00327
4

UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

INTRODUCCIÓN A ALGORITMOS PARALELOS
NOTAS DE CLASE

EJEMPLAR UNICO

T E S I S

QUE PARA OBTENER EL TÍTULO DE
LICENCIADO EN CIENCIAS
DE LA COMPUTACIÓN
P R E S E N T A
DANIEL JUAREZ MELCHOR

FACULTAD DE CIENCIAS
UNAM

DIRECTOR DE TESIS:
M. en I. MARIA DE LUZ GASCA SOTO

FACULTAD DE CIENCIAS
SECCION 2003

UNIVERSIDAD NACIONAL
AVENIDA U
MEXICO

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

Comunico a usted que hemos revisado el trabajo escrito:

"Introducción a Algoritmos Paralelos. Notas de Clase"

realizado por

Daniel Juárez Melchor

Con número de cuenta 093158330, **quién 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. MARIA DE LUZ GASCA SOTO
Propietario	DR. LUIS BERNARDO MORALES MENDOZA
Propietario	M. en C. ENRIQUE CRUZ MARTINEZ
Suplente	M. en C. JOSE DE JESUS GALAVIZ CASAS
Suplente	DRA. AMPARO LOPEZ GAONA

Consejo Departamental de Matemáticas

Dra. Amparo López Gaona

Agradecimientos

A la Universidad Nacional Autónoma de México y a la Facultad de Ciencias, por darme la oportunidad de estudiar.

A mis sinodales Luis Morales, Enrique Cruz, José Galaviz, Amparo Lopez, Lucy Gasca, por compartir sus experiencias y conocimientos.

A mis amigos Monty, La bruja, Bere, Violeta, Jay, Ray, Isma, Petite, Pollita, Buguito, Mamichi, El Choco, Piecito, Betina, Charly,Chio, Pablo Honey y a todos aquellos que en este fragmento no nombré, por su invaluable apoyo a lo largo de la carrera

Dedicada a:

Mis padres Sixto y Concepción

Mis hermanos César Alejandro y José Alberto

Mi esposa Claudia

Índice general

Índice de figuras

PAGINACION

DISCONTINUA

Introducción

El campo de Computación en Paralelo se ha expandido muy rápido en relación con otras áreas de las Ciencias de la Computación. Existen diversos tipos de computadoras en paralelo en operación, poseen de 2 a 65,536 procesadores. Las diferencias entre las diversas máquinas paralelas existentes son enormes. No podemos adoptar un modelo genérico de computación que se adapte a todas las computadoras paralelas. Diseñar algoritmos paralelos, analizarlos y probar que son correctos es mucho más difícil que hacerlo para algoritmos secuenciales.

En este material no es posible cubrir todas las áreas de la computación paralela. Presentamos una variedad de ejemplos usando diferentes modelos de computación y diversas técnicas. Intentamos dar una panorámica de los algoritmos paralelos explorando las dificultades en su diseño. Empezamos con algunas características comunes, describimos brevemente el principal modelo de computación paralela que usaremos durante el curso y finalmente revisaremos algunos ejemplos de algoritmos y técnicas.

Las principales medidas de complejidad para algoritmos secuenciales son el desempeño computacional (número de operaciones elementales) y el espacio de almacenamiento. Estas medidas también son importantes para los algoritmos paralelos, pero debemos además preocuparnos por otros recursos. Un importante recurso es el número de procesadores. Existen problemas que son inherentemente secuenciales y no es posible diseñar un algoritmo paralelo para ellos, aun si hubiese un número infinito de procesadores disponibles. Sin embargo, para la mayoría de los problemas, pueden diseñarse algoritmos paralelos.

Entre más procesadores usemos, hasta cierto límite, nuestro algoritmo paralelo será más rápido. Es importante estudiar las limitaciones de los algoritmos paralelos y ser capaces de caracterizar los problemas, sobre todo los que resultan tener soluciones paralelas muy rápidas. Ya que el número de procesadores es limitado es muy importante usarlos de la manera más eficiente.

Un importante tema es la comunicación entre los procesadores. Generalmente, toma más tiempo intercambiar datos entre dos procesadores que ejecutar una simple operación sobre los datos. Además mientras algunos procesadores estén muy cerca entre sí, otros pueden estar muy alejados. Por lo tanto, resulta muy importante minimizar la comuni-

cación y organizarla de manera efectiva.

Otro aspecto importante es la sincronización, la cual es el mayor problema para los algoritmos paralelos que corren sobre máquinas independientes conectadas vía una red de comunicación. Estos algoritmos son usualmente llamados **Algoritmos Distribuidos.**

Algunos modelos de computación paralela restringen a que todos los procesadores ejecuten exactamente la misma instrucción en cada paso. Las computadoras que siguen esta restricción son llamadas **Máquinas SIMD**, *Single-Instruction Multiple-Data*. Las computadoras en paralelo en las cuales cada procesador ejecuta diferentes procesos son llamadas **Máquinas MIMD**, *Multiple-Instruction Multiple-Data*. A menos que se especifique explícitamente lo contrario, asumiremos que estamos trabajando con este último modelo.

El presente trabajo está organizado de la siguiente manera:

En el Capítulo 2 se describen brevemente los modelos de computación en paralelo, definiciones generales como el Desempeño computacional, la Aceleración del algoritmo y su Eficiencia.

El Capítulo 3 trata los modelos para máquinas de memoria compartida, es decir, una computadora con múltiples procesadores y una unidad de memoria compartida. estudiaremos la manera en que manipulan los conflictos de memoria como lectura y escritura exclusiva, lectura y escritura concurrente y la lectura concurrente y escritura exclusiva.

Estudiamos en el Capítulo 4 los algoritmos para redes. La conexión de redes puede modelarse con gráficas, donde los vertices representan los procesadores y las aristas sus conexiones. Se han sugerido diversas gráficas como modelos de redes interconectadas. Entre ellos los anillos, árboles binarios, estrellas, mallas, entre otras.

El Capítulo 5 estudia la computacion sistolica que simula la producción en linea, donde cada procesador desempeñ una operación simple sobre los datos recibidos de la operación anterior y mueve el resultado al siguiente procesador. Analizaremos su eficiencia en *hardware*, velocidad, número de accesos a memoria y su flexibilidad para aplicarse a diversos problemas.

En el Capítulo 6 introduciremos algunas técncias básicas asi como su aplícación a problemas. La tarea de diseñar algoritmos paralelos difiere de los algoritmos secuenciales. La falta de una metodología es compensada por una colección de técncias y paradigmas efectivas en el manejos de problemas.

Las listas y los árboles se mestran en ellCapítulo 7. Estudiaremos algunos problemas

sobre procesamiento de listas y cálculos de funciones en árboles y los atacaremos con las técnicas Contracción de árboles y la Ruta de Euler.

En el Capítulo 8 presentamos ejemplos especificos como la búsqueda de elementos en una tabla, asi como su ordenamiento. Mostramos algoritmos óptimos basados en la Búsqueda en paralelo, partición, encadenamiento y divide y venceras.

El Capítulo 9 trata las gráficas. Estas pueden ser usadas para representar la comunicación entre sistemas, redes eléctricas, entre otras. Además pueden ser usadas para desarrollar estructuras de datos eficientes.

El Capítulo 10 contiene a la Geometría Computacional. Esta estudia el diseño eficiente de algoritmos que manipulan problemas relacionados con el espacio Euclidiano. Retomamos el algoritmo divide vencerás para calcular la cubierta convexa de un conjunto de puntos.

Y por último en el Capítulo 11 tratamos asuntos importantes como las Cadenas. Analizamos el problema de la búsqueda de ocurrencias de patrones, encontrado en la edición de texto, comparación de cadenas y algunas propiedades periódicas de las cadenas.

Capítulo 1

Modelos de computación en paralelo

Describiremos algunos modelos de computación en paralelo, enfatizando los que usaremos en este material. Incluimos en este capítulo discusiones y definiciones generales que se aplican en diversos modelos. En las siguientes secciones revisaremos un tipo de modelo e incluimos una descripción más detallada del mismo, así como ejemplos de algoritmos en tal modelo.

Denotamos al **Desempeño Computacional** de un algoritmo paralelo como $T(n,p)$, donde n es el tamaño del ejemplar, número de datos, y p representa el número de procesadores.

Al radio

$$S(p) = \frac{T(n,1)}{T(n,p)}$$

se le denomina la **Aceleración del Algoritmo.** Un algoritmo paralelo es más efectivo cuando $S(p) = p$, en tal caso decimos que el algoritmo tiene una **Aceleración Perfecta.** El valor de $T(n,1)$ deberá ser tomado del mejor algoritmo secuencial conocido. Una importante medida sobre la utilización de los procesadores es la **Eficiencia** de un algoritmo paralelo, la cual se define como:

$$E(n,p) = \frac{S(p)}{p} = \frac{T(n,1)}{p \cdot T(n,p)}$$

La eficiencia es el radio del tiempo usado por un procesador, con un algoritmo secuencial, y el tiempo total usado por p procesadores. La eficiencia indica el porcentaje del tiempo no utilizado por los procesadores, comparado con un algoritmo secuencial. Si la eficiencia es $E(n,p) = 1$, entonces la cantidad de trabajo realizada por todos los procesadores a través de la ejecución del algoritmo es igual a la cantidad de trabajo requerida por un algoritmo secuencial. En este caso, obtenemos un uso óptimo de los procesadores.

En realidad, es muy raro obtener eficiencia óptima, ya que la mayoría de las veces los algoritmos paralelos introducen algunas instrucciones o procesos que no son requeridos en el correspondiente algoritmo secuencial. Una de nuestras metas es maximizar la eficiencia.

Cuando diseñamos un algoritmo paralelo podemos fijar p, de acuerdo al número de procesadores disponibles, y tratar de minimizar $T(n,p)$. Pero haciendo lo que potencialmente requeriría un nuevo algoritmo si el número de procesadores cambia. Es deseable encontrar un algoritmo que funcione para tantos valores de p como sea posible.

Ahora discutiremos como transcribir un algoritmo que trabaje con ciertos valores de p a otro que trabaje con un número más pequeño de valores para p, sin cambiar significativamente la eficiencia. En general, podemos modificar un algoritmo con desempeño computacional $T(n,p) = X$ a uno con $T(n,p) = k \cdot X$ para una constante $k > 1$. En otras palabras, podemos usar un factor de k menos procesadores trabajando para un factor de k más tiempo. Construimos el algoritmo modificado reemplazando cada paso del algoritmo original con k pasos, en los cuales un procesador emula la ejecución de un paso de k procesadores. Este principio es llamado Principio de Paralelismo Plegado o Principio de Plegamiento, *parallelism folding principle*. Lamentablemente, este principio no puede aplicarse en todas las situaciones.

Por ejemplo, si k no divide a p, el algoritmo puede depender de un cierto patrón de interconexión entre los procesadores o el algoritmo puede depender de una decisión concerniente a cuál procesador a emular podría requerir tiempo de cálculo. Sin embargo, este principio es muy útil. Muestra que podemos reducir el número de procesadores sin cambiar significativamente la eficiencia. Si, por ejemplo, el algoritmo original, diseñado para una p grande, exhibe buena aceleración entonces podemos obtener algoritmos que se ejecutan casi con la misma aceleración para cualquier valor pequeño de p.

Por lo tanto, deberíamos tratar de obtener la mejor aceleración con el máximo número de procesadores, cuidando que la eficiencia sea buena, es decir que la eficiencia sea muy cercana a 1. Entonces, si tenemos pocos procesadores podemos usar el mismo algoritmo. Por otro lado, algoritmos paralelos con poca eficiencia son útiles solo para un número grande de procesadores.

Por ejemplo, suponga que tenemos un algoritmo con $T(n,1) = n$ y $T(n,n) = \log_2(n)$, lo cual implica que la aceleración es $S(n) = (n/\log_2(n))$, la cual resulta impresionante, y la eficiencia es $E(n,n) = (1/\log_2(n))$. Suponga ahora que el número disponible de procesadores es $p = 256$ y $n = 1024$. El desempeño computacional del algoritmo paralelo es $T(1024,256) = 4$, $\log_2(1024) = 4$, $log_2(210) = 40$, asumiendo que el plegamiento es posible. La aceleración es de casi 25 sobre el algoritmo secuencial. Por otro lado, si $p = 16$, entonces el tiempo de ejecución es 640, lo cual no es una buena aceleración.

Varios modelos de computación paralela difieren principalmente de la manera en que los procesadores se comunican y sincronizan. Consideraremos sólo modelos que asumen total sincronización y nos concentraremos en diferentes paradigmas de comunicación. Los modelos de memoria compartida de acceso aleatorio, *random access shared memory*, de tal manera que cualquier procesador pueda tener acceso a cualquier variable con costo unitario, constante. Este supuesto es irreal, pero es una primera aproximación buena. Los modelos de memoria compartida difieren de la manera en que manipulan sus conflictos de acceso, discutiremos diferentes alternativas en el Capítulo 3.

La memoria compartida es usualmente la forma más fácil de modelar la comunicación, pero resulta ser el modelo más difícil para implantar en hardware. Otros modelos asumen que los procesadores están conectados a través de una red, llamada también red de interconexión.

Una red de interconexión puede ser representada por una gráfica, donde los vértices corresponden a los procesadores y dos vértices están conectados si los procesadores correspondientes tienen una liga directa entre ellos. Cada procesador, usualmente tiene memoria local que puede ser consultada rápidamente. La comunicación se realiza por medio de mensajes, los cuales pueden viajar por diferentes ligas hasta llegar a su destino. Aquí la rapidez de la comunicación depende de la distancia entre los procesadores que están comunicándose. La Figura 2.1 muestra un ejemplo de una red de interconexión.

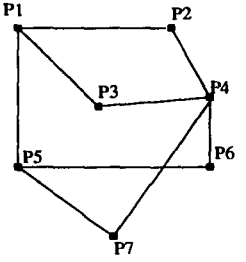


Figura 1.1: Red de Interconexión

Diferentes clases o familias de gráficas han sido estudiadas como redes de interconexión mencionaremos las más populares en el Capítulo 4. Las computadoras en paralelo basadas en este modelo son algunas veces denominadas Multicomputadoras.

Otro modelo de cómputo paralelo es la computación sistólica. Una arquitectura sistóli-

ca se parece a una cadena de montaje donde los datos se mueven a través de los procesadores de un modo rítmico y operaciones simples son efectuadas sobre ellos. En vez de tener acceso a memoria compartida, o no compartida, los procesadores reciben las entradas de sus vecinos, operando sobre ellas y pasándolas. Los algoritmos sistólicos son presentados en el Capítulo 5.

Un modelo teórico básico que usaremos para propósitos ilustrativos es el **circuito**. Un circuito es una gráfica acíclica dirigida en la cual los vértices corresponden a operaciones simples y las aristas muestran el movimiento de los operandos. Por ejemplo, un circuito lógico es uno en el cual todos los in-grados son a lo más dos y todas las operaciones son lógicas: and, or y not. Hay un vértice designado para la entrada, con in-grado 0, y para la salida, con ex-grado 0. La profundidad de un circuito es la trayectoria más larga de la entrada a la salida. La profundidad corresponde al tiempo de ejecución paralelo. La Figura 2.2 muestra un circuito.

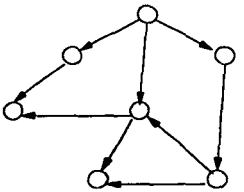


Figura 1.2: Circuito

Capítulo 2

Algoritmos para máquinas de memoria compartida

Una computadora de memoria compartida consiste de diversos procesadores y una unidad de memoria compartida, la Figura 3.1 ilustra este hecho.

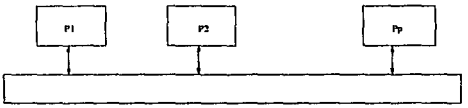


Figura 2.1: Memoria Compartida

Asumiremos que los algoritmos son totalmente sincronizados. Asumimos que los cálculos consisten de *pasos*. En cada paso, cada procesador ejecuta una operación sobre los datos, los lee o escribe usando la memoria compartida. En la práctica cada procesador puede tener memoria local, pero aquí nosotros asumimos que toda la memoria es global.

Los modelos de memoria compartida difieren en la manera en que manipulan los conflictos con la memoria, los cuales son:

- **EREW**, *Exclusive Read Exclusive Write*. No permite que más de un procesador tenga acceso a la misma localidad de memoria a la vez.
- **CREW**, *Concurrent Read Exclusive Write*. Permite que diversos procesadores lean de la misma localidad de memoria al mismo tiempo, pero únicamente un procesador puede escribir en tal localidad.
- **CRCW**, *Concurrent Read Concurrent Write*. El modelo no tiene restricciones sobre los conflictos de memoria.

Los modelos **EREW** y **CREW** están bien definidos, pero no es claro cual es el resultado de dos procesadores escribiendo al mismo tiempo sobre la misma localidad. Existen diversas alternativas para manipular las escrituras concurrentes. El modelo más débil es el **CRCW**, permite que diversos procesadores escriban en la misma localidad al mismo tiempo sólo si todos ellos escriben lo mismo. Si dos procesadores intentan escribir diferentes valores en la misma localidad al mismo instante, el algoritmo falla. Usaremos modelo **CRCW** con estas condiciones en este curso; revisaremos, en la Subsección 3.2, que tal característica es muy poderosa. Una alternativa es asumir que los procesadores están etiquetados y que cuando diversos procesadores escriben a la misma localidad al mismo tiempo, el procesador con la etiqueta más alta tiene éxito, logra escribir. Otra posibilidad es asumir que un procesador arbitrario tenga éxito.

2.1. Suma Paralela

Empezamos con un ejemplo sencillo de algoritmos en paralelo, desarrollado por inducción.

Problema: Encontrar la Suma de dos números binarios de n bits

El algoritmo secuencial clásico inicia con el bit menos significativo y el posible acarreo. No sabemos que va a pasar en el i-ésimo paso hasta que el paso $(i-1)$ sea ejecutado, ya que podría o no tener acarreo. Observemos el siguiente ejemplo mostrado en la Figura 3.2:

```
  1 0 1 0 1 0 1 0 1 0 1 1 1 0
  1 1 1 0 0 1 1 0 1 1 0 1 1 1        n=16
-----------------------------
1 1 0 0 1 0 0 0 1 1 0 0 0 1 0 0 1
```

Figura 2.2: Suma Binaria

Usaremos inducción sobre n. No nos ayuda mucho ir de $(n-1)$ a n, ya que esto implica un algoritmo secuencial iterativo. La estrategia **Divide y Vencerás** es muy exitosa para algoritmos en paralelo, ya que es posible resolver todas las pequeñas partes en paralelo. Supóngase n es potencia de dos: $n = 2^k$, entonces $k = \log_2(n)$. Es posible subdividir en dos problemas de tamaño $(n/2)$ y encontrar la suma de dos parejas en paralelo... pero aún tenemos el problema del acarreo. Si la suma del par menos significativo tiene acarreo, cambiamos la suma del par más significativo.

La observación clave aquí es que existen sólo dos posibilidades: hay o no acarreo.

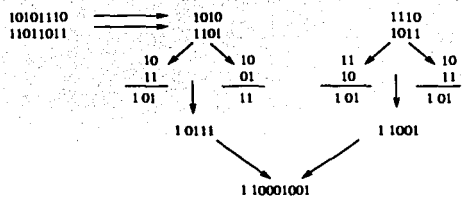


Figura 2.3: Continuación de la suma binaria

Entonces podemos usar una hipótesis de inducción directa que incluya ambos casos. El problema modificado es encontrar la suma de dos números con y sin acarreo inicial. Ahora, suponga que resolvemos el problema modificado para ambos pares. Obtenemos cuatro números:

- **L** y **Lc** : Suma del par menos significativo sin y con acarreo, respectivamente
- **R** y **Rc** : Suma del par más significativo sin y con acarreo, respectivamente.

Para cada una de estas sumas también buscamos si se generan acarreos. La suma final S, sin acarreo inicial, es L y ($RoRc$) dependiendo si L tiene o no acarreo. La suma final Sc, es la misma S, excepto que L es reemplazada por Lc.

Resolvemos el problema de tamaño n, dividiendolo en dos subproblemas de tamaño $(n/2)$. Aplicamos este algoritmo de manera recursiva hasta que el tamaño de cada uno de los ejemplares sea fácilmente manipulable, como se muestra en la Figura 3.3. Asumimos que los procesadores pueden tener acceso directo a diferentes bits de manera independiente. Obtenemos la relación de recurrencia:

$$T(n,n) = T\left(\frac{n}{2}, \frac{n}{2}\right) + O(1)(T(n,n) = O(\log_2 n).$$

Como los subproblemas son totalmente independientes, podemos asumir que usamos el modelo **EREW**. Este algoritmo no es el mejor, pero es un buen ejemplo didáctico.

2.2. Algoritmo para encontrar el máximo elemento

Problema: Dado un arreglo de n elementos distintos, digamos enteros, encontrar el máximo elemento.

Resolveremos este problema para los modelos de memoria compartida **EREW** y **CRCW**. El algoritmo para ambos modelos usa técnicas que son usadas por muchos otros problemas.

2.2.1. Usando el Modelo EREW

Un algoritmo secuencial directo para encontrar el máximo requiere $(n-1)$ comparaciones. Podemos pensar que cada comparación es un juego entre dos números donde el mayor gana. Encontrar el máximo es equivalente a efectuar un torneo, donde el ganador es el mayor de todo el conjunto. Una manera eficiente de realizar un torneo en paralelo es usando un árbol. Los jugadores se dividen en parejas, en cada ronda.

Después de cada ronda, los ganadores se vuelven a dividir en parejas y así hasta obtener una última pareja y, al final, un único ganador. Si hay n jugadores, $n = (2 * E) - k$, el número total de rondas es $k = \log_2(n)$. Podemos obtener un algoritmo paralelo a partir de los torneos, asignando un procesador a cada juego, como si el procesador fuese el arbitro del juego. Debemos suponer entonces que cada procesador conoce a los dos competidores. Esto puede realizarse poniendo a cada ganador del juego en la posición del mayor índice de los dos competidores. De esta forma, si el juego es entre el jugador x_i y el x_j, con $j > i$, entonces el ganador va a la posición j.

En la primera ronda, el procesador P_i compara $x[2i-1]$ con $x[2i]$, para i, $1 \leq i \leq (n/2)$ y los intercambia de ser necesario. En la segunda ronda, el procesador P_i compara $x[4i-2]$ con $x[4i]$, con i tal que $1 \leq i \leq (n/4)$ ...y así. Como cada número es involucrado en un juego a la vez, el modelo **EREW** es suficiente. El tiempo de ejecución claramente es $O(\log_2(n))$. Minimicemos el número de procesadores.

El algoritmo requiere $(n/2)$ procesadores, además tenemos que $T(n,(n/2)) = \log_2(n)$ y $T(n,1) = (n-1)$, entonces la eficiencia es:

$$E\left(n, \frac{n}{2}\right) = \frac{1}{(\log_2(n))}.$$

Si $(n-2)$ procesadores están siempre disponibles, entonces el algoritmo es simple y eficiente. Con algunos cambios, sin embargo, podemos alcanzar un tiempo de ejecución de $O(\log_2(n))$ con eficiencia $O(1)$.

El número total de comparaciones requeridas en este algoritmo es $(n-1)$, el mismo que para un algoritmo secuencial. La razón de la baja eficiencia es que muchos procesadores quedan sin utilizarse en rondas siguientes. Podemos evitar esto reduciendo el número de procesadores y efectuando un almacenamiento balanceado de la siguiente manera: Usamos ahora $(n/\log_2(n))$ procesadores. Dividimos la entrada en grupos de tamaño $(n/\log_2(n))$,

cada grupo tiene casi $(\log_2(n))$ elementos y asignamos un grupo a cada procesador. En la primera fase, cada procesador usa el algoritmo secuencial para encontrar el máximo, requerirá $(\log_2(n))$ pasos. Resta encontrar el máximo entre los máximos encontrados que son $(n/\log_2(n))$.

El desempeño computacional de este algoritmo, asumiendo que n es potencia de 2, es:

$$T\left(n, \frac{n}{(\log_2(n))}\right) = 2 * log_2(n);$$

y la eficiencia es: $E(n,p) = (1/2)$. A continuación formalizamos esta idea.

Definición. Un algoritmo paralelo es **estático** si la asignación de procesadores está predefinida. Es decir, si conocemos a priori, para cada paso i del algoritmo y para cada procesador P_j , la operación y los operandos que el procesador P_j usa en el paso i.

El algoritmo para encontrar el máximo es estático, ya que sabemos que juegos se van a llevar a cabo, es decir, los juegos están pre-determinados.

Lema de Brent. Si existe un algoritmo paralelo estático, en el modelo **EREW**, con desempeño $T(n,p) = O(t)$, tal que el número total de pasos sobre todos los procesadores es s , entonces existe un algoritmo paralelo estático con desepeño:

$$T\left(n, \frac{s}{t}\right) = O(t)$$

Demostracion: Sea $a_i (i = 1, 2, \ldots, t)$ el número total de pasos ejecutados por todos los procesadores, en el paso i del algoritmo. Tenemos que $\sum a_i = s$.

Si $a_i \leq (s/t)$, entonces hay suficientes procesadores para ejecutar el paso i y no tenemos que cambiarlo.

En otro caso, si $a_i \geq (s/t)$ reemplazamos el paso i con $(a_i)/[(s/t)]$ pasos en el cual están disponibles (s/t) procesadores emulando los pasos tomados por los p procesadores en el algoritmo original, siguiendo el principio plegamiento. El número total de pasos es:

$$\sum \frac{a_i}{s/t} \leq \sum \frac{t \cdot a_i}{s} = t + \frac{t}{s} \cdot \sum a_i = 2t.$$

De aquí, el desempeño computacional del algoritmo modificado es $O(t)$. □

Obsérvese que si s es igual a la complejidad del algoritmo secuencial para el problema, entonces el algoritmo modificado tiene una eficiencia de $O(1)$.

Brent muestra que, en algunos casos, la eficiencia del algoritmo paralelo depende principalmente del radio entre el número total de operaciones efectuadas por todos los procesadores y el tiempo de ejecución del algoritmo secuencial .

Necesitamos la restricción del algoritmo estático ya que debemos saber que procesadores vamos a emular. Aunque también es válido para algoritmos no estáticos, suponiendo que la emulación puede ser hecha de manera rápida y eficiente.

Un caso donde el Lema no es válido. Suponga que hay n procesadores y n elementos. Después del primer paso, algunos de los procesadores "deciden", basados en los resultados del primer paso, retirarse. Lo mismo pasa después del segundo, tercero y así sucesivamente. Este algoritmo es similar al algoritmo del torneo, excepto que, en este caso, no sabemos cuáles procesadores se retirarán. Si tratamos de emular los procesadores restantes, digamos después del primer paso, necesitamos conocer cuáles de ellos están aún activos. Pero requieren tiempo para calcularlos.

Usando el Modelo CRCW

Pareciera que el primer algoritmo paralelo no puede encontrar al máximo elemento en menos que $\log_2(n)$ pasos si sólo las comparaciones son usadas, pero no es así. El siguiente algoritmo, cuyo tiempo de ejecución en paralelo es $O(1)$, ilustra la potencia de la escritura concurrente. Usamos una versión de escritura concurrente en la cual dos o más procesadores pueden escribir en la misma localidad al mismo tiempo cuando ellos escriban la misma cosa.

Requerimos $\lceil n \cdot (n-1) \rceil / 2$ procesadores, al procesador P_{ij} se le asigna la pareja de elementos $\{i, j\}$. Además, reservamos una variable compartida v_i para cada x_i, con valor inicial $v_i = 1$, para toda i. En el primer paso, cada procesador compara sus dos elementos y escribe 0 en la v_i asociada al x_i más pequeño, ya que solamente un elemento es mayor que todos los otros, únicamente una v_i conservará el 1. En el segundo paso, el procesador asociado con el ganador puede determinar que es el ganador y anunciar este hecho. Este algoritmo requiere de dos pasos, independientemente del tamaño de n. Su eficiencia, sin embargo, es muy pobre ya que necesita $O(n^2)$ procesadores. A este algoritmo le llamamos **Algoritmo de Dos Pasos**.

Usando EREW

Podemos mejorar el algoritmo de dos pasos usando un método en el modelo **EREW**. Dividimos la entrada en grupos de tal manera que podamos disponer de suficientes procesadores para encontrar el máximo de cada grupo en únicamente dos pasos. Como el número de candidatos decrece, el número disponible de procesadores, por candidato, se incrementa y el tamaño del grupo puede ser aumentado.

El algoritmo de dos pasos muestra que si el tamaño de un grupo es k entonces bastan

$[k \cdot (k-1)]/2$ procesadores para encontrar el máximo del grupo en tiempo constante. Asumimos que tenemos n procesadores y que n es potencia de 2.

En la primera ronda, el tamaño de cada grupo es 2, y el máximo de cada grupo puede ser encontrado en un paso. En la segunda ronda, solo quedan $(n/2)$ elementos y disponemos de n procesadores. Si fijamos el tamaño del grupo a 4, entonces tenemos $(n/8)$ grupos, teniendo disponibles 8 procesadores por grupo. Esto es suficiente, ya que $[4 \cdot (4-1)]/2 = 6$. En la tercera la tercera ronda,tenemos $(n/8)$ elementos resultantes, ahora, calcularemos el máximo tamaño de grupo que podemos permitirnos.

Si el tamaño del grupo es g, entonces el número del grupo será $n/(8 \cdot g)$ y entonces hay $8 \cdot g$ procesadores por grupo. Para usar el algoritmo de dos pasos necesitamos al menos $[g(g-1)]/2$ procesadores por grupo de tamaño g. Por lo tanto, debemos tener que:

$$\frac{g \cdot (g-1)}{2} < 8 \cdot g \Rightarrow g < 17,$$

es más simple usar $g = 16$. Resulta fácil verificar que el tamaño de cada grupo es el cuadrado del tamaño del grupo de la ronda anterior, obteniendo un algoritmo que requiere de $O(\log(\log(n)))$ rondas.

Aunque este algoritmo es un poco más lento que el algoritmo original de dos pasos, $O(\log(\log(n)))$ *vs* $O(1)$, su eficiencia es mucho mejor, $O(1/[\log(\log(n))])$ *vs* $O(1/n)$. Esta técnica es conocida como **Divide y Comprime** *(Divide and Crush)*, ya que divide la entrada en grupos del tamaño suficiente para poder comprimir con lotes de procesadores. Esta técnica no está limitada al modelo **CRCW**.

2.3. Prefijo paralelo

El problema del prefijo es muy importante, pues sirve como el principal constructor de bloques en el diseño de diversos algoritmos paralelos.

Sea " $\cdot$ " una operación arbitraria, binaria y asociativa llamada **producto**. Esta operación puede representar la adición, la multiplicación o el máximo entre dos números.

Problema: Dada una secuencia con n con números $(x_1, x_2, \ldots, x_n)$ calcular el producto $(x_1 \cdot x_2 \cdot \ldots \cdot x_k)$ para toda k tal que $1 \leq k \leq n$.

Denotemos por $Pr(i,j)$ al producto $(x_i \cdot x_{i+1} \cdot \ldots \cdot x_j)$. Nuestra meta es calcular $Pr(1,k)$ para toda k tal que $1 \leq k \leq n$. La versión secuencial es trivial, simplemente calculamos el prefijo en orden. El problema del prefijo paralelo no es fácil de resolver. La estrategia que usaremos es **Divide y Vencerás**. Asumimos que n es potencia de dos, $n = 2^s$.

Hipótesis de inducción. Sabemos cómo resolver el problema del prefijo en paralelo para $(n/2)$ elementos.

Caso base. $n = 1$, es trivial.

Paso inductivo. El algoritmo procede dividiendo la entrada a la mitad y resolviendo cada mitad por inducción. Así, obtenemos los valores:

$$Pr(1,k) \;\; y \;\; Pr\left(\frac{n}{(2+1)}, \frac{n}{(2+k)}\right)$$

para toda k tal que $1 \leq k \leq (n/2)$. Los valores de la primera mitad pueden ser usados directamente. Los valores $Pr(1,m)$ para $(n/2) \leq m \leq n$ pueden obtenerse aplicando la siguiente operación:

$$Pr(1,m) = Pr\left(1, \frac{n}{2}\right) \cdot Pr\left(\frac{n}{2+1}, m\right)$$

con $(n/2) \leq m \leq n$. Ambos términos son conocidos por inducción. La Figura 3.4 muestra el algoritmo PARALLEL_PREFIX_1, una implantación de este proceso. □

```
Parallel_Prefix_1
        // ENTRADA: X arreglo de n enteros entre 1 y n,
        // suponemos que n=2^k
        // SALIDA: x el i-esimo elemento que contiene al
        //i-esimo prefijo

main(){
        Return PP_1(1,n)
        }

        PP_1(L,R){
        If R-L == 1
        Then x(R) = x(L) x(R)
        Else m = (L+R)/2
                Do In Parallel
                PP_1(L,M)
                PP_1(m+1,R)
                For i = m+1 to R
                Do In Parallel
                        x(i) = x(m) x(R)
        }
```

Figura 2.4: ALGORITMO PARALLEL_PREFIX_1

Complejidad

La entrada se divide en dos conjuntos ajenos, en cada llamada recursiva. Ambos problemas pueden resolverse en paralelo con **EREW**. Si tenemos n procesadores, para el problema de tamaño n, entonces $(n/2)$ procesadores pueden resolver cada subproblema.

La combinación de pasos requiere $(n/2)$ que deben ejecutarse en paralelo, pero es necesario el modelo **CREW**, pues cada P_i debe tener acceso a $x(m)$. Aunque muchos procesadores deben leer $x(m)$ al mismo tiempo, ellos escriben en distintas localidades, así que **CRCW** no es necesario. Así, tenemos que:

$$T(n,n) = O(\log n), T(n,1) = O(n)$$

y entonces la eficiencia resulta ser:

$$E(n,n) = O\left(\frac{1}{\log n}\right).$$

Desafortunadamente, no podemos mejorar la eficiencia usando el Lema de Brent. El número total de pasos usados por el algoritmo es $O(n \cdot \log n)$, el gasto viene de la segunda llamada recursiva. Por lo tanto, si queremos mejorar la eficiencia, debemos mejorar el algoritmo para que el número de pasos sea reducido

Mejorando la eficiencia

El truco es usar la misma hipótesis de inducción, dividiendo la entrada de forma diferente. Suponga $n = 2^k$ y hay n procesadores. Sea $E = \{x_i; i \ \ es \ \ par\}$. Si encontramos el producto de todos los elementos de E, entonces encontramos el resto, los impares, fácilmente. Si $Pr(1, 2i)$ se conoce para toda $i, 1 \leq i \leq \frac{n}{2}$, entonces para cada prefijo impar $Pr(1, 2i+1)$ necesitamos calcular un producto más:

$$Pr(1, 2i+1) = Pr(1, 2i) \cdot x(2i+1).$$

Encontramos los prefijos de E en dos fases, primero calculamos en paralelo el producto: $x(2i-1) \cdot x(2i)$ con $1 \leq i \leq (n/2)$ y almacenamos el resultado en $x(2i)$. En otras palabras calculamos el producto de todos los elementos de E con sus vecinos a la izquierda. Entonces, en el segundo paso resolvemos el problema de tamaño $(n/2)$ para E por inducción. El resultado para cada $x(2i$ contiene el prefijo correcto, ya que cada $x(2i)$ incluye el producto con $x(2i-1)$. Si conocemos los prefijos de todos los índices pares sabremos cómo calcular los impares en un paso más, '!en paralelo! Es fácil ver que el modelo requerido es **EREW**. La Figura 3.5 muestra el algoritmo PARALLEL_PREFIX_2.

Complejidad

Ambos ciclos del algoritmo modificado pueden ser ejecutados en paralelo en tiempo $O(1)$ con $(n/2)$ procesadores. La llamada recursiva se aplica a un problema cuyo tamaño es la mitad del original, por lo tanto, el algoritmo se ejecuta en tiempo $T(n,n) = O(\log n)$. El número total de pasos S_n satisface la siguiente relación de recurrencia:

$$S(n) = S\left(\frac{n}{2}\right) + n - 1 \ \ y \ \ S(2) = 1 \Rightarrow S(n) = O(n).$$

```
Parallel_Prefix_2(x,n)
        // ENTRADA: X arreglo de n enteros entre 1 y n,
        // suponemos que n=2^k
        // SALIDA: x el i-esimo elemento que contiene al i-esimo prefijo

main(){
        Return PP_2(1,n)
        }

PP_2(L,R){
If inc = n/2
Then x(n/2) = x(n) x(R)
Else
        For i = 1 to n/(2 inc)
                Do In Parallel
                x(2i inc) = x((2i inc)- inc) x(2i inc)
                PP_2(2 inc)
        For i = 1 to n/(2 inc) -1
                Do In Parallel
                        x(2i inc + inc) = x(2i inc) x(2i inc + inc)
        }
```

Figura 2.5: ALGORITMO PARALLEL_PREFIX_2

Esto implica que podemos usar el Lema de Brent para mejorar la eficiencia. Podemos modificar el algoritmo para que se ejecute en tiempo $O(\log n)$ con únicamente $O(n/\log n)$ procesadores, lo que nos lleva a una eficiencia $O(1)$. La clave de este mejoramiento es usar solo una llamada recursiva, en vez de dos, mientras sea posible efectuar el paso que combina en paralelo.

Capítulo 3

Algoritmos para redes

La conexión de redes puede ser modelada con gráficas, casi siempre gráficas no dirigidas. Los procesadores corresponden a los nodos y dos nodos están conectados entre sí, si existe una liga directa entre los procesadores correspondientes. Cada procesador tiene memoria local y cada uno puede tener acceso a través de la red, la memoria local de los otros procesadores.

De esta forma, toda la memoria puede ser compartida, pero el costo de acceso a una variable depende de las localidades del procesador y la variable. Un acceso a memoria compartida puede ser tan rápido como un acceso local, si la variable está en el mismo procesador, o tan lento como cruzar toda la red, cuando la gráfica es una simple cadena. El costo, usualmente se encuentra en un valor intermedio.

Los procesadores se comunican por **mensajes**. Cuando un procesador quiere tener acceso a una variable compartida que está localizada en otro procesador, envía un mensaje preguntando por tal variable. El mensaje es enviado a través de la red.

Se han sugerido diversas gráficas como modelos para las redes interconectadas: arreglos lineales, anillos, árboles binarios, estrellas, mallas, entre otras.

Aristas adicionales son agregadas a la gráfica para mejorar la comunicación, pero éstas son caras ya que incrementan el área requerida para colocar los cables, los cuales, a su vez, incrementan el tiempo de comunicación. Trataremos de encontrar un modelo intermedio. No existe un tipo de gráfica que sea **buena** para todos los propósitos. El desempeño sobre una cierta gráfica depende fuertemente de los patrones de comunicación del algoritmo específico. Existen, sin embargo, diversas propiedades que son muy útiles, revisaremos con ejemplos algunas.

El **diámetro** de una gráfica es la mayor de las distancias más cortas entre cualesquiera

dos nodos. El diámetro es de gran importancia, ya que determina el número máximo de saltos que un mensaje podría tomar.

Una malla de $n \times n$ tiene diámetro $(2 * n)$. Un árbol binario balanceado tiene diámetro $2\log_2(n+1) - 2$. Un árbol puede enviar, en el peor caso, un mensaje mucho más rápido que una malla. Pero un árbol posee un cuello de botella, todo el tráfico de una mitad del árbol a la otra mitad debe pasar a través de la raíz. Una malla bidimensional no tiene este tipo de problema, además, es muy simétrica, lo cual es importante para algoritmos que comunican en un ritmo simétrico.

Un hipercubo Q_m se define como: $Q_m = K_2$, si $m = 1$ y $Q_m = Q_{m-1} \times K_2$ para $m > 1$. Un hipercubo Q_m o **m-dimensional** consiste de $m = 2^d$. En la Figura 4.1 se ilustra gráficamente Q_1, Q_2 y Q_3.

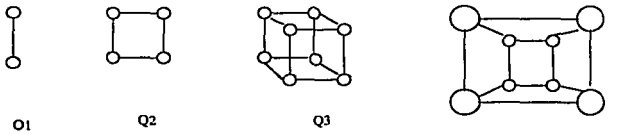


Figura 3.1: hipercubo

Las direcciones son enteros en el rango 0 a $(2^m - 1)$. Cada dirección consta de m bits. El procesador P_i es conectado al P_j si y sólo si i difiere de j en exactamente 1 bit. La distancia entre dos procesadores nunca es mayor que m, ya que podemos ir de P_i a P_j intercambiando a lo más m bits, uno a la vez. El hipercubo provee de una muy buena conexión, ya que existen diferentes rutas entre dos procesadores, es decir, podemos cambiar los bits apropiados en cualquier orden.

También podemos combinar el hipercubo con otra arquitectura. Por ejemplo, incrustando mallas en las caras del hipercubo. En la Figura 4.2 mostramos como se ve de manera gráfica.

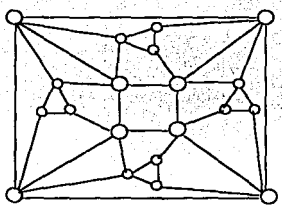


Figura 3.2: Combinacion de una malla y un hipercubo

3.1. Ordenando un arreglo

Empezamos con un problema relativamente simple: Ordenar sobre un arreglo de procesadores. Hay n procesadores $P_1, P_2, \ldots, P_n$ y n datos: $x_1, x_2, \ldots, x_n$; cada procesador mantiene un dato. El objetivo es distribuir la entrada entre los procesadores de tal forma que el dato más pequeo quede en P_1, el segundo más pequeo en P_2 y así de manera sucesiva. En general, podríamos querer asignar más de un dato a cada procesador. Veremos un algoritmo que se adapta a ambos casos. Usaremos una Conexión Lineal: el procesador P_i está conectado con P_{i+1} para toda i. La Figura 4.3 nos ilustra esta arquitectura.

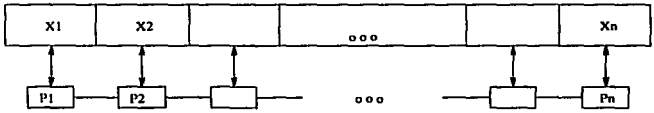


Figura 3.3: Conexión lineal

Cada procesador puede comunicarse sólo con sus vecinos. Entonces, solo comparaciones y posibles intercambios pueden realizarse entre elementos consecutivos en el arreglo. En el peor caso, el algoritmo permite $(n-1)$ pasos, que es el tiempo que toma a una entrada, o dato, moverse de un extremo a otro del arreglo.

Bosquejo del algoritmo

Cada procesador compara su número con uno de sus vecinos, intercambia los números si no están en el orden correcto, hace los mismo con el otro vecino. Deben alternarse los vecinos para no comparar un número más de una vez. El proceso se repite hasta que todos los números queden en el orden correcto. Estos pasos se dividen en **Pasos Impares** y **Pares**. En un Paso Impar, los procesadores etiquetados con un número impar comparan

con vecinos derechos. En el Paso Par, los procesadores etiquetados con un número par comparan con vecinos derechos. La Figura 4.4 lo ilustra gráficamente.

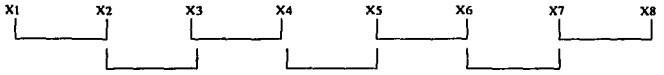


Figura 3.4: Ejemplo de los pasos par e impar

De esta manera, todos los procesadores están sincronizados y una comparación siempre involucra al procesador correcto. Si un procesador no tiene el correspondiente vecino -como el primer procesador en el segundo paso- entonces permanece inactivo, en ese paso. Este algoritmo es llamado Ordenamiento de transposición par-impar.

El algoritmo completo consta únicamente de seis pasos. Cada terminación temprana, puede ser difícil de detectar en una red. Por lo tanto, en muchos casos es mejor permitir que el algoritmo se ejecute como en el peor de los casos.

```
Ordenando_un_Arreglo(X,n)
        // ENTRADA: X arreglo de enteros entre 1 y n,
        // x_i reside en P_i
        // SALIDA: X un arreglo ordenado.
        // El i-esimo elemento mas pequeno queda en P_i
Begin
        Do In Parallel (n/2) veces
                P_{2i-1} && P_{2i} comparan sus elementos
                y los intercambian si es necesario,
                con 1 menor o igual 2i menor o igual n

                P_{2i} && P_{2i+1} comparan sus elementos
                y los intercambian si es necesario,
                con 1 menor o igual 2i menor o igual n

End
```

Figura 3.5: ALGORITMO ORDENANDO_ARREGLO

El algoritmo ORDENANDO_UN_ARREGLO parece natural y claro, pero su justificación no resulta tan obvia, ya que un elemento puede moverse alejándose de su destino final. Justificar un algoritmo paralelo es difícil, dada la interdependencia entre las acciones de los procesadores. El comportamiento de un procesador afecta a los demás y usualmente es difícil enfocarse en un solo procesador y probar que su acción es correcta, tenemos que considerar a todos los procesadores a la vez. La Figura 4.6, nos muestra una implantación del algoritmo.

Ejemplo. Considere el siguiente conjunto de datos { 7,3,6,5,8,1,4,2 }. Para ilustrar la forma en la que trabaja el arlgoritmo citemos el siguiente teorema.

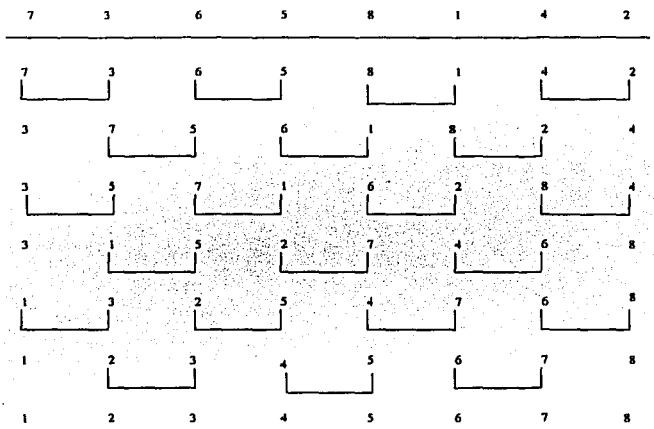


Figura 3.6: Vista gráfica de los procesadores ordenando un arreglo

Teorema. Cuando el Algoritmo ORDENANDO_UN_ARREGLO termina, los números están ordenados.

Demostración. Aplicaremos inducción sobre el número de procesadores o datos.

Base de la inducción. Si $n = 2$, una comparación es suficiente para ordenar los números.

Paso inductivo. Asumimos que el teorema es verdadero para n procesadores y consideramos el caso para $(n + 1)$ procesadores. Enfocamos nuestra atención sobre el máximo elemento, digamos $x(m)$. En el primer paso, $x(m)$ será comparado con $x(m - 1)$ o bien con $x(m + 1)$, dependiendo si m es par o impar. Si m es par, hay intercambio, ya que $x(m) > x(m - 1)$. Pero esto es lo mismo en el caso en que $x(m)$ estuviera inicialmente en P_{m-1} y un intercambio se lleva a cabo.

Por lo tanto, podemos asumir, sin pérdida de generalidad, que m es impar. En este caso, $x(m)$ es comparada con $x(m + 1)$, intercambiada y movida, paso por paso, diagonalmente, a la derecha ya que es mayor que todos los otros números hasta que llega

a la posición $x(n+1)$ y ahí se queda. Esta es su posición correcta, así que el algoritmo trabaja correctamente para el máximo elemento.

Ahora, debemos probar que el resto de los elementos son ordenados correctamente. Hay otros n elementos y nos gustaría usar inducción. Para hacerlo, debemos proyectar la ejecución de los n procesadores en el arreglo tamaño $(n-1)$ a una posible ejecución de $(n-1)$ procesadores en un arreglo de tamaño n. Esta proyección se hace de la siguiente manera: considere la diagonal formada por el movimiento del máximo elemento.

Las comparaciones que involucran al máximo elemento, todas aquellas sobre la diagonal, son ignoradas. Dividimos las otras comparaciones en dos grupos: los de abajo de la diagonal y los de arriba de la diagonal. Entonces movemos el triángulo arriba de la diagonal un paso hacia abajo. En otras palabras, para las comparaciones en el triángulo superior, el paso i es ahora llamado paso $i+1$.Veamoslo en la Figura 4.7

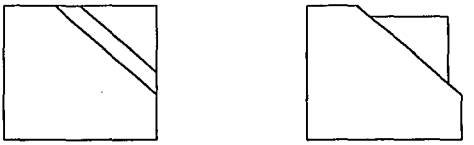


Figura 3.7: Division y comparaciones

Por ejemplo, considere las comparaciones 1 *vs* 8 y 4 *vs* 2 en el primer paso del ejemplo anterior. La primera comparación es sobre la diagonal, así que la ignoramos; la segunda está arriba de la diagonal, esta si la consideramos como parte del Paso 2, en vez del 1. Por lo tanto, el Paso 2 consiste de 7 *vs* 5, 6 *vs* 1 y, del Paso 1, 2 *vs* 4.

Este es un paso par válido que involucra n elementos. Podemos ahora, simplemente, ignorar el Paso 1 –el renglón 1– sobre el lado izquierdo de la diagonal y todas las comparaciones que involucran el máximo elemento, la última columna, y el resto es exactamente una secuencia de comparaciones que pueden resultar de ejecutar el algoritmo para n elementos. Por hipótesis de inducción, la ordenación sobre n elementos es correcta; por lo tanto, el ordenamiento sobre $(n+1)$ elementos también es correcto y requiere sólo de un paso más. □

Hemos discutido el caso de una entrada por procesador. Suponemos ahora que cada procesador mantiene k datos de entrada y consideraremos primero el caso de solo dos procesadores. Asumimos que nuestro objetivo es redistribuir los elementos tal que los k elementos más pequeños estén en P_1 y los k mayores en P_2. En el peor caso, todos los

elementos deben ser movidos, así que no puede ser mayor que $2 * k$ movimientos. Una manera de llevar a cabo el ordenamiento es repitiendo el siguiente paso hasta que el orden sea completado: P_1 envía su mayor elemento a P_2 y P_2 envía su menor elemento a P_1 . El proceso termina cuando el mayor elemento en P_1 no es mayor que el menor elemento en P_2. Este paso es llamado **MergeSlip**. Si usamos este paso como un paso básico en el ordenamiento de transposición par-impar, podemos extender el ordenamiento a varios elementos por procesador. En lugar de una comparación y posible intercambio de elementos vecinos, una operación MergeSlip se realiza.

Aunque el algoritmo de ordenamiento presentado en esta sección es óptimo para un arreglo, su eficiencia es baja. Tenemos n procesadores cada uno trabajando n pasos, por lo tanto el número total de pasos es n^2. La baja eficiencia no es una sorpresa, ya que un algoritmo eficiente de ordenamiento debe ser capaz de intercambiar elementos que estén fuera de orden.

3.2. Encontrando el k-ésimo elemento en un árbol

Asumimos que la red de interconexión es un árbol binario con $n = 2^{h-1}$ hojas. Hay 2^{h-1} procesadores, cada uno asociado con un nodo en el árbol. La entrada es la secuencia de datos $\{x_1, x_2, \ldots, x_n; x_i \neq xk\}$. Cada x_i se encuentra inicialmente en la hoja i.

Problema: Dada una secuencia con n elementos $x_1, x_2, \ldots, x_n$ encontrar el k-ésimo elemento más pequeño

Ilustraremos como a partir de un algoritmo secuencial diseñamos un algoritmo paralelo usando la técnica de encadenamiento o *pipelining*. Asumimos, por simplicidad, que todos los elementos son diferentes. El algoritmo a desarrollar es probabilístico.

En cada paso, se elige un elemento x como **pivote**, aleatoriamente. Se calcula el rango de x, comparándolo con los otros elementos. Dependiendo de si el rango de x es mayor o menor que k, son eliminados los elementos mayores o menores que x. El algoritmo termina cuando el rango del pivote es k .

El número esperado de iteraciones es $O(\log n)$ y el número esperado de comparaciones es $O(n)$. Hay tres fases diferentes en cada iteración:

1. Elegir el pivote.
2. Calcular el rango del pivote.
3. Eliminar elementos.

Primero describimos eficientemente la implantación paralela de cada fase y entonces mejoramos la paralelización total.

Elección del pivote

Para elegir un elemento aleatoriamente puede ejecutarse un torneo sobre el árbol. Cada hoja manda su número, digamos x_i, a su padre; cada x_i compite con su hermano en un volado, el ganador es promovido al padre, otra vez, y el proceso continua hasta que la raíz tiene un ganador: el pivote. Esto se realiza sólo en la primera iteración, discutiremos después cómo hacerlo cuando algunos elementos han sido eliminados. El ganador es emitido hacia abajo del árbol, así todas las hojas pueden ser comparadas con tal número.

Cálculo del rango

Si la identidad del pivote es conocida por todas las hojas, ellas pueden comparar sus números con el pivote en un paso. Las hojas, entonces, mandan un 1 si el número es menor o igual que el pivote, o bien envían un 0, en otro caso, a su padre. El rango del pivote será el número de 1s emitidos, sumar n números en un árbol es fácil de hacer. Entonces, la raíz emite el rango hacia las hojas, para que sepan si su número debe ser o no eliminado.

Eliminación de elementos

Si el rango de x es igual a k, hemos terminado,x es el k-ésimo elemento; en otro caso, si rango$(x) > k$, entonces debemos eliminar las x_i tales que $x_i \leq x$, de otra forma, se tenemos que eliminar las x_i tales que $x_i \leq x$.

Identificamos cuatro olas de comunicación por iteración:

1. Hacia arriba - *el árbol elige un pivote.*
2. Hacia abajo - *el árbol emite el pivote.*
3. Hacia arriba - *el árbol calcula el rango del pivote.*
4. Hacia abajo - *el árbol emite el rango del pivote.*

El problema es que, después de que algunos elementos son eliminados el torneo no debe fallar. En el caso extremo, todos los elementos de la mitad del árbol, excepto uno, podrían ser eliminados. El elemento sobrante en tal mitad deberá ser promovido a la raíz sin competencia alguna. Será entonces elegido con probabilidad 1/2, mientras que otros elementos son elegidos con probabilidades menores.

Queremos conservar la aleatoriedad uniforme de la elección. La preservaremos de la siguiente manera: Procesadores asociados con x_i eliminados en rondas anteriores tendrán un **Valor Nulo**. Cualquier elemento deberá ganarle al valor nulo. Cada competidor tendrá asociado un contador, al principio de cada torneo, cada contador inicia con 1. El

contador indica el número real de contrincantes que participan en el torneo que involucran estos elementos, es decir, el número de elementos en un subárbol que aun no han sido eliminados.

Cuando un elemento gana un juego a algún nodo en el árbol, se promueve hacia arriba y el contador del perdedor se acumula en el del ganador. Cada juego es jugado ahora con una moneda cargada de acuerdo al contador de los competidores. Este proceso garantiza que al final la elección en uniformemente seleccionada entre los elementos participantes.

Por ejemplo, si x gana su primer juego, digamos contra y, y z avanza por default, entonces el contador de x será 2 y el contador de z será 1. Si ahora x juega contra z, entonces el juego tiene ventaja $(2:1)$ a favor de x. Esto nos indica que z tiene una probabilidad de $(1/3)$ de ganarle a x este juego y tanto x como y tienen probabilidad de $(1/2)/(2/3) = (1/3)$ de ganar ambos sus juegos.

Complejidad

El número pasos paralelos involucrados en cada fase es igual a cuatro veces la altura del árbol. Como este algoritmo elimina elementos de la misma forma que el algoritmo secuencial, el número esperado de fases aún es $O(\log n)$. El tiempo esperado de la ejecución es $O(\log_2 n)$.

Bosquejo de un algoritmo mejorado

La raíz del árbol es un cuello de botella. La mayoría de la información debe pasar por la raíz, pero la raíz tiene solo dos conexiones y todas sus hojas están a distancia $(h-1)$ de ella. Si no podemos mejorar las conexiones, debemos, al menos, hacer que la raíz esté tan ocupada como sea posible.

En el algoritmo descrito, la raíz y las hojas están activas un paso y permanecen ociosas durante casi $(2 * h)$ pasos. Podemos mejorar este algoritmo haciendo que todos los procesadores estén ocupados todo el tiempo. Lo hacemos al inicio de cada iteración, en cada paso par, después de que las iteraciones previas sean completadas.

Todas esas iteraciones serán procesadas en una forma encadenada, de arriba hacia abajo del árbol. La razón en que este encadenamiento mejora el tiempo de ejecución a $O(\log n)$. Se toman $(2*h-2)$ pasos para seleccionar un pivote, $(h-1)$ pasos para alcanzar la raíz y $(h-1)$ pasos para que la raíz emita el resultado. Si iniciamos otro torneo en el segundo paso y ejecutamos en paralelo, pero un paso adelante, el primero, entonces podemos seleccionar dos pivotes en $(2 * h - 1)$ pasos. Podemos seleccionar h pivotes en $(3 * h - 2)$ pasos, lo cual es $O(\log n)$.

Todos esos pivotes pueden ser usados para eliminar elementos. Así, en vez de cortar el espacio de búsqueda por casi la mitad con un pivote, lo estamos cortando en casi $1/(h+1)$

de su tamaño original con h pivotes y lo hacemos sin expandir significativamente más tiempo. Podemos también trabajar con las hojas en diferentes fases. Las hojas empiezan un nuevo torneo en cada paso, hasta que el k-ésimo elemento es encontrado y el torneo empuja el resto de los cálculos.

Complejidad

El algoritmo normal requiere $O(\log n)$ pasos para eliminar elementos con un pivote. Como podemos generar $O(\log n)$ pivotes al mismo tiempo, tenemos un factor de $O(\log(\log n))$ sobre todo. El tiempo esperado se reduce a

$$O\left(\frac{\log_2 n}{\log(\log n)}\right).$$

3.3. Multiplicación de matrices sobre una malla

Consideremos una malla bidimensional de $n \times n$. El procesador $P(i,j)$ está asociado al renglón i y a la columna j, además está conectado a los procesadores $P(i+1,j)$, $P(i-1,j)$, $P(i,j+1)$ y $P(i,j-1)$. Las operaciones, sumas y restas, sobre los índices se toman módulo n. Con estos supuestos, el algoritmo que aquí presentamos resulta ser simétrico y elegante. La Figura 4.8 muestra un ejemplo de una malla de 3×3.

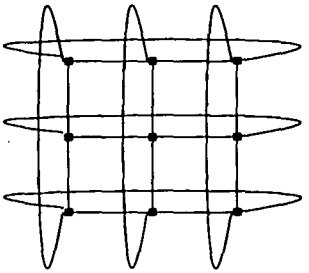


Figura 3.8: Ejemplo de una malla

Problema: Dadas dos matrices **A** y **B**, de $n \times n$, tal que $A(i,j)$ y $B(i,j)$ residen en $P(i,j)$, calcular $C = A \cdot B$ tal que $C(i,j)$ quede en $P(i,j)$.

Usamos un Algoritmo Tradicional para la Multiplicación de Matrices, que se muestra en la Figura 4.9. El problema es mover los datos tal que los números correctos se

encuentren en las posiciones correctas en el tiempo preciso.

Consideremos

$$C(0,0) = \sum_{k=0}^{n-1} A(0,k) \cdot B(k,0)$$

donde $C(0,0)$ representa el producto interno del primer renglón de A con la primera columna de B. Nos gustaría que $C(0,0)$ fuese calculado por $P(0,0)$, el cual deberá contener el resultado final. Esto puede hacerse cambiando el primer renglón de A a la izquierda, un paso a la vez y al mismo tiempo cambiar la primera columna de B hacia arriba, un elemento a la vez. En el primer paso, $P(0,0)$ tiene a $A(0,0)$ y a $B(0,0)$, entonces calcula su producto. En el segundo paso, tiene a $A(0,1)$ -de la derecha- y a $B(1,0)$ -de abajo- entonces suma su producto con el resultado parcial y de esta manera continuamos. El valor de $C(0,0)$ es calculado después de n pasos. El problema es que necesitamos todos los procesadores hagan la misma cosa y todos ellos necesitan compartir los datos. Tenemos que reorganizar los datos no sólo para $C(0,0)$, sino para todos los elementos $C(i,j)$.

El truco es reorganizar los movimientos de los datos de tal manera que, durante la ejecución del algoritmo, todos los procesadores siempre tengan los datos del producto que necesitan. La clave es la distribución inicial de los datos. Reoganizaremos los datos tal que cada procesador $P(i,j)$ tenga a $A(i,i+j)$ y $B(i+j,j)$ las sumas módulo n. Si podemos hacerlo, entonces cada paso consistirá de intercambios simultáneos de renglones y columnas que llevan a los elementos $A(i,i+j+k)$ y $B(i+j+k,j)$ al procesador $P(i,j)$ para toda k tal que $1 \leq k \leq n$, que es exactamente lo que tal procesador necesita. La siguiente Figura ilustra cómo se reorganizan los datos, en una matriz de 4×4:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{21} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \Longrightarrow \begin{bmatrix} a_{12} & a_{13} & a_{14} & a_{11} \\ a_{23} & a_{24} & a_{21} & a_{22} \\ a_{34} & a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

$$\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \Longrightarrow \begin{bmatrix} b_{21} & b_{32} & b_{43} & b_{14} \\ b_{31} & b_{42} & b_{13} & b_{24} \\ b_{41} & b_{12} & b_{23} & b_{34} \\ b_{11} & b_{22} & b_{33} & b_{44} \end{bmatrix}.$$

Observemos que los elementos de la diagonal de la primer matriz, $a_{11}, a_{22}, a_{33}, a_{44}$, se convierten en los elementos de la cuarta columna, después se ordenan los elementos hacia la derecha, de tal forma que da por resultado la segunda matriz.

Para los elementos de la diagonal de la tercer matriz, denotados como $b_{11}, b_{22}, b_{33}, b_{44}$,

se convierten en los elementos del cuarto renglon, las columnas se ordenan por la primera entrada del subíndice. De tal forma obtenemos la cuarta matriz.

```
Multiplica_Matriz(A,B)
        // ENTRADA: A,B matrices de nxn
        // SALIDA: matriz C, resultado del producto
        // de las matrices A y B
begin
For all renglones
        Do In Parallel
                For k = 1 to i Do
                A(i,j):= A(i,(j+k)mod n)
For all columnas
        Do In Parallel
                For k = 1 to i Do
                B(i,j):= B((i+k)mod n,j)
For all i and j
        Do In Parallel
                C(i,j):= A(i,j) B(i,j)
For k = 1 to n-1
        Do For all i and j
           Do In Parallel
                A(i,j):= A(i,(j+1)mod n)
                B(i,j):= B((i+1)mod n,j)
                C(i,j):= C(i,j) + A(i,j) B(i,j)
End
```

Figura 3.9: ALGORITMO MULTIPLICACION_DE_MATRICES

3.4. Trayectorias en un hipercubo

Los ejemplos que hemos visto están lejos de ilustrar la dificultad de adaptar un simple algoritmo a un algoritmo que se ejecute sobre una interconexión de red. Si los algoritmos dependen fuertemente de la arquitectura, entonces la programación se complica. Otra forma de diseñar algoritmos para interconexión.

Otra manera de diseñar algoritmos para interconexión de redes es definiendo algunas primitivas fuertes para implantarlas sobre la red y utilizarlas en el algoritmo. En esta sección se han implantado algunas primitivas sobre una red para recodificar todos los algoritmos que las usen. El problema aquí, es la definición de las primitivas. Después de todo, hay enormes diferencias entre las topologías y no es posible esperar encontrar primitivas que sean buenas para todas ellas. Otra estrategia es diseñar algoritmos que emulen una red usando otra. Esta técnica permitirá una fácil recodificación de los algoritmos entre las redes pero tal "transformación" puede llevarnos a un algoritmo nada eficiente.

Discutiremos de manera breve un esquema general de enrutamiento que nos permitirá diseñar algoritmos por interconexión de redes como si el modelo de memoria compartida estuviese disponible.

Asumimos una conexión hipercubo, un esquema similar ha sido diseñado para otras topologías. En un algoritmo **EREW** de memoria compartida, los procesadores pueden tener acceso a variables arbitrarias en un paso. Suponga que cada procesador P_i es responsable de la variable x_i, como el algoritmo **EREW** no tiene ningún conflicto de lectura y escritura, un paso en el algoritmo consiste de procesadores accesando distintas variables.

Otra forma de ver un paso es como una permutación σ tal que el procesador P_i accesa a la variable $x_{\sigma(i)}$. No todos los procesadores pueden tener acceso a todas la variables a la vez, pero en el peor caso, podemos asumir que si. Nos concentraremos sobre un paso arbitrario del algoritmo **EREW** y tratamos de emularlo usando diversos pasos en un hipercubo. Ahora tenemos un problema de enrutamiento, *routing*.

Asumimos que cada procesador P_i sobre el hipercubo quiere mandar un mensaje al procesador $P_{\sigma(i)}$, el cual mantiene a la variable $x_{\sigma(i)}$. Todos los mensajes son enviados al mismo tiempo y nuestra meta es dirigirlos -enrutarlos- todos, a la vez, hacia el hipercubo rápidamente.

Asumimos, además, que cada arista el hipercubo puede trasmitir sólo un mensaje a la vez. Por lo tanto, el problema no sólo es encontrar las rutas más cortas entre fuentes y destinos, sino también minimizar los conflictos que surjan al tratar de usar la misma ruta al mismo tiempo. Si dos mensajes tratan de usar la misma ruta al mismo tiempo uno de ellos deberá esperar en un *buffer*. Además, hay que minimizar los requerimientos de espacio del *buffer*.

La mejor ruta depende de una permutación particular. Sin embargo, esto generalmente no es posible analizar la permutación para encontrar la mejor ruta, ya que la permutación es distribuida entre los procesadores. Por lo tanto, estamos buscando un esquema que funcione bien, sobre el promedio.

En el siguiente esquema de enrutamiento la clave de la idea es usar aleatoriedad. El enrutamiento consta de dos fases. En la primera fase, cada procesador P_i envía un mensaje al procesador elegido aleatoriamente, bajo una distribución uniforme, de entre todos los procesadores e independientemente del destino, digamos que tal procesador destino es $P_{\sigma(i)}$.

En la segunda fase el mensaje es enviado a lo largo de la ruta más corta entre el procesador que recibió el mensaje, $P_{\sigma(i)}$, y el destino final. En la Figura 4.10, ilustramos el proceso.

Todos los mensajes son enviados de la misma manera, así podemos concentrarnos sólo en un mensaje, es decir de i a j. Sea la representación binaria de $i : b_1 b_2 \ldots b_d$, con posible

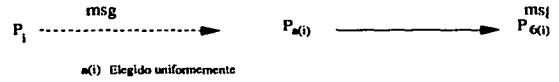


Figura 3.10: Fases de un mensaje

inicio de ceros, y la de $j : c_1c_2 \ldots c_d$.

En la primera fase, necesitamos encontrar aleatoriamente un procesador a. Lo encontraremos considerando la representación binaria de i, bit por bit, y decidiendo aleatoriamente, con probabilidad $(1/2)$, si la ruta se dirige al siguiente vecino o no.

EJEMPLO:

Sean i=000 y j=110

1er bit: Enviar a 100? NO

2do bit: Enviar a 010? SI

3er bit: Enviar a 011? SI

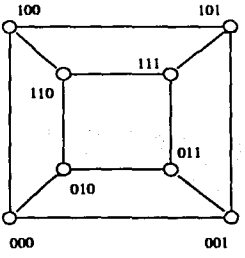


Figura 3.11: Trayectorias en un hipercubo

En la Figura 4.11, el procesador aleatorio es el 011 y la ruta, marcada con negro, pasa por él. Si decidimos no enviar el mensaje, inmediatamente hacemos la siguiente elección, sin esperar la siguiente ronda. Asumimos que el cálculo local es mucho más rápido que el paso del mensaje. Cuando la elección concerniente al último bit ha sido realizada, la ruta aleatoria queda construida. Cada elección se hace localmente.

Como todos los procesadores envían mensajes al mismo tiempo, puede haber más de un mensaje esperando ser enviado a través de la misma ruta, generándose conflictos. En este caso, los mensajes son almacenados en una cola, en orden aleatorio si hay más de uno, y son enviados cuando la arista queda disponible. No es difícil ver que cada número k, en un rango apropiado, tiene la misma probabilidad de ser elegido como un destino aleatorio.

El enrutamiento del procesador a, elegido de manera aleatoria, al j se realiza de manera determinista. Suponga que los procesadores a y j difieren en t bits, con índices tales que $k_1 < k_2 < \ldots < k_t$. El mensaje es enviado al cambiar k_1, luego k_2 y así de manera sucesiva.

En el ejemplo, $a = 011$ y $j = 110$, a continuación ilustramos cómo se construye la ruta:

$$a = 011 \Rightarrow \underline{1}11 \Rightarrow 1\underline{1}1 \Rightarrow 11\underline{0} = j.$$

Este esquema de enrutamiento es simple de implementar. Las rutas no son necesariamente las más cortas. La longitud de cada ruta, sin embargo, es a lo más $2m$. La principal propiedad de estas rutas es que, con muy alta probabilidad, poseen o generan pocos conflictos. Esto es, se espera que la permutación se termine en $O(m)$ pasos. Esquemas han sido sugeridos en otras topologías.

Capítulo 4

Computación sistólica

Una arquitectura sistólica simula la producción en línea. Los procesadores, generalmente llamados **elementos procesantes**, son organizados de una forma muy regular, usualmente en un arreglo de una o dos dimensiones, y los datos se mueven a través de los procesadores en una forma rítmica. Cada procesador desempeña una operación muy simple sobre los datos recibidos de la operación anterior y mueve el resultado a la siguiente *estación*. Cada procesador contiene, si la tiene, memoria muy limitada. La ventaja de la arquitectura sistólica es la eficiencia tanto en hardware, el cual es especializado y simple, como en velocidad, el número de ciclos de acceso a memoria es mínimo. Al igual que en las líneas de ensamblaje, la clave es evitar la necesidad de requerir herramientas o material adicional durante el trabajo. Todo lo que se requiere para ejecutar una operación lega sobre la línea. El gran problema es la inflexibilidad del esquema, además la arquitectura sistólica es eficiente únicamente para ciertos algoritmos.

4.1. Multiplicacion matríz por vector

Problema: Encontrar el producto $x = A \cdot b$ de una matriz A,de $m \times n$ y un vector columna b de tamaño n.

Consideramos que hay n procesadores $P_1, P_2, \ldots, P_n$, estaciones, tal que P_i es responsable de agregar el producto parcial del término que involucra a b_i. El dato se mueve y acciona cada procesador. Asumimos que b reside en el procesador apropiado, o es introducido de forma regular. Los resultados son acumulados como se van moviendo de izquierda a derecha a través de los procesadores.

Todas las x_i son inicialmente cero. En el primer paso x_1, cuyo valor es cero, junto con a_{11} se mueve en P_1 y todas las otras entradas se mueven más cerca. P_1 calcula $(x_1 + a_{11} \cdot b_1)$ y mueve el resultado a su derecha. En el segundo paso P_2 recibe $x_1 = (x_1 + a_{11} \cdot b_1)$ de la izquierda y calcula $(x_1 + a_{12} \cdot b_2)$ y mueve el resultado a su derecha, continuando de

esta manera el proceso. Resumiendo, en cada paso, el procesador P_i recibe un resultado parcial, el cual es igual a:

$$\sum_{k=1}^{i-1} a_{jk} \cdot b_k$$

desde la izquierda, la entrada apropiada de la matriz A y el elemento apropiado del vector b. El procesador P_i calcula $(x + a_{ji} \cdot b_i)$ y lo pasa a su derecha. Cuando x_i deja el arreglo, claramente, tiene el valor final requerido. Todo el producto es calculado en $(m+n)$ pasos. Observemos la Figura 5.1.

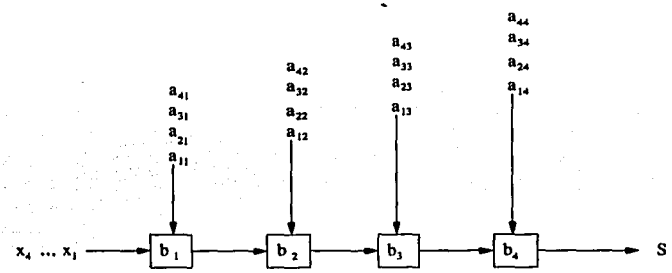


Figura 4.1: Multiplicación de una matriz por un vector

El principal problema para disear algoritmos sistólicos es el movimiento de datos. Cada elemento debe estar en el lugar correcto en el tiempo preciso. El truco en este ejemplo fue introducir retrasos para que la columna i de la matriz llegará a P_i en el paso i.

Este ejemplo es simple, ya que cada elemento de A fue usado sólo una vez. Cuando el mismo valor es usado muchas veces, como usualmente es el caso, resulta ser mucho más complicado disear los movimientos, en la siguiente sección nos enfrentaremos a este tipo de problema.

4.2. El problema de la circunvolución

Problema: Considere dos secuencias de numeros reales

$$x = x_1, x_2, \ldots, x_n, \qquad w = w_1, w_2, \ldots, w_k$$

con $k < n$. Calcular $y_1, y_2, \ldots, y_{n+1-k}$ tal que

$$y_i = w_1 \cdot x_i + w_2 \cdot x_{i+1} + \ldots + w_k \cdot x_{i+k+1}$$

El vector y es llamado la **circunvolución de x y w** . Es posible reducir la circunvolución al problema del producto matriz por vector, $X \cdot w = y$

$$\begin{bmatrix} x_1 & x_2 & \cdot & x_k \\ x_2 & x_3 & \cdot & x_{k+1} \\ x_3 & x_4 & \cdot & x_{k+2} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n+1-k} & x_{n+2-k} & \cdot & x_n \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_k \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n+1-k} \end{bmatrix}$$

Podemos obtener un algoritmo sistólico para este problema por simple sustitución de la matriz X con la matriz A, del ejemplo anterior. Como se ilustra en la Figura 5.2.

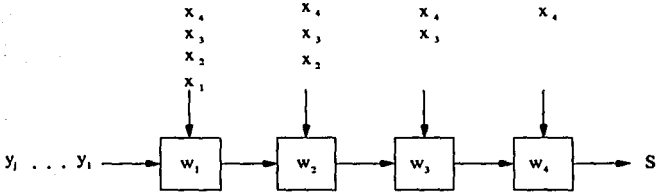


Figura 4.2: Algoritmo sistólico

Note que cada x_i es requerida al mismo tiempo a través del arreglo, excepto para los primeros $(k-1)$ pasos en los cuales no se necesitan. Así una secuencia en línea es requerida. A continuación mostraremos una solución a este problema sin tener que emitir una secuencia completa.

El procesador recibe múltiples entradas, pero envía sólo una salida. Usamos procesadores que reciben entradas de dos direcciones y envían salidas también en ambas direcciones. La idea es mover al vector x de izquierda a derecha y al w de derecha a izquierda. El resultado, y, es acumulado en los procesadores. Esto se ilustra en la Figura 5.3.

Tenemos que diseñar el movimiento de tal forma que los valores apropiados de w y x se encuentren. El problema con mover los vectores en direcciones opuestas, es que ellos

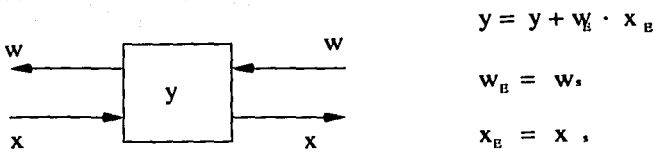


Figura 4.3: Acumulación de resultado en el procesador

se mueven en diferentes ritmos, uno se mueve dos veces dos veces más rápido que el otro. Como resultado, cada elemento de x no alcanzara a la mitad de los elementos de w, y viceversa. La solución es mover los vectores la mitad de velocidad. La entrada izquierda será: x_1, nada, x_2, nada, $x_3 \ldots$ y similarmente para w.

Esta solución es ilustrada en la Figura 5.4, donde los círculos negros corresponden a una localidad vacía.

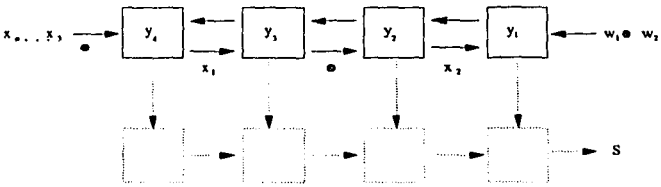


Figura 4.4: Solución al problema de la circunvolución

Cada P_i colecciona el valor de y_i. Cuando w_k deja P_i, el valor final de y_i ha sido calculado y puede moverse del arreglo a través de la trayectoria auxiliar de datos mostrada en líneas punteadas.

Capítulo 5

Técnicas básicas

La tarea de diseñar algoritmos paralelos presenta retos que son considerablemente más difíciles a los encontrados en el terreno secuencial. La falta de una metodología bien definida es compensada por una colección de técnicas y paradigmas que han sido efectivas en el manejo de problemas. Aquí introduciremos algunas técnicas, así como su aplicación a un conjunto de problemas, los cuales son muy interesantes, además de que aparecerán como subproblemas en numerosos casos.

5.1. Árboles Balanceados

El algoritmo PRAM que calcula la suma de n elementos está basado en árboles binarios balanceados en cuyas hojas están los n elementos y cuyos nodos internos representan las sumas. Este algoritmo es un ejemplo de la estrategia general para construir árboles binarios balanceados sobre los elementos de entrada y transversalmente el árbol hacia adelante y hacia atrás desde la raíz. Un nodo interno u usualmente toma información concerniente a los datos almacenados en las hojas del subárbol con raíz en u. El éxito de tal estrategia depende en parte de la existencia de un método veloz de determinar la información almacenada en nodo interno para la información almacenada en sus hijos. Usamos los cálculos de la suma prefija de n elementos para proporcionar otra ilustración de esta estrategia.

5.1.1. Un algoritmo óptimo para sumas prefijas

Consideremos una secuencia de n elementos $\{x_1, \cdots, x_n\}$ de un conjunto S con una operación binaria asociativa denotada por "$*$". La suma prefija de esta secuencia son las n sumas parciales (o productos) definidos por:

$$s_i = x_1 * x_2 * \ldots * x_i, \quad 1 \leq i \leq n.$$

Un algoritmo secuencial trivial calcula s_i desde s_{i-1} como una operación sencilla usando la identidad $s_i = (s_{i-1} * x_i)$, para $2 \leq i \leq n$, y esto toma tiempo $O(n)$. Claramente este algoritmo es inherentemente secuencial.

Podemos usar un árbol binario balanceado para derivar un algoritmo paralelo más rápido que calcule las sumas prefijas. Cada nodo interno representa la aplicación de la operación "$*$" a sus hijos mediante un recorrido hacia adelante en forma transversal en el árbol. De aquí que cada nodo satisface la suma de los elementos almacenados en las hojas de los subárboles enraizados en u. Durante el recorrido de regreso en el árbol, son calculadas las sumas prefijas de los datos almacenados en los árboles a una altura dada.

Empezamos con una versión recursiva describiendo los pasos necesarios para obtener las sumas prefijas después de la llamada recursiva sobre los nodos a la altura 1 y los pasos necesarios para obtener la suma prefija después de las llamadas recursivas sobre los nodos de altura 1 termina.

```
Sumas_Prefijas
        // ENTRADA: Un arreglo de n elementos (x_1,...,x_n)
        // suponemos que n=2^k, k entero no negativo
        // SALIDA: Las sumas sprefijas s_i para i en [1,n]

begin
        1. if n=1 ->{asigna s_1:=x_1; fin}
        2. for i=1 to n/2 pardo
                set y_i:= x_{2i-1}*x_{2i}
        3. Recursivamente, calcular la suma prefija
           de { y_1,y_2,...,y_{n/2} } y almacenalas
           en z_1,z_2,...,z_{n/2}
        4. for i=1 to n pardo
           { i even :asigna s_i:=z_{i/2}
                i=1 :asigna s_i:=x_1
                i impar :asigna s_i:=z_{(i-1)/2 * x_i}
           }
end
```

Figura 5.1: ALGORITMO SUMAS_PREFIJAS

Ejemplo. El algoritmo SUMAS PREFIJAS sobre ocho elementos se muestra en la Figura 6.1 Durante la primera unidad de tiempo, los cuatro elementos son calculados, es decir:

$$y_1 = (x_1 * x_2),\ y_2 = (x_3 * x_4),\ y_3 = (x_5 * x_6),\ y_4 = (x_7 * x_8).$$

La segunda unidad de tiempo corresponde al cálculo de $y_1' = (y_1 * y_2)$ y $y_2' = (y_3 * y_4)$ por medio de una llamada recursiva para manejar ambas entradas. El elemento $y_1'' = (y_1' * y_2')$ es calculado durante la unidad de tiempo 3. Por lo tanto, en la cuarta unidad de tiempo, la suma prefija de esta entrada es generada. El proceso inverso comienza generándose en el tiempo cinco, las sumas prefijas z_1' y z_2' de las dos entradas y_1' y y_2'. Similarmente, en

el tiempo seis, las sumas prefijas z_1, z_2, z_3, z_4 de los cuatro elementos y_1, y_2, y_3, y_4 son generados. Finalmente, las sumas prefijas $\{s_i\}$ de las x_i's son generadas en el tiempo siete.

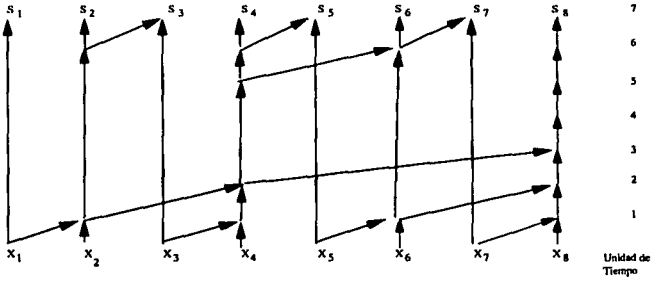


Figura 5.2: La suma prefija de ocho elementos

Para las entradas de tamaño $n = 2^k$ el algoritmo requiere $2k + 1$ unidades de tiempo. En las primeras k unidades de tiempo, el cálculo avanza de las hojas de un árbol binario completo a la raíz. Durante las últimas k unidades de tiempo, recorremos el árbol de regreso y calculamos las sumas prefijas usando datos generados en las primeras k unidades de tiempo.

Nótese que el algoritmo total puede ser ejecutado sin usar variables auxiliares y que usamos algunas variables auxiliares para ilustrar el algoritmo. Ahora estamos listos para revisar el siguiente teorema, el cual establece el desempeño computacional del algoritmo en el peor caso.

Teorema. El algoritmo SUMAS_PREFIJAS calcula las sumas prefijas de n elementos en tiempo $T(n) = O(\log_n)$ usando $W(n) = O(n)$ operaciones.

5.1.2. Un algoritmo no recursivo para la suma prefija

Sea $A(i) = x_i$, donde $1 \leq i \leq n$. Sea $B(h, j)$ y $C(h, j)$ el conjunto de variables auxiliares, donde $0 \leq h \leq \log n$ y $1 \leq j \leq (n/2^h)$. El arreglo B será para guardar la información en los nodos del árbol binario, durante un recorrido transversal hacia adelante del árbol, mientras que el arreglo C sera usado durante el recorrido transversal hacia atras del árbol. La Figura 6.3 ilustra el recorrido transversal hacia adelante, mientras que la Figura 6.4 ilustra el recorrido transversal hacia atras, para $n = 8$.

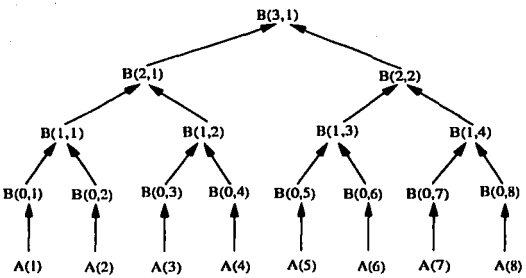


Figura 5.3: Árbol binario usado en el algoritmo no recursivo de la suma prefija

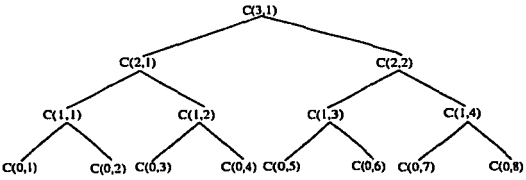


Figura 5.4: Los elementos del arreglo C generados de abajo hacia arriba

Ejemplo. Sea $n = 8$ (véase Figura 6.3 y Figura 6.4). Inicialmente asignamos $B(0,j) = A(j)$, para $1 \leq j \leq 8$. Los $B(0,j)$ corresponden a las hojas del árbol binario. Las variables $B(1,j)$, donde $1 \leq j \leq 4$, corresponden a los nodos internos de altura 1; las variables $B(2,j)$, con $1 \leq j \leq 2$, corresponden a los vértices internos de altura 2. La raíz del árbol binario es almacenada en $B(3,1)$. Entonces recorremos este árbol binario hacia atrás. Empezamos colocando $C(3,1) = B(3,1)$. En el siguiente paso, generamos $C(2,1) = B(2,1)$ y $C(2,2) = B(2,2)$. De aquí, $C(2,1)$ y $C(2,2)$ toman las sumas prefijas correspondientes a las entradas $B(2,1)$ y $B(2,2)$. Similarmente, $C(1,1), C(1,2), C(1,3), C(1,4)$ son las sumas prefijas de $B(1,1), B(1,2), B(1,3), B(1,4)$ Por lo tanto las $C(0,j)$ toman la sumas prefijas de las entradas originales.

5.1.3. Adaptación del principio de calendarización WT

La adaptación del principio de calendarización WT requiere la determinación del número de operaciones en cada paso en paralelo y una solución al correspondiente pro-

blema de distribución de procesadores.

Sea $n = 2^k$. Hay $2\log n+2 = 2k+2$ pasos paralelos en el algoritmo que se muestra en la Figura 6.5. Sea $W_{1,1}$ el número de operaciones ejecutadas en el paso 1, y sea $W_{i,m}$ el número de operaciones ejecutadas en el paso i en el Algoritmo SUMAS_PREFIJAS_NO_RECURSIVAS durante la m-ésima iteración, $i = 2, 3$. Entonces $W_{1,1} = n, W_{2,m} = (n/2^m) = 2^{k-m}$ para $1 \leq m \leq k$ y $W_{3,m} = 2^m$ para $0 \leq m \leq k$. El número total de operaciones está dado por:

$$W(n) = W_{1,1} + \sum_{m=1}^{k} W_{2,m} + \sum_{m=0}^{k} W_{3,m}$$

$$= n + 2^k \sum_{m=1}^{k} 2^{-m} + \sum_{m=0}^{k} 2^m$$

$$= n + n(1 - (1/n)) + 2n - 1 = O(n).$$

El problema de la designación de procesadores se ve a continuación.

```
Sumas_Prefijas_No_Recursivas
        // ENTRADA: Un arreglo A de tama\~no n
        // suponemos que n=2^k, k entero no negativo
        // SALIDA: Un arreglo C talque C(0,j) es la
           j-esima suma prefija, para 1<=j<=n.
begin
        1. if 1<=j<=n pardo
                set B(0,j):=A(j)
        2. for h=1 to log(n) do
                for i<=j<=n/(2^h) pardo
                        setB(h,j):=B(h-1,2j-1)*B(h-1,2j)
        3. for h=log(n) to 0 do
                for i<=j<=n/(2^h) pardo
                { j par :setC(h,j):C(h+1,j/2)
                  j=1 :setC(h,1):B(h,1)
                  j impar>1 :setC(h,j):C(h+1,(j-1)/2)*B(h,j)
                }
end
```

Figura 5.5: ALGORITMO SUMAS_PREFIJAS_NO_RECURSIVAS

Sea **PRAM** nuestro modelo p, donde $p = 2^q \leq n$ procesadores $P_1, P_2, \ldots, P_p$, y sea $l = (1/p) = 2^{k-q}$. La entrada del arreglo es dividida en p subarreglos tales que el procesador P_s es el responsable de los procesos del s-ésimo subarreglo

$$A(l(s-1)+1), A(l(s-1)+2), \ldots, A(ls).$$

En cada altura h del árbol binario, durante cada recorrido transversal hacia adelante o hacia atrás, la generación de los valores de $B(h, \cdot)$ y $C(h, \cdot)$ se divide de similar manera

```
Algoritmo_Procesador_Ps
        // ENTRADA: Un arreglo A de tama\~no n
        // suponemos que n=2^k, k entero no negativo,
        // un indice s que satisface 1 <= s <= p = 2^q,
        // donde p <= n es el numero de procesadores
        // SALIDA: La suma prefija C(0,j) para
        // (n/p)*(s-1)+1 <= j <= ns/p
begin
        1. for j=1 to l=(n/p) do
              set B(0,l(s-1)+j):=A(l(s-1)+j)
        2. for h=1 to k do
           2.1 if(k-h-q=>0) then
                for j= 2^{k-h-q}(s-1)+1 to 2^{k-h-q}s do
                set B(h,j):=B(h-1,2j-1)*B(h-1,2j)
           2.2 else
                {if(s<=2^{k-h})} then
                set B(h,s):=B(h-1,2s-1)*B(h-1,2s)
                }
        3. for h=k to 0 do
           3.1 if (k-h-q=>0) then
                for j=2^{k-q-h}(s-1)+1 to 2^{k-q-h}s do
                { j par :setC(h,j):C(h+1,j/2)
                  j=1 :setC(h,1):B(h,1)
                  j impar>1 :setC(h,j):C(h+1,(j-1)/2)*B(h,j)
                }
           3.2 else
                {if (s<= 2^{k-h}) then
                   { s par :setC(h,s):C(h+1,s/2)
                     s=1 :setC(h,1):B(h,1)
                     s impar>1 :setC(h,s):C(h+1,(s-1)/2)*B(h,s)
                   }
                }
end
```

Figura 5.6: ALGORITMO PARALLEL_PREFIX_1

entre los p procesadores. Esta división se ejecuta de forma parecida a la del algoritmo paralelo que calcula la suma de n números.

En el Algoritmo PARALLEL_PREFIX_1 (Figura 6.6), las Condiciones 2,1 y 3,1 son ejecutadas siempre y cuando el número de operaciones en esa iteración en particular sean mayores o iguales que p. Estas operaciones se distribuyen equitativamente entre los p procesadores. En caso de que el número de operaciones sea menor que p (Pasos 2.2 y 3.2) asignamos una operación por procesador empezando por procesador indexado más pequeño. Note que, para cualquier valor de h en los ciclos definidos de los pasos 2 y 3, el posible número de operaciones concurrentes es

$$\frac{n}{2^h} = 2^{k-h}.$$

El algoritmo ejecutado por el s-ésimo procesador se observa a continuación. La Figura 6.7 ilustra el procesador correspondiente asignado en este caso de $n = 8$ y $p = 2$

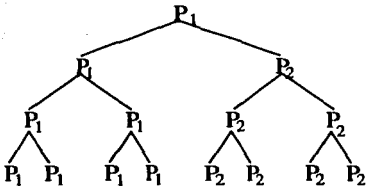


Figura 5.7: Vista de los procesadores y su ejecución

Ejemplo. Sea $n = 8$ y $p = 2$. Consideremos el algoritmo correspondiente al procesador P_2. En el paso 1, P_2 asigna

$$B(0,5) = A(5),\ B(0,6) = A(6),\ B(0,7) = A(7),\ B(0,8) = A(8)$$

como lo muestran la Figuras 6.3 y la Figura 6.7. Durante la ejecución del paso 2, P_2 será activado por $h = 1, 2$ y se desactivará para $h = 3$. El procesador P_2 generará $B(1,3)$, $B(1,4)$, $B(2,2)$ durante el ciclo definido por el paso 2. Similarmente, durante el recorrido transversal hacia atrás del árbol binario, P_2 se desactivará para $h = 3$ y se activará para $h = 2, 1, 0$. De aquí, P_2 genera las sumas

$$C(2,2), C(1,3), C(1,4), C(0,5), C(0,6), C(0,7)\ y\ C(0,8).$$

La Figura 6.4 lo ilustra de manera detallada.

5.2. La Técnica Pointer Jumping

Un árbol enraizado y dirigido T es una gráfica dirigida con un nodo distinguido r llamado *raíz de* T tal que:

1. $\forall v \in V \backslash \{r\}$ el grado exterior de v es 1 y el grado exterior de $r = 0$;
2. $\forall v \in V \backslash \{r\}$ existe una trayectoria dirigida de v a r.

El nodo especial r es llamado **raíz** de T. Esto significa que un árbol enraizado y dirigido es una gráfica dirigida cuya versión no dirigida es un árbol enraizado tal que cada arco de T es dirigido de un nodo a su nodo padre.

La técnica *Pointer Jumping* permite el rápido procesamiento de datos almacenados en un conjunto cuya forma en un árbol enraizado y dirigido.

5.2.1. Encontrando las raices de un bosque

Sea F un bosque de árboles dirigidos y enraizados. El bosque F es especificado por un arreglo P de longitud n tal que $P(i) = j$ si (i, j) es un arco en F; Esto es j es el padre de i en el árbol de F. Por simplicidad, si i es una raíz asignamos $P(i) = i$. El problema consiste en determinar la raíz $S(j)$ del árbol que contiene al nodo j para cada $j \in \{1, \ldots, n\}$.

Un simple algoritmo secuencial (digamos uno basado en primero identificar las raices y luego aplicar DFS o BFS a cada árbol desde su raíz) resuelve el problema en tiempo lineal. Presentemos un algoritmo paralelo rápido que sigue una estrategia totalmente diferente.

Inicialmente, el sucesor $S(i)$ de cada nodo i es $P(i)$. La técnica de *Pointer Jumping* consiste en actualizar el sucesor de cada nodo por el sucesor de su sucesor. Como esta técnica es aplicada repetidamente, el sucesor de un nodo es un antecesor que llega a estar más cerca de la raíz del árbol que lo contiene. La distancia entre un nodo y su sucesor se duplica a no ser que el nodo sucesor sea la raíz. De aquí, después de k iteraciones, la distancia entre i y $S(i)$ como aparecen en el árbol dirigido de F es 2^k a no ser que $S(i)$ sea la raíz. La manera más detallada se ve en la Figura 6.8.

```
Pointer_Jumping
        // ENTRADA: Un bosque de arboles dirigidos a la raiz
        // con un autociclo a la raiz, tal que cada arco es
        // especificado por (i,P(i)), donde 1<=i<=n
        // SALIDA: Para cada vertice i, la raiz S(i) del arbol
        // contenido en i
begin
        1. for i<=i<=n pardo
                set S(i):=P(i)
                while(S(i)) diferente S(S(i)) do
                        set S(i):=S(S(i))

end
```

Figura 5.8: ALGORITMO POINTER_JUMPING

Ejemplo. Una ilustración del Algoritmo POINTER_JUMPING se muestra en la Figura 6.9. La Figura 6.9(a) muestra el bosque inicial, la Figura 6.9(b) es el bosque de la primera iteración y la Figura 6.9(c) muestra el bosque en la tercer iteración. En este caso el bosque consiste de dos árboles: Uno enraizado en el vértice 8 y el otro enraizado en el vértice 13. Los arcos en esta figura corresponden a $(i, P(i)), 1 \leq i \leq 13$. La primera ejecución del ciclo while causa que los vértices $1, 2, 3, 4, 5, 9, 10$ y 11 cambien su sucesor. La segunda iteración ocasiona que los dos árboles tengan profundidad 1. Ahora tenemos $S(i) = S(S(i))$, para toda i, entonces el algoritmo termina.

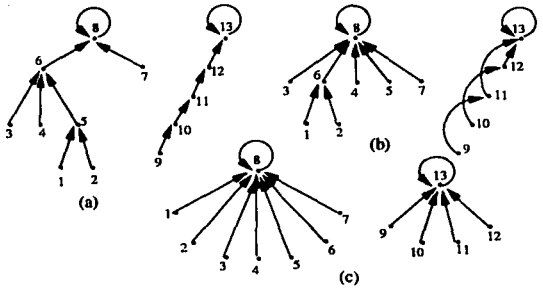


Figura 5.9: Ilustración de la técnica *Pointer jumping*

Sea h la máxima altura de los árboles del bosque. La justificación del algoritmo la hacemos por inducción sobre h.

Como la distancia entre i y $S(i)$ se duplica después de cada iteración hasta que $S(S(i))$ es la raíz del árbol contenido en i. De aquí obtenemos que el número de operaciones es $O(\log h)$. Cada iteración puede ser ejecutada en $O(1)$ en tiempo paralelo, con $O(n)$ operaciones. Por lo tanto el algoritmo se ejecuta en tiempo $O(\log_a h)$ y usa un total de $W(n) = O(n \log h)$ operaciones. Esta complejidad claramente no es óptima a menos que h sea constante, ya que existe un algoritmo secuencial que lo hace en tiempo lineal.

Teorema. Dado un bosque de árboles dirigidos enraizados, el Algoritmo POINTER_JUMPING genera para cada vértice i la raíz del árbol i. El algoritmo se ejecuta en tiempo $O(\log(h))$ usando un total de $O(n \log(h))$ operaciones, donde h es la máxima altura de los árboles del bosque, y n es el número total de vértices en el bosque.

5.2.2. Prefijo Paralelo.

Asumimos que cada nodo i en el bosque F contiene un peso $W(i)$. La técnica *Pointer Jumping* puede ser usada para calcular, para todo nodo i, la suma de los pesos almacenados en los nodos sobre la trayectoria del nodo i a la raíz del árbol que contiene a i. Estas cantidades calculadas generan las sumas prefijas de diferentes secuencias de elementos que están dadas por el orden en que los nodos aparecen sobre la trayectoria.

El algoritmo PREFIJO_PARALELO mostrado en la Figura 6.10 proporciona los detalles.

```
Prefijo_Paralelo
        // ENTRADA: Un bosque de arboles dirigidos a la raiz
        // con un autociclo a la raiz tal que
        // (1)cada arco es especificado por (i,P(i))
        // (2)cada vertice i tiene un peso W(i)
        // (3)para cada raiz r, W(r)=0.
        // SALIDA: para cada vertice i, W(i) es igual a la suma de
        // los pesos de los vertices del camino desde i hasta la raiz
        // de su arbol.
begin
        1. for i<=i<=n pardo
                set S(i):=P(i)
                while(S(i)) diferente S(S(i)) do
                        set W(i):=W(i)+W(S(i))
                        set S(i):=S(S(i))

end
```

Figura 5.10: ALGORITMO PREFIJO_PARALELO sobre árboles dirigidos enraizados

Ejemplo. Retomemos el bosque de la Figura 6.9. Supóngase que inicialmente, $W(i) = 1$ para todo i distinto de 8 y 13, $W(8) = W(13) = 0$. Durante la primera iteración del ciclo, obtenemos:

$$W(1) = W(2) = W(3) = W(4) = W(5) = W(9) = W(10) = W(11) = 2.$$

Los otros vértices permanecen con sus valores W iniciales. Para la segunda iteración tenemos

$$W(1) = W(1) + W(6) = 3,$$
$$W(9) = W(9) + W(11) = 4,$$
$$W(2) = W(10) = 3,$$
$$W(3) = W(4) = W(5) = W(11) = 2$$
$$W(6) = W(7) = W(12) = 1.$$

De esta forma, cada $W(i)$ corresponde a la longitud de la trayectoria de i a su raíz.

5.3. La Técnica Divide y Vencerás

La estrategia básica de **divide y vencerás** consiste de tres principales pasos:

- El primer paso es particionar la entrada en varias secciones de tamaños casi iguales.

- El segundo paso es resolver recursivamente el subproblema definido para cada partición. Nótese que esos subproblemas pueden ser resueltos concurrentemente en una máquina paralela.

- El tercer paso es la combinación de las soluciones de los diferentes subproblemas para dar una solución al problema original.

El éxito de tal estrategia depende de la posibilidad de ejecutar los tres pasos de manera eficiente. Esta estrategia ha sido mostrada como eficiente en el desarrollo de algoritmos secuenciales. Además esto permite la explotación del paralelismo en su forma natural. Ilustraremos esta técnica con el problema de la cubierta convexa.

5.3.1. El Problema de la Cubierta Convexa

Dado un conjunto $S = \{p_1, p_2, \ldots, p_n\}$ de n puntos en el plano, cada uno representado por sus coordenadas (x, y), la **cubierta convexa plana** de S es el polígono convexo más pequeño que contiene a los n puntos de S. Un polígono Q es **convexo** si, para cualesquiera dos puntos p y q en Q, el segmento que los une está enteramente contenido en Q. El problema de la cubierta convexa consiste en determinar el orden (en sentido de las manecillas del reloj) de la lista $CH(S)$ de los puntos de S definiendo la orilla de la cubierta convexa de S.

Ejemplo. Considere el conjunto S de puntos mostrados en la Figura 6.11. En este caso la cubiera convexa de S esta dado como sigue: $CH(S) = (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)$.

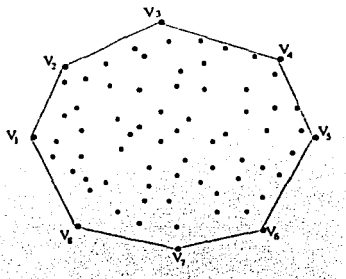


Figura 5.11: La cubierta convexa de un conjunto de puntos

FALTA

PAGINA

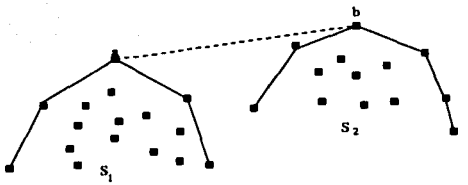


Figura 5.12: La tangente común superior

Sea $UH(S_1) = (q_1, \ldots, q_s)$ y $UH(S_2) = (q'_1, \ldots, q'_t)$ las cubiertas superiores de S_1 y S_2, respectivamente, dadas en orden de izquierda a derecha. Note que en particular $q_1 = p_1$ y que $q'_t = p_n$. Suponga que la tangente común superior ha sido determinada y está dada por (q_i, q'_j).

Supóngase que la tangente común superior ha sido determinada y está dada por (q_i, q'_j). Entonces, $UH(S)$ es el arreglo que consiste de las primeras i entradas de $UH(S_1)$ y las $t - j$ entradas de $UH(S_2)$; es decir, $UH(S) = (q_1, \ldots, q_i, q'_j, \ldots, q'_t)$. Si s y t están dadas, entonces los índices i y j son conocidos, $UH(S)$ y su tamaño pueden ser determinados en $O(1)$ tiempo en paralelo, usando un total de $O(n)$ operaciones.

```
Cubierta_Convexa_Simple
        // ENTRADA: Un conjunto S de n puntos en el plano,
        // ninguno de ellos con la misma coordenada x o y
        // tal que x(p1)<x(p2)<...<x(pn),con n  potencia de 2
        // SALIDA: La cubierta superior de S.

begin
        1. if n<=4 then
                Usar metodo de fuerza bruta para determinar
                UH(S)
                exit.
        2. Sea S1=(p1,p2,...,pn/2) y S2=(pn/2+1,...,pn).
           Recursivamente, calcule UH(S1) y UH(S2) en paralelo.

        3. Encuentre la tangente comun superior UH(S1) y UH(S2)
           y deduzca la cubierta superior de S.
end
```

Figura 5.13: ALGORITMO CUBIERTA_SUPERIOR_SENCILLA

5.4. La Técnica de Partición

La **Estrategia de partición** consiste en:

1. Particionar el problema dado en p subproblemas independientes de tamaños casi iguales;

2. Resolver los p subproblemas concurrentemente, donde p es el número de procesadores disponibles.

De esta forma, se puede solucionar el problema concurrentemente. Sin embargo, en la mayoria de los casos, nos sentiremos afortunados de separar el problema en un conjunto independiente de subproblemas, como se ilustra en el problema de mezclar dos secuencias ordenadas.

Dado un conjunto S con una relación de orden parcial " $\leq$ " . Esto es, " $\leq$ " es reflexivo, antisimétrico y transitivo. S es **linealmente ordenado** o **totalmente ordenado**, si para cada par $a, b \in S$ ocurre que $(a \leq b)$ o $(b \leq a)$.

Sean $A = (a_1, a_2, \ldots, a_n)$ y $B = (b_1, b_2, \ldots, b_n)$ dos secuencias no decrecientes cuyos elementos son tomados de un conjunto linealmente ordenado S. Consideraremos el problema de **mezclar** esas dos secuencias en una secuencia ordenada, que llamaremos

$$C = (c_1, c_2, \ldots, c_{2n}).$$

Un simple algoritmo secuencial resuelve el problema de mezclar las secuencias en tiempo lineal. Nuestra meta es proveer una solución paralela con base en particionar a A y B en tantos parejas de subsecuencias tales que podemos obtener la secuencia ordenada mezclando concurrentemente las parejas resultantes de subsecuencias.

Comparando esta estrategia con la estrategia divide y vencerás donde el principal trabajo requerido es usualmente combinar las soluciones de los subproblemas involucrados más que particionar la entrada.

5.4.1. Un algoritmo sencillo para mezclar

Empezaremos con un par de definiciones. Sea $X = (x_1, x_2, \ldots, x_t)$ una secuencia cuyos elementos provienen de un conjunto S. Sea $x \in S$. El *rango* de x en X denotado por $rank(x : X)$, es el número de elementos de X que son menores o iguales que x. Sea $Y = (y_1, y_2, \ldots, y_s)$ un arreglo arbitrario de elementos de S. El rango de Y sobre X es el problema de determinar el arreglo $rank(Y : X) = (r_1, r_2, \ldots, r_s)$, donde $r_i = rank(y_i : X)$.

Ejemplo. Sea $X = (25, -13, 26, 31, 54, 7)$ y $Y = (13, 27, -27)$. Entonces el

$$rank(Y : X) = (2, 4, 0).$$

Sin pérdida de generalidad, asumamos que todos los elementos aparecen en dos secuencias ordenadas a ser mezcladas, A, B, son distintas. En particular, ningún elemento de A aparece en B.

El problema de mezclar puede ser visto como determinar el rango de cada elemento x de A o de B en el conjunto $A \cup B$. Si el $rank(A \cup B) = i$, entonces $c_i = x$, donde c_i es el i-ésimo elemento de la secuencia ordenada deseada. Ya que $rank(x : A \cup B) = rank(x : A) + rank(x : B)$, podemos resolver el problema de mezclar, determinando los dos arreglos de enteros $rank(x : A)$ y $rank(x : B)$.

Describamos el algoritmo para resolver $rank(B : A)$. El mismo algoritmo puede ser usado para determinar el $rank(A : B)$. Sea b_i un elemento arbitrario de B. Ya que A esta ordenado, podemos encontrar el rango de b_i en A usando el metodo de búsqueda binaria. Recordemos que este método compara a b_i con el elemento en la posición $(n/2)$ de A. Basándonos en la salida de esta comparación, la búsqueda puede restringirse a la mitad superior o a la mitad inferior de A. Este proceso se repite hasta que b_i queda entre dos entradas sucesivas de A, esto es, $a_{j(i)} < b_i < a_{j(i)+1}$, donde $rank(b_i : A) = j(i)$. Nótese que partimos del hecho de que los elementos de A y B son distintos.

La búsqueda binaria sólo determina el rango de un elemento arbitrario de B en A, y corre en $O(\log n)$ tiempo secuencial. Podemos obviamente aplicar este método de manera concurrente a todos los elementos de B para obtener un algoritmo paralelo que diga el rango de B en A lo cual implicaría que tenemos un algoritmo paralelo para mezclar secuencias que lo hace en $O(\log n)$, para una cadena de longitud n. Sin embargo, el total de operaciones usadas por tal algoritmo es de $O(n \log n)$, el cual no es óptimo, ya que existe un algoritmo secuencial que lo hace en tiempo lineal.

5.4.2. Un algoritmo de mezcla óptimo

Un algoritmo de mezcla óptimo puede ser obtenido de la siguiente manera. Escogiendo aproximadamente $(n/\log n)$ elementos de A, y B, cada partición A y B de tamaños casi iguales. Aplicando el método de la búsqueda binaria para encontrar el rango de cada uno de los elementos elegidos en la otra secuencia. Este paso reduce el problema en mezclar pares de subsecuencias, cada uno de estos tiene $O(\log n)$ elementos. Podemos aplicar un algoritmo secuencial en tiempo óptimo a cada uno de los pares de subsecuencias para generar la secuencia ordenada. Presentamos en la Figura 6.15 el algoritmo PARTICION.

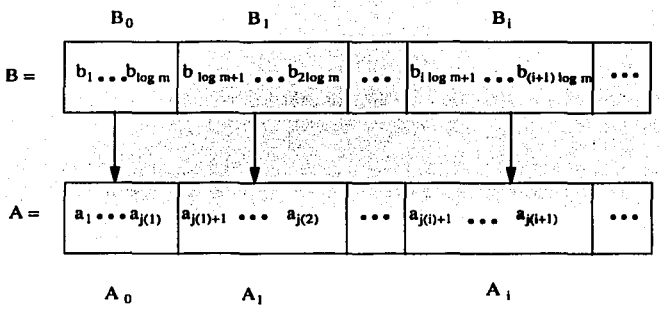


Figura 5.14: Subsecuencias obtenidas por el algoritmo partición

5.5. La Técnica de Encadenamiento

Un **2-3 árbol** es un árbol enraizado en donde cada nodo interno tiene dos o tres hijos, y cada trayectoria de la raíz a las hojas tiene la misma longitud. El número de hojas de un 2-3 árbol de altura h está entre 2^h y 3^h. De aquí, si el número de hojas es n, la altura del árbol es de $\Theta(\log n)$.

Una lista ordenada, de n elementos, $A = (a_1, a_2, \ldots, a_n)$, donde $a_1 < a_2 < \ldots < a_n$ puede ser representada por un 2-3 árbol T, donde las hojas toman los valores de los objetos en orden de izquierda a derecha. Un nodo interno v tomará dos valores, $L[v]$ y $M[v]$, donde $L[v]$ es el mayor objeto almacenado en el primer subárbol de v más a la izquierda y $M[v]$ el mayor objeto almacenado en el segundo subárbol de v. Además, v toma el valor $R[v]$ representando el mayor valor en el tercer subárbol (que esta situado más a la derecha) de v, siempre que v tenga un tercer hijo. El objeto $R[v]$ generalmente no está incluido en la definición de un 2-3 árbol.

Ejemplo. Sea A una lista $A = (1, 2, 3, 4, 5, 6, 7)$. Un 2-3 árbol representado por A se muestra en la Figura 6.16. Hemos usado la notación $i : j : k$ para decir quien es el siguiente nodo interno v e indica que $L[v] = i$, $M[v] = j$ y $R[v] = k$, siempre que el subárbol más a la derecha exista. Observe que el 2-3 árbol correspondiente a A no es único.

```
Particion
        // ENTRADA: Dos arreglos, A=(a1,...,an) y B=(b1,...,bm)
        // en orden creciente, donde ambos log(m) y k(m)=m/log(m)
        // son enteros.
        // SALIDA: k(m) pares (Ai,Bi) de subsecuencias de
        // A y B talque:
        // (1) |Bi|= log(m)
        // (2) @\Sigma_i@ |Ai|=n
        // (3) Cada elemento de Ai y Bi es mas grande que
        // cada elemento de Ai-1 o Bi-1 para 1<=i<=k(m)-1.

begin
        1. set J(0):=0, j(k(m)):=n

        2. for 1<=i<=k(m)-1 pardo
                2.1 Asigna rango b_{ilogm} en A usando el metodo de
                    busqueda binaria, y sea j(i)=rango(b_{ilogm}:A)

        3.for 0<=i<=k(m)-1 pardo
                3.1 set Bi:=(b_{ilogm+1},...,b_{(i+1)logm})
                3.2 set Ai:=(a_{j(i)+1},...,a_{j(i+1)})
                    A(i) esta vacio si j(i)=j(i+1)
end
```

Figura 5.15: ALGORITMO PARTICION

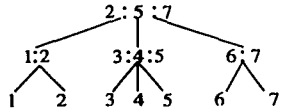


Figura 5.16: Ejemplo de un 2-3 árbol

5.5.1. Operaciones Básicas en un 2-3 árbol.

Un 2-3 árbol es una estructura de datos bien conocida, usada para representar conjuntos de elementos y realizar operaciones de búsqueda, inserción y borrado. De manera más precisa, un 2-3 árbol representa un conjunto que cambia dinámicamente sus elementos, inducido por el procesamiento de una secuencia de instrucciones:

1. Búsqueda de un elemento.
2. Inserción de un elemento.
3. Borrado de un elemento.

Los elementos son tomados de un conjunto linealmente ordenado. Observe que el conjunto de elementos representado por un 2-3 árbol aparece ordenado, y al hacer alguna operacioń como inserción o borrado requiere que la estructura del 2-3 árbol se preserve después de la operación. Cada una de las operaciones inserción, búsqueda, y borrado pueden ser ejecutadas en tiempo $O(\log n)$, donde n es el numero de elementos del conjunto.

Sea T un 2-3 árbol representante de una lista de n elementos, $a_1 < a_2 < \ldots < a_n$. Dado un elemento b, el problema de buscar a b puede ser definido como el problema de encontrar la hoja u de T tal que el valor del dato a_i almacenado en u satisfaga que $a_i \leq b \leq a_{i+1}$.

Este problema puede ser manejado con un método de búsqueda binaria de la siguiente manera. Sea r la raíz de T. La búsqueda de b puede restringirse a uno de los subárboles de r, dependiendo donde b se ajuste entre dos datos $L[r]$ y $M[r]$. Más precisamente, si $b \leq L[r]$, la búsqueda continúa en el subárbol mas a la izquierda de r; de otro modo, si $L < b \leq M[r]$, la búsqueda continúa en el segundo subárbol. Manejamos el caso restante de $b > M[r]$ restringiendo la búsqueda al tercer subárbol. El procedimiento de búsqueda termina cuando se alcanza una hoja. Este proceso toma tiempo proporcional a la altura de T, es decir $O(\log n)$.

Podemos insertar un elemento b en un árbol T, primero aplicando la técnica de búsqueda anterior para localizar a b. Sea u la hoja identificada por el procedimiento de búsqueda. El nodo u contiene un elemento a_i tal que $a_i \leq b < a_{i+1}$. Si $b = a_i$, no hay nada que hacer, ya que el elemento b está presente en el árbol. De otro modo, tenemos que crear una hoja u', la cual toma el valor de b. Insertamos u' inmediatamente a la derecha de u con el mismo padre –llamémosle p– de u. Si p tiene tres hijos, nos detenemos. Asumimos que p tiene cuatro hijos, creamos un nodo p' y distribuimos a los hijos de p apropiadamente en p y p'. En la Figura 6.17 lo ilustra de manera grafica. Podemos insertar el nodo p' en T usando el mismo procedimiento. Esto es, p' es tentativamente asignada al mismo padre de p, el cual debeser examinado para evitar una posible violación a la restricción del número de hijos. Finalmente, si la raíz r tiene cuatro hijos, creamos un nuevo nodo raíz con dos hijos, cada uno de ellos tiene apropiadamente dos hijos de r.

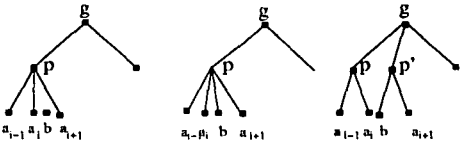


Figura 5.17: El proceso de inserción de objetos en un 2-3 árbol

Durante el proceso de inserción, los valores de L, M y R son ajustados apropiadamente sin incremento asintótico los tiempos secuenciales del procedimiento.

Ejemplo. Observemos la Figura 6.18(a). Supongamos que queremos insertar el dato 5 en el 2-3 árbol T. El siguiente paso será la creación de una nueva hoja que contendrá el

valor 5, dicha hoja será hija del nodo c. La Figura 6.18(b) muestra la ejecución del segundo paso.

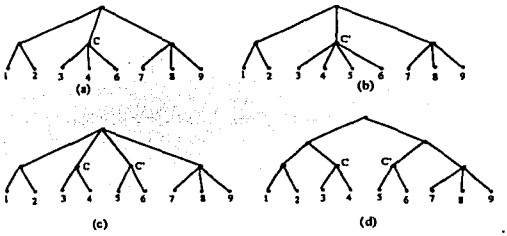


Figura 5.18: Inserción de datos en un 2-3 árbol

Como el nodo c tiene ahora 4 hijos deja de cumplir la condición de un 2-3 árbol. Entonces, creamos un nodo c' y le asignamos como hijos las hojas con los valores 5 y 6. Notamos nuevamente que el árbol t deja de cumplir la condición de un 2-3 árbol, pues la raíz tiene 4 hijos. Creamos una nueva raíz con dos hijos, que nos dan como resultado un 2-3 árbol que se muestra en la Figura 6.18(d).

5.5.2. Insertando una secuencia de objetos en un 2-3 árbol

Supongamos que tenemos un 2-3 árbol T que contiene los n objetos $a_1 < a_2 < \ldots < a_n$. Consideremos el problema de insertar una secuencia de k objetos $b_1 < b_2 < \ldots < b_k$ en T, donde se asume que k es mucho menor que n. Si k es tan grande como n, entonces es más eficiente construir el 2-3 árbol.

Usando el procedimiento de inserción anterior, podemos insertar k elementos, uno a la vez, en tiempo secuencial $O(k \log n)$. A continuación describiremos un algoritmo paralelo que ejecuta la inserción de k elementos en tiempo $O(\log n)$, usando un total de $O(k \log n)$ operaciones. Un componente clave de este algoritmo es el efectivo encadenamiento de múltiples subsecuencias de inserciones.

En pocas palabras, la técnica de encadenamiento es el proceso de romper una tarea en una secuencia de subtareas. Decimos $t_1, \ldots, t_m$, tal que, una vez que t_1 es terminado, la secuencia correspondiente a una nueva tarea puede empezar y puede proceder en la misma velocidad que la tarea previa. Este proceso es similar a la operación de una linea de ensamblaje, donde una subtarea podría ser el complemento de un componente específico de un sistema, y multiples sistemas se ensamblan en línea.

Empezaremos con un preprocesado que simplifica la presentación del algoritmo. Insertaremos b_l y b_k en T usando el algoritmo de la inserción secuencial. Este paso garantiza que cada uno de los b_i's restantes se acomoda entre dos hojas consecutivas de T. Este proceso toma tiempo secuencial $O(\log n)$.

Concurrentemente, colocamos todos los elementos b_i en T aplicando el procedimiento de búsqueda sobre cada b_i por separado. Por lo tanto, para cada b_i obtenemos una hoja que contiene el objeto $a_{i'}$ tal que $a_{i'} \leq b_i < a_{i'+1}$. Esta búsqueda toma $O(\log n)$ pasos en paralelo usando un total de $O(k \log n)$ operaciones.

5.6. La Técnica Aceleramiento en Cascada

Sea $X = \{x_1, x_2, \ldots, x_n\}$ un conjunto de elementos tomados de un conjunto S linealmente ordenado. El problema de calcular el **máximo** elemento de X consiste en encontrar un elemento x_i tal que $x_i \geq x_j$, para todo J, con $1 \leq j \leq n$.

El problema básico puede ser resuelto por un simple algoritmo secuencial en tiempo lineal. El modelo **PRAM** ofrece un interesante caso de estudio para introducir la estrategia de **Aceleramiento en Cascada**.

Asumimos que las x_i's son distintas. De otra manera, podemos reemplazar x_i por el par (x_i, i), para cada i, extendemos la relación " $\geq$" como sigue:

$$(x_i, i) > (x_j, j) \text{ si y solo si } x_i > x_j, \text{ o } x_i = x_j \text{ y } i > j.$$

Podemos determinar el máximo elemento usando un árbol binario balanceado construido sobre la entrada de n elementos, sólo como en el caso del cálculo de la suma de n elementos. El tiempo de ejecución del algoritmo en paralelo es de $O(\log n)$, y el número total de operaciones usadas es de $O(n)$. Este algoritmo paralelo es óptimo.

Podemos obtener un algoritmo más **veloz** usando un esquema basado en un árbol computacional de profundidad logarítmica doble. La idea principal es construir un árbol balanceado con las x_i's como las hojas tales que el número de hijos de un nodo u es igual a $\lceil \sqrt{n_u} \rceil$, donde n_u es el número de hojas en el subárbol enraizado en u. Cada nodo interno será usado para tomar el máximo elemento del subárbol. La condición en el número de hijos fuerza al árbol a tener una profundidad logarítmica doble. El éxito de esta estrategia depende de la existencia de un algoritmo en tiempo constante que ejecute la operación representada por un nodo interno, es decir, un algoritmo que calcule el máximo de un número arbitrario de elementos. Nuestra siguiente tarea es desarrollar tal algoritmo.

5.6.1. Un algoritmo de tiempo constante para calcular el máximo.

Sea A un arreglo de p elementos tomados de un universo S linealmente ordenado. El propósito del siguiente algoritmo es ejecutar las comparaciones entre todos los pares de elementos de A. El máximo elemento puede ser identificado como el único elemento que sale como ganador en todas las comparaciones.

En el Algoritmo MÁXIMO_BASE, el arreglo lógico B toma las salidas de todas las comparaciones, y el símbolo "&" representa la operación lógica **AND**.

```
Maximo_Basico
        // ENTRADA: Un arreglo A, de p distintos elementos
        // SALIDA: Un arreglo Booleano M tal que
        // M(i)=1 si y solo si A(i) es el maximo elementos de A

begin
        1. for i<=i,j<=p pardo if (A(i) =>A(j))
                then set B(i,j):=1
                else set B(i,j):=0

        2. for i<=i<=p pardo
                set M(i):=B(i,1) & B(i,2) & ... & B(i,p)
end
```

Figura 5.19: ALGORITMO MAXIMO_BASE

5.6.2. Un algoritmo de tiempo logarítmico doble.

Dado un árbol enraizado, el **nivel** de un nodo u es el número de aristas en la trayectoria entre u y la raíz del árbol. Por lo tanto el nivel de la raíz en 0.

Por simplicidad asumamos que $n = 2^{2^k}$, para algún entero k. Definimos el **Árbol de Profundidad Logarítmica Doble** con n hojas de la siguiente manera. La raíz del árbol tiene $2^{2^{k-1}}$ (esto es, $\sqrt{n}$), hijos, cada uno de ellos tiene $2^{2^{k-2}}$ hijos, y en general, cada nodo en el i-ésimo nivel tiene $2^{2^{k-i-1}}$ para $0 \leq i \leq k-1$. Cada nodo en el nivel k tendrá dos hojas como hijos.

Ejemplo. El árbol de profundidad logarítmica doble para el caso cuando $n = 16$ se muestra en la Figura 6.20. La raíz tiene cuatro hijos, y cada uno de los nodos internos tiene dos hijos. Cada nodo interno corresponde al cálculo del máximo de sus nodos hijos.

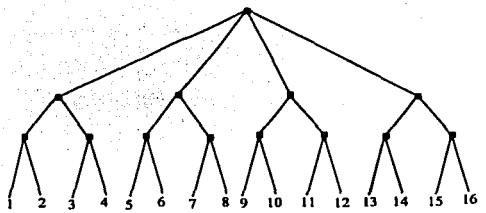


Figura 5.20: Árbol de profundidad logarítmica doble

Un argumento inductivo sobre i muestra el número de nodos en el i-ésimo nivel del árbol de profundidad logarítmica doble es de $2^{2^k-2^{k-i}}$, para $0 \leq i < k$. El número de nodos en el k-ésimo nivel es de $2^{2^k-1} = (n/2)$. Además, la profundidad de este árbol es $(k+1) = (\log\log(n+1))$.

Este árbol puede ser usado para calcular el máximo de n elementos. Cada nodo interno toma el máximo de los elementos almacenados en el subárbol. El algoritmo procede nivel por nivel, de abajo hacia arriba empezando con los nodos de altura 1. Usando el algoritmo anterior el máximo requerido para cualquier nivel dado puede ser calculado en tiempo $O(1)$.

De aquí que el máximo pueda ser calculado en tiempo $T(n) = O(\log\log n)$, entonces el algoritmo para el árbol de profundidad logarítmica doble es **exponencialmente** más rápida que el algoritmo previo que lo hacia el tiempo $O(\log n)$.

Necesitamos estimar el número total de operaciones requeridas por el nuevo método. El número de operaciones requeridas por las tareas en el i-ésimo nivel es:

$$O((2^{2^{k-i-1}})^2)$$

por nodo, para $0 \leq i \leq k$, dando un total de:

$$O((2^{2^{k-i-1}})^2 \cdot 2^{2^k-2^{k-i}}) = O(2^{2^k}) = O(n)$$

operaciones por nivel. De aquí que el número total de operaciones requeridas para todos los cálculos es $W(n) = O(n\log\log n)$, lo que lo hace un algoritmo no óptimo.

5.6.3. Haciendo un algoritmo rápido y óptimo.

Hemos introducido dos algoritmos para calcular el máximo. El primero basado en el árbol binario de profundidad logarítmica doble, el cual es óptimo y se ejecuta en tiempo

logarítmico, el segundo algoritmo no es óptimo pero se ejecuta en tiempo doblemente logarítmico, que es muy veloz. En cuyo caso podemos tratar de usar una estrategia llamada **Aceleramiento en Cascada**, la cual combina los dos algoritmos en un algoritmo rápido y óptimo. Primero describamos, en terminos generales, la estrategia de aceleramiento en cascada:

- Primero empezaremos con el algoritmo óptimo hasta que el tamaño del problema sea reducido a cierto valor.
- A continuación cambiaremos al algoritmo veloz pero no óptimo.
- Después estudiaremos la adaptación de esta estrategia al problema de calcular el máximo de n elementos.

En la Fase 1, aplicamos el algoritmo del árbol binario, empezando desde las hojas y moviéndose hacia arriba en $\lceil \log\log\log n \rceil$ niveles. Ya que el número de candidatos reduce por un factor de $(1/2)$ por nivel mientras escalamos el árbol binario, sabemos que el máximo esta entre el $n' = O(n/\log\log n)$ elementos generados al final del algoritmo del árbol binario. El número total de operaciones usadas hasta aqui es $O(n)$, y el tiempo correspondiente es $O(\log\log\log n)$.

Durante la Fase 2, usamos el árbol de profundidad logarítmica doble basados en los n' elementos generados durante la Fase 1. Esta segunda fase requiere tiempo $O(\log\log n') = O(\log\log n)$ y usa un total de $W(n) = O(n'\log\log n') = O(n)$ operaciones. Por lo tanto el algoritmo es óptimo y se ejecuta en tiempo $O(\log\log n)$

5.7. La Técnica Symetry Breaking

Sea $G = (V, E)$ un ciclo dirigido, esto es, el grado de entrada y el grado de salida de cada vértice es 1, y para cualesquiera dos vértices $u, v \in V$, existe una ruta dirigida desde u hasta v. Una **k-coloración** de G es una función $c : V \to \{0, 1, \ldots, k-1\}$ talque $c(i) \neq c(j)$ si $\langle i, j\rangle \in E$. Estamos interesados en el problema de determinar una 3-coloración de G.

Un algoritmo sencillo consistiría en recorrer el ciclo desde un vértice arbitrario, asignado alternadamente los colores $\{1, 0\}$ a los vértices adyacentes. Un tercer color sería necesario para terminar el ciclo. Este sencillo algoritmo secuencial es claramente óptimo, pero no se ve muy claramente como hacerlo más rápido con un algoritmo en paralelo.

La principal dificultad a la que nos enfrentamos para resolver el problema de la rapidez del algoritmo de coloración usando paralelismo, es la aparente simetría del problema. Asignando un color a muchos vértices en paralelo implica que esos vértices de alguna manera han sido distinguidos de los otros vértices. Pero todos los vértices lucen iguales.

Entonces hay que introducir un mecanismo para particionar los vértices en clases, tal que cada clase pueda asignarsele el mismo color.

5.7.1. Un algoritmo Básico de Coloración.

Asumamos lo siguiente recordando la representación de la entrada de un ciclo dirigido $G = (V, E)$: Los arcos de G están especificados por un arreglo S de longitud n, tal que $S(i) = j$, siempre que $\langle i, j \rangle \in E$, para $1 \leq i, j \leq n$. Note que podemos obtener inmediatamente la relación **predecesor** asignando $P(S(i)) = i$, para $1 \leq i \leq n$.

Suponga que una coloración inicial c de los vértices de G está dada. Dada la expansión binaria de un entero i –decimos, $i = i_{t-1} \cdots i_k \cdots i_1 i_0$– el **$k$-ésimo bit menos significativo** de i es el bit i_k. Podemos reducir el número de colores usando el Algoritmo COLORACION_BASICO que toma ventaja de la representación binaria de los colores.

```
Coloracion_Basico
        // ENTRADA: Un ciclo dirigido cuyos arcos estan
        // especificados por un arreglo S de tamano n y una
        // coloracion c de los vertices.
        // SALIDA: Otra coloracion c' de los vertices del ciclo.

begin

        1. for i<=i<=n pardo
                1. coloca a k en la posicion del bit menos
                   significativo en el cual c(i) y c(S(i)) difieran
                2. coloca c'(i):=2k+c(i)_k

end
```

Figura 5.21: ALGORITMO COLORACION_BASICO

Ejemplo. Sea $c(i) = i$ el color asignado a los vértices del ciclo dirigido mostrados en la Figura 6.22. Para cada vértice, listaremos el valor de k y el correspondiente color nuevo, en dos columnas por separado. El número de colores es reducido a seis en este caso.

v	c	k	c'
1	0001	1	2
3	0011	2	4
7	0111	0	1
1	1110	2	5
2	0010	0	0
15	1111	0	1
4	0100	0	0
5	0101	0	1
6	0110	1	3
8	1000	1	2
10	1010	0	0
11	1011	0	1
12	1100	0	0
9	1001	2	4
13	1101	2	5

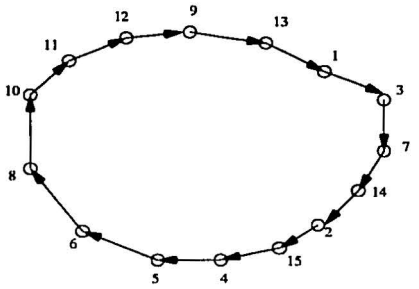


Figura 5.22: Ejemplo de coloración básico en un ciclo dirigido

5.7.2. Un algoritmo 3-coloración muy rápido.

Empezaremos por estimar el número de colores generados por el algoritmo mostrado en la Figura 6.21, en función del número de colores iniciales.

Sea $t > 3$ el número de bits usados para representar cada color en la coloración inicial c. Entonces, cada color usado por c' puede ser representado con $\lceil \log t \rceil + 1$ bits. Entonces,

si el número de colores en c es q, donde q satisface $2^{t-1} < q \leq 2^t$, colorando c' usaremos a lo más $2^{\lceil \log t \rceil + 1} = O(t) = O(\log q)$ colores. Entonces el número de colores decrece exponencialmente, en general.

El procedimiento de coloración básico puede ser aplicado repetidamente. Tal reducción del número de colores ocurre mientras el número t de bits usados para representar los colores satisface $t > \lceil \log t \rceil + 1$, es decir, $t > 3$. Aplicando el procedimiento de coloración básico al caso donde $t = 3$ resultara en una coloración de a lo más seis colores, ya que el nuevo color i está dado por $c'(i) = 2k + c(i)_k$, y de aquí $0 \leq c'(i) \leq 5$, pues $0 \leq k \leq 2$. Ahora estimaremos el número de iteraciones necesitadas para alcanzar este escenario.

Antes de presentar el análisis sobre el número de iteraciones, introduciremos la siguiente notación. Sea $\log^{(i)} x$ definido por $\log^{(1)} x = \log x$, y $\log^{(i)} x = \log(\log^{(i-1)} x)$. Por ejemplo, $\log^{(2)} x = (\log \log n)$, y $\log^{(3)} x = (\log \log \log x)$. Sea $\log^* x = min\{i | \log^{(i)} x \leq 1\}$. La función $(\log^* x)$ es una función que crece muy lentamente y está acotada por 5 para toda $x \leq 2^{65536}$.

Empezando con la coloración inicial $c(i) = i$ para $1 \leq i \leq n$, la primera aplicación del Algoritmo COLORACION_BASE reduce el número de colores a $O(\log n)$ colores. La segunda aplicación reduce el número de colores a $O(\log \log n) = o(\log^{(2)} n)$. De aqui que el número de colores se reduzca a menor o igual a seis colores después de $O(\log^* n)$ iteraciones.

Podemos reducir el número de colores a tres. El siguiente procedimiento de recoloración consiste de tres iteraciones, cada una de las cuales maneja vértices de un color específico. Para $3 \leq l \leq 5$, hacemos: Para cada vértice i con color l, recoloreamos el vértice i con el más pequeño color posible de $\{0, 1, 2\}$. Cada iteración toma tiempo $O(1)$ usando $O(n)$ operaciones. Después de la última iteración, obtenemos una 3-coloración en los tres limites asintóticos.

5.7.3. Un algoritmo óptimo para la 3-coloración.

El algoritmo de 3-coloración previo rompe la simetría dentro de un ciclo dirigido muy rápido, pero usa un número no lineal de operaciones. Sin embargo, para todos los propósitos prácticos, este algoritmo toma menos de seis pasos en paralelo, cada uno de los cuales envuelve una simple operación en cada uno de los vértices.

Si insistimos en optimizar, podemos usar una variación del procedimiento. Recordemos: Ordenar n enteros, cada uno de ellos dentro del rango $[0, O(\log n)]$, puede realizarse en tiempo $O(\log n)$ usando un número lineal de operaciones. El ordenamiento puede lograrse usando una combinación del algoritmo *radix-sort* y el algoritmo de SUMA_PREFIJA.

Nuestro algoritmo óptimo consiste de:

- colorear los vértices con $O(\log n)$ colores usando el procedimiento básico de coloración;
- ordenar los vértices por su color, todos los vértices con el mismo color aparecen consecutivamente;
- aplicando el procedimiento de recoloración sucesivamente a cada conjunto de vértices con el mismo color.

Los detalles se muestran en el Algoritmo 3-COLORACION_BASICO.

```
Coloracion_Basico
                // ENTRADA: Un ciclo dirigido de longitud n cuyos
                // arcos estan especificados por un arreglo S
                // SALIDA: Una 3-coloracion de los v\'ertices de un ciclo

        begin

                1. for i<=i<=n pardo
                   coloca C(i):=i

                2. Aplica el algoritmo de coloracion basica.

                3. Ordena los vertices por sus colores.

                4. for i=3 to 2*log(n) do
                        for todos los vertices de color i pardo
                        color v con el mas pequeno color
                        de {0,1,2} si es diferente de los colores de sus 2
                        vecinos

        end
```

Figura 5.23: ALGORITMO 3-COLORACION_BASICO

Capítulo 6

Listas y árboles

Las listas y los árboles son estructuras de datos básicas, frecuentemente usadas en múltiples procesos. En esta sección estudiaremos algunos problemas básicos sobre el procesamientos de listas y sobre cálculos de funciones en árboles.

Una tarea elemental sobre procesos enlistados es el asignar un número de posición a los nodos, desde el principio hasta el fin de la lista, un problema que conocemos como el **asignar rangos a los elementos de una lista**. Un algoritmo secuencial en tiempo lineal puede ser desarrollado fácilmente. Presentaremos dos algoritmos paralelos para solucionar el problema de asignar rangos a los elementos de una lista, cada uno de ellos usando un número lineal de operaciones.

Para los árboles, introduciremos dos poderosas técnicas: **Las rutas de Euler** y **La contracción de árboles**.

6.1. Asignación de rango a listas

Consideremos una lista ligada L de n nodos cuyo orden es especificado por un arreglo S tal que $S(i)$ contiene un apuntador al nodo siguiente del nodo i en L, para $1 \leq i \leq n$. Asumimos que $S(i) = 0$ cuando i es el ultimo de la lista. Los datos almacenados en los nodos no juegan ningún papel importante. **El problema de asignación de rango a una lista** consiste en determinar la distancia de entre cada nodo i y el nodo final de la lista. A tal distancia le denominaremos el rango de i.

Este es un problema elemental en procesamiento de listas, cuya complejidad secuencial es trivialmente lineal. La técnica *Pointer Jumping* puede ser usada para derivar un algoritmo paralelo para la asignación de rango a una lista. El correspondiente tiempo de ejecución es $O(\log n)$ y el total de operaciones es $O(n \log n)$. Este algoritmo no es óptimo en vista de la existencia de un algoritmo secuencial de tiempo lineal.

6.1.1. Un algoritmo sencillo para la asignación de rango a listas de manera óptima

Dada una lista ligada L con n nodos, nos gustaría calcular un arreglo R tal que $R(i)$ es igual a la distancia del nodo i al final de L. Inicialmente, colocamos $R(i) = 1$ para todos los nodos i, excepto para el ultimo nodo, cuyo valor de R es 0.

Empezaremos estableciendo el algoritmo en paralelo basado en la técnica de Pointer Jumping. Este algoritmo es esencialmente el mismo que fué dado para los arboles enraizados dirigidos.

```
Asignacion_de_Rango a Listas usando Pointer Jumping
        // ENTRADA: Una lista ligada con n tal que
        // 1. El sucesor de cada nodo i esta dado
        // por S(i)
        // 2. El valor de S del ultimo nodo de la
        // lista es igual a 0
        // SALIDA: Para cada i con i<=i<=n la distancia
        // R(i) del nodo i al final de la lista

begin
      1. for i<=i<=n pardo
            if S(i) != 0 then set R(i):=1
            else set R(i):=0

      2. for i<=i<=n pardo
            set Q(i):=S(i)
            while (Q(i) != 0 && Q(Q(i) != 0)) do
                  set R(i):=R(i)+R(Q(i))
                  set Q(i):=Q(Q(i))

end
```

Figura 6.1: ALGORITMO ASIGNACION_DE_RANGO

La estrategia completa para resolver el problema de asignación de rango a listas de manera óptima se presenta a continuación:

Estrategia para la asignación de rango a Listas.

- Reduciremos la lista ligada L hasta que sólo queden $O(n/\log n)$ nodos restantes
- Aplicaremos la técnica de *Pointer Jumping* en la lista pequeña de nodos restantes
- Reestableceremos la lista original y asignaremos rango a todos los nodos borrados en el paso 1.

Hasta el momento sólo hemos examinado cómo implementar el segundo punto de esta estrategia. Ya que el número de elementos es $O(n/\log n)$ el Algoritmo ASIGNACION_RANGO toma tiempo $O(\log n)$ usando un total de $O(n)$ operaciones.

Conjuntos independientes. El método para la reducción de la lista L consiste en borrar un conjunto seleccionado de nodos de L y actualizar los valores intermedios R de los nodos restantes. La clave para implementar un algoritmo paralelo veloz recae en el uso de un conjunto independiente de nodos que serán borrados. Un conjunto de nodos I se dice que es **independiente**, si siempre que $i \in I, S(i) \notin I$. Podemos borrar cada nodo $i \in I$ ajustando el apuntador al sucesor del predecesor de i. Ya que I es independiente, el proceso puede ser ejecutado concurrentemente para todos los nodos en I.

La información con respecto a los nodos borrados será almacenada en algún lugar para que después la lista original pueda ser restaurada y los nodos en I se les pueda asignar rango apropiadamente. En el siguiente procedimiento, usaremos el arreglo U para almacenar dicha información. Necesitaremos un arreglo por separado, ya que el procedimiento de contracción de listas, ilustrado en el Algoritmo BORRANDO_NODOS compactará los elementos restantes en la memoria de manera consecutiva. Sin pérdida de generalidad, asumamos que el arreglo predecesor P esta disponible. De otro modo, éste podrá ser ejecutado en tiempo $O(1)$ usando un total de $O(n)$ operaciones.

```
Borrando_nodos de un conjunto independiente
        // ENTRADA:
        // 1. Arreglos S y P de longitud n representando
        // respectivamente las relaciones de sucesor y
        // predecesor de una lista ligada
        // 2. Un conjunto independiente I de nodos tal
        // que P(i), S(i) != 0
        // 3. Un valor R(i) para cada nodo i
        // SALIDA: La lista obtenida despues de borrar
        // todos los nodos en I con las actualizaciones
        // de los valores de R

begin
      1. Seriamos consecutivamente N(i) a los elementos
         de I, donde 1<=N(i)<= |I|=n'

      2. for all i en I pardo
            set U(N(i)):=(i,S(i),R(i))
            set R(P(i)):=R(P(i))+R(i)
            set U(N(i)):=S(i)
            set U(N(i)):=P(i)

end
```

Figura 6.2: ALGORITMO BORRANDO_NODOS

Ejemplo. La Figura 7.3 ilustra el proceso de contracción de listas para un conjunto

independiente dado. La lista inicial y los valores iniciales de R se muestran en la Figura 7.3(a) encerrados entre corchetes. Un conjunto independiente de nodos consistentes de $\{1, 5, 6\}$. Paso 1 asignamos a los nodos $\{1, 5, 6\}$ los números seriados:

$$N(1) = 1, N(5) = 2, N(6) = 3.$$

Ejecutando el ciclo del paso 2 para el nodo 6, obtenemos:

$$U(N(6)) = U(3) = (6, 8, 2), R(2) = 4, S(2) = 8, yP(8) = 2.$$

Procedemos similarmente para los nodos 5 y 1. Las nuevas ligas y los nuevos valores de R de las listas contraidas se muestran en la Figura 7.3(b). Supongamos que tenemos los rangos de todos los nodos de la lista corta considerada para la actualiización de los valores de R. Ya que $U(3) = (6, 8, 2)$, concluimos que el nodo 8 es el sucesor inicial del nodo 6 y por lo tanto $R(6) = R(8) + 2 = 3$. Mas aún, podemos reinsertar este nodo poniendo

$$P(6) = P(8) = 2, \quad S(P(6)) = S(2) = 6, \quad P(8) = 6.$$

Recordatorio 1. Supongamos que queremos ejecutar el Algoritmo BORRANDO_NODOS siendo ejecutado por un conjunto de p procesadores, donde $p \leq |i|$. Entonces los n' elementos de i se dividen casi uniformemente los p bloques, cada uno de los cuales se manejará separadamente por un procesador. Cada procesador puede meter a cada pila privada la información que concierne a los nodos que borra. La restauración de la lista se logra sacando cada pila.

Recordatorio 2. Si después de borrar los nodos del conjunto independiente I, los nodos restantes no tienen que ser reubicados, el paso 1 del Algoritmo BORRANDO_NODOS es innecesario; de aquí, que el resultado del algoritmo sea en tiempo $O(1)$, usando $O(n)$ operaciones.

Determinación de un conjunto independiente. Podemos manipular el problema de manejar un conjunto independiente, por medio de la coloración de los nodos de la lista L. Renombramos a esa k-coloración de L como una función del conjunto de nodos de L en $\{0, 1, \ldots, k-1\}$ tal que vértices no adyacentes se les asigna el mismo color. Una k-coloración de L puede ser usada para determinar un conjunto independiente de L de la siguiente manera. Un nodo u es un **mínimo local** con respecto a su coloración, si el color de u es el más pequeño que los colores de su predecesor y su sucesor. De manera similar se define un máximo local.

Lema: Dada una k-coloración de los nodos de una lista L de tamaño n, el conjunto I de mínimos locales es un conjunto independiente de tamaño $\Omega(n/k)$. El conjunto I puede ser identificado en tiempo $O(1)$, usando un número lineal de operaciones.

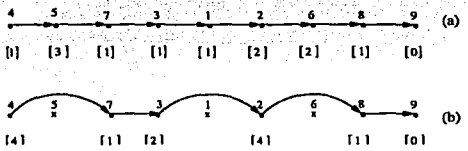


Figura 6.3: Ejemplo de contracción de una lista

Un algoritmo sencillo para asignar rango una lista. En la Figura 7.4 daremos una descripción del Algoritmo sencillo, pero óptimo para la asignación de rango a listas.

Ejemplo: Considere la lista de ocho nodos que se muestra en la Figura 7.5(a), donde el peso de cada nodo se muestra entre corchetes bajo el nodo. Durante la primera iteración del ciclo **while**, identificamos el conjunto independiente $I = \{3, 4, 2, 5\}$ derivado de la 3−coloración mostrada entre paréntesis. Borrando los nodos en I de la lista original obtenemos la nueva lista mostrada en la Figura 7.5(b) con el ajuste de pesos. Observe que la información concerniente a los nodos borrados puede ser almacenada en tripletas con un etiqueta adicional donde se indique el número de iteración. Por ejemplo, la información relacionada con el nodo 4 puede ser almacenada como:

$$U(1, \dot{N}(4)) = (4, S(4), R(4)) = (4, 1, 1),$$

donde el primer parámetro de U, en este caso, 1 indica el número de iteración y el segundo, en este caso, N(4), indica el número seriado asignado al nodo que será borrado.

Durante la segunda iteración, obtenemos la lista mostrada en la Figura 7.5(c). Ya que $n_2 = 2$, vamos al paso 3 y obtenemos:

$$R(6) = R(6) + R(7) = 7, \quad y \quad R(7) = 3.$$

La restauración de la lista original se obtiene en dos iteraciones. Después de la primera iteración, que corresponde a la segunda iteración del ciclo **while**, obtenemos la lista mostrada en la Figura 7.5(d). Es claro que la segunda iteración restaurará la lista original con los rangos correctos asignados a todos los nodos.

6.1.2. Un Algoritmo óptimo para Asignar Rango a Listas en Tiempo $O(\log n)$

La debilidad del algoritmo sencillo para asignar rango a listas visto en el Algoritmo ASIGNACION_RANGO recae en la ejecución de las operaciones costosas para identificar un

```
Algoritmo optimo y sencillo para la asignar rango a listas
        // ENTRADA:
        // 1. Una lista ligada con n nodos tal que
        // el sucesor de cada nodo esta dado por S(i)
        // SALIDA: Para cada nodo i, la distancia de i
        // al final de la lista
 begin
      1. Asigna n_0:=n; k:=0

      2. while n_k > n/logn do
         2.1 set k:=k+1
         2.2 Colorea la lista con tres colores e
             identifica el conjunto I de minimos
             locales
         2.3 Borra los nodos en I, y almacena la
             informacion apropiada recordando los
             nodos borrados
         2.4 sea n_k el tamano de la lista restante
             compacta la lista dentro de bloques
             consecutivos de memoria

      3. Aplica la t\'ecnica de pointer jumping a la
         lista resultante

      4. Reestablece la lista original y asigna
         rango a todos los nodos borrados
         invirtiendo el proceso ejecutado en el
         paso 2

 end
```

Figura 6.4: ALGORITMO ASIGNA_RANGO_A_LISTAS

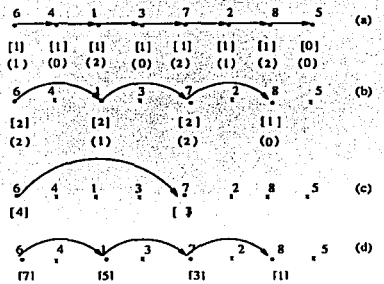


Figura 6.5: Ejemplo de una lista ligada y su procesamiento

conjunto independiente grande, y de la compactación de la lista durante cada iteración del proceso de contracción de listas, que es el paso 2. Desarrollaremos un método alterno para reducir la lista de n nodos a $O(n/\log n)$ nodos sin tener una ejecución para cada operación. Las ideas clave que hay debajo de este método se darán a continuación. Por simplicidad asumamos que $\log n$ es un entero divisible entre n.

Estrategia General. Dividamos el arreglo S en $(n/\log n)$ subarreglos $\{B_i\}$ llamados **Bloques**, cada uno conteniendo $(\log n)$ objetos. Cada bloque B_i será procesado secuencialmente, pero concurrentemente con los otros bloques. Un índice $p(i)$ será usado para apuntar a un nodo en B_i. Denotaremos este nodo por $N(p(i))$. Inicialmente asignamos $p(i) = [(i-1)\log n + 1]$, para $1 \leq i \leq (n/\log n)$, ya que $p(i)$ apunta al primer nodo en cada bloque B_i. Los bloques serán procesados a diferentes velocidades; por lo tanto requerimos de los apuntadores $p(i)$.

Consideremos los nodos iniciales $N(p(i))$ y cada etiqueta de $N(p(i))$ como **activa**, para $1 \leq i \leq (n/\log n)$; etiquetaremos cada uno de los nodos restantes como **inactivos**. Si los nodos activos forman un conjunto independiente , es decir, si ninguno de los nodos sucesores, ni los nodos predecesores para cada $N(p(i))$ está activo, entonces esos nodos pueden ser borrados de la lista inmediatamente. Podemos entonces avanzar cada apuntador $p(i)$ al siguiente lugar consecutivo del bloque B_i, es decir, asignamos $p(i) = p(i) + 1$. En general, sin embargo, sólo un subconjunto de nodos activos, llamados nodos **aislados**, formarán un conjunto independiente, mientras que el resto de los nodos activos formarán sublistas de la lista original.

Ejemplo. Un lista de 16 nodos y algunos de los apuntadores se muestran en la Figura 7.6. En este caso, tenemos cuatro bloques B_1, B_2, B_3, B_4. Inicialmente los nodos activos son $1, 5, 9, 13$; estos están apuntados por los cuatro apuntadores:

$N(p(1)) = 1$, $N(p(2)) = 5$, $N(p(3)) = 9$, $N(p(4)) = 13$. La liga de información de los nodos activos está indicada por los arcos. Por ejemplo:

$$S(1) = 6, S(5) = 13, S(9) = 5, S(13) = 16.$$

Por lo tanto, el nodo 1 es el único nodo aislado, ya que su nodo predecesor y su nodo sucesor no están activos; los nodos activos restantes de la sublista son $9 \rightarrow 15 \rightarrow 13$.

El procedimiento de contracción consiste de $O(\log n)$ etapas, en la cual cada etapa puede ser implementada en tiempo $O(1)$, usando un total de $O(n/\log n)$ operaciones. Inicialmente hay un apuntador $p(i)$ al primer nodo en B_i tal que $N(p(i))$ es etiquetado como activo. Todos los demás nodos son etiquetados como inactivos. Describiremos ahora la primera etapa para nuestro procedimiento. Las siguientes etapas son idénticas; la descripción de una se dá a continuación.

Primera Etapa. Cada nodo activo aislados $N(p(i))$ es borrado de la lista. Cada nodo activo restante es entonces parte de una sublista segura. Nótese que una sublista puede tener tamaño $\Omega(n/\log n)$. Veamos ahora como romper cada sublista en cadenas cortas en tiempo $O(1)$ usando un número lineal de operaciones, sobre el tamaño de los nodos activos restantes.

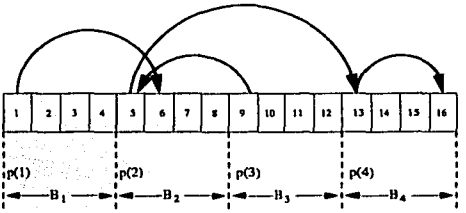


Figura 6.6: Lista particionada por el algoritmo de contracción

Aplicaremos el procedimiento básico de coloración dado en la sección anterior para cada nodo activo restante, empezando con la coloración inicial $c(i) = i$. Recordemos que este procedimiento de coloración consiste en determinar el bit menos significativo en la posición k donde $c(i)$ y $c(S(i))$ difieren, y colocar el nuevo color de i en $c' = 2k + c(i)_k$. Por lo tanto, los nodos activos restantes son colorados adecuadamente con $O(\log\log n)$ colores. Cada nodo cuyo color es un mínimo local es etiquetado como un **regidor**, y cada nodo activo restante es etiquetado como un **súbdito**. La Figura 7.7 nos ilustra que pasa con los nodos activos durante la primera etapa.

De este modo, cada uno de los nodos de una sublista es etiquetado ya sea como regidor o como súbdito. Los regidores de una sublista la particionan en **cadenas**; cada cadena consiste de un regidor r seguido por los súbditos entre r y el siguiente regidor, o el final de la sublista si el siguiente regidor no existe. Sin pérdida de generalidad, asumamos que el nodo más a la izquierda de una sublista, es decir, un nodo activo cuyo predecesor no esta activo, es un regidor, de otra manera lo convertiríamos en uno. El tamaño de cada cadena es de $O(\log\log n)$.

Terminamos la primera etapa avanzando el apuntador de cada súbdito, cada nodo borrado y etiquetando los correspondientes nodos nuevos como activos.

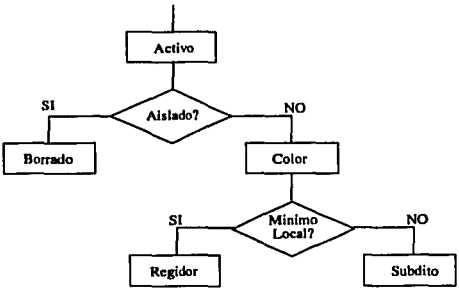


Figura 6.7: Primera etapa del algoritmo de contraccion

Etapa General. La entrada de la etapa general consiste de a lo más de $(n/\log n)$ apuntadores $p(i)$, donde cada $p(i)$ apunta a un elemento en el bloque B_i. Cada nodo $N(p(i))$ es etiquetado como activo o inactivo. Si $N(p(i))$ es un regidor, entonces el siguiente súbdito es borrado y se etiqueta como tal. El nodo $N(p(i))$ empieza como un nodo activo si el nodo borrado era el último súbdito. Ahora procederemos como en la primera etapa, a borrar los nodos activos aislados y romper cada sublista de nodos activos restantes en cadenas de longitud $O(\log\log n)$. La Figura 7.8 nos ilustra qué pasa con un regidor o con un nodo activo durante la etapa general.

Finalmente, los apuntadores de los nodos borrados y de los nodos súbditos avanzan, y los correspondientes nodos nuevos son etiquetados como activos.

Algoritmo de Contracción de Listas. El Algoritmo CONTRACCION_LISTA describe una etapa general ejecutada por el procesador P_i con $1 \leq i \leq (n/\log n)$. Este algoritmo

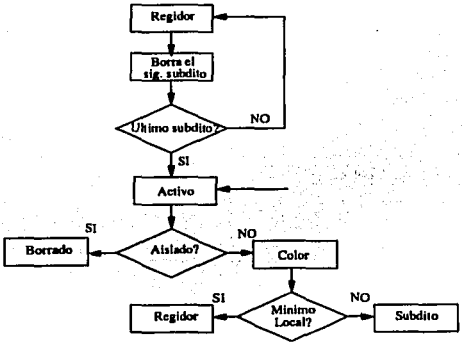


Figura 6.8: Los estados de cada nodo durante la etapa general

tiene como entrada una lista ligada con n datos dada por el Arreglo Sucesor S, el cual fue particionado en $(n/\log n)$ bloques $\{B_i\}$ tal que:

1. Un apuntador $p(i)$ a un nodo $N(p(i))$ en un bloque B_i dado.
2. Cada nodo $p(i)$ es etiquetado como activo o como regidor.
3. Los nodos restantes son etiquetados como inactivos o como gobernados.

La salida de este algoritmo es una lista contraida que tiene las siguientes caracteristicas:

1. Un objeto de cada cadena y los todos los nodos activos aislados son borrados.
2. Cada bloque B_i no vacío tiene un apuntador $p(i)$ tal que cada nodo $N(p(i))$ es etiquetado como activo o regidor.

```
Algoritmo Contraccion_Lista
        // etapa general: (Algoritmo para procesador P_i)
begin
     1. if  N(p(i)) es un regidor, borramos el siguiente
     sujeto. Etiquetamos N(p(i)) activo si este era
     el ultimo sujeto en la cadena.

     2. if N(p(i)) es activa y aislada, borramos N(p(i))
     y etiquetamos N(p(i)) como borrado.

     3. if N(p(i)) esta activo, entonces colorea N(p(i))
     aplicando el procedimiento basico de coloracion.
     Etiqueta N(p(i)) como regidor si su color es
     un minimo local. De otro modo, etiqueta N(p(i))
     como sujeto.

     4. if N(p(i)) es etiquetado como borrado o como
     sujeto, entonces asignamos p(i)=p(i)+1.
     Si p(i) cruza la frontera de B_i, entonces
     Terminamos. De otra forma, etiquetamos N(p(i))
     Como activo.

end
```

Figura 6.9: ALGORITMO CONTRACCION_LISTA

6.2. La Técnica del paseo de Euler

Sea $T = (V, E)$ un árbol dado, y sea $T' = (V, E')$ una gráfica dirigida obtenida de T donde cada arista $(u, v) \in E$ es remplazada por dos arcos $\langle u, v\rangle$ y $\langle v, u\rangle$. Ya que el grado interior de cada vértice de T' es igual al grado exterior, T' es una **Gráfica Euleriana**, es decir, si tenemos un circuito dirigido que recorre cada arco exactamente una vez. Si el **Circuito Euleriano** de T' tiene salida podemos usarlo para hacer cálculos eficientes en paralelo de múltiples funciones sobre T.

6.2.1. Circuitos Eulerianos

Un circuito euleriano $T' = (V, E')$ puede ser definido por la especificación de **la función sucesor** s que proyecta cada arco $e \in E'$ en el arco $s(e) \in E'$ que es el elemento que le sigue a e en el circuito. Una manera sencilla de definir la función sucesor es la siguiente. Para cada vértice $v \in V$ del árbol $T = (V, E)$, fijamos cierto orden sobre el conjunto de vértices adyacentes a v, denotado por $ady(v)$. Digamos $ady(v) = \langle u_0, u_1, \ldots, u_{d-1}\rangle$ donde d es el grado de v. Definimos el sucesor de cada arco $e = \langle u_i, v\rangle$ como sigue:
$s(\langle u_i, v\rangle) = \langle v, u_{(i+1)modulo\ \ d}\rangle$ para $0 \leq i \leq (d-1)$.

Ejemplo. Un árbol $T = (V, E)$ y un ordenamiento de vértices adyacentes a cada vértice v son mostrados en la Figura 7.10 (a) y (b). La función sucesor nos da la tabla mostrada en la Figura 7.10(c). Considere, por ejemplo, el vértice 5 cuyos vértices adya-

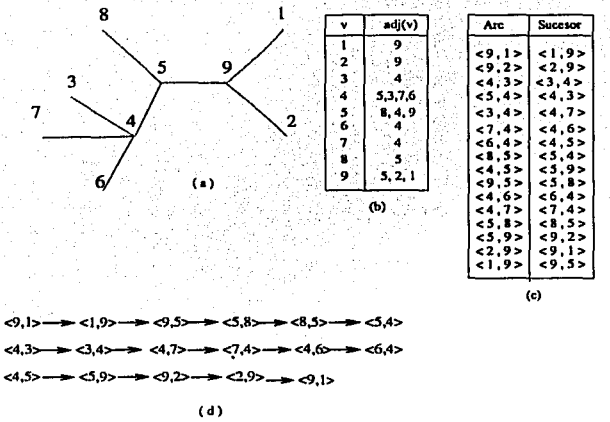


Figura 6.10: El circuito euleriano de un árbol

centes están dados por $ady(5) = \langle 8, 4, 9\rangle$. Tal lista implica que

$$s(\langle 8, 5\rangle) = \langle 5, 4\rangle, s(\langle 4, 5\rangle) = \langle 5, 9\rangle \ y \ s(\langle 9, 5\rangle) = \langle 5, 8\rangle$$

como está indicado en la tabla de sucesores. Observemos detenidamente la tabla de sucesores, donde se especifica un circuito euleriano de la gráfica dirigida $T'' = (V, E')$. Empezando con el arco $\langle 9, 1\rangle$, el circuito definido por la función sucesor se muestra en la Figura 7.10(d).

Ejemplo. Un árbol T y su lista de adyacencias con apuntadores adicionales se muestra en la Figura 7.11. Consideremos un vértice arbitrario, digamos el vértice 5. El vértice 5 aparece en la lista $L[2]$. El nodo que contiene al vértice 5 tiene un apuntador a el nodo que contiene el vértice 2. Por lo tanto podemos usar ese apuntador para deducir el arco $\langle 5, 2\rangle$ de T''. Podemos usar este nodo para concluir que el vértice 6 viene después del vértice 5 en la lista de adyacencia del vértice 2, y deducir el arco $\langle 2, 6\rangle$

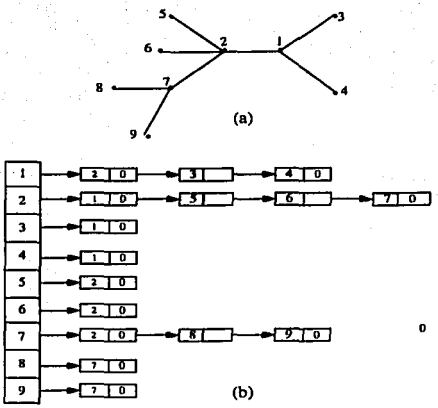


Figura 6.11: Un árbol con su correspondiente lista ligada

6.2.2. Operaciones en Árboles

Discutiremos algunas aplicaciones de la técnica de los paseos de eulerianos y el cálculo de funciones en árboles.

Enraizando un árbol. Los paseos eulerianos de un árbol $T = (V, E)$ pueden ser usados para procesar a T y calcular algunas funciones sobre T de manera eficiente. Empezaremos con el problema de **enraizar a** T en el vértice r, es decir, para cada vértice $v \neq r$, determinaremos el padre $P(v)$ de v cuando T es enraizado en r.

Denotamos $T' = (V, E')$ como la gráfica dirigida obtenida de T remplazando cada arista de E por dos arcos con direcciones opuestas. Definimos el paseo de euleriano de T calculando la función sucesor s. Sea $L[r]$ la lista de adyacencia del vértice r, donde $L[r] = \langle u_0, u_1, \ldots, u_{d-1} \rangle$. Rompiendo el paseo euleriano en r y asignando $s(\langle u_{d-i}, r \rangle) = 0$. Entonces tenemos un camino euleriano dirigido en T' que empieza en r, visita exáctamente una vez cada nodo, y termina en r.

Considere la lista ordenada EP de los arcos sobre un camino euleriano

$$EP = (e_1 = \langle r, u_0 \rangle, e_2 = s(e_1), s(e_2), \ldots, \langle u_{d-1}, r \rangle).$$

La lista EP puede ser vista como el proceso de recorrer un árbol T de la siguiente manera. Empecemos por visitar la raíz r, a continuación visitamos al vértice u_0 adyacente a r. El

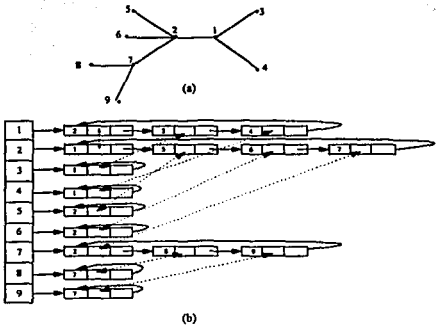


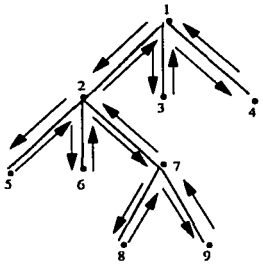
Figura 6.12: Un árbol con su lista de adyacencia

mismo proceso de búsqueda es aplicado al subárbol enraizado en u_0. Una vez que todos los vértices del subárbol enraizado en u_0 son explorados, entonces procedemos a visitar a u_1, y procedemos de la misma forma. Note que el recorrido transversal inducido por el camino euleriano EP es una **búsqueda a profundidad (DFS)** en el árbol T empezando en el vértice r. En particular confirmamos que el arco $\langle p(u), u\rangle$ aparece en la lista EP antes del arco $\langle u, p(u)\rangle$ cuando el árbol T está enraizado en r.

Ejemplo. Considere de nuevo el árbol $T = (V, E)$ mostrado en la Figura 7.12(a). Supongamos que queremos enraizar a T en el vértice 1. Rompemos el correspondiente circuito euleriano asignando $s(\langle 4, 1\rangle) = 0$. Empezando desde la raíz 1, el camino euleriano consiste de una primera búsqueda transversal de los vértices del árbol como sigue:

$$1, 2, 5, 2, 6, 2, 7, 8, 7, 9, 7, 2, 1, 3, 1, 4, 1.$$

En la Figura 7.13 podemos observar, por ejemplo, que el arco $\langle 2, 7\rangle$ aparece antes que el arco $\langle 7, 2\rangle$ sobre el camino euleriano, y que 2 es padre de 7.



(a)

Ruta Euleriana	Pesos	Sumas Prefijas
<1,2>	0	0
<2,5>	0	0
<5,2>	1	1
<2,6>	0	1
<6,2>	1	2
<2,7>	0	2
<7,8>	0	2
<8,7>	1	3
<7,9>	0	3
<9,7>	1	4
<7,2>	1	5
<2,1>	1	6
<1,3>	0	6
<3,1>	1	7
<1,4>	0	7
<4,1>	1	8

(b)

Figura 6.13: El árbol con su camino euleriano

Numeración Postorden. Otra aplicación para la técnica de los paseos eulerianos es el procesamiento de vértices de un árbol en cierto orden. Supongamos, por ejemplo que queremos determinar el postorden transversal del árbol T enraizado en r. El **postorden transversal** de T consiste de los postordenes transversales de los subárboles de r de izquierda a derecha, seguidos de la raíz r. Para nuestro árbol enraizado $T = (V, E)$, definimos el orden de izquierda a derecha de los hijos de un vértice como el orden implícito por el camino euleriano EP. Es decir, el hijo de un vértice v es visitado de izquierda a derecha sobre el camino euleriano EP. Observe que los hijos de la raíz r aparecen ordenados de izquierda a derecha en la lista de adyacencia $L[r]$, pero ese orden no necesariamente ocurre para los nodos restantes.

El objetivo es determinar el **número de postorden** $post(v)$ para cada vértice, esto es, el rango de los v's en postorden transversal de T.

El camino euleriano EP puede ser usado para determinar el postorden transversal de T de la siguiente manera. El camino EP visita cada vértice v en diferentes ocasiones, la primera vez usando el $arco\langle p(v), v\rangle$ y la última vez usando el $arco\langle v, p(v)\rangle$ después de visitar a todos los descendientes de v. Por lo tanto, el orden de la sublista de vértices obtenidos por la retensión de sólo la última ocurrencia de cada vértice definido precisa-

```
Enraizando_un_arbol
        // ENTRADA:
        // 1. Un arbol T definido por la lista de
        // adyacencia de sus vertices
        // 2. Un camino euleriano definido por la
        // funcion sucesor s
        // 3. Un vertice especial r
        // SALIDA: Para cada vertice v!=r, el padre p(v)
        // de v en el arbol enraizado en r

begin
      1. identificamos el ultimo vertice u que aparece
      en la lista de adyacencia de r.
      Asignamos s(<u,r>)=0.

      2. Asignamos peso 1 a cada arco <x,y>, y aplicamos
      el prefijo paralelo en la lista de arcos
      Definida por s.

      3. Para cada arco <x,y>, asignamos x=p(y) siempre
      que la suma prefija de <x,y> sea menor que
      la suma prefija de <y,x>

end
```

Figura 6.14: ALGORITMO ENRAIZANDO_ARBOL

mente el postorden transversal de los vértices de T. El número de postorden de cada vértice es calculado por el Algoritmo ENRAIZANDO_ARBOL.

Ejemplo. Considere el árbol de la Figura 7.13(a) con su paseo euleriano. El peso y la suma prefija de cada arco sobre el correspondiente camino euleriano se muestra en la Figura 7.13(b). De la suma prefija obtenemos lo siguiente:

$$post(5) = 1,\ post(6) = 2,\ post(8) = 3,\ post(9) = 4,\ post(7) = 5,$$

$$post(2) = 6\ , post(3) = 7,\ post(4) = 8,\ post(1) = 9.$$

Calculando el nivel de los vértices. Otra operación muy usada es calcular el nivel de cada vértice v. Lo denotamos $level(v)$. El nivel de un vértice es la distancia, o el número de aristas, entre v y la raíz r. Nuevamente usamos el camino euleriano de T para enraizarlo en r. Asignamos $w(\langle p(v), v\rangle) = +1$, a $w(\langle v, p(v)\rangle) = -1$, y ejecutamos el prefijo paralelo sobre la lista definida por el camino euleriano. Entonces, asignamos a $level(v)$ el resultado de la suma prefija de $\langle p(v), v\rangle$.

```
Numeracion_postorden
        // ENTRADA:
        // 1. Un arbol binario enraizado con
        // raiz r.
        // 2. Un camino euleriano correspondiente
        // definido por la funcion s.
        // SALIDA: El numero postorden post(v)
        // de cada vertice v.

begin
      1. Para cada vertice v!=r, asignamos los
      pesos w(<v,p(v)>)=1 y w(<p(v),v>)=0.

      2. Ejecutamos el prefijo paralelo en la
      lista de arcos definida por s.

      3. Para cada vertice v!=r, asignamos
      post(v) igual a la suma prefija de
      <v,p(v)>. Para v=r, asignamos post(r)=n
      donde n es el numero de vertices en el
      arbol dado.
end
```

Figura 6.15: ALGORITMO NUMERACION_POSTORDEN

6.3. Contracción de árboles

En la sección anterior, presentamos la técnica de los paseos eulerianos y algunas de sus aplicaciones. Sin embargo, existen problemas importantes en árboles que no pueden ser resueltos de manera eficiente con esta técnica por sí sola. Consideremos por ejemplo **el problema de evaluar una expresión.**

Una expresión aritmética dada puede representarse con un árbol binario tal que una constante es asociada con cada hoja, y un operador aritmético (como $+, -, \times, \div$) es asociado con cada nodo interno. El objetivo es calcular el valor de la expresión en la raíz. En algunas ocasiones sólo estaremos interesados en calcular todas las subexpresiones definidas en los nodos internos. A este tipo de problemas los podemos manejar eficientemente con la **contracción del árbol.**

La contracción del árbol es una manera sistemática de reducir un árbol en un vértice simple aplicando sucesivamente la operación de mezclar una hoja con su padre o mezclar un vértice de grado 2 con su padre. El problema de la evaluación de una expresión proporciona una justificación para tal proceso. Una hoja u toma un valor constante, el cual podrá ser incorporado en los datos almacenados por el padre $p(u)$. Borrando un vértice v de grado 2 equivale a posponer la evaluación de la subexoresión v e incorporar su efecto en los datos almacenados en el padre $p(v)$.

A continuación introduciremos una operación que describe el proceso de mezcla.

6.3.1. La Operación Rastrillo (rake).

Sea $T = (V, E)$ un árbol binario enraizado en r, tal que para cada vértice v, donde $v \neq r, p(v)$ representa el padre de v. Asumamos por simplicidad que cada vértice interno tiene exactamente dos hijos.

Introduzcamos la operación primitiva llamada **rastrillo**, sobre T de la siguiente manera. Dada una hoja u tal que $p(u) \neq r$, la operación rastrillo consistirá en borrar u y $p(u)$ de T y conectar los hermanos de u, denotado por $s(u)$, a $p(p(u))$.

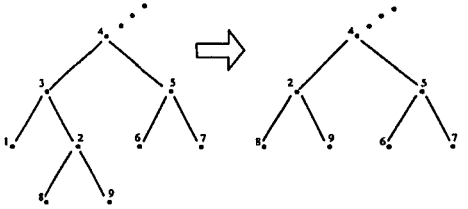


Figura 6.16: El proceso de rasurado de hojas

Ejemplo. La Figura 7.16 nos muestra qué pasa cuando la operación rastrillo es aplicada al nodo 1 del árbol del lado izquierdo. En esta operación resulta borrar los nodos 1 y 3, y hacer al nodo 4 padre del nodo 2, como se muestra en el árbol del lado derecho.

Nuestro algoritmo para la contracción del árbol usa la operación rastrillo como una operación primitiva para reducir la entrada de un árbol binario dado en un árbol de tres nodos que consiste en la raíz y dos hojas. La dificultad técnica principal recae en evitar *rasurar* de manera concurrente dos hojas cuyos padres son adyacentes. Podemos evitar tal dificultad aplicando de manera cuidadosa la operación *rasurar* hojas no consecutivas como aparecen de izquierda a derecha.

Para un arreglo dado A, un subarreglo A_{non} de A consiste de los elementos de A con índice non, es decir, $a_1, a_3, a_5, \ldots$. Definimos el subarreglo A_{par} de la manera similar. Es fácil verificar que, siempre que sea dada la longitud de A, las longitudes de A_{non} y A_{par} pueden ser determinadas en tiempo $O(1)$ usando un número lineal de operaciones.

Mostremos ahora el Algoritmo Contraccion_de_Arbol usando la operación **rastrillo**. Como entrada tenemos un arbol binario enraizado T donde cada vértice tiene exactamente dos hijos. Para cada vértice v diferente de la raíz, el padre $p(v)$ y el hermano $s(v)$. Como salida obtenemos a un árbol binario tres-nodos

```
Contraccion_de_arbol

begin
      1. Etiquetamos las hojas consecutivamente en
      orden de izquierda a derecha, excluyendo
      la hoja mas a la izquierda y mas a la
      derecha y almacenemos las etiquetas de las
      hojas en un arreglo A de tama\'no n

      2. for [log(n+1)]operaciones do
         2.1 Aplicamos la operaci\'on rastrillo
         concurrentemente a todos los elementos
        de A_{non} que son hijos izquierdos
        2.2 Aplicamos la operaci\'on rastrillo
        concurrentemente a los elementos restantes
        en A_{non}
      2.3 Asignamos A:=A_{par}
end
```

Figura 6.17: ALGORITMO CONTRACCION_DE_ARBOL

Ejemplo. Considere el árbol T dado en la Figura 7.18(a). En este caso,

$$A = (1,2,3,4,5,6,7),\ A_{non} = (1,3,5,7),\ A_{par} = (2,4,6).$$

Durante la primera iteración del ciclo **for**, la operación rastrillo es aplicada sólo al vértice 3, como es especificado en el Paso 2,1. Este paso contrae el árbol dado en el árbol mostrado en la Figura 7.18(b). El Paso 2,2 de la primera iteración da como resultado después de aplicar la operación rastrillo a los vértices $1,3,5$ el árbol mostrado en la Figura 7.18(c). Al final de la primera iteración, tenemos que $A = (2,4,6)$. En la segunda iteración del ciclo, la operación rastrillo saca las hojas 2 y 6 (paso 2,2), dando como resultado el árbol de la Figura 7.18(d). Finalmente, la tercera iteración reduce el árbol en un árbol de tres nodos como se muestra en la Figura 7.18(e).

6.3.2. Evaluación de Expresiónes Aritméticas.

Ahora mostraremos como usar el algoritmo de contracción de árboles para la evaluación de expresiones aritméticas. Supongamos que nos dan un árbol binario $T = (V, E)$, tal que cada hoja w contiene una constante c_w, y cada nodo interno u contiene cualquiera de los operadores de adición "+" o de multiplicación "×." Asumimos que cada vértice interno tiene exactamente dos hijos. Es fácil modificar el algoritmo dado en la sección anterior para los casos en donde algunos vértices tengan un solo hijo. **El problema de evaluar expresiones** consiste en determinar el valor de la expresión en la raíz de T, denotada como $val(T)$.

Un acercamiento intuitivo para la evaluación en paralelo de una expresión en un árbol, es evaluar cada subexpresión en un nodo interno v donde sus dos hijos son hojas, para

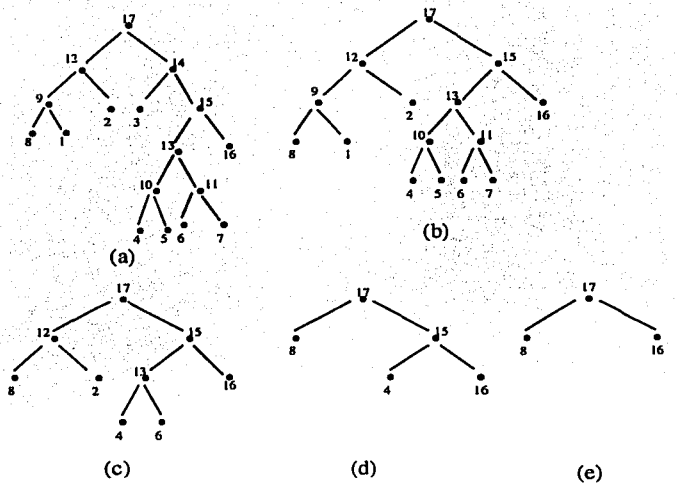


Figura 6.18: El proceso de contracción de un árbol

todos los estos nodos de manera concurrente. Repetimos este proceso hasta que el valor de la raíz queda determinado. Esta aproximación funciona siempre que el árbol este razonablemente balanceado, ya que múltiples nodos son eliminados durante cada etapa. De otra forma, algunos nodos pudieran ser borrados durante cada etapa cuando el árbol de entrada es grande y delgado. En el caso extremo, cuando tenemos una cadena larga con hojas con hojas pegadas a ella, el proceso previo requiere un número lineal de iteraciones y, por lo tanto, no proporciona un mejoramiento sobre el algoritmo secuencial.

Para remediar esta situación, relajemos la condición de que el valor de cada nodo tiene que ser totalmente determinado antes de poder ser eliminado. En lugar de esto, un nodo que cuyo único hijo es una hoja puede ser **parcialmente** evaluado y, entonces, puede ser borrado. Para lograr tal evaluación parcial, asociamos a cada nodo $v \in V$ una etiqueta (a_v, b_v) que representa la expresión lineal $a_v X + b_v$, donde a_v, b_v son constantes y X esta indeterminado esperando por el posible valor desconocido de la subexpresión en el nodo v.

Durante el proceso de evaluación de la expresión del árbol, algunos nodos son borrados y las etiquetas (a_u, b_u) de los nodos restantes son ajustados de tal modo que el siguiente

invariante se mantiene.

Invariante(I): Sea u un nodo interno del árbol actual tal que u tome el operador $\circ \in \{+, \times\}$ y sus hijos v y w cuyas etiquetas son (a_v, b_v) y (a_w, b_w), respectivamente. Entonces, el valor de la subexpresión en u está dado por $val(u) = (a_v val(u) + b_v) \circ (a_w val(w) + b_w)$.

Inicialmente, asignamos la etiqueta $(1, 0)$ a cada nodo $v \in V$. El invariante (I) resulta trivial en este caso. Usamos el algoritmo CONTRACCION_DE_ARBOL para reducir nuestro árbol T de entrada dado en un árbol de tres nodos T' tal que $val(T) = val(T')$. Incrementamos la operación rastrillo ajustando las etiquetas (a, b) para de esta manera mantener el invariante (I).

Sea v una hoja y w su hermano. Cuando aplicamos la operación rastrillo a v, el nodo v y $u = p(v)$ son borrados: por lo tanto sus contribuciones a la subexpresión calculada en el nodo $p(u)$ tendrán que ser incorporadas dentro de la pareja de valores (a_w, b_w) almacenados en w, como se muestra en la Figura 7.19. Asumamos, sin pérdida de generalidad, que v es el hijo izquierdo y u contiene el operador $\circ_u \in \{+, \times\}$. El valor del nodo u esta dado por $val(u) = (a_v c_v + b_v) \circ_u (a_w X + b_w)$, donde X es el valor desconocido del nodo w, asumiendo que el invariante(I) tome antes de la operación rastrillo. Por lo tanto, la contribución de $val(u)$ al nodo $p(u)$ está dado por $E = a_u val(u) + b_u = a_u[(a_v c_v + b_v) \circ_u (a_w X + b_w)] + b_u$, la cual es una expresión lineal en X después de la simplificación. Por lo tanto, podemos remover v y u, además de ajustar la pareja (a_w, b_w) como está implícito después de simplificar la expresión E. El invariante(I) claramente se cumple.

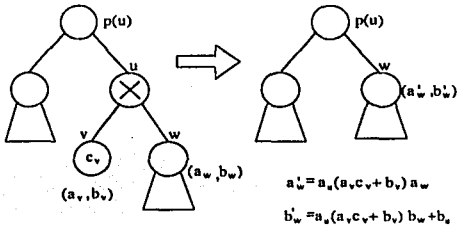


Figura 6.19: Ejemplo de la operación rastrillo

Nuestro algoritmo para evaluar expresiones aritméticas consiste de una asignación inicial de la etiqueta $(1, 0)$ a cada nodo del árbol, y una aplicación del algoritmo de la contracción del árbol tal que cada operación rastrillo es aumentada como se especifica en el parrafo anterior. Una vez que el algoritmo de contracción del árbol termina, obtenemos

un árbol T' de tres nodos con una raíz r tomando un operador $\circ$ y dos hojas u y v que contienen a las constantes c_u y c_v. Sea (a_u, b_u) y (a_v, b_v) las etiquetas de u y v, respectivamente, en el árbol T'. Entonces por el invariante(I) tenemos que

$$val(T) = val(r) = (a_u c_u + b_u) \circ (a_v c_v + b_v).$$

Ejemplo. Un árbol de expresiones con etiquetas iniciales $(1, 0)$ se muestra en la Figura 7.20(a). Durante la primera iteración del ciclo **for** del algoritmo de contracción del árbol, el nodo marcado por $*$ es rasurado, de este modo generamos el árbol de la Figura 7.21(b). Observe las etiquetas de los nodos tomados – 5 cambia de $(1, 0)$ a $(1, 2)$. A continuación los dos nodos tachados son rasurados, y el árbol es rasurado a la Figura 7.21(c). Durante la segunda iteración, un nodo es rasurado, obteniendo el árbol mostrado en la Figura 7.21(d). Finalmente durante la tercer iteración el nodo sencillo es rasurado, y terminamos con el árbol de tres nodos mostrado en la Figura 7.21(e). Consideremos la nueva etiqueta $(4, 14)$ contenida en el nodo 4. Esta etiqueta es derivada de la expresión $2[(2X + 10) + (-5 + 2)] + 0 = 4X + 14$. El valor de la expresión raíz está dada por $(4 \times 4 + 14) + (20 \times 1 + 0) = 50$.

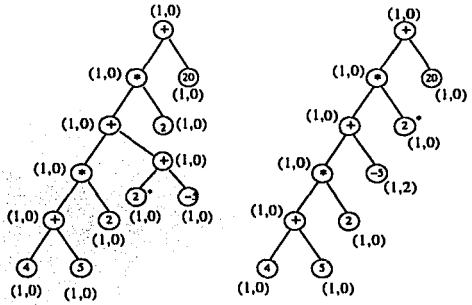


Figura 6.20: Evaluación de expresiones por contracción de árbol

6.3.3. Cálculo de funciones de árboles

Sea $(U, *)$ un semigrupo conmutativo con un elemento identidad e. De aquí, U es un conjunto de elementos con una operación binaria "$*$", tal que $*$ es asociativo, conmutativo, y $x * e = e * x = x$ para toda $x \in U$. Supongamos que dado un árbol arbitrario $T = (V, E)$

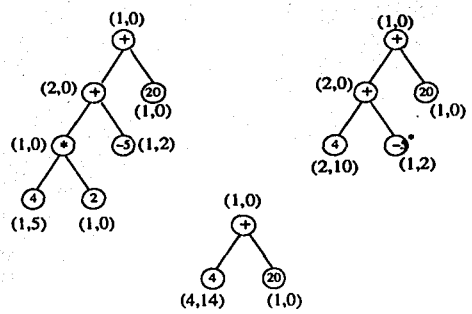


Figura 6.21: Evaluación de expresiones por contracción de árbol (continuación)

tal que cada vértice v es etiquetado con un elemento $l(v) \in U$. Consideremos el problema de calcular, para cada vértice v, el elemento $L(v)$ que resulta de aplicar la operación "$*$" a todos los elementos del subárbol enraizado en v.

Ejemplo. Sea U el conjunto de enteros no negativos, y sea "$*$" el operador mínimo. Agregamos el elemento identidad $+\infty$ a U. Entonces, $L(v)$ es el mínimo elemento en el subárbol enraizado en v.

Esbozaremos un algoritmo óptimo en tiempo $O(\log n)$ para resolver este problema. Empezaremos por construir un árbol binario T de la siguiente manera. Cada vértice v con más de dos hijos es remplazado por un árbol binario balanceado cuyas hojas serán los hijos de v. El siguiente ejemplo nos muestra como se harán los remplazos.

Ejemplo. En la Figura 7.22 cada unos de los vértices $1, 2, 4$ tienen más de dos hijos. El vértice 2 es remplazado con un árbol binario completo con dos vértices adicionales $2'$ y $2''$, y los vértices 1 y 4 sólo requieren la inserción de un sólo vértice adicional. Observe que, si ignoramos los vértices recientemente insertados, el conjunto de descendientes para cada vértice resultará en lo mismo. Aunque la relación de hermanos se preserve, la de los descendientes puede cambiar.

Una vez que T ha sido convertido en árbol binario, asociamos con cada vértice interno nuevo la etiqueta e, el elemento identidad. Es claro que, para cada vértice $v \in V$, $L(v)$ no ha cambiado después de esas modificaciones.

El problema se reduce ahora a evaluar todas las subexpresiones asociadas con todos los vértices, donde la única expresión involucrada es $*$. Este problema es el mismo que

el del problema de evaluación aritmética, excepto que tenemos sólo una operación. Entonces podemos utilizar el algoritmo de la contracción del árbol con la operación rastrillo modificada apropiadamente para calcular $L(v)$ para cada $v \in V$

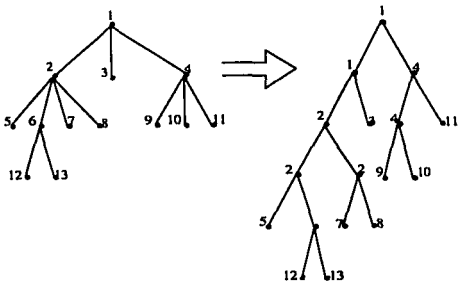


Figura 6.22: Proceso para transformar un árbol arbitrario

Capítulo 7

Búsqueda, Mezcla y Ordenamiento

La tarea de procesar una tabla que consiste de marcas cuyas claves provienen de un conjunto de aristas linealmente ordenado, tiene miltiples aplicaciones. Ejemplos específicos de estas tareas incluyen la búsqueda de una clave en la tabla y el ordenamiento de las claves en la tabla. Tales tareas requieren de comparaciones y movimiento de datos repetidamente. Asumimos que cada llave es una unidad atómica y que no puede manipularse como un entero o como una cadena de bits.

En este capítulo presentaremos algunos algoritmos óptimos para problemas básicos en búsqueda, mezcla y ordenamiento. Muchas técnicas son exploradas, incluyendo la **busqueda en paralelo, particion, encadenamiento, y divide y venceras**. Estas técnicas son usadas para desarrollar algoritmos paralelos eficientes para la búsqueda, mezcla, selección y ordenamiento.

7.1. Búsqueda

Sea $X = (x_1, x_2, \ldots, x_n)$ con n elementos distintos tomados de un conjunto linealmente ordenado $(S, \leq)$ tal que $x_1 < x_2 < \ldots < x_n$. Dado un elemento $y \in S$, nos interesa resolver el siguiente **problema de búsqueda**: identificar el índice i para cada $x_i \leq y < x_{i+1}$, donde $x_0 = -\infty$ y $x_{n+1} = +\infty$, y $-\infty$ y $+\infty$ son dos elementos agregados a S y satisfacen que $-\infty < x < +\infty$, para toda $x \in S$.

El método de búsqueda binaria resuelve este problema en $O(\log n)$ en tiempo secuencial óptimo. Este método consiste en comparar a y con el elemento $(n/2)$ de X. Basados en los resultados de esta comparación, cualquier búsqueda es termina cuando se encuentra el elemento o puede ser restringida a la parte superior o inferior de X. El proceso se repite hasta que el elemento x_i es encontrado como $y = x_i$ o el tamaño del subarreglo bajo consideración es igual a 1. Por lo tanto, en cualquier situación, la solución puede ser determinada.

Una extensión natural del método de búsqueda binaria del procesamiento en paralelo es **la búsqueda en paralelo**, en la cual se compara concurrentemente a y con múltiples elementos de X, decimos p elementos de X, los cuales intercambian a X dentro de $p+1$ segmentos de longitudes casi iguales. El resultado de este paso de la comparación en paralelo es identificar un elemento x_i qu e es igual a y, o restringir la búsqueda a uno de los $p+1$ segmentos. Repetimos este proceso hasta que un elemento x_i es encontrado tal que $y = x_i$ o el número t de elementos en el subarreglo bajo consideración no son mas que p. En el primer caso la solución es encontrada; en el segundo, la solución puede ser determinada por t comparaciones hechas en paralelo.

El algoritmo de búsqueda en paralelo para el procesador P_j se muestra a continuación, donde $1 \leq j \leq p$. El procesador P_1 es responsable de algunas inicializaciones y de cuidar los casos de las fronteras. La variable c_j es usada para indicar la salida de la comparación ejecutada en un paso dado; los índices l y r son apuntadores, izquierdo y derecho, a los límites del subarreglo actual bajo consideración.

Como entrada tenemos un arreglo $X = (x_1, x_2, ..., x_n)$ tal que $x_1 < x_2 < ... < x_n$ hijos, un elemento, el número p de procesadores, donde $p \leq n$ y el procesador j, donde $1 \leq j \leq p$. Como salida un índice i tal que $x_i \leq y < x_{i+1}$.

```
Busqueda en paralelo para el procesador P_j

begin
        1. if (j=1) then do
           1.1 asigna l:=0; r:=n+1; x_0:= -inf
               x_{n+1}:= +inf
           1.2 asigna c_0:=0; c_{p+1}:=1

        2. while (r-l>p) do
           2.1 if (j=1)
                  then {set q_0:=l;q_{p+1}:=r}
           2.2 set q_j:=l+(j*(r-l/p+1))
           2.3 if (y=x_{q_j})
                  then return(q_j)
               else set c_j:=0 if y> x_{q_j}
                         and c_j:=1 if y< x_{q_j}
           2.4 if (c_{j} < c_{j+1})
                  then set l:=q_{j};r:=q_{j+1}
           2.5 if (j=1 & c_{0}<c_{1})
                  then set l:=q_{0};r=q_{1}

        3. if (j<= r - 1) then do
           3.1 case statement:
               y = x_{l+j}:{return (l+j);exit}
               y > x_{l+j}:set c_{j}=0
               y < x_{l+j}:set c_{j}=1
           3.2 if (c_{j-1}<c_{j})
                  then return (l+j-1)
end
```

Figura 7.1: ALGORITMO BUSQUEDA_EN_PARALELO_P_j

Ejemplo. Sea X un arreglo. $X = (2, 4, 6, \ldots, 30)$ que consiste de todos los enteros pares entre 2 y 30, y sea $y = 19$. Suponga que $p = 2$. Después de la ejecución del paso 1, P_1 colocará $l = 0, r = 16, c_0 = 0, c_3 = 1, x_0 = -\infty$, y $x_{16} = +\infty$. El ciclo **while** se ejecuta en tres iteraciones; el efecto de cada iteración se muestra en la siguiente tabla.

iteracion	1	2	3
q_0	0	5	7
q_1	5	6	8
q_2	10	7	9
q_3	16	10	10
c_0	0	0	0
c_1	0	0	0
c_2	1	0	0
c_3	1	1	1
l	5	7	9
r	10	10	10

Durante la ejecución del paso 3,1, P_1 asigna $c_1 = 1$. Por lo tanto, en el paso 3,2, P_1 verifica que $c_0 < c_1$ y regresa el índice 9.

7.2. Mezcla

Consideremos nuevamente el problema de mezclar dos secuencias ordenadas vistas en la sección 6,4. Basadas en la **estrategia de particion**, un algoritmo paralelo en tiempo $O(\log n)$ fue presentado. El correspondiente trabajo total es $O(n)$; por lo tanto, el algoritmo es óptimo.

En esta sección, refinaremos la estrategia partición previa para obtener un algoritmo óptimo en tiempo $O(\log \log n)$ que se ejecuta en *CREW PRAM*. Este problema es uno de los muchos conocidos para obtener algoritmos rápidos en *CREW PRAM*. Comparando el problema de mezclar con sólo el problema de calcular el máximo de n elementos, el cual requiere $\Omega(\log n)$ pasos en paralelo en el modelo *CREW PRAM*, sin tener en cuenta el número de procesadores disponibles. Por lo tanto, la existencia de un algoritmo óptimo de mezcla en tiempo $O(\log \log n)$ que se ejecuta sobre el modelo *CREW PRAM* tal vez sorprenda.

Empezaremos renombrando algunas definiciones presentadas en la sección 6,4. El **rango** de un elemento x en una secuencia dada X, denotada como $rank(x : X)$, es el número de elementos de X que son menores o iguales que x. Si X está ordenado, definimos el **predecesor** de un elemento arbitrario x como el elemento x_r de X tal que $r = rank(x : X)$.

Asignando Rango una secuencia $Y = (y_1, y_2, \ldots, y_m)$ en X equivale a calcular el arreglo entero $rank(Y : X) = (r_1, r_2, \ldots, r_m)$, donde $r_i = rank(y_i : X)$

7.2.1. Asignando Rango a una Secuencia Corta dentro de una Secuencia Ordenada.

Sea X una secuencia ordenada con n elementos distintos, y sea Y una secuencia arbitraria de tamaño m tal que $m = O(n^s)$, donde s es una constante que satisface $0 < s < 1$. El algoritmo de búsqueda en paralelo puede ser usado para asignar rango a cada elemento de Y en X separadamente. Asignamos el número p de procesadores de este algoritmo a $p = \lfloor n/m \rfloor = \Omega(n^{1-s})$. Entonces, cada elemento de Y se le puede asignar rango en X en tiempo $O(\log(n+1)/\log(p+1)) = O(1)$. El número total de operaciones usadas para asignar rango cada elemento es $O(n/m)$, ya que $p = \lfloor n/m \rfloor$ y corre en tiempo $O(1)$.

7.2.2. Un Algoritmo de Mezcla Rápido.

Considere el problema de determinar $rank(B : A)$ para dos secuencias ordenadas A y B de longitud n y m respectivamente. Asumamos que todos los elementos de A y de B son distintos y por lo tanto que ningún elemento de A aparece en B.

```
Hay que cambiar este programa
        // ENTRADA:
        // 1. Un arreglo X=(x_1,x_2,...,x_n)
        // tal que x_1<x_2<...<x_n hijos
        // 2. Un elemento y
        // 3. El numero p de procesadores, donde
        // p<=n
        // 4. El procesador j, donde i<=j<=p
        // SALIDA: Un indice i tal que x_i<=y<x_{i+1}

begin
     1. if (j=1) then do
     1. if (j=1) then do1.1 asigna l:=0; r:=n+1; x_0:= -inf
     1. if (j=1) then dox_{n+1}:= +inf
     1. if (j=1) then do1.2 asigna c_0:=0; c_{p+1}:=1

     2. while (r-l>p) do
        2.1 if (j=1)then {set q_0:=l;q_{p+1}:=r}
        2.2 set q_j:=l+(j*(r-l/p+1))
        2.3 if (y=x_{q_j}) then return(q_j)
         else set c_j:=0 if y> x_{q_j}
          and c_j:=1 if y< x_{q_j}
end
```

Figura 7.2: ALGORITMO ASIGNACION_DE_RANGO

Como en la Seccion 6,4, usamos la estrategia de partición para mezclar dos secuencias A y B. Asignamos rango un conjunto de $\sqrt{m}$ elementos de B que partieron a B en bloques de longitudes casi iguales en la secuencia ordenada A. El cálculo del rango de los elementos seleccionados nos inducirán una partición de A en bloques tal que cada bloque de A tiene que acomodarse entre dos de los elementos escogidos de B. Por lo tanto, el problema general se ha reducido a asignar rango a los elementos de cada bloque de B en el correspondiente bloque de A.

En la Figura 8.2 mostramos el Algoritmo ASIGNACION_DE_RANGO y la Figura 8.3 ilustra las particiones inducidas por el algoritmo.

Ejemplo. Sea $A = (-5, 0, 3, 4, 17, 18, 24, 28)$ y $B = (1, 2, 15, 21)$. Al término del paso 2, obtenemos $j(0) = 0, j(1) = 2$, y $j(2) = 6$, donde 2 y 6 son los rangos de $b_2 = 2$ y $b_4 = 21$, respectivamente. Durante la ejecución del paso 3, introducimos $B_0 = (1), A_0 = (-5, 0), B_1 = (15), A_1 = (3, 4, 17, 18)$. En este caso, $rank(1 : A_0) = 2, rank(15 : A_1) = 2$. En el paso 4 ajustamos los rangos de $b_1 = 1$ y $b_3 = 15$ como se observa en la Figura 8.3:

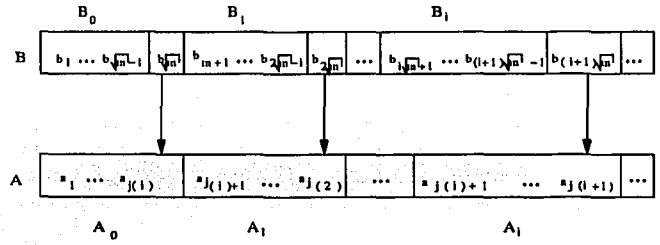


Figura 7.3: Particiones inducidas por el algoritmo

7.3. Ordenamiento

El problema de **ordenar** una secuencia X es el proceso de rearreglar los elementos de X tal que estos aparezcan en orden no decreciente o en orden creciente. Este problema ha sido extensamente estudiado debido a sus múltiples e importantes aplicaciones y su significado teórico intrínseco . Muchas estrategias de solución han sido estudiadas a cierta profundidad y han sido reportados sus desempeños.

En esta sección, sólo proporcionaremos dos posibles implementaciones en paralelo

de la **estrategia mezcla-ordenamiento** sobre el modelo**PRAM**, cada uno requiere $O(n \log n)$ operaciones y por lo tanto es óptimo. La primera implementación produce un algoritmo de ordenamiento en paralelo sencillo que corre en tiempo $O(\log n \log\log n)$. El segundo es significantivamente más complicado y depende intrínsecamente del esquema de encadenamiento; sin embargo, el tiempo de ejecución del algoritmo de ordenamiento es de $O(\log n)$.

7.3.1. Un algoritmo óptimo de ordenamiento sencillo.

Como su nombre lo indica, la **estrategia mezcla-ordenamiento** está basada en el procedimiento de mezcla que es usada para ordenar sucesivamente un número grande de subsecuencias que no se traslapan hasta que la subsecuencia esté completamente ordenada. Una manera posible de implementar esta estrategia, a la que haremos referencia como **mezcla-ordenamiento de dos vías**, es empezar a ordenar parejas de elementos de la secuencia X dada, y después ordenar cada par de parejas consecutivas, y así sucesivamente, hasta que X este ordenada.

El algoritmo mezcla-ordenamiento de dos vías puede ser visto como una aplicación de la estrategia **divide y vencerás** que consiste en:

1. dividir la secuencia de entrada X en dos subsecuencias X_1 y X_2, de tamaños aproximadamente iguales;
2. ordenar X_1 y X_2 separadamente; y finalmente
3. mezclar las dos secuencias ordenadas.

La secuencia de operaciones requeridas por el algoritmo de mezcla-ordenamiento de dos vías pueden ser representadas por un árbol binario de la siguiente manera. Sea T un árbol binario balanceado con n hojas. Los elementos de X están distribuidos uno por hoja. Los nodos de altura 1 representan las listas obtenidas de la mezcla de parejas de elementos consecutivos contenidos en los nodos hijos, hojas en este caso. De manera más general, cada nodo interno representa la subsecuencia que obtenemos después de mezclar las subsecuencias generadas por los nodos hijos. Por lo tanto, cada nodo interno representa la lista ordenada de elementos almacenados en su subárbol.

Ya que estamos interesados en la implementación en paralelo del algoritmo de mezcla-ordenamiento, nuestro problema puede ser reformulado de la siguiente manera. Para cada nodo v del árbol binario balanceado T calculamos la lista ordenada $L[v]$ que contiene todos los elementos almacenados en el subárbol enraizado en v. Claramente la raíz contendrá la lista ordenada.

En este proceso se puede ver cómo se calcula la lista ordenada para cada nodo v de un árbol binario balanceado. El algoritmo formal se muestra a continuación. El nodo (h, j) de un árbol binario es el j-esimo nodo a altura h ordenado de izquierda a derecha.

```
ordenamiento_mezcla
        // ENTRADA: Un arreglo X de orden n
        // suponemos que n=2^l, l entero no negativo
        // SALIDA: Un arbol binario balanceado con
        // n hojas tal que para 0<=h<=log n,
        // L(h,j) contiene la subsecuencia ordenada que
        // consiste de los elementos ordenados en el
        // subarbol enraizado en el nodo (h,j), para
        // 1<=j<=(n/2^h).

begin
        1. if 1<=j<=n pardo
                set L(0,j):=X(j)
        2. for h=1 to log(n) do
                for 1<=j<=n/(2^h) pardo
                    Mezcla (L(h'1,2j-1) & L(h-1,2j))
                    en las listas ordenadas L(h,j)
end
```

Figura 7.4: ALGORITMO MEZCLA_SIMPLE

Ejemplo. Considere la secuencia $X = (12, -5, -7, 51, 6, 28, 3, -8)$. La Figura 8.5 muestra el árbol binario con el contenido inicial de las hojas. Durante la iteración $h = 1$, obtenemos $L(1,1) = (-5, 12), L(1,2) = (-7, 51), L(1,3) = (6, 28), L(1,4) = (-8, 3)$. La siguiente iteración da como resultado las siguientes dos listas: $L(2,1) = (-7, -5, 12, 51)$ y $L(2,2) = (-8, 3, 6, 28)$. Finalmente, la lista generada que queda en la raíz es

$$L(3,1) = (-8, -7, -5, 3, 6, 12, 28, 51).$$

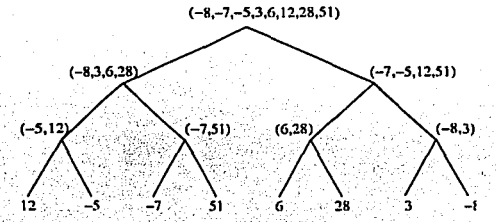


Figura 7.5: Ejemplos de árbol binario con el contenido inicial en las hojas

7.3.2. Un algoritmo de ordenamiento óptimo en tiempo O(log n)

Sea T un árbol binario tal que cada hoja u contiene una lista no ordenada $A(u)$ tomada de un conjunto linealmente ordenado. Consideremos el problema de determinar, para cada nodo v, la lista ordenada $L[v]$ que contiene todos los elementos almacenados en el subárbol enraizado en v. Observe que la lista inicial $A(u)$ de una hoja u podría estar vacía.

Ejemplo. Consideremos el árbol T mostrado en la Figura 8.6(a). La lista que generará en el nodo indicado está dado por $(-9, -7, -6, 2, 5)$.

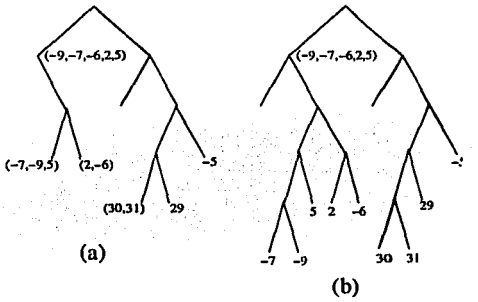


Figura 7.6: Ejemplos de árbol con una lista

Empezaremos haciendo un par de transformaciones. Remplazaremos cada hoja u con un árbol binario balanceado con $|A(u)|$ hojas tal que cada elemento de u es almacenada en una de las hojas. La altura de T se incrementará en $O(\log(max_u|A(u)|))$, pero cada hoja de T contendrá ahora al menos un elemento.

La segunda transformación es para forzar que cada nodo interno tenga dos hijos. Si este no es el caso, una hoja que no contiene ningún elemento podrá ser insertada.

Ejemplo. Aplicando las dos transformaciones al árbol T dado en la Figura 8.6(a) obtenemos un árbol T mostrado en la Figura 8.6(b).

Por lo tanto, asumamos por el resto de la sección que T es un árbol binario tal que tiene al menos un elemento almacenado en una hoja y cada nodo interno tiene exactamente

dos hijos.

Algoritmo Mezcla-Ordenamiento Encadenado. Introducimos en la Sección anterior la noción de una $c - cubierta$, y el arreglo entero $rank(A : B)$ que corresponde al rango de una lista ordenada A en una lista ordenada B. Antes de continuar necesitamos hacer las siguientes definiciones.

Dada una lista ordenada L, el **c-muestra** de L, denotado por $muestra_c(L)$, es la sublista ordenada de L que consiste de cada c–ésimo elemento de L; es decir, si $L = (l_1, l_2, \ldots)$, el $muestra_c(L) = (l_c, l_{2c}, \ldots)$.

Ejemplo. Sea $L = (4, 7, 8, 9, 11, 15, 38)$. El $muestra_3(L)$ está dado por $muestra_3(L) = (8, 15)$.

La estrategia en paralelo de mezcla-ordenamiento presentada recientemente está basada en un árbol transversal hacia delante tal que, para todos los vértices a la altura h dada, las listas $L[v]$ están completamente determinadas antes de procesar los nodos que están en la altura $h + 1$.

La **Estrategia de encadenamiento divide y vencerás** consiste en determinar $L[v]$ sobre un número de etapas tal que, en la etapa s, $L_s[v]$ es una aproximación de $L[v]$ que será mejorada en la siguiente etapa $s + 1$. Al mismo tiempo, la muestra de $L_s[v]$ es propagado hacia adelante y se usa para obtener aproximaciones de la lista que será generada en alturas mayores. El éxito de este método se debe a la combinación intrínseca de encadenamiento y la eficiente mezcla de las listas muestra.

Ahora describiremos el procedimiento para determinar $L_s[v]$ de manera precisa.

Sea $L_0[v] = \phi$ si v es un nodo interno; en otro caso, $L_0[v]$ consiste del objeto almacenado en la hoja v. La **altitud** de un nodo v está definida como $alt(v) = h(T)level(v)$, donde $h(T)$ es la altura de T y $level(v)$ es la longitud del camino desde la raíz hasta v. La lista almacenada en un nodo interno v se actualizará sobre las etapas s satisfaciendo $alt(v) \leq s \leq 3alt(v)$.

Decimos que v está **activo** durante la etapa s si $alt(v) \leq s \leq 3 \cdot alt(v)$. El algoritmo actualizará la lista $L_s[v]$ hasta que el nodo v esté **lleno**, es decir, $L_s[v] = L[v]$ cuando $s \geq 3 \cdot alt(v)$. Es claro que, si el invariante puede mantenerse después de $3 \cdot h(T)$ etapas, el nodo en la raíz estará lleno, y todos los nodos contendrán sus listas ordenadas.

Necesitamos notación adicional antes de describir el algoritmo dado. Definimos $muestra(L_s[x])$ para un nodo arbitrario x de la siguiente manera.

$$Ejemplo(L_s[x]) = \begin{cases} muestra_4(L_s[x]) & si \;\; s \leq 3 \cdot alt(x); \\ muestra_2(L_s[x]) & si \;\; 3 = 3 \cdot alt(x) + 1; \\ muestra_1(L_s[x]) & si \;\; 3 = 3 \cdot alt(x) + 2. \end{cases}$$

Por lo tanto, $muestra(L_s[x])$ es la sublista que consiste de cada cuarto elemento de $L_s[x]$ hasta que éste se llene; entonces en $muestra(L_s[v])$ están los otros elementos de la siguiente etapa, esto es, la etapa $3 \cdot alt(x) + 1$), y cada elemento de la etapa $3 \cdot alt(x) + 2$.

En la Figura 8.7 presentamos el Algoritmo MEZCLA_ENCADENAMIENTO, que mantiene el invariante estático. Es decir, para cada nodo v, $L_s[v]$ estará lleno cuando $s \geq 3 \cdot alt(v)$.

```
ordenamiento_mezcla_encadenamiento
        // ENTRADA: Para cada nodo v de un arbol binario
        // una lista ordenada L_s[v] tal que v esta lleno
        // siempre que s =>*alt(v)-1
        // SALIDA: Para cada nodo v, una lista ordenada
        // L_{s+1}[v] tal que v esta lleno siempre que
        // s =>*alt(v)-1

begin
      para todos los nodos activos v  pardo

        1. Sea u y w los hijos de v.
           Sea L'_{s+1}[u] = muestra(L_s[u]) y
           L'_{s+1}[w] = muestra(L_s[w]).

        2. Mezcla las dos listas L'_{s+1}[u] y L'_{s+1}[w]
           en la lista ordenada L_{s+1}[v].
end
```

Figura 7.7: ALGORITMO MEZCLA_ENCADENAMIENTO

7.4. Selección

Dado un conjunto de elementos $A = (a_1, a_2, \ldots, a_n)$ y un entero k, donde $1 \leq k \leq n$, el **problema de selección** consiste en determinar un elemento a_i de A, que satisfaga $rank(a_i : A) = k$.

Hemos examinado los casos especiales para $k = 1$ y $k = n$, que corresponden a calcular el mínimo y el máximo elemento de A respectivamente (Sección 6,6). Otro caso especial importante es determinar la **mediana**, correspondiente a $k = \lceil n/2 \rceil$. El problema de selección es conocido por admitir un algoritmo secuencial en tiempo lineal basado en la estrategia de **divide y vencerás.**

Renombrando el problema de calcular el máximo (o el mínimo) elemento, podemos

solucionarlo de manera óptima en tiempo $O(\log\log n)$ en el CRCW PRAM y en tiempo $O(\log n)$ en EREW PRAM. Sin embargo, encontrar la mediana se vuelve más complicado.

Presentaremos un algoritmo paralelo para el problema de selección en general que se ejecuta en tiempo $O(\log n \log\log n)$ usando un número lineal de operaciones. Este algoritmo puede verse como la versión en paralelo de un algoritmo secuencial óptimo conocido.

Nuestro problema de selección puede ser resuelto fácilmente si preprocesamos nuestro conjunto ordenando sus elementos, y entonces, nuestra salida será el $k-$ésimo elemento de nuestra lista ordenada. Si usamos el Algoritmo de encadenamiento MEZCLA-ORDENA, esta aproximación nos lleva a un algoritmo más rápido de tiempo $O(\log n)$, pero usa un total de $O(n \log n)$ operaciones, lo cual no es óptimo.

Para desarrollar un algoritmo paralelo óptimo, usamos la técnica de aceleramiento en cascada. Por lo tanto, es necesario un algoritmo óptimo que reduzca el tamaño de la entrada n a $O(n/\log n)$.

La idea que está detrás del algoritmo de reducción de tamaño es sencilla. Supóngase que un elemento a de A es identificado con la propiedad $(n/4) \leq rank(a : A) \leq (3n/4)$. Entonces a induce una partición de A en tres subconjuntos A_1, A_2, A_3, que consisten de elementos más pequeños que a, los elementos iguales que a, los elementos más grandes que a. Sea $s_i = |A_i|$, para $1 \leq i \leq 3$. Observe que $s_1, s_3 \leq (3n/4)$, ya que $(n/4) \leq rank(a : A) \leq (3n/4)$. Basados en los tamaños relativos de s_1, s_2, s_3 y k, podemos aislar el $k-$ésimo elemento mas pequeño en uno de los subconjuntos A_1, A_2 o A_3. Si el $k-$ésimo elemento mas pequeño se encuentra en A_2 hemos terminado; de otra forma el tamaño del arreglo correspondiente se ha reducido al menos en un factor de $(3/4)$.

Repitiendo este proceso $O(\log\log n)$ veces reduce el tamaño de A a $O(n/\log n)$. Podemos usar ahora el Algoritmo de encadenamiento MEZCLA_ORDENAMIENTO para identificar el elemento deseado.

El único detalle esencial que hemos dejado es la descripción del método usado para determinar el elemento a. Podemos determinarlo usando **la regla de la mediana de las medianas** que se explica en el Paso 2.2 y el Paso 2.3 del Algoritmo SELECCION, mostrado en la Figura 8.8.

El algoritmo recibe como entrada los siguientes datos:

- un arreglo $A = (a_1, a_2, \ldots, a_n)$
- un entero positivo k, donde $1 \leq k \leq n$ tal que $\log n$ es un entero que divide a n.

La salida es un elemento $a_i \in A$ tal que $rank(a_i : A) = k$.

```
Seleccion

begin
        1. Asigna n_0:=n; s:=0.

        2. while n_s > n/log n do
           2.1. Asigna s:=s+1.
           2.2. Particiona a A en bloques B_i's
                cada uno consiste de log n elementos
                consecutivos de A. Calcula la mediana
                m_i de cada bloque B_i.
           2.3. Calcula la mediana a de (m_1,...,m_{n/log n})
                usando el algoritmo de mezcla ordenamiento
                encadenado
           2.4. Determina los numeros s_1,s_2 y s_3 de los
                elementos mas pequenyos que, iguales y mas
                grandes que a, respectivamente.
           2.5. Casos:
                s_1 < k <= (s_1 + s_2): salida a y terminamos.
                k <= s_1: compactamos aquellos elementos de A
                mas pequenyos que a en lugares consecutivos,
                y asignamos n_s:=s_1
                k > (s_1 + s_2): compactamos aquellos elementos
                de A mas grandes que a en lugares consecutivos,
                y asignamos n_s:=s_3 y k =k-(s_1+s_2).

        3. Ordena el arreglo que consiste de los elementos
           restantes n_s. El k-esimo elemento es la salida
           deseada.
end
```

Figura 7.8: ALGORITMO SELECCION

Ejemplo. Sea $A = (5, -30, -10, 7, 25, 16, 31, -20, 8, 9, -3, 5, 13, 17, 21, 19)$ y $k = 4$. Los bloques B_i's están dados por $B_1 = (5, -30, -10, 7)$, $B_2 = (25, 16, 31, -20)$, $B_3 = (8, 9, -3, 5)$ y $B_4 = (13, 17, 21, 19)$. Por lo tanto las medianas son $m_1 = -10, m_2 = 16, m_3 = 5$ y $m_4 = 17$. La mediana de los m_i's es $a = 5$. Esto significa que $s_1 = 4, s_2 = 2$, y $s_3 = 10$. Por lo tanto, el elemento deseado se encuentra en $A_1 = (-30, -10, -20, -3)$, ya que $k \leq s_1$. El $k-$ésimo elemento más pequeño puede ser obtenido ordenando los elementos de A_1. De este ordenamiento obtenenemos -3 como el elemento deseado.

Teorema. El Algoritmo SELECCION calcula correctamente el $k-$ésimo elemento más pequeño del arreglo de entrada A. Este algoritmo se ejecuta en tiempo $O(\log n \log\log n)$, usando un número lineal de operaciones.

Demostración. El tiempo de ejecución se puede estimar de la siguiente manera. Considere una iteración arbitraria s del ciclo **while** (Paso 2). Podemos implementar el Paso 2.2 usando un algoritmo secuencial de tiempo lineal para la selección. Cada bloque toma tiempo secuencial de $O(\log n)$ para un total de $O(n_s)$ operaciones. Usando el Algoritmo de encadenamiento MEZCLA-ORDENA, el Paso 2.3 toma tiempo $O(\log n)$, usando un total

de

$$O\left(\frac{n_s}{\log n} \times \log n\right)$$

operaciones.

Para el Paso 2.4 marcamos cada elemento a_i con 1 si $a_i < a$, con 2 si o $a_i = a$ y con 3 si $a_i > a$. Usando el algoritmo de las sumas prefijas, podemos obtener s_1, s_2 y s_3 fácilmente. Por lo tanto, el Paso 2.4 toma tiempo $O(\log n)$ usando un total de $O(n_s)$ operaciones. De manera similar, el Paso 2.5 toma tiempo $O(\log n)$ usando un número lineal de operaciones.

Por lo tanto, cada iteración del ciclo **while** toma tiempo $O(\log n)$, con $O(n_s)$ operaciones. La siguiente afirmación implica que $O(\log \log n)$ iteraciones son suficientes para reducir el tamaño de la lista por debajo de $n/\log n$.

Afirmación: El tamaño de la lista en dos iteraciones consecutivas satisface

$$n_{s+1} \leq \frac{3n_s}{4}.$$

Demostracón de la afirmación. Es claro que basta con mostrar que $(n_s/4) \leq rank(a : A_s) \leq (3n_s/4)$, donde A_s es el arreglo al principio de la iteración s, y $A_0 = A$. Ya que a es la mediana de las m_i's, a es tan grande como una mitad de las m_i's. Pero m_i es tan grande como $(\log n)/2$ elementos en B_i, para cada i. Por lo tanto, a es mayor o igual que

$$\frac{n_s}{2 \log n} \times \frac{\log n}{2} = \frac{n_s}{4}$$

elementos de A_s. Podemos mostrar, de manera similar la otra parte de la inecuación.

Por lo tanto, el Paso 2 toma tiempo $O(\log n \log \log n)$, usando un total de $O(\sum n_s) = O(n)$ operaciones. El Paso 3 requiere tiempo $O(\log n)$ usando $O(n)$ operaciones.

Capítulo 8

Gráficas

Las gráficas pueden ser usadas para representar múltiples relaciones que ocurren en el mundo real, por ejemplo, la comunicación de sistemas, redes eléctricas, sistemas de transporte, entre otras; además pueden ser usadas como herramientas para estructurar tales relaciones, así como el desarrollo eficiente de estructuras de datos. El procesar gráficas de manera eficiente ha sido foco de investigación para los diseñadores de algoritmos. Introduciremos algoritmos paralelos eficientes para manejar algunos de problemas de gráficas representativos.

Primeramente estudiaremos dos algoritmos en paralelo para identificar las componentes conexas de una gráfica. Los recursos requeridos para solucionar este problema determinarán la complejidad de la mayoría de los algoritmos vistos en esta sección. Uno de nuestros algoritmos de componentes conexas será extendido para manejar el problema del árbol de expansión mínima. Por último presentaremos un método basado en la descomposición por orejas de una gráfica.

8.1. Componentes conexas

Sea $G = (V, E)$ una gráfica no dirigida con $|V| = n$ y $|E| = m$. Se dice que dos vértices u y v están **conectados** si $u = v$ o si existe un camino $P = (u = x_1, x_2, \ldots, x_k = v)$, tal que $(x_i, x_{i+1}) \in E$, donde $1 \leq i \leq (k-1)$. Esta relación es una relación de equivalencia sobre V, y por lo tanto las particiones de V en clases de equivalencia $\{V_j\}_{j=1}^{t}$. Las subgráficas $G_j = (V_j, E_j)$, donde $E_j = \{(x, y) \in E | x, y \in V_j\}$, son llamadas **componentes conexas** de G.

Ejemplo. Consideremos la gráfica mostrada en la Figura 9.1. Esta gráfica tiene tres componentes conexas, cada una de ellas se muestra por separado.

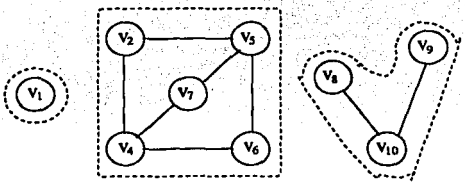


Figura 8.1: Las componentes conexas se muestran encerradas

Un problema básico de la teoría de gráficas es determinar las componentes conexas de una gráfica. Un algoritmo secuencial sencillo basado en componentes conexas de una gráfica se ejecuta de manera óptima en tiempo $O(n+m)$. La búsqueda a profundidad **DFS** de una gráfica $G = (V, E)$ es un método para visitar todos los vértices de G empezando por un vértice $v \in V$ y usando las aristas de G. Veamos un panorama del método de búsqueda.

Sea k la etiqueta para denotar una componente conexa. Inicialmente asignamos $k = 1$. El procedimiento de búsqueda que le aplicamos al vértice v comienza por visitarlo y etiquetarlo con el valor de k. Después tomamos un vértice no etiquetado w que sea adyacente a v y le aplicamos el mismo procedimiento. Una vez que el procedimiento en w termina, regresamos a v y quitamos las etiquetas a los vértices adyacentes a él. Si no existen tales vértices, la búsqueda termina en v. Entonces incrementamos la etiqueta de la componente conexa asignando $k = k+1$, y tomando arbitrariamente un vértice $v' \in V$ no etiquetado. Ahora aplicamos la búsqueda en v'. El algoritmo termina cuando todos los vértices han sido etiquetados. Los vértices pertenecen a la misma componente conexa si reciben la misma etiqueta.

Desarrollar un algoritmo paralelo eficiente se convierte en un reto. A continuación presentaremos un algoritmo que resuelvé el problema de las componentes conexas. Este algoritmo es más apropiado cuando la gráfica es representada por su matriz de adyacencia.

8.1.1. Estrategia general

Empezaremos por algunas definiciones. Un **pseudobosque** es una gráfica dirigida donde cada vértice tiene grado exterior menor o igual que 1.

Ejemplo. La Figura 9.2 (a) muestra un ejemplo de una función y su correspondiente pseudobosque. El pseudobosque, Figura 9.2(b) no corresponde a una función.

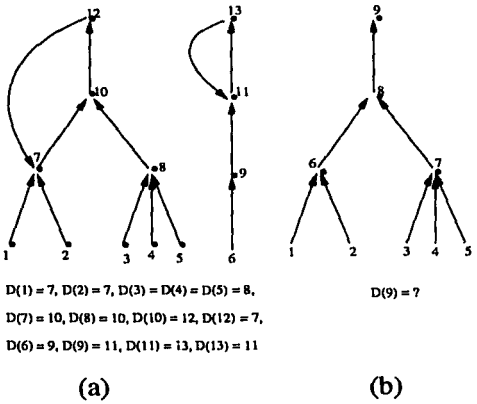


Figura 8.2: Pseudobosques

Ya hemos introducido el concepto de árboles enraizados dirigidos. En particular, si cada vértice se dirige hacia la raíz r, entonces el árbol dirigido correspondiente es llamado **estrella enraizada**. Un árbol dirigido enraizado y una estrella enraizada son mostradas en la Figura 9.3.

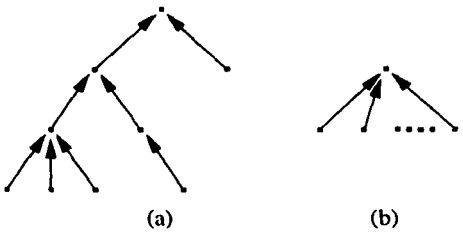


Figura 8.3: (a) Árbol dirigido enraizado.(b) Estrella enraizada.

Un pseudobosque determinado por una función consiste de un conjunto árboles enraizados, cada uno de ellos con un arco adicional que va de la raíz a uno de sus descendientes. Observe que el árbol definido por una función no tiene una única raíz, ya que

cualquier vértice del ciclo único determinado por la función puede ser usado como una raíz.

El algoritmo de componentes conexas genera un vector de salida D de longitud n tal que $D(u)$ es igual al representante de las componentes conexas que contienen a u. Por ejemplo, el vértice más pequeño de una componente conexa es un posible candidato para representante de tal componente. Una vez que la función D es generada, podemos responder en tiempo secuencial $O(1)$ preguntas del estilo ¿Están u y v en la misma componente conexa?

La estrategia general para identificar componentes conexas de una gráfica consiste de un procedimiento iterativo, donde, al principio de cada iteración, los vértices disponibles son particionados en grupos tal que todos los vértices de un grupo pertenecen a la misma componente conexa. Durante cada iteración algunos grupos con vértices adyacentes se mezclan para formar grupos grandes. El algoritmo termina cuando ningún grupo adicional puede ser mezclado. Cada grupo es representado típicamente por un árbol dirigido enraizado, siendo la raíz el representante de tal grupo. Los dos algoritmos difieren en los tipos de árboles dirigidos permitidos al final de cada iteración, y en la cual los grupos son escogidos para mezclarse.

Empezaremos con el algoritmo de componentes conexas ya que es muy conveniente para el manejo de gráficas densas.

8.1.2. Un algoritmo óptimo para gráficas densas

Sea A La matriz de adyacencia de $n \times n$ de una gráfica $G = (V, E)$ no dirigida, donde $V = \{1, 2, \ldots, n\}$. Por lo tanto, $A(i, j) = 1$ si y sólo si $(i, j) \in E$. Definimos la siguiente función C sobre V : $C(v) = min\{u | A(u, v) = 1\}$, y, si v es un vértice aislado, entonces $C(v) = v$. Es decir, $C(v)$ es el más pequeño vértice adyacente a v, siempre y cuando v no sea un vértice aislado; de lo contrario, $C(v)$ es igual al vértice v.

Ejemplo. Consideremos la gráfica mostrada en la Figura 9.5(a). Durante la primera iteración del ciclo, la función C definida en el Paso 2,2 está dada por

$$C(1) = 5, C(5) = 1, C(4) = 2, C(2) = 3, C(3) = 2, C(6) = 3, C(7) = 6, C(8) = 9, C(9) = 8.$$

Por lo tanto, los pseudobosques correspondientes tienen tres árboles dirigidos, como se muestra en la Figura 9.5(b). El Paso 2,3 ocasiona que cada uno de esos árboles enraizados se conviertan en estrellas enraizadas, como se muestra en la Figura 9.5(c). Asignamos a las raices 1, 2 y 8 los números seriados $s(1) = 1, s(2) = 2, s(8) = 3$. La matriz de adyacencia

```
Componentes conexas para graficas densas
        // ENTRADA: La matriz de adyacencia de A de n x n
        // de una grafica no dirigida.
        // SALIDA: Un arreglo D de tamanyo n tal que D(i)
        // es igual al mas pequenyo vertice en la componente
        // conexa de i

begin
        1. Asigna A_0:=A; n_0:=n;k:=0.

        2. while n_k > 0 do
           2.1. Asigna k=k+1.
           2.2. Asigna C(v):=min{u|A_{k-1}(u,v)=1,u!=v}}
           2.3. shrink cada arbol del pseudobosque definido
                por C en una estrella enraizada. La raiz de
                cada estrella que contiene mas de un vertice
                define un nuevo supervertice.
           2.4. Asigna a n_k igual al numero de nuevos
                supervertices, y asigna a A_k la matriz de
                adyacencia de n_k x n_k de la grafica de
                nuevos supervertices.

        3. Para cada vertice v, determina D(v) de la siguiente
           manera. Si al termino del paso 2, C(v)=v, asigna
           D(v)=v, en otro caso invierte el proceso ejecutado
           en el paso 2 expandiendo cada supervertice r dentro
           del conjunto V_r de su arbol dirigido, y haciendo la
           asignacion D(v)=D(r) para cada v que pertenece a V_r.

end
```

Figura 8.4: ALGORITMO COMPONENTES_CONEXAS_EN_GRAFICAS_DENSAS

A_1 está dada por

$$A_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Durante la segunda iteración, la función C da como resultado $C(1) = 2, C(2) = 1$ y $C(3) = 3$; por lo tanto, tenemos una estrella no trivial enraizada en 1. En este caso, $n_2 = 1$, y la estrella no trivial es transformada en un autociclo durante la tercer iteración; por lo tanto $n_3 = 0$. Por lo tanto el algoritmo termina con dos estrellas.

Concluimos que hay dos componentes conexas: una cuyo vértice representativo es 1, y la otro, donde su vértice representativo es 8. Por lo tanto, en el Paso 3, $D(1) = 1$ y $D(8) = 8$. Expandemos los supervértices en orden inverso. El vértice 2 pertenece a la estrella con raíz 1 obtenida en la segunda iteración. De aquí, $D(2) = D(1) = 1$. Los vértices restantes pertenecen a estrellas de la primera iteración. Entonces obtenemos $D(5) = D(1) = 1, D(3) = D(4) = D(6) = D(7) = D(2) = 1$ y $D(9) = D(8) = 8$.

Lema. Sea n_k el número no trivial de estrellas al final de la $k-$ésima iteración del

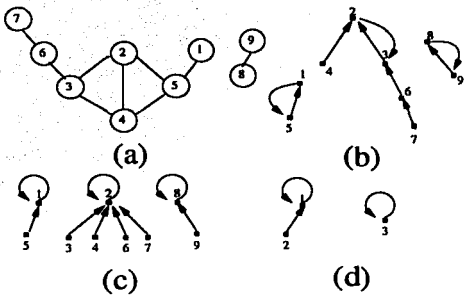


Figura 8.5: El resultado del algoritmo sobre una gráfica

Paso 2 del Algoritmo COMPONENTES_CONEXAS_EN_GRAFICAS_DENSAS. Entonces, $n_k \leq (n_{k-1})/2$, para toda $k \geq 1$.

Demostración. Al final de la $k-$ésima iteración, algunas de las n_{k-1} estrellas determinadas por la $(k-1)-$ésima iteración se transformarán en autociclos –decimos, las estrellas n'_{k-1} y las estrellas restantes $n_{k-1} - n'_{k-1}$ estarán contenidas en una de las estrellas n_k no triviales. De este modo tenemos que $n_k \leq (n_{k-1} - n'_{k-1})/2$, ya que cada estrella no trivial contiene al menos dos vértices. Por lo tanto, $n_k \leq (n_{k-1})/2$.

Ahora establecemos el siguiente teorema.

Teorema. Dada una matriz de adyacencia de $n \times n$ de una gráfica G no dirigida, el Algoritmo mostrado en la Figura 9.4 determina las componentes conexas de G en tiempo $O(\log^2 n)$ usando un total de $O(n^2)$ operaciones.

Demostración. Sea n el número de vértices de la gráfica G. Probamos por inducción sobre n que al final del algoritmo, $D(v)$ será igual al vértice más pequeño en la componente conexa que contiene a v.

El caso base $n = 1$ corresponde a un solo vértice v en G. La primera iteración hará a v en un supervértice, ya que $C(v) = v$. No hay estrellas triviales en este caso; por lo tanto, vamos directamente al Paso 3, donde obtenemos que $D(v) = v$.

Supongamos que $n > 1$. Sea v un vértice arbitrario de G. Si v es un vértice aislado, entonces es sencillo ver que v será un supervértice hasta que el algoritmo termine. Por lo tanto, no hay nada que probar en este caso.

Asumamos que v no es un vértice aislado de G. Durante la primera iteración, v pertenecerá a la estrella enraizada en r, donde r es el más pequeño vértice en la estrella. Si $v = r$, entonces v será uno de los supervértices en la nueva gráfica de supervértices –llamada g'. Por hipótesis de inducción, $D(v)$ se le asignará el más pequeño vértice en su componente conexa en G'. Observamos que cada uno de los supervértices es el más pequeño vértice de la estrella. El caso restante es cuando $v \neq r$, entonces, $D(v)$ será igual a $D(r)$ en el Paso 3, y la prueba es similar a la anterior.

MODELO PRAM. Los Pasos 1, 2.1 y 2.2 del Algoritmo de la Figura 9.4 no requieren acceso simultaneo a memoria. El Paso 2.3 requiere de capacidad concurrente de lectura y para el Paso 2.4 requiere escritura concurrente del mismo valor. El Paso 3 puede ser implementado sin ningún acceso simultáneo a memoria. Por lo tanto el algoritmo puede ser implementado con el modelo **CRCW PRAM**.

8.1.3. Un algoritmo eficiente para gráficas no densas

Cada iteración del Algoritmo revisado en la Figura 9.4 requiere de tiempo $O(\log n)$, esencialmente porque insistimos en formar estrellas enraizadas al final de cada iteración. Además, la definición de la función C no es necesariamente restrictiva. El siguiente algoritmo relajará esas dos restricciones de manera que la iteración pueda ejecutarse en tiempo $O(1)$.

Como en el algoritmo de componentes conexas anterior, manipulamos un bosque de árboles dirigidos tal que los vértices de cada árbol pertenecen a la misma componente conexa. En este caso, cada árbol dirigido es construido con un autociclo a su raíz. Durante cada iteración, examinamos cada arista (u, v) de la entrada de la gráfica, y, bajo ciertas condiciones, combinamos dos árboles que contengan a u y a v si son distintos. Al mismo tiempo cortamos las alturas de los árboles aplicando la Tecnica *pointer jumping* a cada vértice del árbol dirigido.

Sea D la función sobre V que define un pseudobosque que surge al principio de una iteración. Inicialmente, asignamos $D(v) = v$, para cada $v \in V$. Hay dos operaciones básicas que se ejecutan en un psuedobosque durante cada iteración y son las siguientes:

- **Injerto:** Sea T_i y T_j dos árboles distintos en un pseudobosque definido por D. Dada la raíz r_i de T_i y un vértice v de T_j, la operación $D(r_i) = v$ es llamada injerto de T_i en T_j. La Figura 9.6 ilustra esta operación.
- **Pointer jumping:** Dado un vértice v en un árbol T, la operación *pointer jumping* aplicada a v es asignar $D(v) = D(D(v))$. La Figura 9.7 ilustra tal operaciń.

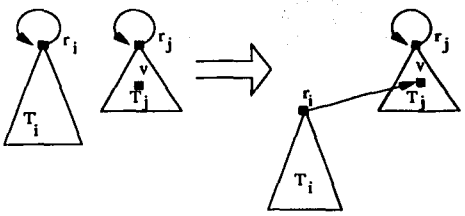


Figura 8.6: La operación injerto T_i sobre v de T_j

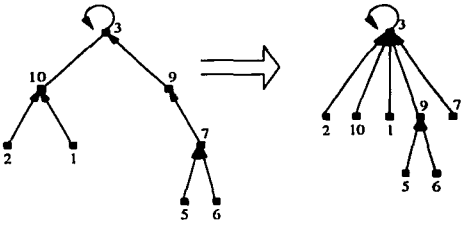


Figura 8.7: La operación Pointer jumping sobre vértices del árbol dirigido

Ahora aplicamos estas dos operaciones durante cada iteración de nuestro nuevo algoritmo de componentes conexas.

Sea T_i y T_j dos árboles dirigidos de un pseudobosuqe al principio de una iteración. Supongamos que existe una arista $(u, v) \in E$ tal que $u \in T_i$ y $v \in T_j$. Es claro que trataremos de mezclar los dos árboles, ya que sus vértices pertenecen a la misma componente conexa. Para complicarlo un poco más, queremos hacerlo en tiempo $O(1)$, requeriremos además que u o v sea raíz o que esté directamente conectado a la raíz de su árbol. Un vértice x satisface la última condición si y sólo si $D(x) = D(D(x))$; por lo tanto, tal condición puede ser verificada en tiempo $O(1)$. Sin embargo, un problema surge si u y v satisfacen esta condición, es decir, si $D(u) = D(D(u))$ y $D(v) = D(D(v))$, lo que significa que pudiera injertarse T_i en T_j y T_j en T_i de manera simultánea. Para evitar tal dificultad, insistiremos en injertar un árbol en el vértice más pequeño del otro árbol. Es decir, dada una arista $(u, v) \in E$, verificamos si $D(u) = D(D(u))$ y $D(v) < D(u)$, si esto ocurre, injertamos T_i en T_j, es decir, asignamos $D(D(u)) = D(v)$.

Injertando un árbol en el vértice más pequeño de otro árbol puede crear una dificultad de tipo diferente. Consideremos el caso cuando el árbol T_i es una estrella enraizada tal que

todos sus vértices son tan pequeños como cualquier otro vértice adyacente. La Figura 9.8. En este caso, T_i no podrá mezclarse con otro árbol hasta que:

- Uno de los vértices adyacentes este directamente conectado a la raíz de su árbol o se convierta en la raíz.
- Este árbol no será injertado en un árbol diferente a T_i. La Figura 9.8 lo ilustra más claramente.

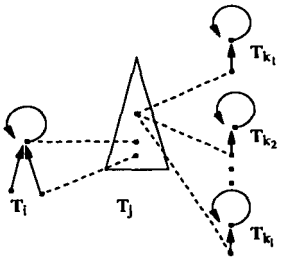


Figura 8.8: El problema de injertar un árbol en un vértice mas pequeño

Para evitar este problema, requerimos intentar injertar, después de que el proceso de injerto inicial haya sido completado, una estrella con raíz r_i en un vértice v será hecho, siempre que exista una arista conectada a la estrella T_i y a v.

La operación *pointer jumping* es aplicada a cada vértice v. Por lo tanto, la altura del árbol dirigido que contiene a v se reduce por lo menos en un factor de 2/3 siempre y cuando el árbol no sea una estrella enraizada.

Estamos listos para presentar el algoritmo. La entrada consiste de un conjunto de aristas (i, j) dadas en orden arbitrario. Una arista aparecerá dos veces, como (i, j) y como (j, i). Además, para cada vértice i introduciremos un vértice *dummy* o mudo i' conectado a i. Inicializamos la función D con $D(i) = D(i') = i$, para toda i. La razón de introducir estos nuevos vértices i' es evitar la posibilidad de tratar de injertar dos estrellas, una dentro de otra, durante la primera iteración del algoritmo. Por lo tanto, empezaremos con n estrellas.

El algoritmo basado en un procedimiento iterativo tiene las siguientes especificaciones. Como entrada del algoritmo un conjunto de aristas (i, j) dado en orden arbitrario, un pseudobosque definido por una función D tal que todos los vértices en cada árbol pertenecen

a la misma componente conexa. La salida es el pseudobosque obtenido después de injertar árboles en los vértices más pequeños de otros árboles, injertar, si es posible, estrellas enraizadas en otros árboles, y ejecutar la operación *pointer jumping* en cada vértice.

```
Componentes_conexas_para_graficas_poco_densas

begin
        1. Ejecuta la operacion injerta de arboles en los vertices
           mas pequenyos de otros arboles de la siguiente manera:

           for todo (i,j) que pertenecen a E pardo
               if (D(i)=D(D(i)) && D(j)<D(i))
                  set D(D(i)):=D(j).

        2. Injerta, si es posible, estrellas enraizadas en otros
           como sigue:

           for todos (i,j) que pertenecen a E pardo
               if (i pertenece a una estrella && D(j)!=D(i))
                  set D(D(i)):=D(j)

        3. Si todos los vertices estan en estrellas enraizadas,
           terminas. En otro caso, ejecuta la operacion pointer
           jumping sobre cada vertice asi:

           for todo i pardo
               set D(i):=D(D(i))

end
```

Figura 8.9: ALGORITMO COMP_CONEXAS_EN_GRAFICAS_POCO_DENSAS

Ejemplo. Sea G la gráfica dada en la Figura 9.10(a). Inicialmente, $D(i) = i$ y cada vértice está en una estrella enraizada, como se muestra en la Figura 9.10(b). En el Paso 1 de la primera iteración, muchas operaciones de injerto del mismo árbol serán intentadas. Por ejemplo, el árbol enraizado en 6 será injertado en el vértice 1 o en el vértice 5. Asumimos en este caso, que el vértice 6 es injertado en el vértice 1. La Figura 9.11(c) nos muestra el resultado de varias operaciones ejecutadas en el Paso 1 de la primera iteración. El Paso 2 injerta la estrella enraizada en el vértice 4 en el árbol enraizado en el vértice 2, como se muestra en la Figura 9.12(e). Después de que se ejecutó el primer paso de la segunda iteración, obtenemos el pseudobosque de la Figura 9.12(f). El Paso 2 no tiene efecto durante la segunda iteración. Al final de la segunda iteración, los vértices de cada componente conexa pertenecen al mismo árbol dirigido. Las siguientes dos iteraciones reducen los dos árboles en estrellas enraizadas, y ahí termina el algoritmo.

Observación. Si el Paso 2 del Algoritmo COMP_CONEXAS_EN_GRAFICAS_POCO_DENSAS es omitido, entonces podría tomar $\Omega(n)$ iteraciones para que el algoritmo termine.

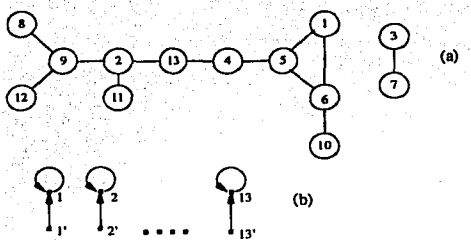


Figura 8.10: Ilustración del algoritmo COMPONENTES_CONEXAS

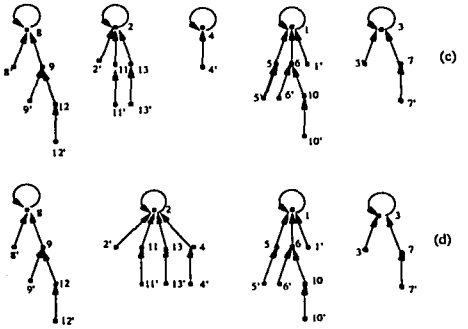


Figura 8.11: Ilustración del algoritmo COMPONENTES_CONEXAS continuación

Teorema. Cuando el Algoritmo COMP_CONEXAS_EN_GRAFICAS_NO_DENSAS revisado en la Figura 9.9 termina, todos los vértices en la misma componente conexa de la raíz r están directamente conectados a r. Todos los árboles se convierten en estrellas enraizadas después de $O(\log n)$ iteraciones, cada iteración requiere tiempo $O(1)$. El número total de operaciones requeridas por el algoritmo es $O((m+n)\log n)$.

Demostración. Probaremos las siguientes dos afirmaciones para establecer que sea correcto el Algoritmo COMP_CONEXAS_EN_GRAFICAS_NO_DENSAS.

Afirmación 1. Al final de cada iteración, el pseudobosque definido por D consiste de árboles dirigidos con autociclos a sus raices tal que todos los vértices en un árbol pertenecen a la misma componente conexa.

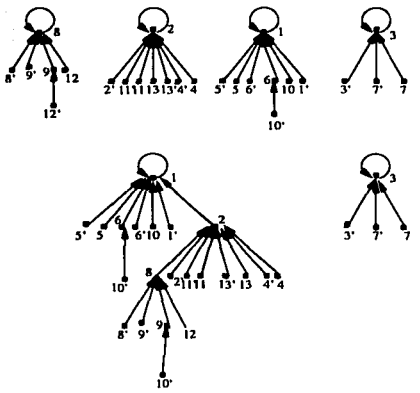


Figura 8.12: Ilustración del algoritmo COMPONENTES_CONEXAS continuación

Afirmación 2. En el Algoritmo COMP_CONEXAS_EN_GRAFICAS_NO_DENSAS, sea r la raíz de una estrella después del Paso 2 de alguna iteración. Todos los vértices v en la componente conexa de r satisfacen que $D(v) = r$, es decir, pertenecen a la estrella enraizada en r.

Demostración de la Afirmación 1. La prueba se hace por inducción sobre la iteración número k. El caso base $k = 0$ es inmediato ya que cada vértice está en una estrella enraizada.

Supongamos que funciona para la $k-$ésima iteración. Durante la $(k+1)-$ésima iteración, la operación injerto ejecutada en el Paso 1 o el Paso 2 combinará dos árboles cuyos vértices pertenecen a la misma componente conexa. El efecto del Paso 3 es cortar las alturas de los árboles dirigidos que no son estrellas. En cada caso, la raíz de cada árbol resultante mantendrá su autociclo. Por lo tanto, la hipótesis de inducción funciona para todos los valores de k.

Demostración de la Afirmación 2. Suponga que existe un vértice v en la componente conexa de r tal que $D(v) \neq r$ (el valor de D en la iteración actual). La existencia de ese vértice implica la existencia de un vértice w tal que $D(w) \neq r$, y w está conectado con una arista al árbol que contiene a r. Pero entonces r tendría que ser injertado en otro árbol en el Paso 2. Por lo tanto todos los vértices u en la componente conexa de r satisfacen que $D(u) = r$.

Para determinar que nuestro algoritmo termina después de $O(\log n)$ iteraciones, mostraremos a continuación la siguiente afirmación.

Afirmación 3. Sea K una componente conexa de G y sea $k = |K|$ el número de vértices en K. Sea $h_k(K)$ la suma de las alturas de los árboles que consisten de los vértices de K al final de la k-ésima iteración. Si esos árboles no forman una sola estrella enraizada, entonces $h_k(K) \leq (2/3)^k|K|$.

Demostración de la Afirmación 3. La prueba es por inducción sobre k. El caso base $k = 0$ es inmediato ya que siempre tenemos que $h_k(K) \leq |K|$.

Denotamos por $h(T)$ a la altura del árbol T. Ya que ningun árbol es injertado en una hoja, injertar el árbol T_i en otro T_j produce un árbol T cuya altura satisface que $h(T) \leq h(T_i)+h(T_j)$. Por lo tanto, después de los Pasos 1 y 2, $h_k(K)$ no se incrementa. Más aún, si esos árboles no son transformados sólo en estrellas, ninguna de ellas será estrellas enraizadas. Por la operación pointer jumping del Paso 3, $h_k(K) \leq (2/3)h_{k-1}(K)$. La prueba se hace por inducción.

Usando la Afirmación 3, es claro que $h(T) \leq 1$ para cada árbol T después de $O(\log n)$ iteraciones. Cada iteración requiere $O(m+n)$ operaciones, y por lo tanto el número total de operaciones es $O((m+n)\log n)$.

MODELO PRAM. El Paso 3 del Algoritmo ilustrado en la Figura 9.9 requiere sólo de capacidad concurrente de lectura. En los Pasos 1 y 2 , el algoritmo tratará de injertar un árbol dado en múltiples árboles. En este caso, se necesitará escritura concurrente. Por lo tanto, nuestro algoritmo puede correr en el modelo **CRCW PRAM.**

8.2. Árboles de expansión de peso mínimo

Sea $G = (V, E)$ una gráfica conexa no dirigida con una función de pesos w sobre el conjunto E de aristas al conjunto de los reales. Un árbol de expansión es una subgráfica $T = (V, E_T)$, $E_T \subset E$, de G tal que T es un árbol. El peso $w(T)$ de un árbol de expansión T es la suma de los pesos de sus aristas. Un árbol de expansión con los pesos más pequeños posibles es llamado un **Árbol de expansión de peso mínimo** (AEM) de G. Determinar un AEM de una gráfica con pesos es un problema importante que genera múltiples aplicaciones.

Asumamos sin pérdida de generalidad que no hay aristas con el mismo peso. En otro caso, podemos asignar para cada arista e, un nuevo peso $w(e)$ que consiste de la pareja $w(e), s(e)$, donde $w(e)$ está dado por el peso de la arista y $s(e)$ es un número seriado. Dadas dos aristas distintas e_1 y e_2, tenemos que $w(e_1) \neq w(e_2)$ por que $s(e_1) \neq s(e_2)$;

además, $w(e_1) < w(e_2)$ si $w(e_1) < w(e_2)$, o $w(e_1) = w(e_2)$ y $s(e_1) < s(e_2)$. La suma de los pesos de múltiples aristas es calculada por componentes.

8.2.1. Una estrategia para calcular árboles de expansión de peso mínimo

La siguiente observación nos da una estrategia eficiente para resolver problemas con AEM.

Lema. Sea $G = (V, E)$ una gráfica conexa con pesos. Para cada vértice $u \in V$, sea $C(u) \in V$ tal que $(u, C(u))$ la arista incidente en u con el mínimo peso. Entonces son ciertas las siguientes afirmaciones:

1. Todas las aristas $(u, C(u))$ pertenecen a AEM.
2. La función C define un pseudobosque tal que cada árbol dirigido tiene un ciclo que contiene exactamente dos arcos.

Demostración. Empezaremos demostrando la *A*firmación 1. Sea T un árbol de expansión mínima. Supongamos que $(u, C(u))$ no está en T para algún $u \in V$. Sea $P = (u, x_1, \ldots, x_s, C(u))$ el camino en T que conecta a u con $C(u)$. Remplazando (u, x_1) por $(u, C(u))$ produce otro árbol de expansión T' tal que $w(T') < w(T)$. Este resultado provoca una contradicción ya que T es un AEM; por lo tanto, $(u, C(u))$ debe estar en T.

Para la *A*firmación 2, si el árbol dirigido T' definido por C contiene un ciclo de longitud mayor o igual que 2, la versión no dirigida de T' contendrá un ciclo. Este resultado es imposible ya que la Afirmación 1 nos garantiza que todas las aristas deben pertenecer al AEM. Más aún, cada árbol dirigido debe tener un ciclo que contiene exactamente dos arcos.

La demostración del lema puede ser ligeramente generalizado al obtener el siguiente resultado.

Lema. Sea $V = \cup V_i$ una partición arbitraria de V con las correspondientes subgráficas $G_i = (V_i, E_i)$, donde $1 \leq i \leq t$. Para cada i, sea e_i la arista con menor peso que conecta a un vértice en V_i a un vértice en $V - V_i$. Entonces, todas las aristas e_i pertenecen al AEM de la gráfica $G = (V, E)$.

Observe que, ya que todos las aristas tienen diferentes pesos, el lema implica que el AEM es único.

Hay tres estrategias bien conocidas para resolver problemas de AEM, todas ellas derivadas de los dos lemas anteriores. La primera nos conduce al **Algoritmo de Prim**, que hace crecer al árbol sucesivamente por medio de la adición de aristas de peso mínimo, iniciando por un solo vértice. La segunda estrategia nos lleva al **Algoritmo de Kruskal**, que procesa las aristas en orden de acuerdo a sus pesos, teniendo precaución de no crear ciclos. La tercer estrategia nos dirige al **Algoritmo de Sollin**, que describiremos a continuación.

8.2.2. Algoritmo de Sollin y su implementación en paralelo

El algoritmo de AEM de Sollin empieza con el bosque $F_0 = (V, \emptyset)$ y los árboles crecen en subconjuntos de V hasta que haya un solo árbol que contenga todos los vértices. Durante cada iteración, la arista incidente de peso mínimo de cada árbol es seleccionada. Las nuevas aristas son agregadas al bosque actual –llamado F_s– para obtener un nuevo bosque, F_{s+1}. Este proceso es continuado hasta que haya un solo árbol. Es claro que el número de árboles en F_{s+1} es a lo más una mitad del número de árboles en F_s. Por lo tanto el Algoritmo de Sollin requiere $O(n)$ iteraciones.

Nuestra implementación en paralelo del Algoritmo de Sollin se ejecuta en tiempo $O(\log^2 n)$, usando un total de $O(n^2)$ operaciones.

Sea W la matriz de pesos dada por la gráfica conexa $G = (V, E)$. Para $i \neq j, W(i, j)$ es igual al peso de la arista (i, j), si el último existe; en otro caso, $W(i, j) = \infty$.

Cuando el algoritmo termina, cada arista perteneciente al AEM está marcada. Podemos entonces generar eficientemente la lista de aristas en el AEM.

La estrategia del algoritmo de componentes conexas para gráficas densas es usado para implementar el algoritmo de Sollin. La diferencia escencial es la definición de la función C. En el Algoritmo ARBOL_EXPANSION_MINIMA, C será usado para seleccionar las aristas en el AEM.

Ejemplo. Considere la gráfica con pesos G dada en la Figura 9.14(a). La función C determinada durante la primera ejecución del ciclo **while** está dada por $C(v_1) = v_6, C(v_2) = v_3, C(v_3) = v_4, C(v_4) = v_3, C(v_5) = v_3, C(v_6) = v_1$ (Figura 9.14(b)). Las aristas

$$(1,6), (2,3), (3,4), (3,5)$$

están marcadas. En el Paso 2,3 reducimos los dos árboles definidos por C en estrellas enraizadas, una con raíz v_4, y la otra con raíz v_1 como se muestra en la Figura 9.14(c).

```
Arboles_de_expansion_minima
        // ENTRADA: Un arreglo W de n x n, que representa
        // una componente conexa con pesos tal que ninguna arista
        // tiene el mismo peso.
        // SALIDA: Una etiqueta para cada arista que pertenece
        // al arbol de expansion minima.

begin
        1. Asigna W_0:=W, n_0:=n,k:=0

        2. while (n_k > 1) do
           2.1. asigna k:=k+1
           2.2. asigna C(v):=u cuando
                W_{k-1}(u,v)=min{W_{k-1}(v,x)|x!=v}.
                Marca (v,C(v)).
           2.3. Contraemos cada arbol dirigido del pseudobosque
                definido por C en una estrella enraizada.
           2.4. Asignamos n_k igual al numero de estrellas
                enraizadas y asignamos W_k la matriz de
                pesos de n_k x n_k de una grafica inducida por
                la vision de cada estrella como un solo vertice.

        3. Reestablecemos cada  arista marcada con su nombre
           original

end
```

Figura 8.13: ALGORITMO ARBOL_EXPANSION_MINIMA

Asignamos números seriados 1 y 2 a los vértices 4 y 1. La matriz W_1 está dada por

$$\begin{bmatrix} \infty & 4 \\ 4 & \infty \end{bmatrix}.$$

Por lo tanto, durante la segunda iteración obtenemos $C(1) = 2$ y $c(2) = 1$. Este resultado implica que $n_2 = 1$, y por lo tanto podemos ejecutar el Paso 3. La arista $(1, 3)$ será recuperada como la arista marcada durante la segunda iteración.

La validez del Algoritmo EXPANSION_MINIMA se sustenta en los dos lemas anteriores y en la demostración de la justificación del Algoritmo COMP_CONEXAS_EN_GRAFICAS_DENSAS.

Teorema. Dada una matriz de pesos de $n \times n$ de una gráfica conexa no dirigida $G = (V, E)$ un AEM puede ser encontrado en tiempo $O(\log^2 n)$, usando un total de $O(n^2)$ operaciones. Por lo tanto el Algoritmo EXPANSION_MINIMA es óptimo.

Model PRAM. El Paso 2.2 del Algoritmo EXPANSION_MINIMA no requiere ningún acceso simultaneo a memoria; los Pasos 2.3 y 2.4 requieren de capacidad concurrente de lectura. Por lo tanto nuestro Algoritmo es del tipo **CREW PRAM.**

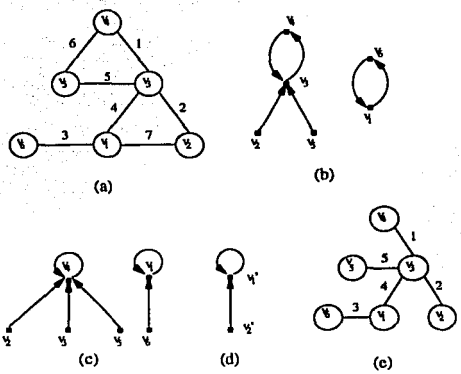


Figura 8.14: Ilustración del Algoritmo

8.3. Descomposición por orejas

Los métodos transversales para gráficas son usados principalmente porque inducen la descomposición de la gráfica en un conjunto estructurado de componentes simples. Las búsquedas **DFS** y **BFS** son dos métodos muy efectivos para encontrar manejar múltiples problemas teóricos de gráficas. Sin embargo introduciremos la técnica llamada **descomposición por orejas**, la cual tiene una eficiente implementación en paralelo.

Una descomposición por orejas es en esencia, una partición ordenada de un conjunto de aristas dentro de un camino simple, que incluye ciclos simples. De manera más formal, sea $G = (V, E)$ una gráfica no dirigida, con $|V| = n$ y $|E| = m$. Sea P_0 un ciclo simple arbitrario de G. Una descomposición por orejas de G empieza con P_0 una partición ordenada de un conjunto de aristas $E = P_0 \cup P_1 \cup \ldots \cup P_k$, tal que, para cada $1 \leq i \leq k$, P_i es un camino simple cuyos puntos finales pertenecen a $P_0 \cup P_1 \cup \ldots \cup P_{i-1}$, pero ninguno de sus vértices internos lo es. Cada camino simple P_i es llamado **oreja**. Si para cada $i > 0$, P_i no es un ciclo, la descomposición es llamada **descomposición abierta por orejas.**

Ejemplo. Considere la gráfica de la Figura 9.15. Una posible descomposición por orejas se muestra. Todas las aristas que pertenecen a la oreja P_i estan etiquetadas por el índice i. Observe que el ejemplo no es un descomposición abierta por orejas, ya que P_2 es un ciclo.

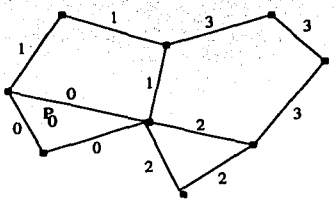


Figura 8.15: Descomposición por orejas de G

Una caracterización de las gráficas con descomposición por orejas esta dada por el siguiente teorema.

Teorema. Una gráfica no dirigida $G = (V, E)$ tiene una descomposición por orejas si y sólo si no tiene puentes, es decir, no existen aristas cuyo borrado desconecten la gráfica. La gráfica G tiene un descomposición abierta por orejas si y sólo si es biconexa.

Sea G con una descomposición abierta por orejas $E = P_0 \cup P_1 \cup \ldots \cup P_k$. Por lo tanto, P_0 es un ciclo simple, P_1 es un camino simple cuyos dos puntos finales distintos están en P_0, P_2 es un camino simple cuyos dos puntos finales distintos están en $P_0 \cup P_1$, y asi de manera sucesiva. Es claro que tal descomposición provoca un conjunto independiente de ciclos. Actualmente, borrar una arista arbitraria de cada oreja resulta en un árbol de expansión de G, como podemos mostrar usando inducción sobre el número de orejas. Por lo tanto el número de orejas es igual al número $(m - n) + 1$ de aristas que no pertenecen a algún árbol, y la descomposición por orejas provoca un ciclo básico de G. Estos hechos funcionan para cualquier descomposición por orejas.

Por otro lado, sabemos que un ciclo básico puede ser generado por un árbol de expansión de G por aristas que no pertenecen a algún árbol. Esta observación suguiere el siguiente método para obtener la descomposición por orejas.

Empezaremos por calcular el árbol de expansión T de G. Nos gustaría asociar una oreja P_e con cada arista que no pertenezca a algún árbol e. Sea C_e el ciclo básico inducido por e. En general, no podemos tomar $P_e = C_e$, ya que aristas en C_e pudieran estar compartidas por muchos otros ciclos básicos. Por lo tanto, hay que romper C_e en caminos, de forma que cada camino pertenezca a una sola oreja. Una manera de completar esta tarea es etiquetar cada arista que no pertenezca a algún árbol $e = (u, v)$ de la siguiente manera. Sea $level(e)$ el nivel del ancestro común más bajo de u y v; es decir, $lca(e) = lca(u, v)$. Entonces $label(e)$ es definida como el par $(level(e), s(e))$, donde $s(e)$ es el número seriado de e y $1 \leq s(e) \leq m$. Para cada arista del árbol g, sea $label(g)$ la etiqueta más pequeña de cualquier arista que no pertenezca a algún árbol cuyo ciclo contenga a g.

Ejemplo. Sea G la gráfica mostrada en la Figura 9.16. La línea negra representa las aristas del árbol de expansión enraizado, las líneas punteadas representan las aristas que no pertenecen a algún árbol. Empezando con las aristas que no pertenecen a algún árbol, obtenemos que $label(e_3) = (1,3), label(e_9) = (1,9), label(e_{10}) = (0,10), label(e_{11}) = (0,11)$. Usando esta información, las etiquetas de las aristas de árboles están dadas por $label(e_1) = (0,11), label(e_2) = (0,10), label(e_4) = (1,9), label(e_5) = (0,11), label(e_6) = (1,9), label(e_7) = (0,10), label(e_8) = (0,11), label(e_{12}) = (1,3), label(e_{13}) = (0,11)$.

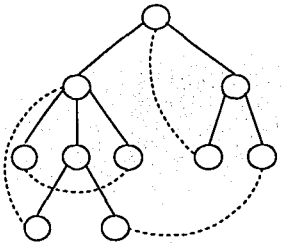


Figura 8.16: Ejemplo de un árbol de expansión de peso mínimo

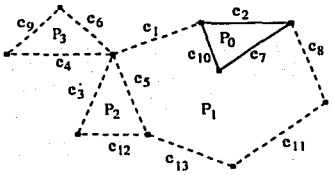


Figura 8.17: La descomposición determinada por el algoritmo

edge	label
e_1	(0,11)
e_2	(0,10)
e_3	(1,3)
e_4	(1,9)
e_5	(0,11)
e_6	(1,9)
e_7	(0,10)
e_8	(0,11)
e_9	(1,9)
e_{10}	(0,10)
e_{11}	(0,11)
e_{12}	(1,3)
e_{13}	(0,11)

Justifiquemos la introducción de etiquetas.

Lema. Sea T un árbol de expansión enraizado de una gráfica $G = (V, E)$, y, para cada arista $g \in E$, sea $label(g)$ la función definida previamente. Dada cualquier arista e que no pertenezca a algún árbol, sea $P_e = \{e\} \cup \{g \in T | label(g) = label(e)\}$. Entonces P_e es un camino simple (o ciclo). Más aún, cada arista de árbol pertenece exactamente a un camino.

Demostracion. Sea la arista que no pertenece a algun árbol dada por $e = (u, v)$. Considere el camino Q en T de v a $lca(u, v)$, y sea $v \neq lca(u, v)$. Sea $h = (x, p(x))$ la primera arista en Q tal que $label(h) \neq label(e)$. La Figura 9.18 lo muestra de manera clara. La diferencia de $label(e)$ y $label(h)$ implica la existencia de una arista que no pertenece a algun árbol (z, x) tal que la etiqueta de (z, x) es más pequenya que $label(e)$. Por lo tanto ninguna de las aristas en Q entre x y $lca(u, v)$ pertenecen a P. El argumento es similar si consideramos el camino de u a $lca(u, v)$. Por lo tanto, P_e es un camino simple.

Ya que las etiquetas de las aristas que no pertenecen a algun árbol son distintas, cada arista de árbol pertenece a un camino.

Ejemplo. Las etiquetas de todas las aristas en la Figura 9.16 fueron determinadas en el ejemplo anterior. Agrupando juntas todas las aristas con la misma etiqueta, obtenemos los siguientes conjuntos: $P_0 = \{e_{10}, e_2, e_7\}, P_1 = \{e_1, e_5, e_8, e_{11}, e_{13}\}, P_2 = \{e, e_{12}\}, P_3 = \{e_4, e_6, e_9\}$. Observe que hemos ordenado estos conjuntos por sus etiquetas. Es fácil verificar que esta secuencia indexada define un descomposición por orejas como se muestra

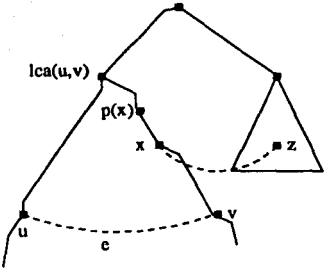


Figura 8.18: Ilustración de la demostración del lema

en la Figura 9.16(c). Este ejemplo no es un descomposición abierta por orejas, ya que P_3 es un ciclo.

En resumen, para cada arista e que no pertenezca a algún árbol, definimos un conjunto P_e de aristas, las cuales cambian de un camino simple tal que cada arista de árbol pertenece exactamente un camino. Por lo tanto la colección $\{P_e\}$ sobre las aristas e que no pertenecen a algún árbol forma una descomposición de E en caminos simples. El hecho de que puedan ser ordenados nos permite el descomposición por orejas.

Teorema. Sea $G = (V, E)$ una gráfica sin puentes. El Algoritmo DESCOMPOSICION_POR_OREJAS encuentra correctamente un descomposición por orejas de G. Este algoritmo puede ser implementado para ejecutarse en timepo $O(\log n)$, usando un total de $O((m+n)\log n)$ operaciones.

```
descomposici\'on por orejas
        // ENTRADA:Una grafica no dirigida G, sin puentes,
        // representada por una secuencia de aristas
        // SALIDA: Un conjunto ordenado de rutas que
        // representan una descomposicion por orejas de G.

begin
        1. Encontrar el arbol de expansion T de G.

        2. Enraizar T en un vertice r arbitrario,
           calcular level(v) y p(v) para cada vertice
           v!=r, donde level(v) y p(v) son el nivel y
           el padre de v, respectivamente.

        3. Para cada arista e que no pertenece a algun arbol
           e=(u,v), calcular lca(e)=lca(u,v) y
           level(e)=level(lca(e)).
           Asigna label(e):=(level(e),s(e)),
           donde s(e) es el numero consecutivo de e

        4. Para cada arista g del arbol calcular label(g)

        5. Para cada arista e que no pertenezca a algun arbol,
           asignar P_e={e}U{g en T|label(g)=label(e)}. Ordena
           las p_e's por su etiqueta label(e).

end
```

Figura 8.19: DESCOMPOSICION_POR_OREJAS

Capítulo 9

Geometría computacional

La geometría computacional estudia el diseño eficiente de algoritmos para manipular problemas computacionales relacionados con colecciones de objetos en el espacio euclidiano. Este tipo de problemas surgen en muchas aplicaciones, tal como la graficación por computadora, diseño asistido por computadora (CAD), robótica, reconocimiento de patrones. El objetivo aquí, es presentar algoritmos eficientes para problemas geométricos en el plano.

Empezaremos refiriéndonos al algoritmo divide y venceras que fue presentado en el Capítulo 6 para calcular la cubierta convexa de un conjunto de puntos. El algoritmo resultante se ejecuta en tiempo $O(\log n)$ usando un total de $O(n \log n)$ operaciones.

9.1. Revisión del problema de la cubierta convexa

Retomemos el problema de la cubierta convexa visto en el Capítulo 6 Sección 3 cuando introdujimos la estrategia **divide y vencerás**. Tal algoritmo presentado calculaba la cubierta convexa de un conjunto de n puntos en tiempo $O(log^2 n)$, usando un total de $O(n\ logn)$ operaciones. En esta sección refinaremos este algoritmo de forma que requiera la combinación de las dos soluciones de los subproblemas creados por la estrategia, y pueda ser ejecutado en tiempo $O(1)$ usando un número lineal de operaciones .

9.1.1. Definiciones.

Empecemos presentando las definiciones, algunas de ellas ya mencionadas en la primera sección.

Sea $p_1 = (x_1, y_1)$ y $p_2 = (x_2, y_2)$ dos puntos del plano Euclidiano. Los **segmentos** $s = p_1p_2$ están definidos por el conjunto de puntos q que satisfacen $q = \alpha p_1 + (1 - \alpha)p_2$, para toda α tal que $0 \leq \alpha \leq 1$. Es decir, s consiste de todos los puntos que descansan

sobre el segmento de linea recta formado por p_1 y p_2, incluyendo estos puntos. Llamaremos a p_1 p_2 **puntos finales** de s.

Dado p, un punto arbitrario en el plano, denotamos $x(p)$ y $y(p)$ a los valores de las coordenadas x y y del punto p, respectívamente. Sea L la linea especificada por la ecuación $y = ax + b$. Un punto $p = \alpha\beta$ está por **debajo** de L (o L está por **arriba** de p) si $\beta \leq a \times \alpha + b$. Podemos, de igual manera, definir p por **arriba** de L o L por **debajo** de p.

Una **cadena poligonal** es un conjunto ordenado de segmentos $P = (s_0 = p_0p_1, s_1 = p_1p_2, \ldots, s_{n-1} = p_{n-1}p_n)$. Una cadena poligonal es **simple** si cada segmento s_i intersecta sólo a s_{i-1} y s_{i+1} y sólo en los puntos finales p_i y p_{i+1}, respectivamente siempre que existan s_{i-1} o s_{i+1}. Para el resto de la sección asumiremos que todas las cadenas poligonales son simples. Si $p_0 = p_n$ en una cadena poligonal p, entonces p define dos regiones: la región contenida en el interior de la cadena, a la que se le llama **polígono** cuyo **límite** es la cadena poligonal p y la región no acotada que yace fuera de p.

Un polígono Q es **convexo** si dados cualesquiera dos puntos $p, q \in Q$ el segmento $s = pq$ está enteramente contenido en Q. Una **tangente** o una **línea de soporte** de un polígono convexo Q es una línea L que pasa a través de un vértice de Q de manera que Q yace por un lado de L.

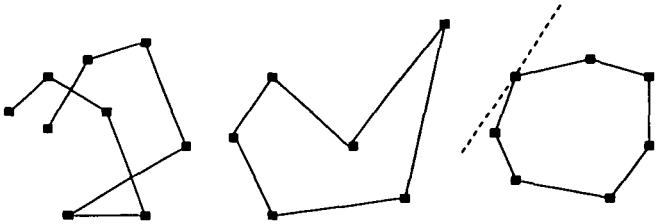


Figura 9.1: Cadenas poligonales y polígonos

Ejemplo. La Figura 10.1(a) muestra una cadena poligonal no simple. El polígono en la figura 10.1(b) es no convexo, mientras que el políono en la figura 10.1(c) es convexo y se muestra con una tangente L (línea punteada en la Figura).

Sea $S = (p_1, p_2, \ldots, p_n)$ un conjunto de n puntos en el plano. **La cubierta convexa** de S es el polígono convexo mínimo que contiene a todos los n puntos de S. El problema de la cubierta convexa es determinar una lista ordenada, en el sentido de las manecillas del reloj, por ejemplo $CH(S)$ de los puntos de S que definen el límite de la cubierta convexa de S. Cada punto de $CH(S)$ se llama un **punto extremo** de S o un **vértice** de $CH(S)$.

Por simplicidad asumamos que no hay dos puntos en S tal que tengan la misma coordenada x o y. Sea p y q los puntos en S con la menor y mayor coordenada x respectivamente. Claramente, los puntos p y q pertenecen a $CH(S)$; Ellos, además, particionan $CH(S)$ en **cubierta superior** $UH(S)$, que consiste de todos los puntos de p a q de $CH(S)$ en sentido de las manecillas del reloj, y **cubierta inferior** $LH(S)$, definida de manera similar de q a p.

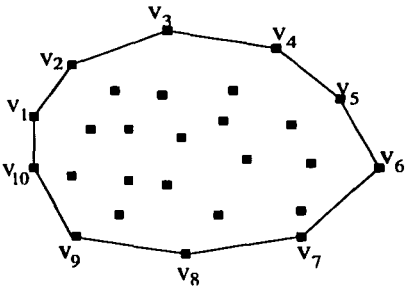


Figura 9.2: El problema de la cubierta convexa de un conjunto de puntos

Ejemplo. Considere el conjunto de puntos S mostrado en la Figura 10.2. En este caso la cubierta convexa está dada por $CH(S) = (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_1)$. La cubierta superior es $UH(S) = (v_1, v_2, v_3, v_4, v_5, v_6)$ y la cubierta inferior es $LH(S) = (v_6, v_7, v_8, v_9, v_{10}, v_1)$.

9.1.2. La estrategia divide y vencerás

Como mencionamos en las primeras secciones, el problema de la cubierta convexa puede ser solucionado con la estrategia **divide y vencerás**. Sea $S = (p_1, p_2, \ldots, p_n)$, donde n es tomado como potencia de 2. Empezamos ordenando los p_i's por su coordenada x. Asumamos por el resto de esta sección que la lista de puntos definida por S satisface que $x(p_1) < x(p_2) < \ldots < x(p_n)$, donde $x(p)$ es la coordenada x de p. El algoritmo desarrollado

en la sección 6.3 consiste en determinar recursivamente $CH(S_1)$ y $CH(S_2)$, donde $S_1 = (p_1, p_2, \ldots, p_{(n/2)})$ y $S_2 = (p_{(n/2)+1}, p_{(n/2)+2}, \ldots, p_n)$. Después, usando la tangente común superior e inferior de $CH(S_1)$ y $CH(S_2)$ se combinan para obtener $CH(S)$.

El análisis presentado en la sección 6,3 asume la existencia de un algoritmo de tiempo secuencial $O(\log n)$ para determinar la tangente común superior e inferior. Presentaremos a continuación un algoritmo paralelo en tiempo $O(1)$ que calcula dichas tangentes para $CH(S_1)$ y $CH(S_2)$, usando un número lineal de operaciones.

9.1.3. Un algoritmo de tiempo constante que calcula la tangente común superior

Sea $S = (p_1, p_2, \ldots, p_n)$ un conjunto de puntos dados tal que $x(p_1) < x(p_2) < \ldots < x(p_n)$, sea $S_1 = (p_1, p_2, \ldots, p_{(n/2)})$ y sea $S_2 = (p_{(n/2)+1}, p_{(n/2)+2}, \ldots, p_n)$. Recordemos que la **tangente común superior** entre $CH(S_1)$ y $CH(S_2)$ es la tangente común tal que $CH(S_1)$ y $CH(S_2)$ están por debajo de ella. Definimos a la **tangente común inferior** de manera similar. Ahora tratemos de desarrollar un algoritmo en tiempo constante que calcule la tangente.

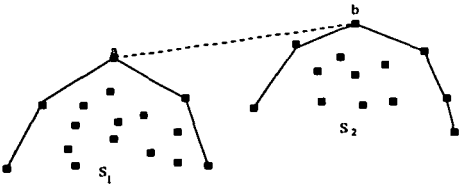


Figura 9.3: La tangente común superior

Sea $UH(S_1) = (r_1, \ldots, r_s)$ y $UH(S_2) = (q_1, \ldots, q_t)$ las cubiertas superiores de S_1 y S_2, respectivamente, dada en orden de izquierda a derecha, donde $r_1, q_j \in \{p_1, \ldots, p_n\}$ para $1 \leq i \leq s$ y $1 \leq j \leq t$. Para combinar ambas cubiertas superiores necesitamos determinar los puntos $u = r_i$ y $v = q_j$ que definen la tangente común superior T. Una vez que la hemos determinado, la cubierta superior $UH(S)$ es el arreglo que consiste de las primeras i entradas de $UH(S_1)$ y las últimas $t - j + 1$ entradas de $UH(S_2)$. Si s y t son parte de la entrada, $UH(S)$ y su tamaño pueden ser determinados en tiempo $O(1)$, usando un total de $O(n)$ operaciones de los índices i y j que definen a u y a v.

Examinemos un poco el problema de determinar la tangente entre el punto r_i de $UH(S_1)$ y $UH(S_2)$; es decir, nos gustaría encontrar un $q_{j(i)}$ tal que $UH(S_2)$ quede por

debajo de la línea determinada por r_i y $q_{j(i)}$. Veamos la siguiente observación.

Lema. Sea r_i un punto arbitrario de $UH(S_1)$. Dado cualquier punto q_l de $UH(S_2)$, podemos determinar en $O(1)$ tiempo secuencial cuando $q_{j(i)}$ está a la derecha de q_l, es igual a q_l, o está a la izquierda de q_l, donde $q_{j(i)}$ es el punto $UH(S_2)$ tal que $r_i q_{j(i)}$ es la tangente a $UH(S_2)$.

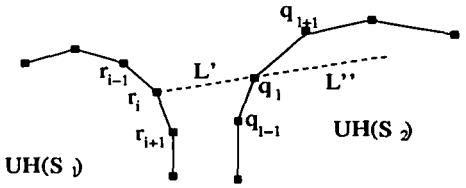


Figura 9.4: Determinando la tangente común de r_i a $UH(S_2)$

Demostración. Sea L la línea determinada por r_i y q_l. Sea L' (L'' respectivamente) la porción de L estrictamente a la izquierda (derecha, respectivamente) de q_l, como se puede ver Figura 10.3. Entonces, si L' está por arriba del segmento $q_{l-1}q_l$ y L'' está por abajo del segmento $q_l q_{l+1}$, entonces $q_{j(i)}$ está a la derecha de q_l. Si $q_l = q_{j(i)}$, entonces q_{l-1} y q_{l+1} están por abajo de L. En otro caso, $q_{j(i)}$ está a la izquierda de q_l.

Ya que r_i y q_l están dados, podemos determinar una de las tres condiciones en $O(1)$ tiempo secuencial. Por lo tanto tenemos el siguiente corolario.

Corolario. Dadas dos cubiertas superiores $UH(S_1), UH(S_2)$ y un punto r_i de $UH(S_1)$, la tangente $r_i q_{j(i)}$ a $UH(S_2)$ puede ser determinada en tiempo $O(\log t/\log k)$ usando k procesadores, donde t es el número de puntos definidos por $UH(S_2)$ y $1 < k \leq t$.

Demostración. El arreglo $UH(S_2) = (q_1, \ldots, q_t)$ está dado de manera ordenada (en sentido de las manecillas del reloj) de izquierda a derecha. Podemos usar el Algoritmo BUSQUEDA_EN_PARALELO para encontrar $q_{j(i)}$ de la siguiente manera. Escogemos k puntos de $UH(S_2)$ que separamos a $UH(S_2)$ en $k+1$ porciones, cada una con aproximadamente el mismo número de puntos. El lema anterior puede ser usado para determinar, en tiempo $O(1)$, la porción que contiene a $q_{j(i)}$ usando k procesadores. Repetimos el proceso hasta que $q_{j(i)}$ sea identificado.

Veamos otra observación clave dada en el siguiente lema.

Lema. Sea (u, v) la tangente común superior de dos cubiertas superiores $UH(S_1)$ y

$UH(S_2)$. Supongamos que, para un punto arbitrario r_i de $UH(S_1)$ damos un $q_{j(i)}$ tal que el segmento de línea $r_i q_{j(i)}$ define la tangente a $UH(S_2)$. Entonces, en $O(1)$ tiempo secuencial, podemos determinar cuando u está a la izquierda de r_i, cuando es igual a r_i o cuando está a la derecha de r_i.

Demostración. Observe que, si $r_i q_{j(i)}$ es tangente a $UH(S_1)$, entonces u debe ser igual a r_i ya que $r_i q_{j(i)}$ es tangente a $UH(S_2)$. De otra forma, u estaría a la izquierda de r_i si y sólo si r_{i-1} está por arriba de $r_i q_{j(i)}$. La Figura 10.5 muestra la construcción.

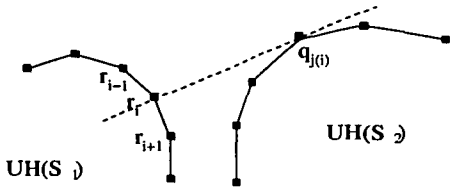


Figura 9.5: Determinación del punto de la tangente

Los dos lemas anteriores nos permiten determinar, para cualquier punto r_i de $UH(S_1)$, cuándo el punto u de la tangente común superior (u, v) aparece a la izquierda de r_i, a la derecha de r_i o igual a r_i en tiempo $O(\log t / \log k)$ usando k procesadores. Si tomamos a k cercano a $\sqrt{t}$, obtenemos un algoritmo en paralelo en tiempo $O(1)$ que determina, para cualquier punto r_i, cuando u es igual a r_i, está a la izquierda de r_i o cuando está a la derecha de r_i.

Estas observaciones juegan un papel importante en el algoritmo de busqueda en paralelo para aislar los puntos u de la siguiente manera. Escogemos $\sqrt{s}$ puntos de $UH(S_1)$, de manera que lo divida en porciones casi iguales. Podemos aislar u dentro de una porción única en tiempo $O(1)$ usando un total de $\sqrt{s}\sqrt{t}$ procesadores. De manera similar podemos aislar a v dentro de una porción de $UH(S_2)$ que contiene aproximadamente $\sqrt{t}$ puntos en tiempo $O(1)$ usando un $\sqrt{s}\sqrt{t}$ procesadores. Los límites de la complejidad pueden mantenerce en tiempo $O(1)$, con un total de $O(\sqrt{s}\sqrt{t}) = O(n)$ operaciones.

Entonces tenemos al menos $\sqrt{s}$ candidatos para el punto u y $\sqrt{t}$ candidatos para el punto v. Para cada par de candidatos, podemos checar, en $O(1)$ tiempo secuencial, cuando un par de puntos constituyen una tangente común superior entre $UH(S_1)$ y $UH(S_2)$. Por lo tanto, podemos probar todos los pares concurrentemente; por lo tanto, podemos identificar la única tangente común superior en tiempo $O(1)$ usando $O(n)$ operaciones.

```
Tangente Comun Superior
        // ENTRADA: Las cubiertas superiores
        // UH(S_1)=(r_1,r_2,...,r_s),
        // UH(S_2)=(q_1,q_2,...,q_t), ordenadas de
        // izquierda a derecha
        // SALIDA: Puntos u y v tal que la linea
        // determinada por u y v es la tangente comun
        // superior entre UH(S_1) y UH(S_2).

begin
        1. Para cada i tal que i<=i<=sqrt(s) encontrar
           q_{i*sqrt(j)} tal que r_{i*sqrt(s)} q_{i*sqrt(s)}
           es la tangente entre UH(S_1) y UH(S_2).

        2. Para cada i tal que i<=i<=sqrt(s), determinar cuando
           u esta a la izquierda, igual o a la derecha de
           r_{i*sqrt(s)}. Si u=r_{i*sqrt(s)} para alguna i,
           terminamos, en otro caso deducimos el bloque
           A=(r_{i*sqrt(s+i)},...,r_{(l+1)*sqrt(s-1)})
           que contiene a u.

        3. Para cada r_{i} en el bloque A, determinamos q_{j(i)}
           tal que r_{i}q_{j(i)} es la tangente a UH(S_2),
           asignamos u:=r_{i} y v:=q_{j(i)} si r_{i}q_{j} es la
           tangente a UH(S_1).

end
```

Figura 9.6: ALGORITMO TANGENTE_COMUN_SUPERIOR

Ahora estamos listos para listar el algoritmo de manera más formal.

Teorema. El Algoritmo TANGENTE_COMUN_SUPERIOR calcula correctamente la tangente común superior de las dos cubiertas superiores $UH(S_1)$ y $UH(S_2)$. Este algoritmo corre en tiempo $O(1)$, usando un número lineal de operaciones.

Demostración. La demostración usa los lemas anteriores y la explicación presentada antes del algoritmo.

Estimamos el tiempo de ejecución asumiendo que hay $(s+t)$ procesadores disponibles. El Paso 1 puede ser ejecutado en $O(1)$ usando $\sqrt{t}$ procesadores para cada i. Ya que $\sqrt{s}t \leq (s+t)$, todas las $q_{j(i\sqrt{s})}$'s pueden ser calculadas en tiempo $O(1)$ usando $(s+t)$ procesadores. El Paso 2 puede ser hecho en tiempo $O(1)$ con $O(\sqrt{s})$ procesadores. De manera similar analizamos el Paso 3. Por lo tanto, el algoritmo por completo puede ser ejecutado en tiempo $O(1)$, usando $(s+t)$ procesadores. Concluimos que el número total de operaciones es $O(s+t) = O(n)$.

9.1.4. Uniendo todas las piezas

Combinando la estrategia **divide y vencerás** y el algoritmo en paralelo que calcula rápidamente la tangente común superior o inferior, obtenemos el siguiente resultado.

Teorema. La cubierta convexa de un conjunto de n puntos en el plano pueden ser calculadas en tiempo $O(\log n)$, usando un total de $O(n \log n)$ operaciones. Por lo tanto, el Algoritmo TANGENTE_COMUN_SUPERIOR es óptimo.

Demostración. Sea $S = (p_1, p_2, \ldots, p_n)$ un conjunto de puntos dado. Preprocesamos S ordenando los puntos por sus coordenadas x. Este preprocesado toma tiempo $O(\log n)$, usamos $O(n \log n)$ operaciones si aplicamos el algoritmo encadenamiento mezcla-ordenamiento. De aquí, asumimos que $x(p_1) < x(p_2) < \ldots < x(p_n)$.

Usamos la estrategia **divide y vencerás** para determinar $CH(S)$ de la siguiente manera. Si $n \leq 4$, identificamos $CH(S)$ por medio de la fuerza bruta. En otro caso llamamos al algoritmo recursivamente para calcular $CH(S_1)$ y $CH(S_2)$. Mezclamos $CH(S_1)$ y $CH(S_2)$ para calcular sus tangentes común superior e inferior y deducir $CH(S)$.

Hay $O(\log n)$ iteraciones, cada una de ellas toma $O(1)$, usando un número lineal de operaciones. Por lo tanto, para terminar el Algoritmo DIVIDE_Y_VENCERAS requerirá tiempo $O(\log n)$ con un total de $O(n \log n)$ operaciones. Por lo tanto, el algoritmo por completo puede ser ejecutado con estos límites.

Ya que el problema de la cubierta convexa puede ser reducido a un ordenamiento, el algoritmo resultante es óptimo.

Modelo PRAM: La capacidad de lectura concurrente es requerido por nuestro algoritmo de ordenamiento, y por el Algoritmo de BUSQUEDA_EN_PARALELO. La capacidad de escritura concurrente no es necesaria. Por lo tanto nuestro algoritmo corre en el modelo **CREW PRAM**.

Capítulo 10

Cadenas

Importantes problemas computacionales pueden ser formulados como una búsqueda de las ocurrencias de un conjunto de objetos dado, llamado **patrones**. Las areas específicas donde lo aplicamos van desde la edición de texto, visión computacional, hasta la biología molecular. Aquí trataremos con cadenas y su ocurrencia exacta en otras cadenas. Herramientas computacionales poderosas basadas en propiedades matemáticas de las cadenas, son bien conocidas y pueden usarse para manejar eficientemente múltiples tareas de procesamiento en cadenas.

La mayoría de las técnicas algorítmicas para **comparar cadenas** tienen que ver con las propiedades periódicas de las cadenas. Tales propiedades serán primeramente presentadas, seguidas del desarrollo de algoritmos en paralelo que comparan cadenas de manera óptima en tiempo logarítmico. El principal paradigma consiste del análisis de patrones seguido del análisis de texto.

10.1. Hechos preliminares sobre cadenas

Sea Σ un **alfabeto** que consiste de un número finito de símbolos. Una **cadena** Y en Σ es una secuencia finita de elementos de Σ. La **longitud** de Y, denotada como $|Y|$, es el número total de elementos en la secuencia definida por Y. Si X y Y son cadenas, la **concatenación** de X y Y es la cadena XY, es decir la secuencia definida por X seguida de la secuencia definida por Y.

Para describir los algoritmos sobre cadenas, asumamos que una cadena Y está representada por un arreglo tal que $Y(i)$ es el i–ésimo elemento de Y, donde $1 \leq i \leq |Y|$.

Sea Y una cadena de longitud m. Para cualquier par de índices i y j tal que $i \leq i \leq j \leq m$, el subarreglo $Y(i:j)$ define una **subcadena** de Y. Por lo tanto una subcadena de Y es cualquier bloque contiguo de caracteres en Y. Una subcadena definida por $Y(1:i)$, para algún i tal que $1 \leq i \leq m$, es llamado **prefijo** de Y, mientras que una subcadena de la forma $Y(j:m)$, para alguna j tal que $1 \leq j \leq m$, es llamado **sufijo** de Y.

Una cadena X **aparece** en Y en la posición i si X es igual a la subcadena de Y de longitud $|X|$ empezando en la localidad i. De manera más formal, $X(j) = Y(i+j-1)$, para todos los índices j tal que $1 \leq j \leq |X|$. También podemos decir que X **se encuentra** en Y en la posición i.

Ejemplo. Sea $\Sigma = \{a, b, c\}$, y sea Y una cadena $Y = aabcabccaa$. Entonces, $X = abc$ es una subcadena de Y que aparece en las posiciones 2 y 5. El prefijo $Y(1 : 5)$ es la subcadena $aabca$, y el sufijo $Y(5 : 10)$ es la subcadena $abccaa$.

10.1.1. Periodicidad en cadenas

Las propiedades de periodicidad de las cadenas proporcionan las bases para ciertas herramientas algorítmicas para la manipulación eficiente de cadenas. Ahora introduciremos la noción del periodo de una cadena, y estudiaremos algunas propiedades al respecto.

Sea Y una cadena de longitud m. Una subcadena X de una cadena Y será llamada **periodo** de Y si $Y = X^k X'$, donde X^k es la cadena que consiste de X concatenada con ella misma k veces, siendo k un entero positivo, y X' es un prefijo de X. Por lo tanto, Y consiste de múltiples copias consecutivas de X, seguidas de un prefijo de X. Es fácil ver que X es un periodo de Y si y sólo si Y es un prefijo de XY.

La definición de un periodo tiene la siguiente implicación importante. Supongamos que una copia de Y es colocada arriba de Y pero trasladada i posiciones de la parte izquierda de Y, para algún i tal que $1 \leq i \leq m-1$. Esto se ve de manera más clara en la Figura 11.1. Entonces, si la porción traslapada es idéntica, Y tiene un periodo que consiste del prefijo $Y(1 : i)$. En otras palabras, si para alguna i, las dos subcadenas $Y(i+1 : m)$ y $Y(1 : m-1)$ son iguales, entonces $Y(1 : i)$ es un periodo de Y. Observe que Y es simpre periodo de si misma.

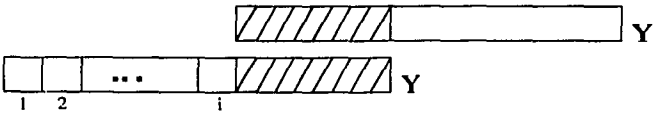


Figura 10.1: Copiando la cadena Y

Ejemplo. Sea $Y = ababababa$. Entonces, el prefijo $U = abab$ es un periodo de Y ya que $Y = U^2 a$. La cadena Y también tiene los siguientes periodos: $ab, ababab, abababab$.

El período de una cadena Y es el periodo más corto de Y. Sea p la longitud (también llamado tamaño) del periodo de Y. Nos referiremos a p como el periodo de Y. Por lo tanto, p es elentero más pequeño entre 1 y m tal que $Y(i) = Y(i+p)$, para toda i que satisfaga $1 \leq i \leq m-p$.

La cadena Y es llamada **periódica** si su periodo p satisface que $p \leq (m/2)$.

Ejemplo. Considere las siguientes dos cadenas: $Y = abcaabcab$ y $Z = abcabcab$. La cadena Y no es periodica, ya que su periodo más corto es $abcaabc$; la cadena Z es periodica, y tiene un periodo $p = 3$.

Un hecho importante sobre los periodos de una cadena están plasmados en el siguiente lema. Antes de dar el lema revisemos una versión sencilla del algoritmo de Euclides para calcular el máximo común divisor δ de dos enteros positivos p y q.

Revisión. Para calcular el máximo común divisor δ de p y q, asignamos $\delta = p = q$ si $p = q$. En otro caso, asumimos sin perdida de generalidad que $p > q$. Entonces, es fácil verificar que $\delta = mcd(p-q, q)$. Por lo tanto, calcular δ se reduce a encontrar el máximo común divisor de $p' = p-q$ y $q' = q$. Claramente, $(p'+q') < (p+q)$. Por lo tanto, el proceso puede continuar hasta que los dos enteros restantessean iguales. Entonces, asignamos a δ el valor encontrado.

Lema (Lema de periodicidad): Si una cadena Y tiene dos periodos de tamaño p y q, y $|Y| \geq (p+q)$, entonces Y tiene periodo de tamaño $mcd(p,q)$.

10.1.2. El arreglo testigo

Introduciremos una función sobre cadenas que jugara un papel importante en nuestro algoritmo paralelo de comparación de cadenas.

Sea Y una cadena de longitud m y de periodo p. Sea $\pi(Y) = min(p, \lceil(m/2)\rceil)$, es decir, $\pi(Y)$ es igual al periodo p si Y es periodica; en otro caso, $\pi(Y)$ es igual a $\lceil(m/2)\rceil$. Una **función testigo** ϕ_{tstg} se define como:
(1) $\phi_{tstg}(1) = 0$
(2) $2 \leq i \leq \pi(Y)$, $\phi_{tstg}(i) = k$, donde k es algún índice para el cual $Y(k) \neq Y(i+k-1)$.

Intuitivamente, imaginamos que una copia de Y es colocada en la cabeza de Y, tal que la primera y la i-esima posición están alineadas. Entonces, las porciones traslapadas de las dos copias no pueden ser identicas, porque de otra forma $Y(1 : i-1)$ sería un periodo de Y. Por lo tanto, existe al menos una posición para la cual los símbolos correspondientes son diferentes. El índice k es una posición dada.

Una función testigo podrá ser representada por un arreglo $TESTIGO(1 : r)$ donde $r = \pi(Y)$.

Ejemplo. Considere las dos cadenas $Y = abcaabcab$ y $Z = abcabcab$, introducidas en el ejemplo anterior. Los siguientes son dos posibles arreglos $TESTIGOS$ correspondientes a Y y Z:
$Y : TESTIGO = (0, 3, 2, 2, 5)$
$Z : TESTIGO = (0, 3, 2)$

Dada una cadena Z de longitud $n \geq m$, considere el problema de determinar todas las posibles posiciones donde la cadena Y pueda aparecer en Z. El arreglo $TESTIGO$ que le corresponde a Y proporciona una herramienta poderosa para descalificar multiples posiciones de Z como posibles candidatos para comparar.

Sea i y j dos posiciones arbitrarias de Z tal que $|i - j| < \pi(Y)$. Y no puede aparecer en las posiciones i y j de manera simultanea. Coloquemos dos copias de Y en la cabeza de Z, una empezando en la posición i, y la otra empezando en la posición j como se muestra en la Figura 11.2. Entonces, $TESTIGO(j - i + 1)$ proporciona una posición en la cual las dos copias de Y difieren. Por lo tanto, una simple prueba de comparación de simbolos en esos lugares y los simbolos que corresponden a esos lugares de Z eliminarán al menos una de las posiciones de i y j para una posible ocurrencia de Y en Z.

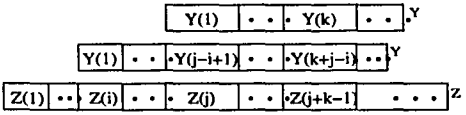


Figura 10.2: Eliminación de índices

El Algoritmo DUELO esta dado de manera más formal.

Ejemplo. Sea $Y = abcabcab$ con el siguiente arreglo $TESTIGO = (0, 3, 2)$, y sea $Z = abcaabcabaa$. Considere las posiciones $i = 5$ y $j = 7$ de Z. Ya que $TESTIGO(7-5+1) = 2$, el algoritmo duelo, verifica $Y(2) = b$ contra $Z(8) = a$. Ya que $Y(2) \neq Z(8)$, el índice $i = 5$ se regresa.

Lema. Sean Y y Z dos cadenas dadas, y sea $i < j$ dos índices distintos de Z tal que $(j - i) < \pi(Y)$. Entonces, que Y no aparezca en Z puede ocurrir en la posición eliminada por la opración $duelo(i, j)$. Más aún, este procedimiento toma tiempo secuencial $O(1)$.

```
Duelo
        // ENTRADA: (1) Una cadena Z de longitud n
        // (2) Un arreglo testigo de otra cadena Y de
        // longitud m <= n
        // (3) Dos indices i,j, donde 1<=i<=j<=n tal que
        // (j-i)< pi(Y)
        // SALIDA: Uno de i o j; cadena Y que no aparece
        // en Z en la posicion eliminada

begin
        1. Asigna k:= TESTIGO(j-i+1)

        2. if Z(j+k-1)!= Y(k)
              return (i)
           else
              return (j)

end
```

Figura 10.3: ALGORITMO DUELO

Demostración: Ya que $2 \leq (j-i+1) \leq p$, tenemos que $TESTIGO(j-i+1) = k \neq 0$. Este hecho implica que $Y(k) \neq Y(k+j-1)$. Consideremos el lugar $(j+k-1)$ de Z. No podemos tener simultaneamente las dos igualdades $Z(j+k-1) = Y(k)$ y $Z(j+k-1) = Y(k+j-1)$ como se muestra en la Figura 11.2. Claramente, si $Y(k) \neq Z(j+k-1)$, entonces Y no puede aparecer en el lugar j en Z. Por lo tanto, el índice i es regresado. Si $Z(j+k-1) = Y(k)$, entonces se regresa j, ya que $Z(j+k-1) \neq Y(k+j-1)$, y por lo tanto Y no puede estar en el lugar i de Z. En cada caso, ninguna comparación se hace en el índice eliminado. Observe que el índice que es regresado no necesariamente representa la comparación en ese lugar.

Bibliografía

[1] **MAMBER, Udi**
Introduction to Algorithms. A Creative Approach.
Addison Wesley, Pu. Co. USA.

[2] **JÁJÁ Joseph**
An Introduction to Parallel Algorithms
University of Maryland, Addison Wesley, Pu. Co. USA.

[3] **CORMEN Thomas H.**
Introduction to Algorithms.
The MIT Press. McGraw Hill Book Company.

[4] **RAWLINS, Gregory E.**
Compared to What? An Introduction to the Analysis of Algorithms
University of Indiana. Computer Science Press.

[5] **XAVIER, C.**
Introduction to Parallel Algorithms
A Wiley-interscience publication