7

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

EL ANALISIS AMORTIZADO Y SUS APLICACIONES

PROYECTO
QUE PARA OBTENER EL TITULO DE
LICENCIADO EN CIENCIAS DE LA COMPUTACION
PRESENTA

FEDERICO JUAREZ ALMARAZ

FACULTAD DE CIENCIAS
UNAM

DIRECTOR DEL PROYECTO:
M. en I. MARIA DE LUZ GASCA SOTO

2002

UNIVERSIDAD NACIONAL
AVENMA DE
MEXICO

DRA. MARÍA DE LOURDES ESTEVA PERALTA
Jefa de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo escrito:

El análisis Amortizado y sus aplicaciones

realizado por **FEDERICO JUÁREZ ALMARAZ**

con número de cuenta **09326457-6** quien cubrió los créditos de la carrera de:

Ciencias de la Computación

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	**M. en I. María de Luz Gasca Soto**
Propietario	**M. en C. Elisa Viso Gurovich**
Propietario	**M. en C. José de Jesús Galaviz Casas**
Suplente	**Dra. Amparo López Gaona**
Suplente	**M. en C. Felipe Humberto Contreras Alcala**

Consejo Departamental de Matemáticas

Dra. Amparo López Gaona

CONSEJO DEPARTAMENTAL
DE
MATEMÁTICAS

EL ANÁLISIS AMORTIZADO Y SUS APLICACIONES

Federico Juárez Almaraz.

Para mi mamá con mucho cariño.

Agradecimientos.

Agradezco primeramente a Dios, por el soplo de la vida. A mi madre por todo el cariño y amor incondicional que nos ha brindado a mi hermana y a mí, toda su vida. Por ser siempre mi modelo a seguir, admirando sobre todo el coraje con el que siempre enfrenta la vida, también por ser mi primera mentora y con todo; me siento apenado pues no se de que forma podría pagarle todo lo que ha dado por mi.

Agradezco a mi hermana Claudia, por el cariño entrañable que me tiene. A Dios por esa especial personalidad que le brindó, tan cambiante como el tiempo, la cual admiro. Hago una mención especial a mi padre Federico Juárez Arredondo, quien también a pesar de ser quien era reservó alguna enseñanza especial para mí. No olvido a mis Abuelas, en particular a Doña Francisca, a la cual le tengo un cariño muy especial por compartir conmigo su gran entendimiento de la vida.

Le doy las gracias a la Facultad de Ciencias de la UNAM, por la que me siento orgulloso, y a la peculiar educación que me ha brindado. También agradezco a todos aquellos profesores que me han instruido durante el tiempo que estado aquí. Le agradezco a la profesora Ma. de Luz Gasca Soto, Lucy, quien no solo me ha dado un enorme apoyo para la realización de este trabajo sino que compartió su saber conmigo y me ha brindado oportunidades para mi desarrollo profesional pero sobre todo le doy gracias por su amistad la cual es muy importante para mi. Claro que no puedo olvidar a toda una personalidad, Elisa Viso, quien con todo y regaños también me ha compartido grandes enseñanzas, le agradezco el tiempo que invirtió en leer, corregir y complementar este trabajo.

Agradezco a mis amigos su apoyo y consejo, desde mis amigos que han permanecido desde el pasado, Alfredo Acosta a quien conozco desde la infancia y el cual es como un hermano. A mi gran amigo incondicional Gary y su familia quienes me han brindado su cariño y amistad. De igual manera a mis amigos del presente, desde mis contemporaneos Karina y Daniel, con los cuales he compartido muchas cosas desde pequeños triunfos así como fracasos, hasta La Honorable Banda que me ha acompañado y otorgado una gran amistad: Yazmín, Liliana, Citlali, Gustavo, Erick, Canek, Rafael, Edgar y Oscar.

Índice General

Listados

Introducción

El objetivo de este trabajo es presentar una técnica del Análisis de Algoritmos denomidanda Análisis Amortizado, la cual generalmente, no es un tema que se revise en los cursos básicos de Análsis de Algoritmos de la Facultad de Ciencias de la UNAM.

Empezamos este trabajo definiendo las estrategías clásicas del Análsis Amortizado. Después éstas se aplican sobre diferentes Tipos de datos abstractos, TDA. Para cada TDA se inicia dando un panorama general de como realizar el análisis amortizado y después se formaliza el mismo. Finalmente, se presentan dos ejemplos para los cuales el impacto de realizar el análisis amortizado es notorio.

Se pretende que este trabajo sea material didáctico para cursos avanzados de Análisis de Algoritmos tanto para la licenciatura como para la maestria en ciencias de la computación. Este trabajo está organizado de la siguiente manera:

En principio, se define qué es el análisis amortizado, qué ventajas ofrece en comparación con el análisis del peor caso y el denominado caso promedio que han sido tradicionalmente la forma de analizar algoritmos. Se describen con detalle dos estrategias esenciales con las que generalmente se trabaja esta técnica.

En el siguiente capítulo se detalla la primera estrategia, denominada *Método del Potencial*, enfocando este análisis al tipo de dato abstracto Colas Binomiales, aquí se detalla como se hace un análisis del peor caso y posteriormente cuál es el razonamiento a seguir para realizar el análisis amortizado y enfatizar cuál es el objetivo del mismo.

Posteriormente, en el Capítulo 3 se realiza el estudio del tipo de dato abstracto *Skew Heap*, con ello se pretende ilustrar la utilidad del análisis amortizado, que es dar cotas más reales a los algoritmos y dejando de manifiesto que su rendimiento en cuanto a su ejecución es adecuado para ser utilizados, esto que generalmente no lo refleja el análisis del peor caso y ni el del caso promedio. Este capítulo también pretende dejar de manifiesto que esta técnica no es sencilla de usar.

En el Capítulo 4 se presenta con detalle el TDA Conjuntos Ajenos. Los Capítulos 5 y 6 están orientados a mostrar el gran potencial que tiene el análisis amortizado aplicándolo al tipo de dato abstracto *Conjuntos Ajenos* cuyas estructuras de datos y heurísticas aplicadas lo vuelven un caso interesante. El análisis que aquí se muestra es muy amplio e ilustra la segunda técnica del análisis amortizado, denominada como *Método del Banquero*, para la cual se define toda una serie de construcciones matemáticas. El Capítulo 7 describe la utilidad del tipo de dato abstracto Conjuntos Ajenos presentando dos aplica-

ciones, en una de las cuales se enfatiza el impacto que tiene sobre el tiempo de ejecución de un clásico algoritmo de gráficas, denominado *Algoritmo de Kruskal.*

Al final del trabajo se incluye un apéndice donde se define la función de Ackerman y su inversa, que son las herramientas fundamentales para el análisis amortizado de los Conjuntos Ajenos.

Capítulo 1

El Análisis Amortizado

Los métodos tradicionales empleados por el análisis de algoritmos para estudiar los Tipos de Datos Abstractos, TDA, algunas veces resultan ser muy pesimistas. Este hecho lleva a un método alternativo llamado **análisis amortizado.**

Se adapta el término de cálculo amortizado al cálculo de la complejidad computacional entendiéndose como "*el promedio* sobre el tiempo de ejecución de una secuencia de las operaciones sobre una estructura de datos o bien como la *acumulación* de los valores de una función que mide los cambios en una estructura de datos durante la ejecución de una secuencia de operaciones de la misma". Las observaciones anteriores motivan el estudio de la amortización: en las estructuras de datos utilizadas se observar toda una secuencia de operaciones en vez de la ejecución de una sola operación. Es decir, nos interesa el tiempo total de la secuencia de operaciones en vez del tiempo individual de una operación. Esto sirve para tomar ventaja de la interacción que tienen todas las operaciones que intervienen cuando se hace uso de un TDA.

R.E. Tarjan, quien formalizó el análisis amortizado, indica que:

> En el análisis del *peor caso*, en el cual se suma el tiempo de los peores casos de todas las operaciones individuales, éste puede llegar a ser indebidamente pesimista puesto que se ignoran los efectos correlacionados de las operaciones sobre la estructura. Por otro lado, un análisis de un *caso promedio* puede llegar a ser incorrecto ya que las suposiciones probabilísticas pueden llevar a un análisis falso. [7]

Así que, la motivación de realizar un análisis amortizado de los TDA's es dar tiempos de ejecución que se ajusten más a la realidad del comportamiento de los TDA's al momento de ser aplicados, pues las técnicas clásicas para hacer el análisis de algoritmos dan cotas de tiempo de ejecución más "grandes" de lo que son realmente.

De lo anterior, Tarjan concluye diciendo:

> En tal situación realizar un *análisis amortizado* en el cual promediamos el tiempo de ejecución en el peor caso por operación sobre una secuencia de operaciones puede producir una respuesta realista y robusta. [7]

Para aclarar la idea del análisis amortizado veremos un ejemplo, sobre un TDA donde el análisis del peor caso ofrece resultados no muy satisfactorios.

Supongamos que se tiene una pila, *stack*, con la habilidad de sacar k entradas a la vez, como en algunos algoritmos de *reconocimiento*.

Una implantación tradicional de este TDA es con una lista ligada de datos. El análisis del peor caso indica que la operación PUSH(x) es $O(1)$ y POP(k) es $O(n)$ ya que requiere k operaciones para los apuntadores, (además k podría ser mayor que el tamaño de la pila). La complejidad total de m_1 llamadas a la operación PUSH y m_2 llamadas a la operación POP es de $O(m_1 + m_2 n)$. Esto es un argumento del análisis clásico de algoritmos.

Consideremos ahora el siguiente argumento: no es posible sacar, con la operación POP, más entradas de las que existen en la pila, las cuales fueron metidas con la operación PUSH. Esto es, el total de los costos de las operaciones POP no pueden exceder el total de los costos de las operaciones PUSH. Por lo tanto, el costo total de cualquier secuencia de m_1 llamadas a la operación $PUSH$ y de m_2 llamadas a POP es de $O(m_1)$ en el peor caso.

El análisis clásico no está mal, simplemente generó una cota superior sobre la complejidad total y tal cota es más grande de lo que realmente sucede. Esto pasa porque el análisis no considera que la secuencia de operaciones POP en el peor caso es imposible: después de un peor caso para la operación POP, el stack queda vacío.

1.1 Las Estrategias de la Amortización

Para realizar el análisis amortizado del tiempo de ejecución de una secuencia de operaciones de un TDA, necesitamos una técnica para "promediar" los costos de las operaciones a lo largo de una secuencia. En general, sobre las operaciones del TDA con complejidad amortizada baja, los tiempos de ejecución de operaciones sucesivas pueden fluctuar considerablemente. Para analizar tal situación, debemos ser capaces de acotar tales fluctuaciones, por lo cual consideraremos dos enfoques para hacer esto:

- El enfoque del Banquero [1].
- El enfoque de los Físicos.

[1] El término que se usa actualmente es "Contable", pero éste fue el término original.

1.1.1 El punto de vista del Banquero

Este enfoque sobre la amortización, es el siguiente:

> Asumimos que la computadora es una máquina traga_monedas. Insertando una sola moneda, que llamaremos un *Crédito*, conseguimos que la máquina funcione una cantidad —constante— fija en el tiempo.

En cada operación colocaremos un cierto número de créditos, definiendo a este crédito *como el tiempo amortizado de la operación*. Así, nuestra meta es mostrar que todas las operaciones pueden ser ejecutadas con los créditos colocados, asumiendo que comenzamos sin crédito y que los créditos no utilizados pueden ser usados en operaciones posteriores.

Además podemos permitir el préstamo de créditos, siempre y cuando cualquier deuda adquirida sea eventualmente pagada por los créditos puestos a lo largo de las operaciones.

El ahorro de crédito es la cantidad de tiempo promediado hacia adelante y el préstamo es el tiempo hacia atrás todo esto sobre la ejecución. Si podemos probar que nunca será necesario prestar crédito para completar la operación entonces el tiempo real de cualquier parte inicial de una secuencia de operaciones queda acotada por la suma de los correspondientes tiempos amortizados.

Si necesitamos hacer préstamos y tales préstamos pueden ser pagados al final de la secuencia, entonces el tiempo total de las operaciones queda acotado por la suma de todos los tiempos amortizados, aunque a la mitad de la secuencia podemos tener una deuda, es decir el tiempo real transcurrido puede exceder la suma de los tiempos amortizados por la cantidad real del préstamo neto[2].

Ahora bien, para guardar el estado de los créditos ahorrados o prestados generalmente es conveniente guardar estos en la estructura de datos. Los lugares de la estructura que contienen créditos son por lo general excepcionalmente difíciles de alcanzar (actualizar los créditos que están ahí ahorrados para pagar el trabajo adicional) o los lugares que contienen *débitos* los cuales son excepcionalmente fáciles de alcanzar o actualizar. Es importante aclarar que esto de los créditos y los débitos, solo es un dispositivo auxiliar de conteo.

Un ejemplo detallado donde se mostrará la aplicación de esta idea será visto en el Capítulo 5, cuando se realice el análisis amortizado del TDA *Conjuntos Ajenos*.

1.1.2 El enfoque de los Físicos sobre la Amortización.

En este enfoque se pretende medir la suceptibilidad del estado de un TDA para que las operaciones caras puedan ser medidas. Esto es, tratar de medir los cambios que sufre alguna característica del TDA a lo largo de la ejecución.

Para este caso definimos el **estado** de una implantación concreta de un TDA como su configuración en algún momento en el tiempo; esto se refiere a tamaño, forma o valores que posee. A cada posible estado S se le asocia un número real $\Phi(S)$, llamado **potencial** de S. Consideraremos que el estado más *susceptible* tendrá el potencial más alto.

[2]Actualmente, no se usan los prestamos al realizar el análisis amortizado, pues al interrumpir la ejecución se puede dar el caso de que el costo obtenido sea negativo.

Retomando el ejemplo del stack, que se mencionó antes, definimos al potencial para esta estructura como el número de datos que contiene —el tamaño del stack—, ya que un stack grande es más susceptible a operaciones POP caras que un stack pequeño.

Consideremos una secuencia de m llamadas al TDA. Sea t_i la complejidad de la i-ésima llamada. Definimos ahora a la **complejidad amortizada** a_i de la i-ésima llamada como: $a_i = t_i + \Phi(S_i) - \Phi(S_{i-1})$, donde S_{i-1} es el estado del TDA justo antes de la i-ésima llamada y S_i es el estado del TDA justo después que ha terminado. Asumimos que $\Phi(S_m) \geq \Phi(S_0)$, por lo cual tenemos que el tiempo total de las m llamadas resulta ser:

$$\begin{aligned}
\sum_{i=1}^{m} t_i &= \sum_{i=1}^{m} [a_i - \Phi(S_i) + \Phi(S_{i-1})] \\
&= (a_1 - \Phi(S_1) + \Phi(S_0)) + (a_2 - \Phi(S_2) + \Phi(S_1)) + \dots \\
&\quad \dots + (a_{m-1} - \Phi(S_{m-1}) + \Phi(S_{m-2})) + (a_m - \Phi(S_m) + \Phi(S_{m-1})) \\
&= a_1 + a_2 + \dots + a_m + \Phi(S_0) - \Phi(S_m) \\
&= \sum_{i=1}^{m} [a_i] - \Phi(S_m) + \Phi(S_0) \\
&\leq \sum_{i=1}^{m} a_i
\end{aligned}$$

De aquí se observa que la complejidad total amortizada es una cota superior para la complejidad total real ya que los potenciales se cancelan.

Consideremos entonces un TDA con operaciones $P_1, P_2, \dots P_v$; por convención la operación que crea o da valores iniciales a la estructura será P_1 y siempre será la primera en ser llamada. Consideremos además una secuencia de m llamadas a tales operaciones, $m = m_1 + m_2 + \dots + m_v$. Estas llamadas pueden estar arbitrariamente mezcladas.

Sea $W_{\Phi_j}(n)$ la complejidad máxima amortizada para un ejemplar de tamaño n, esto pensado en todos los ejemplares de tamaño n. Sea $W_j(n)$ el máximo sobre esos mismos ejemplares conforme a su complejidad real.

Ahora, si la i-ésima operación en la secuencia es P_j entonces $a_i \leq W_{\Phi_j}(n_{i-1})$, donde n_{i-1} es el tamaño real del TDA justo después de la i-ésima llamada; asumiendo que las $W_{\Phi_j}(n)$ son monótonamente no decrecientes tenemos que:

$$\begin{aligned}\sum_{i=1}^{m} t_i &\leq \sum_{i=1}^{m} a_i \\ &\leq \sum_{i=1}^{m} W_{\Phi_j}(n_{i-1}) \\ &\leq \sum_{i=1}^{m} W_{\Phi_j}(n) \\ &\leq m_1 W_{\Phi_1}(n) + \cdots + m_v W_{\Phi_v}(n)\end{aligned}$$

De este modo, se observa que las cotas amortizadas son tan buenas para acotar el costo total de una secuencia de operaciones como lo son las cotas en el análisis del peor de los casos. Estas cotas pueden ser más pequeñas si la función potencial Φ es cuidadosamente elegida. La secuencia de funciones $W_{\Phi_1}(n), W_{\Phi_2}(n), \ldots, W_{\Phi_v}(n)$ se llama **Complejidad Amortizada** de la implantación.

Cabe mencionar que establecer una función potencial $\Phi(S)$ es una cuestión de experiencia así como de ensayo y error. Otro hecho a tomar en cuenta es que el análisis amortizado derrama el costo de una operación que ocasionalmente es cara entre llamadas cercanas que sean baratas, ya sean llamadas anteriores o posteriores. Existe un truco muy útil para simplificar el análisis amortizado: *romper* las operaciones en estados más pequeños y analizarlos individualmente.

Justifiquemos esto: consideremos la i-ésima operación que tiene complejidad real t_i, complejidad amortizada a_i y lleva al TDA del estado S_{i-1} al estado S_i. Ahora suponga algunos estados intermedios alcanzados por el TDA en el curso de la operación que pueden ser identificados, los cuales serán denominados: $S'_1, S'_2, \ldots, S'_k$; sea $S_{i-1} = S'_1$ y $S_i = S'_k$. Sea t'_j el costo real transcurrido en la transición de S'_{j-1} a S'_j para $1 \leq j \leq k$; claramente la complejidad real de estas operaciones es la suma de esos números; así:

$$t_i = \sum_{i=1}^{k} t'_j$$

La complejidad amortizada en cada estado puede ser definida de manera natural,

$$a'_j = t'_j + \Phi(S'_i) + \Phi(S'_{j-1})$$

y de este modo al contar todos los estados intermedios, tenemos que:

$$\begin{aligned}\sum_{j=1}^{k} a'_j &= \sum_{j=1}^{k} t'_j + \Phi(S_j') - \Phi(S_{j-1}')] \\ &= \sum_{j=1}^{k} t'_j + \Phi(S'_k) + \Phi(S'_0) \\ &= t_i + \Phi(S_i) - \Phi(S_{i-1}) \\ &= a_i\end{aligned}$$

Capítulo 2

Una Primera Aplicación: *Colas Binomiales*

En este capítulo presentamos con detalle el análisis amortizado de la estructura de datos denominada Colas Binomiales, las cuales están englobadas en el TDA Colas de Prioridades. Empezamos definiendo las Colas Binomiales, sus operaciones y su complejidad en el peor de los casos, concluyendo con la realización del análisis amortizado de su complejidad.

Para trabajar de manera adecuada con Colas Binomiales introducimos primero el concepto de árboles binomiales. Los árboles Binomiales B_k donde $k \in \mathbf{N}$ se definen inductivamente de la siguiente manera:

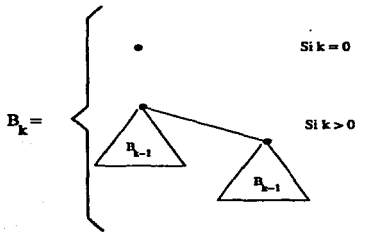


Figura 2.1: Definición de árboles Binomiales

- B_0 es un árbol que consiste de un solo nodo y
- cualquier otro árbol B_k se forma al ligar dos arboles B_{k-1}, esto es, colocando a la raíz de un como hijo de la raíz del otro. Esto se ilustra en la Figura 2.1

La Figura 2.2 ilustra los primeros cuatro árboles binomiales. A continuación se describen algunas propiedades de los árboles binomiales, las cuales son fáciles de justificar por la forma en que éstos se construyen.

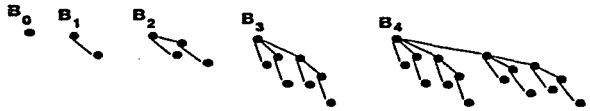


Figura 2.2: Ejemplo de árboles binomiales

Lema 2.1 *Para un árbol binomial B_k cualquiera se cumplen las siguientes propiedades:*

1. *B_k tiene 2^k nodos;*
2. *la altura del árbol es k;*
3. *hay exactamente $\binom{k}{i}$ nodos a la profundidad de i para $i = 0, 1, \ldots, k$,; y*
4. *la raíz tiene grado k, el cual es el mayor de cualquier otro grado de sus nodos; además si los hijos de la raíz son numerados de izquierda a derecha por $k-1, k-2, \ldots, 0$ hijos, entonces el nodo i es la raíz de un subárbol B_i.*

Demostración:

La demostración se hace por inducción sobre k. Para cada propiedad el caso base está dado por el árbol binomial B_0 y es claro que se cumple. Para el paso inductivo asumimos que el lema se cumple para el árbol B_{k-1}; con esta hipótesis tenemos que:

1. El árbol binomial B_k consiste de dos copias de B_{k-1} por lo cual tenemos $2^{k-1} + 2^{k-1} = 2^k$ nodos.
2. Dado que la construcción de un árbol binomial B_k es creado al ligar dos copias de B_{k-1} árboles binomiales en consecuencia B_k es una unidad más que la máxima profundidad en B_{k-1}. Por hipótesis de inducción, la máxima profundidad de B_{k-1} es $(k-1)$; por lo tanto la altura de B_k es $(k-1)+1 = k$.
3. Sea $D(k, i)$ el número de nodos en el nivel i del árbol binomial B_k. Puesto que B_k está compuesto de dos copias ligadas de B_{k-1} un nodo a profundidad i puede aparecer en B_k a profundidad i y también a profundidad $(i+1)$. En otras palabras el número de nodos que estén a profundidad i en B_k son aquellos nodos que estaban a profundidad i en una de las copias de B_{k-1} más los nodos que estaban a profundidad $(i-1)$ en la otra copia de B_{k-1}. En consecuencia tenemos que:

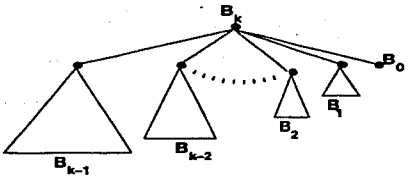


Figura 2.3: Construcción alternativa de los árboles Binomiales

$$\begin{aligned} D(k,i) &= D(k-i) + D(k-1, i-1) \\ &= \binom{k-1}{i} + \binom{k-1}{i-1} \\ &= \binom{k}{i} \end{aligned} \tag{2.1}$$

La igualdad (2.1) se obtiene de la hipótesis de inducción.

4. Observemos que de los nodos de B_k, el que posee el mayor rango sobre los rangos de los árboles B_{k-1} es la raíz cuya diferencia es de una unidad con respecto a la raíz de B_{k-1}. Ya que la raíz de B_{k-1} tiene rango $k-1$, entonces la raíz de B_k tiene grado k.

   Si los hijos de la raíz son numerados según sus rangos, de izquierda a derecha —ver la Figura 2.3—, por hipótesis de inducción cuando el árbol B_{k-1} es ligado a otro árbol B_{k-1} los hijos de la raíz resultante son raíces de los árboles $B_{k-1}, B_{k-2}, \ldots, B_0$.

Un árbol con llaves [1] en los nodos se dice que es un *heap ordenado* si la llave de cualquier hijo no es menor que la de su padre. Así en un heap ordenado la llave mínima está en la raíz del árbol. Con esto una *Cola Binomial* se define *como un bosque de árboles binomiales, donde cada árbol es un heap ordenado, que contiene a lo más un árbol de cada rango.*

Dada la restricción de que sólo puede haber un árbol de cada rango, tenemos que sólo hay una manera de tener n elementos en una cola binomial. Además, dado que el número de nodos en un árbol binomial B_i es exactamente 2^i, podemos pensar en una representación binaria posicional para las colas binomiales donde el dígito correspondiente a la i-ésima potencia va a ser 0 si B_i no está presente, y 1 si sí lo está. Veamos como ejemplo una cola binomial Q con $i = 13$ —esto significa que tiene 13 elementos— la cual está representada por el bosque B_3, B_2, B_0. Podemos también escribir esta representación

[1] El concepto de poner llaves en los nodos de un árbol se deriva de los TDA Colas de Prioridades, donde la llave asociada a cada nodo indica la prioridad del nodo en la cola

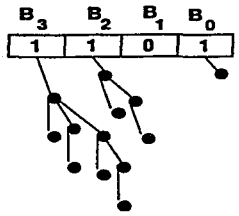


Figura 2.4: Representación binaria de una Cola Binomial de trece elementos

como 1101 la cual no solo representa el número 13 en binario sino que también representa el hecho de que B_3, B_2 y B_0 están presentes en la representación del bosque y B_1 no. Esto lo vemos gráficamente en la Figura 2.4.

2.1 Operaciones de las Colas Binomiales y su Complejidad.

Partiendo de la definición de Colas Binomiales, una condición que es importante preservar al manipularlas es que el bosque de árboles binomiales debe de contener a lo más un árbol de cada rango, para poder mantener esta condición se establece una operación básica denominada FUNDIR, y a partir de ella se definen las demás, como se explicará posteriormente.

Operaciones

Sean Q y T dos Colas Binomiales, sea x un dato con una *llave* —la llave debe ser un elemento de un conjunto con un orden lineal—. Así las operaciones quedan establecidas como:

- INSERTAR(x, Q): Agrega el dato x en la colección de datos de Q
- ELIM_MIN(Q): Localiza al dato que contiene la llave mínima en Q, regresa tal dato y después lo elimina de Q.
- FUNDIR(T, Q): Mezcla todos los datos de T con los datos de Q, dejándolos en Q.

Ahora describiremos las operaciones; para ello sea $\| P \|$ el número de elementos en una Cola Binomial P.

Operación FUNDIR(T, Q):

Mezcla los elementos de T en Q. Si $\| T \| = t$ y $\| Q \| = q$, entonces para el proceso de FUNDIR las Colas de Prioridades T y Q se realiza de manera análoga al proceso de sumar t y q en su representación binaria. Sucesivamente "sumaríamos" pares de árboles Binomiales B_k, esto se realiza simplemente ligando los árboles como se indica en la segunda regla de formación de los árboles binomiales de tal forma que el árbol de la raíz más grande se liga al árbol con la raíz más pequeña lo cual resulta en un árbol binomial B_{k+1}, es claro que esta operación lleva un tiempo de ejecución constante. De este modo, si hay presentes dos árboles B_0 se ligan —suman— para obtener un árbol B_1, que representa un acarreo. En un paso general observamos que a lo más se producen tres árboles B_k, uno en cada cola binomial y el otro es un acarreo; en el caso de que en una posición de la representación haya presente tres árboles, ligamos dos y el tercer árbol es movido a la posición $k+1$ volviéndose un acarreo temporal que eventualmente será acomodado. De la definición de Colas Binomiales se deduce que no pueden haber más de $\log \| Q \|$ árboles binomiales en una Cola Binomial. En consecuencia, en la operación FUNDIR no se realizan más de $\max\{\lfloor \log(t+1) \rfloor + \lfloor \log(q+1) \rfloor\}$ pasos. Por lo tanto, la operación completa requiere un tiempo de $O(\max\{\log \| T \|, \log \| Q \|\})$. De aquí, se deduce que el tiempo de ejecución para el peor caso es $O(\log n)$.

A continuación se explica detalladamente un ejemplo de la operación FUNDIR. En la Figura 2.5, se muestra la mezcla de dos colas binomiales T y Q, cuyos tamaños respectivamente son 3 y 7, esto se ve en el recuadro (a) de la figura. En el recuadro (b), tenemos el resultado de mezclar los árboles B_0, en consecuencia se presenta un acarreo. En el recuadro (c), se funden los árboles B_1 de las colas originales, lo cual produce un acarreo, y se coloca en su lugar el acarreo producido por la mezcla de los árboles B_0. En el recuadro (d) tenemos que la operación FUNDIR ha sido completada.

Operación INSERTAR(x, Q) :

Agrega el dato x a la cola Q. Para realizar esta operación: creamos la cola binomial X que contiene solamente al dato x, y ejecutamos FUNDIR(X, Q). Como consecuencia el tiempo de ejecución en el peor de los casos de una operación INSERTAR es $O(\log n)$, ya que es posible que se produsca un acarreo para todos los árboles binomiales.

Operación ELIM_MIN(Q):

El primer paso es localizar el nodo x que contenga la llave más pequeña. Dado que x es la raíz de alguno de los árboles Binomiales B_i —esto porque son heaps ordenados— el mínimo se puede localizar en un tiempo de $O(\log n)$ pues pueden haber a lo más $(\log \| Q \|)$ árboles binomiales en la cola Q.

El segundo paso consiste en la eliminación. Supongamos que el árbol binomial B_i contiene el mínimo elemento de Q, llamémosle x. Entonces el proceso de eliminación comienza por quitar el árbol B_i de Q, luego desmantelamos parcialmente a B_i esto es, le quitamos la raíz x, como resultado de esta operación obtenemos un árbol B_{i-1} e $i-2$ árboles B_j con $0 \leq j \leq i-2$, que acomodamos en una cola binomial temporal T cuyo

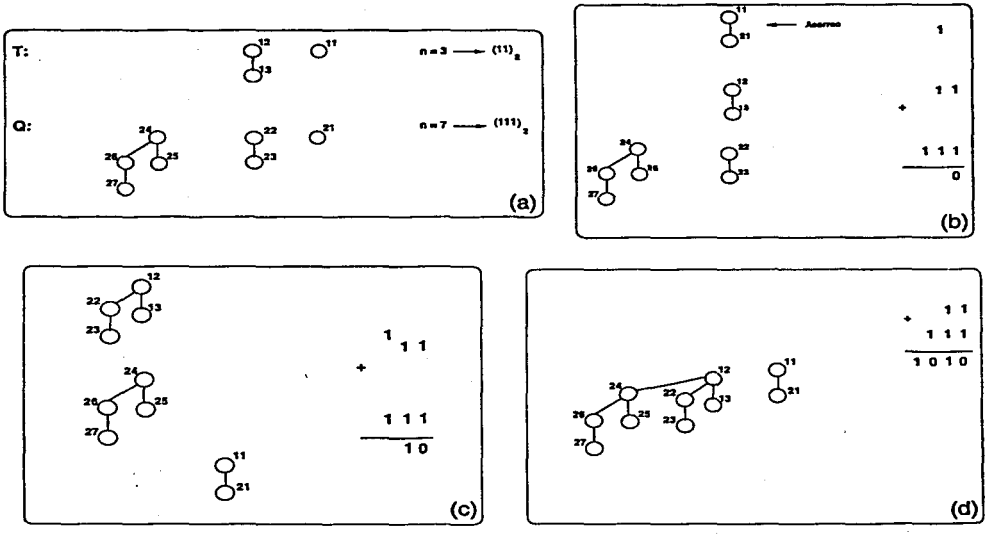


Figura 2.5: Aplicación de la operación FUNDIR.

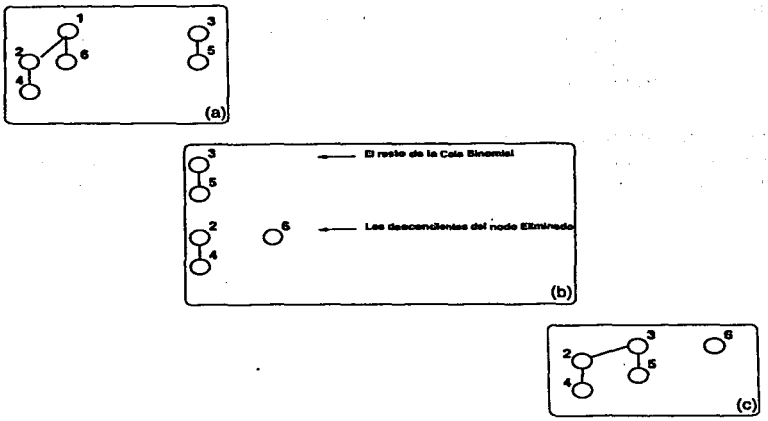


Figura 2.6: Una aplicación de la operación ELIM_MIN

tamaño es $2^i - 1$, esto se obtiene del apartado 4 del Lema 2.1. Ahora que x ha sido separado del árbol B_i, la operación ELIM_MIN puede regresar este valor.

El paso final consiste en mezclar la cola T formado en el paso dos y lo que queda de Q después de quitarle B_i, llamémosle Q'. Dado que el tamaño de cada cola binomial es menor que el tamaño de la cola Q , la operación FUNDE(T, Q') requiere entonces un tiempo de ejecución de $O(\log \| Q \|)$. Por lo tanto, la operación ELIM_MIN requiere un tiempo de ejecución en el peor caso de $O(\log n)$.

En la Figura 2.6 en el recuadro (a), se muestra una Cola Binomial de tamaño 6 a la que se le va a aplicar un ELIM_MIN. Al ser aplicada la operación ELIM_MIN, en el recuadro (b) tenemos ya dos Colas Binomiales como resultado de eliminar el nodo con la llave 1. Finalmente en el recuadro (c), obtenemos una Cola Binomial de tamaño 5 después de aplicar una operación FUNDIR.

2.2 Tiempo Amortizado de las Colas Binomiales

En esta sección determinaremos el tiempo de ejecución para las operaciones de las Colas Binomiales por medio del Análisis Amortizado. Comenzaremos por tratar de determinar cuanto tiempo se emplea en insertar n datos sucesivamente en una cola binomial.

Como un primer intento para poder determinar el costo de la operación INSERTAR, para el caso en que se inserten n datos sucesivos, podríamos hacer un cálculo directo;

para ello podemos definir el costo de una inserción como una unidad de tiempo más una unidad adicional por cada operación de mezclado. Sumando estos costos sobre todas las inserciones realizadas tendremos el total del tiempo de ejecución, que debe ser de n unidades más el número total de pasos de ligado.

Como se mencionó antes, la operación INSERTAR se puede ver como una suma de números binarios. Supongamos que insertamos quince elementos sucesivamente en una cola binaria inicialmente vacía, la ejecución puede ser vista como se muestra en la Figura 2.7.

No de Pasos	B_3	B_2	B_1	B_0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

Figura 2.7: Movimiento de los árboles Binomiales.

Ahora podemos observar lo siguiente:

- En los pasos 1, 3, 5 y todos los pasos con número impar no se requiere paso de ligado, puesto que no hay presente un árbol B_0 en el momento de la inserción, por lo tanto la mitad de las inserciones no requieren un paso de ligado.

- Por un razonamiento similar vemos que una cuarta parte de las inserciones requieren solo un paso de ligado (pasos $2, 6, 10, \ldots$)

- Un octavo de las inserciones requieren dos pasos de ligado

En general el número de ligados por realizar es:

$$\begin{aligned}\frac{n}{2}+\frac{n}{4}+\cdots+\frac{n}{2^n} &= \sum_{i=1}^{n}\frac{n}{2^i}\\ &\leq \sum_{i=1}^{\infty}\frac{n}{2^i}\\ &= n\sum_{i=1}^{\infty}\frac{1}{2^i}\\ &= n\end{aligned}$$

Este cálculo de fuerza bruta no nos ayuda cuando intentamos analizar una secuencia de operaciones que incluya no sólo la operación INSERTAR, así que podemos utilizar otro enfoque para demostrar esto.

Consideremos el resultado de una operación INSERTAR. Si no hay un árbol B_0 presente en el momento de insertar entonces la operación INSERTAR cuesta una unidad, utilizando la misma cuenta anterior. El resultado de esta inserción es que hay ahora un árbol B_0 y así tenemos que sumar un árbol al bosque de árboles binomiales.

Si hay un árbol B_0 presente pero no un árbol B_1 entonces la inserción tiene un costo de dos unidades. El nuevo bosque tendrá un árbol B_1 pero no un árbol B_0 así que el número de árboles en el bosque no cambia.

Una operación INSERTAR que cueste tres unidades creará un B_2 pero destruirá los árboles B_0 y B_1 produciendo una pérdida neta de un árbol en el bosque.

Siguiendo este razonamiento, en general podemos ver que una operación INSERTAR que cuesta c unidades resulta en un incremento neto de $(2-c)$ árboles en el bosque, puesto que cuando un árbol B_{c-1} es creado todos los árboles B_i con $1 \leq i < c-1$ son removidos. De todo esto, podemos concluir que las operaciones INSERTAR caras quitan árboles mientras que las baratas crean árboles.

Sea C_i el costo de la i-ésima inserción. Sea T_i el número de árboles después de la i-ésima inserción. Definamos a $T_0 = 0$ como el número de árboles iniciales. Entonces tenemos el invariante:

$$C_i + (T_i - T_{i-1}) = 2$$

Desglosando los n términos tenemos:

$$\begin{aligned} C_1 + (T_1 - T_0) &= 2 \\ C_2 + (T_2 - T_1) &= 2 \\ C_2 + (T_3 - T_2) &= 2 \\ &\vdots \\ C_{n-1} + (T_{n-1} - T_{n-2}) &= 2 \\ C_n + (T_n - T_{n-1}) &= 2 \end{aligned}$$

Sumando todas las ecuaciones, además de los términos T_i cancelados, tenemos que:

$$\sum_{i=1}^{n} C_i + T_n - T_0 = 2n$$

Como $T_0 = 0$ y T_n es el número de árboles después de n inserciones, ambos no negativos, se tiene que $(T_n - T_0)$ también es no negativo. Por lo tanto,

$$\sum_{i=1}^{n} C_i \leq 2n$$

Esto significa que realizar las n operaciones INSERTAR tiene un costo de $O(n)$. Dado este análisis podemos concluir el siguiente resultado:

Afirmación 2.1 *Una Cola Binomial de n elementos puede ser construido por n inserciones sucesivas en un tiempo de O(n).*

La última discusión que nos llevó a la Afirmación 2.1 es un ejemplo que ilustra la técnica general del análisis amortizado. En este caso para poder realizar el análisis amortizado se utilizó la técnica de hallar una función potencial, para lo cual el potencial de la estructura de datos es simplemente el número de árboles que se forman. Así que cuando teníamos inserciones que usaban solo una unidad en vez de dos unidades que son asignadas, la unidad extra que se genera es ahorrada para después ser manifestada en un incremento del potencial. Cuando en la operación sucede un exceso de tiempo al que le fue asignado entonces el exceso de tiempo causa un decremento del potencial.

Observamos que la función potencial debe:

- Siempre asumir un valor mínimo al inicio de la secuencia. Se sugiere elegir la función potencial tal que su valor inicial sea 0 y que siempre sea no negativa.
- Cancelar un término en el tiempo real. Por ejemplo en la discusión anterior, si el costo real es c entonces el cambio del potencial es de $(2 - c)$, cuando esto se suma obtenemos un costo amortizado de 2.

Concluiremos este capítulo con el siguiente resultado:

Teorema 2.1 *El tiempo amortizado de la ejecución de las operaciones* INSERTAR, ELIM_MIN, *y* FUNDIR *son* $O(1), O(\log n)$ *y* $O(\log n)$ *respectivamente en las Colas Binomiales.*

Demostración:

Definamos la función potencial como el número de árboles. El potencial inicial es 0 y el potencial es siempre no negativo, así que el tiempo amortizado es una cota superior sobre el tiempo real.

El análisis de la operación INSERTAR se sigue de la Afirmación 2.1.
Para la operación FUNDIR asumimos que se tienen n_1 y n_2 número de nodos para T_1 y T_2 árboles, respectivamente. Sea $n = n_1 + n_2$, el tiempo real para ejecutar la mezcla es entonces $O(\log n_1 + \log n_2)$ que es $\max\{O(\log n_1), O(\log n_2)\}$, como $n = n_1 + n_2$ entonces el $\max\{\log n_1, \log n_2\}$ puede ser acotado por $\log n$. Por lo tanto, $O(\log n_1 + \log n_2)$ es $O(\log n)$. Por lo cual después de la mezcla puede haber a lo más $\log n$ árboles entonces el potencial puede incrementarse a lo más $O(\log n)$. Esto da una cota amortizada de $O(\log n)$.

Para la operación ELIM_MIN, se elimina la raíz del árbol con la llave mínima, hacer esto no incrementa a la función potencial, lo que da por resultado dos colas binomiales las cuales son mezclados a través de la operación FUNDIR, por lo cual la cota amortizada para ELIM_MIN es también $O(\log n)$.

Capítulo 3

Buscando Potenciales: Skew Heap

El análisis de las Colas Binomiales es un ejemplo sencillo de una aplicación del análisis amortizado. Revisemos ahora el TDA Skew Heaps. Lo interesante de este ejemplo es que no hay un "orden claro" con respecto a su estructura y, sin embargo, la operación principal, que es MEZCLAR, tiene una cota amortizada de $O(\log n)$.

3.1 Definición de Skew Heaps

Los *Skew Heaps* son árboles binarios que guardan el orden de un *heap* pero sin inducir una estructura sobre el árbol. Los skew heaps, están también englobados en un TDA más general: las colas de prioridades. La operación principal de los skew heaps es la operación MEZCLAR. A partir de ella se definen las demás operaciones de la siguiente manera:

- INSERTAR(x): Crea un árbol de un nodo que contiene a x y mezcla este árbol con la cola de prioridades.
- ENCUENTRA_MIN(): Regresa el valor que hay en la raíz.
- ELIM_MIN(): Elimina la raíz y mezcla sus subárboles derecho e izquierdo.
- DECREMENTA_LLAVE(P, n_Val): Asumimos que P es un apuntador a un nodo dentro de la cola de prioridades, una vez localizado se reduce el valor de la llave de P, entonces se separa a P de su padre. Así obtenemos dos colas de prioridades que son mezcladas.

3.1.1 La operación MEZCLAR

Asumimos que tenemos dos árboles *heap ordenados* denominados como H_1 y H_2, los cuales necesitan ser mezclados. Es claro que si alguno de los árboles es vacío, el resultado de la mezcla es el árbol no vacío. En otro caso, mezclamos los dos árboles comparando sus

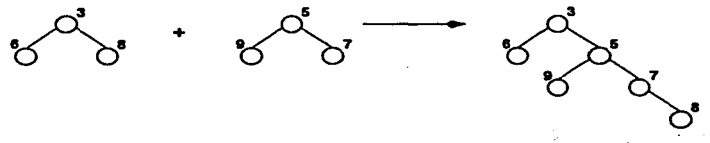


Figura 3.1: Mezcla de las trayectorias derechas en dos heaps ordenados.

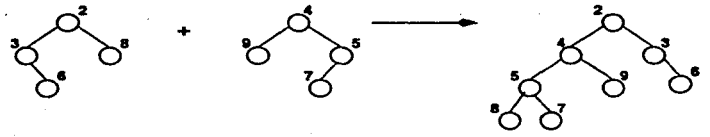


Figura 3.2: Resultado de la mezcla en la trayectoria izquierda en skew heaps.

raíces. De modo recursivo mezclamos los árboles, así la raíz más grande es acomodada dentro del subárbol derecho con raíz menor [1].

La Figura 3.1 muestra el efecto de esta primera estrategia recursiva, en la cual la trayectoria derecha de las dos colas de prioridades son mezcladas formando una nueva cola de prioridades. Notemos que los nodos sobre la trayectoria derecha conservan su subárbol izquierdo original y sólo los nodos sobre la trayectoria derecha son afectados. De aquí, se resalta el hecho de que las trayectorias izquierdas no sean modificadas derivando esto en que los árboles resultantes consistan sólo de trayectorias derechas largas y en consecuencia el total de las operaciones tomen tiempo lineal.

3.1.2 Una modificación a la operación Mezclar.

Una simple modificación a la técnica anterior sería: antes de completar una operación MEZCLAR, intercambiamos los hijos derecho e izquierdo en la trayectoria derecha para cada árbol temporal. En consecuencia, sólo aquellos nodos que estaban en la trayectoria derecha original y aquellos que se encontraran sobre la trayectoria derecha de un subárbol temporal formarán la trayectoria izquierda del árbol resultante - la Figura 3.2 muestra un ejemplo de estos intercambios. *Cuando se ejecuta una operación* MEZCLAR *de esta forma, el árbol heap ordenado es un Skew Heap.*

El resultado que se espera de este intercambio de hijos, es que la longitud de las trayectorias no sea demasiado larga en todo momento. Los Skew Heaps tienen la ventaja de que no es necesario un criterio adicional para mantener la longitud de sus trayectorias,

[1] Cualquiera de los dos subárboles pueden ser utilizados pero por simplicidad usamos el subárbol derecho

tampoco es necesario hacer alguna prueba para determinar cuándo se intercambian los hijos.

3.2 Análisis Amortizado de los Skew Heaps

Para establecer algunas ideas, supongamos que hay dos *heaps*, H_1 y H_2 y con r_1 y r_2 nodos respectivamente sobre sus trayectorias derechas. Con estas suposiciones, el tiempo de ejecución de la operación MEZCLAR es proporcional a $(r_1 + r_2)$. Cuando cargamos una unidad para cada nodo sobre las trayectorias derechas, el costo de MEZCLAR es proporcional al número de cargas. Dado que los árboles no tienen estructura, es posible que todos los nodos en ambos árboles estén repartidos sobre las trayectorias derechas y proporcione una cota de $\Theta(n)$ en el peor caso para la operación MEZCLAR. Se mostrará que el tiempo amortizado para MEZCLAR dos Skew Heaps es $O(\log n)$.

Como en el caso de las *Colas Binomiales* la prueba se realizará introduciendo una función potencial que compense las variaciones de las operaciones en los *heaps*. Necesitamos algún tipo de función potencial que capture el *efecto* de las operaciones de los Skew Heaps. El efecto de la operación MEZCLAR es que cada nodo sobre la trayectoria derecha sea movido hacia la trayectoria izquierda y los que eran sus hijos izquierdos se vuelven hijos derechos. Una idea puede ser clasificar cada nodo como nodo derecho o nodo izquierdo, dependiendo si es o no, un hijo derecho y utilizando el número de nodos derechos como función potencial. Aunque inicialmente el potencial es 0 y siempre sería no negativo, el problema de este potencial es que no se decrementa después de una mezcla y no se reflejaría adecuadamente lo ahorrado en la estructura de datos. Por lo tanto, esta función potencial no puede ser utilizada para probar la cota deseada.

Una idea similar es clasificar los nodos entre *pesados* o *ligeros* dependiendo si el subárbol derecho de cualquier nodo tiene o no más nodos que el subárbol izquierdo.

Definición 3.1 *Un nodo p es* pesado *si el número de descendientes del subárbol derecho de p es al menos la mitad del número de descendientes de p, en otro caso se dice que el nodo es ligero. Notemos que el número de descendientes de un nodo incluye al nodo mismo.*

La Figura 3.3 muestra un skew heap. Los nodos con llaves 15, 3, 6, 12 y 7 son pesados y los restantes son ligeros.

La función potencial que se utilizará entonces es el número de nodos pesados en la colección de heaps. Observamos que es una buena elección debido a que la longitud de la trayectoria derecha contendrá un número desordenado de nodos pesados, porque los nodos sobre las trayectorias tienen sus hijos intercambiados, los cuales se convertirán a nodos ligeros como resultado de la mezcla.

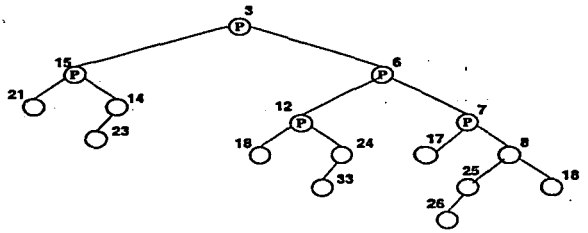


Figura 3.3: Ejemplo de la clasificación de nodos pesados y ligero en un Skew Heap

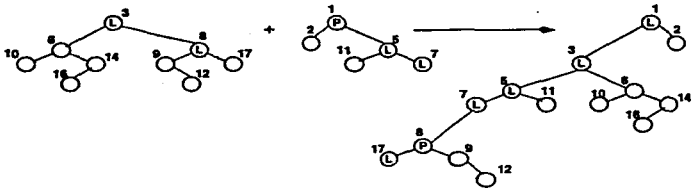


Figura 3.4: Cambio de estado pesado/ligero después de realizar una mezcla

Teorema 3.1 *El tiempo amortizado de mezclar dos Skew Heaps es* $O(\log n)$

Demostración:

Sean H_1 y H_2 dos *heaps ordenados* con n_1 y n_2 nodos respectivamente y sea n su combinación de tamaños esto es $n = n_1+n_2$. Supongamos que la trayectoria derecha de H_1 tiene l_1 nodos ligeros y h_1 nodos pesados para un total de l_1+h_1 nodos. También que H_2 tiene l_2 nodos ligeros y h_2 nodos pesados sobre su trayectoria derecha para un total de l_2+h_2 nodos. Si adoptamos la convención de que el costo de la mezcla de dos skew heaps es el número total de nodos sobre sus trayectorias derechas entonces el tiempo real de la ejecución de la operación MEZCLAR es $l_1+l_2+h_1+h_2$. Ahora bien, sólo los nodos cuyo estado de pesado/ligero puede ser cambiado, son los nodos que están inicialmente sobre la trayectoria derecha de los árboles y serán llevados a la trayectoria izquierda, sin que a los otros nodos les sean alterados sus subárboles. Esto se muestra en la Figura 3.4. Por ejemplo, el nodo 1 cambia su estado de pesado a ligero, también el nodo 8 cambia su estado de ligero a pesado, después de aplicarles la operación MEZCLAR.

De este modo, un nodo pesado que está inicialmente sobre la trayectoria

derecha, después de una mezcla, se vuelve ligero. Los otros nodos que estaban en la trayectoria derecha y que eran ligeros pueden o no seguir siendo ligeros. Con base en esta observación demostraremos la cota superior, de esta manera asumiremos el peor caso, donde todos los nodos se vuelvan pesados después de la mezcla y, por ende se incremente el potencial. Así tenemos que el cambio neto en el número de nodos pesados es a lo más:

$$l_1 + l_2 - (h_1 + h_2).$$

Sumando el tiempo real y el cambio del potencial obtenemos una cota amortizada de:

$$2(l_1 + l_2).$$

Por demostrar que $l_1 + l_2$ es $O(\log n)$. Dado que l_1 y l_2 representan el número de nodos ligeros sobre la trayectoria derecha original y que el subárbol derecho de un nodo ligero es menor que la mitad del tamaño del árbol que arraiga el nodo ligero, el principio de bisección se aplica, con lo cual el número de nodos ligeros en los árboles es a lo más $\lfloor \log n_1 \rfloor + 1$ y $\lfloor \log n_2 \rfloor + 1$ —se suma uno pues las hojas son ligeras por definición.

Con lo cual tenemos que $l_1 \leq \lfloor \log n_1 \rfloor + 1$ y $l_2 \leq \lfloor \log n_2 \rfloor + 1$. De donde obtenemos la siguiente desigualdad:

$$l_1 + l_2 \leq \log n_1 + \log n_2 + 2 \leq 2 \log n$$

De la desigualdad anterior concluimos que el costo de mezclar dos Skew Heaps, es a lo más

$$2 \log n + (h_1 + h_2)$$

y el cambio en el potencial es a lo más

$$2 \log n - (h_1 + h_2).$$

Teorema 3.2 *El costo amortizado de los Skew Heaps es a lo más* $4 \log n$ *para las operaciones* MEZCLAR, INSERTAR *y* ELIM_MIN. *Además, el tiempo de ejecución total esperado para una secuencia de* m *operaciones es de* $O(m \log n)$.

Demostración:

Sea Φ_i el potencial de la colección de skew heaps seguida inmediatamente de la i-ésima operación.

Notemos que $\Phi_0 = 0$, pues al iniciar la ejecución la colección es vacía y que $\Phi_i > 0$. Una operación INSERTAR crea un árbol de un solo nodo cuya raíz es por definición ligera, así que éste no afecta el valor del potencial cuando se realiza una mezcla. La operación ELIM_MIN primero elimina la raíz del árbol con lo cual obtenemos dos subárboles que serán mezclados; este hecho

no incrementa el potencial. Así que necesitamos considerar el costo de la operación MEZCLAR.

Sabemos que para obtener el tiempo amortizado de una operación, es necesario sumar el tiempo real de la operación más el cambio del potencial. Si H_1 y H_2 son dos heaps ordenados tales que H_1 sobre su trayectoria derecha tiene l_1 nodos ligeros y h_1 nodos pesados y, análogamente H_2 tiene l_2 nodos ligeros y h_2 nodos pesados sobre su trayectoria derecha, entonces, para mezclar a H_1 con H_2 según el Teorema 3.1, la operación MEZCLAR tarda a lo más $2\log n + (h_1 + h_2)$ y el cambio en el potencial es de a lo más $2\log n - (h_1 + h_2)$, con lo cual tenemos:

$$(2\log n + (h_1 + h_2)) + (2\log n - (h_1 + h_2)) = 4\log n = O(\log n).$$

Con lo que queda establecido el costo amortizado de las operaciones MEZCLAR, INSERTAR y ELIM_MIN.

Sea c_i el costo de aplicar la operación MEZCLAR, el cual ocurre como resultado de la i-ésima operación; entonces tenemos que:

$$c_i + \Phi_i - \Phi_{i-1} \leq 4\log n.$$

Calculando el costo sobre m operaciones, tenemos:

$$\sum_{i=1}^{m}(c_i + \Phi_i + \Phi_{i-1}) \leq 4m\log n,$$

simplificando,

$$\sum_{i=1}^{m} c_i \leq 4m\log n = O(m\log n)$$

puesto que $\Phi_m - \Phi_0$ es no negativo, con lo que se concluye la demostración.

Los skew heaps son un ejemplo típico de un algoritmo simple con un análisis que resulta no ser obvio. Claro está que el análisis se vuelve sencillo una vez que se encuentra una función potencial adecuada. Lamentablemente no hay un teoría general que permita definir cuál función potencial sea la adecuada.

Capítulo 4

El TDA Conjuntos Ajenos: Planteamiento

En este Capítulo se definen y prueban las propiedades básicas del tipo de datos abstracto Conjuntos Ajenos, en inglés *Disjoin Sets*; se define el algoritmo para Unión de Conjuntos, el cual es conocido como SET-UNION.

4.1 Definición del TDA Conjuntos Ajenos

El tipo de datos abstractos Conjuntos Ajenos mantiene una colección S de conjuntos que son ajenos y dinámicos, $S = S_0, S_1, \ldots, S_k$. Cada elemento de un conjunto es representado por un objeto, digamos x, que no contiene información, excepto su nombre y el conjunto al que pertenece; a este representante se le denomina *elemento canónico.*

El algoritmo para Unión de Conjuntos, SET-UNION, resuelve el problema de mantener una colección de conjuntos ajenos, el problema consiste en ejecutar tres clases de operaciones:

MAKESET : crear un conjunto nuevo;

FINDSET : localizar al conjunto que contiene al elemento dado;

LINK : combina dos conjuntos en uno. (Une conjuntos)

Precisando estas operaciones tenemos:

- CREATE() : Crea una colección vacía de conjuntos ajenos.
- MAKESET(x) : Crea un nuevo conjunto que contiene como único elemento a x, el cual no está contenido en ningún otro conjunto e integra este nuevo conjunto a la colección de conjuntos ajenos.
- FINDSET(x) : Regresa el elemento canónico del conjunto que contiene a x.

- LINK(x, y) : Forma un nuevo conjunto que es la unión de dos conjuntos cuyos elementos canónicos son x y y. Destruye los dos antiguos conjuntos. Selecciona y regresa al elemento canónico del nuevo conjunto. Se asume que $x \neq y$

A continuación se muestra un ejemplo de como se manipulan los conjuntos con estas operaciones:

$$\begin{aligned}
Create &\longrightarrow D = \{\} \\
D.MakeSet(a) &\longrightarrow D = \{\{a\}\} \\
D.MakeSet(b) &\longrightarrow D = \{\{a\},\{b\}\} \\
D.MakeSet(c) &\longrightarrow D = \{\{a\},\{b\},\{c\}\} \\
D.MakeSet(d) &\longrightarrow D = \{\{a\},\{b\},\{c\},\{d\}\} \\
D.FindSet(c) &\longrightarrow c \\
D.Link(a,c) &\longrightarrow D = \{\{a,c\},\{b\},\{d\}\} \\
D.Link(a,d) &\longrightarrow D = \{a,c,d\},\{b\}\} \\
D.MakeSet(f) &\longrightarrow D = \{\{a,c,d\},\{b\},\{f\}\}
\end{aligned}$$

Observemos que en este punto $D.FindSet(c) = D.FindSet(a) = D.FindSet(d) = a$

Este tipo de datos puede usarse cuando los elementos están inicialmente separados y gradualmente se van uniendo, como en el algoritmo de Kruskal para árboles generadores de peso mínimo. En este algoritmo se mantiene un bosque de subárboles de peso mínimo y se van integrando hasta crear el árbol generador de peso mínimo. Aquí cada subárbol o componente conexa, es representado por un conjunto ajeno.

El tipo de datos abstracto Conjuntos Ajenos está vinculado con el concepto de *Relación de Equivalencia.* En este caso los conjuntos son las clases de equivalencia de alguna relación de equivalencia $\mathcal{R}$. La operación $FindSet(x)$ determina a cual clase de equivalencia pertenece x; así la pregunta $FindSet(x) = FindSet(y)$ determina si $(x, y) \in \mathcal{R}$ y la operación $Link(x, y)$ determina que, de ahora en adelante, los elementos x y y son equivalentes, lo que significa que pertenecen a la misma clase y sus clases deben ser unidas. Tomando nuevamente el algoritmo de Kruskal, dos vértices están relacionados si pertenecen a la misma componente conexa o subárbol y sus clases deben ser unidas.

4.2 Estructuras de Datos para Conjuntos Ajenos

Hasta ahora sólo se ha definido cual es el TDA Conjuntos Ajenos; en esta sección presentamos algunas Estructuras de Datos para este TDA.

Podemos pensar en representar a los Conjuntos Ajenos como una secuencia de datos en los cuales guardamos un valor entero para denotar a qué conjunto pertenece cada uno. Por ejemplo la estructura $D = \{\{a, b, c, d\}, \{e, f\}, \{g\}\}$ puede ser representado como en la Figura 4.1. De este modo, al ejecutar un FINDSET(e) simplemente regresaría un 2 y

Figura 4.1: Representación elemental de los Conjuntos Ajenos

esta operación tendría en general un costo $O(1)$. Desafortunadamente, para la operación LINK tendríamos que cambiar todos los valores de un conjunto a otro, lo cual en el peor caso es $O(n)$.

4.2.1 Representación Galler-Fischer

En esta estructura cada conjunto se representa como un árbol enraizado. Los nodos del árbol son los elementos del conjunto, el elemento representante o canónico es la raíz. Los apuntadores van de los hijos a los padres. Así cada nodo tiene un apuntador a su padre, denotado como $p(x)$; en cada árbol la raíz apunta a sí misma. Con esta representación las operaciones quedan determinadas de la siguiente manera:

- Al ejecutar la operación MAKESET(x) se define a $p(x)$ como x.
- Para ejecutar la operación FINDSET(x), se siguen a los apuntadores padre, desde x hasta la raíz del árbol que contiene a x y regresa la raíz.
- En la operación LINK(x, y), se define a la raíz y como el padre de x, $p(x) := y$ y regresa a y como elemento canónico del nuevo conjunto.

La Figura 4.2 ilustra un ejemplo de la representación de Galler-Fisher para los conjuntos $\{s, b, p, d, q, f, g\}, \{x, y, z, w\}$ y $\{k\}$.

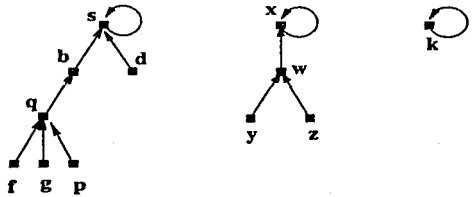


Figura 4.2: Ejemplo de la representación Galler_Fischer

Complejidad.

Las operaciones CREATE, MAKESET y LINK tienen complejidad constante. Para la operación FINDSET, el número de apuntadores recorridos desde x hasta la raíz, será su medida de complejidad. Así el costo de FINDSET es $d(x)$, la profundidad de x en su árbol.

Si la altura de un bosque se define como la altura del árbol más alto, entonces en el peor de los casos, la complejidad de FINDSET(x), cuando x está en el bosque F es $h(F) - 1$. Esto llega a ser n, cuando F consiste de un único árbol que resulta ser una trayectoria.

Notación

Para el análisis de esta estructura de datos y sus variantes definiremos algunas variables con las que estaremos trabajando. Denotaremos con n al número de operaciones MAKESET que se realicen durante la ejecución; con m al número de operaciones FINDSET y con k al número de operaciones LINK. Se asumirá además que $n/2 \leq k \leq n-1$ y que $m \geq n$.

Con lo anterior vemos que la versión simple del elemento canónico hace que para responder quién es el padre de un elemento cualquiera, el algoritmo gaste mucho de su tiempo siguiendo a los apuntadores padre. Cada operación MAKESET requiere tiempo $O(1)$, al igual que cada operación LINK. Una operación FINDSET toma tiempo proporcional al número de nodos sobre la trayectoria encontrada, que es a lo más n. Así el tiempo total es $O(n + mn)$.

Observemos que al aplicar $(n-1)$ operaciones LINK se puede construir un árbol con una trayectoria de n nodos, si se ejecuta repetidamente la operación FINDSET sobre la trayectoria se gasta un tiempo total de $\Omega(n + mn)$. Esta ingenua versión también es conocida como **unión simple**. Se concluye con el siguiente resultado:

Teorema 4.1 *El Algoritmo* SET-UNION *usando Unión Simple requiere tiempo* $\Theta(n+mn)$ *en el peor caso.*

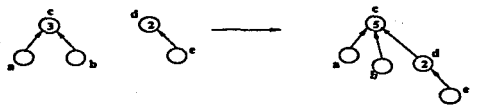
Cambiar la estructura de datos para manipular los árboles, de tal forma que se reduzca la longitud de las trayectorias encontradas por la operación FINDSET, acelera considerablemente el desempeño computacional del algoritmo. En las siguientes secciones veremos ejemplos específicos de esto.

4.3 La Estrategia Unión por Tamaño

El camino obvio para mejorar la ejecución de la estructura de datos Galler-Fisher es asegurando que los árboles estén balanceados. Hay una forma sencilla de hacerlo, conocida como **Unión por Tamaño** o *Union by Size*, su nombre en inglés.

La idea es muy sencilla: se mantiene un registro adicional para almacenar el tamaño de cada árbol —el número de nodos que contiene el árbol— en la raíz, y cuando se realice una operación LINK, el árbol de menor tamaño será ligado al árbol de mayor tamaño; si tienen el mismo tamaño se toma indistintamente la nueva raíz. A continuación un ejemplo, en el cual la etiqueta en la raíz de cada árbol representa el número de nodos en el árbol:

A continuación enunciamos algunos resultados sobre la operación LINK utilizando unión por tamaño. El tamaño de un nodo x, $s(x)$, se define como el número de nodos del árbol enraizado en x. También denotamos al árbol enraizado en x como T_x y su altura por $h(T_x)$.



Teorema 4.2 *Considere cualquier secuencia P de operaciones para conjuntos ajenos, $P = p_0, p_1, \ldots, p_m$, tal que p_0 es la operación* CREATE, *y las otras son* MAKESET, FIND-SET *y* LINK, *utilizando la estrategia* Unión por Tamaño. *Si x es cualquier nodo existente después de aplicar la operación p_m, entonces:*

$$2^{h(T_x)-1} \leq s(T_x)$$

Demostración:

La demostración se realiza por inducción sobre m, la longitud de la secuencia de operaciones.

Caso Base: $m = 0$. Se tiene que la estructura es vacía, así que es cierto por vacuidad.

Hipótesis de Inducción: antes que inicie la operación p_m el estado de la estructura cumple que: $2^{h(T_x)-1} \leq s(T_x), \forall x$.

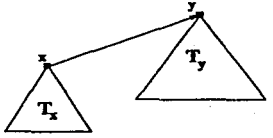
Paso inductivo: $m > 0$. Por demostrar que esto continua siendo cierto después que la operación p_m se ejecuta.

Si p_m es la operación MAKESET(x) se introduce un nuevo árbol T_x de tamaño $s(T_x) = 1$ y altura $h(T_x) = 1$, lo cual satisface la condición.

Si p_m es la operación FIND(x), la estructura no cambia.

Supóngase que p_m es la operación LINK(x, y); sin pérdida de generalidad se asume que $s(T_x) \leq s(T_y)$, entonces x es ligado a y, creando un nuevo árbol T:

T=



Claramente, $h(T) = \max\{1 + h(T_x), h(T_y)\}$ y de aquí $2^{h(T)-1} \leq s(T)$, ya que

$$\begin{aligned} 2^{h(T)-1} &= 2^{\max\{1+h(T_x),h(T_y)\}-1} \\ &= \max\{2^{h(T_x)-1+1}, 2^{h(T_y)-1}\} \\ &\leq \max\{2s(T_x), s(T_y)\} \end{aligned}$$

Ahora definimos el rango de un nodo x, $r(x)$, como la altura de x en el árbol que lo contiene.

Corolario 4.1 *En un bosque construido a partir de nodos simples, por una secuencia de operaciones* LINK, *usando* Unión por Tamaño, *se tiene que el número de nodos de rango i es a lo más $n/2^i$.*

Demostración:

Los rangos se incrementan, estrictamente, a lo largo de cualquier trayectoria en el bosque. De aquí, cualesquiera dos nodos del mismo rango no están relacionados; es decir, se encuentran en conjuntos ajenos. Por lo demostrado en el Teorema 4.2, todo nodo de rango i tiene a lo más 2^i descendientes. Entonces, hay a lo más $n/2^i$ nodos de rango i.

Corolario 4.2 *En un bosque generado, a partir de nodos simples, por una secuencia de operaciones* LINK, *usando* Unión por Tamaño, *se tiene que ninguna trayectoria tiene longitud mayor que* $\log_2 n$.

Demostración:

Partiendo del resultado del Teorema 4.2, tenemos que para cualquier árbol T en el bosque, se cumple $h(T) - 1 \leq \log_2 s(T)$; como por definición $s(T) \leq n$, se sigue que $h(T) - 1 \leq \log_2 n$. Por lo tanto ninguna trayectoria tiene longitud mayor que $\log_2 n$.

Para concluir esta sección observemos que la implantación de tipo de dato abstracto Conjuntos Ajenos usando árboles y ligándolos con la estrategia unión por tamaño tiene una complejidad en el peor caso de $O(1)$ para las operaciones CREATE, MAKESET y LINK, y complejidad de $O(\log n)$ para FINDSET.

4.4 Unión por rango

La heurística llamada Unión por Rango, *Union by Rank*, conserva árboles poco profundos usando una implícita libertad en la implantación de la operación LINK. Con cada nodo x se almacena un entero no negativo denominado **rango**, y denotado por $r(x)$, que es una cota superior sobre la altura de x. Cuando se ejecuta una operación MAKESET, se define $r(x)$ como cero.

Al efectuar una operación LINK(x, y) comparamos el rango de x con el de y. Se tienen las siguientes reglas:

- Si $r(x) < r(y)$ entonces x apunta a y, y es el elemento canónico.
- Si $r(x) > r(y)$ entonces y apunta a x, x se vuelve el elemento canónico

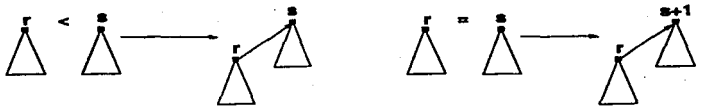


Figura 4.3: Construcción de los Conjuntos Ajenos con de Unión por Rango

- Si $r(x) = r(y)$ entonces x apunta a y, y es el elemento canónico y se incrementa en 1 el rango de y.

La Figura 4.3 ilustra el proceso. Esta estrategia fue inventada por Tarjan [6] y resulta ser una variante de unión por tamaño. Esto se observa en los siguientes resultados.

Lema 4.1 *El número de nodos en un árbol con raíz x es a lo más $2^{r(x)}$.*

Demostración:

La demostración es por inducción sobre el valor de $r(x)$.
Caso Base: Son creados los árboles así que $r(x) = 0$ y por lo tanto $2^0 = 1$ que es el número de nodos.

Hipótesis de Inducción: Para cada árbol en el bosque de Conjuntos Ajenos generado a partir de nodos simples, se cumple que el número de descendientes es de a lo más $2^{r(y)-1}$ siendo y la raíz de cualquier árbol en ese bosque.

Paso Inductivo: Sea T el árbol con raíz en x, tal que el rango $r(x) = k > 0$ y además T es el árbol con el menor número de descendientes.

Ahora supongamos que en la última operación LINK el árbol T fue creado a partir de los árboles T_1 y T_2 y que la raíz de T_1 era x.

Si T_1 tenía rango $r(x) = k$ entonces T_1 sería un árbol de rango k con menos descendientes que T lo cual contradice la suposición de que T es el árbol con el menor número de descendientes.

Por lo tanto, el rango de T_1 es a lo más $k - 1$, en consecuencia el rango de T_2 es a lo más el rango de T_1, por las reglas de *Unión por Rango*. Para asegurar el valor de los rango de T_1 y T_2, dado que T tiene rango k y el rango puede ser incrementado únicamente a causa de T_2, se sigue que $r(T_2) = k - 1$ entonces también $r(T_1) = k - 1$. Por hipótesis de inducción, cada árbol tiene a lo más 2^{k-1} descendientes dando un total de 2^k descendientes esto es $2^{r(x)}$ con lo que se prueba el Lema.

Definiendo el tamaño del árbol con raíz en x, $s(x)$, como en la sección anterior tenemos:

Lema 4.2 *Si x es cualquier nodo en un bosque construido, a partir de nodos simples, por una secuencia de operaciones* LINK, *usando* unión por rango, *entonces* $s(x) \geq 2^{r(x)}$.

Demostración:

Por inducción sobre el número de operaciones LINK. Supongamos que el lema es verdad justo antes de aplicar la operación LINK(e, f). Sean s_a y r_a los valores de las funciones tamaño y rango respectivamente, antes del LINK, y sean r' y s' los valores de las funciones después de la operación LINK. Sea x cualquier nodo.

Hay dos casos:

1. Si $x \neq e$ o $r_a(e) \neq r_a(f)$ entonces,

$$s'(x) \geq s_a(x) \text{ y } r'(x) = r_a(x),$$

lo cual significa que el lema es verdadero después de aplicar el LINK.

2. Si $x = e$ y $r_a(e) = r_a(f)$ entonces

$$s'(x) = s(e) = s_a(e) + s_a(f) \geq 2^{r_a(e)} + 2^{r_a(f)} = 2^{r_a(e)+1} = 2^{r'(e)} = 2^{r'(x)}$$

y por lo tanto el lema es verdadero después del LINK.

De este último lema observamos que las técnicas de Unión por Tamaño y Unión por Rango son equivalentes. En consecuencia tenemos el siguiente corolario que habla sobre la cantidad de subárboles que hay en el bosque de conjuntos ajenos de rango k.

Corolario 4.3 *El número de nodos de rango k es a lo más $n/2^k$.*

Demostración:

Cada nodo de rango k es la raíz de un subárbol de a lo más 2^k nodos, sin que otro nodo en el subárbol pueda tener rango k. De este modo, todos los subárboles con nodos de rango k son ajenos. Por lo tanto, hay a lo más $n/2^k$ subárboles ajenos y por lo cual $n/2^k$ nodos de rango k.

Concluimos calculando el peor caso de la técnica Union por Rango. Las operaciones LINK y MAKESET requieren tiempo constante. Una operación FINDSET toma tiempo $O(\log n)$. Por Lema el 4.1, ningún nodo tiene rango mayor de $\log_2 n$. Así se obtiene que $O(m \log n)$ es una cota sobre el tiempo requerido para una secuencia de m operaciones. Hemos demostrado el siguiente resultado:

Teorema 4.3 *El Algoritmo* SET-UNION *usando Unión por Rango requiere de un tiempo de ejecución en el peor caso de $O(m \log n)$.*

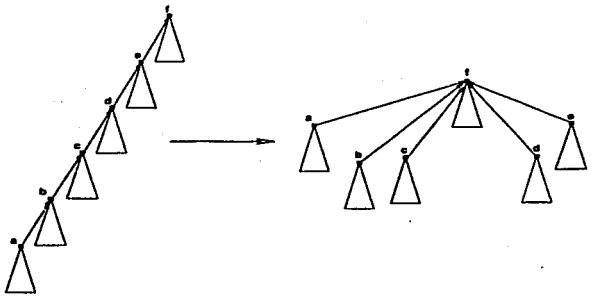


Figura 4.4: Ejemplo de una Compresión de Trayectorias.

4.5 Reducción de Trayectorias

Es posible mejorar el desempeño computacional del algoritmo SET-UNION con algunas heurísticas que reducen las trayectorias generadas por la operación LINK a continuación describiremos las más importantes.

4.5.1 Compresión de Trayectorias

La compresión de trayectorias – *Path Compression*, en inglés– hace que cambie la estructura del árbol durante la ejecución de una operación FINDSET moviendo los nodos lo más cerca de la raíz. Cuando se lleva a cabo una operación FINDSET(x), después de localizar a la raíz r del árbol que contiene a x, se obliga a que cada nodo sobre la ruta de x a r apunte directamente a la raíz. La Figura 4.4 ilustra como se compacta la trayectoria $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f$; los triángulos denotan subárboles.

La compresión de trayectorias tiene la desventaja de que requiere dos pasadas sobre la trayectoria encontrada. Van Leeuwen y Van der Weide [6] proponen dos variantes de compresión de trayectorias que requieren sólo una pasada., la primera es llamada **Partición** (*Path Splitting*), y la segunda es **Bisección** (*Path Halving*). Se presenta otra estrategia dada por Tarjan [6], denominada **Compactación.**

4.5.2 Partición de Trayectorias.

Durante la operación FINDSET, se obliga a que cada nodo a lo largo de la trayectoria encontrada, con excepción del último y el penúltimo nodo, apunte al nodo que se encuentra dos nodos después de él sobre la trayectoria. La Figura 4.5 nos ilustra este proceso.

El método Partición rompe una trayectoria encontrada en dos trayectorias, cada una de ellas con longitud casi la mitad de la original.

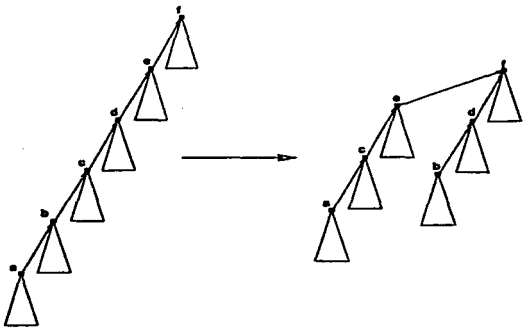


Figura 4.5: Método de Partición de Trayectorias.

4.5.3 Bisección de Trayectorias

Durante una operación FINDSET, se obliga a que cada nodo, **alternadamente en la trayectoria encontrada**, excepto el último y penúltimo nodo, apunte al nodo que se encuentra a dos de él. Se muestra un ejemplo en la Figura 4.6.

La trayectoria más larga en el árbol comprimido tiene longitud de a lo más la mitad de la longitud de la trayectoria original. El método de Bisección requiere solo la mitad de los apuntadores que los que se actualizan en Partición. Tiene la ventaja de guardar los nodos sobre la trayectoria encontrada mientras bisecta la longitud de la trayectoria, así que después de aplicar varias operaciones FINDSET se genera más compresión.

4.5.4 Compactación de Trayectorias

Considérese cualquier trayectoria. Se ejecuta una *Compactación* sobre la trayectoria, haciendo que cada nodo en la trayectoria, excepto el último y el penúltimo nodo, apunte un nodo que esté al menos a distancia dos de él.

La Figura 4.7 ilustra un ejemplo de esta técnica. Aquí el nodo a, después de la compactación, apunta al nodo e que estaba a cuatro nodos de distancia en la trayectoria.

El método de compresión es, localmente, la mejor forma de compactación, ya que mueve a los nodos tan cerca de la raíz, como sea posible, mientras el de partición es el peor, localmente. El método de bisección, por su parte, no es una forma de compactación, pero con unos cambios mínimos es posible obtener una buena cota de él.

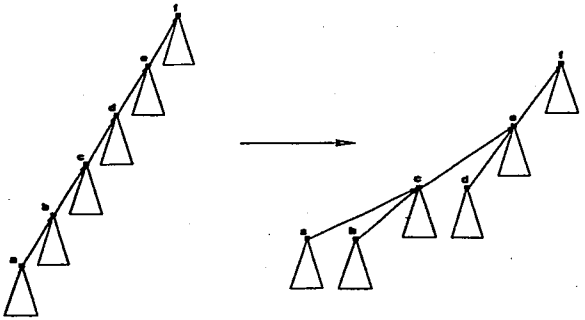


Figura 4.6: Bisección de Trayectorias.

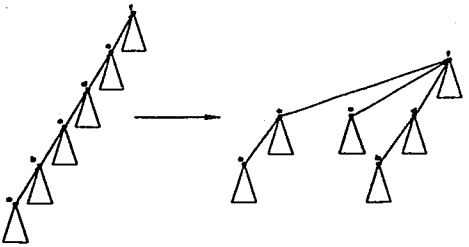


Figura 4.7: Compactación de Trayectorias (El método de Tarjan).

4.6 Dos resultados más

En esta sección revisaremos dos Teoremas que nos ayudaran a demostrar algunos resultados del siguiente capítulo, uno de los cuales involucra a las heurísticas para la reducción de trayectorias.

Teorema 4.4 *En cualquier trayectoria generada a partir del algoritmo* SET-UNION *los rangos de los nodos sobre una trayectoria, desde una hoja a la raíz se incrementan monótonamente.*

Demostración:

El teorema es claro si no se aplica alguna de las estrategias de reducción de trayectorias, pues de las reglas para las técnicas *Unión por Tamaño* y *Unión*

por Rango, según se demostró en las Secciones 3.3 y 3.4, siempre se mantiene el rango del árbol más grande o se incrementa en uno en caso de tener los mismos rangos, o los mismos tamaños. Si después se aplica alguna técnica de *Reducción de Trayectorias*, si algún nodo v es descendiente de w, claramente v tuvo que ser descendiente de w cuando sólo se consideraba la operación LINK. Por lo cual, el rango de v es estrictamente menor que el rango de w.

Concluimos la sección mostrando que para las técnicas de Unión por Tamaño y Unión por Rango, su tiempo promedio de ejecución también es su peor caso.

Teorema 4.5 *El algoritmo* SET-UNION, *usando las estrategias* Unión por Tamaños *y* Unión por Rango, *para la operación* LINK, *se ejecuta en tiempo* $\Theta(n + m \log_2 n)$, *en el peor caso.*

Demostración:

En las subsecciones 4.3 y 4.4 obtuvimos que $O(n + m \log_2 n)$ es una cota sobre el tiempo requerido para una secuencia de m operaciones. Para la cota inferior pensemos en un árbol binomial B_i que tiene tamaño 2^i y altura i. Para cualquier valor de n es posible construir un árbol B_i que contenga $2^{\lfloor \log_2 n \rfloor - 1}$ nodos usando cualquier regla de unión, ya que los árboles ligados en cada paso son isomorfos. Si se repite la ejecución de un FINDSET sobre la trayectoria de longitud $\lfloor \log_2 n \rfloor - 1$, el tiempo total es $\Omega(n + m \log n)$. Con lo cual se concluye la demostración.

Capítulo 5

El TDA Conjuntos Ajenos: Cotas Amortizadas

Una vez que hemos establecido el TDA Conjuntos Ajenos, tanto sus operaciones como propiedades, pasamos a establecer su tiempo de ejecución desde el punto de vista amortizado. El análisis amortizado en este caso es muy interesante. Por un lado, al aplicar las estrategias de reducción de trayectorias el tiempo de la ejecución se vuelve lineal en el sentido amortizado. Por otro lado, resulta ser muy formal y ceremoniosa la forma en que es atacado y resuelto este problema. Se pretende contar cuánto cuesta la operación FINDSET cuando se utilizan las estrategias de compresión de trayectorias y cómo se van acomodando los datos a lo largo de la ejecución del algoritmo SET-UNION. Además llama la atención el uso de una extraña función denominada *función de Ackerman* cuyo crecimiento es exponencial y cuya función inversa, denominada la función α, tiene un crecimiento muy lento.
Comenzaremos por dar un panorama general de cómo se pretende abordar el problema; utilizar esto nos da la pauta para mostrar un buen ejemplo de cómo se trabaja y razona con la técnica del banquero para el análisis amortizado. Continuaremos dando una demostración más formal sobre las cotas obtenidas.

5.1 Un Esquema de Conteo

En esta sección determinaremos una cota muy restringida sobre el tiempo de ejecución de una secuencia de $m = \Omega(n)$ operaciones FINDSET y LINK. También supondremos que las operaciones FINDSET y LINK pueden efectuarse en cualquier orden, y también supondremos que a las operaciones LINK le son aplicadas cualquiera de las estrategias Unión por Tamaño o Unión por Rango. En este sentido la operación FINDSET está implementada con la técnica de Compactación de Trayectorias.

Ahora bien, la estrategia a seguir es: sabemos que una operación FINDSET sobre un nodo v cuesta un tiempo proporcional al número de nodos sobre la trayectoria que va de v a la raíz del árbol que lo contiene. Entonces cargaremos una unidad de costo por cada nodo sobre la trayectoria desde v a la raíz durante cada FINDSET. Para ayudarnos

a contar las cargas, depositaremos una moneda imaginaria dentro de cada nodo sobre la trayectoria. Este conteo es estrictamente un mecanismo de ayuda y no forma parte de la implementación; es equivalente al uso de la función potencial. Cuando el algoritmo termine, recolectaremos todas las monedas que fueron depositadas para determinar el costo total.

Un detalle más acerca de nuestro mecanismo de conteo: serán utilizadas dos tipos de monedas: Pesos y Quetzales. Con esto se mostrará que durante la ejecución del algoritmo solamente podemos poner un cierto número de Pesos durante cada FINDSET, sin tomar en cuenta cuantos nodos haya, y también se mostrará que sólo puede haber un cierto número de Quetzales depositados en cada nodo —sin considerar cuantas operaciones FINDSET se hayan realizado.

Para aplicar el sistema de conteo dividimos los nodos por rangos y éstos en grupos. A cada grupo le llamamos **grupo de rango**. Para poder contar los cambios suponemos que depositamos monedas. Recordemos que hay dos tipos de monedas: "Pesos" y "Quetzales". Sobre cada operación FINDSET se deposita un *Peso* dentro de un "monedero general" y algunos *Quetzales* dentro de nodos específicos.

Después contamos el número total de Quetzales depositados y contamos los depósitos por nodo. Así, sumando todos los depósitos por cada nodo de rango r obtenemos el total depositado para cada rango r, luego sumamos todos los depósitos para cada rango r en el grupo g.

Sumamos todos los depósitos para cada grupo de rango g, obteniendo el número total de Quetzales depositados en el bosque; para terminar sumamos este número a la cantidad total de Pesos dentro del monedero general y obtener así la respuesta.

5.1.1 Los grupos de rango

Como mencionamos anteriormente repartiremos los rangos dentro de grupos. Los que se establecen con rango r se dice que están dentro del grupo $G(r)$, G será una función que se determinará después, al balancear las cargas de Pesos y Quetzales. El rango más grande en cualquier grupo de rango g es $F(g)$, donde $F = G^{-1}$ que denota a la función inversa de G.

Entonces, el número de rangos en cualquier grupo de rango g, $g > 0$, es: $F(g) - F(g-1)$. Esto nos indica que $G(n)$ es una cota superior muy amplia sobre el grupo de mayor rango[1].

Como ejemplo supongamos que partimos los rangos como en la Figura 5.1. En este caso $G(r) = \sqrt{r}$. El rango más grande en el grupo g es $F(g) = g^2$, observamos que el grupo g, $g > 0$, contiene rangos de $F(g-1) + 1$ hasta $F(g)$ inclusive. Esta fórmula no se aplica para nodos de rango 0 así que por convención el grupo de rango 0 contiene elementos de rango 0. Note que los grupos están hechos de rangos consecutivos (aunque esto no es obligatorio).

Recordemos que cada instrucción LINK toma tiempo constante y cada raíz guarda su rango. De este modo, las operaciones LINK son esencialmente gratuitas para el análisis.

Cada operación FINDSET toma un tiempo proporcional al número de nodos sobre la trayectoria; comenzando en el nodo i hasta la raíz. Así que depositaremos una moneda

[1]Recordemos que en el capítulo anterior se definió a n como el número de operaciones MAKESET

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

Figura 5.1: Posible partición de los rangos dentro de grupos.

por cada vértice sobre la trayectoria, si esto fuera todo lo que hiciéramos no podríamos decir mucho sobre la cota, esto es por que no hemos tomado ventaja de las heurísticas de reducción de trayectorias, por lo cual hay que tomar en cuenta algunos hechos al respecto.

La observación clave es: como resultado de aplicar alguna reducción de trayectorias, un nodo obtiene un nuevo padre y el nuevo padre tiene garantizado un rango mayor que el padre anterior.

Incorporamos esta nueva información dentro de la prueba utilizando el siguiente conteo: para cada nodo v sobre la trayectoria a partir del nodo consultado i hasta la raíz, depositaremos una moneda conforme a las siguientes reglas[2]:

1. Si v es la raíz o si el padre de v es la raíz, o si el padre de v está en un grupo de rango diferente del grupo de v, entonces el precio es de una unidad bajo estas condiciones. Así que depositamos un Peso.

2. En otro caso depositamos un Quetzal dentro del nodo.

Lema 5.1 *Para cualquier ejecución de la operación* FINDSET, *el número total de monedas depositadas en un monedero o en un nodo es exactamente igual al número de nodos recorridos durante la ejecución de* FINDSET.

Demostración:

Cuando se realiza una operación FINDSET(x), lo que se hace es posicionarse en el nodo x y seguir la trayectoria hacia la raíz del árbol, en ese momento puede suceder que:

- x sea la raíz, por lo que únicamente se deposita un Peso;
- que el padre de x sea la raíz entonces se deposita un Peso en x y un Quetzal en la raíz del árbol que contiene a x con lo que tenemos dos monedas depositada y dos nodos recorridos;
- en el caso en que x esté a n grupos de rango de la raíz, recursivamente por cada nivel en el que el padre del nodo está sobre la trayectoria de x a la raíz se deposita un Peso; en caso de que no sea cumpla la condición 1 entonces depositamos un Quetzal.

[2]estas reglas simulan cuentas de ahorro.

Con lo anterior vemos que el número total de monedas depositadas durante la ejecución de la operación FINDSET es igual al número de nodos que hay en la trayectoria desde x a la raíz del árbol que lo contiene.

Del Lema 5.1 se concluye que necesitamos sumar todos los Pesos depositados bajo la Regla 1 con todos los Quetzales depositados bajo la Regla 2. Veamos ahora como sumar estas cantidades.

Los Quetzales son depositados dentro de un nodo cuando éste es comprimido y su padre está en el mismo grupo de rango del nodo. Dado que los nodos consiguen un padre de rango mayor después de aplicar una reducción de trayectorias y puesto que el tamaño de un grupo de rango es finito, eventualmente los nodos llegarán a obtener un padre que no está en su grupo de rango. En consecuencia sólo hay un número limitado de Quetzales que pueden ser puestos dentro de cualquier nodo y este número es aproximadamente el tamaño del grupo de rango.

Por otro lado, los depósitos de Pesos son también limitados, esencialmente por el número de grupos de rango. Así que queremos elegir grupos de rango que sean lo más pequeños, limitando los depósitos de Quetzales, pero que esto no sea demasiado, para limitar las cargas de Pesos. Esto queda justificado en los siguientes resultados.

Lema 5.2 *Sobre una secuencia completa del algoritmo* SET-UNION *con m operaciones, el total de Pesos depositados bajo la Regla 1 suma $m(G(n)+2)$.*

Demostración:

Para cualquier operación FINDSET son depositados a lo más dos Pesos: Un Peso depositado en la raíz y otro en su hijo, si es que tiene hijos. Por el Teorema 4.4, para los vértices que apuntan hacia arriba en la trayectoria, se tiene que sus rangos son monótonamente crecientes, por lo cual nunca se decrementan cuando subimos en la trayectoria. Dado que hay a lo más $G(n)$ grupos de rango —inclusive el grupo 0 — solamente $G(n)$ vértices pueden ser calificados con un depósito de la Regla 1 para cualquier operación FINDSET particular. De este modo, durante cualquier ejecución de FINDSET a lo más hay $G(n)+2$ Pesos que pueden ser depositados. Entonces, puede haber a lo más $m(G(n)+2)$ Pesos que pueden ser depositados bajo la Regla 1 para una secuencia de m operaciones FINDSET.

Lema 5.3 *Para cualquier nodo simple en un grupo de rango g el número total de Quetzales depositados es a lo más $F(g)$.*

Demostración:

Si un Quetzal es depositado dentro de un vértice v bajo la Regla 2, v será movido por alguna técnica de reducción de trayectorias y obtendrá un padre de rango mayor que el de su padre anterior. Dado que el rango más grande en su grupo es $F(g)$ se garantiza que después de $F(g)$ Quetzales depositados, los padres de v no alargaran los grupos de rango de v.

La cota dada en el Lema 5.3 puede ser mejorada utilizando únicamente el tamaño del grupo del rango en vez de la extensión de sus miembros; sin embargo, esto no mejora la cota que es obtenida para el algoritmo SET_UNION. Siguiendo con los resultados tenemos:

Lema 5.4 *El número de nodos, $N(g)$, en un grupo de rango g, $g > 0$, es a lo más $n/2^{F(g-1)}$.*

Demostración:

Por el Corolario 4.3 hay a lo más $n/2^r$ nodos de rango r. Haciendo la suma sobre los rangos en el grupo g obtenemos:

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

Teorema 5.1 *El número máximo de Quetzales depositados en todos los vértices en el grupo de rango g es a lo más $n(F(g)/2^{F(g-1)})$.*

Demostración:

Simplemente hay que multiplicar los valores obtenidos en el Lema 5.3 con el obtenido en el Lema 5.4, esto es:

$$F(g)(n/2^{F(g-1)}) = n(F(g)/2^{F(g-1)}).$$

Grupo	Rango
0	0
1	1
2	2
3	3,4
4	5 hasta 16
5	17 hasta 65536
6	65536 hasta 2^{65536}
7	Es un rango muy grande

Figura 5.2: Partición de los grupos de rango.

Teorema 5.2 *La suma total de Quetzales depositados bajo la Regla 2, es a lo más*

$$n\sum_{i=1}^{G(n)}\frac{F(g)}{2^{F(g-1)}}.$$

Demostración:

Dado que el grupo de rango 0 contiene únicamente elementos de rango 0 éste no contribuye a las cantidades depositadas bajo la regla 2, pues estos elementos no tienen un padre en el mismo grupo de rango. La cota se obtiene por la suma de los otros grupos de rango.

Dados los resultados anteriores las cantidades depositadas por las Reglas 1 y 2, dan en total:

$$m(G(n)+2)+n\sum_{i=1}^{G(n)}\frac{F(g)}{2^{F(g-1)}} \tag{5.1}$$

Sin embargo, no hemos especificado quién es $G(n)$ o su inversa $F(n)$. Obviamente, estamos en libertad de elegir cualquier función que se desee, pero lo que podemos hacer en ese sentido es elegir a $G(n)$ de tal forma que minimice la cota en la ecuación anterior. Sin embargo, si $G(N)$ es muy pequeña, $F(n)$ será muy grande y en consecuencia violaría la cota. Una aparente elección razonable, es escoger a $F(i)$ como una función recursiva definida de tal forma que $F(0) = 0$ y $F(i) = 2^{F(i-1)}$. Con esto podemos decir que $G(n) = 1 + \log^* n$; en la Figura 5.2 se muestra la partición de rangos, propuesta en base a los resultados demostrados.

Notemos que el grupo 0 tiene rango 0, lo cual es requerido en la prueba del último teorema. Así pues, F es muy similar a la función de Ackerman de una sola variable, diferenciándose sólo en la definición del caso base. Con esta elección de F y G podemos completar el análisis.

Teorema 5.3 *El tiempo de ejecución del algoritmo* SET_UNION *con* $m = \Omega(n)$ *operaciones* FINDSET *es de* $O(m \log^* n)$.

Demostración:

Colocando las definiciones de F y G en la Ecuación 5.1, el número total de Pesos es $O(mG(n)) = O(m \log^* n)$. Puesto que $F(g) = 2^{F(g-1)}$ el número total de Quetzales es $nG(n) = O(n \log^* n)$. Como supusimos que $m = \Omega(n)$, la cota queda establecida.

5.2 Formalización del Análisis.

Para el algoritmo *Set_Union* utilizando el método de unión por rango y utilizando la técnica de bisección o partición de trayectorias, los investigadores van der Weider y van Leewen [6] obtuvieron una cota en el tiempo de ejecución de $O(m \log n)$ bajo la suposición de que $m > n$, que fue lo que revisamos en la sección anterior. Ahora probaremos que el algoritmo se ejecuta en un tiempo de $O(n + m\alpha(m + n, n))$, ya sea utilizando *unión por rango* o *unión por tamaño* y cualquiera de las técnicas de reducción de trayectorias, esto es: partición, compresión o bisección de trayectorias. Ello nos llevará a concluir que cualquier combinación de estos métodos son asintóticamente óptimos. Para mostrar esta cota incluiremos otra técnica denominada *Partición Múltiple* la cual es atribuida a Tarjan [6]. Se modificará la técnica para obtener una cota exacta tanto para $m < n$ como para $m > n$.

5.2.1 Las Herramientas.

Ya que las estrategias de compresión, partición y bisección son una forma de compactación, durante esta sección no se hará referencia a un método en particular, a menos que sea necesario. Es decir, hablaremos en general de compactación.

A continuación se explica la técnica de Tarjan [6] sobre la Partición Múltiple. Considere cualquier secuencia de operaciones MAKESET, FINDSET y LINK, implantadas de tal forma que la operación FINDSET compacta la trayectoria encontrada, ver la subsección 4.5.4. El tiempo requerido para ejecutar la secuencia de operaciones está acotado por una constante que multiplica a la suma de n y el número total de nodos sobre la trayectoria encontrada, asumiendo que la compactación de trayectoria toma tiempo lineal sobre el número total de nodos en la trayectoria.

Para analizar el número de nodos en la trayectoria encontrada, se requiere de algunas herramientas. Por el momento, no se especificarán los métodos para la operación LINK, ya que es posible usar la misma estructura de la prueba para analizar compactación con cualquiera de los tres métodos de unión.

Se calculan los cambios de los apuntadores, provocados por la compactación con respecto a un bloque fijo, llamado *bosque de referencia*. El bosque de referencia de una secuencia de operaciones MAKESET, FINDSET y LINK es el bosque generado al ejecutar

todas la operaciones de esta secuencia, ignorando las operaciones FINDSET. De esta forma, no se lleva a cabo ninguna compactación.

Al padre de un nodo x, es el primer valor diferente de x asignado a $p(x)$ por la operación LINK o FINDSET. Durante esta sección, *se utiliza el rango de un nodo como su* **altura** *en el bosque de referencia*, denotado como $rango(x)$. El rango de un nodo se fija a través de la ejecución del algoritmo. Si la estrategia de unión por rango es usada, el rango del nodo x es el último valor asignando a $r(x)$ por el algoritmo, teniendo que $rango(x) = r(x)$.

Las siguientes propiedades se satisfacen para cualquier secuencia de operaciones MAKE-SET, FIND y LINK, utilizando compactación.

- Para cualquier nodo x, su padre $p(x)$ es siempre un ancestro propio de x en el bloque de referencia. Así los rangos se incrementan estrictamente a lo largo de cualquier trayectoria encontrada.
- Sea p el valor de la función padre justo antes de ejecutar una operación FINDSET, y sea p' el valor de la función padre justo después de una operación FINDSET. Si x es un nodo sobre una trayectoria encontrada, con x diferente tanto del último como del penúltimo nodo en la trayectoria entonces, la compactación asegura que $p'(x)$ es un ancestro común de $p(p(x))$ en el bosque de referencia.
- La compresión provoca que el rango del padre de x se incremente de $rango(x)$ a al menos $rango(p(p(x)))$. El rango nunca cambia, los padres sí, ver el Teorema 4.4.

Analizando estos incrementos en los rangos, es posible acotar el número total de nodos sobre la trayectoria encontrada. Para obtener mejores cotas, agruparemos los cambios de los rangos en niveles y contamos separadamente en cada nivel de cambio.

Para agrupar los cambios de rango, definimos una colección de particiones sobre los números enteros desde cero hasta el máximo rango de un nodo. Con esto hay una partición para cada nivel i, con $i \in [0, k]$, donde k es un parámetro que será elegido después. Los bloques de una partición de nivel i, son intervalos definidos por:

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

donde posteriormente diremos cómo se llega a el intervalo frontera $B(i, j)$ y al número de intervalos l_i en el nivel i.

Para esta definición, tiene sentido requerir de la función frontera $B(i, j)$, la cual está definida para $i \in [0, k]$ y $j \in [0, l_i]$, y debe tener las siguientes propiedades:

a) $B(0, j) = j$ para $j \in [0, l_i]$

b) $B(i, 0) = 0$ para $i \in [0, k]$

c) $B(i, j) < B(i, j + 1)$ para $i \in [1, k], j \in [0, l_i - 1]$

d) $B(i, l_i) > h$ para $i \in [0, k]$, donde h es el rango máximo de un nodo

e) $l_k = 1$.

De estas propiedades observamos que la propiedad (c) implica que los bloques de la partición en el nivel i son intervalos ajenos no vacíos. Las propiedades (b) y (d) implican que cada valor entero en el rango $[0, h]$ está en el mismo bloque de la partición en el nivel i. La propiedad (a) implica que cada bloque de nivel cero es un punto. La propiedad (e) implica que la partición en el nivel k consiste de un solo bloque.

Cada nivel de los bloques particiona a los nodos por rangos. Para $i \in [0, k]$ y $j \in [0, l_i - 1]$ definimos a n_{ij} como el número de nodos con rango en el bloque $(bloque(i, j) - bloque(i - 1, 0))$. Entonces, para cualquier i se cumple que:

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

Lo que se intenta es que la partición llegue a ser más *amplia* siempre que el nivel se incremente. Como una medida de esta amplitud, utilizaremos la función b_{ij}, para la cual $i \in [0, k]$, $j \in [1, l_i - 1]$, que representa el número de bloques de niveles $(i - 1)$ cuya intersección con $bloque(i, j)$ es no vacía.

Conforme el algoritmo se ejecuta, cada nodo x tiene un *nivel*, definiéndose como el valor mínimo de i tal que el $rango(x)$ y $rango(p(x))$ están en el mismo bloque de la partición en el nivel i. Dado que la partición de nivel cero consiste solamente de puntos aislados y la partición del nivel k consiste de un solo bloque, $nivel(x) \in [1, k]$ a menos de que x sea la raíz del árbol, en cuyo caso $nivel(x) = 0$. Mientras que el rango de un nodo está fijo, el nivel de un nodo puede incrementarse pero no decrementarse mientras el algoritmo se ejecuta.

Para acotar el número de nodos sobre las trayectorias encontradas, asignamos una *carga* por cada operación FINDSET. La carga es asignada entre los nodos de la trayectoria encontrada y a la operación FINDSET en sí misma, de manera que dependa sólo de los niveles de los nodos justo antes de que una operación FINDSET se ejecute. La carga asignada para un nodo dado es asignado entre los niveles.

Las reglas para la asignación de cargas son un poco complicadas por la generalidad de los resultados que se intentan obtener. Se utilizarán dos reglas para asignar cambios:

Regla1.- Cargo a la operación FINDSET: Cargar $3k$ a la operación

Regla2.- Cargo a un nodo: Sea x cualquier nodo sobre la trayectoria encontrada, x diferente del primero y último nodo. Sea i el nivel máximo de cualquier nodo que preceda a x sobre la trayectoria. Si se cumple

$$\min\{i, nivel(p(x))\} \geq nivel(x),$$

cargar al nodo x

$$\min\{i, nivel(p(x))\} - nivel(x) + 1$$

de esta cantidad, cargar uno a cada nivel en el rango

$$[nivel(x), \min\{i, nivel(p(x))\}]$$

Nota: Las reglas de cargo no cambian el penúltimo nodo sobre la trayectoria encontrada.

Lema 5.5 *La cantidad de* cargas *para una operación* FINDSET *es al menos el número de nodos sobre la trayectoria.*

Demostración:

Sea $x_0 \to x_1 \to \cdots \to x_h$ una trayectoria encontrada, que inicia en el nodo x_0 y termina en el nodo x_n. Sea *nivel* la función nivel justo antes de ejecutar la operación FINDSET y sean:

$$P = \{i \mid i \in [0,h] \text{ y } nivel(x_i) \leq nivel(x_{i+1})\},$$
$$N = \{i \mid i \in [0,h] \text{ y } nivel(x_i) > nivel(x_{i+1})\}.$$

Entonces,

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

Lo cual implica,

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

y

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

Sea $Y = y_0, y_1, \ldots, y_g$ una subsecuencia de $x_0, x_1, \ldots, x_h$ consistente de todos los nodos x_i cuyo nivel excede el nivel de todos los nodos anteriores sobre la trayectoria encontrada. Notemos que $x_0 = y_0$, ninguno de los nodos y_i fue cargado por la operación FINDSET, y $g \leq k-1$. Consideremos la cantidad cargada para la operación FINDSET por la Regla 2. Si $x_i \in P$, pero $x_i + 1$ no está en la subsecuencia Y, entonces un cargo del nivel $(nivel(x_{i+1} - nivel(x_i)) + 1)$ es asignado a x_i. En este caso, x_i no puede estar en la subsecuencia de Y. Si $x_i \in P$ y $x_{i+1} = y_{i+1}$ para algún entero j, un cargo de al menos

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

es asignado a x_i. Esto incluye la posibilidad de que $y_{j-1} = x_i$; en tal caso, el cargo asignado a x_i es cero. Así, el total de cargos, incluyendo el cargo a la operación FINDSET, es al menos:

$$\begin{aligned}&\sum_{i\in P}(nivel(x_{i+1}) - nivel(x_i) + 1) - \sum_{j=1}^{g}(nivel(y_j) - nivel(y_{j-1}) + 1) + 3k \\ &\qquad \geq h - nivel(x_0) - nivel(y_g) + nivel(y_0) - (k-1) + 3k \geq h + 1\end{aligned}$$

dado que $y_0 = x_0$, se concluye la demostración.

El siguiente lema da una cota sobre la carga total para las operaciones FINDSET.

Lema 5.6 *El total de cargas para todas las operaciones* FINDSET *y, de aquí, el número total de nodos sobre las trayectorias encontradas es al menos:*

$$3km + \sum_{i=1}^{k} \sum_{j=0}^{l_i-1} b_{ij} n_{ij}$$

Demostración:

Si el nodo x es cargado en el nivel i entonces $1 \leq nivel(x) \leq i$ antes de que la operación FINDSET sea ejecutada y $nivel(x) \geq i$ después de efectuar el FINDSET. Esto indica que $nivel(p(x)) = i$ después de la operación FINDSET, si $nivel(x) > i$ después del FINDSET. Decir que $nivel(p(x)) = i$ significa que $r(p(x))$ y $r(p(p(x)))$ están en diferentes bloque en el nivel $(i-1)$.

La operación FINDSET provoca que $p(x)$ y, de este modo, $r(p(x))$ estén en un nuevo bloque en el nivel $i-1$. Esto puede suceder al menos $(b_{ij}-1)$ veces sin incrementar el nivel de x, ya que $bloque(i,j)$ intersecta solamente $b_{i,j}$ bloques del nivel $i-1$. Así, después de que x es cambiado b_{ij} veces en el nivel i, su nivel es al menos $i+1$ y nunca llega a ser cargado en el nivel i.

Para que x sea cargado en el nivel i debe tener predecesor, digamos x', sobre la trayectoria encontrada y tal que $nivel(x') \geq i$ antes de la ejecución de la operación FINDSET. Por definición de los niveles, $rango(x')$ y $rango(p'(x))$ están en diferentes bloques en el nivel $i-1$, antes del la operación FINDSET. Dado que el $rango(x) \geq rango(p(x'))$ y $rango(x) \notin bloque(i-1,0)$. Esto implica que el número de nodos cuyo rango está en $bloque(i,j)$ y que pueden ser cargados en el nivel i es al menos n_{ij}.

Sumando lo anterior a la cantidad de cargas por las operaciones FINDSET obtenemos un total de cargas de al menos:

$$3km + \sum_{i=1}^{k} \sum_{j=0}^{l_i-1} b_{ij} n_{ij}.$$

A continuación se van a acotar los valores de n_{ij}, para ello tomaremos indistintamente la operación LINK esto es, sin distinguir si hacemos unión por tamaño o unión por rango.

Lema 5.7 *Usando la operación* LINK, *implantada ya sea con unión por rango o unión por tamaño, se tiene que:*

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

Demostración:

Por el Corolario 4.3 tenemos que,

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

Lema 5.8 *En cualquier secuencia de operaciones para conjuntos, que sea implantada usando cualquier forma de* compactación *para la operación* FINDSET *y utilizando* unión por tamaño *o* unión por rango *para la operación* LINK, *el número total de nodos sobre una trayectoria encontrada es a lo más:*

$$3m\alpha(m+n,n) + 4m + 13n$$

Demostración:

Para esta demostración elegimos:
$k = \alpha(m+n,n) + 1$,
$l_i = \min\{j \mid A(i,j) > \log_2 n\}$, para $i \in [1, \alpha(m+n,n)]$, $l_k = 1$ y;

$$\begin{aligned} B(i,j) &= A(i,j) \quad \text{para} \quad i \in [1,\alpha(m+n,n)],\ j \in [1,l_i], \\ B(k,1) &= \lfloor \log_2 n \rfloor + 1. \end{aligned}$$

La Figura 5.3 muestra una partición múltiple para la compactación con Unión por Rango o Unión por Tamaño para la operación LINK. El nivel cero es omitido, pues consiste solamente de puntos aislados. Se usa una escala logarítmica.

Con las definiciones anteriores es fácil ver que la función frontera $B(i,j)$ tiene las propiedades (c), (d) y (e). Por el Corolario 4.2 ningún nodo tiene rango que exceda de $\log_2 n$. Ahora haremos una estimación de los valores para b_{ij} de la siguiente manera:

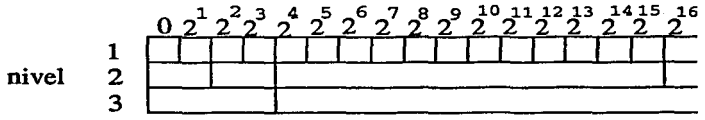


Figura 5.3: Partición Múltiple para compactación de trayectorias.

(i) $b_{i0} = 2$ para $i \in [1, \alpha(m+n, n)]$: Para $i \in [1, \alpha(m+n, n)]$,

$$bloque(i, 0) = bloque(i-1, 0) \cup bloque(i-1, 1)$$

dado que $A(1,1) = 2$ y $A(i,1) = A(i-1, 2)$ para $i \geq 2$

(ii) $b_{ij} \leq A(i,j)$ para $i \in [1, \alpha(m+n, n)], j \in [1, l_i - 1]$.
Para $j \in [i, l_1 - 1]$,
$bloque(1, j) = [A(1,j), A(1, j+1) - 1] = [2^j, 2^{j+1} - 1]$,
así que $b_{ij} = 2^j = A(1,j)$.
Para $i \in [2, \alpha(m+n, n)]$, $j \in [1, l_i - 1]$ tenemos,

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

De este modo $b_{ij} \leq A(i,j)$.

(iii) $b_{k0} = \lfloor (m+n)/n \rfloor$:
Ahora tenemos que por la definición de la función α
$b_{k0} = l_{\alpha(m+n,n)} = \min\{j \mid A(\alpha(m+n, n), j) > \log n\} \leq \lfloor (m+n)/n \rfloor$.

Para estimar la cota obtenida en el Lema 5.6 partiremos la suma

$$\sum_{i=1}^{k} \sum_{j=0}^{l_i - 1} b_{ij} n_{ij}$$

en tres partes.
Primero,

$$\sum_{j=0}^{l_k - 1} n_{kj} b_{kj} = n_{k0} b_{k0} \leq n \lfloor (m+n)/n \rfloor \leq m + n$$

Segunda,

$$\begin{aligned}\sum_{i=1}^{k-1} b_{i0}n_{ij} &\leq \sum_{i=1}^{k-1} \frac{2n}{2^{B(i-1,1)-1}} \quad \text{por el Lema 5.7} \\ &\leq 4n\sum_{i=1}^{k-1} \frac{1}{2^{B(i-1,1)-1}} \\ &\leq 4n.\end{aligned}$$

Y la tercera parte consiste en que para $i \in [1, \alpha(m+n,n)]$,

$$\begin{aligned}\sum_{j-1}^{l_i-1} b_{ij}n_{ij} &\leq \sum_{j=1}^{l_i-1} \frac{A(i,j)n}{2^{A(i,j)-1}} \quad \text{por lema 5.7} \\ &\leq \sum_{h=A(i,1)}^{\infty} \frac{hn}{2^{h-1}} \\ &= \frac{n(A(i,1)+1)}{2^{A(i,1)-2}}\end{aligned}$$

Lo cual implica,

$$\begin{aligned}\sum_{i=1}^{k-1}\sum_{j=1}^{l_i-1} b_{ij}n_{ij} &\leq \sum_{i=1}^{k-1} \frac{n(A(i,1)+1)}{2^{A(i,1)-2}} \\ &\leq n\sum_{h=2}^{\infty} \frac{h+1}{2^{h-2}} = 8n.\end{aligned}$$

Combinando estas estimaciones, se tiene que el total cargado para todas la operaciones FINDSET es a lo más de:
$3m(\alpha(m+n,n)+1) + m + 13n = 3m(\alpha(m+n)) + 3m + m + 13n.$

Con la estimación obtenida en el Lema 5.8, por fin estamos listos para demostrar la cota generalizada para los algoritmos del TDA Conjuntos Ajenos.

Teorema 5.4 *El Algoritmo* SET-UNION *con Unión por Rango o Unión por Tamaño para la operación* LINK *y cualquiera de las estrategias de Reducción de Trayectorias para la operación* FINDSET *requiere tiempo* $O(n + m\alpha(m+n,n))$, *y es asintóticamente óptimo.*

Demostración:

Para los métodos de compresión y partición el teorema es inmediato del Lema 5.8. Para Bisección se tiene que cambiar el método de partición múltiple.

A cada nodo sobre la trayectoria encontrada, empezando con el primero, se le denomina **nodo esencial**. Los nodos esenciales sobre la trayectoria encontrada, excepto el último, son aquellos cuyos padres cambiaron cuando la trayectoria fue bisectada. Para cada nodo x se define el nivel, $nivel(x)$ como el mínimo valor de i tal que los rangos $r(x)$ y $r(p(x))$ están en el mismo bloque de partición del nivel i.

La regla para hacer un cargo a un nodo es la siguiente:

- Carga a Nodo: Sea x cualquier nodo esencial sobre la trayectoria de búsqueda diferente al primero y el último nodo. Sea i el máximo nivel de cualquier nodo esencial que precede a x sobre la trayectoria. Si

$$\min\{i, nivel(p^2(x))\} \geq nivel(x),$$

cargamos $\min\{i, nivel(p^2(x))\} - nivel(x) + 1$ al nodo x. De esta cantidad cargamos 1 a cada nivel en el rango:

$$[nivel(x), \min\{i, nivel(p^2(x))\}].$$

Las demostraciones de los Lemas 5.5, 5.6 y 5.8, son aplicadas y sirven para acotar la cantidad total de los nodos esenciales sobre la trayectoria encontrada. Es necesario reemplazar $p(x)$ por $p^2(x)$ en la prueba del Lema 5.6, ya que al menos la mitad de los nodos en cada trayectoria encontrada son esenciales; así tenemos dos veces la cota obtenida en el Lema 5.8 y con ello se cumple la cota.

El Teorema 5.4 proporciona seis diferentes algoritmos para SET-UNION, los cuales son asintóticamente óptimos. Quizá el mejor es la versión que usa Unión por Rango y bisección, ya que requiere menos espacio que Unión por Tamaño y utiliza solamente una pasada sobre cada trayectoria encontrada.

Capítulo 6

El TDA Conjuntos Ajenos: Aplicaciones

Hasta ahora hemos hecho un análisis profundo sobre el TDA Conjuntos Ajenos es tiempo de ver un par de ejemplos donde este es aplicado, así como el impacto que ejerce su aplicación sobre la solución de estos problemas.

6.1 El Antecesor Común más Cercano

Este es un problema interesante que aparece para árboles enraizados, el cual tiene aplicaciones en el campo de la biología, pues con este árbol se puede representar y analizar por ejemplo, la evolución de un cultivo de bacterias.

El problema fuera de línea del Antecesor Común más Cercano

> Dado un árbol y una lista de parejas de nodos del árbol encontrar el antecesor común más cercano para cada pareja dada.

Como un ejemplo de este problema, observemos la Figura 6.1 donde se muestra un árbol cuya lista de pares está compuesta de cinco nodos —los cuales están marcados con minúsculas— $(x, y), (u, z), (w, x), (z, w), (w, y)$, con lo cual para el par de nodos u y z, el nodo C es el antecesor común más cercano a ellos, —los nodos A y B también son antecesores comunes pero no son los más cercanos. Con esta idea vemos en la Figura 6.1 que para la lista dada de parejas sus respectivos antecesores comunes son los nodos A, C, A, B y y. Al problema se le denomina *fuera de línea* puesto que es necesario revisar toda la secuencia de requerimientos antes de dar la primera respuesta.

El algoritmo se ejecuta sobre el árbol haciendo un recorrido en *post-orden*. Cuando está a punto de regresar el procesamiento de un nodo, examina la lista de pares para ver si hay algún antecesor común calculándolo durante la ejecución. Supongamos que u es el nodo a procesar, (u, v) está en la lista de pares y que ya finalizó el llamado recursivo a v, entonces, tenemos ya suficiente información para determinar el antecesor común más cercano de u y v, que denotaremos como $ACMC(u, v)$.

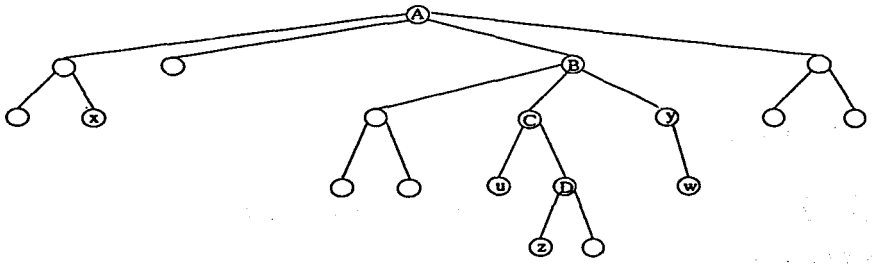


Figura 6.1: Ejemplo para el problema del Antecesor Común más Cercano

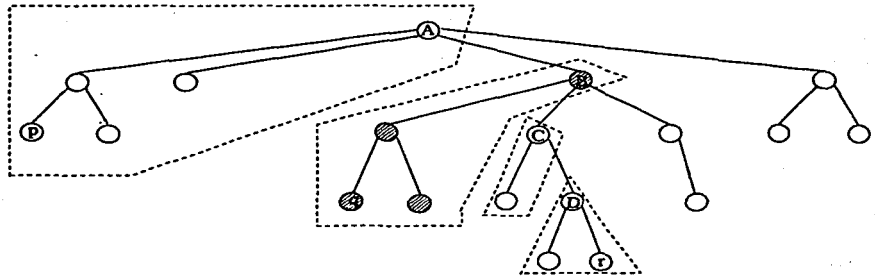


Figura 6.2: Funcionamiento del Algoritmo *ACMC*.

Para entender como funciona este algoritmo examinemos la Figura 6.2. En la figura se muestra que el algoritmo está a punto de terminar la llamada recursiva al nodo D. Todos los nodos que están sombreados tuvieron que haber sido visitados por una llamada recursiva, excepto los nodos sobre el camino a D, y todas las llamadas recursivas a los nodos sombreados ya han sido terminadas. Siempre se marca el nodo después de que su llamada recursiva es finalizada.

Si un nodo cualquiera v es marcado, entonces el $ACMC(D, v)$ es algún nodo sobre la trayectoria de D. Para tener un control sobre quién es el antecesor común más cercano de un nodo introducimos el siguiente concepto:

> Se define como *ancla* de un nodo visitado v —pero no necesariamente marcado— al nodo sobre la trayectoria recorrida actualmente por una llamada recursiva y que está "cerca" de v.

En la Figura 6.2 se observa el conjunto antes de que regrese de la llamada recursiva a

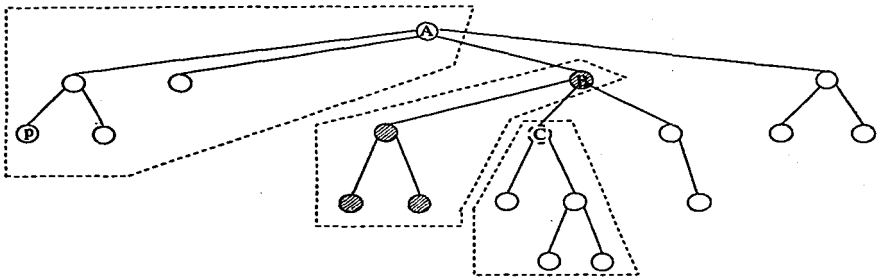


Figura 6.3: Nueva situación al terminar la llamada recursiva a C.

D; luego D es marcado como visitado y el $ACMC(D, v)$ es el ancla de v en la trayectoria actual; otros ejemplos de este concepto son el ancla de p que es A; B es el ancla de q, y r está sin marcar puesto que tiene que ser visitado; pero podemos decir que por definición el ancla de r es r mismo, hasta que r sea visitado por primera vez. Como se muestra en la Figura 6.2, cada nodo sobre la trayectoria que está siendo recorrida es un ancla, al menos de sí mismo. A partir de esta definición tenemos que los nodos visitados forman clases de equivalencia: *Los nodos están relacionados si ellos tienen la misma ancla y podemos considerar que cada nodo no visitado forma por sí mismo una clase de equivalencia.* Ahora supongamos una vez más que el par (D, v) está en la lista. Entonces tenemos tres casos:

1. v no está marcado, por lo que no tenemos información para determinar $ACMC(D, v)$. Sin embargo, cuando v sea marcado estaremos en posibilidad de determinarlo.

2. v ha sido marcado pero no está en el subárbol de D, entonces el valor de $ACMC(v, D)$ es el ancla de v.

3. v está en el subárbol de D así que $ACMC(v, D) = D$; notemos que éste no es un caso especial pues el ancla de v es D.

Solo falta asegurar que en cualquier momento podemos determinar el ancla de cualquier nodo visitado. Esto es fácil de hacer utilizando el TDA Conjuntos Ajenos. Después de que se regresa de la llamada recursiva, realizaríamos una operación LINK. Por ejemplo, después de que se regresa de la llamada recursiva a D, Figura 6.2, todos los nodos descendientes de D deben cambiar su ancla de D al nodo C. Así que antes de que la llamada recursiva a D regrese, unimos el conjunto anclado a D con el conjunto anclado a C y entonces calculamos el $ACMC(C, v)$ de todos aquellos nodos v que estén marcados antes de completar la llamada recursiva a C. La nueva situación se ilustra en la Figura 6.3. Así que necesitamos mezclar las dos clases de equivalencia en una. Al aplicar los algoritmos de Conjuntos Ajenos podemos obtener una nueva ancla para un nodo v utilizando la operación FINDSET.

Una vez revisado el panorama general de este problema lo definiremos y solucionaremos de manera más formal.

Definición 6.1 *El* Antecesor Común más Cercano *de dos nodos u y v de un árbol enraizado T es un nodo w que es antecesor tanto de u como de v y tiene la mayor profundidad en T.*

De la definición 6.1 queda aclarado el concepto de ser el más cercano, pues es el nodo que siendo antecesor común, posee la profundidad más grande dentro del árbol. Por ello en el ejemplo de la Figura 6.1, para el par (u, z) los nodos A y B son antecesores comunes, pero dijimos que no los más cercanos.

Ahora daremos un algoritmo para resolver este problema. Definimos una rutina llamada ACC, cuyo parámetro inicial será la raíz del árbol T en el cual iniciará el recorrido en *post_orden*. Para el control de los nodos ancla en cada paso del proceso, se mantendrá un arreglo de nodos denominado *color*; al iniciar el algoritmo todos los nodos serán pintados de blanco -- recordemos que todo nodo es ancla de sí mismo— y conforme se va regresando de las llamadas recursivas del recorrido sobre el árbol y se forman los conjuntos ajenos, serán pintados de negro. Así el algoritmo queda conformado como se muestra en el Listado 6.1.1.

Listado 6.1.1 Algoritmo del ACMC

```
 1 ACC(u)
 2  Make-Set(u)
 3  antecesor[Find_Set(u)]<-u
 4  for(para cada hijo v de u en T)
 5       do ACC(u)
 6          Link(u,v)
 7          antecesor[Find-Set(u)]<-u
 8  color[u]<- negro
 9  for(para cada nodo v tal que v esta en la lista de parejas)
10       do if color[v] = negro
11             then imprime ''El antecesor comun mas cercano de'' u
12                                  ''y'' v ''es'' antecesor[Find-Set(v)].
```

6.2 El problema del Árbol Generador de Peso Mínimo

En el diseño de circuitos electrónicos, a menudo es necesario hacer que los *pins*[1] de varios componentes eléctricos equivalentes estén conectados juntos. Para la interconexión de un conjunto de n pins podemos hacer un alambrado de $n - 1$ cables cada uno conectado a dos pins. De todos los posibles alambrados que podemos utilizar, el único que usaremos es el cableado de menor costo pues es generalmente el más deseable.

Podemos modelar el problema de cableado con una gráfica no dirigida y conexa; digamos que la gráfica $G = (V, E)$ representa el alambrado, donde el conjunto V es el conjunto de pins, E es el conjunto de posibles interconexiones entre los pares de pins de tal suerte

[1] Un pin es la entrada de corriente a la compuerta de un dispositivo lógico

que cada arista $e = (u, v) \in E$ está formado con $u, v \in V$. Asociamos un valor $c(e)$ que especifica el costo de conectar u y v. Queremos obtener entonces un subconjunto acíclico $T \subseteq E$ que conecta todos los vértices y cuyo peso total

$$c(T) = \sum_{e \in T} c(e)$$

sea mínimo. Dado que T es una gráfica acíclica y conecta todos los vértices, debe formar un árbol el cual será denominado como *árbol generador*. Entonces, al problema de determinar el árbol T según las condiciones anteriores se conoce como *el problema del árbol generador de peso mínimo*, lo cual será abreviado como *AGPM*.

Para continuar con la solución al problema del AGPM, hace falta definir el concepto de *componente conexa*. Sea $G = (V, E)$ una gráfica y sean u y v dos vértices en G. Decimos que u está conectado a v si existe en G un camino que une a u con v. Una gráfica G es conexa si cualesquiera par de vértices están conectados. Una gráfica que no está conectada se denomina disconexa o no-conexa. Es decir, una gráfica G es no conexa si existen dos vértices digamos u y v, para los cuales no existe un camino que los una.

Una subgráfica H de una gráfica G es una *componente conexa* de G, si H es una subgráfica conexa maximal de G. La relación "estar conectado a" es una relación de equivalencia sobre el conjunto de vértices de una gráfica y, por lo tanto, produce una partición $V_1, V_2, \ldots, V_k$ de $V(G)$. Las subgráficas $(V_1, E_1), (V_2, E_2), \ldots, (V_k, E_k)$, son las componentes conexas de G. La gráfica de la Figura 6.4 tiene tres componentes conexas

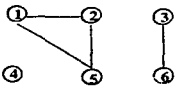


Figura 6.4: Ejemplo de Componentes Conexas de una Gráfica.

$\{1, 2, 5\}$, $\{3, 6\}$ y $\{4\}$. Cada vértice en $\{1, 2, 5\}$ está conectado a cualquier otro vértice en $\{1, 2, 5\}$.

Si G no es una gráfica conexa entonces, por cada componente conexa de G, podemos buscar un AGPM obteniendo entonces un *bosque generador de peso mínimo* de G, el cual denotamos como **BGPM**.

Hay toda una variedad de algoritmos para encontrar BGPM, pero en términos generales todos ellos trabajan de la misma forma. En general se toman dos conjuntos de aristas,digamos X y Y; al principio estos conjuntos son vacíos e irán creciendo gradualmente mientras se mantenga la siguiente invariante:

> Existe un Bosque Generador de Peso Mínimo de G el cual contiene todas las aristas de X y ninguna arista de Y.

Con lo anterior tenemos que al asignar valores iniciales a X y Y en vacío, queda establecido la invariante. Si $X \cup Y = E$, la invariante implica que la gráfica $F = (V, X)$ es BGPM de G. El siguiente teorema muestra como agregar una arista a X mientras se mantiene la invariante:

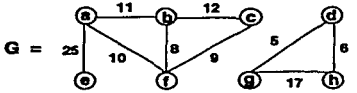
Teorema 6.1 *Sea $G = (V, E)$ una gráfica y supongamos que hay dos subconjuntos de E, llamados X y Y que cumplen con la siguiente condición:* existe un Bosque Generador de Peso Mínimo F de G el cual contiene todas las aristas de X y ninguna de las aristas de Y.

Sea C cualquier componente conexa de la subgráfica (V, X) y sea

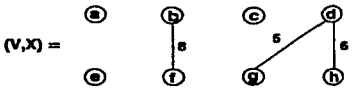
$$J(C) = \{(u, v) \in E - Y \mid u \in C \text{ y } v \notin C\},$$

$J(C)$ son las aristas de G que unen a C con el resto de G, excluyendo cualquiera de las aristas de Y. Supongamos que e es una arista de costo mínimo en $J(C)$; entonces existe un BGPM *de G llamado F' el cual contiene todas las aristas de $X \cup \{e\}$ y ninguna de Y.*

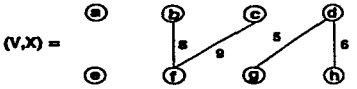
Antes de comenzar la demostración veamos un ejemplo. Consideremos la siguiente gráfica disconexa:



y supongamos que $X = \{(b, f), (d, g), (d, h)\}$ y $Y = \{\}$ los cuales satisfacen las hipótesis del teorema. Entonces,



Supongamos que la componente conexa C de (V, X) consiste de los vértices b y f en cuyo caso $J(C) = \{(b, a), (b, c), (f, a), (f, c)\}$. Puesto que la arista (f, c) tiene costo mínimo sobre las aristas de $J(C)$, el teorema dice que (f, c) puede ser agregado a X, con lo cual nos queda:



Demostración del Teorema 6.1:

Sea $G = (E, V)$ una gráfica, sean X y Y subconjuntos de E para los cuales existe un BGPM F que contiene todas las aristas de X y ninguna arista de Y; además sea C una componente conexa de la subgráfica (V, X) y sea

$$J(C) = \{(u, v) \in E - Y \mid u \in C \text{ y } v \notin C\}$$

el conjunto de aristas de G que une a C con el resto de G. Sea e una arista de peso mínimo en $J(C)$; lo que hay que demostrar es que si $F' = X \cup \{e\}$, entonces F' es un BGPM.

Por notación manejaremos a los bosques F y F' como conjuntos de aristas. Si $e \in F$ entonces $F' = F$ y la demostración se finaliza. Así que supongamos que $e \notin F$. Sea $e = (u, v)$, dado que u y v están conectadas por e, debe haber alguna trayectoria en F conectando a u con v. Esta trayectoria comienza en u que está en C y termina en v que no está en C; además tal trayectoria no contiene ninguna arista de Y, así que debe contener al menos una arista $e' \in J(C)$, observemos la Figura 6.5.

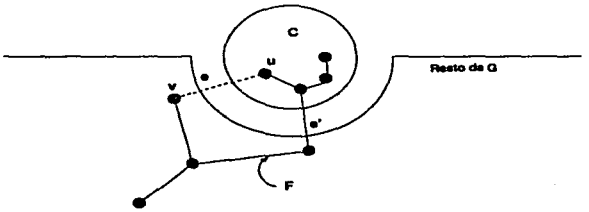


Figura 6.5: Posible estado de un camino de u a v

Dado que e es la arista de costo mínimo en $J(C)$ y como $e' \in F$, tenemos que $c(e) \leq c(e')$. Ahora bien, sea

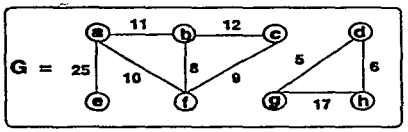
$$F' = F \cup \{e\} - \{e'\}.$$

Entonces F' es el bosque generador de G y, $c(F') = c(F) + c(e) - c(e') \leq c(F)$. Pero F era un bosque generador de peso mínimo así que $c(F) \leq c(F')$. Con esto se concluye que $c(F) = c(F')$, $c(e) = c(e')$ y F' también es un bosque generador de peso mínimo. Por lo tanto F' contiene todas las arista de $X \cup \{e\}$ y ninguna de las aristas de Y.

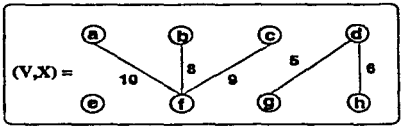
Notemos que el teorema no es difícil de aplicar para construir de forma manual el BGPM, solamente elegimos cualquier componente conexa C, la cual estará conformada inicialmente por un único vértice, agregamos la arista incidente con mínimo peso y

repetimos el proceso hasta completar el árbol. Se procede de manera similar con las demás componentes conexas, hasta completar el BGPM.

El algoritmo toma los costos de las aristas con respecto a un orden no decreciente. Por ejemplo, consideremos la gráfica G en el recuadro (a) de la Figura 6.6, las primeras



(a)



(b)

Figura 6.6: La Gráfica de ejemplo

aristas que toma son las que tienen costo $5, 6, 8, 9$ y 10. Ahora tales aristas pertenecen a X obteniéndose el Bosque parcial mostrado en el recuadro (b) de la misma Figura.

Después el algoritmo considerará como rechazadas a las aristas con costos $11, 12$ y 17 finalmente aceptará la arista con costo 25 terminado con el BGPM mostrado en la Figura 6.7:

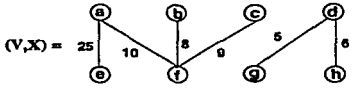


Figura 6.7: Bosque Generador de Peso Mínimo

6.2.1 Análisis y diseño del algoritmo de Kruskal

Formalicemos la última parte de nuestra discusión. Aplicando el Teorema 6.1, podemos encontrar el BGPM de una gráfica G; siguiendo la discusión es claro que primero hay que colocar las aristas de G en una cola de prioridades ordenándolas respecto a sus costos. Ahora bien, el algoritmo debe manipular las componentes conexas que se van formando

conforme lo que dice el Teorema 6.1. En este caso se utiliza el TDA Conjuntos Ajenos para mantener las componentes conexas, las cuales son por definición conjuntos ajenos. Cada conjunto contiene los vértices en un árbol del BGPM que se va formando. La operación FINDSET(u) regresa al representante de la componente conexa –o conjunto– que contiene a u. De este modo, podemos determinar cuándo dos vértices u y v pertenecen a la misma componente conexa, simplemente con probar que FINDSET(u) es igual a FINDSET(v). La combinación de componentes conexas es realizada por una operación LINK. Esta última descripción corresponde al *Algoritmo de Kruskal*, el cual se muestra formalmente en el Listado 6.2.1.

Listado 6.2.1 Algoritmo de Kruskal

```
1 grafica Kruskal(grafica G=(V,E)) {
2  /*Conjunto de aristas inicializado
3     en vacio*/
4   set X = {}, e = {};
5  /*Contadores de aristas y de
6    componentes conexas*/
7   int ka = 0, kcc = 1;
8  /*Estructura de Conjuntos Ajenos*/
9   Conj_Aj A;
10  /*Cola de prioridades*/
11  set Q = Ordena(E);
12
13  /*Calcula el numero de Componentes
14    conexas, con el Algoritmo BFS Modificado*/
15  kcc = BFS_CC(G);
16
17  /*Se contruyen los conjuntos ajenos con los
18    nodos de la grafica*/
19  A = Make_Set(V);
20
21  while (ka < |V| - kcc){
22    e = Elim_Min(Q);
23    /*Eliminar una arista (u,w) de costos minimo de Q*/
24    if(FindSet(e.u, A) != FindSet(e.w, A)){
25      /*Agrega la arista del BGPM que se almacena en la grafica
26      F=(V,X)*/
27      Agregar(e,X);
28      /*Unimos las componenetes conexas de los vertices u y v*/
29      Link(e.u, e.w, A);
30    }
31
32  }
33  return (V,X);
34 }
```

El tiempo de ejecución del algoritmo de Kruskal para una gráfica $G = (V, E)$ depende de dos factores:

1. la cola de prioridades donde ordenamos los pesos de las aristas y

2. la manipulación de las componentes conexas a través del TDA Conjuntos Ajenos.

El primer punto es fácil de determinar pues existe una gran variedad de implementaciones concretas de las colas de prioridades. Además, es bien sabido que las mejores implementaciones requieren de un tiempo de ejecución de $O(|E|\log|E|)$. Para el segundo punto primero asumiremos que la implementación del TDA Conjuntos Ajenos se realiza con la representación Galler-Fisher, aplicando la técnica de Unión por Rango para la operación LINK; también asumimos que se utiliza la estrategia de Compactación de Trayectorias;

Ahora bien, observemos lo detallado en el Listado 6.2.1, el algoritmo recibe una gráfica no necesariamente conexa G. Un resultado bien conocido de la Teoría de Gráficas nos dice que: dada una gráfica G no conexa con $|V|$ vértices y k componentes conexas, el número de aristas en un Bosque Generador de G es $|V|-k$. En base a esto, es fácil crear una versión de BFS para contar el número de componentes conexas de una gráfica no conexa en tiempo $O(\max\{|V|,|E|\})$, es decir, en tiempo $O(|E|)$. Esta rutina es invocada en la línea 11 del Listado 6.2.1.

Observemos que la creación de los primeros conjuntos ajenos toma un tiempo de $O(|V|)$ —linea 19. El ciclo **while** especificado entre las líneas 21 a 32, se ejecutará a lo más $|E|$ veces. Además notemos que este ciclo depende básicamente de las operaciones de Conjuntos Ajenos. Por lo tanto, es un algoritmo SET-UNION, entonces calculamos el número de veces que se ejecuta cada operación del TDA Conjuntos Ajenos. Para poder aplicar los resultados obtenidos para el TDA Conjuntos Ajenos en el Capítulo 5, tenemos que se ejecutan a lo más $2|E|$ operaciones FINDSET y, a lo más $|V|$ operaciones LINK con esto en mente el costo del ciclo **while** es:

- Según el Teorema 5.4, el tiempo de ejecución del ciclo es $O(|V|+|E|\alpha(|E|,|V|))$, tomando en cuenta que en terminos practicos $\alpha(|E|,|V|)\leq 4$, se puede considerar que $\alpha(|E|,|V|)$ es constante. Si la gráfica G tiene pocas aristas el ciclo **while** es casi lineal en $|V|$. Si la gráfica G es muy densa, el costo del ciclo es $O(|E|)$.

Esto muestra que el proceso de determinar si alguna componente conexa forma parte del BGPM de la gráfica G es eficiente, por otro lado también existen casos —por ejemplo si G fuera una gráfica plana— en que el ciclo se puede ejecutara a lo más $|V|$ veces.

Lamentablemente, estas mejoras se ven afectadas por el tiempo que se invierte en ordenar las aristas con respecto a sus pesos, por lo cual se concluye que en el peor caso el Algoritmo de Kruskal tiene un tiempo de ejecución global de $O(|E|\log|E|)$.

Conclusiones

El objetivo de este trabajo fue realizar una recopilación de material didáctico sobre el análisis amortizados para cursos de Análisis de Algoritmos o Estructuras de Datos Avanzadas, en los cuales se requiere hacer un estudio más preciso tiempo de ejecución. Al termino de este trabajo pódemos hacer las siguientes observaciones.

1. Esta técnica no es sencilla. Para poder realizar el análisis amortizado de algún TDA en particular, se exige un amplio conocimiento del TDA, pues siempre se pretende medir, aquella *característica* que va cambiando a lo largo de una secuencia de operaciones.

   Por ejemplo, para realizar el análisis amortizado de las Colas Binomiales se observó que una característica útil para medir el costo de las operaciones es el número de árboles que se van generando en el bosque conforme se van realizando las operaciones, una vez localizada esta característica se procede finalmente a medirla con cualquiera de las técnicas mencionadas.

   También es necesario tener en cuenta que tal característica debe facilitar la distribución de costos de las operaciones. Por ejemplo, esto se observa en el cambio del potencial. En caso de que esto no suceda puede ser que el análisis está mal planteado, lo cual podría generar cotas tan pesimistas —o más— como las cotas dadas por el peor caso.

2. Una ventaja que ofrece este análisis es que da una mejor apreciación de algunos TDA. Por ejemplo, los Skew Heaps bajo esta perspectiva ofrecen cotas de ejecución tan "buenas" como para ser consideradas como una opción para ser utilizadas de la misma manera que las Colas Binomiales o Heaps. Teniendo la ventaja adicional de que es sencillo implementarlas pues no hay que revisar ninguna condición de balance para los árboles que se forman.

   Otro ejemplo es el TDA Conjuntos Ajenos que en sentido amortizado ofrece una cota lineal, ofreciendo algoritmos veloces y también resultan ser sencillos de implementar.

3. Esta técnica es muy poderosa. Ofrece una forma puntual de atacar el problema de calcular las cotas de ejecución de un algoritmo. Hay algoritmos que presentan de modo "natural" la característica a medir para obtener su complejidad.

   Esto se observa en el ejemplo de la pila con POP múltiple, en el cual el tamaño del stak es la medida de su costo de ejecución. Un ejemplo más elaborado serían las Colas Binomiales.

Apéndice A

Como se vió en el Capítulo 5, el tiempo de ejecución del TDA Conjuntos Ajenos combinando las estrategias de Unión por Tamaño o por Rango, así como las heurísticas de Reducción de Trayectorias es de $O(m, \alpha(m, n))$, para n conjuntos ajenos iniciales y m operaciones sobre los conjuntos ajenos. En este apéndice veremos cual es la definición formal de la función de Ackerman y su crecimiento explosivo, así como su función inversa que es denominada α cuyo crecimiento es lento; ambas se vuelven las herramientas esenciales para poder determinar las cotas amortizadas del TDA Conjuntos Ajenos. En este sentido, la cota de ejecución más sencilla, que es $O(m \log^* n)$, se obtiene también de la definición de la función de Ackerman en su versión más simple bajo el supuesto de que $m = \Omega(n)$.

6.2.2 La función de Ackerman

Para poder entender la función de Ackerman y su inversa, necesitamos algo más de notación. Primero definimos la notación para expresar la aplicación de una función exponencial en forma iterada: para cualquier entero i, $i \geq 0$, la expresión que determinamos con la función $g(i)$ se define recursivamente como:

$$g(i) = \begin{cases} 2^1 & \text{Si } i = 0 \\ 2^2 & \text{Si } i = 1 \\ 2^{g(i-1)} & \text{Si } i > 1 \end{cases}$$

Intuitivamente, el parámetro i indica "el tamaño de la pila de números 2" que hay que poner sobre el exponente. Por ejemplo:

$$g(j) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}\}j}, \quad \forall \ j \geq 1.$$

Ahora definiremos la contraparte de la noción anterior. Sea $log^{(i)} n$ para cualquier entero i, $i \geq 0$, una función definida recursivamente como:

$$\log^{(i)} n = \begin{cases} n & \text{Si } i = 0, \\ \log(\log^{(i-1)} n) & \text{Si } i > 1 \text{ y } \log^{(i-1)} > 0, \\ indefinida & \text{Si } i > 0 \text{ y } \log^{(i-1)} n \leq 0 \text{ ó } \log^{(i-1)} n \text{ está indefinida} \end{cases}$$

Observemos la notación $\log^{(i)} n$ que es la función logaritmo aplicado i veces en una sucesión comenzando con el argumento n; es una noción diferente a lo que indica la

notación $\log^i n$ la cual significa, el logaritmo de n elevado a la i-ésima potencia. Con lo anterior finalmente definimos el logaritmo iterado como:

$$\log^* n = \min\{i \geq 0 \mid \log^{(i)} n \leq 1\}$$

Con lo cual observamos que la función $\log^*$ es esencialmente la función inversa de la exponencial iterada.

Ahora ya estamos listos para mostrar la definición de la *función de Ackerman*, la cual se define recursivamente de la siguiente forma, para cualesquiera enteros i, j, $i, j \geq 0$:

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

A continuación se muestran los valores de la función de Ackerman para valores pequeños de i y j.

$A(i,j)$	$j=0$	$j=1$	$j=2$
$i=0$	1	2	3
$i=1$	2	3	4
$i=2$	3	5	7
$i=3$	5	13	29
$i=4$	13	65,533	$2^{65,533}-3$
$i=5$	65,533	$A(4,65533)$	$A(4,A(4,65533))$
$i=6$	$A(4,65533)$	$A(5,A(4,65533))$	$A(5,A(6,1))$

$A(i,j)$	$j=3$	$j=4$	$j=5$
$i=0$	4	5	6
$i=1$	5	6	7
$i=2$	9	11	13
$i=3$	61	125	253
$i=4$	$2^{2^{65533}}-3$	$A(3,2^{65,533}-3)$	A(3,A(4,4))
$i=5$	$A(4,A(5,2))$	$A(4,A(5,3))$	$A(4,A(5,4))$
$i=6$	$A(5,A(6,2))$	$A(5,A(6,3))$	$A(5,A(6,4))$

La Figura 6.8, muestra esquemáticamente que la función de Ackerman tiene un crecimiento explosivo. La primera columna muestra la exponencial iterada con los números del renglón i, lo cual presenta ya un rápido crecimiento. El segundo renglón consiste de el extenso espacio del subconjunto de los renglones formados en la columna anterior; así va desarrollándose conforme la recursión va avanzando.

Ahora bien, se define la función inversa de Ackerman como:

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

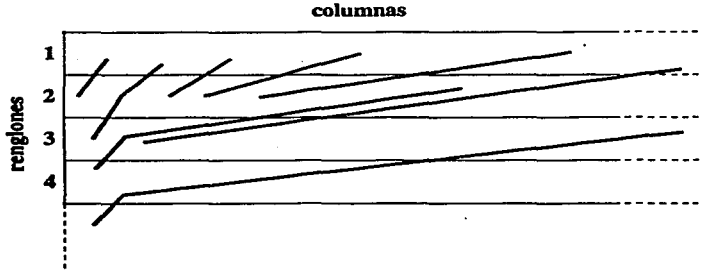


Figura 6.8:

Una vez vistas nuestras funciones, consideremos lo siguiente: si fijamos el valor de n entonces conforme m se incrementa la función $\alpha(m,n)$ se vuelve monótonamente decreciente. De este modo, $\lfloor m/n \rfloor$ es monótonamente creciente conforme el valor de m se incrementa. En consecuencia, desde que se fija el valor de n, los valores mínimos de i deben ser monótonamente decrecientes, para que $A(i, \lfloor m/n \rfloor)$ se mantenga por arriba de $\log n$. Esta propiedad en particular es la que corresponde con nuestra intuición sobre los bosques generados por las operaciones elementales del TDA Conjuntos Ajenos, cuando se les aplica alguna de las estrategias de Reducción de Trayectorias: para un número fijo n de distintos elementos, conforme el número m de operaciones se incrementa, nosotros esperamos que la longitud promedio de las trayectorias decrezca debido a las estrategias de reducción de trayectorias. Entonces, si ejecutamos m operaciones en un tiempo de $O(m\alpha(m,n))$, el tiempo promedio de operación es $O(\alpha(m,n))$, lo cual es monótonamente decreciente conforme m se incrementa.

Por las propiedades de la función de Ackerman y su inversa, se tiene que, para cualquier propósito práctico $\alpha(m,n) < 4$. Primero notemos que el valor de $\lfloor m/n \rfloor$ es al menos 1, esto bajo el supuesto de que $m \geq n$. Ya que la función es estrictamente creciente con cada argumento, $\lfloor m/n \rfloor \geq 1$, lo cual implica que $A(i, \lfloor m/n \rfloor) \geq A(i,1)$. En particular:

$$A(4, \lfloor m/n \rfloor) \geq A(4,1).$$

Pero también tenemos que:

$$A(4,1) = A(3,2) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}\}16}.$$

Resulta que esta cantidad es mayor que el número de átomos en el universo visible, lo cual es alrededor de 10^{80}. Estos son valores de n imprácticos y muy grandes para los cuales se cumple que $A(4,1) \geq \log n$ y así $\alpha(m,n) \leq 4$ para cualquier próposito práctico. Notemos que la cota $O(m \log^* n)$ es un poco más débil que la cota $O(m\alpha(m,n))$. Entonces tenemos que $\log^* 65536 = 4$ y $\log^* 2^{65536} = 5$, así que $log^* \leq 5$ para cualquier propósito práctico.

Bibliografía

[1] **Cormen, T.H.,C.L. Leiserson, y R.L. Riverst,** *Introduction to Algorithms.* MIT Press ans McGrawHill, New York. 1990.

[2] **AHUJA, R. K. and MAGNANTI, T. L. and ORLIN, J. B.,**
Networks Flows: Theory, Algorithms, and Applications, Prentice Hall, 1993.

[3] **Brown M.R.**
Implementation and Analysis of Binomial Queue Algorithms,
SIAM Journal on Computing 7(1978), 298-319.

[4] **KINGSTON, J.H.,**
Algorithms and Data Structures: Design, Correctness and Analysis,
Addison Wesley Co., Australia, 1990.

[5] **MANBER, U.**
Introduction to Algorithms. A Creative Approach, Addison-Wesley Publishing Co., 1st, USA, 1989.

[6] **LEEUWEN, J. V. and TARJAN, R. E.**
Worst-Case Analysis of Set Union Algorithms, Journal of ACM, 31(2):245-281, 1984.

[7] **R. E. Tarjan,**
Amortized Computational Complexity,
SIAM J. Alg. Disc. Math., No. 2 (1985)
pp. 306-318.

[8] **R. E. Tarjan,**
Data Structures and Network Algorithms,
CBMS-CNFS. Regional Conference
Series in Applied Math. Society for Industrial and Applied Math., USA, Fifth edition, 1988.

[9] **Weiss, M.A.,**
Data Structures and Algorithm Analysis in C++,
The Benjamin/Cummings Publishing Company, Inc, 1994.

[10] **Weiss, M.A.,**
Algorithms, Data Structures and Problem
Solving with C++,
The Benjamin/Cummings Publishing Company, Inc, 1994.