22, 1981.

[17] Harold N. Gabow. An almost-linear algorithm for two-processor scheduling. *Journal Association Computing*, 29:766–780, 1982.

[18] Harold N. Gabow and Robert E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[19] B. A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Comm. the Association for Computing Machinery*, 7(5):301–303, 1964.

[20] G. Gambosi, G. F. Italiano, and M. Talamo. Getting back to the past in the union-find problem. In *Proc. Symposium on Teoretical Aspects of Computer Science 88, Fifth Annual Symposium*, Lecture Notes in Computer Science, pages 8–17, Berlin, 1988. Springer.

[21] M. S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. *Society for Industrial and Applied Mathematics Journal Computing*, 1:188–202, 1972.

[22] John E. Hopcroft and Jeffrey D. Ullman. An $n \log(n)$ algorithm for detecting reducible graph. In *Proc. Sixth Annual Princeton Conf. on Inf. Sciences and Systems*, pages 119–122, 1972.

[23] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *Society for Industrial and Applied Mathematics Journal of Computing*, 2(4):294–303, 1973.

[24] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*, pages 262–263. Addison-Wesley, Reading, Massachusetts, second edition, 1973.

[25] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1973.

[26] Donald E. Knuth and A. Schönhage. The expected linearity of a simple equivalence algorithm. *Theoretical Computer Science*, 6:281–315, 1978.

[27] M. J. Lao. A new data structure for the union-find problem. *Information Processing Letters*, 9:39–45, 1979.

[28] W. Lipski, Jr. and F. Preparata. Efficient algorithms for finding maximum maching in convex bipartite graphs and related problems. *Acta Informatica*, 15:329–346, 1981.

[29] H. Mannila and E. Ukkonen. The set union problem with backtracking. In *Proc. Thirteenth International Colloquium on Automata, Languages and Programming*, number 226 in Lecture Notes in Computer Science, pages 236–246, Berlin, 1986. Springer.

[30] S. Micali and V. V. Viazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum maching for general graphs. In *Proc. twenty-first Annual IEEE Symposium on Foundation of Computer Sciences*, pages 17–27, 1980.

[31] C. H. Papadimitriu and M. Yannakakis. Scheduling interval-ordered tasks. *Society for Industrial and Applied Mathematics Journal Computing*, 8:405–409, 1979.

[32] F. P. Preparata and W. Lipski, Jr. Three layers are enough. In *Proc. Twenty-third Annual IEEE Symposium on Foundation of Computer Sciences*, pages 350–357, 1982.

[33] James Roskind and Robert E. Tarjan. A note on finding minimum-cost edge-disjoint spanning trees. *Mathematics of Operations Research*, 10(4):701–708, 1985.

[34] A. Schönhage. Storage modification machines. *Society for Industrial and Applied Mathematics Journal Computing*, 9:490–508, 1980.

[35] Ravi Sethi. Scheduling graphs on two processors. *Society for Industrial and Applied Mathematics Journal Computing*, 5:73–82, 1976.

[36] R. E. Stearns and D. J. Rosenkrantz. Table machine simulation. In *Conference Record,IEEE Tenth Annual Symposium on Switching and Automata Theory*, pages 118–128, 1000.

[37] Robert E. Tarjan. Testing flow graphs reducibility. *Journal of Computer and Systems Sciences*, 9:355–365, 1974.

[38] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the Association for Computing Machinery*, 22(2):215–225, 1975.

[39] Robert E. Tarjan. Solving path problems on directed graphs. Technical Report STAN-CS-75-528, Computer Science Departament, Stanford University, 1975.

[40] Robert E. Tarjan. Edge-disjoint spanning trees and depth-first search. *Journal Association Computing*, 6:171–185, 1976.

[41] Robert E. Tarjan. Application of path compression on balanced trees. *Journal of the Association for Computing Machinery*, 26(4):690–715, 1979.

[42] Robert E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint set. *Journal of Computer and System Science*, 18(2):110–127, 1979.

[43] Robert E. Tarjan. *Data Structures and Network Algorithms*. Society for industrial and Applied Mathematics, 1983.

[44] Robert E. Tarjan. Amortized computational complexity. *Society for Industrial and Applied Mathematics Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[45] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the Association for Computing Machinery*, 31(2):245–281, 1984.

[46] A. M. Turing. On computable numbers, with an application to entscheidungs problem. *Proc. London Mathematics Society*, 43(2):544–546, 1936.

[47] T. van der Weiden. Datastructures: An axiomatic approach and the use of binomial trees in developing and analyzing algorithms. Technical report, Mathematisch Centrum, Amsterdam, 1980.

[48] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B. Elsevier and The MIT Press, 1990.

[49] Jan van Leeuwen and T. van der Weiden. Alternative path compression techniques. Technical Report RUU-CS-77-3, Rijksuniversiteit Utrecht, Utrecht, The Netherlands, 1977.

[50] J. Westbrook and Robert E. Tarjan. Amortized analysis of algorithms for set union with backtracking. *Society for Industrial and Applied Mathematics Journal Computing*, 18:1–11, 1989.

[51] Andrew C.-C. Yao. On the average behavior of set merging algorithms. In *Proc. Eighth Annual of the Association for Computing Machinery Symposium on the Theory of Computing*, pages 192–195, 1976.

[52] Andrew C.-C. Yao. On the expected performance of path compression algorithms. *Society for Industrial and Applied Mathematics Journal Computing*, 14(1):129–133, 1985.