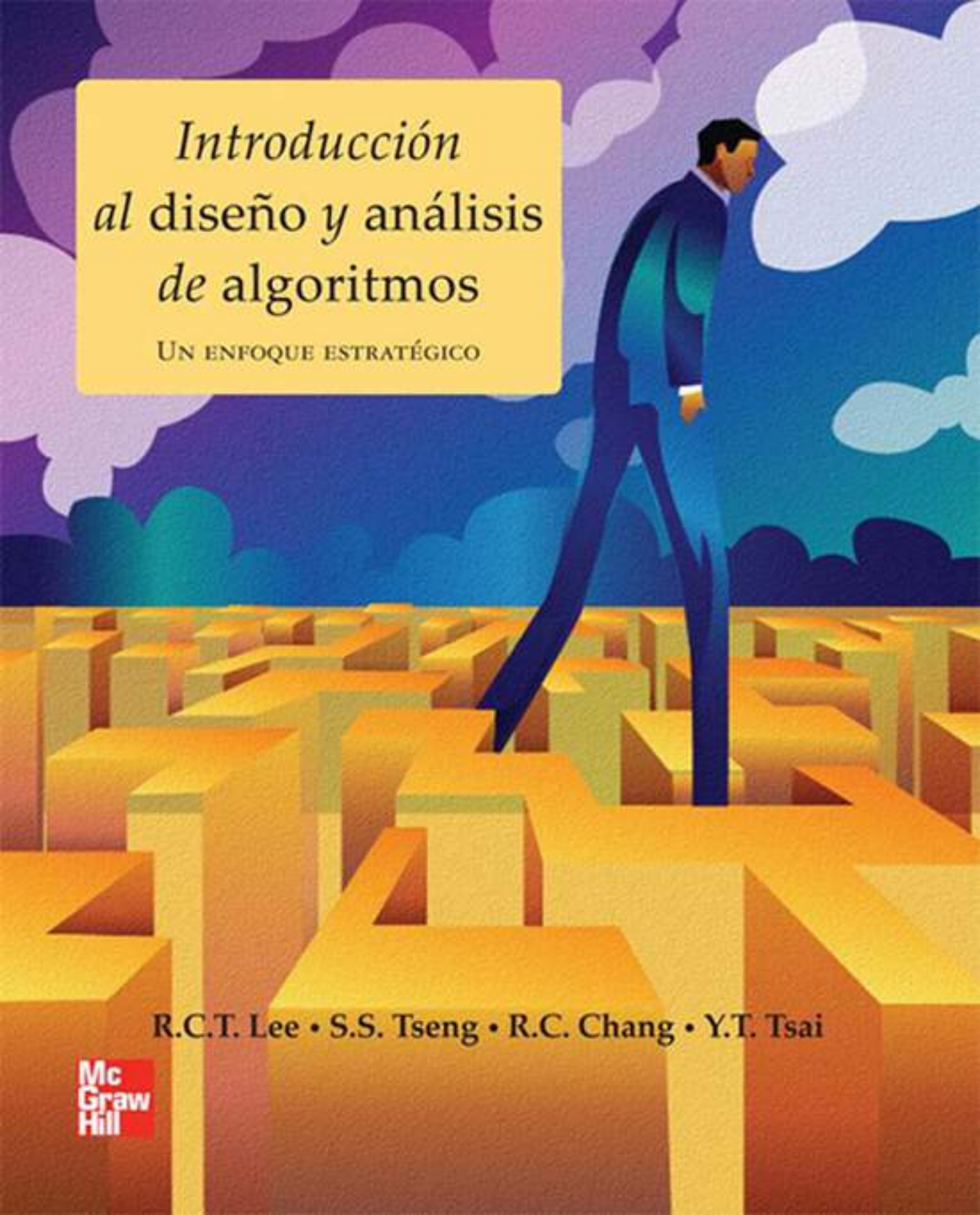

Introducción
al diseño y análisis
de algoritmos
Un enfoque estratégico
R.C.T. Lee • S.S. Tseng • R.C. Chang • Y.T. Tsai
Mc
Graw
Hill

Introducción al diseño y análisis de algoritmos

Un enfoque estratégico

Introducción al diseño y análisis de algoritmos

Un enfoque estratégico

R. C. T. Lee
National Chi Nan University, Taiwan

S. S. Tseng
National Chiao Tung University, Taiwan

R. C. Chang
National Chiao Tung University, Taiwan

Y. T. Tsai
Providence University, Taiwan

Revisión técnica

Miguel A. Orozco Malo
Instituto Tecnológico y de Estudios Superiores de Monterrey, Campus Ciudad de México

Jorge Valeriano Assem
Universidad Nacional Autónoma de México

Carlos Villegas Quezada
Universidad Iberoamericana

MÉXICO • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA
LISBOA • MADRID • NUEVA YORK • SAN JUAN • SANTIAGO
AUCKLAND • LONDRES • MILÁN • MONTREAL • NUEVA DELHI
SAN FRANCISCO • SINGAPUR • SAN LUIS • SIDNEY • TORONTO

Director Higher Education: Miguel Ángel Toledo Castellanos
Director editorial: Ricardo del Bosque Alayón
Editor sponsor: Pablo Eduardo Roig Vázquez
Editora de desarrollo: Ana Laura Delgado Rodríguez
Supervisor de producción: Zeferino García García

Traducción: Hugo Villagómez Velázquez

**INTRODUCCIÓN AL DISEÑO
Y ANÁLISIS DE ALGORITMOS.
Un enfoque estratégico**

Edificio Punta Santa Fe
Prolongación Paseo de la Reforma 1015, Torre A
Piso 17, Colonia Desarrollo Santa Fe
Delegación Álvaro Obregón
C.P. 01376, México, D.F.
Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. Núm. 736

Traducido de la primera edición de: Introduction to the Design and Analysis of Algorithms.
A Strategic Approach.

ISBN 007-124346-1

ISBN-13: 978-970-10-6124-4
ISBN-10: 970-10-6124-1

1234567890 0986543210

Impreso en México *Printed in Mexico*

The McGraw-Hill Companies

Acerca de los autores

R.C.T. LEE se recibió como maestro en ciencias en el Departamento de Ingeniería Eléctrica de la Universidad Nacional de Taiwán y se doctoró en el Departamento de Ingeniería Eléctrica y Ciencias de Computación de la Universidad de California en Berkeley. Es profesor en los departamentos de Ciencias de Computación e Ingeniería de la Información en la Universidad Nacional Chi Nan. El profesor Lee es miembro de la IEEE. Es coautor del libro *Symbolic Logic and Mechanical Theorem Proving*, que ha sido traducido al japonés, ruso e italiano.

S.S TSENG y **R.C. CHANG** recibieron sus grados de maestría y doctorado en el Departamento de Ingeniería de Computación de la Universidad Nacional Chiao Tung de Taiwán, donde ambos son profesores en el Departamento de Computación y Ciencias de la Información.

Y.T. TSAI terminó la maestría en el Departamento de Computación y Ciencias de la Información en la Universidad Nacional Chiao Tung de Taiwán, y se doctoró en el Departamento de Ciencias de Computación en la Universidad Nacional Tsing-Hua. Actualmente es profesor asociado en el Departamento de Ciencias de la Información y Administración en la Universidad Providence.

Contenido

Prefacio

Existen múltiples razones para estudiar algoritmos. La principal es eficiencia. Suele creerse que para obtener altas velocidades de cálculo basta contar con una computadora de muy alta velocidad. Sin embargo, no es completamente cierto. Un buen algoritmo implementado en una computadora lenta puede ejecutarse mucho más rápido que un mal algoritmo implementado en una computadora rápida. Imagine que un programador requiere encontrar un árbol de expansión mínima para un problema suficientemente grande. Si su programa examina todos los posibles árboles de expansión, no existirá ni hoy, ni en el futuro, una computadora capaz de resolver el problema. En cambio, si conoce el método de Prim, con una PC le basta. Otro ejemplo: cuando alguien quiere resolver el problema del reconocimiento del habla, es muy difícil siquiera comenzar a resolver el problema. Pero, si conoce que el problema de subsecuencia común más larga puede ser resuelto por medio de programación dinámica, se sorprendería con la sencillez con la que resolvería el problema.

El estudio de algoritmos no es exclusivo de las ciencias de la computación. Los ingenieros en comunicaciones también utilizan y estudian programación dinámica o algoritmos A*. Sin embargo, el grupo más grande de científicos interesados en algoritmos, ajenos a las ciencias de la computación, es el del campo de la biología molecular. Por ejemplo, si se desea comparar dos secuencias de ADN, dos proteínas, o secuencias RNA, estructuras 3-dimensionales, entonces es necesario conocer algoritmos eficientes.

Además, estudiar algoritmos puede resultar divertido. Los autores continúan entusiasmados cada que encuentran un algoritmo nuevo y bien diseñado, o alguna idea novedosa sobre diseño y análisis de algoritmos. Se sienten con la responsabilidad moral de compartir su diversión y emoción. Muchos problemas, que aparentan ser difíciles, pueden ser resueltos con algoritmos polinomiales, mientras que otros problemas, que a primera vista parecen sencillos, pueden ser NP-completo. Por ejemplo, el árbol de expansión mínima parece ser un problema difícil para nuevos lectores, aunque luego descubren que puede resolverse con algoritmos polinomiales. En contraste, si se representa como un problema euclidiano del agente viajero, repentinamente se convierte en un problema NP difícil. Otro caso es el 3-satisfactibilidad, que es un problema NP, pero si se decrece su dimensión, el problema de 2-satisfactibilidad se convierte en un problema P. Para quien estudia algoritmos, es fascinante descubrir esto.

En este libro utilizamos un enfoque orientado al diseño para la introducción al estudio de algoritmos. De hecho, no estamos dando una introducción a los algoritmos, sino que presentamos estrategias para diseñar algoritmos. La razón es muy simple: es mucho más importante conocer los principios (estrategias) del diseño de algoritmos, que únicamente conocer los algoritmos.

Sin embargo, no todos los algoritmos se basan en las estrategias que presentamos, sino que a veces, son la estrategia misma. Por ejemplo, la estrategia prune-and-search, el análisis amortizado, los algoritmos en línea y la aproximación polinomial son ideas bastante novedosas e importantes. Gran parte de los algoritmos nuevos se basan en el análisis amortizado, tal como se explica en el capítulo de ese tema.

Comenzamos con una introducción a las estrategias que se utilizan en el diseño de algoritmos polinomiales. Cuando es necesario enfrentarse a problemas que parecen difíciles y no se conocen algoritmos polinomiales se introduce el concepto del problema NP-completo. Es sencillo aplicar la teoría NP-completo, aunque en ocasiones es difícil comprender su significado en el mundo real. La idea central es porque de hecho cada problema NP se relaciona con un conjunto de fórmulas booleanas y el resultado para este problema es "sí", si y sólo si las fórmulas son correctas. Una vez que el lector comprende esto, es sencillo que descubra la importancia de la teoría del problema NP-completo. Estamos seguros que los ejemplos que presentamos para explicar esta idea ayudarán a los estudiantes a comprender dicho concepto.

Éste es un libro de texto para estudiantes de licenciatura y posgrado. En nuestra experiencia hemos visto que no se alcanzan a cubrir todos los temas en un solo semestre. Por ello, recomendamos que, si sólo se cuenta con un semestre para el curso, se revisen brevemente todos los capítulos.

El capítulo que trata sobre la teoría de los problemas NP-completo es sumamente importante y debe quedar claro a todos los alumnos. El capítulo más difícil es el 10, porque utiliza muchas matemáticas, y trata sobre análisis amortizado. El profesor debe poner énfasis en las ideas básicas del análisis amortizado para que los alumnos comprendan por qué una estructura básica, ligada a un buen algoritmo, puede desempeñarse bien, en un sentido amortizado. Otro capítulo que presenta dificultad es el 12, que es sobre algoritmos en línea.

La mayoría de los algoritmos son fáciles de comprender y hemos hecho un esfuerzo por presentarlos de la manera más clara posible. Cada algoritmo se presenta acompañado de ejemplos y cada ejemplo se apoya en figuras. Este libro cuenta con más de 400 figuras, para hacerlo útil a aquellos que comienzan a estudiar este tema.

Asimismo, se incluye bibliografía de libros y revistas especializadas en diseño y análisis de algoritmos. Se resaltan los descubrimientos más nuevos, de manera que los lectores puedan localizarlos y ampliar su estudio. La bibliografía es extensa, abarca 825 libros y artículos, de 1 095 autores.

Hemos presentado resultados experimentales donde fue apropiado. El instructor debe alentar a sus alumnos para que implementen ellos mismos los algoritmos. Al final de cada capítulo enlistamos algunos artículos o resultados de investigaciones como sugerencias de lecturas complementarias. Es conveniente que los alumnos realicen estas lecturas, ya que les ayudarán a comprender mejor los algoritmos.

Es imposible nombrar a todas las personas que han ayudado a los autores con este libro. Sin embargo, todos pertenecen a una clase: son alumnos y colegas (y alumnos que se han convertido en colegas) de los autores. En el seminario semanal hemos tenido discusiones, que nos han señalado nuevas rutas en el diseño de algoritmos y nos han ayudado a escoger el material que está en este libro. Nuestros estudiantes de licenciatura fueron quienes revisaron diversas revistas académicas sobre el tema y recopilaron los mejores documentos sobre algoritmos, con el fin de tener una base de datos de palabras clave. Dicha base de datos fue imprescindible para escribir este libro. Finalmente, participaron en la lectura del borrador de esta obra, nos ofrecieron sus críticas y realizaron ejercicios. Nunca hubiéramos completado este libro sin la ayuda de nuestros alumnos y colegas. Estamos sumamente agradecidos con todos ellos.

capítulo

1

Introducción

En este libro se presenta el diseño y análisis de algoritmos. Tal vez tenga sentido analizar primero una cuestión muy importante: ¿por qué es necesario estudiar algoritmos? Suele creerse que para obtener altas velocidades de cálculo basta contar con una computadora de muy alta velocidad. Sin embargo, lo anterior no es completamente cierto. Esto se ilustra con un experimento cuyos resultados evidencian que un buen algoritmo implementado en una computadora lenta puede ejecutarse mucho mejor que un mal algoritmo implementado en una computadora rápida. Considere los dos algoritmos de ordenamiento, el ordenamiento por inserción y el algoritmo quick sort (ordenamiento rápido), que se describen de manera aproximada a continuación.

El ordenamiento por inserción analiza de izquierda a derecha, uno por uno, los datos de una sucesión. Una vez que se ha analizado cada dato, se inserta en un sitio idóneo en una sucesión ya ordenada. Por ejemplo, suponga que la sucesión de datos a ordenar es

11, 7, 14, 1, 5, 9, 10.

El ordenamiento por inserción funciona sobre la sucesión anterior como se muestra a continuación:

Sucesión ordenada	**Sucesión no ordenada**
11	7, 14, 1, 5, 9, 10
7, 11	14, 1, 5, 9, 10
7, 11, 14	1, 5, 9, 10
1, 7, 11, 14	5, 9, 10
1, 5, 7, 11, 14	9, 10
1, 5, 7, 9, 11, 14	10
1, 5, 7, 9, 10, 11, 14	

En el proceso anterior, considere el caso en que la sucesión ordenada es 1, 5, 7, 11, 14 y que el siguiente dato a ordenar es 9. Primero, 9 se compara con 14. Como 9 es menor que 14, a continuación 9 se compara con 11. De nuevo es necesario continuar. Al comparar 9 con 7, se encuentra que 9 es mayor que 7. Así, 9 se inserta entre 9 y 11.

El segundo algoritmo de clasificación se denomina quick sort, y se presentará con todo detalle en el capítulo 2. Mientras tanto, solamente se proporcionará una descripción a muy alto nivel de este algoritmo. Suponga que se quiere ordenar la siguiente sucesión de datos:

10, 5, 1, 17, 14, 8, 7, 26, 21, 3.

El quick sort usa el primer elemento de los datos; a saber, 10, para dividir los datos en tres subconjuntos: los elementos menores que 10, los mayores que 10 y los iguales a 10. Es decir, los datos se representan como sigue:

(5, 1, 8, 7, 3) (10) (17, 14, 26, 21).

Ahora es necesario ordenar dos secuencias:

(5, 1, 8, 7, 3)
y (17, 14, 26, 21).

Observe que estas secuencias pueden ordenarse independientemente y que el método de división puede aplicarse en forma recurrente a los dos subconjuntos. Por ejemplo, considere

17, 14, 26, 21.

Al usar 17 para dividir el conjunto anterior de datos, se tiene

(14) (17) (26, 21).

Después de ordenar la secuencia anterior 26, 21 en 21, 26 se obtiene

14, 17, 21, 26

que es una secuencia ordenada.

De manera semejante,

(5, 1, 8, 7, 3)

puede ordenarse como

(1, 3, 5, 7, 8).

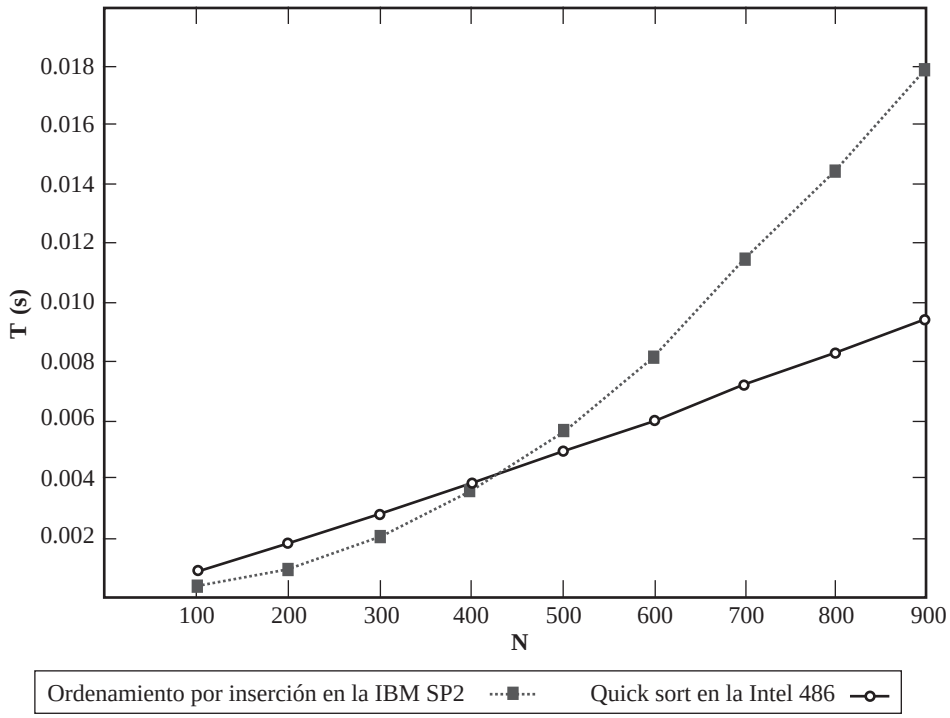
Al combinar todas las secuencias ordenadas en una se tiene

1, 3, 5, 7, 8, 10, 14, 17, 21, 26

que es una secuencia ordenada.

Después se demostrará que el quick sort es mucho mejor que el ordenamiento por inserción. La pregunta es: ¿cuánto mejor? Para comparar el quick sort con el ordenamiento por inserción, aquélla se implementó en una Intel 486, y ésta, en una IBM SP2. La IBM SP2 es una supercomputadora que derrotó al campeón de ajedrez en 1997, mientras que la Intel 486 es sólo una computadora personal. Para cada número de puntos, aleatoriamente se generaron 10 conjuntos de datos y para ambos algoritmos se obtuvo el tiempo promedio. En la figura 1-1 se muestran los resultados del experimento.

FIGURA 1-1 Comparación del desempeño del ordenamiento por inserción y del quick sort.



Puede verse que cuando el número de datos es menor a 400, el tiempo que requiere el algoritmo quick sort implementado en la Intel 486 es inferior al que requiere el ordenamiento por inserción implementado en la IBM SP2. Para un número de datos mayor que 400, la Intel 486 funciona mucho mejor que la IBM SP2.

¿Qué significa el experimento anterior? Ciertamente, indica un hecho importante: una computadora rápida con un algoritmo inferior puede desempeñarse peor que una computadora lenta con un algoritmo superior. En otras palabras, *si usted es rico, y no conoce mucho sobre algoritmos, tal vez no esté preparado para competir con alguien pobre que sepa mucho de algoritmos*.

Una vez aceptado que es importante estudiar algoritmos, es necesario poder analizar algoritmos a fin de determinar su desempeño. En el capítulo 2 se presenta una breve introducción de algunos conceptos fundamentales relacionados con el análisis de algoritmos. En este capítulo nuestro análisis de algoritmos es introductorio en todos los sentidos. Al final del capítulo 2 se presenta una bibliografía que incluye algunos libros excelentes sobre análisis de algoritmos.

Luego de presentar el concepto de análisis de algoritmos, volvemos la atención a la complejidad de los problemas. Se observa que hay problemas fáciles y problemas difíciles. Un problema es fácil si puede resolverse mediante un algoritmo eficiente, tal vez un algoritmo con complejidad temporal polinomial. Al revés, si un problema no puede resolverse con ningún algoritmo con complejidad temporal polinomial, entonces debe tratarse de un problema difícil. Por lo general, dado un problema, si ya se sabe que existe un algoritmo que resuelve el problema en tiempo polinomial, se tiene la certeza de que es un problema fácil. No obstante, si no se ha encontrado ningún algoritmo polinomial que resuelva el problema, difícilmente puede concluirse que jamás podrá encontrarse alguno con complejidad temporal polinomial que resuelva el problema. Por fortuna, existe una teoría de algoritmos NP-completos que puede aplicarse para medir la complejidad de un problema. Si se demuestra que un problema es NP-completo, entonces se considerará como un problema difícil y la probabilidad de encontrar un algoritmo de tiempo polinomial para resolver el problema es bastante pequeña. Normalmente el concepto de problemas NP-completos (NP-completeness) se presenta al final de los libros de texto.

Resulta interesante observar aquí que existe una multitud de problemas que no parecen difíciles, pero que en realidad son problemas NP-completos. A continuación se proporcionan algunos ejemplos.

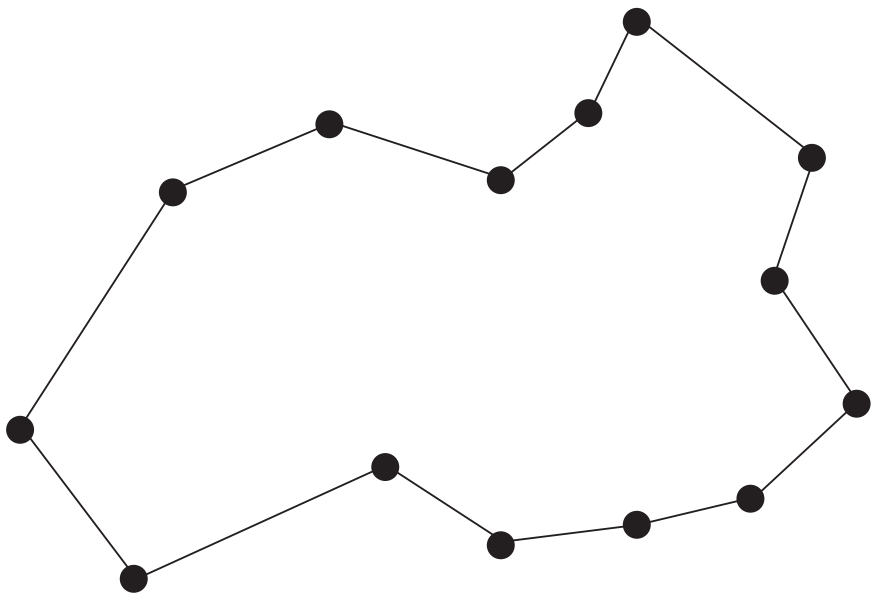
Antes que todo, considere el problema de la mochila 0/1. Una descripción informal de este problema es la siguiente: imagine que se está huyendo de un ejército invasor. Es necesario abandonar la amada patria. Se desea llevar algunos artículos de valor. Pero el peso total de los artículos no puede exceder cierto límite. ¿Cómo es posible maximizar el valor de los artículos que es posible llevar sin exceder el límite de peso? Por ejemplo, suponga que se cuenta con los siguientes artículos:

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
Valor	10	5	1	9	3	4	11	17
Peso	7	3	3	10	1	9	22	15

Si el límite de peso es 14, entonces la mejor solución es seleccionar P_1, P_2, P_3 y P_5. Resulta que este problema es un problema NP-completo. A medida que aumenta la cantidad de artículos se hace más difícil encontrar una solución óptima.

Otro problema que también es aparentemente fácil es un problema NP-completo. Se trata del problema del agente viajero. Para explicarlo intuitivamente, se considera que hay muchas ciudades. El diligente agente viajero debe viajar a cada ciudad, pero requiere que ninguna ciudad sea visitada dos veces y que el recorrido sea el más corto. Por ejemplo, en la figura 1-2 se muestra una solución óptima en un ejemplo del problema del agente viajero, donde la distancia entre dos ciudades es su distancia euclidiana.

FIGURA 1-2 Solución óptima de un ejemplo del problema del agente viajero.



Finalmente, se considerará el problema de partición. En este problema se tiene un conjunto de enteros. La pregunta es: ¿es posible partir estos enteros en dos subcon-

juntos S_1 y S_2 de modo que la suma de S_1 sea igual a la suma de S_2? Por ejemplo, el conjunto

$$\{1, 7, 10, 9, 5, 8, 3, 13\}$$

puede partirse como

$$S_1 = \{1, 10, 9, 8\}$$

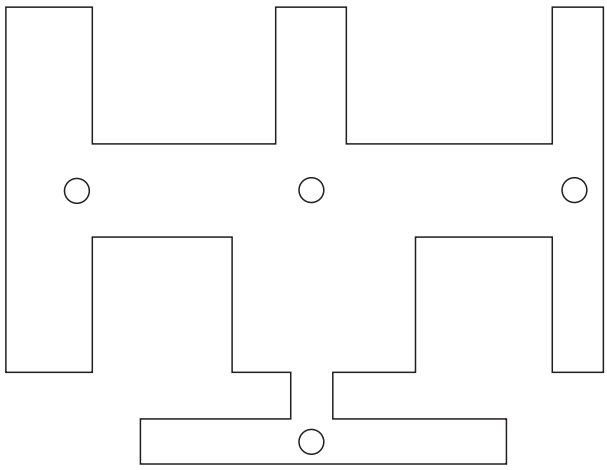
y $S_2 = \{7, 5, 3, 13\}$.

Puede demostrarse que la suma de S_1 es igual a la suma de S_2.

Este problema de partición debe resolverse muy a menudo. Por ejemplo, un concepto clave de encriptación incluye este problema. Aunque parece sencillo, se trata de un problema NP-completo.

He aquí otro problema que resultará de interés para muchos lectores. Todos saben que hay galerías de arte que guardan obras maestras de valor incalculable. Es de suma importancia que estas obras de arte no sean dañadas ni robadas. Así, es necesario que las galerías estén custodiadas por guardianes. Considere la figura 1-3, que representa una galería de arte.

FIGURA 1-3 Una galería de arte y sus guardianes.



Si en la galería de arte hay cuatro guardianes, como se indica con los círculos en la figura 1-3, entonces cada muro de la galería estará controlado al menos por un guardián. Lo anterior significa que toda la galería de arte estará suficientemente vigilada por estos cuatro guardianes. El problema de la galería de arte es: dada una galería de arte en forma poligonal, determinar el número mínimo de guardianes y sus ubicacio-

nes, de modo que toda la galería de arte quede vigilada. Para muchos lectores quizá sea una gran sorpresa enterarse de que también este problema es NP-completo.

Incluso si se aprecia el hecho de que los buenos algoritmos son esenciales, aún podría plantearse la pregunta de si es importante estudiar el diseño de algoritmos porque es posible la obtención fácil de buenos algoritmos. En otras palabras, si un buen algoritmo puede obtenerse simplemente por intuición, o por sentido común, entonces no vale la pena el esfuerzo de estudiar el diseño de algoritmos.

Cuando después se presente el problema del árbol de expansión mínima, se verá que este problema combinatorio aparentemente explosivo en realidad cuenta con un algoritmo muy sencillo que puede usarse para resolver de manera eficiente el problema. A continuación se proporciona una descripción informal del problema del árbol de expansión mínima.

Imagine que se tienen muchas ciudades, como se muestra en la figura 1-4. Suponga que se quiere conectar todas las ciudades en un árbol de expansión (un árbol de expansión es una gráfica que une sin ciclos todas las ciudades) de forma que se minimice la distancia total del árbol. Por ejemplo, para el conjunto de ciudades que se muestra en la figura 1-4, un árbol de expansión mínima se muestra en la figura 1-5.

FIGURA 1-4 Conjunto de ciudades para ilustrar el problema del árbol de expansión mínima.

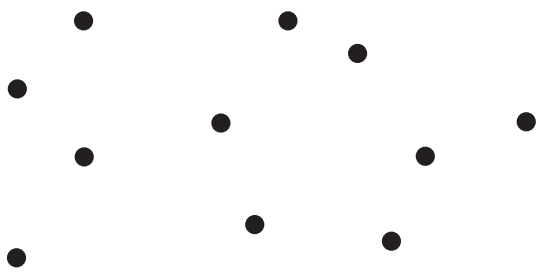
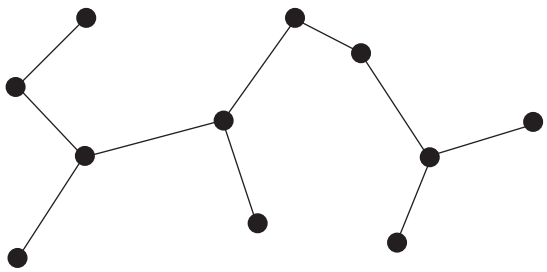


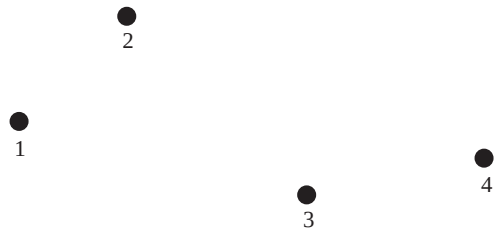
FIGURA 1-5 Árbol de expansión mínima para el conjunto de ciudades de la figura 1-4.



Un algoritmo intuitivo para encontrar un árbol de expansión mínima consiste en encontrar todos los árboles de expansión posibles. Para cada árbol de expansión se determina su longitud total y una vez que se han enumerado de manera exhaustiva todos los árboles de expansión posibles se puede encontrar un árbol de expansión mínima. Resulta que lo anterior es muy costoso debido a que, en efecto, la cantidad de árboles de expansión posibles es bastante grande. Puede demostrarse que el número total de árboles de expansión posibles para n ciudades es n^{n-2}. Suponga que $n = 10$. Es necesario enumerar 10^8 árboles. Cuando n es igual a 100, este número aumenta a 100^{98} (10^{196}), que es tan grande que no hay ninguna computadora capaz de manipular esta cantidad de datos. En Estados Unidos, una compañía telefónica puede tener que enfrentar el caso en que n es igual a 5 000. Así, es necesario contar con un mejor algoritmo.

En realidad, para resolver el problema de árbol de expansión mínima existe un algoritmo excelente. Este algoritmo se ilustrará con un ejemplo. Considere la figura 1-6. El algoritmo en cuestión primero encuentra que d_{12} es la menor de las distancias que comunican las ciudades. En consecuencia, es posible conectar las ciudades 1 y 2, como se muestra en la figura 1-7*a*). Luego de hacer lo anterior, {1, 2} se considera como un conjunto y el resto de las demás ciudades se considera como otro conjunto. Luego se encuentra que la distancia más corta entre estos dos conjuntos de ciudades es d_{23}. Se unen las ciudades 2 y 3. Por último, se conectan 3 y 4. En la figura 1-7 se muestra todo el proceso.

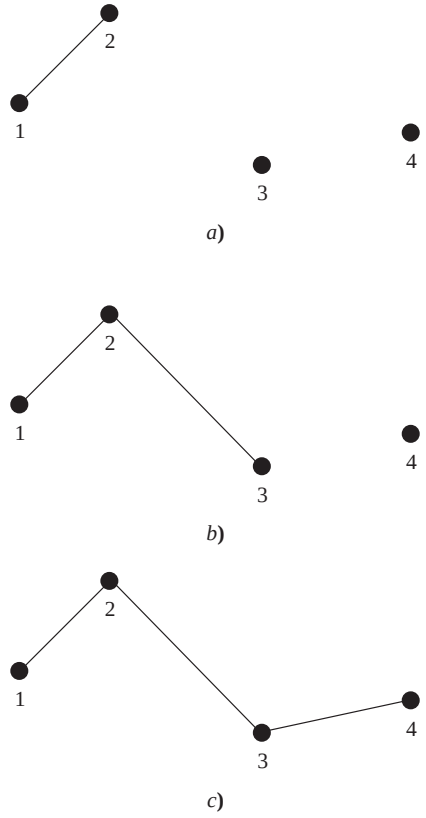
FIGURA 1-6 Ejemplo para ilustrar un algoritmo eficiente de árbol de expansión mínima.



Puede demostrarse que este sencillo algoritmo eficiente siempre produce una solución óptima. Es decir, el árbol de expansión mínima final que produce siempre es un árbol de expansión mínima. La demostración de que este algoritmo es correcto no es de ninguna manera fácil. La estrategia detrás de este algoritmo se denomina método codicioso (greedy). Normalmente un algoritmo basado en el método codicioso es bastante eficiente.

Desafortunadamente muchos problemas semejantes al problema del árbol de expansión mínima no pueden resolverse con el método codicioso. Por ejemplo, uno de tales problemas es el problema del agente viajero.

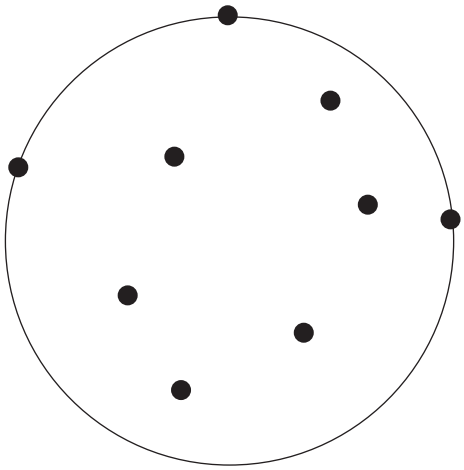
FIGURA 1-7 Ilustración del algoritmo del árbol de expansión mínima.



Un ejemplo que no es posible resolver fácilmente mediante una búsqueda exhaustiva es el problema con un centro. En este problema se tiene un conjunto de puntos y es necesario encontrar un círculo que cubra todos estos puntos de modo que se minimice el radio del círculo. Por ejemplo, en la figura 1-8 se muestra una solución óptima de un problema con un centro. ¿Cómo empezará a trabajarse este problema? Después, en este libro, se demostrará que el problema con un centro puede resolverse con la estrategia de prune-and-search.

El estudio del diseño de algoritmos es casi un estudio de estrategias. Gracias a muchos investigadores ha sido posible descubrir estrategias excelentes que pueden aplicarse para diseñar algoritmos. No es posible afirmar que todo algoritmo excelente debe estar basado en una de las estrategias generales. Sin embargo, definitivamente sí puede afirmarse que para estudiar algoritmos es verdaderamente valioso tener un conocimiento completo de las estrategias.

FIGURA 1-8 Solución de un problema con un centro.



A continuación se recomienda una lista de libros sobre algoritmos que constituyen excelentes referencias como lecturas adicionales.

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

Basse, S. y Van Gelder, A. (2000): *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, Reading, Mass.

Brassard, G. y Bratley, P. (1988); *Algorithmics: Theory and Practice*, Prentice-Hall, Englewood Cliffs, Nueva Jersey.

Coffman, E. G. y Lueker, G. S. (1991): *Probabilistic Analysis of Packaging & Partitioning Algorithms*, John Wiley & Sons, Nueva York.

Cormen, T. H. (2001): *Introduction to Algorithms*, McGraw-Hill, Nueva York.

Cormen, T. H., Leiserson, C. E. y Rivest, R. L. (1990): *Introduction to Algorithms*, McGraw-Hill, Nueva York.

Dolan A. y Aldous J. (1993): *Networks and Algorithms: An Introductory Approach*, John Wiley & Sons, Nueva York.

Evans, J. R. y Minieka, E. (1992): *Optimization Algorithms for Networks and Graphs*, 2a. ed., Marcel Dekker, Nueva York.

Garey, M. R. y Johnson, D. S. (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, California.

Gonnet, G. H. (1983): *Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, Mass.

Goodman, S. y Hedetniemi, S. (1980): *Introduction to the Design and Analysis of Algorithms*, McGraw-Hill, Nueva York.

Gould, R. (1988): *Graph Theory*, Benjamin Cummings, Redwood City, California.

Greene, D. H. y Knuth, D. E. (1981): *Mathematics for the Analysis of Algorithms*, Birkhäuser, Boston, Mass.

Hofri, M. (1987): *Probabilistic Analysis of Algorithms*, Springer-Verlag, Nueva York.

Horowitz, E. y Sahni, S. (1976): *Fundamentals of Data Structures*, Computer Science Press, Rockville, Maryland.

Horowitz, E., Sahni, S. y Rajasekaran, S. (1998): *Computer Algorithms*, W. H. Freeman, Nueva York.

King, T. (1992): *Dynamic Data Structures: Theory and Applications*, Academic Press, Londres.

Knuth, D. E. (1969): *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, Reading, Mass.

Knuth, D. E. (1973): *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass.

Kozen, D. C. (1992): *The Design and Analysis of Algorithms*, Springer-Verlag, Nueva York.

Kronsjö, L. I. (1987): *Algorithms: Their Complexity and Efficiency*, John Wiley & Sons, Nueva York.

Kucera, L. (1991): *Combinatorial Algorithms*, IOP Publishing, Filadelfia.

Lewis, H. R. y Denenberg, L. (1991): *Data Structures and Their Algorithms*, Harper Collins, Nueva York.

Manber, U. (1989): *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, Reading, Mass.

Mehlhorn, K. (1987): *Data Structures and Algorithms*: *Sorting and Searching*, Springer-Verlag, Nueva York.

Moret, B. M. E. y Shapiro, H. D. (1991): *Algorithms from P to NP*, Benjamin Cummings, Redwood City, California.

Motwani, R. y Raghavan P. (1995): *Randomized Algorithms*, Cambridge University Press, Cambridge, Inglaterra.

Mulmuley, K. (1998): *Computational Geometry: An Introduction through Randomized Algorithms*, Prentice-Hall, Englewood Cliffs, Nueva Jersey.

Neapolitan, R. E. y Naimipour, K. (1996): *Foundations of Algorithms*, D. C. Heath and Company, Lexington, Mass.

Papadimitriou, C. H. (1994): *Computational Complexity*, Addison-Wesley, Reading, Mass.

Purdom, P. W. Jr. y Brown, C. A. (1985): *The Analysis of Algorithms*, Holt, Rinehart and Winston, Nueva York.

Reingold, E. Nievergelt, J. y Deo, N. (1977): *Combinatorial Algorithms, Theory and Practice*, Prentice-Hall, Englewood Cliffs, Nueva Jersey.

Sedgewick, R. y Flajolet, D. (1996): *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, Mass.

Shaffer, C. A. (2001): *A Practical Introduction to Data Structures and Algorithm Analysis*, Prentice-Hall, Englewood Cliffs, Nueva Jersey.

Smith, J. D. (1989): *Design and Analysis of Algorithms*, PWS Publishing, Boston, Mass.

Thulasiraman, K. y Swamy, M. N. S. (1992): *Graphs: Theory and Algorithms*, John Wiley & Sons, Nueva York.

Uspensky, V. y Semenov, A. (1993): *Algorithms: Main Ideas and Applications*, Kluwer Press, Norwell, Mass.

Van Leeuwen, J. (1990): *Handbook of Theoretical Computer Science: Volume A: Algorithms and Complexity*, Elsevier, Amsterdam.

Weiss, M. A. (1992): *Data Structures and Algorithm Analysis*, Benjamin Cummings, Redwood City, California.

Wilf, H. S. (1986): *Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, Nueva York.

Wood, D. (1993): *Data Structures, Algorithms, and Performances*, Addison-Wesley, Reading, Mass.

Para estudios avanzados, se recomiendan los libros siguientes:

Para geometría computacional

Edelsbrunner, H. (1987): *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlín.

Mehlhorn, K. (1984): *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, Berlín.

Mulmuley, K. (1998): *Computational Geometry: An Introduction through Randomized Algorithms*, Prentice-Hall, Englewood Cliffs, Nueva Jersey.

O'Rourke, J. (1998): *Computational Geometry in C*, Cambridge University Press, Cambridge, Inglaterra.

Pach, J. (1993): *New Trends in Discrete and Computational Geometry*, Springer-Verlag, Nueva York.

Preparata, F. P. y Shamos, M. I. (1985): *Computational Geometry: An Introduction*, Springer-Verlag, Nueva York.

Teillaud, M. (1993): *Towards Dynamic Randomized Algorithms in Computational Geometry: An Introduction*, Springer-Verlag, Nueva York.

Para teoría de gráficas

Even, S. (1987): *Graph Algorithms*, Computer Science Press, Rockville, Maryland.

Golumbic, M. C. (1980): *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, Nueva York.

Lau, H. T. (1991): *Algorithms on Graphs*, TAB Books, Blue Ridge Summit, PA.

McHugh, J. A. (1990): *Algorithmic Graph Theory*, Prentice-Hall, Londres.

Mehlhorn, K. (1984): *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, Springer-Verlag, Berlín.

Nishizeki, T. y Chiba, N. (1988): *Planar Graphs: Theory and Algorithms*, Elsevier, Amsterdam.

Thulasiraman, K. y Swamy, M. N. S. (1992): *Graphs: Theory and Algorithms*, John Wiley & Sons, Nueva York.

Para optimización combinatoria

Lawler, E. L. (1976): *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, Nueva York.

Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G. y Shamoys, D. B. (1985): *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, John Wiley & Sons, Nueva York.

Martello, S. y Toth, P. (1990): *Knapsack Problem Algorithms & Computer Implementations*, John Wiley & Sons, Nueva York.

Papadimitriou, C. H. y Steiglitz, K. (1982): *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, Nueva Jersey.

Para estructuras de datos avanzadas

Tarjan, R. E. (1983): *Data Structures and Network Algorithms*, Society of Industrial and Applied Mathematics, Vol. 29.

Para biología computacional

Gusfield, D. (1997): *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, Cambridge, Inglaterra.

Pevzner, P. A. (2000): *Computational Molecular Biology: An Algorithmic Approach*, The MIT Press, Boston.

Setubal, J. y Meidanis, J. (1997): *Introduction to Computational Biology*, PWS Publishing Company, Boston, Mass.

Szpankowski, W. (2001): *Average Case Analysis of Algorithms on Sequences*, John Wiley & Sons, Nueva York.

Waterman, M. S. (1995): *Introduction to Computational Biology: Maps, Sequences and Genomes*, Chapman & Hall/CRC, Nueva York.

Para algoritmos de aproximación

Hochbaum, D. S. (1996): *Approximation Algorithms for NP-Hard Problems*, PWS Publisher, Boston.

Para algoritmos aleatorios

Motwani, R. y Raghavan, P. (1995): *Randomized Algorithms*, Cambridge University Press, Cambridge, Inglaterra.

Para algoritmos en línea

Borodin, A. y El-Yaniv, R. (1998): *Online Computation and Competitive Analysis*, Cambridge University Press, Cambridge, Inglaterra.

Fiat, A. y Woeginger, G. J. (editores) (1998): The State of the Arts, *Lecture Notes in Computer Science*, Vol. 1442. Springer-Verlag, Nueva York.

Hay muchos periódicos académicos que regularmente publican artículos sobre algoritmos. A continuación se recomiendan los más conocidos:

- *Acta Informatica*
- *Algorithmica*
- *BIT*
- *Combinatorica*
- *Discrete and Computational Geometry*
- *Discrete Applied Mathematics*
- *IEEE Transactions on Computers*
- *Information and Computations*
- *Information Processing Letters*
- *International Journal of Computational Geometry and Applications*
- *International Journal of Foundations on Computer Science*
- *Journal of Algorithms*
- *Journal of Computer and System Sciences*
- *Journal of the ACM*

- *Networks*
- *Proceedings of the ACM Symposium on Theory of Computing*
- *Proceedings of the IEEE Symposium on Foundations of Computing Science*
- *SIAM Journal on Algebraic and Discrete Methods*
- *SIAM Journal on Computing*
- *Theoretical Computer Science*

capítulo

2

Complejidad de los algoritmos y cotas inferiores de los problemas

En este capítulo se analizarán algunas cuestiones básicas relacionadas con el análisis de algoritmos. Esencialmente se intentará aclarar los siguientes temas:

1. Algunos algoritmos son eficientes y otros no. ¿Cómo medir la eficiencia de un algoritmo?
2. Algunos problemas son fáciles de resolver y otros no. ¿Cómo medir la dificultad de un problema?
3. ¿Cómo saber si un algoritmo es el óptimo para un problema? Es decir, ¿cómo es posible saber que no existe otro algoritmo mejor para resolver el mismo problema? Mostraremos que todos estos problemas están relacionados entre sí.

2-1 Complejidad temporal de un algoritmo

Suele decirse que un algoritmo es bueno si para ejecutarlo se necesita poco tiempo y si requiere poco espacio de memoria. Sin embargo, por tradición, un factor más importante para determinar la eficacia de un algoritmo es el tiempo necesario para ejecutarlo. A lo largo de todo este libro, a menos que se indique lo contrario, el criterio importante es el tiempo.

Para medir la complejidad temporal de un algoritmo, es tentador escribir un programa para este algoritmo y ver qué tan rápido corre. Esto no es lo apropiado porque hay muchos factores no relacionados con el algoritmo que afectan el desempeño del programa. Por ejemplo, la habilidad del programador, el lenguaje usado, el sistema operativo e incluso el compilador del lenguaje particular, todos estos factores afectan el tiempo necesario para ejecutar el programa.

En el análisis de algoritmos siempre se escogerá un tipo de operación particular que ocurra en el algoritmo, y se realizará un análisis matemático a fin de determinar el número de operaciones necesarias para completar el algoritmo. Por ejemplo, en todos

los algoritmos de ordenamiento debe hacerse la comparación de datos, por lo que el número de comparaciones suele emplearse para medir la complejidad temporal de algoritmos de ordenamiento.

Por supuesto, es legítimo refutar que en algunos algoritmos de ordenamiento la comparación de datos no es un factor dominante. De hecho, es fácil poner ejemplos de que en algunos algoritmos de ordenamiento el intercambio de datos es lo que consume más tiempo. Según tal circunstancia, parece que debería usarse el intercambio de datos, y no la comparación, para medir la complejidad temporal de este algoritmo de ordenamiento particular.

Suele decirse que el costo de ejecución de un algoritmo depende del tamaño del problema (n). Por ejemplo, el número de puntos en el problema euclidiano del agente viajero definido en el apartado 9-2 es el tamaño del problema. Como es de esperar, la mayoría de los algoritmos requiere más tiempo para completar su ejecución a medida que n crece.

Suponga que para ejecutar un algoritmo se requieren $(n^3 + n)$ pasos. Se diría a menudo que la complejidad temporal de este algoritmo es del orden de n^3. Debido a que el término n^3 es de orden superior a n y a medida que n se hace más grande, el término n pierde importancia en comparación con n^3. A continuación daremos un significado formal y preciso a esta informal afirmación.

Definición
$f(n) = O(g(n))$ si y sólo si existen dos constantes positivas c y n_0 tales que $|f(n)| \leq c|g(n)|$ para toda $\geq n_0$.

Por la definición anterior se entiende que, si $f(n) = O(g(n))$, entonces $f(n)$ está acotada en cierto sentido por $g(n)$ cuando n es muy grande. Si se afirma que la complejidad temporal de un algoritmo es $O(g(n))$, quiere decir que para ejecutar este algoritmo siempre se requiere de menos que c veces $|g(n)|$ a medida que n es suficientemente grande para alguna c.

A continuación se considerará el caso en que para completar un algoritmo se requieren $(n^3 + n)$ pasos. Así,

$$\begin{aligned} f(n) &= n^3 + n \\ &= \left(1 + \frac{1}{n^2}\right) n^3 \\ &\leq 2n^3 \quad \text{para} \quad n \geq 1. \end{aligned}$$

Por consiguiente, puede afirmarse que la complejidad temporal es $O(n^3)$ porque es posible que c y n_0 sean 2 y 1, respectivamente.

A continuación aclararemos una cuestión muy importante, que suele ser una interpretación equivocada sobre el orden de magnitud de la complejidad temporal de los algoritmos.

Suponga que tiene dos algoritmos A_1 y A_2 que resuelven el mismo problema. Sean $O(n^3)$ y $O(n)$ las complejidades temporales de A_1 y A_2, respectivamente. Si a una misma persona se le solicita escribir dos programas para A_1 y A_2 y ejecutarlos en el mismo ambiente de programación, ¿el programa para A_2 correrá más rápido que el programa para A_1? Un error común es pensar que el programa para A_2 siempre se ejecutará más rápido que el programa para A_1. En realidad, esto no necesariamente es cierto por una sencilla razón: puede requerirse más tiempo para ejecutar un paso en A_2 que en A_1. En otras palabras, aunque el número de pasos necesarios para ejecutar A_2 sea menor que los requeridos para A_1, en algunos casos A_1 se ejecuta más rápido que A_2. Suponga que cada paso de A_1 tarda 1/100 del tiempo necesario para cada paso de A_2. Entonces los tiempos de cómputo reales para A_1 y A_2 son n^3 y $100n$, respectivamente. Para $n < 10$, A_1 corre más rápido que A_2. Para $n > 10$, A_2 corre más rápido que A_1.

El lector comprende ahora la importancia de la constante que aparece en la definición de la función $O(g(n))$. No es posible ignorarla. Sin embargo, no influye cuán grande sea la constante, su importancia decrece a medida que n crece. Si las complejidades de A_1 y A_2 son $O(g_1(n))$ y $O(g_2(n))$, respectivamente, y $g_1(n) < g_2(n)$ para toda n, se entiende que a medida que n aumenta lo suficiente, A_1 se ejecuta más rápido que A_2.

Otra cuestión que debe recordarse es que siempre es posible, por lo menos teóricamente, codificar (e implementar) en hardware (hardwire) cualquier algoritmo. Es decir, siempre es posible diseñar un circuito a la medida para implementar un algoritmo. Si dos algoritmos están implementados en hardware, el tiempo necesario para ejecutar un paso de uno de los algoritmos puede igualarse al tiempo necesario que requiere en el otro algoritmo. En tal caso, el orden de magnitud es mucho más importante. Si las complejidades temporales de A_1 y A_2 son $O(n^3)$ y $O(n)$, respectivamente, entonces se sabe que A_2 es mejor que A_1 si ambos están implementados en hardware. Por supuesto, el análisis anterior tiene sentido sólo si se domina a la perfección la habilidad de implementar en hardware los algoritmos.

La importancia del orden de magnitud puede verse al estudiar la tabla 2-1. En esta tabla observamos lo siguiente:

1. Es muy importante si puede encontrarse un algoritmo con menor orden de complejidad temporal. Un caso típico lo constituye la búsqueda. En el peor de los casos, una búsqueda secuencial a través de una lista de n números requiere $O(n)$ operaciones. Si se tiene una lista ordenada de n números, es posible usar búsqueda

TABLA 2-1 Funciones de complejidad temporal.

Función de complejidad temporal	Tamaño del problema (n)			
	10	**10^2**	**10^3**	**10^4**
$\log_2 n$	3.3	6.6	10	13.3
n	10	10^2	10^3	10^4
$n \log_2 n$	0.33×10^2	0.7×10^3	10^4	1.3×10^5
n^2	10^2	10^4	10^6	10^8
2^n	1 024	1.3×10^{30}	$>10^{100}$	$>10^{100}$
$n!$	3×10^6	$>10^{100}$	$>10^{100}$	$>10^{100}$

binaria y la complejidad temporal se reduce a $O(\log_2 n)$, en el peor de los casos. Para $n = 10^4$, la búsqueda secuencial puede requerir 10^4 operaciones, mientras que la búsqueda binaria sólo requiere 14 operaciones.

2. Aunque las funciones de complejidad temporal como n^2, n^3, etc., pueden no ser deseables, siguen siendo tolerables en comparación con una función del tipo 2^n. Por ejemplo, cuando $n = 10^4$, entonces $n^2 = 10^8$, pero $2^n > 10^{100}$. El número 10^{100} es tan grande que no importa cuán rápida sea una computadora, no es capaz de resolver este problema. Cualquier algoritmo con complejidad temporal $O(p(n))$, donde $p(n)$ es una función polinomial, es un algoritmo polinomial. Por otra parte, los algoritmos cuyas complejidades temporales no pueden acotarse con una función polinomial son algoritmos exponenciales.

Hay una gran diferencia entre algoritmos polinomiales y algoritmos exponenciales. Lamentablemente, existe una gran clase de algoritmos exponenciales y no parece haber alguna esperanza de que puedan sustituirse por algoritmos polinomiales. Todo algoritmo para resolver el problema euclidiano del agente viajero, por ejemplo, es un algoritmo exponencial, hasta ahora. De manera semejante, hasta el presente todo algoritmo para resolver el problema de satisfactibilidad, según se define en la sección 8-3, es un algoritmo exponencial. Aunque como se verá, el problema de árbol de expansión mínima, según se define en el apartado 3-1, puede resolverse con algoritmos polinomiales.

El análisis anterior fue muy vago en cuanto a los datos. Ciertamente, para algunos datos un algoritmo puede terminar bastante rápido, pero para otros datos puede tener un comportamiento completamente distinto. Estos temas se abordarán en el siguiente apartado.

2-2 ANÁLISIS DEL MEJOR CASO, PROMEDIO Y PEOR DE LOS ALGORITMOS

Para cualquier algoritmo, se está interesado en su comportamiento en tres situaciones: el mejor caso, el caso promedio y el peor caso. Por lo general el análisis del mejor caso es el más fácil, el análisis del peor caso es el segundo más fácil y el más difícil es el análisis del caso promedio. De hecho, aún hay muchos problemas abiertos que implican el análisis del caso promedio.

Ejemplo 2-1 Ordenamiento por inserción directa

Uno de los métodos de ordenamiento más sencillos es el ordenamiento por inserción directa. Se tiene una secuencia de números $x_1, x_2, \ldots, x_n$. Los números se recorren de izquierda a derecha y se escribe x_i a la izquierda de x_{i-1} si x_i es menor que x_{i-1}. En otras palabras, desplazamos a x_i de manera continua hacia la izquierda hasta que los números a su izquierda sean menores que o iguales a él.

Algoritmo 2-1 □ Ordenamiento por inserción directa

Input: $x_1, x_2, \ldots, x_n$.
Output: La secuencia ordenada de $x_1, x_2, \ldots, x_n$.

```
For j := 2 to n do
Begin
    i := j − 1
    x := x_j
    While x < x_i and i > 0 do
    Begin
        x_{i+1} := x_i
        i := i − 1
    End
    x_{i+1} := x
End
```

Considere la secuencia de entrada 7, 5, 1, 4, 3, 2, 6. El ordenamiento por inserción directa produce la secuencia ordenada siguiente:

7
5, 7
1, 5, 7
1, 4, 5, 7
1, 3, 4, 5, 7
1, 2, 3, 4, 5, 7
1, 2, 3, 4, 5, 6, 7.

En nuestro análisis, como medida de la complejidad temporal del algoritmo se usará el número de intercambios de datos $x := x_j$, $x_{i+1} := x_i$ y $x_{i+1} := x$. En este algoritmo hay dos ciclos: uno exterior (**for**) y otro interior (**while**). Para el ciclo exterior siempre se ejecutan dos operaciones de intercambio de datos; a saber, $x := x_j$ y $x_{i+1} := x$. Debido a que el ciclo interior puede o no ser ejecutado, el número de intercambios de datos realizados para x_i en el ciclo interior se denotará por d_i. Así, evidentemente, el número total de movimientos de datos para el ordenamiento por inserción directa es

$$\begin{aligned} X &= \sum_{i=2}^{n} (2 + d_i) \\ &= 2(n-1) + \sum_{i=2}^{n} d_i \end{aligned}$$

Mejor caso: $\sum_{i=2}^{n} d_i = 0$, $X = 2(n-1) = O(n)$

Esto ocurre cuando los datos de entrada ya están ordenados.

Peor caso: El peor caso ocurre cuando los datos de entrada están ordenados a la inversa. En este caso,

$$\begin{aligned} d_2 &= 1 \\ d_3 &= 2 \\ &\vdots \\ d_n &= n-1. \end{aligned}$$

De este modo,

$$\sum_{i=2}^{n} d_i = \frac{n}{2}(n-1)$$

$$X = 2(n-1) + \frac{n}{2}(n-1) = \frac{1}{2}(n-1)(n+4) = O(n^2).$$

Caso promedio: Cuando se está considerando x_i, ya se han ordenado $(i-1)$ datos. Si x_i es el más grande de todos los i números, entonces el ciclo interior no se ejecuta y dentro de este ciclo interior no hay en absoluto ningún movimiento de datos. Si x_i es el segundo más grande de todos los i números, habrá un intercambio de datos, y así sucesivamente. La probabilidad de que x_i sea el más grande es $1/i$. Esta también es la probabilidad de que x_i sea el j-ésimo más grande para $1 \le j \le i$. En consecuencia, el promedio $(2 + d_i)$ es

$$\frac{2}{i} + \frac{3}{i} + \cdots + \frac{i+1}{i} = \sum_{j=1}^{i} \frac{(j+1)}{i}$$

$$= \frac{i+3}{2}.$$

La complejidad temporal media para el ordenamiento por inserción directa es

$$\sum_{i=2}^{n} \frac{i+3}{2} = \frac{1}{2}\left(\sum_{i=2}^{n} i + \sum_{i=2}^{n} 3\right)$$

$$= \frac{1}{4}(n-1)(n+8) = O(n^2).$$

En resumen, la complejidad temporal del ordenamiento por inserción directa para cada caso es:

Mejor caso: $2(n-1) = O(n)$.

Caso promedio: $\frac{1}{4}(n+8)(n-1) = O(n^2)$.

Peor caso: $\frac{1}{2}(n-1)(n+4) = O(n^2)$.

Ejemplo 2-2 El algoritmo de búsqueda binaria

La búsqueda binaria es un famoso algoritmo de búsqueda. Después de ordenar un conjunto numérico en una secuencia creciente o decreciente, el algoritmo de búsqueda binaria empieza desde la parte media de la secuencia. Si el punto de prueba es igual

al punto medio de la secuencia, finaliza el algoritmo; en caso contrario, dependiendo del resultado de comparar el elemento de prueba y el punto central de la secuencia, de manera recurrente se busca a la izquierda o a la derecha de la secuencia.

Algoritmo 2-2 □ Búsqueda binaria

Input: Un arreglo ordenado $a_1, a_2, \ldots, a_n$, $n > 0$ y X, donde $a_1 \le a_2 \le a_3 \le \cdots \le a_n$.
Output: j si $a_j = X$ y 0 si no existe ninguna j tal que $a_j = X$.

$i := 1$ (* primer elemento *)
$m := n$ (* último elemento *)
While $i \le m$ do
Begin
$j := \left\lfloor \frac{i+m}{2} \right\rfloor$
If $X = a_j$ then output j y parar
If $X < a_j$ then $m := j - 1$
else $i := j + 1$
End
$j := 0$
Output j

El análisis del mejor caso para la búsqueda binaria es más sencillo. En el mejor caso, la búsqueda binaria termina en un solo paso.

El análisis del peor caso también es bastante sencillo. Resulta fácil ver que para completar la búsqueda binaria se requiere cuando mucho de $(\lfloor \log_2 n \rfloor + 1)$ pasos. A lo largo de todo este libro, a menos que se indique lo contrario, log n significa $\log_2 n$.

Para simplificar este análisis, se supondrá que $n = 2^k - 1$.

Para el análisis del caso promedio, se observa que si hay n elementos, entonces hay un elemento para el cual el algoritmo termina exitosamente en un solo paso. Este elemento se localiza en la $\left\lfloor \frac{1+n}{2} \right\rfloor$-ésima posición de la secuencia ordenada. Hay dos elementos que hacen que la búsqueda binaria termine exitosamente después de dos pasos. En general, hay 2^{t-1} elementos que hacen que la búsqueda binaria termine exitosa-

mente después de t pasos, para $t = 1, 2, \ldots, \lfloor \log n \rfloor + 1$. Si X no está en la lista, entonces el algoritmo termina sin éxito después de $\lfloor \log n \rfloor + 1$ pasos. En total, puede decirse que hay $(2n + 1)$ casos distintos: n casos que hacen que la búsqueda termine exitosamente y $(n + 1)$ casos que hacen que la búsqueda termine sin éxito.

Sean $A(n)$ el número medio de comparaciones efectuadas en la búsqueda binaria y $k = \lfloor \log n \rfloor + 1$. Entonces

$$A(n) = \frac{1}{2n+1}\left(\sum_{i=1}^{k} i2^{i-1} + k(n+1) \right).$$

A continuación se demostrará que

$$\sum_{i=1}^{k} i2^{i-1} = 2^k(k-1) + 1. \tag{2-1}$$

La fórmula anterior puede demostrarse por inducción sobre k. La ecuación (2-1) es evidentemente cierta para $k = 1$. Suponga que la ecuación (2-1) se cumple para $k = m$, $m > 1$. Entonces se demostrará que es verdadera para $k = m + 1$. Es decir, suponiendo que la ecuación (2-1) es válida, se demostrará que

$$\sum_{i=1}^{m+1} i2^{i-1} = 2^{m+1}(m+1-1) + 1 = 2^{m+1} \cdot m + 1.$$

Observe que

$$\sum_{i=1}^{m+1} i2^{i-1} = \sum_{i=1}^{m} i2^{i-1} + (m+1)2^{m+1-1}.$$

Al sustituir la ecuación (2-1) en la fórmula anterior se obtiene

$$\begin{aligned} \sum_{i=1}^{m+1} i2^{i-1} &= 2^m(m-1) + 1 + (m+1)2^m \\ &= 2^m \cdot 2m + 1 \\ &= 2^{m+1} \cdot m + 1. \end{aligned}$$

Así, se ha demostrado la validez de la ecuación (2-1). Al usar la ecuación (2-1)

$$A(n) = \frac{1}{2n+1}((k-1)2^k + 1 + k2^k).$$

Cuando n es muy grande, se tiene

$$\begin{aligned} A(n) &\approx \frac{1}{2^{k+1}}(2^k(k-1) + k2^k) \\ &= \frac{(k-1)}{2} + \frac{k}{2} \\ &= k - \frac{1}{}. \end{aligned}$$

En consecuencia, $A(n) < k = O(\log n)$.

Ahora quizás el lector se pregunte si el resultado obtenido es válido para n en general, habiendo partido de la suposición de que $n = 2^k$. Pensemos que $t(n)$ es una función no decreciente, y $t(n) = O(f(n))$ es la complejidad temporal de nuestro algoritmo, obtenida al suponer que $n = 2^k$ y $f(bn) \le c'f(n)$ para una $b \ge 1$ y c' es una constante (esto significa que f es una función continua y que toda función polinomial también es así). Entonces

$t(2^k) \le cf(2^k)$ donde c es una constante

Sea $n' = 2^{k+x}$ para $0 \le x \le 1$

$$\begin{aligned} t(n') &= t(2^{k+x}) \\ &\le t(2^{k+1}) \le cf(2^{k+1}) \\ &= cf(2^{k+x} \cdot 2^{1-x}) \\ &\le cc'f(2^{k+x}) = c''f(n'). \end{aligned}$$

En consecuencia, $t(n') = O(f(n'))$.

El análisis anterior muestra que es posible suponer que $n = 2^k$ para obtener la complejidad temporal de un algoritmo. En el resto del libro, siempre que sea necesario, se supondrá que $n = 2^k$ sin explicar por qué es posible hacer esta suposición.

En resumen, para la búsqueda binaria, se tiene

Mejor caso: $O(1)$.

Caso promedio: $O(\log n)$.

Peor caso: $O(\log n)$.

Ejemplo 2-3 Ordenamiento por selección directa

El ordenamiento por selección directa es tal vez el tipo de ordenamiento más sencillo. No obstante, el análisis de este algoritmo es bastante interesante. El ordenamiento por selección directa puede describirse fácilmente como sigue:

1. Se encuentra el número más pequeño. Se hace que este número más pequeño ocupe la posición a_1 mediante el intercambio de a_1 y dicho número.
2. Se repite el paso anterior con los números restantes. Es decir, se encuentra el segundo número más pequeño y se le coloca en a_2.
3. El proceso continúa hasta que se encuentra el número más grande.

Algoritmo 2-3 □ Ordenamiento por selección directa

Input: $a_1, a_2, \ldots, a_n$.
Output: La secuencia ordenada de $a_1, a_2, \ldots, a_n$.

```
For j := 1 to n − 1 do
Begin
    f := j
    For k := j + 1 to n do
        If a_k < a_f then f := k
    a_j ↔ a_f
End
```

En el algoritmo anterior, para encontrar el número más pequeño de $a_1, a_2, \ldots, a_n$, primeramente se pone una bandera (o pivote) f e inicialmente $f = 1$. Luego a_f se compara con a_2. Si $a_f < a_2$, no se hace nada; en caso contrario, se hace $f = 2$; a_f se compara con a_3, y así sucesivamente.

Resulta evidente que en el ordenamiento por selección directa hay dos operaciones: la comparación de dos elementos y el cambio de la bandera. El número de comparaciones entre dos elementos es un número fijo; a saber, $n(n - 1)/2$. Es decir, sin importar cuáles sean los datos de entrada, siempre es necesario efectuar $n(n - 1)/2$ comparaciones. En consecuencia, para medir la complejidad temporal del ordenamiento por selección directa se escogerá el número de cambios de la bandera.

El cambio de la bandera depende de los datos. Considere $n = 2$. Sólo hay dos permutaciones:

(1, 2)
y (2, 1).

Para la primera permutación, ningún cambio de la bandera es necesario, mientras que para la segunda se requiere un cambio de la bandera.

Sea $f(a_1, a_2,\ldots, a_n)$ que denota el número de cambios de la bandera necesarios para encontrar el número más pequeño para la permutación $a_1, a_2,\ldots, a_n$. La tabla siguiente ilustra el caso para $n = 3$.

a_1,	a_2,	a_3	$f(a_1, a_2, a_3)$
1,	2,	3	0
1,	3,	2	0
2,	1,	3	1
2,	3,	1	1
3,	1,	2	1
3,	2,	1	2

Para determinar $f(a_1, a_2,\ldots, a_n)$ se observa lo siguiente:

1. Si $a_n = 1$, entonces $f(a_1, a_2,\ldots, a_n) = 1 + f(a_1, a_2,\ldots, a_{n-1})$ porque en el último paso debe haber un cambio de la bandera.
2. Si $a_n \neq 1$, entonces $f(a_1, a_2,\ldots, a_n) = f(a_1, a_2,\ldots, a_{n-1})$ porque en el último paso no debe haber ningún cambio de la bandera.

Sea $P_n(k)$ que denota la probabilidad de que una permutación $a_1, a_2,\ldots, a_n$ de $\{1, 2,\ldots, n\}$ requiera k cambios de la bandera para encontrar el número más pequeño. Por ejemplo, $P_3(0) = \frac{2}{6}$, $P_3(1) = \frac{3}{6}$ y $P_3(2) = \frac{1}{6}$. Así que, el número promedio de cambios de bandera para encontrar el número más pequeño es

$$X_n = \sum_{k=0}^{n-1} kP_n(k).$$

El número promedio de cambios de bandera para el ordenamiento por selección directa es

$$A(n) = X_n + A(n - 1).$$

Para encontrar X_n se usarán las siguientes ecuaciones, que ya se analizaron:

$$\begin{aligned} f(a_1, a_2, \ldots, a_n) &= 1 + f(a_1, a_2, \ldots, a_{n-1}) \quad \text{si } a_n = 1 \\ &= f(a_1, a_2, \ldots, a_{n-1}) \quad \text{si } a_n \neq 1. \end{aligned}$$

Con base en las fórmulas anteriores, se tiene

$$P_n(\mathrm{k}) = P(a_n = 1)P_{n-1}(k-1) + P(a_n \neq 1)P_{n-1}(k).$$

Pero $P(a_n = 1) = 1/n$ y $P(a_n \neq 1) = (n-1)/n$. En consecuencia, se tiene

$$P_n(k) = \frac{1}{n} P_{n-1}(k-1) + \frac{n-1}{n} P_{n-1}(k). \tag{2-2}$$

Además, se tiene la siguiente fórmula, relacionada con las condiciones iniciales:

$$\begin{aligned} P_1(k) &= 1 \qquad \text{si } k = 0 \\ &= 0 \qquad \text{si } k \neq 0 \\ P_n(k) &= 0 \qquad \text{si } k < 0, \text{ y si } k = n. \end{aligned} \tag{2-3}$$

Para proporcionar al lector un conocimiento adicional sobre las fórmulas, se observa que

$$P_2(0) = \frac{1}{2}$$

y $P_2(1) = \frac{1}{2}$;

$$P_3(0) = \frac{1}{3} P_2(-1) + \frac{2}{3} P_2(0) = \frac{1}{3} \times 0 + \frac{2}{3} \times \frac{1}{2} = \frac{1}{3}$$

y $P_3(2) = \frac{1}{3} P_2(1) + \frac{2}{3} P_2(2) = \frac{1}{3} \times \frac{1}{2} + \frac{2}{3} \times 0 = \frac{1}{6}$.

A continuación, se demostrará que

$$X_n = \sum_{k=1}^{n-1} kP_n(k) = \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2} = H_n - 1, \tag{2-4}$$

donde H_n es el n-ésimo número armónico.

La ecuación (2-4) puede demostrarse por inducción. Es decir, se quiere demostrar que

$$X_n = \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2} = H_n - 1$$

es trivialmente cierta para $n = 2$, y suponiendo que la ecuación (2-4) se cumple cuando $n = m$ para $m > 2$, se demostrará que la ecuación (2-4) es verdadera para $n = m + 1$. Así, se quiere demostrar que

$$X_{m+1} = H_{m+1} - 1$$

$$X_{m+1} = \sum_{k=1}^{m} kP_{m+1}(k)$$

$$= P_{m+1}(1) + 2P_{m+1}(2) + \cdots + mP_{m+1}(m).$$

Al usar la ecuación (2-2) se tiene

$$X_{m+1} = \frac{1}{m+1} P_m(0) + \frac{m}{m+1} P_m(1) + \frac{2}{m+1} P_m(1) + \frac{2m}{m+1} P_m(2)$$

$$+ \cdots + \frac{m}{m+1} P_m(m-1) + \frac{m^2}{m+1} P_m(m)$$

$$= \frac{1}{m+1} P_m(0) + \frac{m}{m+1} P_m(1) + \frac{1}{m+1} P_m(1) + \frac{1}{m+1} P_m(1)$$

$$+ \frac{2m}{m+1} P_m(2) + \frac{1}{m+1} P_m(2) + \frac{2}{m+1} P_m(2)$$

$$+ \frac{3m}{m+1} P_m(3) + \cdots + \frac{1}{m+1} P_m(m-1) + \frac{m-1}{m+1} P_m(m-1)$$

$$= \frac{1}{m+1} (P_m(0) + P_m(1) + \cdots + P_m(m-1)) + \frac{m+1}{m+1} P_m(1)$$

$$+ \frac{2m+2}{m+1} P_m(2) + \cdots + \frac{(m-1)(m+1)}{m+1} P_m(m-1)$$

$$= \frac{1}{m+1} + (P_m(1) + 2P_m(2) + \cdots + (m-1)P_m(m-1))$$

$$= \frac{1}{m+1} + H_m - 1 \qquad \text{(por hipótesis de inducción)}$$

$$= H_{m+1} - 1.$$

Debido a que la complejidad temporal del ordenamiento por selección directa es

$$A(n) = X_n + A(n-1),$$

se tiene

$$\begin{aligned} A(n) &= H_n - 1 + A(n-1) \\ &= (H_n - 1) + (H_{n-1} - 1) + \cdots + (H_2 - 1) \\ &= \sum_{i=2}^{n} H_i - (n-1) \end{aligned} \tag{2-5}$$

$$\begin{aligned} \sum_{i=1}^{n} H_i &= H_n + H_{n-1} + \cdots + H_1 \\ &= H_n + \left(H_n - \frac{1}{n}\right) + \cdots + \left(H_n - \frac{1}{n} - \frac{1}{n-1} - \cdots - \frac{1}{2}\right) \\ &= nH_n - \left(\frac{n-1}{n} + \frac{n-2}{n-1} + \cdots + \frac{1}{2}\right) \\ &= nH_n - \left(1 - \frac{1}{n} + 1 - \frac{1}{n-1} + \cdots + 1 - \frac{1}{2}\right) \\ &= nH_n - \left(n - 1 - \frac{1}{n} - \frac{1}{n-1} - \cdots - \frac{1}{2}\right) \\ &= nH_n - n + H_n \\ &= (n+1)H_n - n. \end{aligned}$$

En consecuencia,

$$\sum_{i=2}^{n} H_i = (n+1)H_n - H_1 - n. \tag{2-6}$$

Al sustituir la ecuación (2-6) en la ecuación (2-5) se tiene

$$\begin{aligned} A(n) &= (n+1)H_n - H_1 - (n-1) - n \\ &= (n+1)H_n - 2n. \end{aligned}$$

A medida que n es suficientemente grande,

$$A(n) \cong n \log_e n = O(n \log n).$$

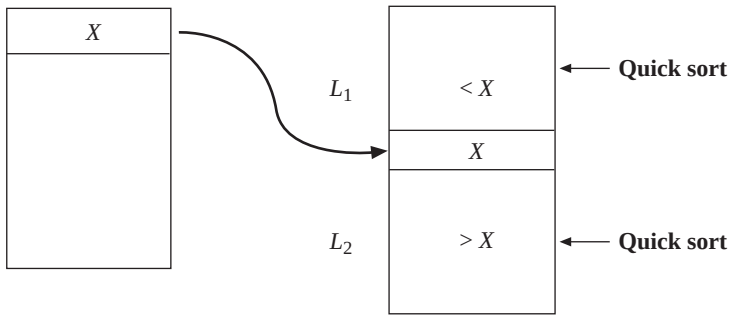
Las complejidades temporales del ordenamiento por selección directa pueden resumirse como sigue:

Mejor caso: $O(1)$.
Peor caso: $O(n \log n)$.
Caso promedio: $O(n^2)$.

Ejemplo 2-4 El algoritmo quick sort

El algoritmo quick sort se basa en la estrategia divide-y-vencerás (divide-and-conquer), que se ilustrará con todo detalle en otro momento. Por ahora, es posible afirmar que esta estrategia divide un problema en dos subproblemas, que se resuelven de manera individual e independiente. Los resultados se unen después. Al aplicar la estrategia divide-y-vencerás al ordenamiento se obtiene un método para clasificar denominado quick sort. Dado un conjunto de números $a_1, a_2, \ldots, a_n$, se escoge un elemento X para dividir dicho conjunto en dos listas, como se muestra en la figura 2-1.

FIGURA 2-1 Quick sort.



Después de la división, el quick sort puede aplicarse en forma recursiva tanto a L_1 como a L_2, con lo cual se obtiene una lista ordenada, ya que L_1 contiene a todos los a_i menores que o iguales a X y L_2 contiene a todos los a_i mayores que X.

El problema es: ¿cómo dividir la lista original? Ciertamente, no debe usarse X para recorrer toda la lista y decidir si un a_i es menor que o igual a X. Hacer lo anterior provoca una gran cantidad de intercambio de datos. Como se muestra en nuestro algoritmo, se utilizan dos apuntadores que se mueven al centro y realizan intercambios de datos según sea necesario.

Algoritmo 2-4 □ Quick sort (*f, l*)

Input: $a_f, a_{f+1},\ldots, a_l$.
Output: La secuencia ordenada de $a_f, a_{f+1},\ldots, a_l$.

```
If f ≥ l then Return
X := a_f
i := f + 1
j := l
While i < j do
Begin
    While a_j ≥ X y j ≥ f + 1 do
        j := j − 1
    While a_i ≤ X  y i ≤ f + l do
        i := i + 1
    Si i < j entonces a_i ↔ a_j
End
Quick sort(f, j − 1)
Quick sort(j + 1, l)
```

El siguiente ejemplo ilustra la característica más importante del algoritmo quick sort:

$X=3$	a_1	a_2	a_3	a_4	a_5	a_6
$a_j=a_6<X$	3	6	1	4	5	2
	$\uparrow i$					$\uparrow j$
	2	6	1	4	5	3
	$\uparrow i$					$\uparrow j$
$a_i=a_2>X$	2	6	1	4	5	3
		$\uparrow i$				$\uparrow j$
	2	3	1	4	5	6
		$\uparrow i$				$\uparrow j$
	2	3	1	4	5	6
		$\uparrow i$			$\uparrow j$	
	2	3	1	4	5	6
		$\uparrow i$		$\uparrow j$		

$a_j = a_3 < X$

2	3	1	4	5	6
	$\uparrow i$	$\uparrow j$			
2	1	3	4	5	6
	$\uparrow i$	$\uparrow j$			
2	1	3	4	5	6
		$i \uparrow \uparrow j$			
$\lvert\leftarrow \leq 3 \rightarrow\rvert$		$=3$		$\lvert\leftarrow \geq 3 \rightarrow\rvert$	

El mejor caso de quick sort ocurre cuando *X* divide la lista justo en el centro. Es decir, *X* produce dos sublistas que contienen el mismo número de elementos. En este caso, la primera ronda requiere *n* pasos para dividir la lista original en dos listas. Para la ronda siguiente, para cada sublista, de nuevo se necesitan *n*/2 pasos (ignorando el elemento usado para la división). En consecuencia, para la segunda ronda nuevamente se requieren $2 \cdot (n/2) = n$ pasos. Si se supone que $n = 2^p$, entonces en total se requieren *pn* pasos. Sin embargo, $p = \log_2 n$. Así, para el mejor caso, la complejidad temporal del quick sort es $O(n \log n)$.

El peor caso del quick sort ocurre cuando los datos de entrada están ya ordenados o inversamente ordenados. En estos casos, todo el tiempo simplemente se está seleccionando el extremo (ya sea el mayor o el menor). Por lo tanto, el número total de pasos que se requiere en el quick sort para el peor caso es

$$n + (n - 1) + \cdots + 1 = \frac{n}{2}(n + 1) = O(n^2).$$

Para analizar el caso promedio, sea $T(n)$ que denota el número de pasos necesarios para llevar a cabo el quick sort en el caso promedio para *n* elementos. Se supondrá que después de la operación de división la lista se ha dividido en dos sublistas. La primera de ellas contiene *s* elementos y la segunda contiene $(n - s)$ elementos. El valor de *s* varía desde 1 hasta *n* y es necesario tomar en consideración todos los casos posibles a fin de obtener el desempeño del caso promedio. Para obtener $T(n)$ es posible aplicar la siguiente fórmula:

$$T(n) = \underset{1 \leq s \leq n}{\text{Promedio}}(T(s) + T(n - s)) + cn$$

donde *cn* denota el número de operaciones necesario para efectuar la primera operación de división. (Observe que cada elemento es analizado antes de dividir en dos sublistas la lista original.)

$$\begin{aligned}
&\underset{1\le s\le n}{\text{Promedio}}\,(T(s)+T(n-s))\\
&=\frac{1}{n}\sum_{s=1}^{n}\,(T(s)+T(n-s))\\
&=\frac{1}{n}\,(T(1)+T(n-1)+T(2)+T(n-2)+\cdots+T(n)+T(0)).
\end{aligned}$$

Debido a que $T(0)=0$,

$$T(n)=\frac{1}{n}\,(2T(1)+2T(2)+\cdots+2T(n-1)+T(n))+cn$$

o, $(n-1)T(n)=2T(1)+2T(2)+\cdots+2T(n-1)+cn^2$.

Al sustituir $n=n-1$ en la fórmula anterior, se tiene

$$(n-2)T(n-1)=2T(1)+2T(2)+\cdots+2T(n-2)+c(n-1)^2.$$

En consecuencia,

$$\begin{aligned}
&(n-1)T(n)-(n-2)T(n-1)=2T(n-1)+c(2n-1)\\
&(n-1)T(n)-nT(n-1)=c(2n-1)\\
&\frac{T(n)}{n}=\frac{T(n-1)}{n-1}+c\left(\frac{1}{n}+\frac{1}{n-1}\right).
\end{aligned}$$

En forma recursiva,

$$\begin{aligned}
&\frac{T(n-1)}{n-1}=\frac{T(n-2)}{n-2}+c\left(\frac{1}{n-1}+\frac{1}{n-2}\right)\\
&\quad\vdots\\
&\frac{T(2)}{2}=\frac{T(1)}{1}+c\left(\frac{1}{2}+\frac{1}{1}\right).
\end{aligned}$$

Se tiene

$$\frac{T(n)}{n} = c\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2}\right) + c\left(\frac{1}{n-1} + \frac{1}{n-2} + \cdots + 1\right)$$

$$= c(H_n - 1) + cH_{n-1}$$

$$= c(H_n + H_{n-1} - 1)$$

$$= c\left(2H_n - \frac{1}{n} - 1\right)$$

$$= c\left(2H_n - \frac{n+1}{n}\right).$$

Finalmente, se tiene

$$T(n) = 2cnH_n - c(n+1)$$
$$\cong 2cn \log_e n - c(n+1)$$
$$= O(n \log n).$$

En conclusión, las complejidades temporales para el quick sort son

Mejor caso: $O(n \log n)$.
Caso promedio: $O(n \log n)$.
Peor caso: $O(n^2)$.

Ejemplo 2-5 El problema de encontrar rangos

Suponga que se tiene un conjunto de números a_1, a_2,…, a_n. Se dice que a_i domina a a_j si $a_i > a_j$. Si se quiere determinar la cantidad de a_j dominados por un número a_i, entonces simplemente estos números pueden ordenarse en una secuencia, con lo cual el problema se resuelve de inmediato.

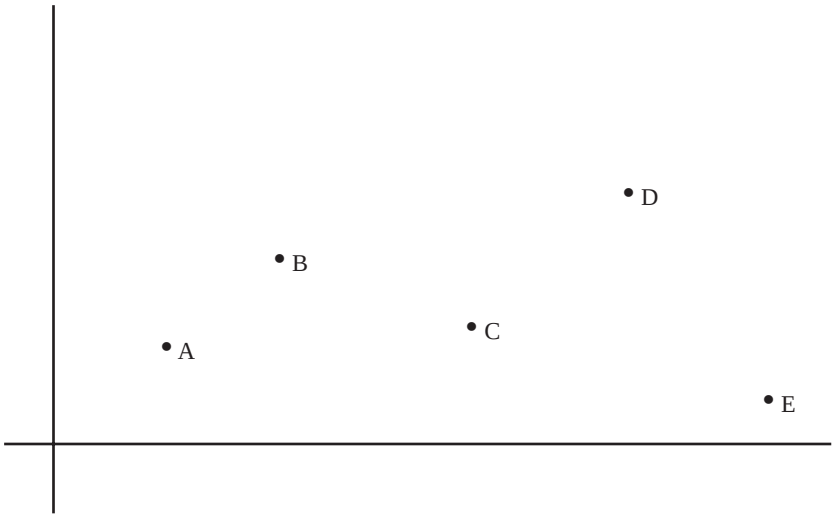
A continuación, este problema se extiende al caso de dos dimensiones. Es decir, cada dato es un punto en el plano. En este caso, primero se define lo que se entiende por dominancia de puntos en el espacio bidimensional.

Definición

Dados dos puntos $A = (a_1, a_2)$ y $B = (b_1, b_2)$, A domina a B si y sólo si $a_i > b_i$ para $i = 1, 2$. Si ocurre que A no domina a B ni B domina a A, entonces A y B no pueden compararse.

Considere la figura 2-2.

FIGURA 2-2 Un caso para ilustrar la relación de dominancia.



Para los puntos de la figura 2-2 se tiene la siguiente relación:

1. *B*, *C* y *D* dominan a *A*.
2. *D* domina a *A*, *B* y *C*.
3. No es posible comparar ninguno de los demás pares de puntos.

Una vez que se ha definido la relación de dominancia, es posible definir el rango de un punto.

Definición

Dado un conjunto *S* de *n* puntos, el rango de un punto *X* es el número de puntos dominados por *X*.

Para los puntos en la figura 2-2, los rangos de *A* y *E* son cero porque no dominan a ningún punto. Los rangos de *B* y *C* son uno porque *A*, y sólo *A*, es domina-

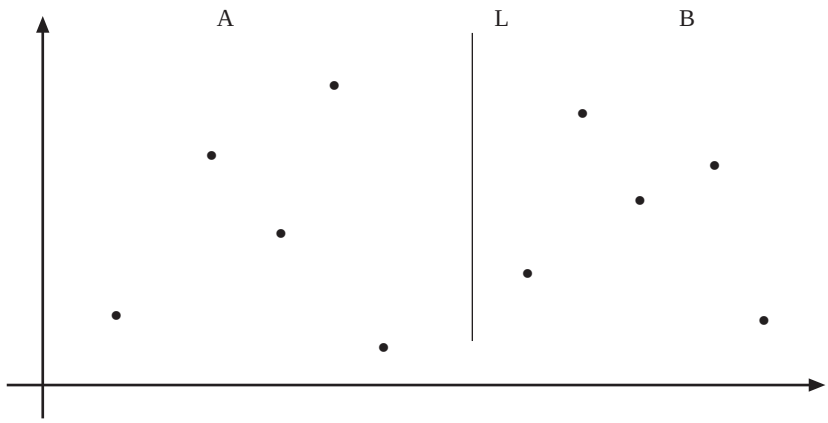
do por ellos. El rango de *D* es tres porque los tres puntos *A*, *B* y *C* son dominados por *D*.

El problema consiste en encontrar el rango de cada punto.

Una forma directa de resolver este problema es realizar una comparación exhaustiva de todos los pares de puntos. La complejidad temporal de este método es $O(n^2)$. A continuación se demostrará que este problema puede resolverse en $O(n \log_2 n^2)$ en el peor caso.

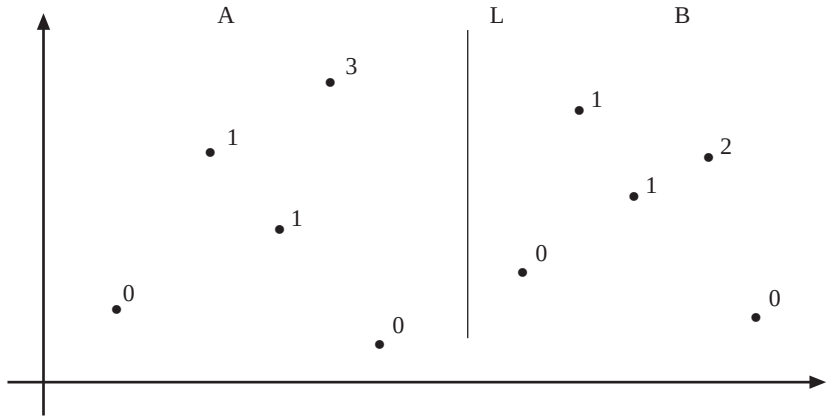
Considere la figura 2-3. El primer paso consiste en encontrar una línea recta *L* perpendicular al eje *x* que separe al conjunto de puntos en dos subconjuntos del mismo tamaño. Sean *A*, que denota el subconjunto a la izquierda de *L*, y *B*, que denota el subconjunto a la derecha de *L*. Resulta evidente que el rango de cualquier punto en *A* no es afectado por la presencia de *B*. Sin embargo, el rango de un punto en *B* puede ser afectado por la presencia de *A*.

FIGURA 2-3 El primer paso para resolver el problema de determinación del rango.



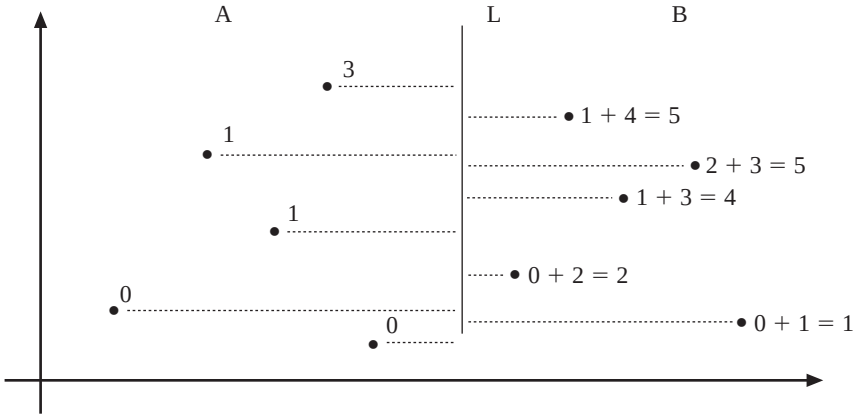
Supongamos que encontramos los rangos locales de los puntos de *A* y *B* por separado. Es decir, encontramos los rangos de los puntos en *A* (sin tener en cuenta a *B*) y los rangos de los puntos en *B* (sin considerar a *A*). A continuación representamos los rangos locales de puntos de la figura 2-3 en la figura 2-4:

FIGURA 2-4 Los rangos locales de los puntos *A* y *B*.



Luego se proyectan todos los puntos sobre *L*. Observe que un punto P_1 en *B* domina a un punto P_2 en *A* si y sólo si el valor *y* de P_1 es mayor que el de P_2. Sea *P* un punto en *B*. El rango de *P* es el rango de *P*, entre los puntos en *B*, más el número de puntos en *A* cuyos valores *y* son más pequeños que el valor *y* de *P*. Esta modificación se ilustra en la figura 2-5.

FIGURA 2-5 Modificación de rangos.



El siguiente algoritmo determina el rango de cualquier punto en un plano.

Algoritmo 2-5 □ Algoritmo para encontrar un rango

Input: Un conjunto S de puntos en el plano $P_1, P_2, \ldots, P_n$.

Output: El rango de todos los puntos en S.

Paso 1. Si S contiene un solo punto, devuelve su rango como 0. En caso contrario, escoge una línea de corte L perpendicular al eje x de modo que el valor X de $n/2$ puntos de S sea menor que L (este conjunto de puntos se denomina A) y que el valor X de los demás puntos sea mayor que L (este conjunto de puntos se denomina B). Observe que L es el valor X mediana (medida central estadística) de este conjunto.

Paso 2. En forma recurrente, este algoritmo para encontrar un rango se usa para encontrar los rangos de los puntos en A y los rangos de los puntos en B.

Paso 3. Los puntos en A y en B se clasifican según sus valores y. Estos puntos se analizan secuencialmente y se determina, para cada punto en B, el número de puntos en A cuyos valores y son menores que su valor y. El rango de este punto es igual al rango de este punto entre los puntos en B (lo cual se determinó en el paso 2), más el número de puntos en A cuyos valores y son menores que su valor y.

El algoritmo 2-5 es recursivo. Se basa en la estrategia divide-y-vencerás, que separa un problema en dos subproblemas, resuelve de manera independiente estos dos subproblemas y después une las subsoluciones en la solución final. La complejidad temporal de este algoritmo depende de las complejidades temporales de los pasos siguientes:

1. En el paso 1 hay una operación con la que se encuentra la mediana de un conjunto de números. Más adelante, en el capítulo 7, se demostrará que la menor complejidad temporal de cualquier algoritmo para encontrar la mediana es $O(n)$.
2. En el paso 3 hay una operación de ordenamiento. En la sección 2-3 de este capítulo se demostrará que la menor complejidad temporal de cualquier algoritmo de ordenamiento es $O(n \log n)$. La inspección se lleva a cabo en $O(n)$ pasos.

Sea $T(n)$ que denota el tiempo total necesario para completar el algoritmo para encontrar rangos. Entonces

$$T(n) \leq 2T(n/2) + c_1 n \log n + c_2 n$$
$$\leq 2T(n/2) + cn \log n$$

donde c_1, c_2 y c son constantes para un n suficientemente grande.

Sea $n = 2^p$. Entonces

$$
\begin{aligned}
T(n) &\le 2T(n/2) + cn \log n \\
&\le 2(2T(n/4) + cn \log (n/2)/2) + cn \log n \\
&= 4T(n/4) + c(n \log (n/2) + n \log n) \\
&\quad \vdots \\
&\le nT(1) + c(n \log n + n \log (n/2) + n \log (n/4) + \cdots + n \log 2) \\
&= nT(1) + cn\left(\frac{1+\log n}{2}\log n\right) \\
&= c_3 n + \frac{c}{2} n \log^2 n + \frac{c}{2} n \log n.
\end{aligned}
$$

En consecuencia, $T(n) = O(n \log^2 n)$.

La complejidad temporal anterior es para el peor caso y para el caso promedio, ya que la complejidad temporal mínima $O(n \log n)$ para clasificar también es válida para el caso promedio. De manera semejante, la complejidad temporal de $O(n)$ para encontrar la mediana también es válida para el caso promedio.

Observe que este algoritmo para encontrar el rango de cada punto es mucho mejor que un algoritmo en que se utilice una búsqueda exhaustiva. Si ésta se lleva a cabo en todos los pares, entonces para completar el proceso se requieren $O(n^2)$ pasos.

2-3 LA COTA INFERIOR DE UN PROBLEMA

En el apartado anterior, muchos ejemplos de algoritmos para encontrar complejidades temporales nos enseñaron la forma de determinar la eficiencia de un algoritmo. En esta sección, el problema de la complejidad temporal se abordará desde un punto de vista completamente distinto.

Considere el problema de encontrar rangos, o el problema del agente viajero. Ahora se pregunta: ¿cómo medir la dificultad de un problema?

Esta pregunta puede responderse de una forma muy intuitiva. Si un problema puede resolverse con un algoritmo con baja complejidad temporal, entonces el problema es sencillo; en caso contrario, se trata de un problema difícil. Esta definición intuitiva conduce al concepto de cota inferior de un problema.

Definición

La cota inferior de un problema es la complejidad temporal mínima requerida por cualquier algoritmo que pueda aplicarse para resolverlo.

Para describir la cota inferior se usará la notación Ω, que se define como sigue:

Definición

$f(n) = \Omega(g(n))$ si y sólo si existen constantes positivas c y n_0 tales que para toda $n > n_0$, $|f(n)| \geq c|g(n)|$.

La complejidad temporal usada en la definición anterior se refiere a la complejidad temporal del peor caso, aunque también puede usarse la complejidad temporal del caso promedio. Si se utiliza la complejidad temporal del caso promedio, la cota inferior se denomina cota inferior del caso promedio; en caso contrario, se denomina cota inferior del peor caso. En todo este libro, a menos que se indique otra cosa, cuando se mencione una cota inferior se entiende que se trata de la cota inferior del peor caso.

Por la definición, casi parece necesario enumerar todos los algoritmos posibles, determinar la complejidad temporal de cada algoritmo y encontrar la complejidad temporal mínima. Por supuesto, lo anterior es utópico porque sencillamente no pueden enumerarse todos los algoritmos posibles; nunca se tiene la certeza de que no vaya a inventarse un nuevo algoritmo que pueda producir una mejor cota inferior.

El lector debe observar que una cota inferior, por definición, no es única. Una cota inferior famosa es la cota inferior para el problema de ordenamiento, que es igual a $\Omega(n \log n)$. Imagine que antes de encontrar esta cota inferior, alguien puede demostrar que una cota inferior para el ordenamiento es $\Omega(n)$ porque cada dato debe examinarse antes de terminar el ordenamiento. De hecho, es posible ser más radical. Por ejemplo, antes de sugerir $\Omega(n)$ como una cota inferior para ordenar, se podría sugerir $\Omega(1)$ como la cota inferior porque para terminar todo algoritmo realiza al menos un paso.

Con base en el análisis anterior, se sabe que puede haber tres cotas inferiores: $\Omega(n \log n)$, $\Omega(n)$ y $\Omega(1)$ para ordenar. Todas son correctas aunque, evidentemente, la única importante es $\Omega(n \log n)$. Las otras dos son cotas inferiores triviales. Debido a que una cota inferior es trivial si es demasiado baja, es deseable que la cota inferior sea lo más alta posible. La búsqueda de cotas inferiores siempre parte de una cota inferior bastante baja sugerida por un investigador. Luego, alguien mejoraría esta cota inferior al encontrar una cota inferior más alta. Esta cota inferior más alta sustituye a la anterior y se convierte en la cota inferior significativa de este problema. Esta situación prevalece hasta que se encuentra una cota inferior aún más alta.

Es necesario entender que cada cota inferior más alta se encuentra mediante un análisis teórico, no por pura suposición. A medida que la cota inferior aumenta, inevitablemente surge la pregunta de si existe algún límite para la cota inferior. Considere, por ejemplo, la cota inferior para ordenar. ¿Hay alguna posibilidad de sugerir que $\Omega(n^2)$ sea una cota inferior para clasificar? No, porque hay un algoritmo para ordenar,

digamos el algoritmo de ordenamiento heap sort, cuya complejidad temporal en el peor caso es $O(n \log n)$. En consecuencia, se sabe que la cota inferior para clasificar es a lo sumo $\Omega(n \log n)$.

A continuación se considerarán los dos casos siguientes:

Caso 1. En el presente se encuentra que la máxima cota inferior de un problema es $\Omega(n \log n)$ y que la complejidad temporal del mejor algoritmo disponible para resolver este problema es $O(n^2)$.

En este caso hay tres posibilidades:

1. La cota inferior del problema es demasiado baja. En consecuencia, es necesario tratar de encontrar una cota inferior más precisa, o alta. En otras palabras, es necesario intentar mover hacia arriba la cota inferior.
2. El mejor algoritmo disponible no es suficientemente bueno. Por lo tanto, es necesario tratar de encontrar un mejor algoritmo cuya complejidad temporal sea más baja. Dicho de otra manera, es necesario intentar mover hacia abajo la mejor complejidad temporal.
3. La cota inferior puede mejorarse y también es posible mejorar el algoritmo. Por consiguiente, es necesario tratar de mejorar ambos.

Caso 2. La cota inferior actual es $\Omega(n \log n)$ y se requiere un algoritmo cuya complejidad temporal sea $O(n \log n)$.

En este caso se dice que ya no es posible mejorar más ni la cota inferior ni el algoritmo. En otras palabras, se ha encontrado un algoritmo óptimo para resolver el problema, así como una cota inferior verdaderamente significativa para el problema.

A continuación se recalcará la cuestión anterior. Observe que desde el inicio de este capítulo se planteó una pregunta: ¿cómo se sabe que un algoritmo es el óptimo? Ahora ya se tiene la respuesta. *Un algoritmo es el óptimo si su complejidad temporal es igual a una cota inferior de este problema. Ya no es posible mejorar más ni la cota inferior ni el algoritmo.*

A continuación se presenta un resumen de algunos conceptos importantes concernientes a las cotas inferiores:

1. La cota inferior de un problema es la complejidad temporal mínima de todos los algoritmos que puede aplicarse para resolverlo.
2. La mejor cota inferior es la más alta.
3. Si la cota inferior conocida actual de un problema es más baja que la complejidad temporal del mejor algoritmo disponible para resolver este problema, entonces es posible mejorar la cota inferior moviéndola hacia arriba. El algoritmo puede mejorarse moviendo hacia abajo su complejidad temporal, o ambas cosas.

4. Si la cota inferior conocida actual es igual a la complejidad temporal de un algoritmo disponible, entonces ya no es posible mejorar más ni el algoritmo ni la cota inferior. El algoritmo es un algoritmo óptimo y la cota inferior es la máxima cota inferior, que es la verdaderamente importante.

En las siguientes secciones se analizarán algunos métodos para encontrar cotas inferiores.

2-4 LA COTA INFERIOR DEL PEOR CASO DEL ORDENAMIENTO

Para muchos algoritmos, la ejecución del algoritmo puede describirse con árboles binarios. Por ejemplo, considere el caso del ordenamiento por inserción directa. Se supondrá que el número de entradas es tres y que todos los datos son diferentes. En este caso hay seis permutaciones distintas:

a_1	a_2	a_3
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

Una vez que se aplica el ordenamiento por inserción directa al conjunto de datos anteriores, cada permutación evoca una respuesta distinta. Por ejemplo, suponga que la entrada es (2, 3, 1). El ordenamiento por inserción directa se comporta ahora como sigue:

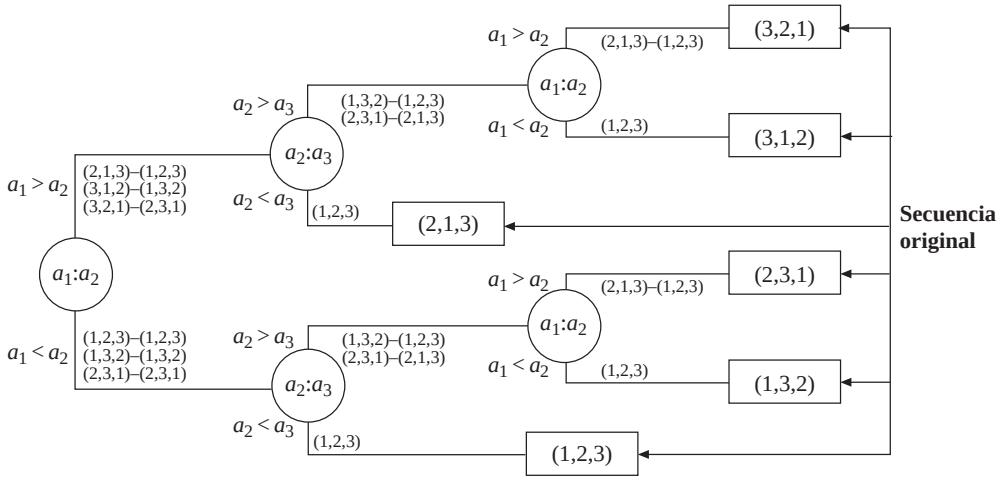
1. $a_1 = 2$ se compara con $a_2 = 3$. Debido a que $a_2 > a_1$, no se realiza ningún cambio de datos.
2. $a_2 = 3$ se compara con $a_3 = 1$. Debido a que $a_2 > a_3$, se intercambian a_2 y a_3. Es decir, $a_2 = 1$ y $a_3 = 3$.
3. $a_1 = 2$ se compara con $a_3 = 1$. Debido a que $a_1 > a_2$, se intercambian a_1 y a_2. Es decir, $a_1 = 1$ y $a_2 = 2$.

Si los datos de entrada son (2, 1, 3), entonces el algoritmo se comporta como sigue:

1. $a_1 = 2$ se compara con $a_2 = 1$. Debido a que $a_1 > a_2$, se intercambian a_1 y a_2. Es decir, $a_1 = 1$ y $a_2 = 2$.
2. $a_2 = 2$ se compara con $a_3 = 3$. Debido a que $a_2 < a_3$, no se realiza ningún cambio de datos.

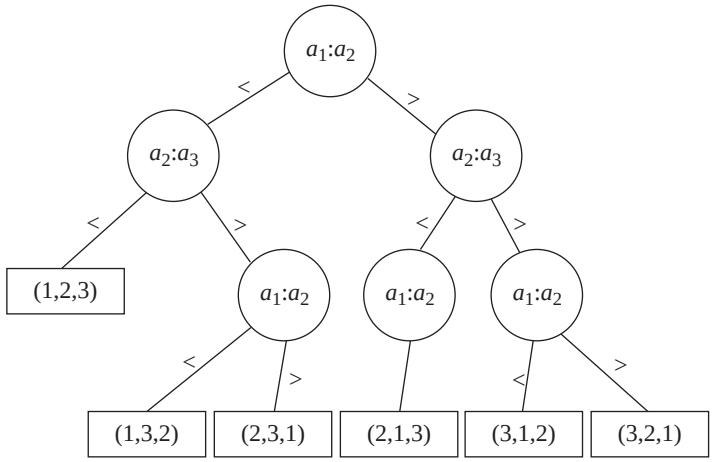
En la figura 2-6 se muestra la forma en que es posible describir el ordenamiento por inserción directa mediante un árbol binario cuando se clasifican tres datos. Este árbol binario puede modificarse fácilmente para manipular el caso en que hay cuatro datos. Es fácil ver que el ordenamiento por inserción directa, cuando se aplica a cualquier número de datos, puede describirse mediante un árbol de decisión binario.

FIGURA 2-6 Ordenamiento por inserción directa con tres elementos representados por un árbol.



En general, cualquier algoritmo de ordenamiento cuya operación básica sea una operación de comparación e intercambio puede describirse mediante un árbol de decisión binario. En la figura 2-7 se muestra cómo es posible describir el ordenamiento por el método de la burbuja (bubble sort) mediante un árbol de decisión binario. En la figura 2-7 se supone que hay tres datos distintos entre sí. Esta figura también está bastante simplificada para que no ocupe mucho espacio. Aquí no se analizará el ordenamiento por el método de la burbuja, ya que es bastante conocido.

FIGURA 2-7 Árbol de decisión binaria que describe el ordenamiento por el método de la burbuja.



La acción de un algoritmo de ordenamiento basado en operaciones de comparación e intercambio sobre un conjunto de datos de entrada particular corresponde a una ruta que va de la raíz del árbol a un nodo hoja. En consecuencia, cada nodo hoja corresponde a una permutación particular. *La ruta más larga que va de la raíz del árbol a un nodo hoja, que se denomina profundidad (altura) del árbol, representa la complejidad temporal del peor caso de este algoritmo*. Para encontrar la cota inferior del problema de ordenamiento es necesario encontrar la profundidad más pequeña de algún árbol de entre todos los algoritmos de ordenamiento de modelado de árboles de decisión binarios posibles.

A continuación se mencionan algunas cuestiones importantes:

1. Para todo algoritmo de ordenamiento, su árbol de decisión binario correspondiente tendrá $n!$ nodos hoja a medida que haya $n!$ permutaciones distintas.
2. La profundidad de un árbol binario con un número fijo de nodos hoja será mínima si el árbol está balanceado.
3. Cuando un árbol binario está balanceado, la profundidad del árbol es $\lceil \log X \rceil$, donde X es el número de nodos hoja.

A partir del razonamiento anterior, es posible concluir fácilmente que *una cota inferior del problema de ordenamiento es* $\lceil \log n! \rceil$. Es decir, el número de comparaciones necesarias para ordenar en el peor caso es por lo menos $\lceil \log n! \rceil$.

Quizás este $\log n!$ sea algo misterioso para muchos. Simplemente se ignora cuán grande es este número; a continuación se analizarán dos métodos de aproximación a $\log n!$

Método 1.

Se usa el hecho de que

$$\begin{aligned}
\log n! &= \log(n(n-1)\ldots 1) \\
&= \sum_{i=1}^{n} \log i \\
&= (2-1)\log 2 + (3-2)\log 3 + \cdots + (n-n+1)\log n \\
&> \int_1^n \log x dx \\
&= \log e \int_1^n \log_e x dx \\
&= \log e[x \log_e x - x]_1^n \\
&= \log e(n \log_e n - n + 1) \\
&= n \log n - n \log e + 1.44 \\
&\geq n \log n - 1.44n \\
&= n \log n\left(1 - \frac{1.44}{\log n}\right).
\end{aligned}$$

Si se hace $n = 2^2$, $n \log n\left(1 - \frac{1.44}{2}\right) = 0.28n \log n$.

Así, al hacer $n_0 = 2^2$ y $c = 0.28$, se tiene

$$\log n! \geq cn \log n \qquad \text{para } n \geq n_0.$$

Es decir, una cota inferior para el peor caso de ordenamiento es $\Omega(n \log n)$.

Método 2. Aproximación de Stirling

Con la aproximación de Stirling se aproxima el valor $n!$ a medida que n es muy grande por medio de la siguiente fórmula:

$$n! \cong \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

Esta fórmula puede encontrarse en casi cualquier libro de cálculo avanzado. La tabla 2-2 ilustra qué tan bien la aproximación de Stirling se acerca a $n!$ En la tabla, S_n será la aproximación de Stirling.

TABLA 2-2 Algunos valores de la aproximación de Stirling.

n	$n!$	S_n
1	1	0.922
2	2	1.919
3	6	5.825
4	24	23.447
5	120	118.02
6	720	707.39
10	3 628 800	3 598 600
20	2.433×10^{18}	2.423×10^{18}
100	9.333×10^{157}	9.328×10^{157}

Al usar la aproximación de Stirling se tiene

$$\log n! = \log \sqrt{2\pi} + \frac{1}{2} \log n + n \log \frac{n}{e}$$

$$= \log \sqrt{2\pi} + \frac{1}{2} \log n + n \log n - n \log e$$

$$\geq n \log n - 1.44n.$$

Con base en ambos métodos es posible afirmar que el número mínimo de comparaciones requerido por el ordenamiento es $\Omega(n \log n)$, en el peor caso.

En este instante es necesario observar que la afirmación anterior no significa que no es posible encontrar una cota inferior. En otras palabras, es posible que algún descubrimiento nuevo pudiera dar a conocer que la cota inferior del ordenamiento es en realidad más alta. Por ejemplo, es posible que alguien pudiera descubrir que la cota inferior del ordenamiento fuera $\Omega(n^2)$.

En la siguiente sección se presentará un algoritmo de ordenamiento cuya complejidad temporal para el peor caso es igual a la cota inferior que acaba de deducirse. Debido a la existencia de tal algoritmo, ya no es posible hacer más alta esta cota inferior.

2-5 ORDENAMIENTO HEAP SORT: UN ALGORITMO DE ORDENAMIENTO ÓPTIMO EN EL PEOR CASO

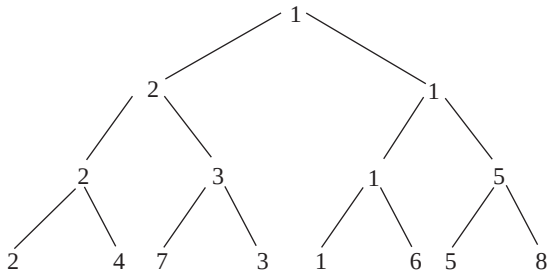
El ordenamiento heap sort forma parte de los algoritmos de ordenamiento cuyas complejidades temporales son $O(n \log n)$. Antes de presentar este tipo de ordenamiento, se analizará el ordenamiento por selección directa a fin de ver por qué no es óptimo en el peor caso. En el ordenamiento por selección directa se requieren $(n - 1)$ pasos para obtener el número más pequeño, y luego, $(n - 2)$ pasos para obtener el segundo núme-

ro más pequeño, y así sucesivamente (siempre en los peores casos). Por consiguiente, en el peor caso, para el ordenamiento por selección directa se requieren $O(n^2)$ pasos. Si el ordenamiento por selección directa se analiza con mayor detenimiento, se observa que cuando se trata de encontrar el segundo número más pequeño, la información que se obtuvo al encontrar el primer número más pequeño no se usa en absoluto. Por eso el ordenamiento por selección directa se comporta de manera tan torpe.

A continuación se considerará otro algoritmo de ordenamiento, denominado knockout sort, que es mucho mejor que el ordenamiento por selección directa. Este ordenamiento es semejante al ordenamiento por selección directa en el sentido de que encuentra el número más pequeño, el segundo más pequeño, y así sucesivamente. No obstante, mantiene cierta información después de encontrar el primer número más pequeño, por lo que es bastante eficiente para encontrar el segundo número más pequeño.

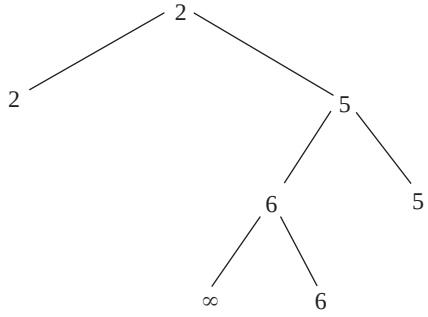
Considere la secuencia de entrada 2, 4, 7, 3, 1, 6, 5, 8. Es posible construir un árbol de knockout sort para encontrar el segundo número más pequeño, como se muestra en la figura 2-8.

FIGURA 2-8 Árbol de knockout sort para encontrar el número más pequeño.



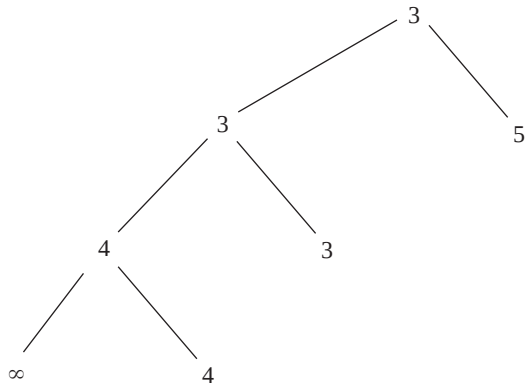
Una vez que se encuentra el número más pequeño, es posible comenzar a buscar el segundo número más pequeño al sustituir 1 por ∞. Así, sólo es necesario analizar una pequeña porción del árbol de knockout sort, como se muestra en la figura 2-9.

FIGURA 2-9 Determinación del segundo número más pequeño.



Cada vez que se encuentra un número más pequeño, se sustituye por ∞ y así es más fácil encontrar el siguiente número más pequeño. Por ejemplo, ahora ya se han encontrado los dos primeros números más pequeños. Luego, el tercer número más pequeño puede encontrarse como se muestra en la figura 2-10.

FIGURA 2-10 Determinación del tercer número más pequeño con ordenamiento por knockout sort.



Para encontrar la complejidad temporal del ordenamiento por knockout sort:

El primer número más pequeño se encuentra después de $(n - 1)$ comparaciones. Para todas las demás selecciones, sólo se requieren $\lceil \log n \rceil - 1$ comparaciones. En consecuencia, el número total de comparaciones es

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

Así, la complejidad temporal del ordenamiento por knockout sort es $O(n \log n)$, que es igual a la cota inferior que se encontró en el apartado 2-4. El ordenamiento por knockout sort es, por consiguiente, un algoritmo de ordenamiento óptimo. Debe observarse que la complejidad temporal $O(n \log n)$ es válida para los casos mejor, promedio y peor.

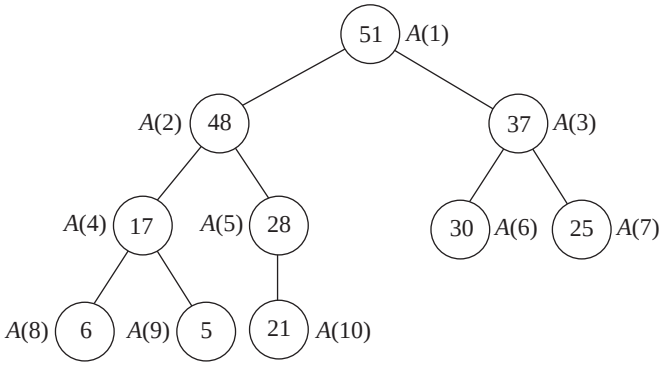
El ordenamiento por knockout sort es mejor que el ordenamiento por inserción directa porque usa información previa. Desafortunadamente, el árbol de knockout sort requiere espacio adicional. Para implementar el ordenamiento por knockout sort se requieren aproximadamente $2n$ posiciones. Este ordenamiento puede mejorarse por el ordenamiento heap sort, el cual se abordará en el resto de esta sección.

De manera semejante al ordenamiento por knockout sort, para almacenar los datos en el ordenamiento heap sort se utiliza una estructura de datos especial. Esta estructura se denomina heap. Un heap es un árbol binario que cumple las siguientes condiciones:

1. El árbol está completamente balanceado.
2. Si la altura del árbol binario es h, entonces las hojas pueden estar al nivel h o al nivel $h - 1$.
3. Todas las hojas al nivel h están a la izquierda tanto como sea posible.
4. Los datos asociados con todos los descendientes de un nodo son menores que el dato asociado con este nodo.

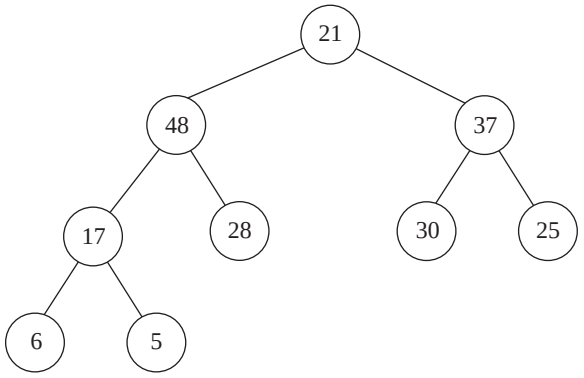
En la figura 2-11 se muestra un heap para 10 números.

FIGURA 2-11 Un heap.



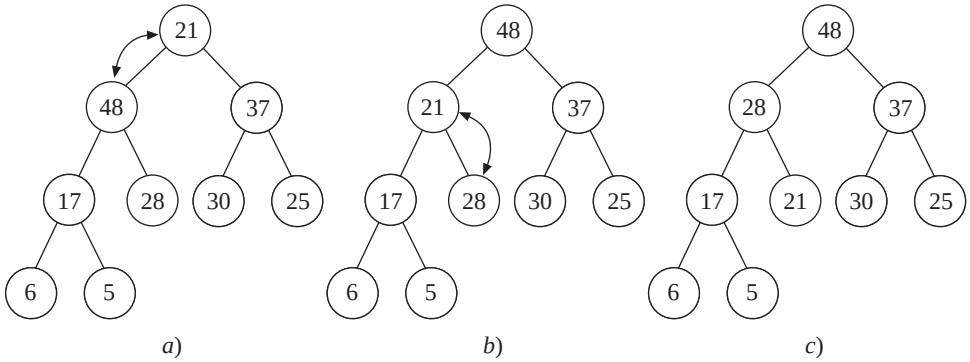
Por definición, la raíz del heap, $A(1)$, es el número más grande. Se supondrá que el heap ya está construido (la construcción de un heap se abordará más tarde). Así, es posible partir de $A(1)$. Luego de $A(1)$, que es el número más grande, el primer heap deja de ser un heap. Entonces, $A(1)$ se sustituye por $A(n) = A(10)$. Así, se tiene el árbol que se muestra en la figura 2-12.

FIGURA 2-12 Sustitución de $A(1)$ por $A(10)$.



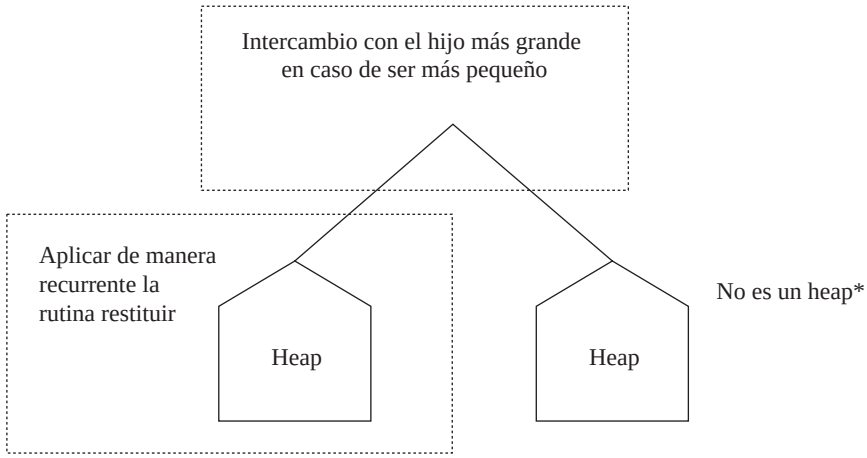
El árbol binario balanceado de la figura 2-12 no es un heap; sin embargo, es posible restituirlo fácilmente, como se muestra en la figura 2-13. El árbol binario de la figura 2-13*c*) es un heap.

FIGURA 2-13 Restitución de un heap.



La rutina restituir puede entenderse mejor con la figura 2-14.

FIGURA 2-14 La rutina restituir.



* Hasta no restituirlo, no es un heap. *(N. del R.T.)*

Algoritmo 2-6 □ Restore(*i, j*)

Input: $A(i), A(i+1),\ldots, A(j)$.
Output: $A(i), A(i+1),\ldots, A(j)$ como un heap.

Si $A(i)$ no es una hoja y si un hijo de $A(i)$ contiene un elemento más grande que $A(i)$, entonces

```
Begin
    Hacer que A(h) sea el hijo A(i) con el elemento más grande
    Intercambiar A(i) y A(h)
    Restore(h, j)                    (*restitución de un heap*)
End
```

El parámetro j se usa para determinar si $A(i)$ es una hoja o no y si $A(i)$ tiene dos hijos. Si $i > j/2$, entonces $A(i)$ es una hoja y restituir (i, j) no requiere hacer nada en absoluto porque $A(i)$ ya es un heap de por sí.

Puede afirmarse que hay dos elementos importantes en el ordenamiento heap sort:

1. Construcción del heap.
2. Eliminación del número más grande y restitución del heap.

Suponiendo que el heap ya está construido, entonces el ordenamiento heap sort puede describirse como sigue:

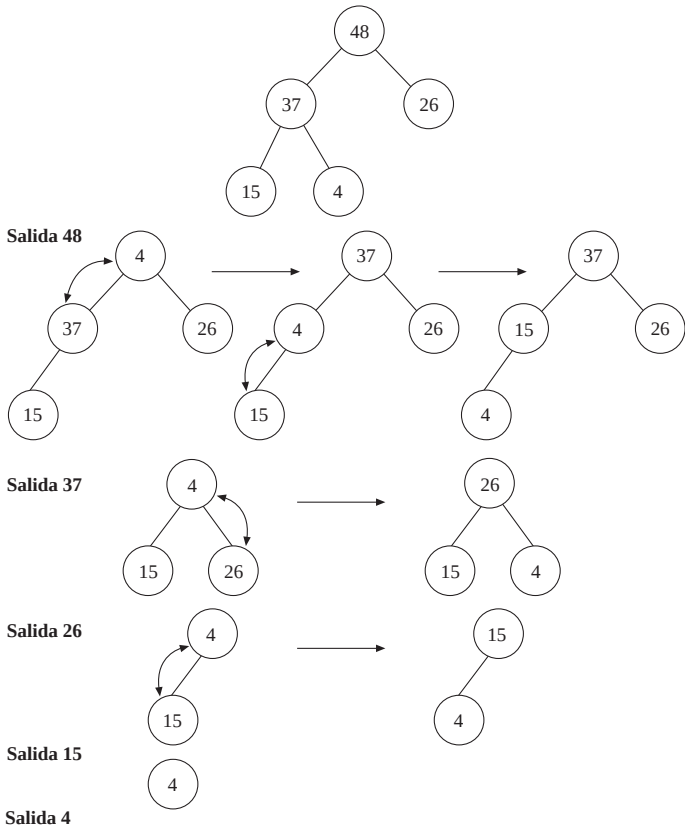
Algoritmo 2-7 □ Ordenamiento heap sort

Input: $A(1), A(2), \ldots, A(n)$ donde cada $A(i)$ es un nodo de un heap ya construido.
Output: La secuencia ordenada de las $A(i)$.

```
For i := n down to 2 do                (*ciclo decreciente*)
Begin
    Output A(1)
    A(1) := A(i)
    Delete A(i)
    Restore (1, i − 1)
End
Output A(1)
```

Ejemplo 2-6 Ordenamiento heap sort

Los pasos siguientes muestran un caso típico de ordenamiento heap sort.



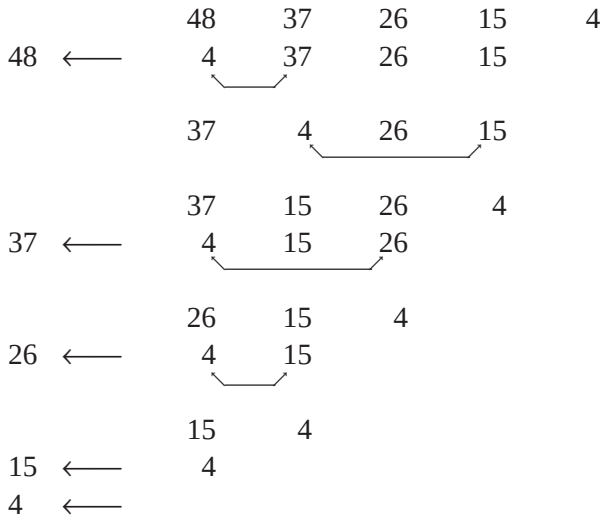
Una bella característica de un heap es que es posible representarlo mediante un arreglo. Es decir, no se requieren apuntadores porque un heap es un árbol binario completamente balanceado. Cada nodo y sus descendientes pueden determinarse de manera única. La regla es más bien simple: los descendientes de $A(h)$ son $A(2h)$ y $A(2h + 1)$ en caso de existir.

El heap de la figura 2-11 se almacena ahora como:

$A(1)$	$A(2)$	$A(3)$	$A(4)$	$A(5)$	$A(6)$	$A(7)$	$A(8)$	$A(9)$	$A(10)$
51	48	37	17	28	30	25	6	5	21

Considere, por ejemplo, $A(2)$. Su hijo izquierdo es $A(4) = 17$.
Considere $A(3)$. Su hijo derecho es $A(7) = 25$.

En consecuencia, todo el proceso de ordenamiento heap sort puede manejarse mediante una tabla o cuadro. Por ejemplo, el ordenamiento heap sort en el ejemplo 2-6 ahora puede describirse como sigue:



La construcción de un heap

Para construir un heap considere la figura 2-14, donde el árbol binario no es un heap. Sin embargo, ambos subárboles debajo de la parte superior del árbol son heaps. Para esta clase de árboles es posible "construir" un heap usando la subrutina restitución. La construcción se basa en la idea anterior. Se empieza con cualquier árbol binario completamente balanceado arbitrario y gradualmente se transforma en un heap al invocar repetidamente la subrutina restitución.

Sea $A(1), A(2),\ldots, A(n)$ un árbol binario completamente balanceado cuyos nodos hoja al nivel más elevado se encuentran lo más a la izquierda posible. Para este árbol binario, puede verse que $A(i)$, $i = 1, 2,\ldots, \lfloor n/2 \rfloor$ debe ser un nodo interno con descendientes y que $A(i)$, $i = \lfloor n/2 \rfloor + 1,\ldots, n$ debe ser un nodo hoja sin descendientes. Todos los nodos hoja pueden considerarse trivialmente como heaps. Así, no es necesario realizar ninguna operación sobre ellos. La construcción de un heap comienza desde la restitución del subárbol cuya raíz está en $\lfloor n/2 \rfloor$. El algoritmo para construir un heap es como sigue:

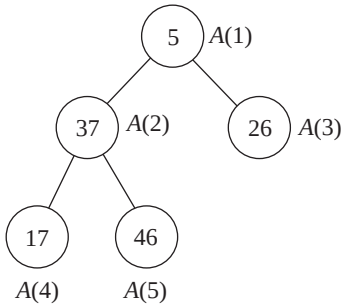
Algoritmo 2-8 □ Construcción de un heap

Input: $A(1), A(2),\ldots, A(n)$.
Output: $A(1), A(2),\ldots, A(n)$ como heap.

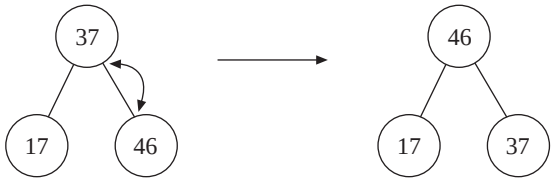
```
For i := ⌊n/2⌋ down to 1 do
    Restore (i, n)
```

Ejemplo 2-7 Construcción de un heap

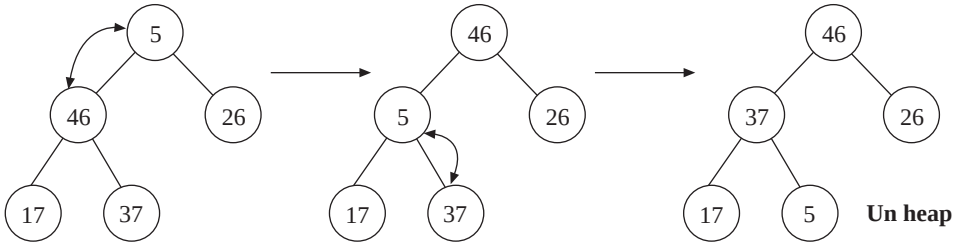
Considere el siguiente árbol binario que no es un heap.



En este heap, $n = 5$ y $\lfloor n/2 \rfloor = 2$. Por consiguiente, el subárbol cuya raíz está en $A(2)$ se restituye como sigue:



Luego se restituye $A(1)$:



El ordenamiento heap sort es una mejora del knockout sort porque para representar el heap se usa un arreglo lineal. A continuación se analiza la complejidad temporal del ordenamiento heap sort.

Análisis del peor caso de la construcción de un heap

Considere que hay n números para ordenar. La profundidad d de un árbol binario completamente balanceado es por lo tanto $\lfloor \log n \rfloor$. Para cada nodo interno es necesario

realizar dos comparaciones. Sea L el nivel de un nodo interno. Entonces, en el peor caso, para ejecutar la subrutina restituir es necesario efectuar $2(d - L)$ comparaciones. El número máximo de nodos en el nivel L es 2^L. Así, el número total de comparaciones para la etapa de construcción es a lo sumo

$$\sum_{L=0}^{d-1} 2(d-L)2^L = 2d\sum_{L=0}^{d-1} 2^L - 4\sum_{L=0}^{d-1} L2^{L-1}.$$

En el apartado 2-2, en la ecuación (2-1) se demostró que

$$\sum_{L=0}^{k} L2^{L-1} = 2^k(k-1)+1.$$

En consecuencia,

$$\begin{aligned}\sum_{L=0}^{d-1} 2(d-L)2^L &= 2d\sum_{L=0}^{d-1} 2^L - 4\sum_{L=0}^{d-1} L2^{L-1}\\ &= 2d(2^d - 1) - 4(2^{d-1}(d-1-1)+1)\\ &= 2d(2^d-1) - 4(d2^{d-1} - 2^d + 1)\\ &= 4\cdot 2^d - 2^d - 4\\ &= 4\cdot 2^{\lfloor \log n\rfloor} - \lfloor 2\log n\rfloor - 4\\ &= cn - \lfloor 2\log n\rfloor - 4 \qquad \text{donde } 2 \le c \le 4\\ &\le cn.\end{aligned}$$

Así, el número total de comparaciones necesarias para construir un heap en el peor caso es $O(n)$.

Complejidad temporal de eliminar elementos de un heap

Como se demostró, después de que se construye un heap, la parte superior de éste es el número más grande y ahora ya es posible eliminarlo (o sacarlo). A continuación se analiza el número de comparaciones necesarias para sacar todos los elementos numéricos de un heap que consta de n elementos. Observe que después de eliminar un número, en el peor caso, para restituir el heap se requieren $2\lfloor \log i\rfloor$ comparaciones si quedan i elementos. Así, el número total de pasos necesarios para eliminar todos los números es

$$2\sum_{i=1}^{n-1} \lfloor \log i \rfloor.$$

Para evaluar esta fórmula, se considerará el caso de $n = 10$.

$\lfloor \log 1 \rfloor = 0$
$\lfloor \log 2 \rfloor = \lfloor \log 3 \rfloor = 1$
$\lfloor \log 4 \rfloor = \lfloor \log 5 \rfloor = \lfloor \log 6 \rfloor = \lfloor \log 7 \rfloor = 2$
$\lfloor \log 8 \rfloor = \lfloor \log 9 \rfloor = 3.$

Se observa que hay

2^1 números iguales a $\lfloor \log 2^1 \rfloor = 1$
2^2 números iguales a $\lfloor \log 2^2 \rfloor = 2$

y $10 - 2^{\lfloor \log 10 \rfloor} = 10 - 2^3 = 2$ números iguales a $\lfloor \log n \rfloor$.

En general,

$$2\sum_{i=1}^{n-1} \lfloor \log i \rfloor = 2\sum_{i=1}^{\lfloor \log n \rfloor - 1} i2^i + 2(n - 2^{\lfloor \log n \rfloor})\lfloor \log n \rfloor$$

$$= 4\sum_{i=1}^{\lfloor \log n \rfloor - 1} i2^{i-1} + 2(n - 2^{\lfloor \log n \rfloor})\lfloor \log n \rfloor.$$

Al usar $\sum_{i=1}^{k} i2^{i-1} = 2^k(k-1)+1$ (ecuación 2-1 en el apartado 2-2)

se tiene

$$2\sum_{i=1}^{n-1} \lfloor \log i \rfloor$$

$$= 4\sum_{i=1}^{\lfloor \log n \rfloor - 1} i2^{i-1} + 2(n - 2^{\lfloor \log n \rfloor})\lfloor \log n \rfloor$$

$$= 4(2^{\lfloor \log n \rfloor - 1}(\lfloor \log n \rfloor - 1 - 1) + 1) + 2n\lfloor \log n \rfloor - 2\lfloor \log n \rfloor 2^{\lfloor \log n \rfloor}$$

$$= 2 \cdot 2^{\lfloor \log n \rfloor}\lfloor \log n \rfloor - 8 \cdot 2^{\lfloor \log n \rfloor - 1} + 4 + 2n\lfloor \log n \rfloor - 2 \cdot 2^{\lfloor \log n \rfloor}\lfloor \log n \rfloor$$

$$= 2 \cdot n\lfloor \log n \rfloor - 4 \cdot 2^{\lfloor \log n \rfloor} + 4$$

$$= 2n\lfloor \log n \rfloor - 4cn + 4 \qquad \text{donde } 2 \le c \le 4$$

$$= O(n \log n).$$

En consecuencia, la complejidad temporal del peor caso para obtener todos los elementos de un heap en orden clasificado es $O(n \log n)$.

En resumen, se concluye que la complejidad temporal del peor caso del ordenamiento heap sort es $O(n \log n)$. Aquí se recalca que el ordenamiento heap sort alcanza esta complejidad temporal de $O(n \log n)$ esencialmente porque utiliza una estructura de datos de modo que cada operación de salida requiere a lo sumo $\lfloor \log i \rfloor$ pasos, donde i es el número de elementos restantes. Este inteligente diseño de estructura de datos es fundamental para el ordenamiento heap sort.

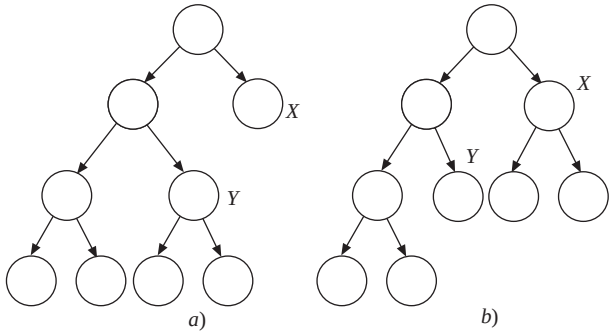
2-6 LA COTA INFERIOR DEL CASO PROMEDIO DEL ORDENAMIENTO

En el apartado 2-4 se estudió la cota inferior del peor caso del ordenamiento. En esta sección se deducirá la cota inferior del caso promedio del problema de ordenamiento. Seguirá usándose el modelo del árbol de decisión binario.

Como ya se analizó, todo algoritmo de ordenamiento basado en comparaciones puede describirse mediante un árbol de decisión binario. En este árbol la ruta que va de su raíz a un nodo hoja corresponde a la acción del algoritmo en respuesta a un caso particular de entrada. Además, la longitud de esta ruta es igual al número de comparaciones ejecutadas para este conjunto de datos de entrada. Definiremos *la longitud de la ruta externa de un árbol como la suma de las longitudes de las rutas que van de la raíz a cada uno de los nodos hoja*. Así, la complejidad temporal media de un algoritmo de ordenamiento basado en comparaciones es igual a la longitud de la ruta externa del árbol de decisión binario correspondiente a este algoritmo dividida entre el número de nodos hoja, que es $n!$

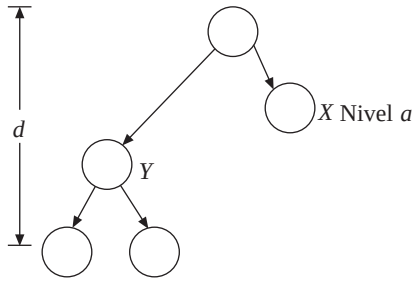
Para encontrar la cota inferior de la complejidad temporal del ordenamiento es necesario determinar la mínima longitud de la ruta externa de todos los árboles binarios posibles con $n!$ nodos hoja. Entre todos los árboles binarios posibles con un número fijo de nodos hoja, la mínima longitud de la ruta externa se minimiza si el árbol está balanceado. Es decir, todos los nodos hoja están en el nivel d o en el nivel $d - 1$ para alguna d. Considere la figura 2-15. En la figura 2-15*a*), el árbol no está balanceado. La longitud de la ruta externa de este árbol es $4 \times 3 + 1 = 13$. Ahora ya es posible reducir esta longitud de la ruta externa eliminando los dos descendientes de Y y asignándolos a X. Así, la longitud de la ruta externa se convierte ahora en $2 \times 3 + 3 \times 2 = 12$.

FIGURA 2-15 Modificación de un árbol binario no balanceado.



El caso general se describe ahora en la figura 2-16. En esta figura, suponga que en el nivel a hay un nodo hoja y que la profundidad del árbol es d, donde $a \leq d - 2$. Este árbol puede modificarse de modo que la longitud de la ruta externa se reduzca sin cambiar el número de nodos hoja. Para modificar el árbol, se selecciona cualquier nodo en el nivel $d - 1$ que tenga descendientes en el nivel d. El nodo hoja que está en el nivel a y el nodo que está en el nivel $d - 1$ se denotan por X y Y, respectivamente. Se quitan los descendientes de Y y se asignan a X. Para el nodo Y, éste originalmente tenía dos descendientes y la suma de la longitud de sus rutas es $2d$. Luego, Y se vuelve un nodo hoja y la longitud de su ruta es $d - 1$. Esta eliminación reduce la longitud de la ruta externa por $2d - (d - 1) = d + 1$. Para X, éste originalmente era un nodo hoja. Ahora se convierte en un nodo interno y sus dos nodos descendientes se vuelven nodos hoja. En un principio, la longitud de la ruta era a. Ahora la suma de las dos longitudes de las rutas es $2(a + 1)$. Así, esta colocación incrementa la longitud de la ruta externa por $2(a + 1) - a = a + 2$. El cambio neto es $(d + 1) - (a + 2) = (d - a) - 1 \geq 2 - 1 = 1$. Es decir, el cambio neto es disminuir la longitud de la ruta externa.

FIGURA 2-16 Árbol binario no balanceado.



En consecuencia, se concluye que un árbol binario no balanceado puede modificarse de modo que la longitud de la ruta externa disminuya y la longitud de la ruta externa de un árbol binario se minimiza si y sólo si el árbol está balanceado.

Considere que en total hay x nodos hoja. Ahora se calculará la longitud de la ruta externa de un árbol binario balanceado que tiene c nodos hoja. Esta longitud de la ruta externa se encuentra aplicando el siguiente razonamiento:

1. La profundidad del árbol es $d = \lceil \log c \rceil$. Los nodos hoja sólo pueden aparecer en el nivel d o en el nivel $d - 1$.
2. En el nivel $d - 1$ hay x_1 nodos hoja y en el nivel d hay x_2 nodos hoja. Luego,

$$x_1 + x_2 = c.$$

3. Para simplificar el análisis se supondrá que el número de nodos en el nivel d es par. El lector podrá observar fácilmente que si el número de nodos en el nivel d es impar, entonces el siguiente resultado sigue siendo verdadero. Para cada dos nodos en el nivel d, en el nivel $d - 1$ hay un nodo padre. Este nodo padre no es un nodo hoja. Así, se tiene la siguiente ecuación

$$x_1 + \frac{x_2}{2} = 2^{d-1}.$$

4. Al resolver estas ecuaciones se obtiene

$$\frac{x_2}{2} = c - 2^{d-1}$$
$$x_2 = 2(c - 2^{d-1})$$
$$x_1 = 2^d - c.$$

5. La longitud de la ruta externa es

$$x_1(d - 1) + x_2 d$$
$$= (2^d - c)(d - 1) + (2c - 2^d)d$$
$$= c + cd - 2^d.$$

6. Debido a que $d = \lceil \log c \rceil$, al sustituir $\log c \leq d < \log c + 1$ en la ecuación anterior, se tiene $c + cd - 2^d \geq c + c(\log c) - 2.2^{\log c} = c \log c - c$. Así, la longitud de la ruta externa es mayor que $c \log c - c = n! \log n! - n!$ En consecuencia, la complejidad temporal del ordenamiento en el caso promedio es mayor que

$$\frac{n! \log n! - n!}{n!} = \log n! - 1.$$

Al usar el resultado que se analizó en el apartado 2-4, ahora se concluye que *la cota inferior en el caso promedio del problema de ordenamiento es* $\Omega(n \log n)$.

En el ejemplo 2-4 del apartado 2-2 se demostró que la complejidad temporal en el caso promedio del quick sort es $O(n \log n)$. Así, el quick sort es óptimo por lo que se refiere a su desempeño en el caso promedio.

En el ejemplo 2-3 del apartado 2-2 se demostró que la complejidad temporal en el caso promedio del ordenamiento por selección directa también es $O(n \log n)$. Sin embargo, debe entenderse que esta complejidad temporal es en términos del cambio de señal. El número de comparaciones para el ordenamiento por selección directa es $n(n - 1)/2$ en los casos promedio y peor. Debido a que el número de comparaciones es un factor temporal dominante en la programación práctica, resulta que en la práctica el ordenamiento por selección directa es bastante lento.

La complejidad temporal en el caso promedio del famoso ordenamiento por burbuja, así como el ordenamiento por inserción directa, es $O(n^2)$. La experiencia indica que el ordenamiento por burbuja es mucho más lento que el quick sort.

Finalmente se abordará el ordenamiento heap sort que se analizó en el apartado 2-5. La complejidad temporal en el peor caso del ordenamiento heap sort es $O(n \log n)$ y la complejidad temporal en el caso promedio del ordenamiento heap sort nunca ha sido determinada. Sin embargo, se sabe que debe ser mayor o igual a $O(n \log n)$ debido a la cota inferior que se encontró en esa sección. Pero no puede ser mayor que $O(n \log n)$ porque su complejidad temporal en el peor caso es $O(n \log n)$. En consecuencia, es posible deducir el hecho de que la complejidad temporal en el caso promedio del ordenamiento heap sort es $O(n \log n)$.

2-7 CÓMO MEJORAR UNA COTA INFERIOR MEDIANTE ORÁCULOS

En la sección previa se demostró cómo usar el modelo del árbol de decisión binario a fin de obtener una cota inferior para el ordenamiento. Es simplemente afortunado que la cota inferior haya sido tan buena. Es decir, existe un algoritmo cuya complejidad temporal en el peor caso es exactamente igual a esta cota inferior. En consecuencia, puede tenerse la certeza de que ya no es posible hacer más alta esta cota inferior.

En esta sección se presentará un caso en que el modelo del árbol de decisión binario no produce una cota inferior muy significativa. Es decir, se mostrará que aún es posible mejorar la cota inferior obtenida usando el modelo del árbol de decisión binario.

Considere el problema de fusión. Si el algoritmo de fusión (merge) se basa en la operación de comparación e intercambio, entonces es posible usar el modelo del árbol de decisión. Se puede deducir una cota inferior de fusión aplicando el razonamiento para la obtención de la cota inferior del ordenamiento. En el ordenamiento, el número

de nodos hoja es $n!$, de modo que la cota inferior para el ordenamiento es $\lfloor \log_2 n! \rfloor$. En la fusión, el número de nodos hoja sigue siendo el número de casos distintos que quieren distinguirse. Así, dadas dos secuencias ordenadas A y B de m y n elementos, respectivamente, ¿cuántas secuencias fusionadas diferentes posibles hay? De nuevo, para simplificar el análisis, se supondrá que todos los $(m + n)$ elementos son distintos. Después de que n elementos se han fusionado en m elementos, en total hay $\binom{m+n}{n}$ formas de fusionarlos sin perturbar el orden original de las secuencias A y B. Esto significa que es posible obtener una cota inferior para la fusión como

$$\left\lceil \log \binom{m+n}{n} \right\rceil.$$

Sin embargo, jamás se ha determinado ningún algoritmo de fusión con el que se obtenga esta cota inferior.

A continuación se considerará un algoritmo de fusión convencional que compara los elementos superiores de dos listas ordenadas y da como salida el menor. Para este algoritmo de fusión, la complejidad temporal en el peor caso es $m + n - 1$, que es mayor que o igual a

$$\left\lceil \log \binom{m+n}{n} \right\rceil.$$

Es decir, se tiene la siguiente desigualdad:

$$\left\lceil \log \binom{m+n}{n} \right\rceil \leq m + n - 1.$$

¿Cómo puede establecerse un puente en la brecha? Según el análisis hecho, es posible ya sea incrementar la cota inferior o encontrar un mejor algoritmo cuya complejidad temporal sea más baja. De hecho, resulta interesante observar que cuando $m = n$, otra cota inferior de la fusión es $m + n - 1 = 2n - 1$.

Esto se demostrará mediante el enfoque del oráculo. Un oráculo proporcionará un caso muy difícil (un dato de entrada particular). Si se aplica cualquier algoritmo a este conjunto de datos, el algoritmo deberá trabajar bastante para resolver el problema. Al usar este conjunto de datos, es posible deducir una cota inferior para el peor caso.

Suponga que se tienen dos secuencias $a_1, a_2, \ldots, a_n$ y $b_1, b_2, \ldots, b_n$. Además, considere el muy difícil caso en que $a_1 < b_1 < a_2 \ldots a_n < b_n$. Suponga que algún algoritmo de fusión ya ha fusionado correctamente $a_1, a_2, \ldots, a_{i-1}$ con $b_1, b_2, \ldots, b_{i-1}$ y que produce la siguiente secuencia ordenada:

$$a_1, b_1, \ldots, a_{i-1}, b_{i-1}.$$

No obstante, suponga que este algoritmo de fusión no compara a_i con b_i. Resulta evidente que no hay ninguna forma en que el algoritmo haga una decisión correcta sobre si a_i, o b_i, debe colocarse al lado de b_{i-1}. Así, es necesario comparar a_i y b_i. Aplicando un razonamiento semejante puede demostrarse que es necesario comparar b_i y a_{i+1} después que a_i se ha escrito al lado de b_{i-1}. En resumen, cada b_i debe compararse con a_i y a_{i+1}. En consecuencia, cuando $m = n$, cualquier algoritmo de fusión requiere efectuar en total $2n - 1$ comparaciones. Quisiéramos recordar al lector que esta cota inferior $2n - 1$ para fusionar sólo es válida para el caso en que $m = n$.

Debido a que el algoritmo de fusión convencional requiere $2n - 1$ comparaciones para el caso en que $m = n$, puede concluirse que el algoritmo de fusión convencional es óptimo porque su complejidad temporal en el peor caso es igual a esta cota inferior.

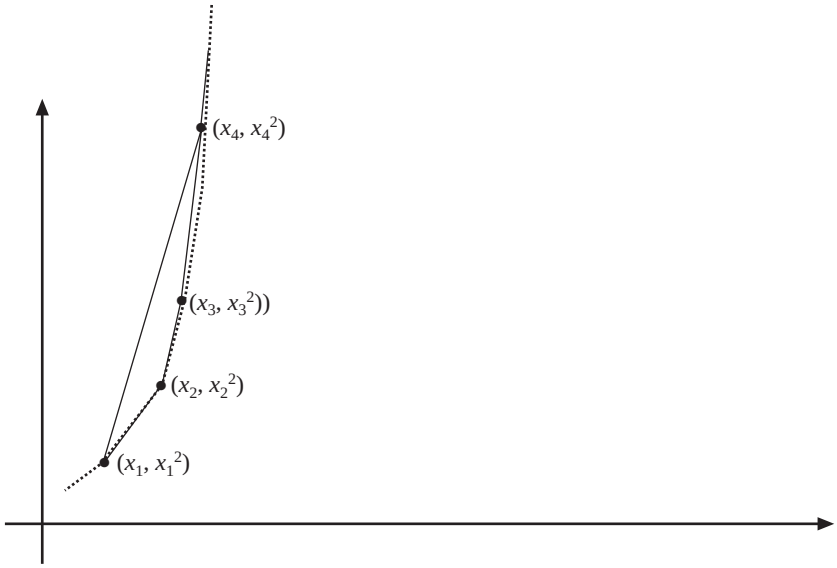
El análisis anterior muestra que algunas veces es posible mejorar una cota inferior con una más alta.

2-8 DETERMINACIÓN DE LA COTA INFERIOR POR TRANSFORMACIÓN DEL PROBLEMA

En la sección previa se encontraron cotas inferiores mediante el análisis directo de los problemas. Algunas veces esto parece difícil. Por ejemplo, el problema de la cubierta convexa (convex hull) consiste en encontrar el menor polígono (cubierta) convexo de un conjunto de puntos en el plano. ¿Cuál es la cota inferior del problema de la cubierta convexa? Parece más bien difícil encontrar directamente una cota inferior para este problema. No obstante, a continuación se demostrará que es fácil obtener una cota inferior bastante significativa mediante la transformación del problema de ordenamiento, cuya cota inferior se conoce, para este problema.

Sea $x_1, x_2, \ldots, x_n$ el conjunto de puntos a clasificar y, sin perder la generalidad, puede suponerse que $x_1 < x_2 \ldots < x_n$. Luego cada x_i se asocia con x_i^2 a fin de obtener un punto bidimensional (x_i, x_i^2). Todos estos puntos recién creados están en la parábola $y = x^2$. Considere la cubierta convexa construida con estos $n(x_i, x_i^2)$ puntos. Como se muestra en la figura 2-17, esta cubierta convexa consta de una lista de números clasificados. En otras palabras, al resolver el problema de la cubierta convexa también es posible resolver el problema de ordenamiento. El tiempo total del ordenamiento es igual al tiempo necesario para la transformación más el tiempo necesario para resolver el problema de la cubierta convexa. Así, la cota inferior del problema de la cubierta convexa es igual a la cota inferior del problema de ordenamiento menos el tiempo necesario para la transformación. Es decir, la cota inferior del problema de la cubierta convexa no es menor que $\Omega(n \log n) - \Omega(n) = \Omega(n \log n)$ cuando la transformación requiere $\Omega(n)$ pasos. El lector observará que esta cota inferior no puede hacerse más alta porque hay un algoritmo para resolver el problema de la cubierta convexa en $\Omega(n \log n)$ pasos.

FIGURA 2-17 Cubierta convexa construida a partir de los datos de un problema de ordenamiento.



En general, suponga que se quiere encontrar una cota inferior para el problema P_1. Sea P_2 un problema cuya cota inferior se desconoce. Además, suponga que P_2 puede transformarse en P_1, de modo que P_2 puede resolverse después que se ha resuelto P_1. Sean $\Omega(f_1(n))$ y $\Omega(f_2(n))$ que denotan las cotas inferiores de P_1 y P_2, respectivamente. Sea $O(g(n))$ el tiempo necesario para transformar P_2 en P_1. Entonces

$$\Omega(f_1(n)) + O(g(n)) \geq \Omega(f_2(n))$$
$$\Omega(f_1(n)) \geq \Omega(f_2(n)) - O(g(n)).$$

A continuación se proporciona otro ejemplo para demostrar la factibilidad de este enfoque. Suponga que se quiere encontrar la cota inferior del problema del árbol de expansión euclidiano mínimo. Debido a que es difícil obtener directamente la cota inferior de P_1, se considera P_2, que nuevamente es el problema de ordenamiento. Entonces se define la transformación: para todo x_i, sea $(x_i, 0)$ un punto bidimensional. Puede verse que el ordenamiento se completará tan pronto como se construya el árbol de expansión mínimo a partir de los $(x_i, 0)$. Nuevamente se supondrá que $x_1 < x_2 < \cdots < x_n$. Entonces en el árbol de expansión mínimo hay un borde entre $(x_i, 0)$ y $(x_j, 0)$ si y sólo si $j = i + 1$. En consecuencia, la solución del problema del árbol de expansión mínimo euclidiano también es una solución del problema de ordenamiento. De nuevo se observa que una cota inferior significativa del árbol de expansión mínimo euclidiano es $\Omega(n \log n)$.

2-9 NOTAS Y REFERENCIAS

En este capítulo se presentan algunos conceptos básicos del análisis de algoritmos. Para profundizar en el estudio del tema de análisis de algoritmos, consulte los siguientes autores: Basse y Van Gelder (2000); Aho, Hopcroft y Ullman (1974); Greene y Knuth (1981); Horowitz, Sahni y Rajasekaran (1998) y Purdom y Brown (1985a). Varios ganadores del premio Turing son excelentes investigadores de algoritmos. En 1987, la ACM Press publicó una colección de conferencias de 20 ganadores del premio Turing (Ashenhurst, 1987). En este volumen, todas las conferencias de Rabin, Knuth, Cook y Karp se refieren a las complejidades de los algoritmos. El premio Turing de 1986 fue otorgado a Hopcroft y Tarjan. La conferencia de Tarjan durante la recepción del premio Turing trataba del diseño de algoritmos y puede consultarse en Tarjan (1987). Weide (1977) también aportó una investigación sobre técnicas de análisis de algoritmos.

En este capítulo se presentaron varios algoritmos de ordenamiento. Para conocer un análisis más detallado sobre ordenamiento y búsqueda, puede consultarse la obra de Knuth (1973). Para más material sobre el análisis del ordenamiento por inserción directa, la búsqueda binaria y el ordenamiento por selección directa, consulte las secciones 5.2.1, 6.2.1 y la sección 5.2.3, respectivamente, de la obra de Knuth (1973). El quick sort fue obra de Hoare (1961, 1962). El ordenamiento heap sort fue descubierto por Williams (1964). Para más detalles sobre la determinación de la cota inferior de un ordenamiento, consulte la sección 5.3.1 de la obra de Knuth (1973); acerca del ordenamiento knockout sort consulte la sección 5.2.3 del mismo autor (1973).

La terminología básica de un árbol puede encontrarse en muchos libros de texto sobre estructura de datos. Por ejemplo, vea las secciones 5.1 y 5.2 de Horowitz y Sahni (1976). La profundidad de un árbol también se conoce como la altura de un árbol. Para saber más sobre el análisis de la longitud de la ruta externa y el efecto del árbol binario completo, consulte la sección 2.3.4.5 del libro de Knuth (1969).

En la sección 6.1 de la obra de Preparata y Shamos (1985) puede encontrarse un estudio sobre el problema del árbol de expansión mínimo. Hay más información sobre el problema de la determinación del rango en la sección 4.1 de libro de Shamos (1978) y en la sección 8.8.3 del libro de Preparata y Shamos (1985). La demostración de que la complejidad temporal de tiempo medio para encontrar la mediana es $O(n)$ puede consultarse en la sección 3.6 de la obra de Horowitz y Sahni (1978).

Para consultar material sobre el mejoramiento de una cota inferior a través de oráculos, vea la sección 5.3.2 del libro de Knuth (1973) y también la sección 10.2 de la obra de Horowitz y Sahni (1978). En cuanto a material sobre la determinación de cotas inferiores mediante la transformación del problema, consulte las secciones 3.4 y 6.1.4 del libro de Shamos (1978) y también las secciones 3.2 y 5.3 del de Preparata y Shamos (1985). En Shamos (1978) y Preparata y Shamos (1985) hay muchos ejemplos que prueban las cotas inferiores por transformación.

2-10 BIBLIOGRAFÍA ADICIONAL

Las teorías sobre cotas inferiores siempre han atraído a los investigadores. Algunos artículos que se han publicado recientemente sobre este tema son de los siguientes autores: Dobkin y Lipton (1979); Edwards y Elphick (1983); Frederickson (1984); Fredman (1981); Grandjean (1988); Hasham y Sack (1987); Hayward (1987); John (1988); Karp (1972); McDiarmid (1988); Mehlhorn , Naher y Alt (1988); Moran, Snir y Manber (1985); Nakayama, Nishizeki y Saito (1985); Rangan (1983); Traub y Wozniakowski (1984); Yao (1981), y Yao (1985).

Para algunos artículos muy interesantes de reciente publicación, consulte Berman, Karpinski, Larmore, Plandowski y Rytter (2002); Blazewicz y Kasprzak (2003); Bodlaender, Downey, Fellows y Wareham (1995); Boldi y Vigna (1999); Bonizzoni y Vedova (2001); Bryant (1999); Cole (1994); Cole y Hariharan (1997); Cole, Farach-Colton, Hariharan, Przytycka y Thorup (2000); Cole, Hariharan, Paterson y Zwick (1995); Crescenzi, Goldman, Papadimitriou, Piccolboni y Yannakakis (1998); Darve (2000); Day (1987); Decatur, Goldreich y Ron (1999); Demri y Schnoebelen (2002); Downey, Fellows, Vardy y Whittle (1999); Hasewaga y Horai (1991); Hoang y Thierauf (2003); Jerrum (1985); Juedes y Lutz (1995); Kannan, Lawler y Warnow (1996); Kaplan y Shamir (1994); Kontogiannis (2002); Leoncini, Manzini y Margara (1999); Maes (1990); Maier (1978); Marion (2003); Martinez y Roura (2001); Matousek (1991); Naor y Ruah (2001); Novak y Wozniakowski (2000); Owolabi y McGregor (1988); Pacholski, Szwast y Tendera (2000); Peleg y Rubinovich (2000), y Ponzio, Radhakrishnan y Venkatesh (2001).

Ejercicios

2.1 Proporcione los números de intercambios necesarios para los casos mejor, peor y promedio en el ordenamiento por burbuja, cuya definición puede encontrarse en casi todos los libros de texto sobre algoritmos. Los análisis de los casos mejor y peor son triviales. El análisis para el caso promedio puede realizarse mediante el siguiente proceso:

1. Defina la inversa de una permutación. Sea $a_1, a_2, \ldots, a_n$ una permutación del conjunto $(1, 2, \ldots, n)$. Si $i < j$ y $a_j < a_i$, entonces (a_i, a_j) se

denomina inversión de esta permutación. Por ejemplo, (3, 2) (3, 1) (2, 1) (4, 1) son, todas, inversiones de la permutación (3, 2, 4, 1).

2. Encuentre la relación entre la probabilidad de que una permutación dada tenga exactamente k inversiones y la probabilidad de que el número de permutaciones de n elementos tenga exactamente k inversiones.
3. Aplique inducción para demostrar que el número medio de intercambios necesarios para el ordenamiento por burbuja es $n(n-1)/4$.

2.2 Escriba un programa para ordenamiento por burbuja. Realice un experimento para convencerse de que, en efecto, el desempeño medio del algoritmo es $O(n^2)$.

2.3 Encuentre el algoritmo del algoritmo de Ford-Johnson para ordenar, que aparece en muchos libros de texto sobre algoritmos. Se demostró que este algoritmo es óptimo para $n \leq 12$. Implemente este algoritmo en una computadora y compárelo con cualquier otro algoritmo de ordenamiento. ¿Le agrada este algoritmo? En caso negativo, intente determinar qué falla en el análisis.

2.4 Demuestre que para clasificar cinco números se requieren por lo menos siete comparaciones. Luego, demuestre que el algoritmo de Ford-Johnson alcanza esta cota inferior.

2.5 Demuestre que para encontrar el número más grande en una lista de n números, por lo menos se requieren $n - 1$ comparaciones.

2.6 Demuestre que para encontrar el segundo elemento más grande de una lista de n números por lo menos se requieren $n - 2 + \lceil \log n \rceil$ comparaciones.

Sugerencia: No es posible determinar el segundo elemento más grande sin haber encontrado el primero. Así, el análisis puede efectuarse como sigue:

1. Demuestre que para encontrar el elemento más grande se requieren por lo menos $n - 1$ comparaciones.
2. Demuestre que siempre hay alguna secuencia de comparaciones que obliga a encontrar al segundo elemento más grande en $\lceil \log n \rceil - 1$ comparaciones adicionales.

2.7 Demuestre que si $T(n) = aT\left(\frac{n}{b}\right) + n^c$, entonces para n una potencia de b y

$$T(1) = k,\ T(n) = ka^{\log_b n} + n^c\left(\frac{b}{a-b^c}\right)\left(\left(\frac{a}{b^c}\right)^{\log_b n} - 1\right).$$

2.8 Demuestre que si $T(n) = \sqrt{n}T(\sqrt{n}) + n$, $T(m) = k$ y $m = n^{1/2^i}$, entonces

$$T(n) = kn^{(2^i-1)/2^i} + in.$$

2.9 Lea el teorema 10.5 que aparece en la obra de Horowitz y Sahni (1978). La demostración de este teorema constituye un buen método para encontrar una cota inferior.

2.10 Demuestre que la búsqueda binaria es óptima para todo algoritmo de búsqueda que sólo realice comparaciones.

2.11 Dados los siguientes pares de funciones, ¿cuál es el menor valor de n de modo que la primera función sea mayor que la segunda?

a) 2^n, $2n^2$.
b) $n^{1.5}$, $2n\log_2 n$.
c) n^3, $5n^{2.81}$.

2.12 ¿El tiempo $\Omega(n \log n)$ es una cota inferior para el problema de clasificar n enteros que varían de 1 a C, donde C es una constante? ¿Por qué?

capítulo

3

El método codicioso

El método codicioso (greedy) es una estrategia para resolver problemas de optimización. Se supondrá que es posible resolver un problema mediante una secuencia de decisiones. El método codicioso utiliza el siguiente enfoque: en cada etapa, la decisión es óptima. Para algunos problemas, como se verá, estas soluciones localmente óptimas se agregarán para integrar una solución globalmente óptima. Aquí se pone énfasis en que este método codicioso sólo es capaz de resolver algunos problemas de optimización. En casos en que las decisiones localmente óptimas no den por resultado una solución globalmente óptima, el método codicioso podría seguir siendo recomendable porque, como se verá más tarde, por lo menos produce una solución que suele ser aceptable.

A continuación, la característica del método codicioso se describirá con un ejemplo. Considere el caso en que dado un conjunto de n números se solicita escoger k números, de entre todas las formas que hay para elegir los k números del conjunto dado, de modo que la suma de estos k números sea máxima.

Para resolver este problema podrían probarse todas las formas posibles que hay para escoger k números de un conjunto de n números. Por supuesto, ésta es una forma absurda de resolver el problema, ya que simplemente podrían escogerse los k números más grandes, mismos que constituirían la solución. O bien, podría afirmarse que el algoritmo para resolver este problema es como sigue:

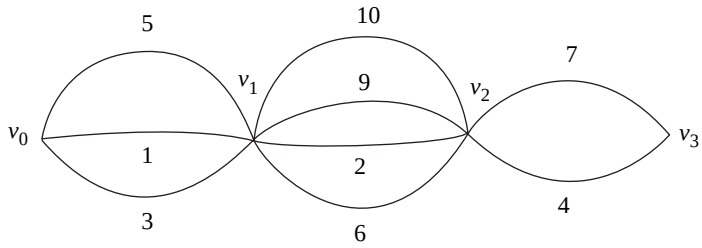
```
For i := 1 to k do
    Escoger el número más grande y eliminarlo de la entrada.
Endfor
```

El algoritmo anterior es un método codicioso típico. En cada etapa se selecciona el número más grande.

Se considerará otro caso en el que también es posible aplicar el método codicioso. En la figura 3-1 se solicita encontrar la ruta más corta de v_0 a v_3. Para esta gráfica particular, el problema puede resolverse encontrando una ruta más corta entre v_i y v_{i+1},

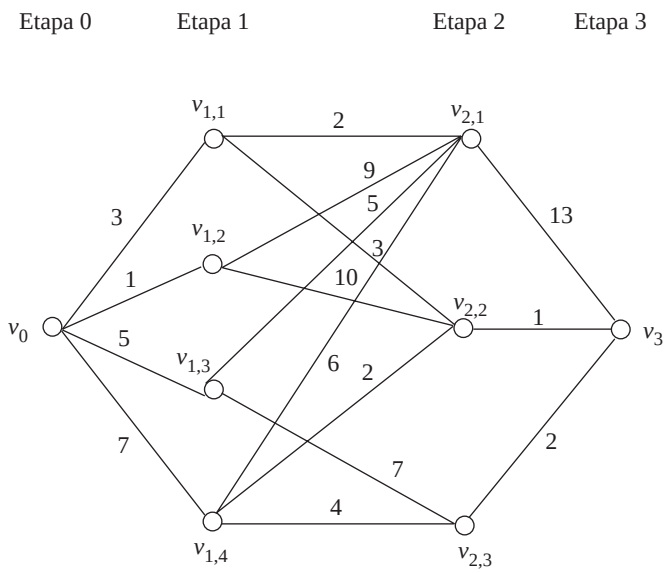
$i = 0$ hasta 2. Es decir, primero se determina la ruta más corta entre v_0 y v_1, luego entre v_1 y v_2, y así sucesivamente. Debe resultar evidente que este procedimiento termina por proporcionar una solución óptima.

FIGURA 3-1 Un caso en que funciona el método codicioso.



Sin embargo, fácilmente puede proporcionarse un ejemplo en que el método codicioso no funciona. Considere la figura 3-2.

FIGURA 3-2 Un caso en que no funciona el método codicioso.



En la figura 3-2 nuevamente se solicita obtener una ruta más corta de v_0 a v_3. Si se utiliza el método codicioso, en la etapa 1 se encontrará una ruta más corta de v_0 a algún

nodo en la etapa 1. Así, se selecciona $v_{1,2}$. En el siguiente movimiento se encontrará la ruta más corta entre $v_{1,2}$ y algún nodo en la etapa 2. Se selecciona $v_{2,1}$. La solución final es

$$v_0 \rightarrow v_{1,2} \rightarrow v_{2,1} \rightarrow v_3.$$

La longitud total de esta ruta es $1 + 9 + 13 = 23$.

Esta solución, aunque se obtuvo de manera rápida, no es una solución óptima. De hecho, la solución óptima es

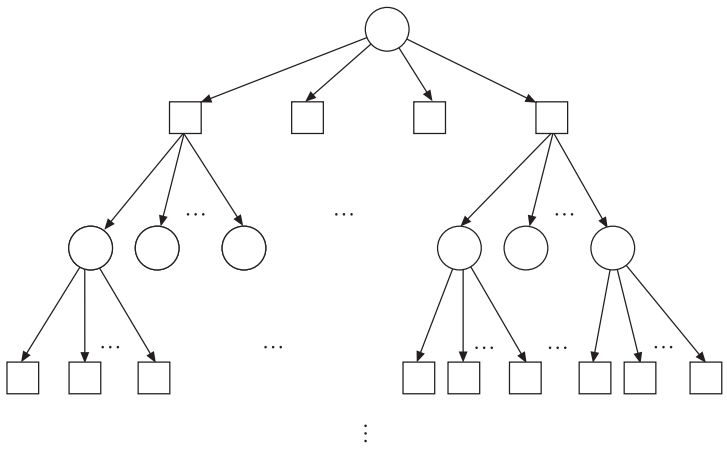
$$v_0 \rightarrow v_{1,1} \rightarrow v_{2,2} \rightarrow v_3$$

cuyo costo es $3 + 3 + 1 = 7$.

En capítulos posteriores se presentarán métodos para resolver el problema.

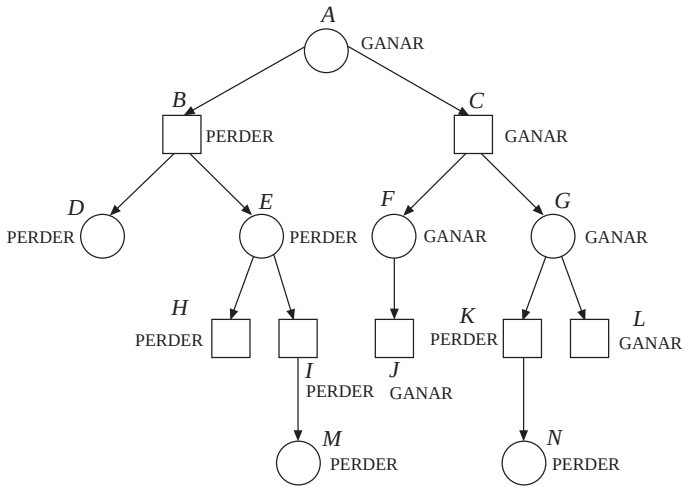
¿Qué falla cuando el método codicioso se aplica para resolver el problema que se muestra en la figura 3-2? Para contestar esta pregunta, por un momento la atención se dirigirá al juego de ajedrez. Un buen jugador de ajedrez no simplemente observa el tablero y hace un movimiento que es el mejor para él en ese momento; más bien mira "hacia delante". Es decir, tiene que imaginar la mayor parte de los movimientos posibles que puede efectuar y luego supone la forma en que podría reaccionar su oponente. Debe entender que su oponente también "mira hacia delante". Así, toda la partida podría verse como el árbol que se muestra en la figura 3-3.

FIGURA 3-3 Árbol de juego.



En la figura 3-3 un círculo representa un jugador y un cuadro representa a su oponente. El primer movimiento inteligente y correcto sólo puede hacerse cuando es posible *podar* (*prune*: eliminar algunas ramas) este árbol de juego. Considere la figura 3-4, que representa un árbol de final de juego ficticio.

FIGURA 3-4 Árbol de final de juego.



Con base en este árbol de final de juego es posible decidir qué movimiento realizar razonando que:

1. Debido a que el oponente siempre quiere que perdamos, *I* y *K* se identifican como PERDER; si se alcanza cualquiera de estos estados, definitivamente perderemos la partida.
2. Debido a que uno siempre quiere ganar, *F* y *G* se identifican como GANAR. Además, *E* debe identificarse como PERDER.
3. De nuevo, como el oponente siempre quiere que perdamos, *B* y *C* se identifican como PERDER y GANAR, respectivamente.
4. Debido a que uno quiere ganar, *A* se identifica como GANAR.

A partir del razonamiento anterior, *se entiende que cuando se toman decisiones, a menudo se mira hacia delante. Sin embargo, el método codicioso jamás hace ningún trabajo en ver hacia delante*. Considere la figura 3-2. Para encontrar una ruta más corta de v_0 a v_3, también es necesario ver hacia delante. Para seleccionar un nodo de los nodos de la etapa 1, es necesario conocer la distancia más corta de cada nodo de la etapa 1 a v_3. Sea $dmin(i, j)$ la distancia mínima entre los nodos i y j. Entonces

$$dmin(v_0, v_3) = \begin{cases} 3 + dmin(v_{1,1}, v_3) \\ 1 + dmin(v_{1,2}, v_3) \\ 5 + dmin(v_{1,3}, v_3) \\ 7 + dmin(v_{1,4}, v_3). \end{cases}$$

Ahora el lector puede ver que como el método codicioso no mira hacia delante, puede fracasar en cuanto a proporcionar una solución óptima. Sin embargo, en el resto de este capítulo se presentarán muchos ejemplos interesantes en los que el método codicioso funciona.

3-1 Método de Kruskal para encontrar un árbol de expansión mínima

Uno de los problemas más famosos que pueden resolverse con el método codicioso es el problema del árbol de expansión mínima. En esta sección se presentará el método de Kruskal para encontrar un árbol de expansión mínima. Los árboles de expansión mínima pueden definirse sobre puntos del espacio euclidiano o sobre una gráfica. Para el método de Kruskal, los árboles de expansión mínima se definen sobre gráficas.

Definición

Sea $G = (V, E)$ una gráfica no dirigida conexa ponderada, donde V representa el conjunto de vértices y E representa el conjunto de aristas. Un árbol de expansión de G es un árbol no dirigido $S = (V, T)$ donde T es un subconjunto de E. El peso total de un árbol de expansión es la suma de todos los pesos de T. Un árbol de expansión mínima de G es un árbol de expansión de G cuyo peso total es mínimo.

Ejemplo 3-1 Árbol de expansión mínima

En la figura 3-5 se muestra una gráfica que consta de cinco vértices y ocho aristas. En la figura 3-6 se muestran algunos de los árboles de expansión de esta gráfica. En la figura 3-7 se muestra un árbol de expansión mínima de la gráfica, cuyo peso total es $50 + 80 + 60 + 70 = 260$.

FIGURA 3-5 Gráfica no dirigida conexa ponderada.

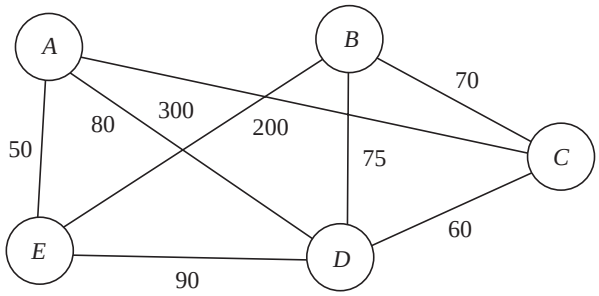
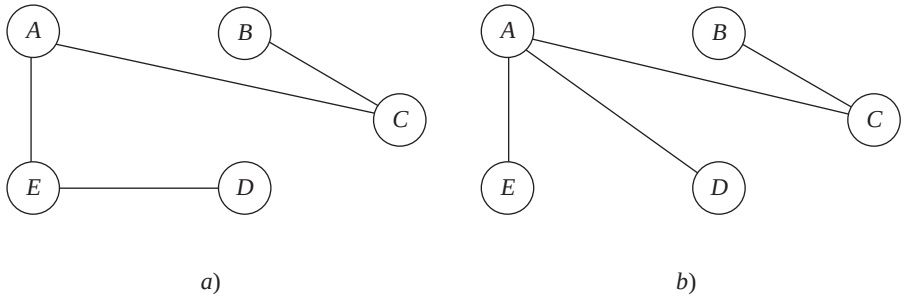
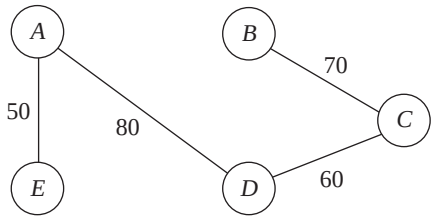


FIGURA 3-6 Algunos árboles de expansión.



a) *b*)

FIGURA 3-7 Árbol de expansión mínima.



El método de Kruskal para construir un árbol de expansión mínima puede describirse brevemente como sigue:

1. Del conjunto de aristas, seleccionar la de menor peso. Ésta constituye la subgráfica inicial construida parcialmente que después será desarrollada en un árbol de expansión mínima.
2. A esta gráfica construida parcialmente súmese la siguiente arista ponderada más pequeña si esto no provoca la formación de un ciclo. En caso contrario, eliminar la arista seleccionada.
3. Terminar si el árbol de expansión contiene $n - 1$ aristas. En caso contrario, ir a 2.

Algoritmo 3-1 □ Algoritmo de Kruskal del árbol de expansión mínima

Input: Una gráfica no dirigida, conexa y ponderada $G = (V, E)$.
Output: Un árbol de expansión mínima para G.
$T := \phi$
Mientras T contiene menos de $n - 1$ aristas, hacer
Begin
Elegir una arista (v, w) de E del peso más pequeño

```
        Delete (v, w) de E
        Si [al agregar (v, w) a T no se produce ningún ciclo en T] entonces
            Add (v, w) a T
        En otro caso
            Discard (v, w)
End
```

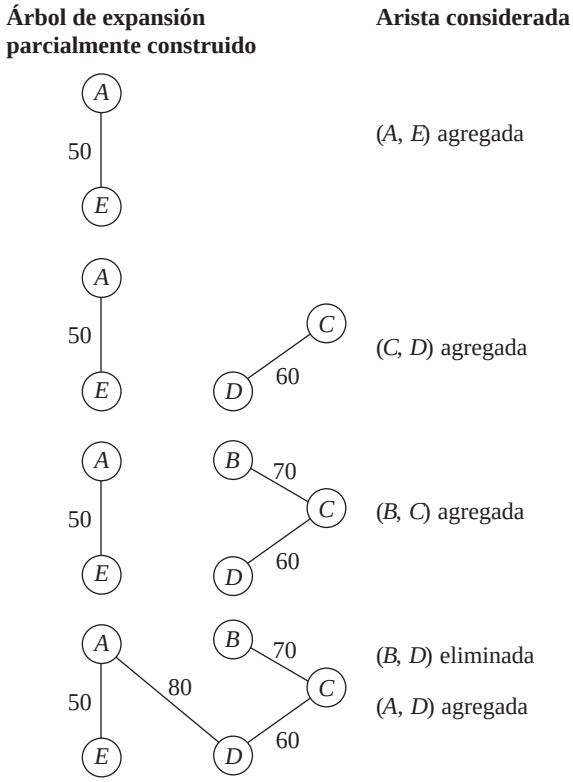
Ejemplo 3-2 Algoritmo de Kruskal

Considere la gráfica de la figura 3-5. Las aristas están ordenadas en la siguiente secuencia:

(*A*, *E*) (*C*, *D*) (*B*, *C*) (*B*, *D*) (*A*, *D*) (*E*, *D*) (*E*, *B*) (*A*, *C*).

Luego se construye el árbol de expansión mínima que se muestra en la figura 3-8.

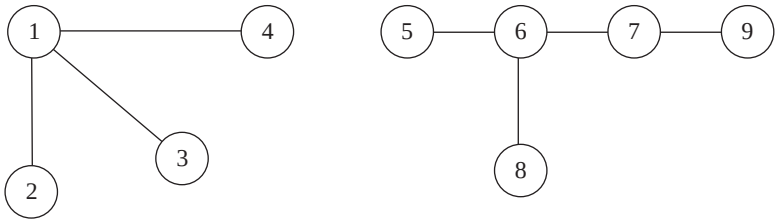
FIGURA 3-8 Determinación de un árbol de expansión mínima aplicando el algoritmo de Kruskal.



El lector debe observar que nunca fue necesario clasificar las aristas. De hecho, el heap considerado en el apartado 2-5 puede usarse para seleccionar la siguiente arista con el peso más pequeño.

Hay otro problema que es necesario resolver: ¿cómo puede determinarse eficientemente si la arista agregada formará un ciclo? La respuesta de esta pregunta es muy fácil. Durante el proceso del algoritmo de Kruskal, la subgráfica parcialmente construida es un bosque generador que consta de muchos árboles. En consecuencia, cada conjunto de vértices de un árbol puede preservarse en un conjunto individual. Considere la figura 3-9. En esta figura se observan dos árboles que pueden representarse como $S_1 = \{1, 2, 3, 4\}$ y $S_2 = \{5, 6, 7, 8, 9\}$. Suponga que la siguiente arista que ha de agregarse es (3, 4). Debido a que ambos vértices 3 y 4 están en S_1, esto originará que se forme un ciclo. Por consiguiente, no es posible agregar (3, 4). De manera semejante, no es posible agregar (8, 9) porque ambos vértices 8 y 9 están en S_2. No obstante, es posible agregar (4, 8).

FIGURA 3-9 Un bosque generador.



Con base en el razonamiento anterior puede verse que el algoritmo de Kruskal es dominado por las siguientes acciones:

1. Ordenamiento, que requiere $O(m \log m)$ pasos, donde m es el número de aristas en la gráfica.
2. Unión de dos conjuntos. Esto es necesario cuando se fusionan dos árboles. Cuando se inserta una arista que vincula dos subárboles, esencialmente se está encontrando la unión de dos conjuntos. Por ejemplo, suponga que la arista (4, 8) se agrega al árbol de expansión en la figura 3-9; al hacer lo anterior, los dos conjuntos {1, 2, 3, 4} y {5, 6, 7, 8, 9} se fusionan en {1, 2, 3, 4, 5, 6, 7, 8, 9}. Así, es necesario realizar una operación; a saber, la unión de dos conjuntos.
3. Determinación de un elemento en un conjunto. Observe que cuando se comprueba si es posible agregar o no una arista, es necesario verificar si dos vértices están o no en un conjunto de vértices. Así, es necesario realizar una operación, denominada operación encontrar, con la cual se determina si un elemento está o no en un conjunto.

En el capítulo 10 se mostrará que para efectuar y para encontrar las operaciones de unión se requieren $O(m)$ pasos. Así, el tiempo total del algoritmo de Kruskal es dominado por el ordenamiento, que es $O(m \log m)$. En el peor caso, $m = n^2$. En consecuencia, la complejidad temporal del algoritmo de Kruskal es $O(n^2 \log n)$.

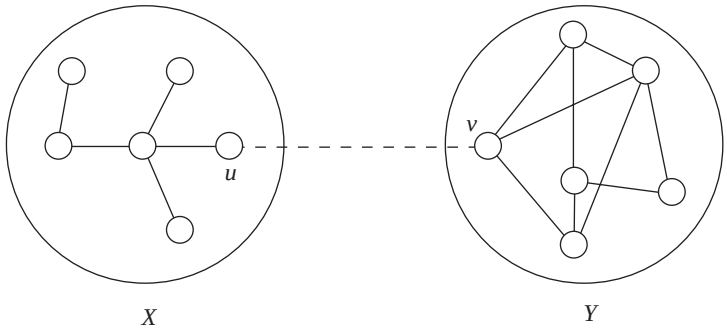
A continuación se demostrará que el algoritmo de Kruskal es correcto. Es decir, que produce un árbol de expansión mínima. Se supondrá que los pesos asociados con todas las aristas son distintos y que $|e_1| < |e_2| < \cdots < |e_m|$, donde $m = |E|$. Sean T el árbol de expansión producido por el algoritmo de Kruskal y T' un árbol de expansión mínima. Se demostrará que $T = T'$. Suponga lo contrario. Así, sea e_i la arista con peso mínimo en T que no aparece en T'. Resulta evidente que $i \neq 1$; e_i se agrega a T'. Esto necesariamente forma un ciclo en T'. Sea e_j una arista en este ciclo que no es una arista en T. Esta e_j debe existir. En caso contrario, todas las aristas en este ciclo pertenecen a T, lo cual significa que en T hay un ciclo imposible. Hay dos casos posibles. Caso 1: el peso de e_j es menor que el peso de e_i; por ejemplo, $j < i$. Sea T_k el árbol producido por el algoritmo de Kruskal después de comprobar si es posible o no agregar e_k. Resulta evidente que T_k, para $k < i$, es un subárbol tanto de T como de T' porque e_i es la arista de menor peso en T que no aparece en T', como se supuso. Debido a que $j < i$ y a que e_j no es una arista en T, entonces e_j no debe ser seleccionada por el algoritmo de Kruskal. La razón de esto es que al agregar e_j a T_{j-1} se forma un ciclo. No obstante, ya que T_{j-1} también es un subárbol de T' y como e_j es una arista de T', al agregar e_j a T_{j-1} no puede formarse un ciclo. Así, este caso es imposible. Caso 2: el peso de e_j es mayor que el de e_i; por ejemplo, $j > i$. En este caso se elimina e_j. De este modo se crea un nuevo árbol de expansión cuyo peso total es menor que el de T'. Así, T' debe ser el mismo que T.

Resulta evidente que el algoritmo de Kruskal utiliza el método codicioso. En cada paso, la siguiente arista que ha de agregarse es localmente óptima. Resulta interesante observar que el resultado final es globalmente óptimo.

3-2 Método de Prim para encontrar un árbol de expansión mínima

En el apartado 3-1 se presentó el algoritmo de Kruskal para encontrar un árbol de expansión mínima. En esta sección se presenta un algoritmo descubierto de manera independiente por Dijkstra y Prim. El algoritmo de Prim construye paso a paso un árbol de expansión mínima. En cualquier momento, sea X el conjunto de vértices contenidos en el árbol de expansión mínima parcialmente construido. Sea $Y = V - X$. La siguiente arista (u, v) que ha de agregarse es una arista entre X y Y ($u \in X$ y $v \in Y$) con el menor peso. La situación se describe en la figura 3-10. La siguiente arista agregada es (u, v) y después que se agrega esta arista, v se agrega a X y se elimina de Y. Un punto importante en el método de Prim es que puede empezarse en cualquier vértice, lo cual es muy conveniente.

FIGURA 3-10 Ilustración del método de Prim.



A continuación se presentará brevemente el algoritmo de Prim. Después se describirá con más detalle.

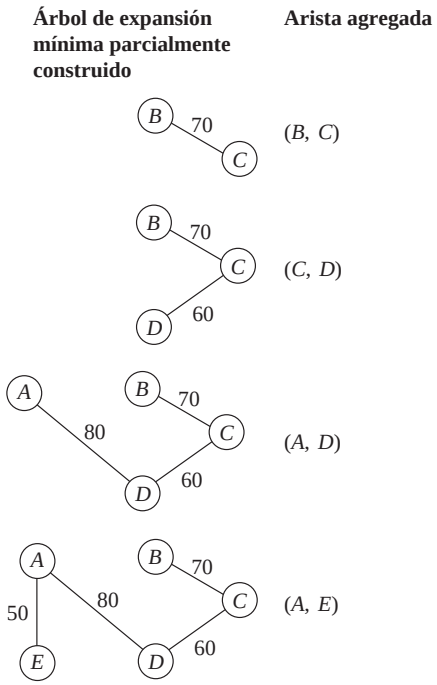
Algoritmo 3-2 □ Algoritmo básico de Prim para encontrar un árbol de expansión mínima

Input: Una gráfica no dirigida, conexa y ponderada $G = (V, E)$.
Output: Un árbol de expansión mínima para G.
Paso 1. Hacer x cualquier vértice en V. Sean $X = \{x\}$ y $Y = V - \{x\}$.
Paso 2. Seleccionar una arista (u, v) de E tal que $u \in X$, $v \in Y$ y (u, v) tiene el menor peso de todas las aristas entre X y Y.
Paso 3. Conectar u con v. Hacer $X = X \bigcup \{v\}$ y $Y = Y - \{v\}$.
Paso 4. Si Y es vacío, se termina y el árbol resultante es un árbol de expansión mínima. En caso contrario, ir al paso 2.

Ejemplo 3-3 Algoritmo básico de Prim

Considere nuevamente la figura 3-5. A continuación se mostrará cómo el algoritmo de Prim generaría un árbol de expansión mínima. Se supondrá que el vértice B se selecciona como punto inicial. En la figura 3-11 se ilustra este proceso. En cada paso del proceso, la arista que ha de agregarse incrementa al mínimo el costo total. Por ejemplo, cuando el árbol parcialmente construido contiene los vértices B, C y D, el conjunto de vértices restante es $\{A, E\}$. La arista de peso mínimo que une $\{A, E\}$ y $\{B, C, D\}$ es (A, D). Así, se agrega (A, D). Debido a que el vértice A no está contenido en el árbol de expansión mínima parcialmente construido, esta nueva arista no forma ningún ciclo.

FIGURA 3-11 Determinación de un árbol de expansión mínima con el algoritmo de Prim y vértice inicial *B*.



Quizás el lector esté interesado en saber si es posible empezar con algún otro vértice. En efecto, suponga que inicialmente se empieza con el vértice *C*. En la figura 3-12 se ilustra el árbol de expansión mínima que se construye.

FIGURA 3-12 Determinación de un árbol de expansión mínima con el algoritmo básico de Prim con vértice inicial *C*.

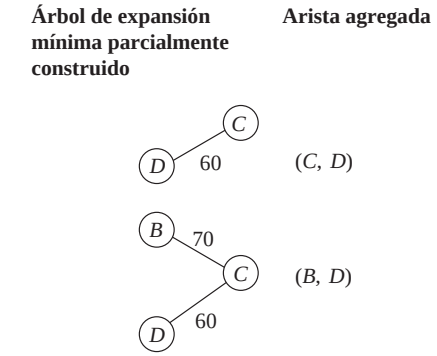
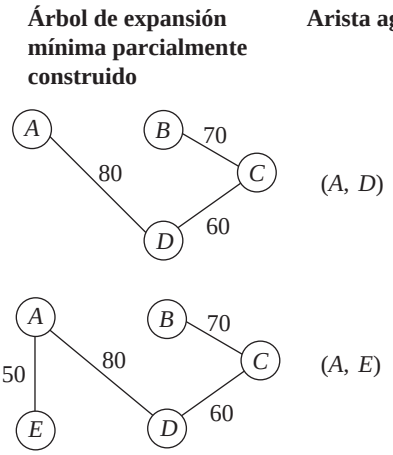
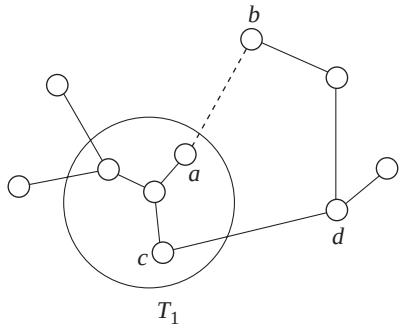


FIGURA 3-12 (*continuación*)



A continuación se demostrará que el algoritmo de Prim es correcto. Es decir, que el árbol de expansión que produce es en efecto un árbol de expansión mínima. Sea $G = (V, E)$ una gráfica conexa ponderada. Sin pérdida de generalidad puede suponerse que los pesos de todas las aristas son distintos. Sea T un árbol de expansión mínima para G. Sea T_1 un subárbol de T, como se muestra en la figura 3-13. Sean V_1 el conjunto de vértices en T_1 y $V_2 = V - V_1$. Sea (a, b) una arista de peso mínimo en E tal que $a \in V_1$ y $b \in V_2$. Se demostrará que (a, b) debe estar en T. Suponga lo contrario. Entonces debe haber una ruta en T de a a b porque un árbol es conexo. Sea (c, d) una arista en esa ruta tal que $c \in V_1$ y $d \in V_2$. El peso de (c, d) debe ser mayor que el peso de (a, b). En consecuencia, es posible crear otro árbol de expansión más pequeño al eliminar (c, d) y agregar (a, b). Esto prueba que T no debe ser un árbol de expansión mínima. Por consiguiente, (a, b) debe estar en T y el algoritmo de Prim es correcto.

FIGURA 3-13 Árbol de expansión mínima para explicar que el algoritmo de Prim es correcto.



El algoritmo que acaba de presentarse es sólo un breve bosquejo del algoritmo de Prim. Se presentó de modo que fuese fácil de comprender. A continuación se empezará desde el principio: $X = \{x\}$ y $Y = V - \{x\}$. Para encontrar la arista de peso mínimo entre X y Y es necesario analizar todas las aristas que inciden en x. En el peor caso, lo anterior se efectúa en $n - 1$ pasos, donde n es el número de vértices en V. Suponga que y se agrega a X. Es decir, suponga que $X = \{x, y\}$ y $Y = V - \{x, y\}$. Para encontrar la arista de menor peso entre X y Y parece que hay un problema: ¿es necesario analizar de nuevo las aristas que inciden en x? (Por supuesto, no es necesario volver a analizar la arista entre x y y porque ambas están en X.) Prim sugirió una forma inteligente para evitar este problema, preservando dos vectores.

Considere que hay n vértices, identificados como 1, 2, …, n. Sean dos vectores C_1 y C_2. Sean X el conjunto de vértices de árbol parcialmente construido en el algoritmo de Prim y $Y = V - X$. Sea i un vértice en Y. De todas las aristas que inciden sobre los vértices de X y el vértice i en Y, sea la arista (i, j), $j \in X$, la arista de menor peso. Los vectores C_1 y C_2 se usan para almacenar esta información. Sea $w(i, j)$ el peso de la arista (i, j). Así, en cualquier paso del algoritmo de Prim,

$$C_1(i) = j$$
y $$C_2(i) = w(i, j).$$

A continuación se demostrará que estos dos vectores pueden usarse para evitar el análisis repetido de las aristas. Sin pérdida de generalidad, inicialmente se supone $X = \{1\}$ y $Y = \{2, 3, \ldots, n\}$. Resulta evidente que para cada vértice i en Y, $C_1(i) = 1$ y $C_2(i) = w(i, 1)$ si la arista $(i, 1)$ existe. El menor $C_2(i)$ determina el siguiente vértice que ha de agregarse a X.

De nuevo puede suponerse que el vértice 2 se selecciona como el punto que ha de agregarse a X. Así, $X = \{1, 2\}$ y $Y = \{3, 4, \ldots, n\}$. El algoritmo de Prim requiere la determinación de la arista de menor peso entre X y Y. Pero, con ayuda de $C_1(i)$ y $C_2(i)$, ya no es necesario analizar las aristas incidentes sobre el vértice i. Suponga que i es un vértice en Y. Si $w(i, 2) < w(i, 1)$, $C_1(i)$ se cambia de 1 a 2 y $C_2(i)$ se cambia de $w(i, 1)$ a $w(i, 2)$. Si $w(i, 2) \geq w(i, 1)$, no se hace nada. Una vez que la actualización se ha completado para todos los vértices en Y, es posible escoger un vértice que ha de agregarse a X mediante el análisis de $C_2(i)$. El menor $C_2(i)$ determina el siguiente vértice que ha de agregarse. Como puede ver el lector, ahora se ha evitado exitosamente el análisis repetido de todas las aristas. Cada arista se examina una sola vez.

A continuación se proporciona el algoritmo de Prim con más detalle.

Algoritmo 3-3 □ Algoritmo de Prim para construir un árbol de expansión mínima

Input: Una gráfica no dirigida, conexa y ponderada $G = (V, E)$.

Output: Un árbol de expansión mínima de G.

Paso 1. Hacer $X = \{x\}$ y $Y = V - \{x\}$, donde x es cualquier vértice en V.

Paso 2. Asignar $C_1(y_j) = x$ y $C_2(y_j) = \infty$ para todo vértice y_j en V.

Paso 3. Para cada vértice y_j en V, se analiza si y_j está en Y y si la arista (x, y_j) existe. Si y_j está en Y, entonces la arista (x, y_j) existe y $w(x, y_j) = b < C_2(y_j)$, hacer $C_1(y_j) = x$ y $C_2(y_j) = b$; en caso contrario, no se hace nada.

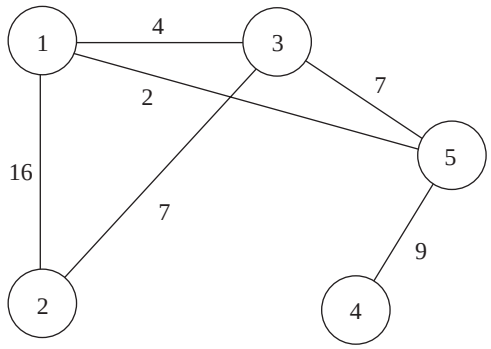
Paso 4. Hacer y un vértice en Y tal que $C_2(y)$ es mínimo. Hacer $z = C_1(y)$ (z debe estar en X). Conectar y en la arista (y, z) con z en el árbol parcialmente construido T. Hacer $X = X + \{y\}$ y $Y = Y - \{y\}$ y $C_2(y) = \infty$.

Paso 5. Si Y está vacío, se termina y el árbol resultante T es un árbol de expansión mínima; en caso contrario, se hace $x = y$ y se va al paso 3.

Ejemplo 3-4 Algoritmo de Prim

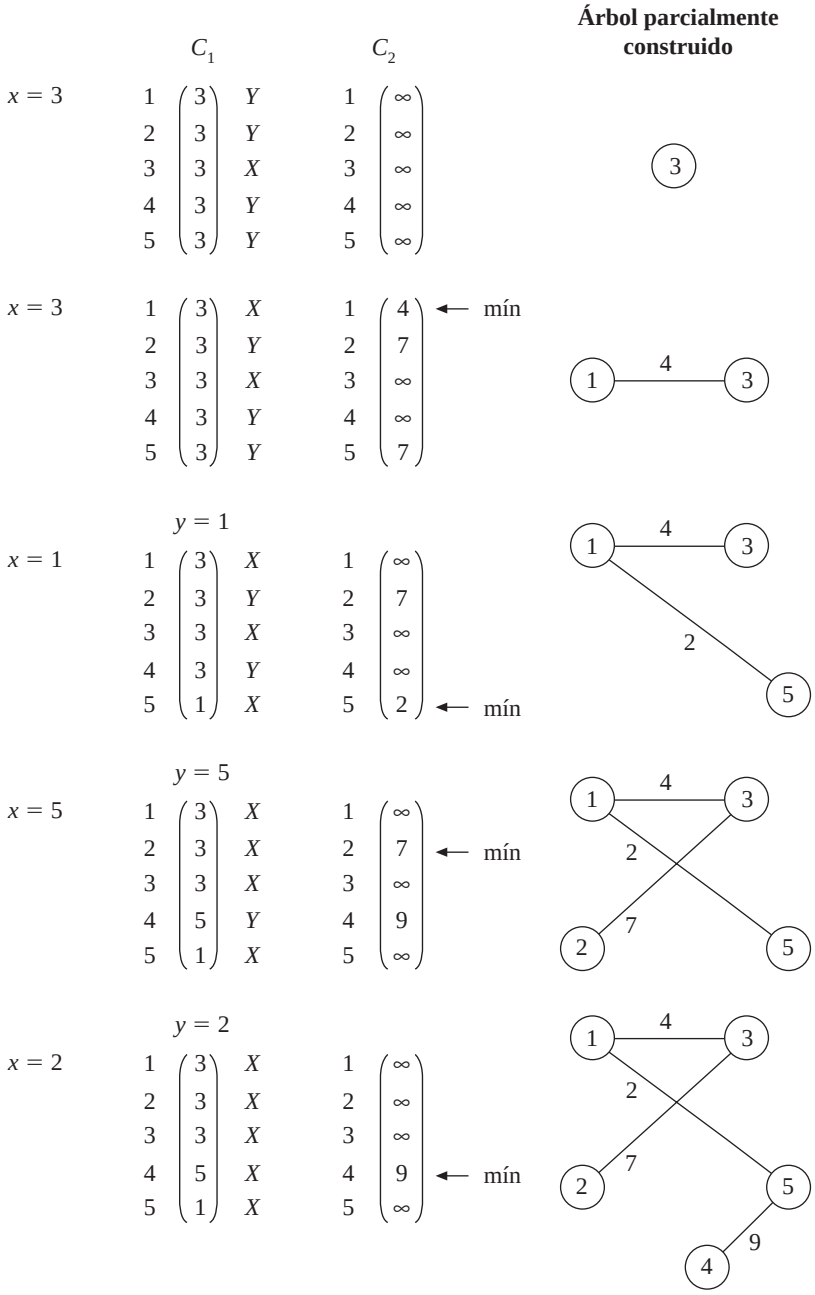
Se considerará la gráfica que se muestra en la figura 3-14.

FIGURA 3-14 Gráfica para demostrar el algoritmo de Prim.



Con la figura 3-15 se demostrará la forma en que funciona el algoritmo de Prim para generar un árbol de expansión mínima. Se supondrá que inicialmente se seleccionó el vértice 3.

FIGURA 3-15 Determinación de un árbol de expansión mínima aplicando el algoritmo de Prim y vértice inicial 3.



Para el algoritmo de Prim, siempre que un vértice se agrega al árbol parcialmente construido es necesario examinar cada elemento de C_1. En consecuencia, la complejidad temporal del algoritmo de Prim, tanto en el peor caso como en el caso promedio, es $O(n^2)$, donde n es el número de vértices en V. Observe que la complejidad temporal del algoritmo de Kruskal es $O(m \log m)$, donde m es el número de aristas en E. En caso de que m sea pequeño, es preferible el método de Kruskal. En el peor caso, ya se mencionó, m puede ser igual a $O(n^2)$ y la complejidad temporal del peor caso del algoritmo de Kruskal se vuelve $O(n^2 \log n)$, que es mayor que el del algoritmo de Prim.

3-3 EL PROBLEMA DE LA RUTA MÁS CORTA DE ORIGEN ÚNICO

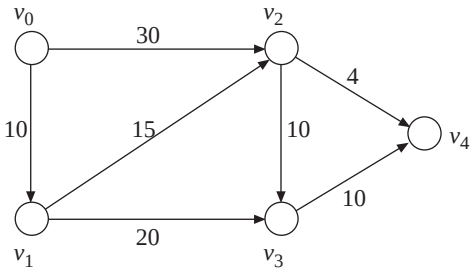
En el problema de la ruta más corta se cuenta con una gráfica dirigida $G = (V, E)$ donde cada arista está asociada con un peso no negativo. Este peso puede considerarse como la longitud de esta arista. La longitud de una ruta en G se define como la suma de las longitudes de las aristas en esta ruta. El problema de la ruta más corta de origen único consiste en encontrar todas las rutas más cortas desde un vértice fuente (o inicial) determinado, que se denota por v_0, a todos los demás vértices en V.

En esta sección se demostrará que dicho problema puede resolverse con el método codicioso. Debido a que este hecho fue indicado por Dijkstra, el algoritmo que se presentará se conoce como algoritmo de Dijkstra. La esencia de este algoritmo es bastante semejante a la del algoritmo del árbol de expansión mínima recientemente presentado. La idea básica es más bien simple: las rutas más cortas desde v_0 hasta todos los demás vértices se encuentran una por una. Primero se encuentra el vecino más próximo de v_0. Luego se encuentra el segundo vecino más próximo de v_0. Este proceso se repite hasta que se encuentra el n-ésimo más próximo de v_0, donde n es el número de vértices distintos a v_0 que hay en la gráfica.

Ejemplo 3-5 Problema de la ruta más corta de origen único

Considere la gráfica dirigida en la figura 3-16. Encuentre todas las rutas más cortas que salen de v_0.

FIGURA 3-16 Gráfica para demostrar el método de Dijkstra.



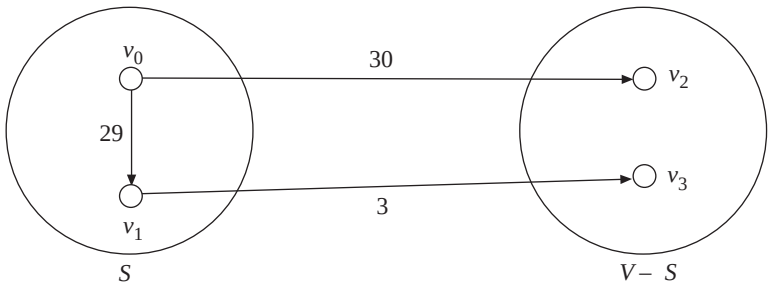
Este algoritmo determina primero que el vecino más próximo de v_0 es v_1. La ruta más corta de v_0 a v_1 es v_0v_1. Luego se determina que el segundo vecino más próximo de v_0 es v_2 y que su ruta correspondiente más corta es $v_0v_1v_2$. Observe que aunque esta ruta consta de sólo dos aristas, sigue siendo más corta que la ruta v_0v_2, que contiene sólo una arista. Se encuentra que los vecinos más próximos tercero y cuarto de v_0 son v_4 y v_3, respectivamente. Todo el proceso puede ilustrarse con la tabla 3-1.

TABLA 3-1 Ilustración para encontrar las rutas más cortas desde v_0.

i	***i*-ésimo vecino más próximo de v_0**	**Ruta más corta desde v_0 hasta el *i*-ésimo vecino más próximo (longitud)**
1	v_1	v_0v_1 (10)
2	v_2	$v_0v_1v_2$ (25)
3	v_4	$v_0v_1v_2v_4$ (29)
4	v_3	$v_0v_1v_3$ (30)

Así como ocurre en el problema del árbol de expansión mínima, en el algoritmo de Dijkstra también se dividen los conjuntos de vértices en dos conjuntos: S y $V - S$, donde S contiene a todos los i vecinos más próximos de v_0 que se han encontrado en los i primeros pasos. Así, en el $(i + 1)$-ésimo paso, la tarea es encontrar el $(i + 1)$-ésimo vecino más próximo de v_0. En este momento, es muy importante no llevar a cabo una acción incorrecta. Considere la figura 3-17. En esta figura se muestra que ya se ha encontrado el primer vecino más próximo de v_0, que es v_1. Puede parecer que como v_1v_3 es el vínculo más corto entre S y $V – S$, debería escogerse a v_1v_3 como la siguiente arista y a v_3 como el segundo vecino más próximo. Esto sería erróneo porque se tiene interés en encontrar las rutas más cortas que salen de v_0. Para este caso, el segundo vecino más próximo es v_2, no v_3.

FIGURA 3-17 Dos conjuntos de vértices, S y $V – S$.



A continuación se explicará un truco importante usado en el algoritmo de Dijkstra para encontrar de manera eficiente el siguiente vecino más próximo de v_0. Esto puede explicarse si se considera la figura 3-16. $L(v_i)$ denota la distancia más corta de v_0 a v_1 que se ha encontrado hasta el momento. Desde el principio, $S = \{v_0\}$ y se tiene

$$L(v_1) = 10$$
y $$L(v_2) = 30$$

ya que v_1 y v_2 están unidos a v_0.

Debido a que $L(v_1)$ es la ruta más corta, v_1 es el primer vecino más próximo de v_0. Sea $S = \{v_0, v_1\}$. Así, sólo v_2 y v_3 están unidos con S. Para v_2, su $L(v_2)$ previo era igual a 30. No obstante, después que v_1 se coloca en S, es posible usar la ruta $v_0v_1v_2$ cuya longitud es $10 + 15 = 25 < 30$. Así, en cuanto corresponde a v_2, su $L(v_2)'$ se calcula como:

$$\begin{aligned} L(v_2)' &= \text{mín}\{L(v_2), L(v_1) + \text{longitud de } v_1v_2\} \\ &= \text{mín}\{30,\ 10 + 15\} \\ &= 25. \end{aligned}$$

El análisis anterior muestra que la distancia más corta de v_0 a v_2 encontrada hasta el momento puede no ser suficientemente corta debido al vértice que acaba de agregarse. Si ocurre esta situación, es necesario actualizar la distancia más corta.

Sea u el último vértice que se ha agregado a S. Sea $L(w)$ la distancia más corta de v_0 a w que se ha encontrado hasta el momento. Sea $c(u, w)$ la longitud de la arista que une u y w. Así, es necesario actualizar $L(w)$ según la siguiente fórmula:

$$L(w) = \text{mín}(L(w), L(u) + c(u, w)).$$

A continuación se resume el algoritmo de Dijkstra para resolver el problema de la ruta más corta de origen único.

Algoritmo 3-4 □ Algoritmo de Dijkstra para generar rutas más cortas de origen único

Input: Una gráfica dirigida $G = (V, E)$ y un vértice fuente v_0. Para cada arista $(u, v) \in E$, hay un número no negativo $c(u, v)$ asociado con ésta. $|V| = n + 1$.

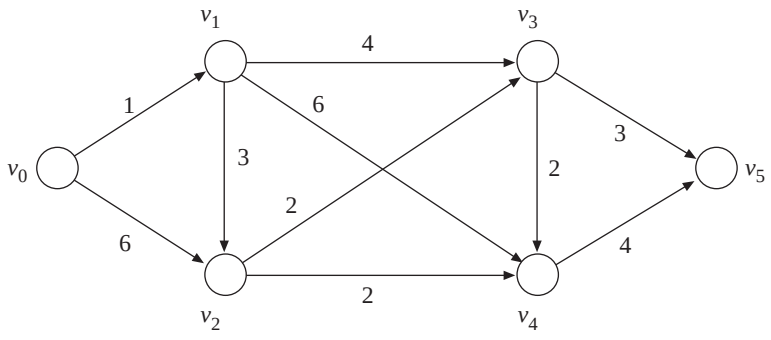
Output: Para cada $v \in V$, la longitud de una ruta más corta de v_0 a v.

$S := \{v_0\}$
For $i := 1$ to n *do*
Begin
  If $(v_0, v_i) \in E$ then
    $L(v_i) := c(v_0, v_i)$
  else
    $L(v_i) := \infty$
End
For $i := 1$ to n do
Begin
  Elegir u de $V - S$ de modo que $L(u)$ sea el menor
  $S := S \cup \{u\}$ (* colocar u en S*)
  Para todo w en $V - S$ do
    $L(w) := \text{mín}(L(w), L(u) + c(u, w))$
End

Ejemplo 3-6 Algoritmo de Dijkstra

Considere la gráfica dirigida que se muestra en la figura 3-18.

FIGURA 3-18 Gráfica dirigida ponderada.



El algoritmo de Dijkstra procederá de la siguiente forma:

1. $S = \{v_0\}$

   $L(v_1) = 1$

   $L(v_2) = 6$

   Todas las demás $L(v_i)$ son iguales a ∞.

   $L(v_1)$ es la más pequeña. v_0v_1 es la ruta más corta de v_0 a v_1.

   $S = \{v_0, v_1\}$

2. $L(v_2) = \text{mín}(6, L(v_1) + c(v_1, v_2))$

   $= \text{mín}(6, 1 + 3)$

   $= 4$

   $L(v_3) = \text{mín}(\infty, L(v_1) + c(v_1, v_3))$

   $= 1 + 4$

   $= 5$

   $L(v_4) = \text{mín}(\infty, L(v_1) + c(v_1, v_4))$

   $= 1 + 6$

   $= 7$

   $L(v_2)$ es la más pequeña. $v_0v_1v_2$ es la ruta más corta de v_0 a v_2.

   $S = \{v_0, v_1, v_2\}$

3. $L(v_3) = \text{mín}(5, L(v_2) + c(v_2, v_3))$

   $= \text{mín}(5, 4 + 2)$

   $= 5$

   $L(v_4) = \text{mín}(7, L(v_2) + c(v_2, v_4))$

   $= \text{mín}(7, 4 + 2)$

   $= 6$

   $L(v_3)$ es la más pequeña. $v_0v_1v_3$ es la ruta más corta de v_0 a v_3.

   $S = \{v_0, v_1, v_2, v_3\}$

4. $L(v_4) = \text{mín}(6, L(v_3) + c(v_3, v_4))$

   $= \text{mín}(6, 5 + 2)$

   $= 6$

   $L(v_5) = \text{mín}(\infty, L(v_3)\ c(v_3, v_5))$

   $= 5 + 3$

   $= 8$

$L(v_4)$ es la más pequeña. $v_0v_1v_2v_4$ es la ruta más corta de v_0 a v_4.

$S = \{v_0, v_1, v_2, v_3, v_4\}$

5. $L(v_5) = \text{mín}(8, L(v_4)\ c(v_4, v_5))$
   $= \text{mín}(8, 6 + 4)$
   $= 8$

$v_0v_1v_3v_5$ es la ruta más corta de v_0 a v_5.

En la tabla 3-2 se resume la salida.

TABLA 3-2 Rutas más cortas desde el vértice v_0.

Vértice	Distancia más corta a v_0 (longitud)
v_1	v_0v_1 (1)
v_2	$v_0v_1v_2$ (1 + 3 = 4)
v_3	$v_0v_1v_3$ (1 + 4 = 5)
v_4	$v_0v_1v_2v_4$ (1 + 3 + 2 = 6)
v_5	$v_0v_1v_3v_5$ (1 + 4 + 3 = 8)

Complejidad temporal del algoritmo de Dijkstra

Es fácil ver que la complejidad temporal del peor caso del algoritmo de Dijkstra es $O(n^2)$ debido a las operaciones repetidas para calcular $L(w)$. Por otro lado, el número mínimo de pasos para resolver el problema de la ruta más corta de fuente única es $\Omega(e)$, donde e es el número de aristas en la gráfica porque es necesario examinar cada arista. En el peor caso, $\Omega(e) = \Omega(n^2)$. Por consiguiente, en este sentido el algoritmo de Dijkstra es óptimo.

3-4 Problema de mezclar 2 listas (2-way merge)

Se cuenta con dos listas ordenadas L_1 y L_2, $L_1 = (a_1, a_2, \ldots, a_{n_1})$ y $L_2 = (b_1, b_2, \ldots, b_{n_2})$. L_1 y L_2 pueden fusionarse en una lista ordenada si se aplica el algoritmo de mezcla lineal que se describe a continuación.

Algoritmo 3-5 □ Algoritmo de mezcla lineal

Input: Dos listas ordenadas, $L_1 = (a_1, a_2, \ldots, a_{n_1})$ y $L_2 = (b_1, b_2, \ldots, b_{n_2})$.

Output: Una lista ordenada que consta de los elementos en L_1 y L_2.

Begin
  $i := 1$
  $j := 1$
do
  if $a_i > b_j$ then output b_j y $j := j + 1$
  else output a_j e $i := i + 1$
while ($i \leq n_1$ y $j \leq n_2$)
  if $i > n_1$ then output $b_j, b_{j+1}, \ldots, b_{n_2}$,
  else output $a_i, a_{i+1}, \ldots, a_{n_1}$.
End.

Puede verse fácilmente que el número de comparaciones requeridas es $m + n - 1$ en el peor caso. Cuando m y n son iguales, puede demostrarse que el número de comparaciones para el algoritmo de mezcla lineal es óptimo. Si se requiere mezclar más de dos listas ordenadas, el algoritmo de mezcla lineal es óptimo y puede seguir aplicándose, ya que mezcla dos listas ordenadas, repetidamente. Estos procesos de mezcla se denominan mezcla de 2 listas porque cada paso de mezcla sólo mezcla dos listas ordenadas. Suponga que se tienen tres listas ordenadas L_1, L_2 y L_3 que constan de 50, 30 y 10 elementos, respectivamente. Así, es posible mezclar L_1 y L_2 para obtener L_4. Este paso de mezcla requiere $50 + 30 - 1 = 79$ comparaciones en el peor caso. Luego se mezclan L_4 y L_3 usando $80 + 10 - 1 = 89$ comparaciones. El número de comparaciones necesarias en esta secuencia de mezclas es 168. En forma alterna, primero pueden mezclarse L_2 y L_3 y luego L_1. El número de comparaciones necesarias es sólo 128. Hay muchas sucesiones de mezcla diferentes que requieren números de comparaciones distintos. Por ahora el interés lo constituye el siguiente problema: hay m listas ordenadas. Cada una de ellas consta de n_i elementos. ¿Cuál es la sucesión óptima del proceso de mezcla para mezclar estas listas ordenadas aplicando el número mínimo de comparaciones?

En lo que sigue, para simplificar el análisis se usará $n + m$, en vez de $n + m - 1$, para indicar el número de comparaciones necesarias para mezclar dos listas de tamaños n y m, respectivamente, ya que evidentemente esto no afecta el diseño del algoritmo. Se considerará un ejemplo en el que se tiene $(L_1, L_2, L_3, L_4, L_5)$ con tamaños (20, 5, 8, 7, 4). Imagine que estas listas se mezclan como se muestra a continuación:

L_1 se mezcla con L_2 para obtener Z_1 con $20 + 5 = 25$ comparaciones
Z_1 se mezcla con L_3 para obtener Z_2 con $25 + 8 = 33$ comparaciones
Z_2 se mezcla con L_4 para obtener Z_3 con $33 + 7 = 40$ comparaciones
Z_3 se mezcla con L_5 para obtener Z_4 con $40 + 4 = 44$ comparaciones
Total $= 142$ comparaciones.

El patrón de mezcla puede representarse mediante un árbol binario como se muestra en la figura 3-19*a*).

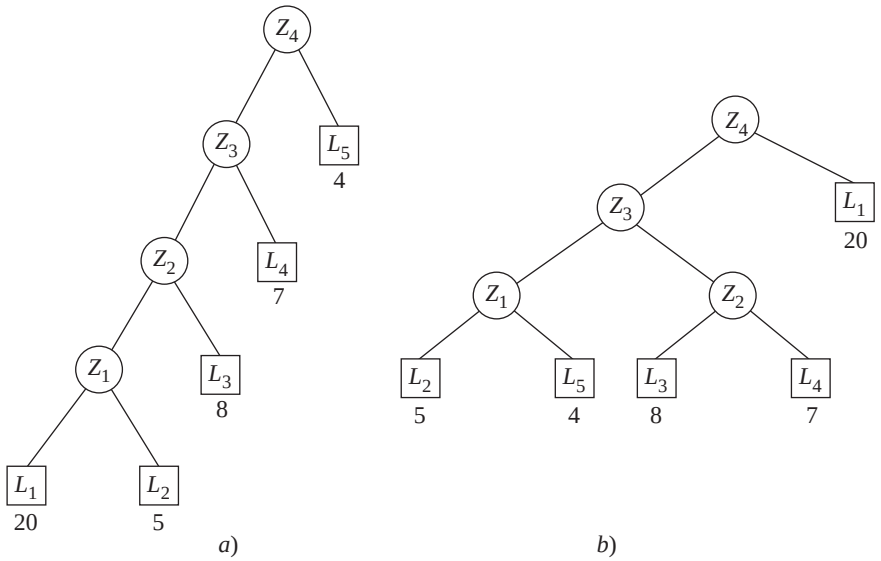
Sea d_i la profundidad de un nodo hoja del árbol binario. Sea n_i el tamaño de la lista L_i asociada con este nodo hoja. Así, puede verse fácilmente que el número total de comparaciones correspondiente a este proceso de mezcla es $\sum_{i=1}^{5} d_i n_i$. En nuestro caso, $d_1 = d_2 = 4$, $d_3 = 3$, $d_4 = 2$ y $d_5 = 1$. Entonces, el número total de comparaciones necesarias puede calcularse como $4 \cdot 20 + 4 \cdot 5 + 3 \cdot 8 + 2 \cdot 7 + 1 \cdot 4 = 80 + 20 + 24 + 14 + 4 = 142$, que es correcto.

Suponga que se usa un método codicioso en el que siempre se mezclan dos listas presentes más cortas. Entonces el patrón de mezcla es

L_2 se mezcla con L_5 para obtener Z_1 con $5 + 4 = 9$ comparaciones
L_3 se mezcla con L_4 para obtener Z_2 con $8 + 7 = 15$ comparaciones
Z_1 se mezcla con Z_2 para obtener Z_3 con $9 + 15 = 24$ comparaciones
Z_3 se mezcla con L_1 para obtener Z_4 con $24 + 20 = 44$ comparaciones
Total $= 92$ comparaciones.

El proceso de mezcla anterior se muestra como un árbol binario en la figura 3-19*b*).

FIGURA 3-19 Secuencias de mezcla distintas.



De nuevo es posible aplicar la fórmula $\sum_{i=1}^{5} d_i n_i$. En este caso, $d_1 = 1$ y $d_2 = d_3 = d_4 = d_5 = 3$. Puede calcularse que el número total de comparaciones es $1 \cdot 20 + 3 \cdot (5 + 4 + 8 + 7) = 20 + 3 \cdot 24 = 92$, que es menor que el cálculo correspondiente a la figura 3-19*a*).

A continuación se presenta un método codicioso para encontrar una mezcla óptima de 2 listas:

Algoritmo 3-6 □ Un algoritmo codicioso para generar un árbol óptimo de mezcla de 2 listas

Input: m listas ordenadas, L_i, $i = 1, 2, \ldots, m$, donde cada lista L_i consta de n_i elementos.

Output: Un árbol de mezcla de 2 listas óptimo.

Paso 1. Generar m árboles, donde cada árbol tiene exactamente un nodo (nodo externo) con peso n_i.

Paso 2. Elegir dos árboles T_1 y T_2 con pesos mínimos.

Paso 3. Crear un nuevo árbol T cuya raíz tenga a T_1 y T_2 como sus subárboles y cuyo peso sea igual a la suma de los pesos de T_1 y T_2.

Paso 4. Reemplazar T_1 y T_2 por T.

Paso 5. Si sólo queda un árbol, stop y return; en caso contrario, ir al paso 2.

Ejemplo 3-7

Se tienen seis listas ordenadas de longitudes 2, 3, 5, 7, 11 y 13. Encontrar un árbol binario extendido con la longitud de ruta ponderada mínima para estas listas.

Primero se mezclan 2 y 3 y se busca la solución del problema para mezclar 5 listas ordenadas con longitudes 5, 5, 7, 11 y 13. Luego se mezclan 5 y 5, y así sucesivamente. La secuencia de mezcla se muestra en la figura 3-20.

FIGURA 3-20 Secuencia de mezcla óptima de 2 listas.

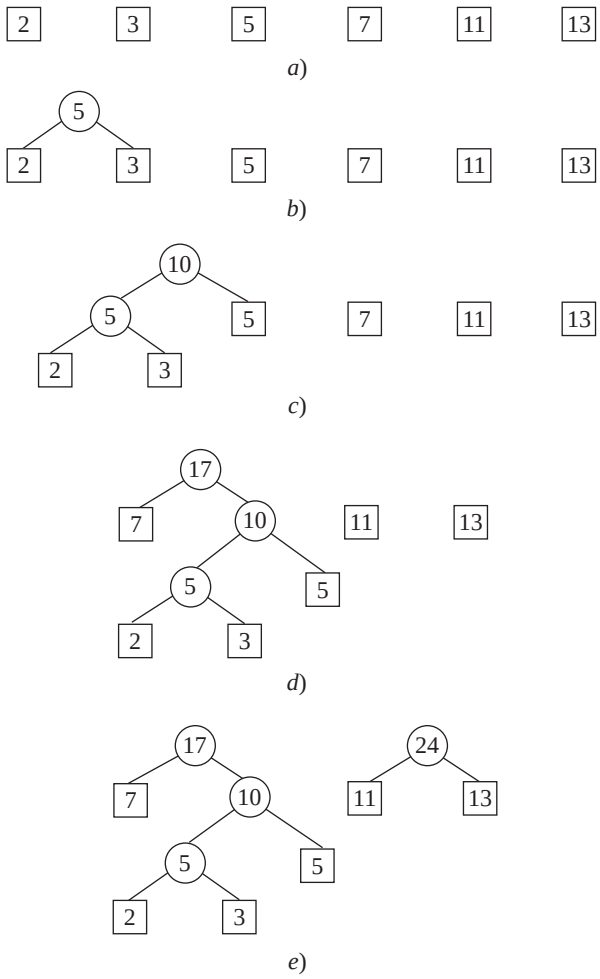
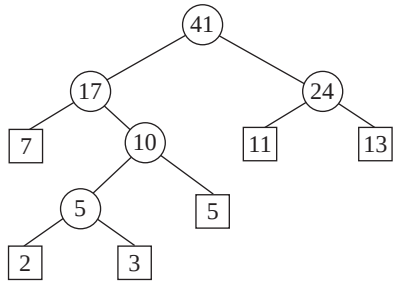


FIGURA 3-20 (*Continuación*).



f)

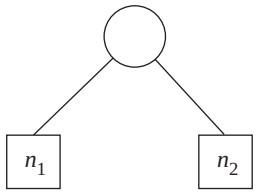
Para demostrar que el algoritmo codicioso anterior es correcto, primero se prueba que existe un árbol de mezcla de 2 listas óptimo donde los dos nodos hoja de tamaños mínimos se asignan a dos hermanos. Sea *A* un nodo interno situado a la distancia máxima desde la raíz. Por ejemplo, el nodo identificado con 5 en la figura 3-20 es tal nodo. Los hijos de *A*, por ejemplo L_i y L_j, deben ser nodos hoja. Suponga que los tamaños de los hijos de *A* son n_i y n_j. Si n_i y n_j no son los más pequeños, es posible intercambiar L_i y L_j con los dos nodos más pequeños sin incrementar los pesos del árbol de mezcla de 2 listas. Así, se obtiene este árbol donde los dos nodos hoja de menor tamaño se asignan como hermanos.

Con base en el análisis anterior, puede suponerse que *T* es un árbol óptimo de mezcla de 2 listas para $L_1, L_2, \ldots, L_m$ con longitudes $n_1, n_2, \ldots, n_m$, respectivamente, y sin pérdida de generalidad, $n_1 \leq n_2 \ldots \leq n_m$, donde las dos listas de menor longitud, a saber, L_1 y L_2, son hermanos. Sea *A* el padre de L_1 y L_2. Sea T_1 el árbol en que *A* se sustituye por una lista de longitud $n_1 + n_2$. Sea $W(X)$ el peso de un árbol de mezcla de 2 listas. Entonces se tiene

$$W(T) = W(T_1) + n_1 + n_2. \tag{3-1}$$

Ahora es posible demostrar por inducción que el algoritmo codicioso es correcto. Resulta evidente que este algoritmo produce un árbol de mezcla de 2 listas óptimo para $m = 2$. Luego, se supone que el algoritmo produce un árbol de mezcla de 2 listas óptimo para $m - 1$ listas. Para el caso del problema que implique *m* listas $L_1, L_2, \ldots, L_m$, se combinan las dos primeras listas, a saber, L_1 y L_2. Luego se aplica el algoritmo a este caso del problema con $m - 1$ listas. Sea T_2 el árbol de mezcla de 2 listas óptimo producido por el algoritmo. En T_2 hay un nodo hoja de longitud $n_1 + n_2$. Este nodo se separa de modo que tenga dos hijos, a saber, L_1 y L_2 con longitudes n_1 y n_2, respectivamente, como se muestra en la figura 3-21. Este árbol de nueva creación se denota por T_3. Se tiene

FIGURA 3-21 Un subárbol.



$$W(T_3) = W(T_2) + n_1 + n_2. \quad \textbf{(3-2)}$$

Se afirma que T_3 es un árbol de mezcla de 2 listas óptimo para $L_1, L_2, \ldots, L_m$. Suponga lo contrario. Entonces

$$W(T_3) > W(T),$$

que implica

$$W(T_2) > W(T_1).$$

Esto es imposible, ya que por la hipótesis de inducción T_2 es un árbol de mezcla de 2 listas óptimo para $m - 1$ listas.

Complejidad temporal del algoritmo codicioso para generar un árbol de mezcla de 2 listas óptimo

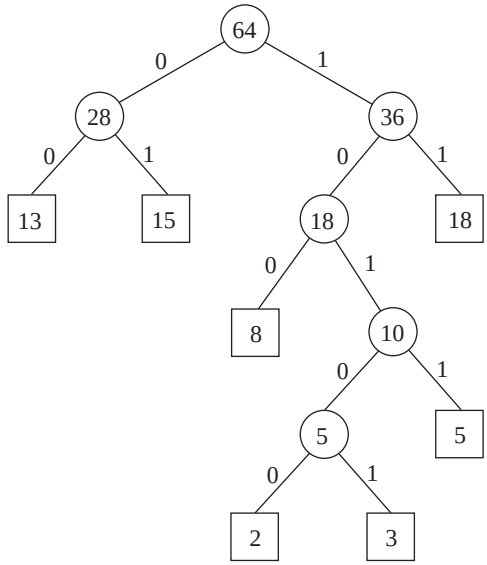
Para los m números dados $n_1, n_2, \ldots, n_m$, es posible construir un mini heap a fin de representar estos números donde el valor de la raíz es menor que los valores de sus hijos. Así, la reconstrucción del árbol después de eliminar la raíz, que tiene el menor valor, puede realizarse en un tiempo $O(\log n)$. Y la inserción de un nuevo nodo en un mini heap también puede efectuarse en un tiempo $O(\log n)$. Debido a que el ciclo principal se lleva a cabo $n - 1$ veces, el tiempo total para generar un árbol binario extendido óptimo es $O(n \log n)$.

Ejemplo 3-8 Códigos de Huffman

Considere un problema de telecomunicaciones donde se quiere representar un conjunto de mensajes por medio de una secuencia de ceros y unos. En consecuencia, para enviar un mensaje simplemente se transmite una secuencia de ceros y unos. Una aplicación del árbol binario extendido con la longitud de ruta externa ponderada óptima es la generación de un conjunto óptimo de códigos; es decir, cadenas binarias, para estos

mensajes. Se supondrá que hay siete mensajes cuyas frecuencias de acceso son 2, 3, 5, 8, 13, 15 y 18. Para minimizar los costos de transmisión y decodificación, pueden usarse cadenas cortas para representar mensajes de uso frecuente. Luego es posible construir un árbol binario óptimo extendido al mezclar primero 2 y 3 y luego 5 y 5, y así sucesivamente. El árbol binario extendido se muestra en la figura 3-22.

FIGURA 3-22 Árbol de código de Huffman.



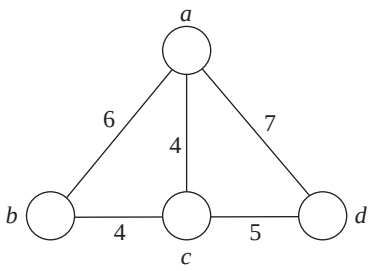
Así, los códigos correspondientes a los mensajes con frecuencias 2, 3, 5, 8, 13, 15 y 18 son 10100, 10101, 1011, 100, 00, 01 y 11, respectivamente. Los mensajes usados más a menudo se codifican mediante códigos cortos.

3-5 EL PROBLEMA DEL CICLO BASE MÍNIMO (MINIMUM CYCLE BASIS) RESUELTO CON EL ALGORITMO CODICIOSO

En esta sección se presentará el problema del ciclo base mínimo y también se demostrará cómo este problema puede resolverse con el algoritmo codicioso.

Considere la figura 3-23. En la gráfica no dirigida de la figura 3-23 hay tres ciclos; específicamente son: {*ab*, *bc*, *ca*}, {*ac*, *cd*, *da*} y {*ab*, *bc*, *cd*, *da*}.

FIGURA 3-23 Gráfica que contiene ciclos.



Mediante una operación idónea, dos ciclos pueden combinarse en otro ciclo. Esta operación es la operación de suma de anillos que se define como sigue: sean A y B dos ciclos. Entonces $C = A \oplus B = (A \cup B) - (A \cap B)$ puede ser un ciclo. Para los ciclos en el caso anterior, sea

$$A_1 = \{ab, bc, ca\}$$
$$A_2 = \{ac, cd, da\}$$
y $$A_3 = \{ab, bc, cd, da\}.$$

Puede demostrarse fácilmente que

$$A_3 = A_1 \oplus A_2$$
$$A_2 = A_1 \oplus A_3$$
y $$A_1 = A_2 \oplus A_3.$$

Este ejemplo muestra que $\{A_1, A_2\}$, $\{A_1, A_3\}$ y $\{A_2, A_3\}$ pueden ser considerados como ciclos base para la gráfica de la figura 3-23 porque para cada uno de ellos, todos los ciclos de esta gráfica pueden ser generados usando ésta. Formalmente, *un ciclo base de una gráfica es un conjunto de ciclos tales que cada ciclo en la gráfica puede generarse aplicando la operación suma de anillos a algunos ciclos de esta base*.

Se asume que cada arista está asociada con un peso. El peso de un ciclo es el peso total de todas las aristas en él. *Éste es el peso total de todos los ciclos del ciclo base*. El problema del ciclo base ponderado se define como sigue: *dada una gráfica, encontrar un ciclo base mínimo para esta gráfica*. Para la gráfica de la figura 3-23, el ciclo base mínimo es $\{A_1, A_2\}$, porque tiene el menor peso.

El método codicioso que resuelve el problema del ciclo base ponderado se apoya en los siguientes conceptos:

1. Es posible determinar el tamaño del ciclo base mínimo. Sea K este tamaño.

2. Suponga que es posible determinar todos los ciclos. (La determinación de todos los ciclos en una gráfica no es nada fácil. Sin embargo, es irrelevante para este problema, por lo que la técnica no se abordará aquí.) Todos los ciclos se clasifican en una sucesión no decreciente según sus pesos.
3. A partir de la sucesión no ordenada de los ciclos, al ciclo base se suman los ciclos uno por uno. Para cada ciclo sumado se comprueba si es una combinación lineal de algunos ciclos que ya existen en el ciclo base parcialmente construido. En caso afirmativo, se elimina el ciclo en cuestión.
4. Detener el proceso si el ciclo base tiene K ciclos.

Nuestro método codicioso tiene un típico enfoque codicioso. Suponga que se quiere encontrar un conjunto de objetos al minimizar algún parámetro. Algunas veces es posible determinar el tamaño mínimo K de este conjunto y después aplicar el método codicioso para agregar objetos uno por uno a este conjunto hasta que el tamaño del conjunto es igual a K. En la siguiente sección se presentará otro ejemplo con el mismo enfoque del método codicioso.

Ahora regresaremos al problema original. Primero se demostrará que es posible determinar el tamaño de un ciclo base mínimo. No se proporcionará una demostración formal del método. En vez de ello, de manera informal el concepto se ilustrará mediante un ejemplo. Considere la figura 3-24. Suponga que se construye un árbol de expansión de esta gráfica como se muestra en la figura 3-25*a*). Este árbol de expansión carece de ciclo. No obstante, si al árbol de expansión se le agrega una arista, como se muestra en la figura 3-25*b*), entonces se forma un ciclo.

De hecho, como se muestra en la figura 3-25, cada adición de una arista crea un nuevo ciclo independiente. El número de ciclos independientes es igual al número de aristas que es posible agregar al árbol de expansión. Ya que el número de aristas en un árbol de expansión es $|V| - 1$, entonces el total de aristas que puede agregarse es igual a

$$|E| - (|V| - 1) = |E| - |V| + 1.$$

FIGURA 3-24 Gráfica que muestra la dimensión de un ciclo base mínimo.

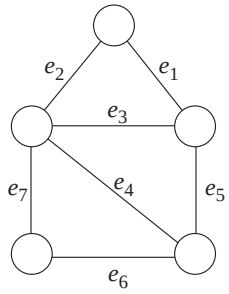
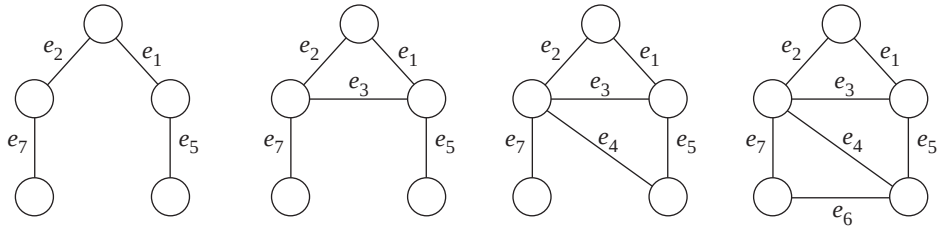


FIGURA 3-25 Relación entre un árbol de expansión y los ciclos.

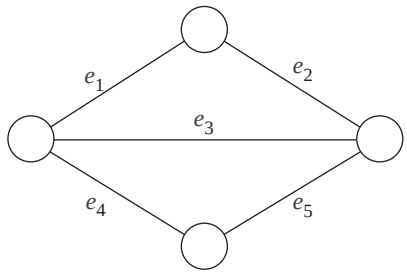


Se ha demostrado la fórmula para encontrar el tamaño de un ciclo base mínimo. Lo único que queda por hacer es demostrar cómo determinar si un ciclo es una combinación lineal de un conjunto de ciclos. Sin embargo, no se pretende describir un método formal. Sólo se describen las técnicas a través de un ejemplo. Para realizar la descripción de las técnicas, los ciclos se representan con una matriz de incidencia. Cada renglón corresponde a un ciclo y cada columna corresponde a una arista. La comprobación de dependencia, o independencia, puede hacerse por eliminación Gaussiana, excepto cuando una operación implica dos renglones en una operación de suma de anillos (u or-exclusivo). A continuación esto se ilustra con un ejemplo. Considere la figura 3-26. Hay tres ciclos: $C_1 = \{e_1, e_2, e_3\}$, $C_2 = \{e_3, e_4, e_5\}$ y $C_3 = \{e_1, e_2, e_5, e_4\}$.

La matriz que representa los dos primeros ciclos se muestra a continuación:

$$\begin{array}{c} \\ C_1 \\ C_2 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & & \\ & & 1 & 1 & 1 \end{bmatrix} \end{array}$$

FIGURA 3-26 Gráfica que ilustra la comprobación de independencia de los ciclos.



Si se agrega C_3, la matriz se convierte en:

$$\begin{array}{c} \\ C_1 \\ C_2 \\ C_3 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & & \\ & & 1 & 1 & 1 \\ 1 & 1 & & 1 & 1 \end{bmatrix} \end{array}$$

La operación o excluyente sobre el renglón 1 y el renglón 3 produce la siguiente matriz:

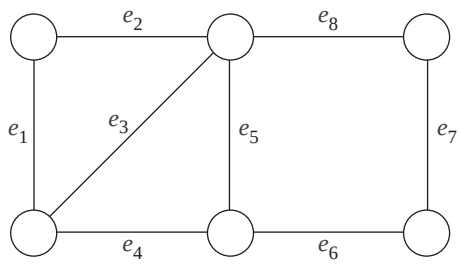
$$\begin{array}{c} \\ C_1 \\ C_2 \\ C_3 \end{array} \begin{array}{c} \begin{array}{ccccc} e_1 & e_2 & e_3 & e_4 & e_5 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & & \\ & & 1 & 1 & 1 \\ & & 1 & 1 & 1 \end{bmatrix} \end{array}$$

La operación excluyente sobre C_2 y C_3 produce un renglón vacío que muestra que C_3 es una combinación lineal de C_1 y C_2.

A continuación se completa el análisis mediante la consideración de la figura 3-27. Se supone que el peso de cada arista es 1. Hay seis ciclos que se muestran a continuación:

$C_1 = \{e_1, e_2, e_3\}$
$C_2 = \{e_3, e_5, e_4\}$
$C_3 = \{e_2, e_5, e_4, e_1\}$
$C_4 = \{e_8, e_7, e_6, e_5\}$
$C_5 = \{e_3, e_8, e_7, e_6, e_4\}$
$C_6 = \{e_2, e_8, e_7, e_6, e_4, e_1\}$.

FIGURA 3-27 Gráfica que ilustra el proceso de cálculo del ciclo base mínimo.



Para este caso, $|E| = 8$ y $|V| = 6$. Así, $K = 8 - 6 + 1 = 3$. Este método codicioso funciona como sigue:

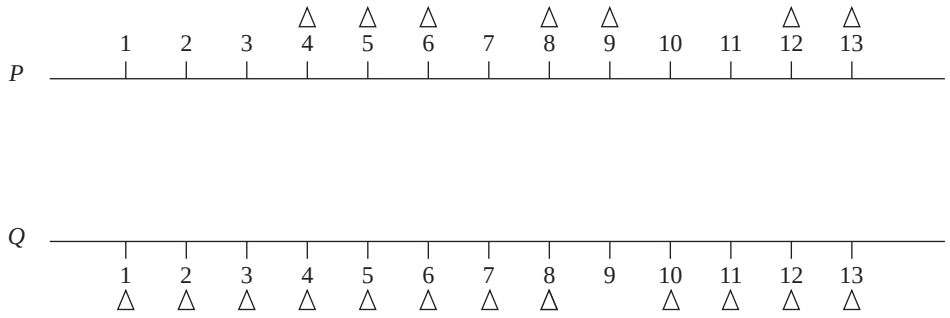
1. Se agrega C_1.
2. Se agrega C_2.
3. Se agrega C_3 y se elimina porque se encuentra que C_3 es una combinación lineal de C_1 y C_2.
4. Se agrega C_4. Debido a que ahora el ciclo base ya tiene tres ciclos, el proceso se detiene cuando $K = 3$. Se encuentra que el ciclo base mínimo es $\{C_1, C_2, C_4\}$.

3-6 Solución por el método codicioso del problema de 2 terminales (uno a cualquiera/uno a muchos)

En el diseño VLSI existe un problema de direccionamiento de un canal. Hay muchas versiones de este problema y en el capítulo 5 también se presentará uno de estos problemas que puede resolverse con el método del algoritmo *A**. El problema de direccionamiento de un canal que se analizará en esta sección es una versión muy simplificada del problema general de direccionamiento de un canal. La razón por la que este problema se denomina de 2 terminales-uno a cualquiera se clarificará en el siguiente párrafo.

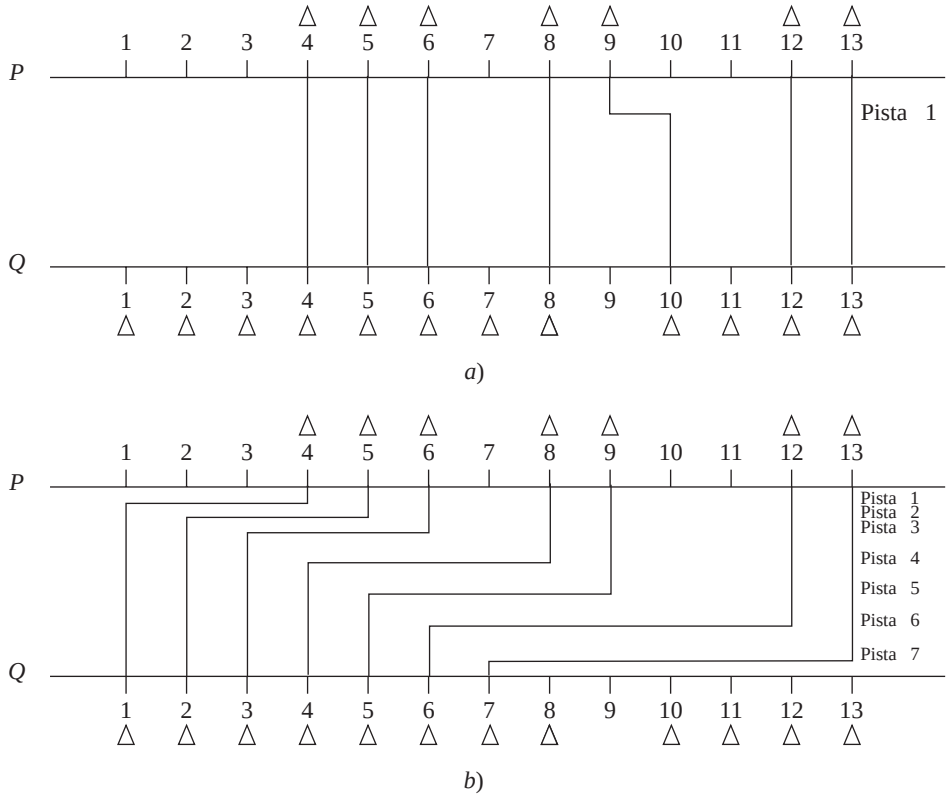
Considere la figura 3-28, donde se han marcado algunas terminales. Cada terminal marcada de un renglón superior debe conectarse o unirse con una terminal de un renglón inferior de manera uno a uno. Se requiere que ningún par de líneas se corte. Además, todas las líneas son verticales u horizontales. Toda línea horizontal corresponde a una pista (track).

FIGURA 3-28 Problema de 2 terminales (uno a cualquiera).



Puede haber muchas soluciones para el mismo problema. En la figura 3-29 se muestran dos soluciones factibles para el caso del problema de la figura 3-28. Puede verse que la solución de la figura 3-29*a*) usa menos pistas que la de la figura 3-28*b*).

FIGURA 3-29 Dos soluciones factibles para el ejemplo del problema en la figura 3-24.

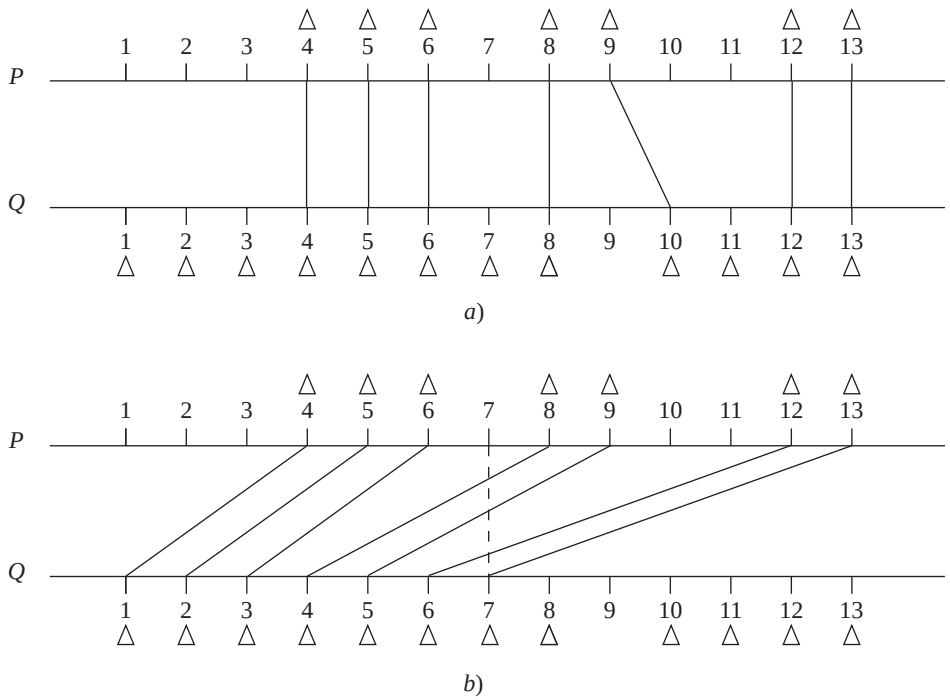


Para un problema real de direccionamiento de un canal, por supuesto que sería deseable minimizar el número de pistas. En esta versión simplificada sólo se intenta minimizar la densidad de una solución.

Para explicar el significado de "densidad", las dos soluciones vuelven a trazarse (figura 3-30) en la figura 3-29. Ahora el lector debe imaginar que para explorar de izquierda a derecha se utiliza una recta vertical. Esta recta vertical "cortará" las líneas de la solución. En cada punto, la densidad local de la solución es el número de líneas que corta la recta vertical.

Por ejemplo, como se muestra en la figura 3-30*b*), la recta vertical (identificada por la línea punteada o discontinua) corta cuatro líneas en la terminal 7. Así, la densidad local en la terminal 7 de la solución en la figura 3-30*b*) es 4. La densidad de una solución es la máxima densidad local. Puede verse fácilmente que las densidades de las soluciones en la figura 3-30*a*) y 3-30*b*) son 1 y 4, respectivamente.

FIGURA 3-30 Figura 3-29 vuelta a trazar incorporando la recta de búsqueda.



¿Por qué se usa la densidad como el parámetro que se intenta minimizar? La respuesta es sencilla: así se simplifica el problema. Si como parámetro se usa el número de pistas, es más difícil resolver el problema. Por otra parte, la densidad es una cota inferior del número de pistas. En consecuencia, es posible usarla como indicador de cuán buena es la solución.

Para encontrar una solución con la densidad mínima se usa el mismo truco que en la última sección, cuando se intentó resolver el problema del ciclo base mínimo. Es decir, se tiene un método para determinar la densidad mínima de un caso de un problema. Una vez que se ha determinado ésta, puede aplicarse el método codicioso para encontrar una solución con esta densidad mínima. No se analizará cómo se determina esta densidad mínima, ya que es demasiado complicado e irrelevante para el análisis del método codicioso. La parte fundamental de este análisis es demostrar que nuestro método codicioso siempre encuentra tal solución con densidad mínima.

A continuación se ilustrará la forma en que funciona el método codicioso. Se nos proporciona un problema ejemplo donde ya se ha determinado la densidad mínima. Para el ejemplo del problema que se muestra en la figura 3-28, la densidad mínima es 1.

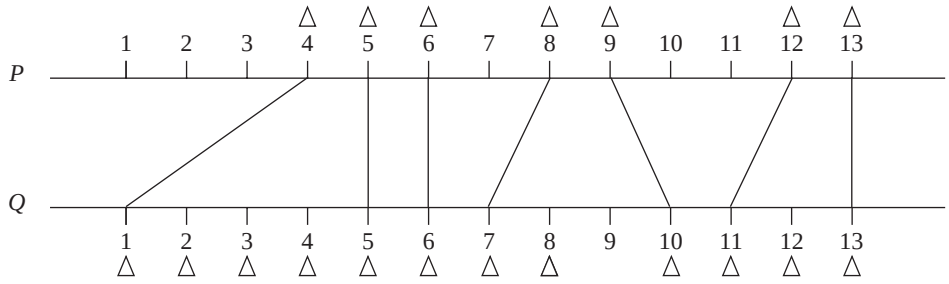
Sean $P_1, P_2, \ldots, P_n$ las terminales del renglón (que representa la pista) superior y $Q_1, Q_2, \ldots, Q_m$, $m > n$, las terminales del renglón inferior que pueden unirse. También se supone que todas las P_i y las Q_i están identificadas de izquierda a derecha. Es decir, si $j > i$, entonces $P_j(Q_j)$ está a la derecha de $P_i(Q_i)$.

Dadas la densidad mínima d y las notaciones anteriores, nuestro método codicioso procede como sigue:

1. P_1 es conectada con Q_1.
2. Después que P_i se une con, por ejemplo, Q_j, se comprueba si P_{i+1} puede ser conectada a Q_{j+1}. Si la adición de la recta que une P_{i+1} con Q_{j+1} incrementa la densidad a $d + 1$, entonces se intenta conectar P_{i+1} con Q_{j+2}.
3. El paso anterior se repite hasta que todas las P_i están conectadas.

A continuación se ilustrará cómo funciona nuestro método codicioso al aplicarlo al caso del problema en la figura 3-28. Primero se decide que $d = 1$. Con esta información, las terminales se unen como en la figura 3-31. Se observa que P_5 no puede unirse con Q_2, Q_3 y Q_4 porque las uniones incrementarían la densidad a 2, lo cual excede la densidad mínima. De manera semejante y exactamente por la misma razón, no es posible unir P_9 con Q_8.

FIGURA 3-31 Caso del problema de la figura 3-28 resuelto con el método codicioso.



Observe que nuestro algoritmo jamás produce alguna solución cuando los segmentos de recta que unen terminales se cortan entre sí, como en la figura 3-32. En realidad, puede verse fácilmente que jamás se tendrá una unión así porque esta unión puede transformarse en otra unión con una densidad menor o igual, como se muestra en la figura 3-33.

FIGURA 3-32 Intersección cruzada.

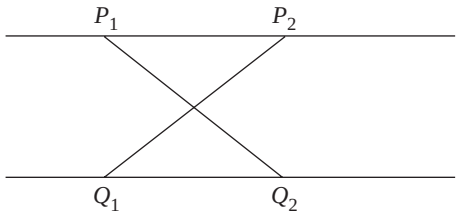
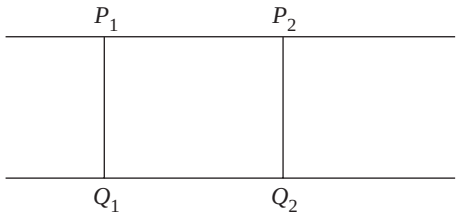


FIGURA 3-33 Interconexión transformada a partir de las intersecciones de la figura 3-33.



Finalmente se mostrará cómo funciona nuestro método codicioso. Debido a que se supone que la densidad mínima es d, entonces existe una solución factible S_1, ((P_1, Q_{j_1}), (P_2, Q_{j_2}), …, (P_n, Q_{j_n})), con densidad d. Se demostrará que esta solución puede transformarse en la solución S_2, ((P_2, Q_{i_1}), (P_2, Q_{i_2}), …, (P_n, Q_{i_n})), obtenida con nuestro método codicioso. Esto se demostrará por inducción.

Supongamos que las k primeras uniones en S_1 pueden transformarse en las k primeras uniones en S_2 sin violar el requerimiento de la densidad d. Luego se demostrará que esta transformación puede hacerse para $k = k + 1$. Esta hipótesis es trivialmente cierta para $k = 1$. Si esto se cumple para k, entonces se tiene una solución parcial ((P_1, Q_{i_1}), (P_2, Q_{i_2}), …, (P_k, Q_{i_k})), sin violar el requerimiento de la densidad d. Considere P_{k+1}. P_{k+1} está unido con $Q_{j_{k+1}}$ en S_1. Suponga que existe una terminal $Q_{j_{k+1}}$ a la izquierda de $Q_{j_{k+1}}$ con la que es posible unir con P_{k+1} sin violar el requerimiento de la densidad mínima, entonces P_{k+1} puede unirse con esta terminal y esto es lo que se obtiene al usar nuestro método codicioso. Si ocurre lo contrario, entonces nuestro método codicioso también une P_{k+1} con $Q_{j_{k+1}}$.

El análisis anterior muestra que cualquier solución factible puede transformarse en una solución obtenida al aplicar el método codicioso y, en consecuencia, éste funciona.

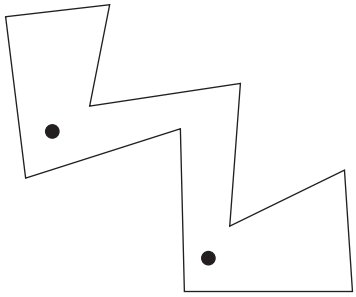
En este último ejemplo en que se utiliza el método del algoritmo codicioso, se determina el valor mínimo del parámetro que se intenta optimizar y luego el método co-

dicioso se aplica directamente para alcanzar esta meta. En el problema del ciclo base, los ciclos se agregaban uno por uno hasta que se obtenía el tamaño mínimo del ciclo base. En este problema primero se calcula la densidad mínima. El algoritmo codicioso une codiciosamente las terminales, y en cualquier instante nuestro algoritmo asegura que la densidad resultante no excede a *d*.

3-7 EL PROBLEMA DEL MÍNIMO DE GUARDIAS COOPERATIVOS PARA POLÍGONOS DE 1-ESPIRAL RESUELTO POR EL MÉTODO CODICIOSO

El problema del mínimo de guardias cooperativos es una variante del problema de la galería de arte, que se define como sigue: se tiene un polígono, que representa la galería de arte, y se requiere colocar el número mínimo de guardias en el polígono de modo que cada punto del polígono sea visible por lo menos para un guardia. Por ejemplo, considere la figura 3-34. Para este polígono, el número mínimo de guardias necesarios es 2. El problema de la galería de arte es un problema NP-difícil (NP-hard).

FIGURA 3-34 Una solución del problema de la galería de arte.

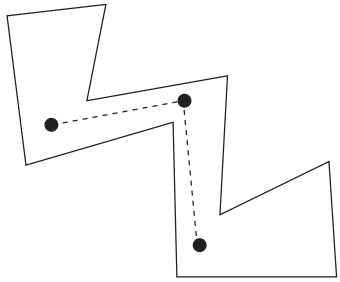


El problema del mínimo de guardias cooperativos agrega más restricciones al problema de la galería de arte. Observe que puede resultar extremadamente peligroso que un guardia esté en una parte de la galería de arte si los otros guardias no pueden verlo. La relación entre los guardias se representa mediante una gráfica de visibilidad *G* de los guardias. En *G*, cada guardia se representa por un vértice y entre dos vértices hay una arista si y sólo si los guardias correspondientes pueden verse mutuamente. Además de encontrar un conjunto de guardias que puedan ver el polígono dado, también se requiere que la gráfica de visibilidad de estos guardias sea conexa. En otras palabras, se requiere que ningún guardia esté aislado y que entre cada par de guardias haya una ruta. Este problema se denomina problema del *mínimo de guardias cooperativos*.

Considere nuevamente la figura 3-34. La gráfica de visibilidad correspondiente a los dos guardias es evidentemente un conjunto de dos vértices que están aislados. Para

cumplir el requerimiento del problema del mínimo de guardias cooperativos es necesario agregar otro guardia, lo cual se muestra en la figura 3-35.

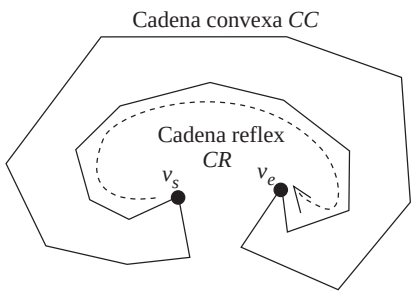
FIGURA 3-35 Una solución del problema del mínimo de guardias cooperativos de la figura 3-34.



De nuevo puede demostrarse que el problema del mínimo de guardias cooperativos es NP-difícil. Por lo tanto, es bastante improbable que este problema del mínimo de guardias cooperativos pueda resolverse con cualquier algoritmo poligonal sobre polígonos generales. Sin embargo, se demostrará que existe un algoritmo codicioso para este problema para polígonos de 1-espiral.

Antes de definir este tipo de polígonos, primero se definen las cadenas reflex (convexas). *Una cadena reflex* (*convexa*), *denotada por CR* (*CC*), *de un polígono simple es una cadena de aristas de este polígono si todos los vértices en esta cadena son reflex* (*convexos*) *con respecto al interior del polígono*.* También se estipula que una cadena reflex se refiere a una cadena reflex *máxima*; es decir, que no está contenida en ninguna otra cadena reflex. Un *polígono P de 1-espiral* es un polígono simple cuya frontera puede partirse en una cadena reflex y una cadena convexa. En la figura 3-36 se muestra un polígono de 1-espiral típico.

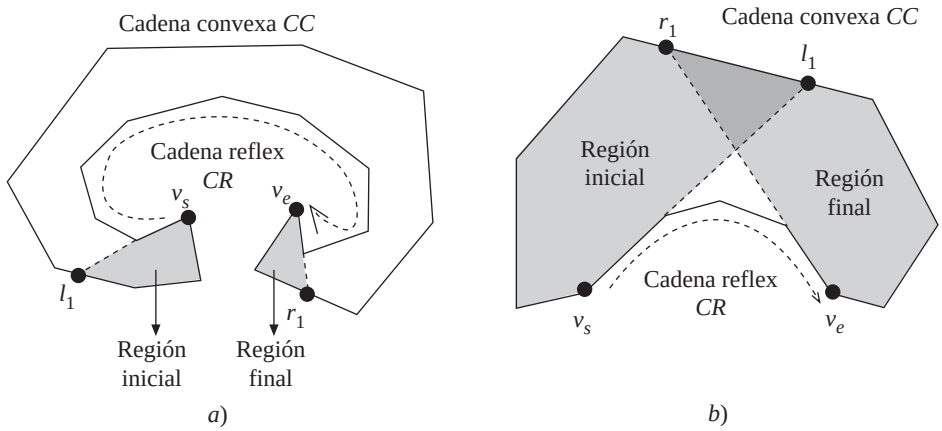
FIGURA 3-36 Un polígono de 1-espiral típico.



* En esta sección los autores utilizan el mismo texto para definir a dos términos (cadena reflex y cadena convexa). *(N. del R.T.)*

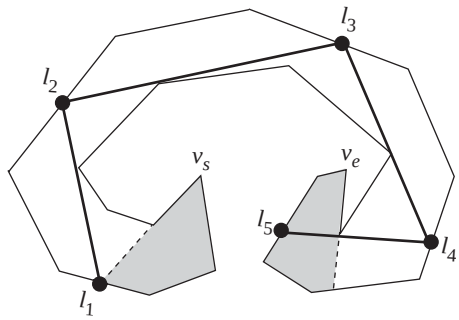
Cuando la frontera de un polígono de 1-espiral se recorre en sentido contrario al movimiento de las manecillas del reloj, el vértice inicial (o final) de la cadena reflex se denomina $v_s(v_e)$. Dados v_s y v_e, ahora es posible definir dos regiones, denominadas región inicial y región final. Se trazará una recta, empezando en $v_s(v_e)$ a lo largo de la primera (última) arista de la cadena reflex hasta que toca la frontera del polígono en $l_1(r_1)$. Este segmento de recta $\overline{v_s l_1}(\overline{v_e r_1})$ y la primera (última) parte de la cadena convexa que empieza en $v_s(v_e)$ forma una región que se denomina *región inicial* (*final*). En la figura 3-37 se muestran dos ejemplos. Observe que las regiones inicial y final pueden traslaparse, por lo que ahí se requieren dos guardias estacionados: uno en la región inicial y otro en la región final.

FIGURA 3-37 Las regiones inicial y final en polígonos de 1-espiral.



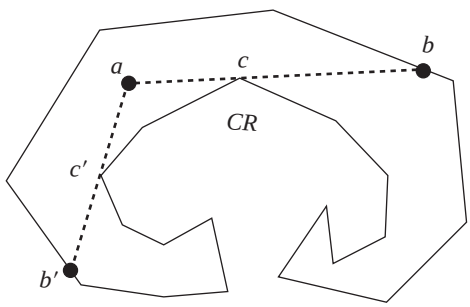
Nuestro algoritmo codicioso resuelve el problema del mínimo de guardias cooperativos para polígonos de 1-espiral como sigue. Primero se ubica un guardia en l_1. Luego se plantea la pregunta: ¿qué área puede observar este guardia? Esto puede contestarse trazando una tangente de *CR* empezando en l_1 hasta que toque un punto l_2 en *CC*. Se coloca un guardia en l_2. Esto es bastante razonable. Observe que si a la izquierda de $\overline{l_1 l_2}$ no hay guardia, entonces la gráfica de visibilidad resultante no es conexa. Este proceso se repite hasta que se alcanza la región final. En la figura 3-38 se muestra un caso típico. En esa figura, los guardias estacionados en l_1, l_2, l_3, l_4 y l_5 constituyen una solución óptima del problema del mínimo de guardias cooperativos.

FIGURA 3-38 Un conjunto de guardias $\{l_1, l_2, l_3, l_4, l_5\}$ en un polígono de 1-espiral.



Antes de proporcionar el algoritmo formal es necesario definir algunos nuevos conceptos. Considere la figura 3-39. Sea a un punto en un polígono de 1-espiral P. Es posible trazar dos tangentes con respecto a CR desde a. Si el exterior de CR está completamente a la derecha (izquierda) de una tangente trazada desde a, se denomina la *tangente izquierda* (*derecha*) de a con respecto a CR. A continuación se trazará la tangente izquierda (derecha) de a con respecto a CR hasta tocar la frontera de P en $b(b')$. Así, $\overline{ab}(\overline{ab'})$ se denomina *segmento de recta de apoyo izquierdo* (*derecho*) con respecto a a. Esto se ilustra en la figura 3-39. Si un punto a está en la región inicial (final), entonces el segmento de recta de apoyo derecho (izquierdo) con respecto a a se define como $\overline{av_s}(\overline{av_e})$.

FIGURA 3-39 Los segmentos de recta de apoyo izquierdo y derecho con respecto a a.



A continuación se presenta un algoritmo para resolver el problema del mínimo de guardias cooperativos para polígonos de 1-espiral.

Algoritmo 3-7 □ Un algoritmo para resolver el problema del mínimo de guardias cooperativos para polígonos de 1-espiral

Input: Un polígono de 1-espiral P.

Output: Un conjunto de puntos que es la solución del problema del mínimo de guardias cooperativos.

Paso 1. Buscar la cadena reflex *CR* y la cadena convexa *CC* de *P*.

Paso 2. Buscar los puntos de intersección l_1 y r_1 de *CC* con las rectas dirigidas que empiezan desde v_s y v_e pasando por las aristas primera y última de *CR*, respectivamente.

Paso 3. Hacer $k = 1$.

Mientras l_k no está en la región final, hacer (*inicio de ciclo*)

Dibujar la tangente izquierda de l_k con respecto a *CR* hasta que toca *CC* en l_{k+1}. ($\overline{l_k\, l_{k+1}}$ es un segmento de recta de apoyo izquierdo con respecto a l_k.)
Hacer $k = k + 1$.

End While. (*fin de ciclo*)

Paso 4. Report $\{l_1, l_2, \ldots, l_k\}$.

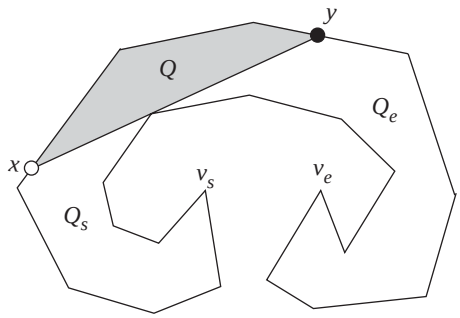
A continuación se establecerá que el algoritmo 3-7 es correcto. Primero se presentan algunas notaciones y términos. Una subcadena de la frontera del polígono *P* desde el punto *a* hasta el punto *b* en sentido contrario al movimiento de las manecillas del reloj se denota por *C*[*a*, *b*].

Suponga que *A* es un conjunto de guardias cooperativos que pueden ver todo el polígono simple *P* y que la gráfica de visibilidad de *A* es conexa, entonces *A* se denomina solución factible del problema del mínimo de guardias cooperativos para *P*. Observe que *A* no necesariamente es mínima.

Sean los puntos *x* y *y* que están en la cadena convexa de *P* y $\overline{xy}$ un segmento de recta de apoyo. $\overline{xy}$ parte a *P* en tres subpolígonos, *Q*, Q_s y Q_e. *Q* es el subpolígono acotado por *C*[*y*, *x*] y $\overline{xy}$, aunque es exclusiva de *x* o *y*. Q_s y Q_e son los otros dos subpolígonos que contienen a v_s y v_e, respectivamente. En la figura 3-40 se ilustra esta situación.

Suponga que *A* es una solución factible del problema del mínimo de guardias cooperativos para *P*. Entonces, debe haber por lo menos un guardia de *A* situado dentro de *Q*; en caso contrario, la gráfica de visibilidad de *A* no es conexa porque dos guardias cualesquiera ubicados dentro de Q_s y Q_e, respectivamente, no pueden verse el uno al otro.

FIGURA 3-40 Segmento de recta de apoyo $\overline{xy}$ y las regiones Q, Q_s y Q_e.



Sean P un polígono de 1-espiral y $\{l_1, l_2, \ldots, l_k\}$ el conjunto de puntos resultante del algoritmo 3-7 aplicado a P. Sea L_i la región acotada por $C[l_i, l_{i-1}]$ y $\overline{l_{i-1}l_i}$, aunque es exclusiva de l_{i-1}, para $1 \leq i \leq k$. Sea A cualquier solución factible del problema del mínimo de guardias cooperativos para P. Entonces, hay por lo menos un guardia de A ubicado en cada L_i, para $1 \leq i \leq k$. Además, resulta evidente que los L_i son ajenos. Sea N el tamaño de A; entonces, $N \geq k$. Esto significa que k es el número mínimo de guardias en cualquier solución factible del problema del mínimo de guardias cooperativos para P.

Después de establecer la característica de minimalidad de $\{l_1, l_2, \ldots, l_k\}$, a continuación se establecerá la visibilidad de $\{l_1, l_2, \ldots, l_k\}$. Es decir, debe demostrarse que cada punto en P es visible por lo menos desde un guardia en $\{l_1, l_2, \ldots, l_k\}$. Puede verse que una colección observa el polígono de 1-espiral si y sólo si pueden ver las aristas de la cadena reflex. Sea $\overline{l_i l_{i+1}}$ el segmento de recta que entra en contacto con la cadena reflex en c_i, para $1 \leq i \leq k-1$, y sean $c_0 = v_s$ y $c_k = v_e$. Resulta evidente que l_i puede ver a $C[c_i, c_{i-1}]$, para $1 \leq i \leq k-1$. Así, $\{l_1, l_2, \ldots, l_k\}$ puede ver toda la cadena reflex $C[v_s, v_e]$ y en consecuencia puede ver todo el polígono de 1-espiral. Además, resulta evidente que la gráfica de visibilidad de $\{l_1, l_2, \ldots, l_k\}$ es conexa según el algoritmo 3-7 en sí. Esto significa que la salida del algoritmo 3-7 es una solución factible del problema del mínimo de guardias cooperativos. Debido a que el número de guardias es mínimo, se concluye que el algoritmo 3-7 produce una solución óptima.

Así como se hizo para el análisis de la complejidad temporal del algoritmo 3-7, sea n el número de vértices de un polígono de 1-espiral. Los pasos 1 y 2 pueden realizarse en tiempo $O(n)$ mediante una exploración de la frontera de P. Para el paso 3 pueden efectuarse una exploración lineal en la cadena reflex en sentido contrario al movimiento de las manecillas del reloj y una exploración lineal en la cadena convexa en el sentido del movimiento de las manecillas del reloj para encontrar todos los segmentos de recta de apoyo izquierdos. El paso 3 también puede llevarse a cabo en tiempo $O(n)$. Así, el algoritmo 3-7 es lineal.

3-8 LOS RESULTADOS EXPERIMENTALES

Para mostrar la potencia de la estrategia del método codicioso, el algoritmo de Prim para árboles generadores mínimos se implementó en una PC IBM con lenguaje C. También se implementó un método directo que generaría todos los conjuntos posibles de $(n - 1)$ aristas de una gráfica $G = (V, E)$. Si estas aristas constituyen un ciclo, se ignoran; en caso contrario, se calcula la longitud total de este árbol de expansión.

El árbol de expansión mínima se encuentra luego que se han analizado todos los árboles generadores. Este método directo también fue implementado en lenguaje C. Cada conjunto de datos corresponde a una gráfica $G = (V, E)$ generada de manera aleatoria. En la tabla 3-3 se resumen los resultados experimentales. Resulta evidente que el problema del árbol de expansión mínima no puede resolverse con el método directo y que el algoritmo de Prim es bastante eficaz.

TABLA 3-3 Resultados experimentales al probar la eficacia del algoritmo de Prim.

Tiempo de ejecución (segundos) (Promedio de 20 veces)							
$\|V\|$	$\|E\|$	**Directo**	**Prim**	$\|V\|$	$\|E\|$	**Directo**	**Prim**
10	10	0.305	0.014	50	200	–	1.209
10	20	11.464	0.016	50	250	–	1.264
10	30	17.629	0.022	60	120	–	1.648
15	30	9738.883	0.041	60	180	–	1.868
20	40	–	0.077	60	240	–	1.978
20	60	–	0.101	60	300	–	2.033
20	80	–	0.113	70	140	–	2.527
20	100	–	0.118	70	210	–	2.857
30	60	–	0.250	70	280	–	2.967
30	90	–	0.275	70	350	–	3.132
30	120	–	0.319	80	160	–	3.791
30	150	–	0.352	80	240	–	4.176
40	80	–	0.541	80	320	–	4.341
40	120	–	0.607	80	400	–	4.560
40	160	–	0.646	90	180	–	5.275
40	200	–	0.698	90	270	–	5.714
50	100	–	1.030	90	360	–	6.154
50	150	–	1.099	90	450	–	6.264

3-9 Notas y referencias

Para más análisis sobre el método codicioso, consulte Horowitz y Sahni (1978) y Papadimitriou y Steiglitz (1982). Korte y Louasz (1984) introdujeron un marco de referencia estructural para el método codicioso.

El algoritmo de Kruskal y el algoritmo de Prim para árboles de expansión pueden consultarse en Kruskal (1956) y Prim (1957), respectivamente. El algoritmo de Dijkstra apareció originalmente en Dijkstra (1959).

El algoritmo codicioso que genera árboles de mezcla óptimos apareció primero en Huffman (1952). Schwartz (1964) proporcionó un algoritmo para producir el conjunto de código Huffman. El algoritmo para encontrar el ciclo base mínimo se debe a Horton (1987).

El método codicioso para resolver el problema de 2-terminales de una a muchas asignaciones de canales fue propuesto por Atallah y Hambrusch (1986). El algoritmo lineal para resolver el problema del mínimo de guardias cooperativos para un polígono de 1-espiral fue proporcionado por Liaw y Lee (1994).

3-10 Bibliografía adicional

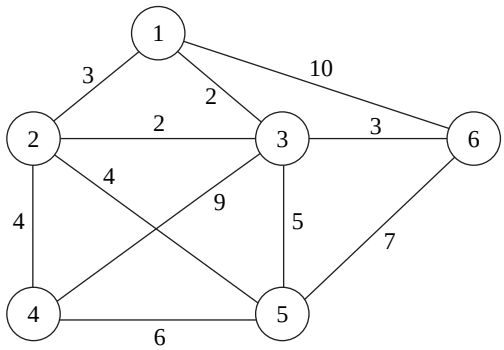
Aunque el concepto de método codicioso fue descubierto hace mucho tiempo, aún siguen publicándose muchos artículos interesantes para analizarlo. Los siguientes artículos se recomiendan ampliamente para el lector interesado en profundizar la investigación del método codicioso: Bein y Brucker (1986); Bein, Brucker y Tamir (1985); Blot, Fernandez de la Vega, Paschos y Saad (1995); Coffman, Langston y Langston (1984); Cunningham (1985); El-Zahar y Rival (1985); Faigle (1985); Fernandez-Baca y Williams (1991); Frieze, McDiarmid y Reed (1990); Hoffman (1988), y Rival y Zaguia (1987).

El método codicioso, por supuesto, puede usarse para aproximar algoritmos (capítulo 9 de este libro). Los siguientes artículos constituyen buen material de consulta: Gonzalez y Lee (1987); Levcopoulos y Lingas (1987), y Tarhio y Ukkonen (1988).

Para conocer algunos artículos bastante interesantes de reciente aparición, consulte Akutsu, Miyano y Kuhara (2003); Ando, Fujishige y Naitoh (1995); Bekesi, Galambos, Pferschy y Woeginger (1997); Bhagavathi, Grosch y Olariu (1994); Cidon, Kutten, Mansour y Peleg (1995); Coffman, Langston y Langston (1984); Cowureur y Bresler (2000); Csur y Kao (2001); Erlebach y Jansen (1999); Gorodkin, Lyngso y Stormo (2001); Gudmundsson, Levcopoulos y Narasimhan (2002); Hashimoto y Barrera (2003); Iwamura (1993); Jorma y Ukkonen (1988); Krogh, Brown, Mian, Sjolander y Haussler (1994); Maggs y Sitaraman (1999); Petr (1996); Slavik (1997); Tarhio y Ukkonen (1986); Tarhio y Ukkonen (1988); Tomasz (1998); Tsai, Tang y Chen (1994); Tsai, Tang y Chen (1996), y Zhang, Schwartz, Wagner y Miller (2003).

Ejercicios

3.1 Use el algoritmo de Kruskal para encontrar un árbol de expansión mínima de la gráfica siguiente.



3.2 Use el algoritmo de Prim para encontrar un árbol de expansión mínima de la gráfica en el problema 3.1.

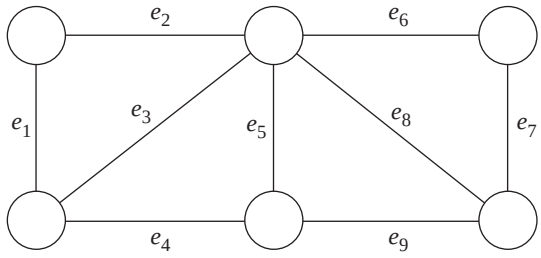
3.3 Demuestre que el algoritmo de Dijkstra es correcto.

3.4 *a*) Demuestre por qué el algoritmo de Dijkstra no funciona correctamente cuando la gráfica en consideración contiene aristas de costo negativo.

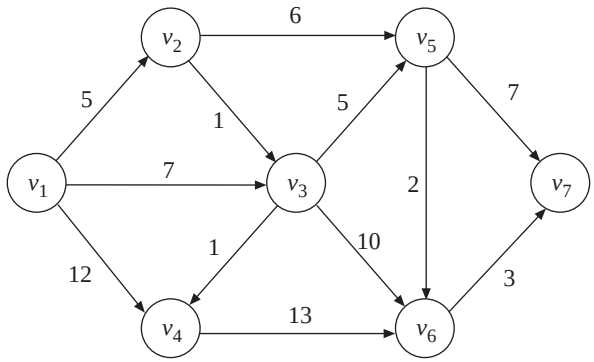
b) Modifique el algoritmo de Dijkstra de modo que sea capaz de calcular la ruta más corta desde el nodo fuente hasta cada nodo en una gráfica arbitraria con aristas de costo negativo, pero sin ciclos negativos.

3.5 Obtenga un conjunto de códigos Huffman óptimo para los ocho mensajes $(M_1, M_2, \ldots, M_8)$ con frecuencias de acceso $(q_1, q_2, \ldots, q_8) = (5, 10, 2, 6, 3, 7, 12, 14)$. Trace el árbol decodificador para este conjunto de códigos.

3.6 Obtenga un ciclo base mínimo para la gráfica siguiente.



3.7 Use el algoritmo de Dijkstra para encontrar las rutas más cortas de V_1 a todos los demás nodos.



3.8 Proporcione un método codicioso, con su heurística, para resolver el problema 0/1 de la mochila y también proporcione un ejemplo para mostrar que lo anterior no siempre conduce a una solución óptima.

3.9 Implemente el algoritmo de Prim y el algoritmo de Kruskal. Realice experimentos para comparar los desempeños de cada algoritmo.

3.10 Lea el apartado 12-4 de la obra de Papadimitriou y Steiglitz (1982), cuyo tema es la relación entre matroides y el algoritmo codicioso.

3.11 El problema de la mochila se define como sigue:

Dados enteros positivos $P_1, P_2, \ldots, P_n, W_1, W_2, \ldots, W_n$ y M. Encontrar $X_1, X_2, \ldots, X_n, 0 \leq X_i \leq 1$ tales que

$$\sum_{i=1}^{n} P_i X_i$$

sea maximizada bajo la restricción

$$\sum_{i=1}^{n} W_i X_i \leq M.$$

Proporcione un método codicioso para encontrar una solución óptima del problema de la mochila y demuestre que es correcto.

3.12 Considere el problema de programar n trabajos en una máquina. Describa un algoritmo para encontrar un programa de modo que su tiempo promedio para completarse sea mínimo. Demuestre que su algoritmo es correcto.

3.13 El algoritmo de Sollin fue propuesto primero por Boruvka (1926) para encontrar un árbol de expansión mínima en una gráfica conexa G. Inicialmente cada vértice en G se considera como un árbol de nodo único y no se selecciona ninguna arista. En cada paso se selecciona una arista de costo mínimo e para cada árbol T de modo que e tenga exactamente un vértice en T. En caso de ser necesario, se eliminan las copias de las aristas elegidas. El algoritmo termina si sólo se obtiene un árbol o si se seleccionan todas las aristas. Demuestre que el algoritmo es correcto y encuentre el número máximo de pasos del algoritmo.

capítulo

4

La estrategia divide-y-vencerás

Esta estrategia constituye un poderoso paradigma para definir algoritmos eficientes. Este método primero divide un problema en dos subproblemas más pequeños de modo que cada subproblema sea idéntico al problema original, excepto porque su tamaño de entrada es menor. Luego, ambos subproblemas se resuelven y las subsoluciones se fusionan en la solución final.

Una cuestión muy importante sobre esta estrategia es que divide el problema original en dos subproblemas idénticos al problema original. Por lo tanto, también estos dos subproblemas pueden resolverse aplicando la estrategia divide-y-vencerás. O, para decirlo con otras palabras, se resuelven de manera recurrente o recursiva.

Consideremos el sencillo problema de encontrar el máximo (número mayor) de un conjunto S de n números. La estrategia divide-y-vencerás resolvería este problema al dividir la entrada en dos conjuntos, cada uno con $n/2$ números. Sean S_1 y S_2 estos dos conjuntos. Luego se encuentra el máximo de S_1 y S_2, respectivamente. Sea X_i, $i = 1, 2$, el máximo de S_i. Así, el máximo de S puede encontrarse al comparar X_1 y X_2. El que sea mayor de éstos será el máximo de S.

En el análisis anterior casualmente se mencionó que es necesario encontrar el máximo X_i. Pero, ¿cómo se encuentra X_i? Nuevamente puede aplicarse la estrategia divide-y-vencerás. Es decir, S_i se divide en dos subconjuntos, se encuentran los máximos de estos subconjuntos y después de fusionan los resultados.

En general, un algoritmo divide-y-vencerás consta de tres pasos.

Paso 1. Si el tamaño del problema es pequeño, éste se resuelve aplicando algún método directo; en caso contrario, el problema original se separa en dos subproblemas, de preferencia del mismo tamaño.

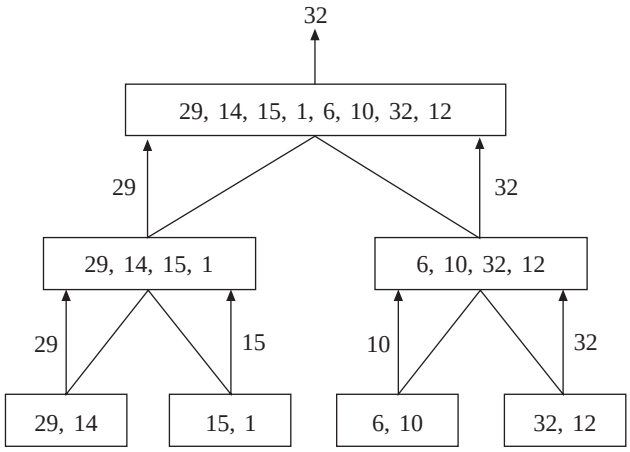
Paso 2. Los dos subproblemas se resuelven de manera recurrente aplicando el algoritmo divide-y-vencerás a los subproblemas.

Paso 3. Las soluciones de los subproblemas se fusionan en una solución del problema original.

La recurrencia del paso 2 crea subproblemas cada vez más pequeños. Al final estos subproblemas son tan pequeños que cada uno puede resolverse directa y fácilmente.

A continuación se mostrará el significado de la estrategia divide-y-vencerás mediante su aplicación para resolver el problema de encontrar el máximo. Imagine que se tienen ocho números: 29, 14, 15, 1, 6, 10, 32 y 12. La estrategia divide-y-vencerás encuentra el máximo de estos ocho números, lo cual se muestra en la figura 4-1.

FIGURA 4-1 Estrategia divide-y-vencerás para encontrar el máximo de ocho números.



Como se muestra en la figura 4-1, primero se separan los ocho números en dos subconjuntos y luego cada uno de ellos se divide en dos subconjuntos que constan de dos números. Debido a que un subconjunto sólo contiene dos números, este subconjunto ya no se divide más. El máximo se determina con una simple comparación de estos dos números.

Una vez que se determinan los cuatro máximos, comienza la fusión. Así, 29 se compara con 15 y 10 se compara con 32. Finalmente, al comparar 32 y 29 se encuentra que el máximo es 32.

En general, la complejidad $T(n)$ de un algoritmo divide-y-vencerás se determina con la siguiente fórmula:

$$T(n) = \begin{cases} 2T(n/2) + S(n) + M(n) & n \geq c \\ b & n < c \end{cases}$$

donde $S(n)$ denota los pasos temporales necesarios para dividir el problema en dos subproblemas, $M(n)$ denota los pasos temporales necesarios para fusionar las dos subsoluciones y b es una constante. Para el problema de encontrar el máximo, la separación y la fusión requieren un número constante de pasos. En consecuencia, se tiene

$$T(n) = \begin{cases} 2T(n/2) + 1 & n > 2 \\ 1 & n = 2. \end{cases}$$

Si se supone que $n = 2^k$, se tiene

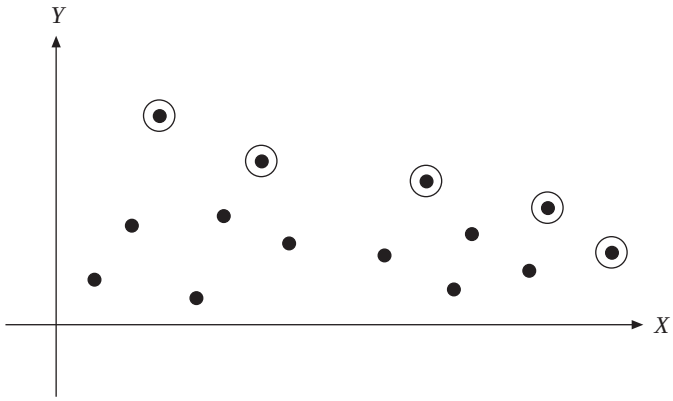
$$\begin{aligned} T(n) &= 2T(n/2) + 1 \\ &= 2(2T(n/4) + 1) + 1 \\ &\vdots \\ &= 2^{k-1}T(2) + \sum_{j=0}^{k-2} 2^j \\ &= 2^{k-1} + 2^{k-1} - 1 \\ &= 2^k - 1 \\ &= n - 1. \end{aligned}$$

El lector debe observar que la estrategia divide-y-vencerás se aplicó dos veces para encontrar el máximo sólo con fines ilustrativos. Resulta evidente que en este caso la estrategia divide-y-vencerás no es demasiado efectiva porque con una exploración directa de estos n números y al hacer $n - 1$ comparaciones también se encuentra el máximo y se tiene la misma eficacia que con la estrategia divide-y-vencerás. No obstante, en los siguientes apartados se demostrará que en muchos casos, sobre todo en problemas geométricos, la estrategia divide-y-vencerás es realmente buena.

4-1 El problema de cómo encontrar puntos máximos en un espacio bidimensional

En el espacio bidimensional se dice que un punto (x_1, y_1) domina a (x_2, y_2) si $x_1 > x_2$ y $y_1 > y_2$. Un punto se denomina máximo si ningún otro punto lo domina. Dado un conjunto de n puntos, el problema de encontrar los máximos consiste en encontrar todos los puntos máximos de entre estos n puntos. Por ejemplo, los puntos encerrados en un círculo en la figura 4-2 son puntos máximos.

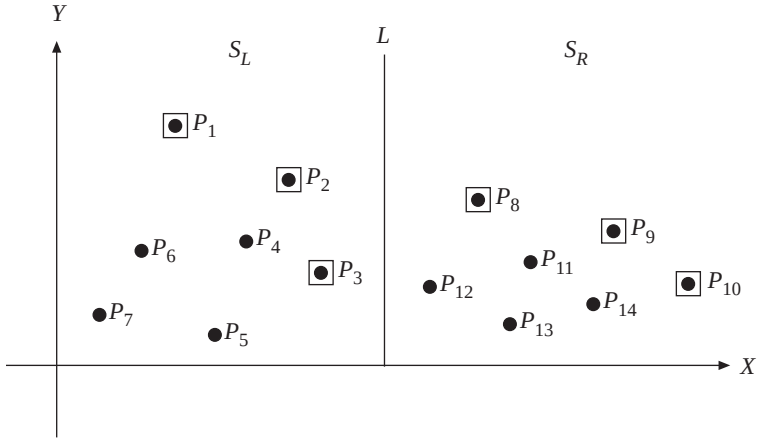
FIGURA 4-2 Puntos máximos.



Un método directo para resolver el problema de encontrar los máximos consiste en comparar cada par de puntos. Este método directo requiere $O(n^2)$ comparaciones de puntos. Como se mostrará a continuación, si se aplica la estrategia divide-y-vencerás, el problema puede resolverse en $O(n \log n)$ pasos, lo cual constituye, de hecho, una mejora bastante aceptable.

Si se aplica la estrategia divide-y-vencerás, primero se encuentra la recta mediana L perpendicular al eje x, que divide en dos subconjuntos todo el conjunto de puntos, como se muestra en la figura 4-3. Los conjuntos de puntos a la izquierda de L y a la derecha de L se denotan por S_L y S_R, respectivamente.

FIGURA 4-3 Los puntos máximos de S_L y S_R.



El siguiente paso es encontrar en forma recurrente los puntos máximos de S_L y S_R. Como se muestra en la figura 4-3, P_1, P_2 y P_3 son puntos máximos de S_L y P_8, P_9 y P_{10} son puntos máximos de S_R.

El proceso de fusión es muy sencillo. Debido a que el valor x de un punto en S_R siempre es mayor que el valor x de todo punto en S_L, un punto en S_L es un máximo si y sólo si su valor y no es menor que el valor y de un máximo de S_R. Por ejemplo, para el caso que se muestra en la figura 4-3, P_3 debe eliminarse porque es dominado por P_8 en S_R. El conjunto de puntos máximos de S es P_1, P_2, P_8, P_9 y P_{10}.

Con base en el análisis anterior, podemos resumir como sigue el algoritmo divide-y-vencerás para resolver el problema de cómo encontrar los puntos máximos en un espacio bidimensional:

Algoritmo 4-1 □ Un método divide-y-vencerás para encontrar puntos máximos en un plano

Input: Un conjunto de n puntos planos.
Output: Los puntos máximos de S.
Paso 1. Si S sólo contiene un punto, proporcionarlo como el máximo. En caso contrario, encontrar una recta L perpendicular al eje x que separe el conjunto de puntos en dos subconjuntos S_L y S_R, cada uno con $n/2$ puntos.
Paso 2. Los puntos máximos de S_L y S_R se encuentran de manera recursiva.
Paso 3. Los puntos máximos de S_L y S_R se proyectan sobre L y se ordenan según sus valores y. Sobre las proyecciones se realiza una exploración lineal y cada uno de los puntos máximos de S_L se elimina si su valor y es menor que el valor y de algún punto máximo de S_R.

La complejidad temporal de este algoritmo depende de las complejidades temporales de los pasos siguientes:

1. En el paso 1 hay una operación que encuentra la mediana de n números. Después se demostrará que esto puede hacerse en $O(n)$ pasos.
2. En el paso 2 hay dos subproblemas, cada uno con $n/2$ puntos.
3. En el paso 3, la proyección y la exploración lineal pueden hacerse en $O(n \log n)$ pasos después de ordenar los n puntos según sus valores y.

Hacer $T(n)$ la complejidad temporal del algoritmo divide-y-vencerás para encontrar puntos máximos de n puntos en un plano. Entonces

$$T(n) = \begin{cases} 2T(n/2) + O(n) + O(n \log n) & n > 1 \\ 1 & n = 1. \end{cases}$$

Sea $n = 2^k$. Es fácil demostrar que

$$\begin{aligned} T(n) &= O(n \log n) + O(n \log^2 n) \\ &= O(n \log^2 n). \end{aligned}$$

Así, se ha obtenido un algoritmo $O(n \log^2 n)$. Es necesario recordar al lector que si se aplica un método directo para revisar cada par de puntos exhaustivamente, se requieren $O(n^2)$ pasos.

Se observa que la estrategia divide-y-vencerás es dominada por el ordenamiento en los pasos de fusión (o mezcla). De alguna manera, ahora no se está realizando un trabajo muy eficiente porque el ordenamiento debe hacerse de una vez por todas. Es decir, debe hacerse un preordenamiento. Si esto se lleva a cabo, entonces la complejidad de la mezcla es $O(n)$ y el número total de pasos necesarios es

$$O(n \log n) + T(n)$$

donde

$$T(n) = \begin{cases} 2T(n/2) + O(n) + O(n) & n > 1 \\ 1 & n = 1 \end{cases}$$

y fácilmente se encuentra que $T(n)$ es $O(n \log n)$. Así, la complejidad temporal total de usar la estrategia divide-y-vencerás para encontrar puntos máximos con preordenamiento es $O(n \log n)$. En otras palabras, se ha obtenido un algoritmo aún más eficiente.

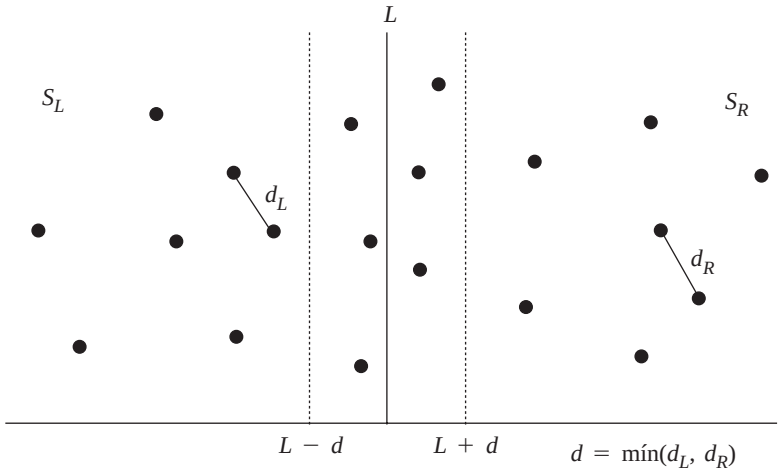
4-2 EL PROBLEMA DEL PAR MÁS CERCANO (CLOSEST NEIGHBOR)

Este problema se define como sigue: dado un conjunto S de n puntos, encontrar un par de puntos que sean los más cercanos entre sí. El problema unidimensional del par más cercano puede resolverse en $O(n \log n)$ pasos al ordenar los n números y realizando una exploración lineal. Al examinar la distancia entre cada par de puntos adyacentes en la lista ordenada es posible determinar el par más cercano.

En el espacio bidimensional la estrategia divide-y-vencerás puede aplicarse para diseñar un algoritmo eficiente a fin de encontrar el par más cercano. Así como se hizo cuando se resolvió el problema de cómo encontrar el máximo, el conjunto S primero se

parte en S_L y S_R de modo que todo punto en S_L esté a la izquierda de todo punto en S_R y el número de puntos en S_L sea igual al número de puntos en S_R. Es decir, se encuentra una recta vertical L perpendicular al eje x de modo que S se corte en dos subconjuntos del mismo tamaño. Al resolver el problema del par más cercano en S_L y S_R, respectivamente, se obtienen d_L y d_R, donde d_L y d_R denotan las distancias de los pares más cercanos en S_L y S_R, respectivamente. Sea $d = \text{mín}(d_L, d_R)$. Si el par más cercano (P_a, P_b) de S consta de un punto en S_L y un punto en S_R, entonces P_a y P_b deben estar en el interior de una franja central en la recta L y acotada por las rectas $L - d$ y $L + d$, como se muestra en la figura 4-4.

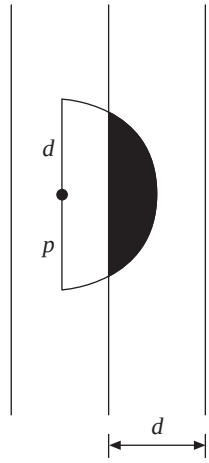
FIGURA 4-4 El problema del par más cercano.



El análisis anterior indica que durante el paso de fusión sólo deben examinarse los puntos que se encuentran en la franja. Aunque en promedio el número de puntos de la franja central no puede ser muy grande, en el peor de los casos habrá no más de n puntos. Así, el método de la fuerza bruta para encontrar el par más cercano en la franja requiere calcular $n^2/4$ distancias y comparaciones. Este tipo de paso de fusión no es bueno para nuestro algoritmo divide-y-vencerás. Afortunadamente, como se demostrará a continuación, el paso de fusión puede completarse en tiempo $O(n)$.

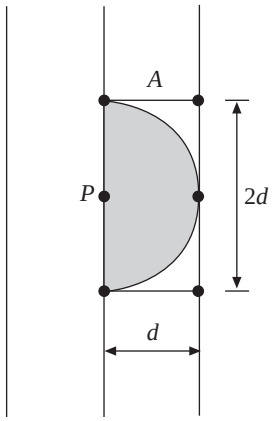
Si un punto P en S_L y un punto Q en S_R constituyen un par más cercano, entonces la distancia entre P y Q debe ser menor que d. Por lo tanto, no es necesario considerar un punto que esté demasiado lejos de P. Considere la figura 4-5. En esta figura basta analizar la región sombreada. Si P está exactamente en la recta L, esta región sombreada es máxima. Incluso en este caso, esta región sombreada estará contenida en el rectángulo A de $d \times 2d$, como se muestra en la figura 4-6. Así, sólo es necesario examinar los puntos dentro de este rectángulo A.

FIGURA 4-5 Región limitada a examinar en el proceso de fusión del algoritmo divide-y-vencerás del par más cercano.



La pregunta que falta es: ¿cuántos puntos hay en el rectángulo *A*? Como se sabe que la distancia entre cada par de puntos tanto en S_L como en S_R es mayor o igual a *d*, en este rectángulo sólo puede haber a lo sumo seis puntos, que se muestran en la figura 4-6. Con base en esta observación se sabe que en el paso de fusión, para cada punto en la franja, solamente es necesario examinar un número limitado de puntos en la otra mitad de la franja. Sin pérdida de generalidad, puede suponerse que *P* está en la mitad izquierda de la franja. Sea y_p el valor *y* de *P.* Para *P* sólo es necesario examinar puntos en la otra mitad de la franja cuyos valores *y* estén entre $y_p + d$ y $y_p - d$. Como vimos antes, cuando mucho hay seis puntos así.

FIGURA 4-6 Rectángulo *A* que contiene los posibles vecinos más cercanos de *P*.



Como ya se analizó, antes de aplicar el algoritmo divide-y-vencerás es necesario ordenar los n puntos según su valor y. Después de hacer el preordenamiento, el paso de fusión se lleva a cabo en sólo $O(n)$ pasos.

A continuación se aplica el algoritmo divide-y-vencerás para resolver el problema bidimensional del par más cercano.

Algoritmo 4-2 □ Un algoritmo divide-y-vencerás para resolver el problema bidimensional del par más cercano

Input: Un conjunto S de n puntos en el plano.

Output: La distancia entre los dos puntos más cercanos.

Paso 1. Ordenar los puntos en S según sus valores y y sus valores x.

Paso 2. Si S contiene un solo punto, regresar ∞ como su distancia.

Paso 3. Buscar una recta mediana L perpendicular al eje x para dividir S en dos subconjuntos del mismo tamaño: S_L y S_R. Todo punto en S_L está a la izquierda de L y todo punto en S_R está a la derecha de L.

Paso 4. Recursivamente aplicar el paso 2 y el paso 3 para resolver los problemas de los pares más cercanos de S_L y S_R. Sea $d_L(d_R)$ la distancia entre el par más cercano en $S_L(S_R)$. Sea $d = \text{mín}(d_L, d_R)$.

Paso 5. Proyectar sobre la recta L todos los puntos dentro de la franja acotada por $L - d$ y $L + d$. Para un punto P en la semifranja acotada por $L - d$ y L, hacer y_p su valor y. Para cada uno de tales puntos P, en la semifranja acotada por L y $L + d$ encontrar todos los puntos cuyo valor y esté entre $y_p + d$ y $y_p - d$. Si la distancia d' entre P y un punto en la otra semifranja es menor que d, sea $d = d'$. El valor final de d es la respuesta.

La complejidad temporal del algoritmo completo es igual a $O(n \log n) + T(n)$ y

$$T(n) = 2T(n/2) + S(n) + M(n)$$

donde $S(n)$ y $M(n)$ son las complejidades temporales del paso de separación en el paso 3 y del paso de fusión en el paso 4. El paso de separación se lleva a cabo en $O(n)$ pasos porque los puntos ya están ordenados según sus valores x. El paso de fusión primero tiene que encontrar todos los puntos dentro de $L - d$ y $L + d$. De nuevo, ya que los puntos están ordenados sobre el eje x, puede llevarse a cabo en $O(n)$ pasos. Para cada punto cuando mucho es necesario examinar otros 12 puntos, 6 de ellos en cada semifranja. Así, esta exploración lineal requiere $O(n)$ pasos. En otras palabras, la complejidad temporal del paso de fusión es $O(n)$.

Así,

$$T(n) = \begin{cases} 2T(n/2) + O(n) + O(n) & n > 1 \\ 1 & n = 1. \end{cases}$$

Puede demostrarse fácilmente que

$T(n) = O(n \log n)$.

La complejidad temporal total es

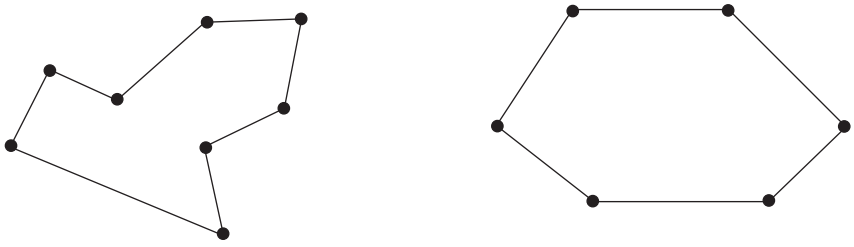
$O(n \log n)$(complejidad del preordenamiento) + $O(n \log n) = O(n \log n)$.

4-3 EL PROBLEMA DEL CONVEX HULL

Este problema se mencionó por primera vez en el capítulo 2. En esta sección se demostrará que este problema del convex hull puede resolverse de manera elegante mediante la estrategia divide-y-vencerás.

Un polígono convexo es un polígono con la propiedad de que cualquier segmento de recta que une dos puntos cualesquiera dentro del polígono debe estar dentro del polígono. En la figura 4-7*a*) se muestra un polígono no convexo y en la figura 4-7*b*) se muestra un polígono convexo.

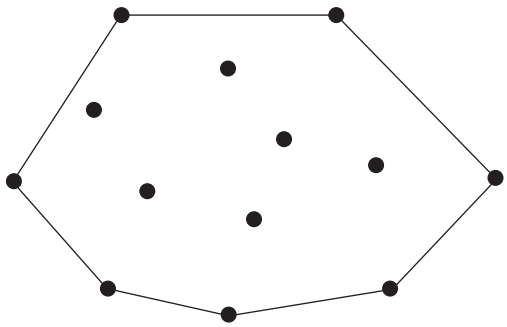
FIGURA 4-7 Polígonos cóncavo y convexo.



a) *b*)

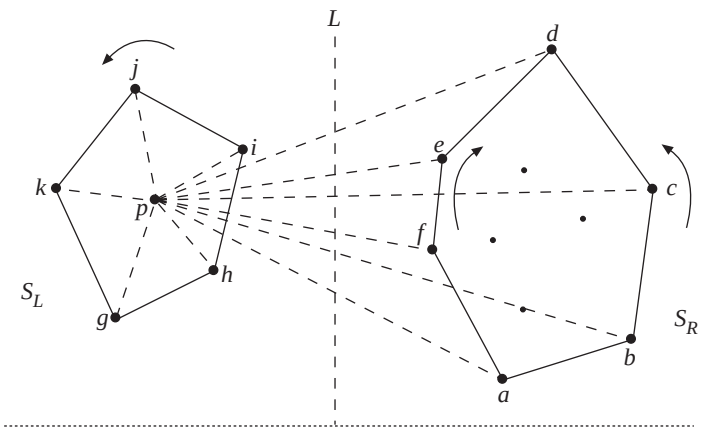
El convex hull de un conjunto de puntos planos se define como el menor polígono convexo que contiene todos los puntos. Por ejemplo, en la figura 4-8 se muestra un convex hull típico.

FIGURA 4-8 Un convex hull.



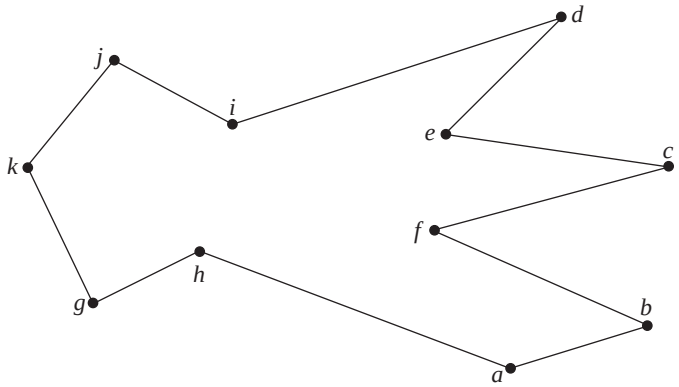
Para encontrar un convex hull puede aplicarse la estrategia divide-y-vencerás. Considere la figura 4-9. En esta figura, el conjunto de puntos planos se ha dividido en dos subconjuntos S_L y S_R por medio de una recta perpendicular al eje *x*. Luego se construyen conjuntos convexos para S_L y S_R, denominados Hull(H_L) y Hull(H_R), respectivamente. Para combinar Hull(H_L) y Hull(H_R) en un convex hull puede aplicarse el algoritmo de Graham.

FIGURA 4-9 Estrategia divide-y-vencerás para construir un convex hull.



Para llevar a cabo el algoritmo de búsqueda de Graham se escoge un punto interior. Este punto se considera como el origen. Así, cada otro punto forma un ángulo polar con respecto a este punto interior. Luego todos los puntos se ordenan con respecto a estos ángulos polares. El algoritmo de Graham analiza los puntos uno por uno y elimina aquellos que producen ángulos reflexivos, como se ilustra en la figura 4-10. En esta figura se eliminan *h, f, e* e *i*. Los puntos restantes son vértices de un convex hull.

FIGURA 4-10 Búsqueda de Graham.



Para llevar a cabo una búsqueda de Graham después que se han construido Hull(S_L) y Hull(S_R), es posible seleccionar un punto interior P de Hull(S_L). Por ser un polígono convexo, debe resultar evidente para el lector que no es necesario examinar los puntos en el interior de los convex hull. Desde la perspectiva de P, Hull(S_R) está en la sección cuyo ángulo origen es menor que o igual a π. Este punto (en forma de cuña) está formado por los dos vértices u y v de Hull(S_R) que pueden encontrarse en tiempo lineal aplicando el siguiente procedimiento: se traza una recta horizontal que pasa por P. Si esta recta corta a Hull(S_R), entonces Hull(S_R) está en la sección determinada por los dos vértices de Hull(S_R) que tienen el mayor ángulo polar $< \pi/2$ y el menor ángulo polar $> 3\pi/2$, respectivamente. Si la recta horizontal que pasa por P no corta a Hull(S_R), la sección está determinada por los vértices que originan el ángulo polar mayor y el menor con respecto a P. En la figura 4-9 estos puntos son a y d. De esta forma, hay tres secuencias de puntos que tienen ángulos polares crecientes con respecto a P, un punto interior de S_L. Las tres secuencias son:

1. g, h, i, j, k
2. a, b, c, d

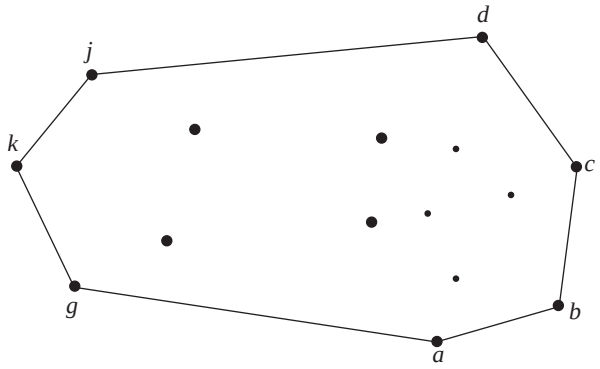
y 3. f, e.

Estas tres secuencias pueden fusionarse en una. En nuestro caso, la secuencia fusionada es

$g, h, a, b, f, c, e, d, i, j, k.$

Ahora es posible aplicar la búsqueda de Graham a esta secuencia para eliminar puntos que no pueden ser vértices del convex hull. Para el caso que se muestra en la figura 4-9, el convex hull resultante se muestra en la figura 4-11.

FIGURA 4-11 El convex hull para los puntos en la figura 4-9.



A continuación se presenta el algoritmo divide-y-vencerás para construir un convex hull.

Algoritmo 4-3 □ Un algoritmo para construir un convex hull con base en la estrategia divide-y-vencerás

Input: Un conjunto S de puntos planos.

Output: Un convex hull para S.

Paso 1. Si S no contiene más de cinco puntos, para encontrar el convex hull se usa búsqueda exhaustiva y luego se regresa.

Paso 2. Buscar una recta mediana perpendicular al eje x que divida a S en S_L y S_R; S_L está a la izquierda de S_R.

Paso 3. Recursivamente se construyen conjuntos convexos para S_L y S_R. Estos conjuntos convexos se denotan por Hull(S_L) y Hull(S_R), respectivamente.

Paso 4. Buscar un punto interior P de S_L. Buscar los vértices v_1 y v_2 de Hull (S_R) que divide los vértices de Hull (S_R) en dos secuencias de vértices que tienen ángulos polares crecientes con respecto a P. Sin pérdida de generalidad, se supone que el valor y de v_1 es mayor que el valor y de v_2. Luego se forman tres secuencias como sigue:

a) Secuencia 1: todos los vértices del convex hull de Hull(S_L) en dirección contraria al movimiento de las manecillas del reloj.

b) Secuencia 2: los vértices del convex hull de Hull(S_R) de v_2 a v_1 en dirección contraria al movimiento de las manecillas del reloj.

c) Secuencia 3: los vértices del convex hull de Hull(S_R) de v_2 a v_1 en dirección del movimiento de las manecillas del reloj.

Estas tres secuencias se fusionan y se lleva a cabo una búsqueda de Graham. Se eliminan los puntos que son reflexivos (porque producen una concavidad en un polígono convexo). Los puntos restantes forman el convex hull.

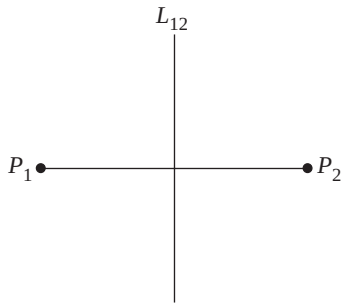
La complejidad temporal del algoritmo anterior es determinada esencialmente por el proceso de separación y el proceso de fusión. La complejidad temporal del proceso de separación es $O(n)$ porque éste es un proceso de determinación de la mediana. La complejidad temporal del proceso de fusión es $O(n)$ porque para llevar a cabo la búsqueda del punto interior, la determinación de los puntos extremos, la fusión y la búsqueda de Graham se requieren $O(n)$ pasos. Así, $T(n) = 2T(n/2) + O(n) = O(n \log n)$.

4-4 DIAGRAMAS DE VORONOI CONSTRUIDOS CON LA ESTRATEGIA DIVIDE-Y-VENCERÁS

El diagrama de Voronoi constituye, como el árbol de expansión mínima mencionado en el capítulo 3, una estructura de datos muy interesante. Esta estructura de datos puede usarse para almacenar información importante sobre los vecinos más cercanos de puntos en un plano.

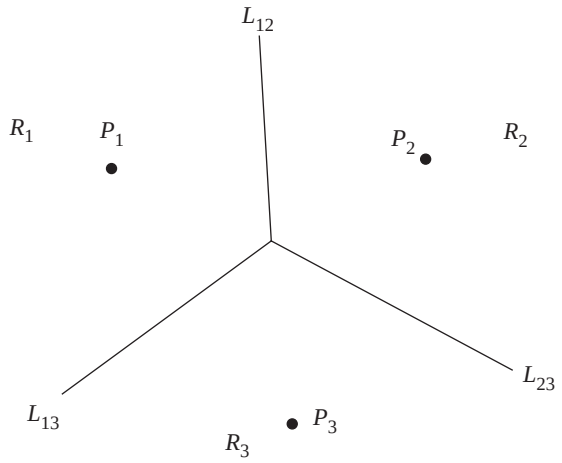
Se considerará el caso de dos puntos como se muestra en la figura 4-12. En esta figura, la recta L_{12} es una bisectriz perpendicular de la recta que une P_1 y P_2. L_{12} es el diagrama de Voronoi para P_1 y P_2. Para todo punto X situado en el semiplano que está a la izquierda (derecha) de L_{12}, el vecino más cercano de X, de entre P_1 y P_2, es $P_1(P_2)$. En cierto sentido, dado cualquier punto X, para encontrar un vecino más cercano de X no es necesario calcular la distancia entre X y P_1 y la distancia entre X y P_2. En vez de ello, sólo se requiere determinar a qué lado de L_{12} está X. Esto puede hacerse por sustitución de las coordenadas de X en L_{12}. Dependiendo del signo del resultado de esta sustitución es posible determinar dónde esta ubicado X.

FIGURA 4-12 Diagrama de Voronoi para dos puntos.



Considere la figura 4-13, donde cada L_{ij} es una bisectriz perpendicular de la recta que une P_i y P_j. Los tres hiperplanos L_{12}, L_{13} y L_{23} constituyen el diagrama de Voronoi para los puntos P_1, P_2 y P_3. Si un punto está ubicado en R_i, entonces su vecino más cercano entre P_1, P_2 y P_3 es P_i.

FIGURA 4-13 Diagrama de Voronoi para tres puntos.



Para definir el diagrama de Voronoi de un conjunto de puntos en el plano, primero se debe definir el polígono de Voronoi.

Definición

Dados dos puntos P_i y P_j en un conjunto S de n puntos, sea $H(P_i, P_j)$ el semiplano que contiene los puntos más cercanos a P_i. El polígono de Voronoi asociado con P_i es una región poligonal convexa que no tiene más de $n - 1$ lados, definida por $V(i) = \bigcap_{i \neq j} H(P_i, P_j)$.

En la figura 4-14 se muestra un polígono de Voronoi. Dado un conjunto de n puntos, el diagrama de Voronoi incluye todos los polígonos de Voronoi de estos puntos. Los vértices del diagrama de Voronoi se denominan puntos de Voronoi, y sus segmentos se denominan bordes de Voronoi. (El nombre Voronoi se refiere a un famoso matemático ruso.) En la figura 4-15 se muestra un diagrama de Voronoi para seis puntos.

FIGURA 4-14 Polígono de Voronoi.

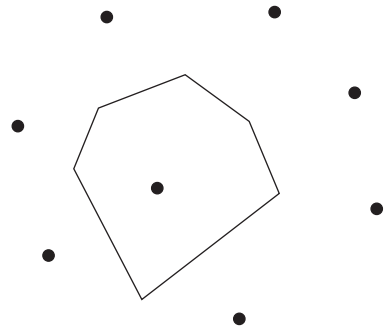
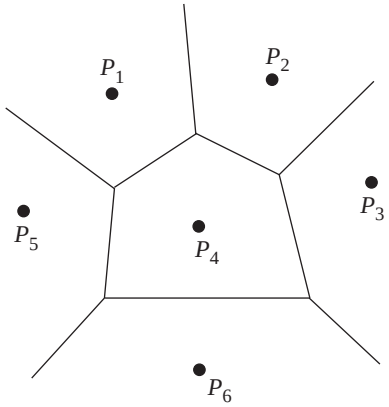


FIGURA 4-15 Diagrama de Voronoi para seis puntos.

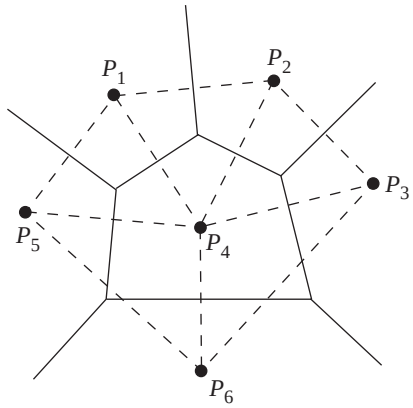


La línea recta dual de un diagrama de Voronoi se denomina triangulación Delaunay, en honor de un famoso matemático francés. En una triangulación Delaunay hay un segmento de recta que une P_i y P_j si y sólo si los polígonos de Voronoi de P_i y P_j comparten el mismo borde. Por ejemplo, para el diagrama de Voronoi en la figura 4-15, su triangulación Delaunay se muestra en la figura 4-16.

Los diagramas de Voronoi son muy útiles para varias cosas. Por ejemplo, es posible resolver el denominado problema de los pares más cercanos al extraer información del diagrama de Voronoi. A partir del diagrama de Voronoi también es posible encontrar un árbol de expansión mínima.

Un diagrama de Voronoi puede construirse a la manera divide-y-vencerás con el algoritmo de la página siguiente.

FIGURA 4-16 Una triangulación Delaunay.



Algoritmo 4-4 □ Algoritmo divide-y-vencerás para construir diagramas de Voronoi

Input: Un conjunto S de n puntos planos.
Output: El diagrama de Voronoi de S.
Paso 1. Si S contiene un solo punto, return.
Paso 2. Find una recta mediana L perpendicular al eje x que divida a S en S_L y S_R de modo que S_L (S_R) esté a la izquierda (derecha) de L y el tamaño de S_L sea igual al tamaño de S_R.
Paso 3. Recursivamente, construir diagramas de Voronoi de S_L y S_R. Denote por $VD(S_L)$ y $VD(S_R)$ estos diagramas.
Paso 4. Construir un hiperplano lineal por partes divisorio HP que sea el lugar geométrico de puntos simultáneamente más cercanos a un punto en S_L y a un punto en S_R. Eliminar todos los segmentos de $VD(S_L)$ que estén a la derecha de HP y todos los segmentos de $VD(S_R)$ que estén a la izquierda de HP. La gráfica resultante es el diagrama de Voronoi.

Este algoritmo puede entenderse al considerar la figura 4-17. Para fusionar $VD(S_L)$ y $VD(S_R)$, se observa que sólo aquellas partes de $VD(S_L)$ y $VD(S_R)$ cercanas a L interfieren entre sí. Los puntos alejados de L no son afectados por el paso de fusión, de modo que permanecen intactos.

El paso más importante al fusionar $VD(S_L)$ y $VD(S_R)$ es construir el hiperplano lineal por partes divisorio HP. Este hiperplano tiene la siguiente propiedad: si un punto P está dentro de la parte izquierda (derecha) de HP, entonces el vecino más cercano de

P debe ser un punto en $S_L(S_R)$. Véase la figura 4-17. El *HP* para $VD(S_L)$ y $VD(S_R)$ se muestra en la figura 4-18.

FIGURA 4-17 Dos diagramas de Voronoi después del paso 2.

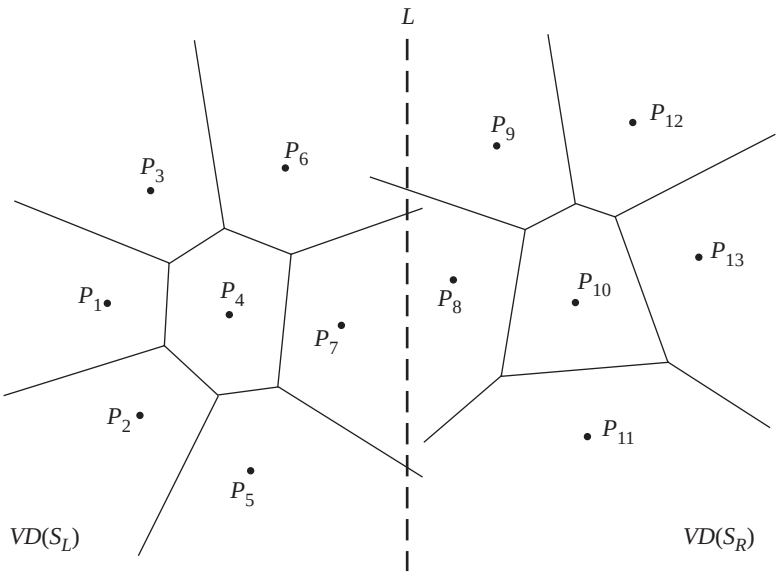
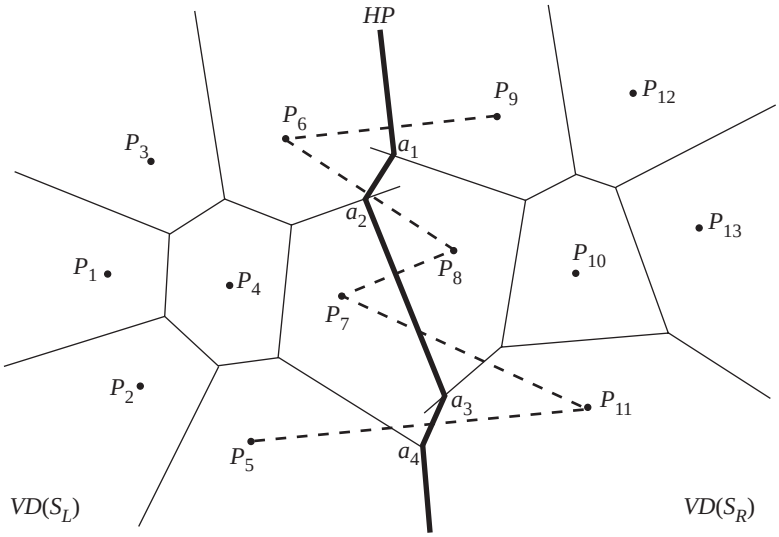
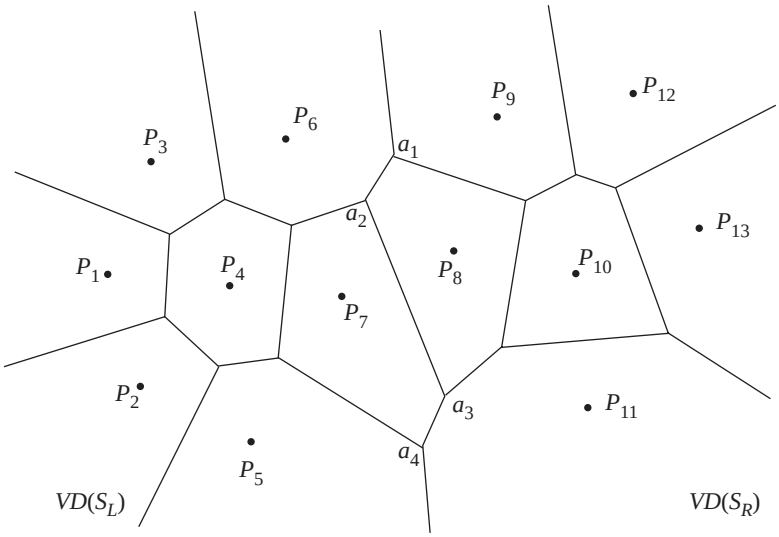


FIGURA 4-18 El hiperplano lineal por partes para el conjunto de puntos que se muestra en la figura 4-17.



Después de eliminar todos los segmentos de $VD(S_L)$ a la derecha de *HP* y todos los segmentos de $VD(S_R)$ a la izquierda de *HP*, se obtiene el diagrama de Voronoi resultante que se muestra en la figura 4-19.

FIGURA 4-19 Diagrama de Voronoi de los puntos en la figura 4-17.



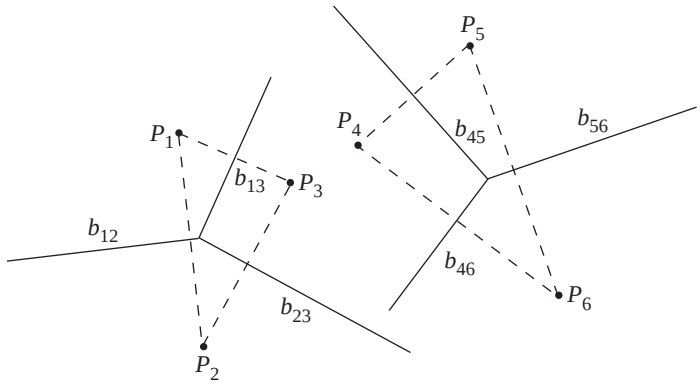
Sean Hull (S_L) y Hull (S_R) denotados los convex hull de S_L y S_R respectivamente. Existen dos segmentos de línea entre S_L y S_R que unen el Hull(S_L) y el Hull(S_R) en un convex hull. Sea $\overline{P_a P_b}$ el segmento superior, donde P_a y P_b pertenecen a S_L y S_R respectivamente. El primer paso para la construcción de *HP* es encontrar P_a y P_b. Se seleccionan de P_6 y P_9. Luego se traza una bisectriz perpendicular de la recta $\overline{P_6 P_9}$. Éste es el segmento inicial de *HP*. Luego, como se muestra en la figura 4-18, en a_1, *HP* se corta con la bisectriz de la recta $\overline{P_9 P_8}$. Así, se sabe que el lugar geométrico estará más cercano a P_8 que a P_9. En otras palabras, el siguiente segmento es una bisectriz de $\overline{P_6 P_8}$.

Desplazándose hacia abajo, *HP* corta a $VD(S_L)$ en a_2. El segmento de $VD(S_L)$ que corta a *HP* es la bisectriz de $\overline{P_6 P_7}$. Así, el nuevo segmento es la bisectriz de $\overline{P_7 P_8}$.

A continuación, el proceso de construcción de un diagrama de Voronoi se ilustrará con otro ejemplo. Considere los seis puntos en la figura 4-20.

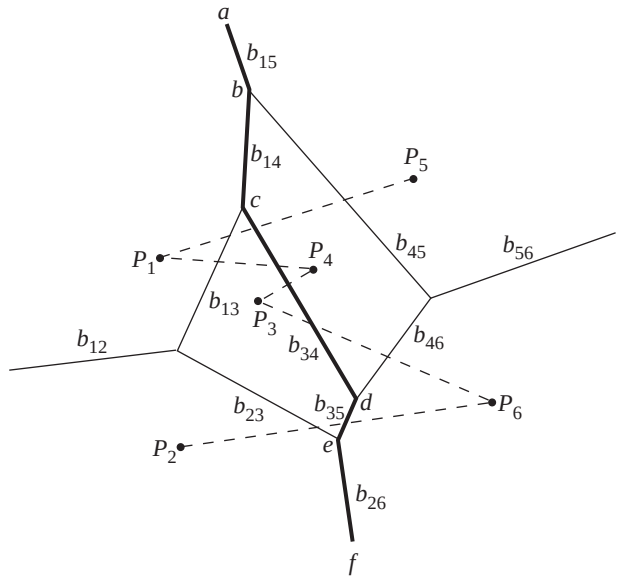
En la figura 4-20 se muestran dos diagramas de Voronoi construidos para $\{P_1, P_2, P_3\}$ y $\{P_4, P_5, P_6\}$, respectivamente, donde se han identificado todos los trazos. Por ejemplo, b_{13} es la bisectriz perpendicular de la recta $\overline{P_1 P_3}$. El proceso de construcción de un diagrama de Voronoi requiere la construcción de *conjuntos convexos*. Los dos conjuntos convexos se muestran en la figura 4-20. Ambos son triángulos. Después de que se construyen los dos conjuntos convexos, se concluye que $\overline{P_1 P_5}$ y $\overline{P_2 P_6}$ son los dos segmentos que unen los dos conjuntos convexos.

FIGURA 4-20 Otro caso que ilustra la construcción de diagramas de Voronoi.



Así, el paso de fusión empieza desde que se encuentra la bisectriz perpendicular de $\overline{P_1P_5}$, como se muestra en la figura 4-21.

FIGURA 4-21 Paso de fusión en la construcción de un diagrama de Voronoi.



Todo el paso de fusión consta de los pasos siguientes:

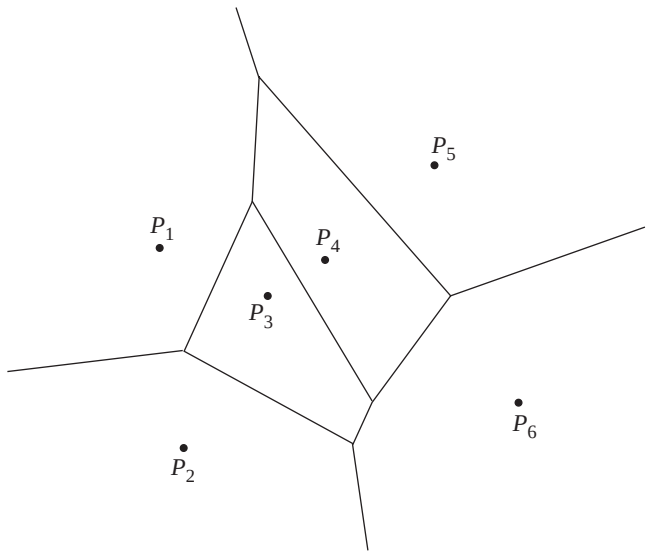
1. La bisectriz perpendicular de $\overline{P_1P_5}$ corta a b_{45} en b. Luego se encuentra la bisectriz perpendicular de $\overline{P_1P_4}$. Esta recta se identifica como b_{14}.
2. b_{14} corta a b_{13} en c. La siguiente bisectriz perpendicular de $\overline{P_3P_4}$ es b_{34}.
3. b_{34} corta a b_{46} en d. La siguiente bisectriz perpendicular de $\overline{P_3P_6}$ es b_{36}.
4. b_{36} corta a b_{23} en e. La siguiente bisectriz perpendicular de $\overline{P_2P_6}$ es b_{26}.

Debido a que P_2 y P_6 son los puntos más bajos de los dos conjuntos convexos, el trazo b_{26} se extiende al infinito y entonces se encuentra el hiperplano lineal por partes divisorio *HP*. Así, *HP* está definido por $\overline{ab}$, $\overline{bc}$, $\overline{cd}$, $\overline{de}$ y $\overline{ef}$. Si un punto cae a la derecha (izquierda) de este *HP*, entonces su vecino más cercano debe ser uno de $\{P_4, P_5, P_6\}(\{P_1, P_2, P_3\})$.

En la figura 4-21 se muestran todas las rectas cuyas bisectrices perpendiculares constituyen el hiperplano divisorio. Estas rectas se identifican como rectas discontinuas. Estos segmentos de recta son $\overline{P_5P_1}$, $\overline{P_1P_4}$, $\overline{P_4P_3}$, $\overline{P_3P_6}$ y $\overline{P_6P_2}$.

En el paso de fusión, la última parte es la eliminación de todos los $VD(S_L)$ ($VD(S_R)$) que se extiendan a la derecha (izquierda) de *HP*. El diagrama de Voronoi resultante se muestra en la figura 4-22.

FIGURA 4-22 Diagrama de Voronoi resultante.



A continuación se presenta el algoritmo para fusionar dos diagramas de Voronoi.

Algoritmo 4-5 □ Algoritmo que fusiona dos diagramas de Voronoi en un diagrama de Voronoi

Input: *a*) S_L y S_R, donde S_L y S_R están divididos por una recta perpendicular L. *b*) $VD(S_L)$ y $VD(S_R)$.

Output: $VD(S)$, donde $S = S_L \cup S_R$.

Paso 1. Buscar los conjuntos convexos de S_L y S_R, que se denotan por Hull(S_L) y Hull(S_R), respectivamente. (Después se proporcionará un algoritmo para encontrar un convex hull en este caso.)

Paso 2. Buscar los segmentos $\overline{P_aP_b}$ y $\overline{P_cP_d}$ que unen a Hull(S_L) y Hull(S_R) en un convex hull (P_a y P_c pertenecen a S_L, y P_b y P_d pertenecen a S_R). Suponga que $\overline{P_aP_b}$ está por arriba de $\overline{P_cP_d}$. Hacer $x = a$, $y = b$, $SG = \overline{P_xP_y}$ y $HP = \phi$.

Paso 3. Buscar la bisectriz perpendicular de SG, que se denota por BS. Hacer $HP = HP \cup BS$. Si $SG = \overline{P_cP_d}$, ir a paso 5; en caso contrario, ir a paso 4.

Paso 4. Se deja que BS corte primero un trazo desde $VD(S_L)$ o $VD(S_R)$. Este trazo debe ser una bisectriz perpendicular de $\overline{P_xP_z}$ o $\overline{P_yP_z}$ para alguna z. Si este trazo es la bisectriz perpendicular de $\overline{P_yP_z}$, entonces hacer $SG = \overline{P_xP_z}$; en caso contrario, hacer $SG = \overline{P_zP_y}$. Ir a paso 3.

Paso 5. Se eliminan los bordes de $VD(S_L)$ que se extienden a la derecha de HP y los bordes de $VD(S_R)$ que se extienden a la izquierda de HP. La gráfica resultante es el diagrama de Voronoi de $S = S_L \cup S_R$.

A continuación se describirán en detalle las propiedades de HP, aunque antes se define la distancia entre un punto y un conjunto de puntos.

Definición

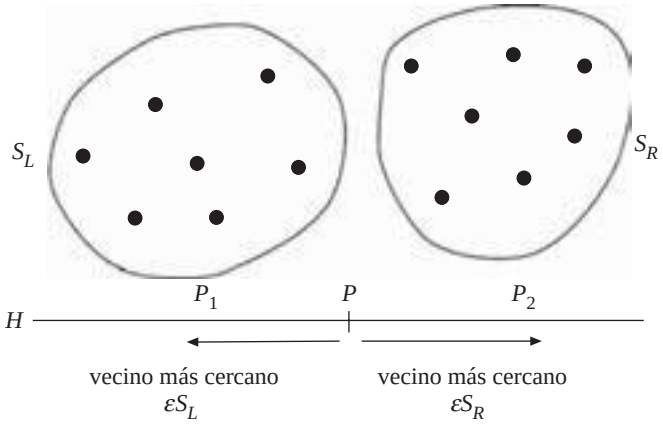
Dados un punto P y un conjunto S de puntos $P_1, P_2, \ldots, P_n$, sea P_i un vecino más cercano de P. La distancia entre P y S es la distancia entre P y P_i.

Usando esta definición puede afirmarse lo siguiente: el HP obtenido con el algoritmo 4-5 es el lugar geométrico de los puntos que preservan distancias iguales a S_L y S_R. Es decir, para cada punto P en HP, sean P_i y P_j un vecino más cercano de S_L y S_R de P, respectivamente. Entonces la distancia entre P y P_i es igual a la distancia entre P y P_j.

Por ejemplo, considere la figura 4-21. Sea P un punto sobre el segmento de recta $\overline{cd}$. Para S_L, el vecino más cercano de P es P_3 y para S_R, el vecino más cercano de P es P_4. Debido a que $\overline{cd}$ es la bisectriz perpendicular de $\overline{P_3P_4}$, cada punto en $\overline{cd}$ es equidistante a P_3 y P_4. Así, para cada punto en $\overline{cd}$, está a la misma distancia de S_L y S_R.

Al usar la propiedad anterior de *HP*, fácilmente se observa que *toda recta horizontal H corta a HP por lo menos una vez*. Este hecho puede verse fácilmente si se considera la figura 4-23.

FIGURA 4-23 Relación entre una recta horizontal H y S_L y S_R.

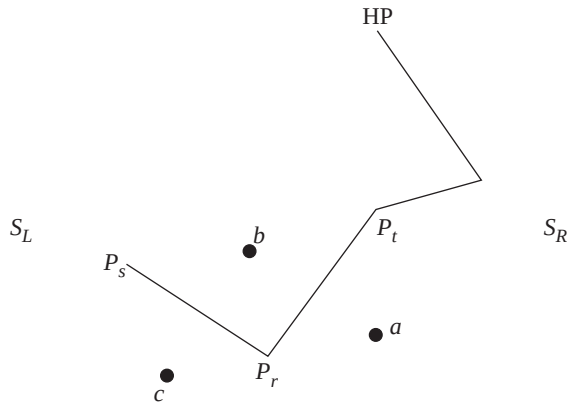


Imagine que se efectúa un desplazamiento de izquierda a derecha sobre la recta H. Al principio, para un punto como P_1, el vecino más cercano de P_1 debe ser un miembro de S_L. Imagine también que se efectúa un desplazamiento de derecha a izquierda. Para un punto como P_2, el vecino más cercano de P_2 debe ser un miembro de S_R. Debido a que H es una recta contigua, debe haber un punto P tal que a la izquierda (derecha) de P cada punto tenga como su vecino más cercano a un elemento de $S_L(S_R)$. *Así, P es equidistante a S_L y S_R. O bien, la distancia entre P y su vecino más cercano en S_L debe ser igual a la distancia entre P y su vecino más cercano en S_R.* Debido a que *HP* es el lugar geométrico de los puntos equidistantes a S_L y S_R, este punto P también debe estar en *HP*. Así, se ha demostrado que toda recta horizontal debe cortar a *HP* por lo menos una vez. El lector debe consultar la figura 4-21 para convencerse de la validez de la observación anterior.

Al considerar la figura 4-21 se observa que *HP* posee otra propiedad interesante; es monótono en *y*. A continuación se demostrará que *toda recta horizontal H corta a HP cuando mucho una vez.*

Suponga lo contrario. Entonces hay un punto P_r donde *HP* gira hacia arriba, como se muestra en la figura 4-24.

FIGURA 4-24 Ilustración de la monotonía de *HP*.



En la figura 4-24, $\overline{P_tP_r}$ es la bisectriz perpendicular de $\overline{ab}$ y $\overline{P_tP_r}$ es la bisectriz perpendicular de $\overline{bc}$. Debido a que *HP* es el lugar geométrico que separa S_L y S_R, ocurre una de dos, que *a* y *c* pertenecen a S_L y *b* pertenece a S_R o que *a* y *c* pertenecen a S_R y *b* pertenece a S_L. Ambas situaciones son imposibles porque se tiene una recta perpendicular al eje *x* para dividir a *S* en S_L y S_R. Es decir, P_r no puede existir.

Debido a que toda recta horizontal *H* corta a *HP* por lo menos una vez y cuando mucho una vez, se concluye que *toda recta horizontal H corta a HP en uno y sólo en un punto. Es decir, HP es monótono en y.*

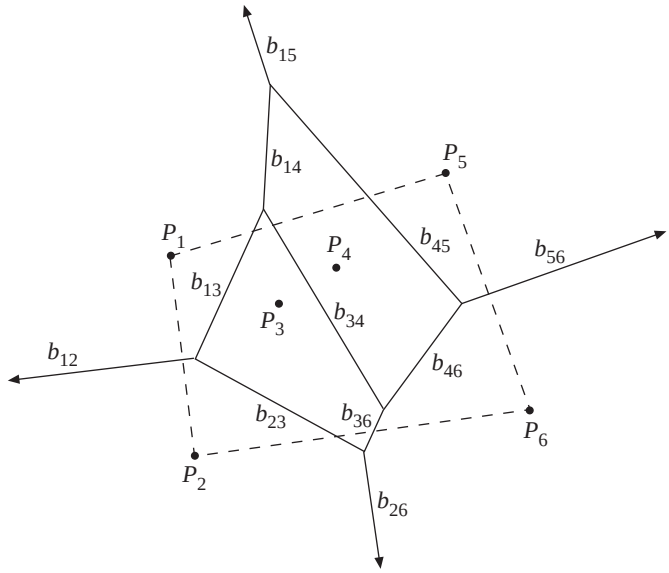
Hay otra propiedad importante de los diagramas de Voronoi que es útil para deducir la complejidad temporal del algoritmo para construir un diagrama de Voronoi. Esta propiedad es: *el número de bordes de un diagrama de Voronoi es cuando mucho* $3n - 6$*, donde n es el número de puntos.* Observe que cada diagrama de Voronoi corresponde a una triangulación Delaunay. Debido a que una triangulación Delaunay es una gráfica plana, puede contener cuando mucho $3n - 6$ bordes. Como hay una correspondencia uno a uno entre los bordes en el diagrama de Voronoi y los bordes en la triangulación Delaunay, el número de bordes en un diagrama de Voronoi es cuando mucho $3n - 6$.

Una vez que se ha obtenido la cota superior de los bordes de Voronoi, a continuación es posible deducir la cota superior de los vértices de Voronoi. Observe que cada vértice de Voronoi corresponde a un triángulo en la triangulación Delaunay. Debido a que una triangulación Delaunay es una gráfica plana, una triangulación puede considerarse como una cara de esta gráfica plana. Sean *F*, *E* y *V* el número de caras, bordes y vértices de la triangulación Delaunay. Entonces, según la relación de Euler, $F = E - V + 2$. En una triangulación Delaunay, $V = n$ y *E* es cuando mucho $3n - 6$. En consecuencia, *F* es a lo sumo $(3n - 6) - n + 2 = 2n - 4$. Es decir, *el número de vértices de Voronoi es cuando mucho* $2n - 4$.

Ya casi es posible deducir la complejidad temporal del proceso de fusión. Recuerde que en este proceso es necesario encontrar dos conjuntos convexos. El algoritmo que se presentó en el apartado 4-3 no puede aplicarse aquí porque su complejidad temporal es $O(n \log n)$, que es demasiado alta para nuestros fines. A continuación se demostrará que estos conjuntos convexos pueden encontrarse fácilmente, ya que para construir estos conjuntos convexos es posible usar $VD(S_L)$ y $VD(S_R)$.

Considere la figura 4-25 que muestra un diagrama de Voronoi con cuatro trazos infinitos; a saber, b_{12}, b_{15}, b_{56} y b_{26}. Los puntos asociados con estos trazos infinitos son P_2, P_1, P_5 y P_6. De hecho, ahora es posible construir el convex hull uniendo estos puntos, lo cual se muestra con las líneas discontinuas en la figura 4-25.

FIGURA 4-25 Construcción de un convex hull a partir de un diagrama de Voronoi.



Después de que se ha elaborado un diagrama de Voronoi, el convex hull puede construirse al analizar todos los bordes de Voronoi hasta que se encuentra un trazo infinito. Sea P_i el punto a la izquierda de este trazo infinito. Entonces P_i es el vértice de un convex hull. A continuación se analizan los bordes de Voronoi del polígono de Voronoi de P_i hasta que se encuentra un trazo infinito. El proceso anterior se repite hasta que se regresa al trazo inicial. Debido a que cada borde aparece en exactamente dos polígonos de Voronoi, ningún borde se examina más de dos veces. *En consecuencia, una vez que se construye un diagrama de Voronoi, es posible encontrar un convex hull en tiempo* $O(n)$.

A continuación se deduce la complejidad temporal del paso de fusión del algoritmo para construir un diagrama de Voronoi con base en la estrategia divide-y-vencerás:

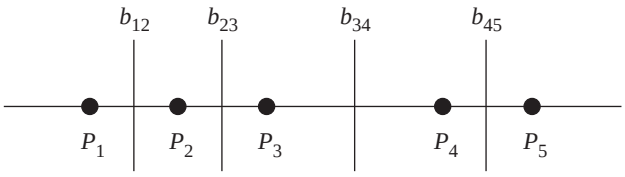
1. Los conjuntos convexos pueden encontrarse en tiempo $O(n)$ porque $VD(S_L)$ y $VD(S_R)$ ya se han construido y es posible encontrar un convex hull si se encuentran los trazos infinitos.
2. Los bordes que unen dos conjuntos convexos para formar un nuevo convex hull pueden determinarse en tiempo $O(n)$. Esto se explicó en el apartado 4-3.
3. Debido a que en $VD(S_L)$ y $VD(S_R)$ hay a lo sumo $3n - 6$ bordes y a que *HP* contiene a lo sumo n segmentos (debido a que *HP* es monótono en y), todo el proceso de construcción de *HP* requiere cuando mucho $O(n)$ pasos.

Así, el proceso de fusión del algoritmo de construcción del diagrama de Voronoi es $O(n)$. Por tanto,

$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \\ &= O(n \log n). \end{aligned}$$

Se concluye que la *complejidad temporal del algoritmo divide-y-vencerás para construir un diagrama de Voronoi es O*(n *log* n). A continuación se demostrará que éste es un algoritmo óptimo. Considere un conjunto de puntos en una recta. El diagrama de Voronoi de este conjunto de puntos consta de un conjunto de rectas bisectrices, como se muestra en la figura 4-26. Después que se han construido estas rectas, una exploración lineal de estos bordes de Voronoi lleva a cabo la función de ordenamiento. En otras palabras, *el problema del diagrama de Voronoi no puede ser más sencillo que el problema de ordenamiento*. En consecuencia, una cota inferior del problema del diagrama de Voronoi es $\Omega(n \log n)$ y así se concluye que el algoritmo es óptimo.

FIGURA 4-26 Diagrama de Voronoi para un conjunto de puntos en una recta.



4-5 Aplicaciones de los diagramas de Voronoi

Se ha analizado el concepto de diagramas de Voronoi y se ha demostrado cómo usar la estrategia divide-y-vencerás para construir un diagrama de Voronoi. En este apartado se demostrará que los diagramas de Voronoi poseen muchas aplicaciones interesantes. Estas aplicaciones, por supuesto, no guardan ninguna relación con la estrategia divide-y-vencerás. Mediante la presentación de aplicaciones de los diagramas de Voronoi esperamos estimular el interés del lector en la geometría computacional.

El problema euclidiano de búsqueda del vecino más cercano

La primera aplicación consiste en resolver el problema euclidiano de búsqueda del vecino más cercano, que se define como sigue: *se tienen un conjunto con n puntos planos: $P_1, P_2,\ldots, P_n$ y un punto de prueba P. El problema consiste en encontrar un vecino más cercano de P de entre los P_i y la distancia que se utiliza es la distancia euclidiana.*

Un método directo para resolver este problema consiste en aplicar una búsqueda exhaustiva. Este algoritmo podría ser un algoritmo $O(n)$. Al usar el diagrama de Voronoi, el tiempo de búsqueda puede reducirse a $O(\log n)$ con tiempo de procesamiento previo $O(n \log n)$.

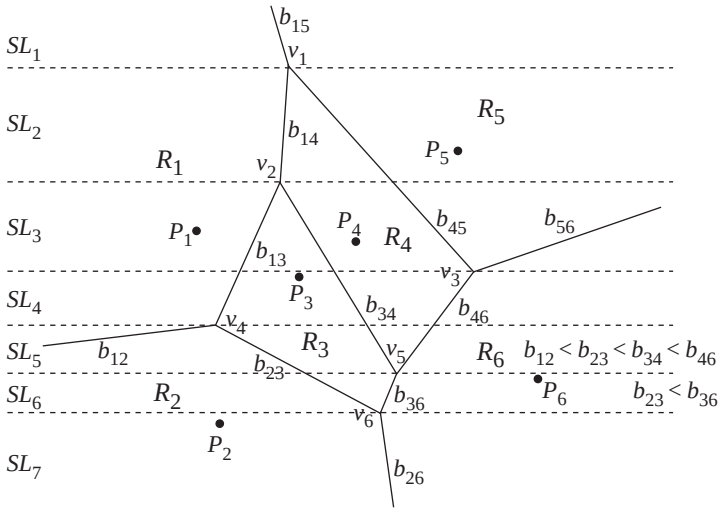
Es posible usar el diagrama de Voronoi para resolver el problema euclidiano de búsqueda del vecino más cercano debido a las propiedades fundamentales de los diagramas de Voronoi. Observe que el diagrama de Voronoi divide todo el plano en regiones $R_1, R_2,\ldots, R_n$. Dentro de cada región R_i hay un punto P_i. Si un punto de prueba cae dentro de la región R_i, entonces su vecino más cercano, de entre todos los puntos, es P_i. En consecuencia, una búsqueda exhaustiva podría evitarse transformando simplemente el problema en un problema de localización de una región. Es decir, si es posible determinar la región R_i en que se encuentra un punto de prueba, entonces se puede determinar un vecino más cercano de este punto de prueba.

Un diagrama de Voronoi es una gráfica plana. En consecuencia, es posible usar las propiedades especiales de una gráfica plana como se ilustra en la figura 4-27. En esta figura se ha vuelto a trazar el diagrama de Voronoi de la figura 4-22. Observe que hay seis vértices de Voronoi. El primer paso es ordenar estos vértices según sus valores y. Así, como se muestra en la figura 4-27, los vértices de Voronoi se identifican como $V_1, V_2,\ldots, V_6$ según sus valores decrecientes y. Para cada vértice de Voronoi se traza una recta horizontal que pase por este vértice. Tales rectas horizontales dividen todo el espacio en franjas, como se muestra en la figura 4-27.

Dentro de cada franja hay vértices de Voronoi que pueden ordenarse linealmente, que dividen nuevamente cada franja en regiones. Considere SL_5 en la figura 4-27. Dentro de SL_5 hay tres vértices de Voronoi: b_{23}, b_{34} y b_{46}. Estos vértices pueden ordenarse como

$$b_{23} < b_{34} < b_{46}.$$

FIGURA 4-27 Aplicación de los diagramas de Voronoi para resolver el problema euclidiano de búsqueda del vecino más cercano.



Estos tres bordes dividen la franja en cuatro regiones: R_2, R_3, R_4 y R_6. Si entre los bordes b_{23} y b_{34} se encuentra un punto, debe estar en R_3. Si se encuentra a la derecha de b_{46}, debe estar dentro de la región R_6. Debido a que estos bordes están ordenados, es posible usar una búsqueda binaria para determinar la ubicación de un punto de prueba en tanto concierna a los bordes de Voronoi.

Esencialmente, el algoritmo euclidiano de búsqueda del vecino más cercano consta de dos pasos fundamentales:

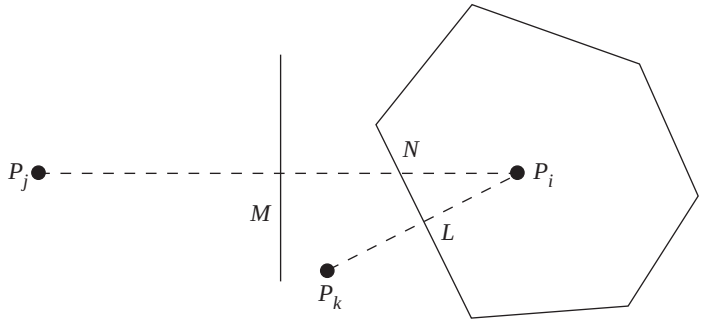
1. Efectuar una búsqueda binaria para determinar la franja en que se encuentra el punto de prueba. Debido a que cuando mucho hay $O(n)$ vértices de Voronoi, la búsqueda puede hacerse en tiempo $O(\log n)$.
2. Dentro de cada franja, realizar una búsqueda binaria para determinar la región en que se encuentra este punto. Debido a que cuando mucho hay $O(n)$ vértices de Voronoi, esta búsqueda puede hacerse en tiempo $O(\log n)$.

El tiempo total de búsqueda es $O(\log n)$. Resulta fácil darse cuenta de que el tiempo de procesamiento previo es $O(n \log n)$, que esencialmente es el tiempo necesario para construir el diagrama de Voronoi.

El problema euclidiano de todos los vecinos más cercanos

La siguiente aplicación es resolver el problema euclidiano de todos los vecinos más cercanos. Se tiene un conjunto con n puntos planos: P_1, P_2,…, P_n. El problema euclidiano del par más cercano consiste en encontrar un vecino más cercano de todo P_i. Este problema puede resolverse fácilmente usando el diagrama de Voronoi debido a la siguiente propiedad de los diagramas de Voronoi. *Si P es un vecino más cercano de P_i, entonces P_i y P_j comparten el mismo borde de Voronoi. Además, el punto medio de $\overline{P_iP_j}$ está ubicado exactamente en este borde de Voronoi compartido comúnmente*. Demostraremos esta propiedad por contradicción. Suponga que P_i y P_j no comparten el mismo borde de Voronoi. Por la definición de polígonos de Voronoi, la bisectriz perpendicular de $\overline{P_iP_j}$ debe estar fuera del polígono de Voronoi asociado con P_i. Sea $\overline{P_iP_j}$ que corta a la bisectriz en M y a algún borde de Voronoi en N, como se ilustra en la figura 4-28.

FIGURA 4-28 Ilustración de la propiedad del vecino más cercano de los diagramas de Voronoi.



Suponga que el borde de Voronoi está entre P_i y P_k y que $\overline{P_iP_k}$ corta al borde de Voronoi en L. Entonces se tiene

$$\overline{P_iN} < \overline{P_iM}$$

y $$\overline{P_iL} < \overline{P_iN}.$$

Esto significa que

$$\overline{P_iP_k} = 2\overline{P_iL} < 2\overline{P_iN} < 2\overline{P_iM} = \overline{P_iP_j}.$$

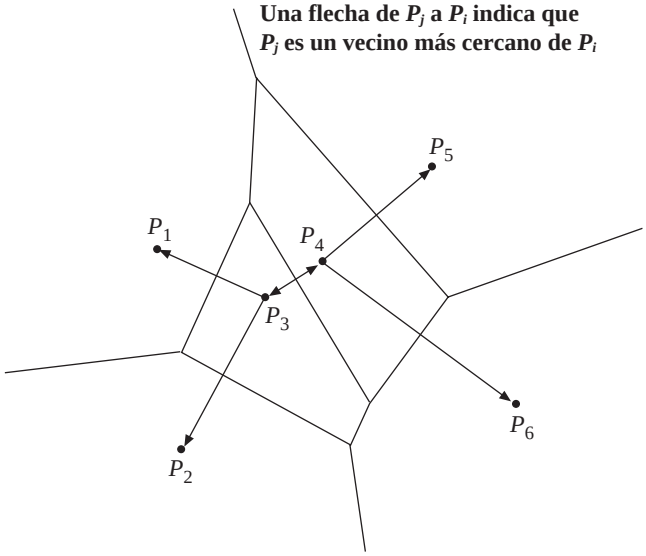
Lo anterior es imposible porque P_j es un vecino más cercano de P_i.

Dada la propiedad anterior, el problema euclidiano de todos los vecinos más cercanos puede resolverse analizando todos los bordes de Voronoi de cada polígono de

Voronoi. Debido a que cada borde de Voronoi es compartido exactamente por dos polígonos de Voronoi, ningún borde de Voronoi se analiza dos veces. Es decir, este problema euclidiano de todos los vecinos más cercanos puede resolverse en tiempo lineal una vez que se ha construido el diagrama de Voronoi. Así, este problema puede resolverse en tiempo $O(n \log n)$.

En la figura 4-29 se ha vuelto a trazar la figura 4-22. Para P_4, es necesario analizar cuatro bordes y se encontrará que el vecino más cercano de P_4 es P_3.

FIGURA 4-29 La relación de todos los vecinos más cercanos.



4-6 LA TRANSFORMADA RÁPIDA DE FOURIER

El problema de la transformada rápida de Fourier consiste en resolver lo siguiente:

$$A_j = \sum_{k=0}^{n-1} a_k e^{\frac{2\pi ijk}{n}}, 0 \le j \le n-1$$

donde $i = \sqrt{-1}$ y $a_0, a_1, \ldots, a_{n-1}$ son números dados. Un método directo requiere $O(n^2)$ pasos. Si se aplica la estrategia divide-y-vencerás, la complejidad temporal puede reducirse a $O(n \log n)$.

Para simplificar el análisis, sea $w_n = e^{\frac{2\pi i}{n}}$. Así, la transformada de Fourier es para calcular lo siguiente:

$$A_j = a_0 + a_1 w_n^j + a_2 w_n^{2j} + \cdots + a_{n-1} w_n^{(n-1)j}.$$

Sea $n = 4$. Así, se tiene

$$\begin{aligned}
A_0 &= a_0 + a_1 + a_2 + a_3 \\
A_1 &= a_0 + a_1 w_4 + a_2 w_4^2 + a_3 w_4^3 \\
A_2 &= a_0 + a_1 w_4^2 + a_2 w_4^4 + a_3 w_4^6 \\
A_3 &= a_0 + a_1 w_4^3 + a_2 w_4^6 + a_3 w_4^9.
\end{aligned}$$

Las ecuaciones anteriores pueden volverse a agrupar como se muestra a continuación:

$$\begin{aligned}
A_0 &= (a_0 + a_2) + (a_1 + a_3) \\
A_1 &= (a_0 + a_2 w_4^2) + w_4(a_1 + a_3 w_4^2) \\
A_2 &= (a_0 + a_2 w_4^4) + w_4^2(a_1 + a_3 w_4^4) \\
A_3 &= (a_0 + a_2 w_4^6) + w_4^3(a_1 + a_3 w_4^6)
\end{aligned}$$

Debido a que $w_n^2 = w_{n/2}$ y $w_n^{n+k} = w_n^k$, se tiene

$$\begin{aligned}
A_0 &= (a_0 + a_2) + (a_1 + a_3) \\
A_1 &= (a_0 + a_2 w_2) + w_4(a_1 + a_3 w_2) \\
A_2 &= (a_0 + a_2) + w_4^2(a_1 + a_3) \\
A_3 &= (a_0 + a_2 w_4^2) + w_4^3(a_1 + a_3 w_4^2) \\
&= (a_0 + a_2 w_2) + w_4^3(a_1 + a_3 w_2).
\end{aligned}$$

Sea

$$\begin{aligned}
B_0 &= a_0 + a_2 \\
C_0 &= a_1 + a_3 \\
B_1 &= a_0 + a_2 w_2 \\
C_1 &= a_1 + a_3 w_2.
\end{aligned}$$

Entonces

$$A_0 = B_0 + w_4^0 C_0$$
$$A_1 = B_1 + w_4^1 C_1$$
$$A_2 = B_0 + w_4^2 C_0$$
$$A_3 = B_1 + w_4^3 C_1.$$

La ecuación anterior muestra que la estrategia divide-y-vencerás puede aplicarse elegantemente para resolver el problema de la transformada de Fourier. Basta calcular B_0, C_0, B_1 y C_1. Las A_j pueden calcularse fácilmente. En otras palabras, una vez que se obtiene A_0, A_2 puede obtenerse de inmediato. De manera semejante, una vez que se obtiene A_1, A_3 puede obtenerse directamente.

Pero, ¿qué ocurre con las B_i y con las C_i? Observe que las B_i son la transformada de Fourier de los datos de entrada con número impar, y que las C_i representan la transformada de Fourier de los datos de entrada con número par. Esto constituye la base de la aplicación de la estrategia divide-y-vencerás al problema de la transformada de Fourier. Un problema grande se divide en dos subproblemas, que se resuelven de manera recursiva y luego se fusionan las soluciones.

Considere A_j en el caso general.

$$\begin{aligned} A_j &= a_0 + a_1 w_n^j + a_2 w_n^{2j} + \cdots + a_{n-1} w_n^{(n-1)j} \\ &= (a_0 + a_2 w_n^{2j} + a_4 w_n^{4j} + \cdots + a_{n-2} w_n^{(n-2)j}) \\ &\quad + w_n^j (a_1 + a_3 w_n^{2j} + a_5 w_n^{4j} + \cdots + a_{n-1} w_n^{(n-2)j}) \\ &= (a_0 + a_2 w_{n/2}^j + a_4 w_{n/2}^{2j} + \cdots + a_{n-2} w_{n/2}^{\frac{(n-2)j}{2}}) \\ &\quad + w_n^j (a_1 + a_3 w_{n/2}^j + a_5 w_{n/2}^{2j} + \cdots + a_{n-1} w_{n/2}^{\frac{(n-2)j}{2}}). \end{aligned}$$

B_j y C_j se definen como sigue:

$$B_j = a_0 + a_2 w_{n/2}^j + a_4 w_{n/2}^{2j} + \cdots + a_{n-2} w_{n/2}^{\frac{(n-2)j}{2}}$$

y $$C_j = a_1 + a_3 w_{n/2}^j + a_5 w_{n/2}^{2j} + \cdots + a_{n-1} w_{n/2}^{\frac{(n-2)j}{2}}.$$

Entonces

$$A_j = B_j + w_n^j C_j.$$

También puede demostrarse que

$$A_{j+n/2} = B_j + w_n^{j+n/2} C_j.$$

Para $n = 2$, la transformada de Fourier es como sigue:

$$A_0 = a_0 + a_1$$
$$A_1 = a_0 - a_1.$$

A continuación se presenta el algoritmo de la transformada rápida de Fourier basado en el método divide-y-vencerás:

Algoritmo 4-6 □ Un algoritmo de la transformada rápida de Fourier basado en la estrategia divide-y-vencerás

Input: $a_0, a_1, \ldots, a_{n-1}$, $n = 2^k$.

Output: $A_j = \sum_{k=0}^{n-1} a_k e^{\frac{2\pi ijk}{n}}$ para $j = 0, 1, 2, \ldots, n-1$.

Paso 1. Si $n = 2$,

$A_0 = a_0 + a_1$

$A_1 = a_0 - a_1$

Return.

Paso 2. Recursivamente buscar los coeficientes de la transformada de Fourier de $a_0, a_2, \ldots, a_{n-2}$($a_1, \ldots, a_{n-1}$). Los coeficientes se denotan por $B_0, B_1, \ldots, B_{n/2}$($C_0, C_1, \ldots, C_{n/2}$).

Paso 3. Para $j = 0$ hasta $j = \frac{n}{2} - 1$

$A_j = B_j + w_n^j C_j$

$A_{j+n/2} = B_j + w_n^{j+n/2} C_j$.

Resulta evidente que la complejidad temporal del algoritmo anterior es $O(n \log n)$. Hay una transformada inversa de Fourier que transforma $A_0, A_1, \ldots, A_{n-1}$ de vuelta en $a_0, a_1, \ldots, a_{n-1}$ como sigue:

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} A_k e^{\frac{-2\pi ijk}{n}} \quad \text{para} \quad j = 0, 1, 2, \ldots, n-1.$$

Se considerará un ejemplo. Sean $a_0 = 1$, $a_1 = 0$, $a_2 = -1$, $a_3 = 0$. Entonces

$$B_0 = a_0 + a_2 = 1 + (-1) = 0$$
$$B_1 = a_0 - a_2 = 1 - (-1) = 2$$
$$C_0 = a_1 + a_3 = 0 + 0 = 0$$
$$C_1 = a_1 - a_3 = 0 - 0 = 0$$
$$w_4 = e^{\frac{2\pi i}{4}} = i$$
$$w_4^2 = -1$$
$$w_4^3 = -i.$$

Así,

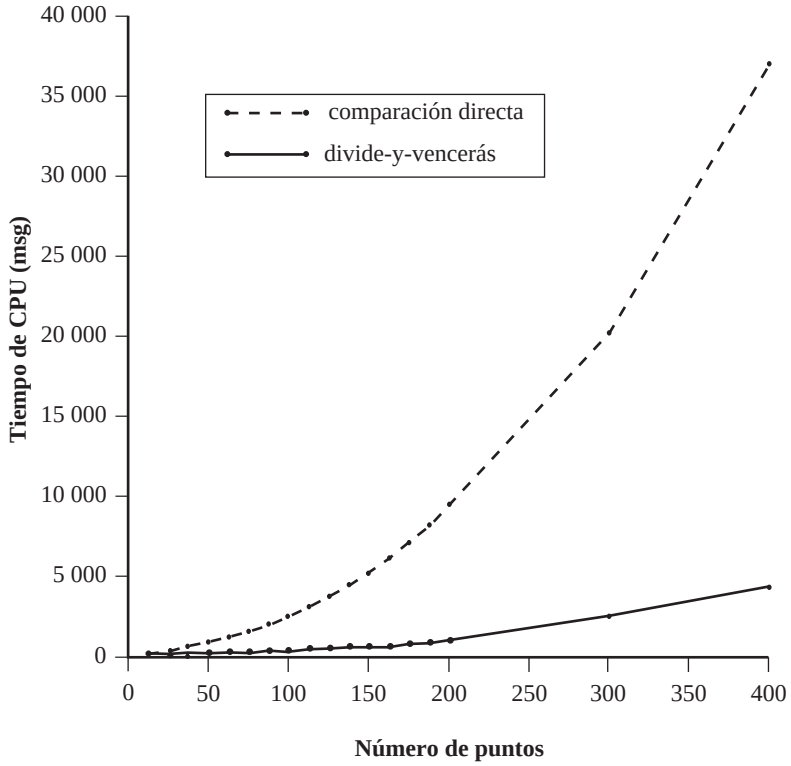
$$A_0 = B_0 + w_4^0 C_0 = B_0 + C_0 = 0 + 0 = 0$$
$$A_1 = B_1 + w_4 C_1 = 2 + 0 = 2$$
$$A_2 = B_0 + w_4^2 C_0 = 0 - 0 = 0$$
$$A_3 = B_1 + w_4^3 C_1 = 2 + 0 = 2.$$

Debe ser fácil ver que la transformada inversa de Fourier debe transformar correctamente las A_i de vuelta en las a_i.

4-7 LOS RESULTADOS EXPERIMENTALES

Para demostrar que la estrategia divide-y-vencerás es útil, se implementó el algoritmo del par más cercano basado en los algoritmos divide-y-vencerás, y el algoritmo de comparación directa que examina exhaustivamente cada par de puntos. Ambos programas se implementaron en una computadora personal IBM. Los resultados experimentales se resumen en la figura 4-30. Como se muestra en esa figura, mientras n es pequeña el método directo se desempeña mejor. No obstante, a medida que n se hace grande, el algoritmo divide-y-vencerás es mucho más rápido. Por ejemplo, cuando n es igual a 7 000, el algoritmo divide-y-vencerás es casi 200 veces más rápido que el método directo. Los resultados son bastante predecibles porque la complejidad temporal del método directo es $O(n^2)$ y la complejidad temporal del algoritmo divide-y-vencerás es $O(n \log n)$.

FIGURA 4-30 Resultados experimentales del problema de encontrar el par más cercano.



4-8 Notas y referencias

La estrategia divide-y-vencerás suele aplicarse para resolver problemas de geometría computacional. Consulte a Preparata y Shamos (1985) y a Lee y Preparata (1984). La estrategia divide-y-vencerás también se analiza en Horowitz y Sahni (1978) y en Aho, Hopcroft y Ullman (1974).

Para un análisis completo del problema de máximos, consulte a Pohl (1972). El problema del par más cercano y el problema de cómo encontrar los máximos fueron resueltos por Bentley (1980).

Los diagramas de Voronoi fueron estudiados primero por Voronoi (1908). De nuevo, más información sobre los diagramas de Voronoi puede consultarse en Preparata y Shamos (1985). Para una generalización de los diagramas de Voronoi a órdenes y dimensiones superiores, consulte las obras de Lee (1982) y Brown (1979).

En Levcopoulos y Lingas (1987) y Preparata y Shamos (1985) pueden encontrarse diferentes algoritmos de conjuntos convexos. El algoritmo de Graham fue propuesto por Graham (1972). Aplicando la estrategia divide-y-vencerás también puede encontrarse un convex hull tridimensional. Esto fue indicado por Preparata y Hong (1977).

El hecho de que la transformada de Fourier puede resolverse con el método de divide-y-vencerás fue señalado por Cooley y Tukey (1965). Gentleman y Sande (1966) analizan aplicaciones de la transformada rápida de Fourier. El libro de Brigham (Brigham, 1974) está dedicado totalmente a la transformada rápida de Fourier.

La estrategia divide-y-vencerás puede aplicarse para obtener un algoritmo eficiente de multiplicación matricial. Consulte la obra de Strassen (1969).

4-9 BIBLIOGRAFÍA ADICIONAL

La estrategia divide-y-vencerás sigue siendo un tema interesante para muchos investigadores. Es especialmente eficaz para geometría computacional. Para investigación adicional, se recomiendan los siguientes artículos: Aleksandrov y Djidjev (1996); Bentley (1980); Bentley y Shamos (1978); Blankenagel y Gueting (1990); Bossi, Cocco y Colussi (1983); Chazelle, Drysdale y Lee (1986); Dwyer (1987); Edelsbrunner, Maurer, Preparata, Rosenberg, Welzl y Wood (1982); Gilbert, Hutchinson y Tarjan (1984); Gueting (1984); Gueting y Schilling (1987); Karp (1994); Kumar, Kiran y Pandu (1987); Lopez y Zapata (1994); Monier (1980); Oishi y Sugihara (1995); Reingold y Supowit (1983); Sykora y Vrto (1993); Veroy (1988); Walsh (1984), y Yen y Kuo (1997).

Para algunos artículos bastante interesantes y de reciente publicación, consulte Abel (1990); Derfel y Vogl (2000); Dietterich (2000); Even, Naor, Rao y Schieber (2000); Fu (2001); Kamidoi, Wakabayashi y Yoshida (2002); Lee y Sheu (1992); Liu (2002); Lo, Rajopadhye, Telle y Zhong (1996); Melnik y Garcia-Molina (2002); Messinger, Rowe y Henry (1991); Neogi y Saha (1995); Rosler (2001); Rosler y Ruschendorf (2001); Roura (2001); Tisseur y Dongarra (1999); Tsai y Katsaggelos (1999); Verma (1997); Wang (1997); Wang (2000); Wang, He, Tang y Wee (2003); Yagle (1998), y Yoo, Smith y Gopalarkishnan (1997).

Ejercicios

4.1 La búsqueda binaria, ¿usa la estrategia divide-y-vencerás?

4.2 Para multiplicar directamente dos números de n bits u y v se requieren $O(n^2)$ pasos. Al aplicar el método divide-y-vencerás, el número puede separarse en dos partes iguales y calcular el producto aplicando el método siguiente:

$$uv = (a \cdot 2^{n/2} + b) \cdot (c \cdot 2^{n/2} + d) = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd.$$

Si $ad + bc$ se calcula como $(a + b)(c + d) - ac - bd$, ¿cuál es el tiempo de cálculo?

4.3 Demuestre que en quick sort, la pila máxima necesaria es $O(\log n)$.

4.4 Implemente el algoritmo de la transformada rápida de Fourier con base en el método divide-y-vencerás. Compárelo con el método directo.

4.5 Implemente el algoritmo para encontrar rangos con base en el método divide-y-vencerás. Compárelo con el método directo.

4.6 Sea $T\left(\frac{n^2}{2^r}\right) = nT(n) + bn^2$, donde r es un entero y $r \leq 1$. Encuentre $T(n)$.

4.7 Lea la sección 3-7 del libro de Horowitz y Sahni (1978) para conocer el método de multiplicación matricial de Strassen con base en el algoritmo divide-y-vencerás.

4.8 Sea

$$T(n) = \begin{cases} b & \text{para } n = 1 \\ aT(n/c) + bn & \text{para } n > 1 \end{cases}$$

donde a, b y c son constantes no negativas.

Demuestre que si n es una potencia de c, entonces

$$T(n) = \begin{cases} O(n) & \text{si } a < c \\ O(n \log_a n) & \text{si } a = c \\ O(n^{\log_c a}) & \text{si } a > c. \end{cases}$$

4.9 Demuestre que si $T(n) = mT(n/2) + an^2$, entonces $T(n)$ es satisfecho por

$$T(n) = \begin{cases} O(n^{\log m}) & \text{si } m > 4 \\ O(n^2 \log n) & \text{si } m = 4 \\ O(n^2) & \text{si } m < 4. \end{cases}$$

4.10 Una clase muy especial de algoritmo de ordenamiento, basado también en el algoritmo divide-y-vencerás, es el ordenamiento de fusión impar-par (odd-even merge sorting), inventado por Batcher (1968). Lea la sección 7.4 del libro de Liu (1977) para conocer este algoritmo de ordenamiento. Este algoritmo, ¿es idóneo como algoritmo secuencial? ¿Por qué? (Éste es un famoso algoritmo de ordenamiento paralelo.)

4.11 Diseñe un algoritmo con tiempo $O(n \log n)$ para encontrar la subsecuencia monótona creciente más larga de una secuencia de n números.

capítulo

5

La estrategia de árboles de búsqueda

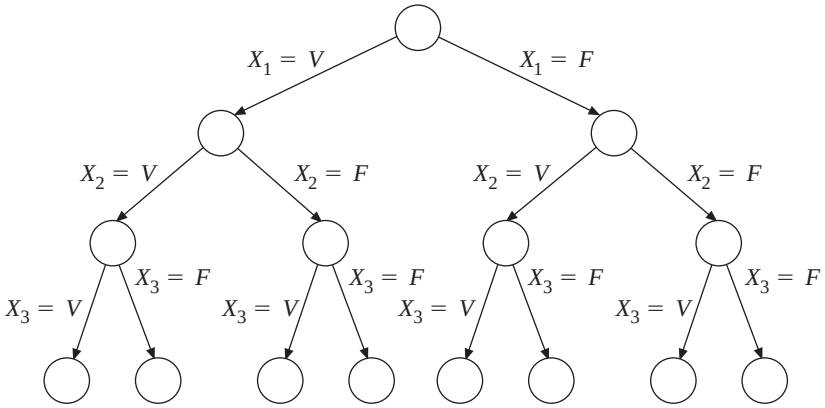
En este capítulo se demostrará que las soluciones de muchos problemas pueden representarse mediante árboles, por lo que resolver tales problemas se convierte en un problema de árboles de búsqueda. Se considerará el problema de satisfacibilidad que se abordará en el capítulo 8. Dado un conjunto de cláusulas lógicas, un método para determinar si este conjunto de cláusulas es satisfacible consiste en examinar todas las combinaciones posibles. Es decir, si hay n variables x_1, x_2, …, x_n, entonces simplemente se analizan todas las 2^n combinaciones posibles. En cada combinación, a x_i se le asigna un valor ya sea V (Verdadera) o F (Falsa). Suponga que $n = 3$. Entonces es necesario analizar las siguientes combinaciones:

x_1	x_2	x_3
F	F	F
F	F	V
F	V	F
F	V	V
V	F	F
V	F	V
V	V	F
V	V	V

Las ocho combinaciones anteriores, por otra parte, pueden representarse mediante un árbol, como se muestra en la figura 5-1. ¿Por qué es informativo el árbol? Porque una representación de árbol de las combinaciones muestra que en el nivel superior realmente hay dos clases de combinaciones:

Clase 1: Las combinaciones en que $x_1 = V$.
Clase 2: Las combinaciones en que $x_1 = F$.

FIGURA 5-1 Representación de árbol de ocho combinaciones.

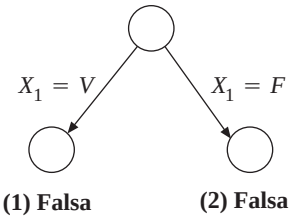


Así, de manera recursiva es posible clasificar cada clase de combinaciones en dos subclases. Con este punto de vista, es posible determinar la satisfacibilidad sin necesidad de analizar todas las combinaciones: sólo todas las clases de combinaciones. Por ejemplo, suponga que se tiene el siguiente conjunto de cláusulas:

$$\neg x_1 \quad \textbf{(1)}$$
$$x_1 \quad \textbf{(2)}$$
$$x_2 \vee x_5 \quad \textbf{(3)}$$
$$x_3 \quad \textbf{(4)}$$
$$\neg x_2 \quad \textbf{(5)}$$

Ahora es posible expandir un árbol parcial, que se muestra en la figura 5-2.

FIGURA 5-2 Árbol parcial para determinar el problema de satisfacibilidad.



Con base en el árbol de la figura 5-2, puede verse fácilmente que la combinación de $x_1 = V$ hace falsa la cláusula (1) y que la combinación de $x_1 = F$ hace falsa la cláusula (2). Debido a que en toda combinación a x_1 se asigna V o F, no es necesario examinar cada combinación. A continuación se establece la insatisfacibilidad del conjunto de cláusulas anterior.

Muchos otros problemas semejantes también pueden resolverse aplicando técnicas con árbol de búsqueda. Considere el famoso problema del rompecabezas (pasatiempo) de 8 piezas (8-puzzle). En la figura 5-3 se muestra un cuadrado de 3 por 3 que puede contener 9 piezas (mosaicos), aunque sólo tiene ocho piezas móviles numeradas, y una casilla vacía. El problema consiste en mover las piezas numeradas hasta alcanzar el estado final, que se muestra en la figura 5-4. Las piezas numeradas sólo pueden moverse horizontal o verticalmente hacia la casilla vacía. Así, para el arreglo original que se muestra en la figura 5-3, sólo hay dos movimientos posibles (mover el 3 o el 4), como se muestra en la figura 5-5.

FIGURA 5-3 Posición inicial del problema del rompecabezas de 8 piezas.

2	3	
5	1	4
6	8	7

FIGURA 5-4 Meta final del problema del rompecabezas de 8 piezas.

1	2	3
8		4
7	6	5

El problema del rompecabezas de 8 piezas se convierte en un problema de árbol de búsqueda porque gradualmente es posible ir expandiendo el árbol solución. El problema se resuelve cuando se alcanza el nodo que representa la meta final. Esto se presentará en apartados posteriores.

Por último, se demostrará que el espacio solución del problema del ciclo Hamiltoniano también puede representarse convenientemente por medio de un árbol. Dada una gráfica $G = (V, E)$, que es una gráfica conexa con n vértices, un ciclo Hamiltoniano es una ruta de viaje redondo a lo largo de n bordes de G que pasa una vez por cada vértice y después regresa a su posición inicial. Considere la figura 5-6. La ruta representada por la siguiente secuencia es un ciclo Hamiltoniano: 1, 2, 3, 4, 7, 5, 6, 1. Considere la figura 5-7. En esta gráfica no hay ningún ciclo Hamiltoniano.

El problema del ciclo Hamiltoniano consiste en determinar si una gráfica determinada contiene o no un ciclo Hamiltoniano. Éste es un problema NP completo. No

FIGURA 5-5 Dos movimientos posibles para una posición inicial del problema del rompecabezas de 8 piezas.

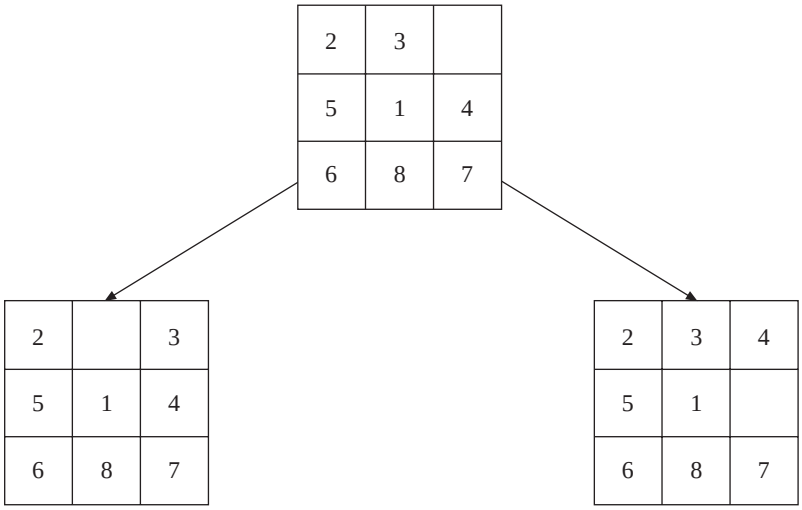


FIGURA 5-6 Gráfica que contiene un ciclo Hamiltoniano.

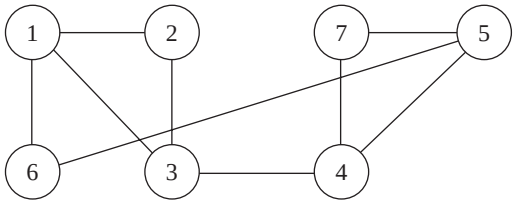
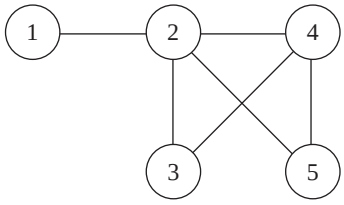


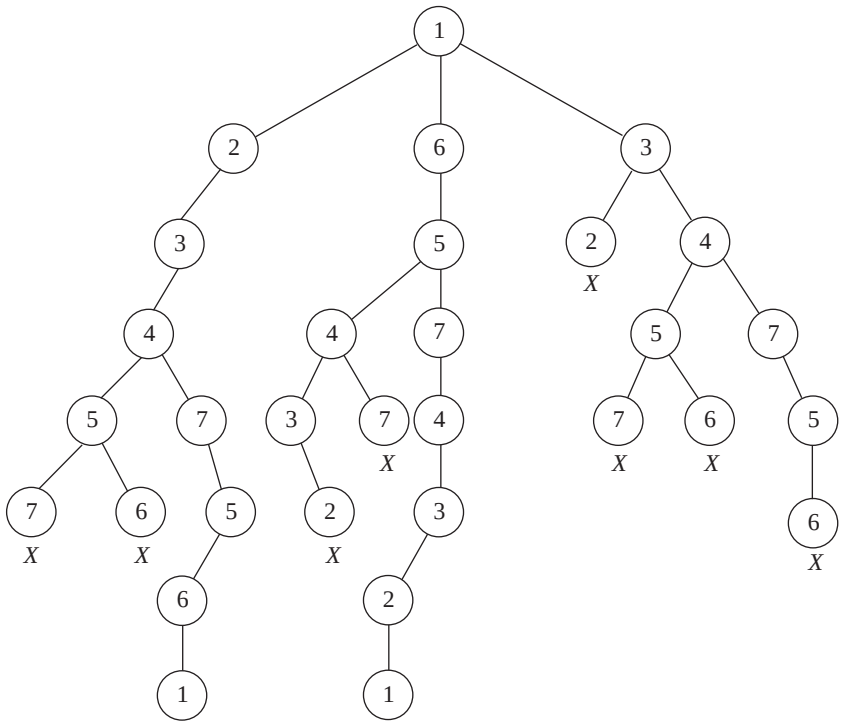
FIGURA 5-7 Gráfica que no contiene ningún ciclo Hamiltoniano.



obstante, sigue siendo posible resolverlo mediante el análisis de todas las soluciones posibles, mismas que pueden representarse de manera conveniente por medio de un árbol. Observe que un ciclo Hamiltoniano debe pasar por cada nodo. En consecuencia, puede suponerse que el nodo 1 es el nodo inicial. Considere nuevamente la figura 5-6.

La búsqueda de un ciclo Hamiltoniano puede describirse mediante un árbol, que se muestra en la figura 5-8. Muestra que hay sólo un ciclo Hamiltoniano, ya que el ciclo 1, 2, 3, 4, 7, 5, 6, 1 es equivalente a 1, 6, 5, 7, 4, 3, 2, 1.

FIGURA 5-8 Representación de árbol sobre la existencia o no de un ciclo Hamiltoniano en la gráfica de la figura 5-6.



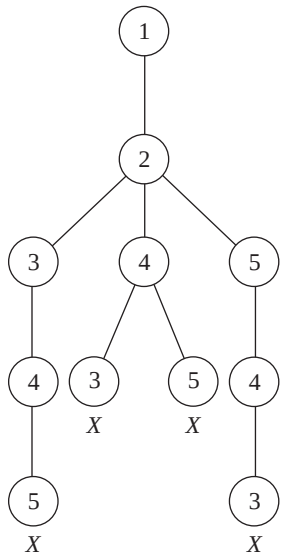
Si se analiza la figura 5-7, su representación de árbol sobre la existencia o no de un ciclo Hamiltoniano en la gráfica correspondiente se muestra en la figura 5-9. En este caso puede verse fácilmente que no existe ningún ciclo Hamiltoniano.

Se ha mostrado que muchos problemas pueden representarse por medio de árboles. En el resto de este capítulo se analizarán las estrategias de poda de árboles para resolver los problemas.

5-1 Búsqueda de primero en amplitud (breadth-first search)

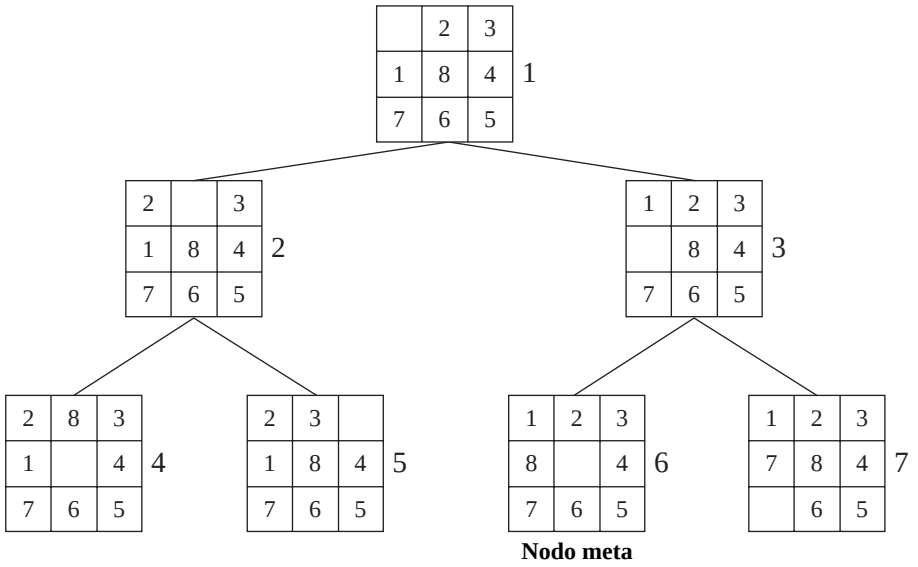
La búsqueda de primero en amplitud es quizá la forma más directa de podar un árbol. En esta búsqueda, todos los nodos de un nivel del árbol se revisan antes de analizar

FIGURA 5-9 Árbol que muestra la inexistencia de ciclo Hamiltoniano.



los nodos del siguiente nivel. En la figura 5-10 se muestra una búsqueda de primero amplitud típica que resuelve un problema del rompecabezas de 8 piezas.

FIGURA 5-10 Árbol de búsqueda producido por una búsqueda de primero amplitud.



Se observa que el nodo 6 representa un nodo meta. En consecuencia, es necesario detener la búsqueda. La estructura de datos básica de la búsqueda de anchura es una cola que contiene todos los nodos expandidos. El siguiente método ilustra la búsqueda de primero amplitud.

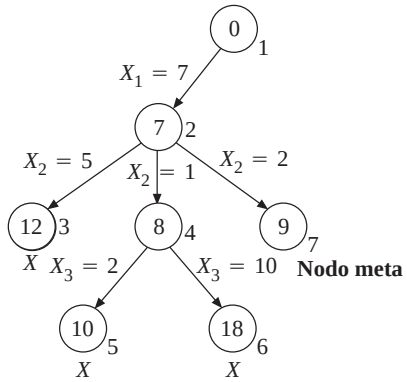
Búsqueda de primero amplitud

Paso 1. Formar una cola de un elemento que consta del nodo raíz.
Paso 2. Probar si el primer elemento en la cola es un nodo meta. En caso afirmativo, detenerse; en caso contrario, ir a paso 3.
Paso 3. Quitar este primer elemento de la cola. Agregar los descendientes (nodos hijos) del primer elemento, en caso de haber alguno, al final de la cola.
Paso 4. Si la cola está vacía, regresar Falla. En caso contrario, ir a paso 2.

5-2 Búsqueda de primero en profundidad (depth-first search)

La búsqueda de primero en profundidad siempre escoge el nodo más profundo para expandirlo. Se considerará la siguiente suma del problema del subconjunto. Se tiene $S = \{7, 5, 1, 2, 10\}$ y se quiere determinar si existe un subconjunto S' tal que la suma de elementos en S' sea igual a 9. Este problema puede resolverse fácilmente mediante el árbol de búsqueda de primero en profundidad que se muestra en la figura 5-11. En esta figura, muchos nodos están terminados porque resulta evidente que no llevan a ninguna solución. Los números encerrados en un círculo en la figura 5-11 representan la suma de un subconjunto. Observe que siempre se selecciona el nodo más profundo para expandirlo en el proceso.

FIGURA 5-11 Una suma del problema del subconjunto resuelta por búsqueda en profundidad.



A continuación se considerará el problema del ciclo Hamiltoniano. Para la gráfica que se muestra en la figura 5-12, es posible encontrar un ciclo Hamiltoniano mediante búsqueda de primero en profundidad, que se muestra en la figura 5-13.

FIGURA 5-12 Gráfica que contiene un ciclo Hamiltoniano.

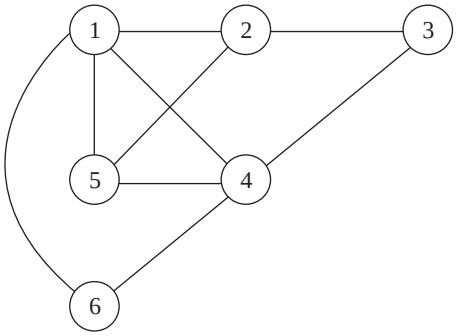
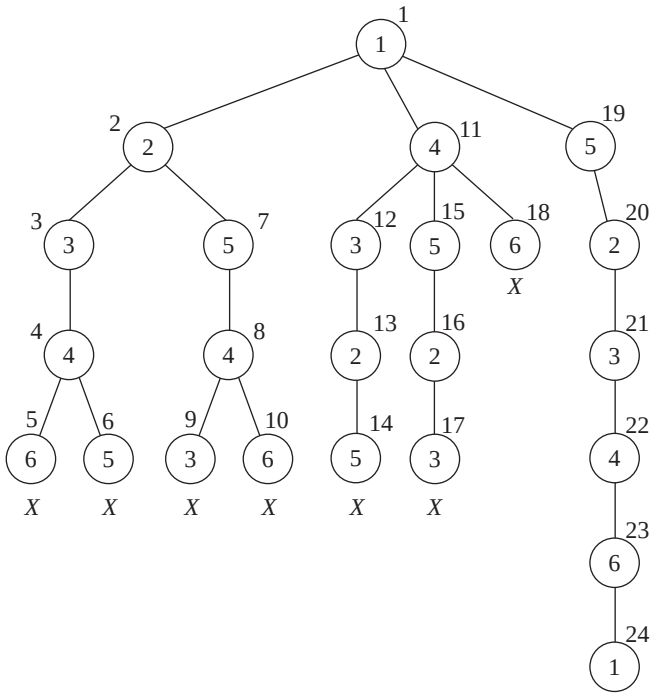


FIGURA 5-13 Ciclo Hamiltoniano producido por búsqueda de primero en profundidad.



Ahora, la búsqueda de primero en profundidad se resume como sigue:

Búsqueda de primero en profundidad

Paso 1. Formar una pila de un elemento que conste del nodo raíz.
Paso 2. Probar si el elemento superior en la pila es un nodo meta. Si es así, detener el proceso; en caso contrario, ir a paso 3.
Paso 3. Quitar este elemento superior de la pila y agregar sus descendientes, en caso de haber alguno, a la parte superior de la pila.
Paso 4. Si la pila está vacía, regresar Falla. En caso contrario, ir a paso 2.

5-3 Método de ascenso de colina (hill climbing)

Después de leer el apartado sobre búsqueda de primero en profundidad, quizás el lector se pregunte sobre un problema: de entre todos los descendientes, ¿qué nodo debe seleccionarse para expandirlo? En esta sección se presentará un método denominado ascenso de colina, que es una variante de la búsqueda de primero en profundidad en la cual se aplica algún método codicioso como ayuda para decidir la dirección en que hay que moverse en el espacio de búsqueda. En términos generales, el método codicioso usa alguna medida heurística para ordenar las opciones. Y, mientras mejor sea la heurística, mejor es el ascenso de colina.

Nuevamente se considerará el problema del rompecabezas de 8 piezas. Suponga que el método codicioso usa la siguiente función de evaluación simple $f(n)$ para ordenar las opciones:

$$f(n) = w(n)$$

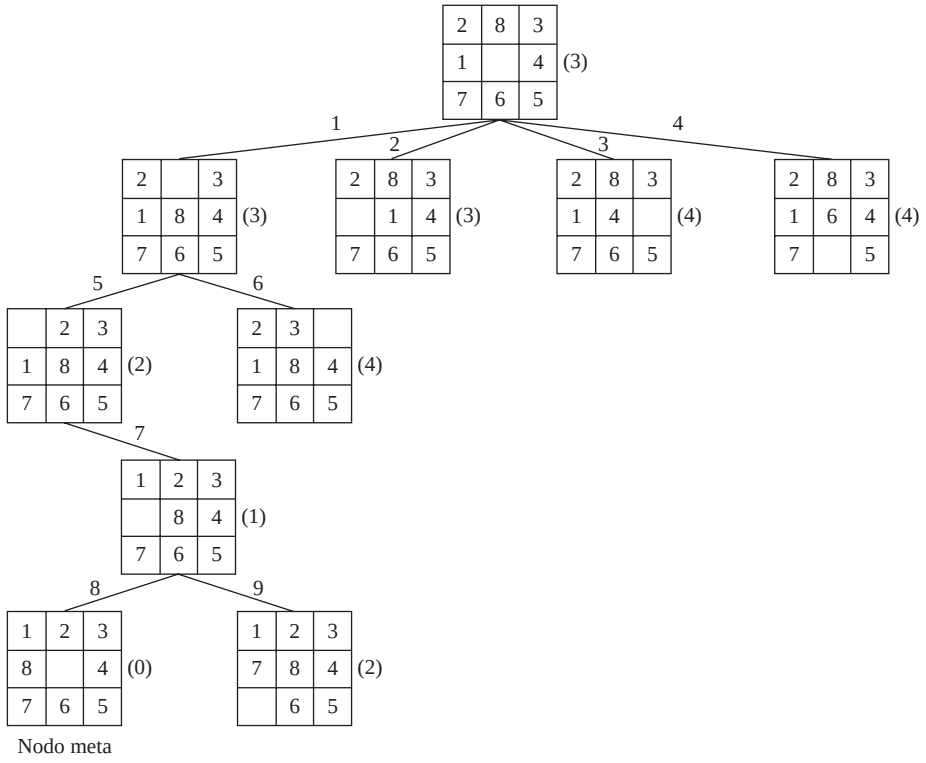
donde $w(n)$ es el número de piezas móviles mal colocadas en el nodo n. Entonces, si el nodo inicial está colocado como se muestra en la figura 5-14, entonces $f(n)$ es igual a 3 porque 1, 2 y 8 están mal colocados.

FIGURA 5-14 Nodo inicial de un problema del rompecabezas de 8 piezas.

2	8	3
1		4
7	6	5

En la figura 5-15 se muestra el resultado de aplicar el método de ascenso de colina usando *f* como heurística para ordenar las opciones de entre todos los descendientes de un nodo. El valor de *f* para cada nodo se muestra en la figura 5-15. Sobre cada borde también se indica el orden en que se desarrollan los nodos. Observe que el método de ascenso de colina sigue usando la búsqueda de primero en profundidad, excepto que de entre todos los descendientes de un nodo, este método selecciona el nodo localmente óptimo para expandirlo.

FIGURA 5-15 Problema del rompecabezas de 8 piezas resuelto con el método de ascenso de colina.



El lector no debe tener una impresión equivocada de que el método de ascenso de colina es extraordinariamente eficiente debido al ejemplo mostrado en la figura 5-15. En esta figura, de entre los primeros nodos expandidos, hay dos nodos que tienen el mismo valor que la función de evaluación. Si se hubieran expandido los otros nodos, para llegar a la solución se requeriría mucho más tiempo.

Método del ascenso de colina

Paso 1. Formar una pila consistente de un único elemento nodo raíz.
Paso 2. Probar si el primer elemento de la pila es un nodo meta. Si es así, detenerse; en caso contrario, ir a paso 3.
Paso 3. Quitar el primer elemento de la pila y expandir el elemento. Agregar los descendientes del elemento eliminado a la pila ordenada por la función de evaluación.
Paso 4. Si la lista está vacía, regresar falla. En caso contrario, ir a paso 2.

5-4 Estrategia de búsqueda de primero el mejor (best-first search)

Esta estrategia constituye una forma para combinar en un solo método las ventajas de la búsqueda del primero en profundidad y la búsqueda del primero en amplitud. En búsqueda de primero el mejor hay una función de evaluación y se selecciona el nodo con el menor costo de entre todos los nodos que se han expandido hasta el momento. Puede verse que el método de búsqueda de primero el mejor, a diferencia del método de ascenso de colina, posee una visión general.

Método de búsqueda de primero el mejor

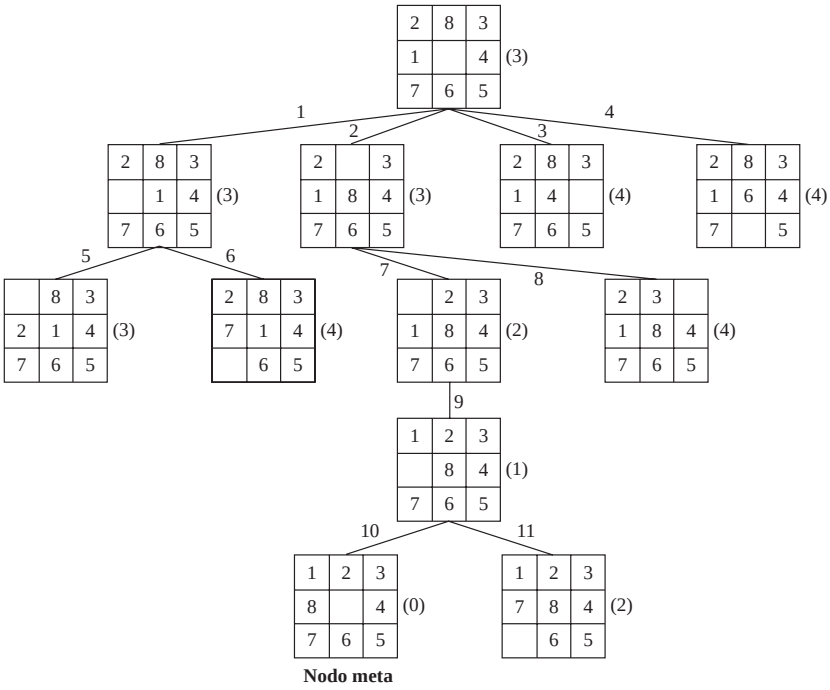
Paso 1. Construir un heap usando la función de evaluación. Primero, formar un heap de 1 elemento (el nodo raíz).
Paso 2. Probar si el elemento raíz en el heap es un nodo meta. Si es así, detenerse; en caso contrario, ir a paso 3.
Paso 3. Quitar el elemento raíz del heap y expandir el elemento. Agregar los descendientes (nodos hijos) del elemento eliminado al heap.
Paso 4. Si el heap está vacío, hay una falla. De otra forma, ir a paso 2.

Si se aplica la misma heurística que con el método de ascenso de colina para esta primera mejor búsqueda, entonces el problema del rompecabezas de 8 piezas se resuelve como se muestra en la figura 5-16.

5-5 Estrategia de ramificar-y-acotar (branch-and-bound)

En las secciones previas se mostró que muchos problemas pueden resolverse aplicando las técnicas del árbol de búsqueda. Observe que ninguno de esos problemas es de optimización. Debe resultar interesante para el lector observar que ninguno de los métodos anteriores puede usarse para resolver un problema de optimización. En esta sección

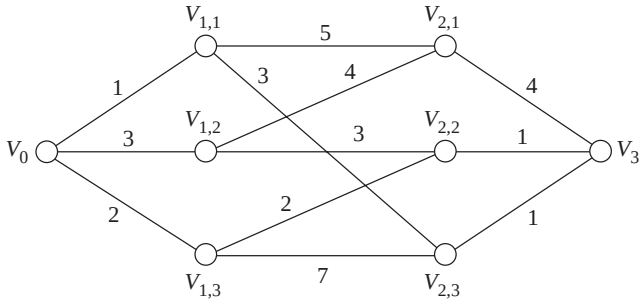
FIGURA 5-16 Problema del rompecabezas de 8 piezas resuelto por el método de búsqueda de primero el mejor.



se presenta la estrategia de ramificar-y-acotar, que quizás es una de las más eficientes para resolver un problema combinatorio grande. Básicamente, sugiere que un problema puede tener soluciones factibles. No obstante, una vez que se descubre que muchas soluciones pueden no ser óptimas, es necesario tratar de acotar el espacio solución.

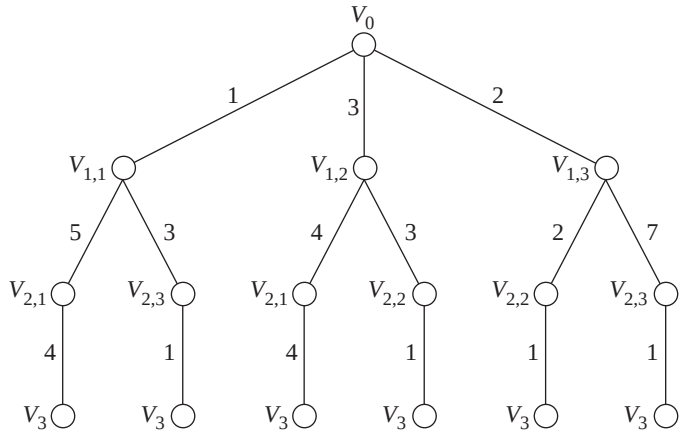
A continuación se explicarán los principios básicos de la estrategia de ramificar-y-acotar utilizando la figura 5-17.

FIGURA 5-17 Problema multietapas de una gráfica de búsqueda.



En la figura 5-17, el problema consiste en encontrar una ruta más corta de V_0 a V_3. Este problema puede resolverse de manera eficiente transformando primero el problema en un problema de árbol de búsqueda como se muestra en la figura 5-18.

FIGURA 5-18 Representación de árbol de soluciones del problema de la figura 5-17.



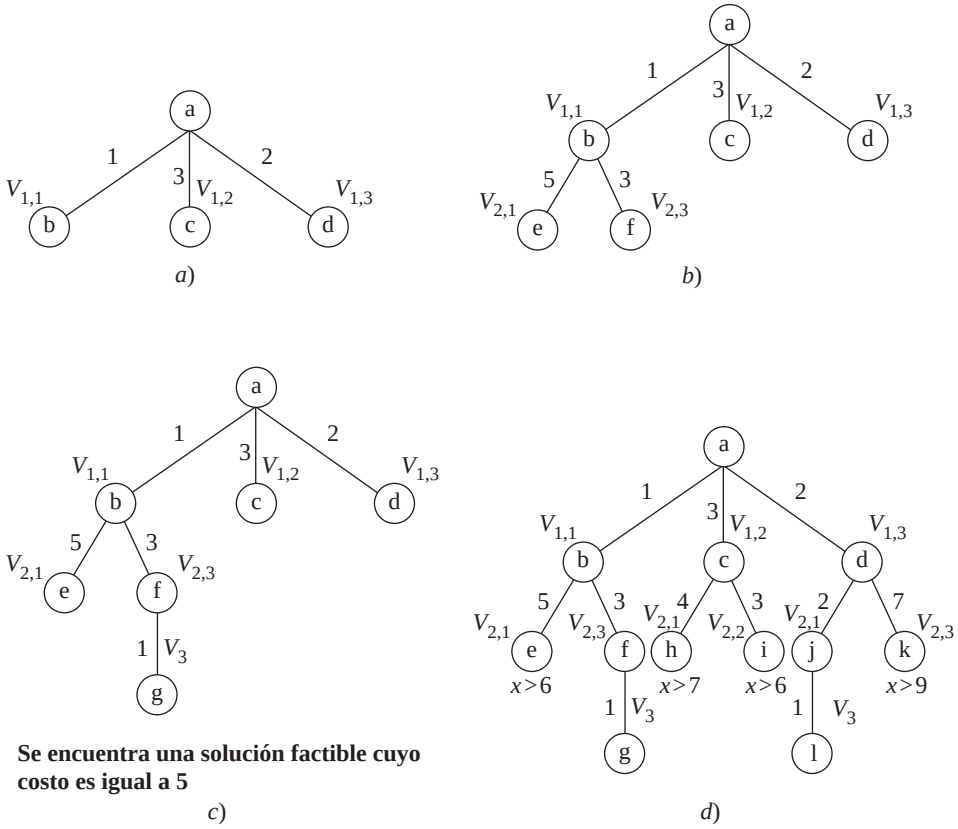
En la figura 5-18 se observan las seis soluciones factibles. ¿Cómo puede la estrategia de ramificar-y-acotar ser de utilidad para encontrar la ruta más corta sin necesidad de realizar una búsqueda exhaustiva? Considere la figura 5-19, que ilustra el proceso de usar algún tipo de método de ascenso de colina. En la figura 5-19*a*), desde la raíz del árbol de búsqueda se han desarrollado tres nodos. De entre estos tres nodos, se ha escogido uno para expandirlo. Puede haber muchas formas para seleccionar el siguiente nodo a expandir.

En nuestro caso, se considerará que se ha usado el método de ascenso de colina. Es decir, de entre los nodos que se han desarrollado más recientemente, el que se escoge para expandirlo a continuación siempre es el nodo asociado con el menor costo.

Usando este principio, se expandirá el nodo *b*. Sus dos descendientes se muestran en la figura 5-19*b*). Debido a que el nodo *f* corresponde a $V_{2,3}$ y su costo asociado es el más pequeño, de entre los nodos *e* y *f*, el que se expandirá es el nodo *f*. Debido a que el nodo *g* es un nodo meta, debe encontrarse una solución factible cuyo costo sea igual a 5, como se muestra en la figura 5-19*c*).

El costo de esta solución factible, que es igual a 5, sirve como una cota superior de la solución óptima. Cualquier solución con costo superior a 5 no puede ser una solución óptima. En consecuencia, esta cota puede usarse para eliminar prematuramente

FIGURA 5-19 Ilustración de la estrategia de ramificar-y-acotar.



muchas ramas. Por ejemplo, el nodo *e* jamás conducirá a una solución óptima porque cualquier solución con el nodo *e* tendrá un costo superior a 6.

Como se muestra en la figura 5-19, es posible evitar una búsqueda exhaustiva de todo el espacio solución. Por supuesto, es necesario indicar que hay otra solución que también es óptima.

El ejemplo anterior ilustra el principio básico de la estrategia de ramificar-y-acotar. Esta estrategia consta de dos mecanismos importantes: un mecanismo para generar ramificaciones y un mecanismo para generar una cota de modo que sea posible eliminar muchas ramificaciones. Aunque esta estrategia suele ser muy eficiente, en los peores casos, aún puede generarse un árbol muy grande. Por lo tanto, es necesario darse cuenta de que la estrategia de ramificar-y-acotar es eficiente en casos promedio.

5-6 Un problema de asignación de personal resuelto con la estrategia de ramificar-y-acotar

A continuación se mostrará cómo un problema de asignación de personal, que es NP completo, puede resolverse de manera eficiente aplicando la estrategia de ramificar-y-acotar. Considere un conjunto de personas $P = \{P_1, P_2, \ldots, P_n\}$ ordenado linealmente, donde $P_1 < P_2 < \cdots < P_n$. Puede imaginarse que el ordenamiento de las personas es determinado por algún criterio como estatura, edad, antigüedad, etc. También se considera un conjunto de trabajos $J = \{J_1, J_2, \ldots, J_n\}$, de los cuales se supone que están parcialmente ordenados. A cada persona puede asignarse un trabajo. Sean las personas P_i y P_j, a quienes se asignan los trabajos $f(P_i)$ y $f(P_j)$, respectivamente. Se requiere que si $f(P_i) \leq f(P_j)$, entonces $P_i \leq P_j$. La función f puede interpretarse como una asignación factible que mapea personas en sus trabajos idóneos. También requiere que si $i \neq j$, entonces $f(P_i) \neq f(P_j)$.

Considere el siguiente ejemplo. $P = \{P_1, P_2, P_3\}$, $J = \{J_1, J_2, J_3\}$ y el ordenamiento parcial de J es $J_1 \leq J_3$ y $J_2 \leq J_3$. En este caso, $P_1 \rightarrow J_1$, $P_2 \rightarrow J_2$ y $P_3 \rightarrow J_3$ son combinaciones factibles, mientras $P_1 \rightarrow J_1$, $P_2 \rightarrow J_3$ y $P_3 \rightarrow J_2$ no lo son.

Además, se supone que hay un costo C_{ij} en el que incurre una persona P_i a la que se asigna el trabajo J_j. Sea X_{ij} igual a 1 si a P_i se asigna J_j e igual a 0 en caso contrario. Entonces, el costo total correspondiente a una asignación factible f es

$$\sum_{i,j} C_{ij}X_{ij}$$

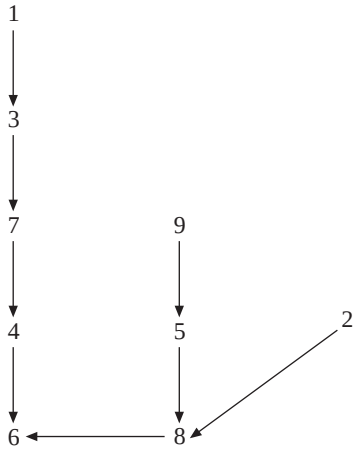
El problema de asignación de personal se define con precisión como sigue: se tiene un conjunto de personas $P = \{P_1, P_2, \ldots, P_n\}$ ordenado linealmente, donde $P_1 < P_2 < \cdots < P_n$ y un conjunto de trabajos $J = \{J_1, J_2, \ldots, J_n\}$ ordenado parcialmente. El costo C_{ij} es igual al costo de asignar P_i a J_j. Cada persona se asigna a un trabajo a ningún par de personas se asigna el mismo trabajo. El problema consiste en encontrar una asignación factible óptima que minimice la siguiente cantidad

$$\sum_{i,j} C_{ij}X_{ij}.$$

Así, este problema es de optimización, del cual puede mostrarse que es NP-difícil. Aquí no se abordará la dificultad NP.

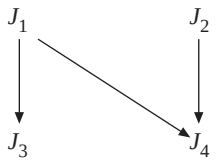
Para resolver el problema se usará el concepto de "ordenamiento topológico". Para un ordenamiento parcial dado de un conjunto S, una secuencia lineal $S_1, S_2, \ldots, S_n$ está ordenada topológicamente con respecto a S si $S_i \leq S_j$ en el ordenamiento parcial implica que S_i está ubicado antes que S_j en la secuencia. Por ejemplo, para el ordenamiento parcial que se muestra en la figura 5-20, una secuencia ordenada topológicamente correspondiente es 1, 3, 7, 4, 9, 2, 5, 8, 6.

FIGURA 5-20 Ordenamiento parcial.



Sea $P_1 \rightarrow J_{k_1}$, $P_2 \rightarrow J_{k_2}$, …, $P_n \rightarrow J_{k_n}$ una asignación factible. Según la definición de nuestro problema, los trabajos están ordenados parcialmente y las personas están ordenadas linealmente. En consecuencia, J_{k_1}, J_{k_2}, …, J_{k_n} debe ser una secuencia ordenada topológicamente con respecto al ordenamiento parcial de los trabajos. Esta idea se ilustrará con un ejemplo. Considere $J = \{J_1, J_2, J_3, J_4\}$ y $P = \{P_1, P_2, P_3, P_4\}$. El ordenamiento parcial de J se muestra en la figura 5-21.

FIGURA 5-21 Un ordenamiento parcial de los trabajos.



Las siguientes secuencias están ordenadas topológicamente:

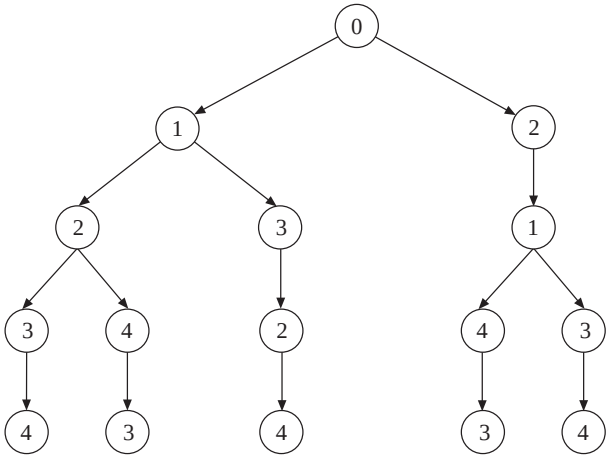
J_1, J_2, J_3, J_4
J_1, J_2, J_4, J_3
J_1, J_3, J_2, J_4
J_2, J_1, J_3, J_4
J_2, J_1, J_4, J_3.

Cada secuencia representa una asignación factible. Por ejemplo, la primera secuencia corresponde a la asignación factible

$P_1 \rightarrow J_1, P_2 \rightarrow J_2, P_3 \rightarrow J_3, P_4 \rightarrow J_4$.

Para encontrar todas las secuencias ordenadas topológicamente pueden usarse fácilmente tres técnicas de búsqueda. Por ejemplo, para el ordenamiento parcial que se muestra en la figura 5-21, en la figura 5-22 se observa un árbol que muestra todas las secuencias ordenadas topológicamente.

FIGURA 5-22 Representación de árbol de todas las secuencias ordenadas topológicamente correspondientes a la figura 5-21.



El árbol en la figura 5-22 se genera usando tres pasos básicos.

1. Tomar un elemento que no esté precedido por ningún otro elemento en el ordenamiento parcial.
2. Seleccionar este elemento como un elemento en una secuencia ordenada topológicamente.
3. Eliminar del conjunto de ordenamiento parcial este elemento que acaba de seleccionarse. El conjunto resultante sigue estando parcialmente ordenado.

Por ejemplo, para el ordenamiento parcial que se muestra en la figura 5-21, en el principio, J_1 y J_2 son los elementos sin predecesores. Así, están en el mismo nivel del árbol. Considere el nodo correspondiente a 1. Si se elimina 1, el conjunto parcialmente ordenado ahora contiene a 2, 3 y 4. Sólo 2 y 3 carecen de predecesores en este nuevo conjunto. En consecuencia, se generan 2 y 3.

Una vez explicado cómo el espacio solución de nuestro problema se describe por medio de un árbol, procederemos a la demostración de cómo aplicar la estrategia de ramificar-y-acotar para encontrar una solución óptima.

Dada una matriz de costos, de inmediato es posible calcular una cota inferior de nuestras soluciones. Esta cota inferior se obtiene al reducir la matriz de costos de forma que no se afecten las soluciones y de que en cada renglón y en cada columna haya por lo menos un cero y que todos los elementos restantes de la matriz de costos sean no negativos.

Observe que si de cualquier renglón o cualquier columna de la matriz de costos se resta una constante, una solución óptima permanece sin cambio. Considere la tabla 5-1, donde se muestra una matriz de costos. Para este conjunto de datos, es posible restar 12, 26, 3 y 10 de los renglones 1, 2, 3 y 4, respectivamente. Después de lo anterior es posible restar 3 de la columna 2. La matriz resultante es una matriz de costos reducida donde todo renglón y toda columna contienen por lo menos un cero y todos los elementos restantes de la matriz son no negativos, como se muestra en la tabla 5-2. El costo total restado es $12 + 26 + 3 + 10 + 3 = 54$. Ésta es la cota inferior de nuestras soluciones.

TABLA 5-1 Una matriz de costos para un problema de asignación de personal.

Personas \ Trabajos	1	2	3	4
1	29	19	17	12
2	32	30	26	28
3	3	21	7	9
4	18	13	10	15

TABLA 5-2 Una matriz de costos reducida.

Personas \ Trabajos	1	2	3	4	
1	17	4	5	0	
2	6	1	0	2	Total = 54
3	0	15	4	6	
4	8	0	0	5	

En la figura 5-23 se muestra un árbol de enumeración asociado con esta matriz de costos reducida. Si se usa la mínima cota inferior, las subsoluciones que no pueden conducir a soluciones óptimas se podan en una etapa mucho más temprana, lo cual se muestra en la figura 5-24.

FIGURA 5-23 Árbol de enumeración asociado con la matriz de costos reducida en la tabla 5-2.

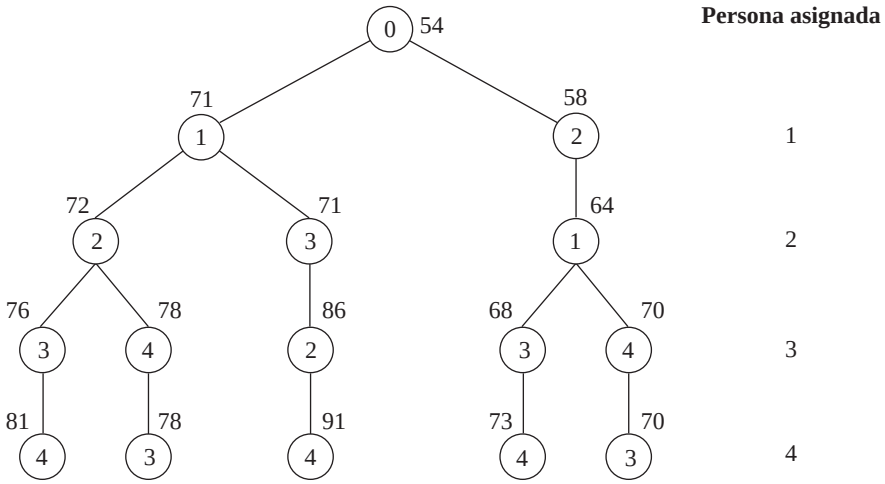
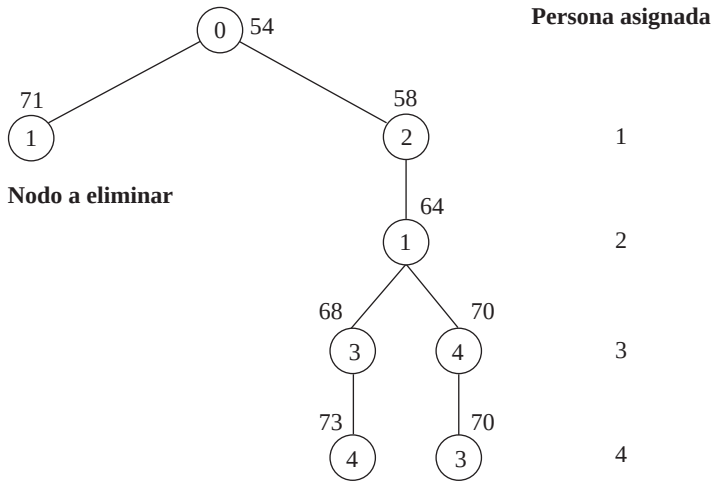


FIGURA 5-24 Acotamiento de las subsoluciones.



En la figura 5-24 puede verse que después de que se encuentra una solución con costo 70, de inmediato es posible acotar todas las soluciones empezando con la asignación de P_1 a J_1 porque su costo es 71, que es mayor que 70.

¿Por qué se restaron costos de la matriz de costos? Suponga que no se hiciera esto. Luego, considere el nodo correspondiente a asignar $P_1 \rightarrow J_1$. El costo asociado con este nodo es sólo 29. Imagine que se ha encontrado una solución factible con costo 70; a saber, al asignar $P_1 \rightarrow J_2$, $P_2 \rightarrow J_1$, $P_3 \rightarrow J_4$ y $P_4 \rightarrow J_3$. Aunque se ha encontrado una cota superior, ésta no puede usarse para acotar al nodo correspondiente a $P_1 \rightarrow J_1$ porque su costo es sólo 29, menor que 70.

Observe nuevamente la figura 5-24. Ahora puede verse que el costo asociado con asignar P_1 a J_1 es 71, en vez de 29. Así, aparece una cota. ¿Por qué es posible tener un costo tan elevado? Esto se debe a que se han restado costos de la matriz de costos original de modo que cada renglón y cada columna contienen un cero. Así, después de restar, se tiene una mejor cota inferior para todas las soluciones; a saber, 54. En otras palabras, ninguna solución puede tener un costo menor que 54. Con esta información, se sabe que la cota inferior de asignar P_1 a J_1 es $54 + 17 = 71$, en vez de sólo 29. Por supuesto, una cota inferior más alta conduce a una eliminación más temprana.

5-7 EL PROBLEMA DE OPTIMIZACIÓN DEL AGENTE VIAJERO RESUELTO CON LA ESTRATEGIA DE RAMIFICAR-Y-ACOTAR

El problema de decisión del agente viajero es un problema NP-completo. Así, este problema es difícil de resolver en los peores casos. Sin embargo, como se verá en esta sección, el problema del agente viajero puede resolverse aplicando la estrategia de ramificar-y-acotar. Esto es, si se tiene suerte, es posible evitar una búsqueda exhaustiva a través del espacio solución.

El principio fundamental al aplicar la estrategia de ramificar-y-acotar para resolver el problema de optimización del agente viajero consta de dos partes.

1. Hay una forma de dividir el espacio solución.
2. Hay una forma de pronosticar una cota inferior para una clase de soluciones. También hay una forma de encontrar una cota superior de una solución óptima. Si la cota inferior de una solución excede a esta cota superior, entonces esta solución no puede ser óptima. Así, es necesario eliminar la ramificación asociada con esta solución.

El problema del agente viajero se define sobre una gráfica, o en puntos planos. Si el problema del agente viajero se define sobre un conjunto de puntos planos, es posible usar muchos trucos para que el algoritmo sea todavía más eficiente. En esta sección se

supondrá que el problema se define sobre una gráfica. Para simplificar el análisis, en nuestro ejemplo se supone que entre un vértice y él mismo no hay ningún arco y que entre cada par de vértices hay un arco asociado con un costo no negativo. El problema del agente viajero consiste en encontrar un recorrido, empezando desde cualquier vértice, que pase por todos los demás vértices y regrese al vértice inicial, con costo mínimo.

Considere la matriz de costos en la tabla 5-3.

TABLA 5-3 Matriz de costos para un problema del agente viajero.

i \ *j*	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	∞	3	93	13	33	9	57
2	4	∞	77	42	21	16	34
3	45	17	∞	36	16	28	25
4	39	90	80	∞	56	7	91
5	28	46	88	33	∞	25	57
6	3	88	18	46	92	∞	7
7	44	26	33	27	84	39	∞

La estrategia de ramificar-y-acotar divide la solución en dos grupos: un grupo que incluye un arco particular y otro grupo que excluye este arco. Cada división incurre en una cota inferior y el árbol de búsqueda se atravesará con la cota inferior "más baja".

Antes que todo se observa, como se analizó en el apartado previo, que si una constante se resta de cualquier renglón o cualquier columna de la matriz de costos, una solución óptima no cambia. Para el ejemplo de la tabla 5-3, si el costo mínimo se resta de cada renglón de la matriz de costos, la cantidad total que se resta es una cota inferior para la solución del problema del agente viajero. Así, es posible restar 3, 4, 16, 7, 25, 3 y 26 de los renglones 1 a 7, respectivamente. El costo total restado es $3 + 4 + 16 + 7 + 25 + 3 + 26 = 84$. De esta forma es posible obtener una matriz reducida, que se muestra en la tabla 5-4.

En la matriz que se muestra en la tabla 5-4, cada renglón contiene un cero. Sin embargo, algunas columnas, a saber, las columnas 3, 4 y 7, aún no contienen ningún cero. Entonces, además se resta 7, 1 y 4 de las columnas 3, 4 y 7, respectivamente. (El costo restado es $7 + 4 + 1 = 12$.) La matriz reducida resultante se muestra en la tabla 5-5.

TABLA 5-4 Una matriz de costos reducida.

i \ *j*	1	2	3	4	5	6	7
1	∞	0	90	10	30	6	54
2	0	∞	73	38	17	12	30
3	29	1	∞	20	0	12	9
4	32	83	73	∞	49	0	84
5	3	21	63	8	∞	0	32
6	0	85	15	43	89	∞	4
7	18	0	7	1	58	13	∞

TABLA 5-5 Otra matriz de costos reducida.

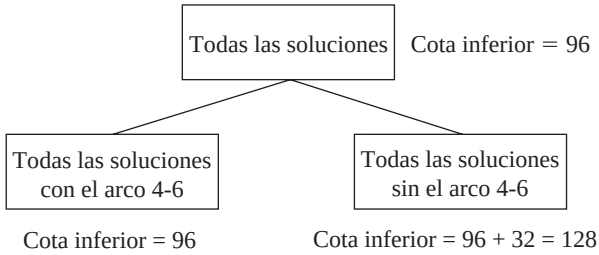
i \ *j*	1	2	3	4	5	6	7
1	∞	0	83	9	30	6	50
2	0	∞	66	37	17	12	26
3	29	1	∞	19	0	12	5
4	32	83	66	∞	49	0	80
5	3	21	56	7	∞	0	28
6	0	85	8	42	89	∞	0
7	18	0	0	0	58	13	∞

Debido a que se restó un costo total de $84 + 12 = 96$, se sabe que una cota inferior de este problema del agente viajero es 96.

A continuación se considerará el siguiente problema: suponga que se sabe que un recorrido incluye al arco 4-6, cuyo costo es cero. ¿Cuál es la cota inferior del costo de este recorrido? Es muy fácil encontrar la respuesta: la cota inferior sigue siendo 96.

Suponga que se sabe que el recorrido excluye al arco 4-6. ¿Cuál es la nueva cota inferior? Observe la tabla 5-5. Si un recorrido no incluye al arco 4-6, entonces debe incluir algún otro arco que salga de 4. El arco con el menor costo que sale de 4 es el arco 4-1, cuyo costo es 32. El arco con el menor costo que sale de 6 es 5-6, cuyo costo es 0. Así, la nueva cota inferior es $96 + (32 + 0) = 128$. En consecuencia, se tiene el árbol binario que se muestra en la figura 5-25.

FIGURA 5-25 Nivel más elevado de un árbol decisión.



¿Por qué se escogió el arco 4-6 para dividir la solución? Esto se debe al hecho de que el arco 4-6 provoca el mayor incremento de la cota inferior. Suponga que para esta división se usa el arco 3-5. En este caso, la cota inferior sólo puede incrementarse por $1 + 17 = 18$.

A continuación se considerará el subárbol izquierdo. En éste se incluye el arco 4-6. Así, es necesario eliminar el cuarto renglón y la sexta columna de la matriz de costos. Además, como se usa el arco 4-6, no es posible usar el arco 6-4. Es necesario igualar $c_{6\text{-}4}$ a infinito. La matriz resultante se muestra en la tabla 5-6.

TABLA 5-6 Una matriz de costos reducida si se incluye el arco 4-6.

i \ *j*	1	2	3	4	5	7
1	∞	0	83	9	30	50
2	0	∞	66	37	17	26
3	29	1	∞	19	0	5
5	3	21	56	7	∞	28
6	0	85	8	∞	89	0
7	18	0	0	0	58	∞

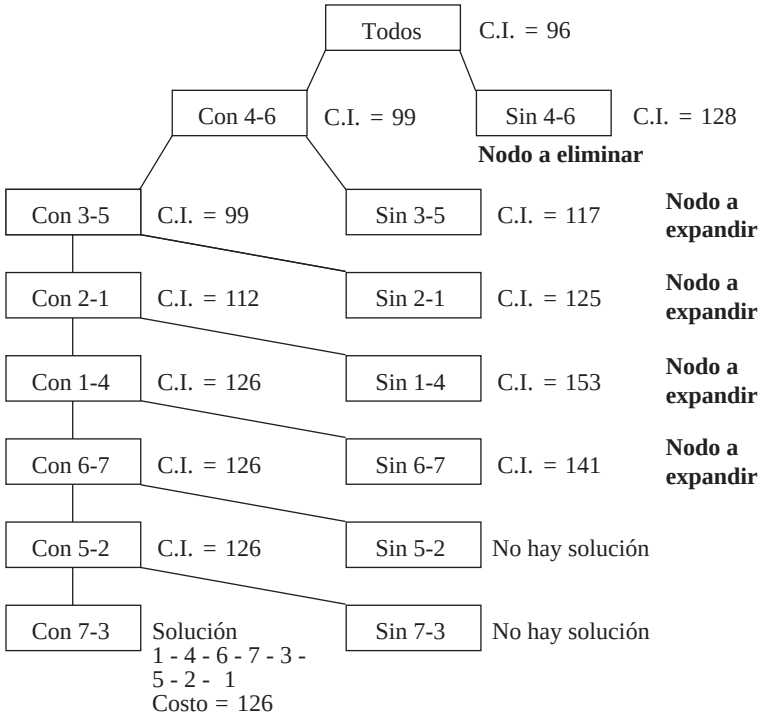
De nuevo, se observa que el renglón 5 aún no contiene ningún cero. Así, es posible restar 3 del renglón 5. La matriz de costos reducida para el subárbol izquierdo se muestra en la tabla 5-7. También es necesario sumar 3 a la cota inferior del subárbol izquierdo (soluciones con el arco 4-6).

En cuanto a la matriz de costos del subárbol derecho, soluciones sin el arco 4-6, basta igualar $c_{4\text{-}6}$ a ∞. El proceso de separación continuaría y produciría el árbol que se muestra en la figura 5-26. En este proceso, si se sigue la ruta de menor costo, se obtiene

TABLA 5-7 Una matriz de costos reducida para la de la tabla 5-6.

i \ *j*	1	2	3	4	5	7
1	∞	0	83	9	30	50
2	0	∞	66	37	17	26
3	29	1	∞	19	0	5
5	0	18	53	4	∞	25
6	0	85	8	∞	89	0
7	18	0	0	0	58	∞

FIGURA 5-26 Una solución de ramificar-y-acotar de un problema del agente viajero.



una solución factible con costo 126. Este costo 126 sirve como cota superior y permite eliminar muchas ramificaciones porque sus cotas inferiores exceden a esta cota.

Aquí es necesario mencionar otro punto. Éste puede explicarse al considerar la matriz de costos reducida de todas las soluciones con los arcos 4-6, 3-5 y 2-1 incluidos, como se muestra en la tabla 5-8.

TABLA 5-8 Una matriz de costos reducida.

i \ *j*	**2**	**3**	**4**	**7**
1	∞	74	0	41
5	14	∞	0	21
6	85	8	∞	0
7	0	0	0	∞

El arco 1-4 puede usarse para dividir y el subárbol resultante es como se muestra en la tabla 5-9.

TABLA 5-9 Una matriz de costos reducida.

i \ *j*	**2**	**3**	**7**
5	14	∞	21
6	85	8	0
7	0	0	∞

Observe que ya se ha decidido que los arcos 4-6 y 2-1 están incluidos en la solución. Ahora es necesario agregar el arco 1-4. Resulta evidente que es necesario impedir el uso del arco 6-2. Si se usa este arco, se forma un ciclo que está prohibido. Así, es necesario igualar $c_{6\text{-}2}$ a ∞. En consecuencia, se tendrá la matriz de costos para el subárbol izquierdo como se muestra en la tabla 5-10.

En general, si las rutas i_1-i_2-…-i_m y j_1-j_2-…-j_n ya se han incluido y debe sumarse una ruta de i_m a j_1, entonces es necesario evitar la ruta de j_n a i_1.

TABLA 5-10 Una matriz de costos reducida.

i \ *j*	2	3	7
5	14	∞	21
6	∞	8	0
7	0	0	∞

5-8 EL PROBLEMA 0/1 DE LA MOCHILA (KNAPSACK) RESUELTO CON LA ESTRATEGIA DE RAMIFICAR-Y-ACOTAR

Este problema se define como sigue: se tienen enteros positivos P_1, P_2, …, P_n, W_1, W_2, …, W_n y M. El problema consiste en encontrar X_1, X_2, …, X_n, $X_i = 0$ o 1, $i = 1$, 2, …, n, tales que

$$\sum_{i=1}^{n} P_i X_i$$

se maximice sujeta a

$$\sum_{i=1}^{n} W_i X_i \leq M$$

Éste es un problema NP-difícil. No obstante, como se verá, para resolverlo sigue siendo posible aplicar la estrategia de ramificar-y-acotar. Por supuesto, en los peores casos, incluso esta estrategia requiere un número exponencial de pasos para resolver dicho problema.

El problema original 0/1 de la mochila es un problema de maximización que no puede resolverse con la estrategia de ramificar-y-acotar. Para resolver el problema 0/1 de la mochila, es necesario modificarlo en un problema de minimización como sigue: dados enteros positivos P_1, P_2, …, P_n, W_1, W_2, …, W_n y M, encontrar X_1, X_2, …, X_n, $X_i = 0$ o 1, $i = 1$, …, n tales que

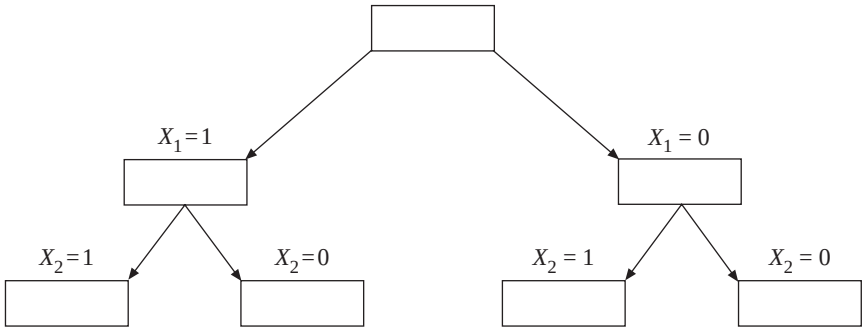
$$-\sum_{i=1}^{n} P_i X_i$$

se maximice sujeta a

$$\sum_{i=1}^{n} W_i X_i \leq M.$$

Cualquier estrategia de ramificar-y-acotar requiere un mecanismo para efectuar la ramificación. En el problema 0/1 de la mochila, este mecanismo se ilustra en la figura 5-27. La primera ramificación separa todas las soluciones en dos grupos: las soluciones con $X_1 = 0$ y las soluciones con $X_1 = 1$. Para cada grupo, X_2 se usa para separar las soluciones. Como puede verse, después de que se han enumerado las n X_i's, se encuentra una solución factible.

FIGURA 5-27 Mecanismo de ramificación en la estrategia de ramificar-y-acotar para resolver el problema 0/1 de la mochila.



Antes de explicar la estrategia de ramificar-y-acotar para resolver el problema 0/1 de la mochila, se recordará cómo se utilizó esta estrategia para resolver el problema del agente viajero. Cuando la estrategia de ramificar-y-acotar se usa para resolver el problema del agente viajero, se aplica el siguiente principio fundamental: las soluciones se separan en dos grupos. Para cada grupo se encuentra una cota inferior. Al mismo tiempo, se intenta buscar una solución factible. Siempre que se encuentra una solución factible, también se encuentra una cota superior. La estrategia de ramificar-y-acotar termina el desarrollo de un nodo si y sólo si se cumple una de las siguientes condi-ciones:

1. El nodo en sí representa una solución no factible. Así, ya no tiene sentido continuar ningún desarrollo.
2. La cota inferior de este nodo es superior o igual a la mínima cota superior que se ha encontrado hasta el momento.

A continuación se mostrará un mejoramiento adicional de la estrategia de ramificar-y-acotar para resolver el problema 0/1 de la mochila. Las soluciones siguen separándose en dos grupos. *Para cada grupo se encuentra no sólo una cota inferior, sino que también se encuentra una cota superior al determinar una solución factible.* A medida que se desarrolla un nodo, se espera encontrar una solución con el menor costo. Esto significa que se desea encontrar una nueva cota mínima superior a medida que se desarrolla el nodo. Si se sabe que la cota superior de un nodo no puede hacerse más baja porque ya es igual a su cota inferior, entonces este nodo ya no debe expandirse más. En términos generales, la ramificación termina si y sólo si se cumple una de las siguientes condiciones:

1. *El nodo en sí representa una solución no factible.*
2. *La cota inferior de este nodo es mayor o igual a la mínima cota superior encontrada hasta el momento.*
3. *Las cotas inferior y superior de este nodo son iguales.*

La pregunta es: ¿cómo encontrar una cota superior y una cota inferior de un nodo? Observe que una cota inferior puede considerarse como el valor de la mejor solución que puede alcanzarse. Un nodo del árbol corresponde a una solución parcialmente construida. En consecuencia, una cota inferior de este nodo corresponde a la máxima ganancia posible asociada con esta solución parcialmente construida. En cuanto a la cota superior de un nodo, se entiende el costo de una solución factible correspondiente a esta solución parcialmente construida. El método se ilustrará con un ejemplo.

Considere los datos siguientes:

i	1	2	3	4	5	6
P_i	6	10	4	5	6	4
W_i	10	19	8	10	12	8

$$M = 34$$

Es necesario observar que $P_i/W_i \geq P_{i+1}/W_{i+1}$ para $i = 1, 2, \ldots, 5$. Este ordenamiento es necesario, como se verá después.

¿Cómo puede encontrarse una solución factible?

Una solución factible puede encontrarse fácilmente si se empieza desde la i más pequeña disponible, explorando hacia las i más grandes hasta exceder a M. Por ejemplo, puede hacerse $X_1 = X_2 = 1$. Así, $\sum_{i=1}^{n} W_i X_i = 10 + 19 = 29 < M = 34$. Esto significa

que $X_1 = X_2 = 1$ es una solución factible. (Observe que ya no es posible hacer $X_3 = 1$ porque $\sum_{i=1}^{3} W_i X_i > 34$.)

¿Cómo puede encontrarse una cota inferior?

Para encontrar una cota inferior, recuerde que una cota inferior corresponde al mejor valor que puede alcanzar la función de costos. Observe que el problema 0/1 de la mochila es un problema de optimización restringido porque X_i está restringido a 0 y 1.

Si se relaja esta restricción, se obtiene un mejor resultado que puede usarse como la cota inferior buscada. Puede dejarse que X_i esté entre 0 y 1. De ese modo, el problema 0/1 de la mochila se convierte en el problema de la mochila que se define a continuación: dados enteros positivos $P_1, P_2, \ldots, P_n, W_1, W_2, \ldots, W_n$ y M, encontrar $X_1, X_2, \ldots, X_n$, $0 \leq X_i \leq 1$, $i = 1, \ldots, n$, tales que

$$-\sum_{i=1}^{n} P_i X_i$$

se maximice sujeta a

$$\sum_{i=1}^{n} W_i X_i \leq M$$

Sean $-\sum_{i=1}^{n} P_i X_i$ una solución óptima para el problema 0/1 de la mochila y $-\sum_{i=1}^{n} P_i X_i'$ una solución óptima para el problema de la mochila. Sean $Y = -\sum_{i=1}^{n} P_i X_i$ y $Y' = -\sum_{i=1}^{n} P_i X_i'$. Es fácil demostrar que $Y' \leq Y$. Es decir, una solución para el problema de la mochila puede usarse como una cota inferior de la solución óptima del problema 0/1 de la mochila.

Además, hay una cuestión bastante interesante relacionada con el problema de optimización de la mochila. Es decir, el método codicioso puede usarse para encontrar una solución óptima del problema de la mochila (consulte el ejercicio 3-11). Considere el conjunto de datos antes proporcionado. Suponga que ya se ha hecho $X_1 = X_2 = 1$. No es posible hacer X_3 igual a 1 porque $W_1 + W_2 + W_3$ excede a M. No obstante, si se deja que X_3 sea igual a algún número entre 0 y 1, se obtiene una solución óptima del problema de la mochila, que es una cota inferior de la solución óptima del problema 0/1 de la mochila. El valor idóneo de X_3 puede encontrarse como sigue: debido a que

$M = 34$ y $W_1 + W_2 = 10 + 19 = 29$, lo mejor que puede ser X_3 es tal que $W_1 + W_2 + W_3X_3 = 10 + 19 + 8X_3 = 34$. Así, $X_3 = (34 - 29)/8 = 5/8$. Con este valor, se encuentra que una cota inferior es $-(6 + 10 + 5/8 \times 4) = -18.5$. Se usa el límite superior. En consecuencia, la cota inferior es –18.

Se considerará el caso en que $X_1 = 1$, $X_2 = 0$ y $X_3 = 0$. Debido a que es $W_1 + W_4 + W_5 = 32 < 34$ y $W_1 + W_4 + W_5 + W_6 = 40 > 34$, la cota inferior puede encontrarse al resolver la siguiente ecuación:

$$W_1 + W_4 + W_5 + W_6X_6 = 32 + 8X_6 = 34.$$

Se tiene: $W_6X_6 = 34 - 32 = 2$, y
$X_6 = \frac{2}{8} = \frac{1}{4}$. Esto corresponde a una cota inferior de

$$-\left(P_1 + P_4 + P_5 + \frac{1}{4}P_6\right) = -\left(6 + 5 + 6 + \frac{1}{4} \times 4\right) = -18.$$

Observe que nuestro método para encontrar una cota inferior es correcto porque $P_i/W_i \geq P_{i+1}/W_{i+1}$ y nuestro método codicioso encuentra correctamente una solución óptima para el problema de la mochila (no para el problema 0/1 de la mochila) para esta condición.

¿Cómo puede encontrarse una cota superior?

Considere el caso siguiente:

$$X_1 = 1, X_2 = 0, X_3 = 0, X_4 = 0.$$

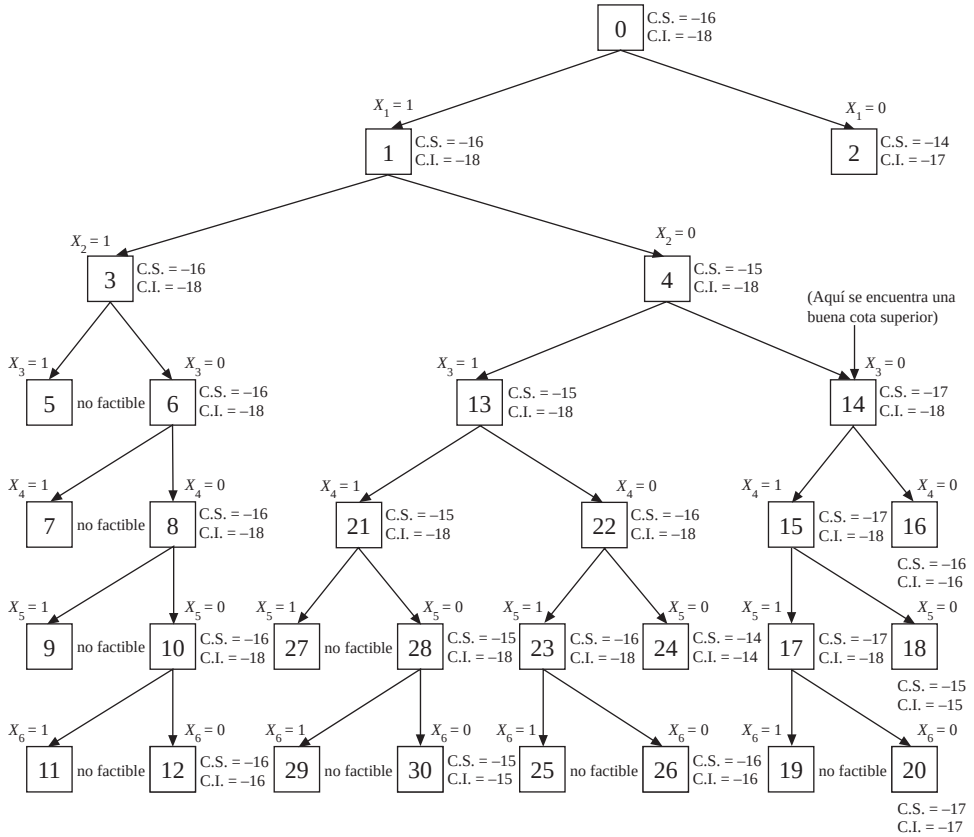
En este caso, una cota superior corresponde a

$$X_1 = 1, X_2 = 0, X_3 = 0, X_4 = 0, X_5 = 1, X_6 = 1.$$

Esta cota superior es $-(P_1 + P_5 + P_6) = -(6 + 6 + 4) = -16$. Lo que significa que por cuanto toca a este nodo, si se desarrolla aún más, se obtiene una solución factible con costo –16. Razón por la que –16 se denomina la cota superior de este nodo.

Todo el problema puede resolverse como se indica con el árbol que se muestra en la figura 5-28. El número en cada nodo indica la secuencia en que se desarrolla. Se usa la regla de la mejor búsqueda. Es decir, se desarrolla el nodo con la mejor cota inferior. Si dos nodos tienen la misma cota inferior, se desarrolla el que tiene la mínima cota superior.

FIGURA 5-28 Problema 0/1 de la mochila resuelto con la estrategia de ramificar-y-acotar.



En el árbol que se muestra en la figura 5-28,

1. el nodo 2 se terminó porque su cota inferior es igual a la cota superior del nodo 14.
2. todos los demás nodos se eliminaron porque la cota inferior local es igual a la cota superior local.

5-9 Un problema de calendarización del trabajo (job scheduling) resuelto con el método de ramificar-y-acotar

Aunque resulta fácil explicar los principios básicos de la estrategia de ramificar-y-acotar, de ninguna manera resulta fácil utilizarla de manera efectiva. Aún es necesario

inventar reglas inteligentes de ramificar-y-acotar para utilizar esta estrategia. En esta sección se mostrará la importancia de contar con reglas de acotamiento ingeniosas.

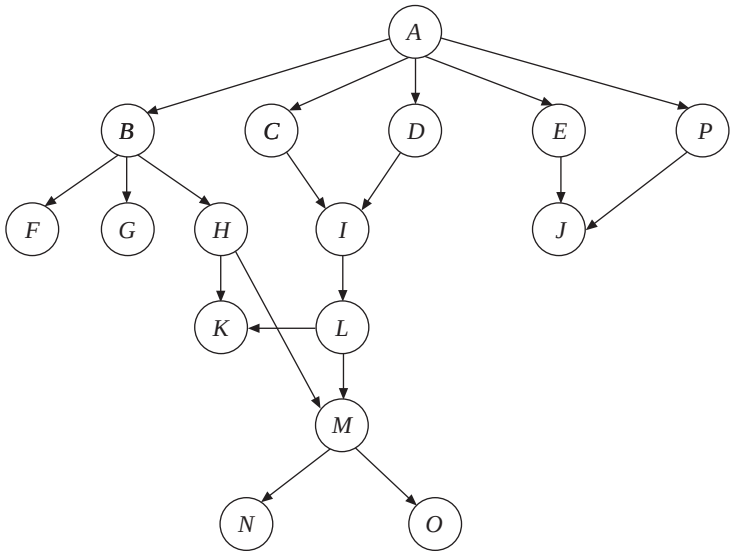
Nuestro problema es un problema de calendarización del trabajo con base en las siguientes hipótesis:

1. Todos los procesadores son idénticos y cualquier trabajo puede realizarse en cualquier procesador.
2. Hay un ordenamiento parcial de trabajos. Un trabajo no puede ejecutarse si uno de sus trabajos precedentes, en caso de existir, aún no ha sido efectuado.
3. Siempre que un procesador esté ocioso y haya un trabajo a efectuar, este procesador debe empezar a realizar el trabajo.
4. Cada trabajo requiere el mismo tiempo de ejecución.
5. Se proporciona un perfil temporal que especifica el número de procesadores que es posible usar simultáneamente en cada periodo.

El objeto de la calendarización es minimizar el tiempo de terminación máximo, que es el periodo en que se termina el último trabajo.

En la figura 5-29 se proporciona un ordenamiento parcial. Como puede verse, el trabajo *I* debe esperar los trabajos *C* y *D* y el trabajo *H* debe esperar al trabajo *B*. Al principio, sólo el trabajo *A* puede ejecutarse de inmediato.

FIGURA 5-29 Ordenamiento parcial de un problema de calendarización del trabajo.



Se considerará el perfil temporal que se muestra en la tabla 5-11.

TABLA 5-11 Un perfil temporal.

T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
3	2	2	2	4	5	3	2	3

Este perfil temporal indica que en el tiempo $t = 1$, sólo pueden usarse tres procesadores y que en $t = 5$, cuatro procesadores pueden estar activos.

Para el ordenamiento parcial en la figura 5-29 y el perfil temporal anterior, hay dos soluciones posibles:

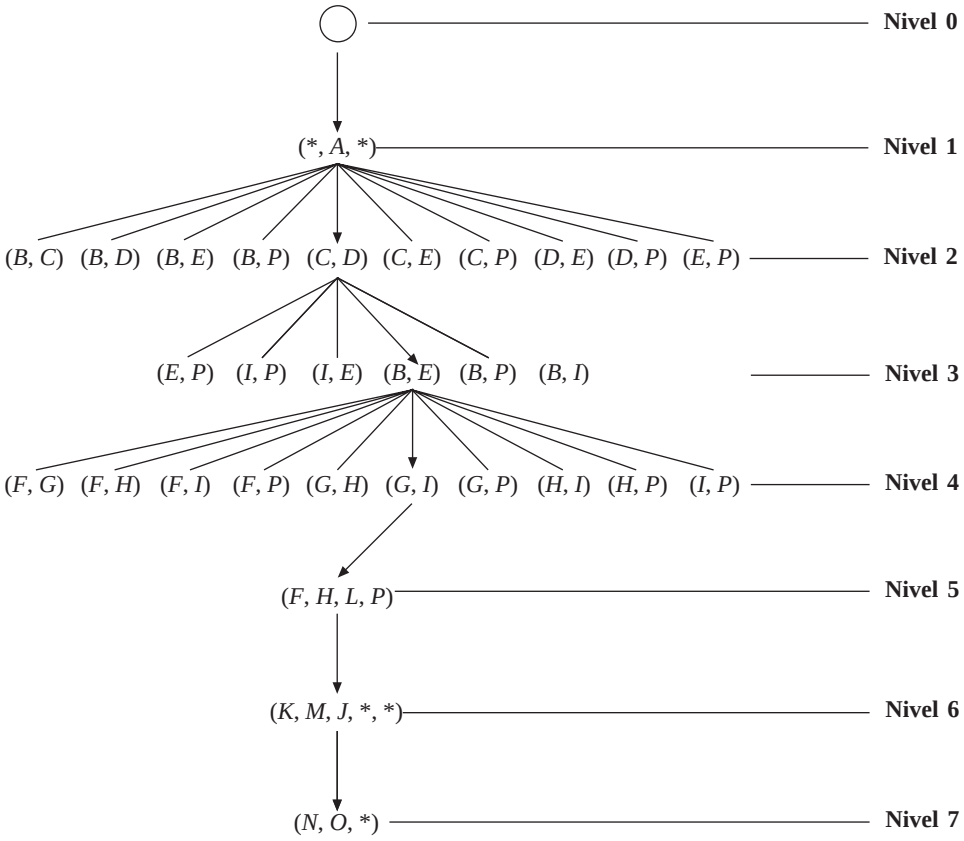
	T_1	T_2	T_3	T_4	T_5	T_6			
Solución 1:	*A*	*B*	*C*	*H*	*M*	*J*			
	*	*D*	*I*	*L*	*E*	*K*	Tiempo = 6		
	*				*F*	*N*			
					P	*O*			
						G			

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	
Solución 2:	*A*	*B*	*D*	*F*	*H*	*L*	*M*	*N*	Tiempo = 8
	*	*C*	*E*	*G*	*I*	*J*	*K*	*O*	
	*				*P*	*	*		
					*	*			
						*			

Resulta evidente que la solución 1 es mejor que la solución 2. Nuestra calendarización del trabajo se define como sigue: se tiene una gráfica con ordenamiento parcial y un perfil temporal; encontrar una solución con los pasos temporales mínimos para terminar todos los trabajos. Este problema se denomina problema de calendarización del trabajo con el mismo tiempo de ejecución con restricción de precedencia y perfil temporal. Se ha demostrado que es un problema NP-difícil.

Primero se demostrará que este problema de calendarización del trabajo puede resolverse con tres técnicas de búsqueda. Suponga que el ordenamiento parcial del trabajo es el que se muestra en la figura 5-29 y que el perfil temporal es el de la tabla 5-11. Entonces, en la figura 5-30 se muestra un árbol de una solución parcial.

FIGURA 5-30 Parte de un árbol solución.



* Indica un procesador ocioso.

La parte superior del árbol es un nodo vacío. Por el perfil temporal, se sabe que el número máximo de trabajos que es posible ejecutar es tres. No obstante, como el trabajo *A* es el único trabajo que es posible ejecutar en el principio, el nivel 1 del árbol solución consta de sólo un nodo. Después de que se ejecuta el trabajo *A*, por el perfil temporal, se sabe que ahora es posible ejecutar dos trabajos. El nivel 2 del árbol solución en la figura 5-30 muestra todas las combinaciones posibles. Debido a limitaciones de espacio, sólo se muestran los descendientes de los nodos como un subconjunto de nodos del árbol solución.

En el párrafo anterior se mostró cómo el problema de calendarización del trabajo con perfil temporal puede resolverse mediante la estrategia del árbol de búsqueda. En las siguientes secciones se mostrará que a fin de usar la estrategia de ramificar-y-acotar, se requiere inventar buenas reglas de ramificación y acotamiento, de modo que se evite la búsqueda en todo el árbol solución.

Las cuatro reglas siguientes pueden usarse en el método de ramificar-y-acotar. No se demostrará la validez de estas reglas. Sólo se describirán de manera informal.

Regla 1: Efecto de los sucesores comunes

Esta regla puede explicarse informalmente al considerar de nuevo la figura 5-30. En este caso, la raíz del árbol solución es el nodo que consta de un solo trabajo; a saber, *A*. Este nodo tiene muchos descendientes inmediatos, de los cuales dos son (*C*, *E*) y (*D*, *P*). Como se muestra en la figura 5-29, los trabajos *C* y *D* comparten el mismo descendiente inmediato; a saber, el trabajo *I*. De manera semejante, los trabajos *E* y *P* comparten el mismo descendiente; a saber, el trabajo *J*. En este caso, la regla 1 estipula que sólo un nodo, (*C*, *E*) o (*D*, *P*), requiere ser desarrollado en el árbol solución, ya que la longitud de cualquier solución óptima encabezada con el nodo (*C*, *E*) es igual a la de la solución óptima encabezada con el nodo (*D*, *P*).

¿Por qué es posible concluir lo anterior? Considere cualquier solución factible que salga de (*D*, *P*). En alguna parte de la solución factible están los trabajos *D* y *P*. Debido a que *C* y *D* comparten el mismo descendiente; a saber, el trabajo *I*, es posible intercambiar *C* y *D* sin modificar la factibilidad de la solución. Por un razonamiento semejante, también es posible intercambiar *P* y *E*. Así, para cualquier solución factible encabezada con (*C*, *E*), es posible transformarla en otra solución factible encabezada con (*D*, *P*), sin modificar la longitud de la solución. Así, la regla 1 es válida.

Regla 2: Primera estrategia del nodo interno

El nodo interno de la gráfica de precedencia de trabajos debe procesarse antes que el nodo hoja.

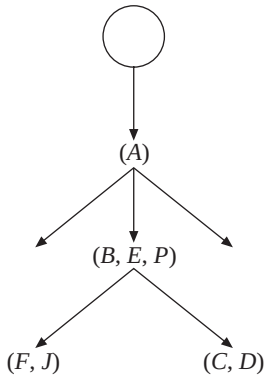
Esta regla puede explicarse informalmente al considerar la gráfica de precedencia de trabajos en la figura 5-29 y el perfil temporal en la tabla 5-12.

TABLA 5-12 Un perfil temporal.

T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
1	3	2	3	2	2	3	2	3

En la figura 5-31 se muestra una parte del árbol solución para este caso. Después de expandir el nodo (*B*, *E*, *P*) del árbol solución, para este nodo se tienen muchos descendientes inmediatos, dos de los cuales son (*C*, *D*) y (*F*, *J*), como se ilustra en la figura 5-31.

FIGURA 5-31 Árbol de solución parcial.



Debido a que los trabajos *C* y *D* son nodos internos en la gráfica de precedencia de trabajos donde *F* y *J* son nodos hoja, la regla 2 sugiere que es necesario atravesar el nodo (*C*, *D*) y eliminar el nodo (*F*, *J*) por una razón semejante a la que se usó al explicar la regla 1.

La regla 2 sugiere que el conjunto candidato debe dividirse en dos subconjuntos, uno para los nodos internos activos y otro para los nodos hoja activos de la gráfica de precedencia de trabajos. El primero tiene prioridad para su procesamiento. Debido a que este conjunto es de menor tamaño que el otro, reduce el número de opciones posibles. El conjunto de nodos hoja activos se escoge sólo cuando el tamaño del conjunto de nodos internos activos es menor que el número de procesadores activos. Debido a que los nodos hoja no tienen sucesores, da lo mismo la forma en que se seleccionen. En consecuencia, es posible escoger arbitrariamente cualquier grupo de ellos. Como se ilustra en la figura 5-30, después de atravesar el nodo (*B*, *E*) al nivel 3, se tienen cinco trabajos en el conjunto candidato actual (*F*, *G*, *H*, *I*, *P*) y hay dos sucesores activos. Así, se generan las 10 combinaciones posibles en total. Pero, si sólo se consideran nodos internos, entonces sólo se tienen tres nodos generados. Se trata de (*H*, *I*), (*H*, *P*) e (*I*, *P*). Los otros siete nodos no se generan nunca.

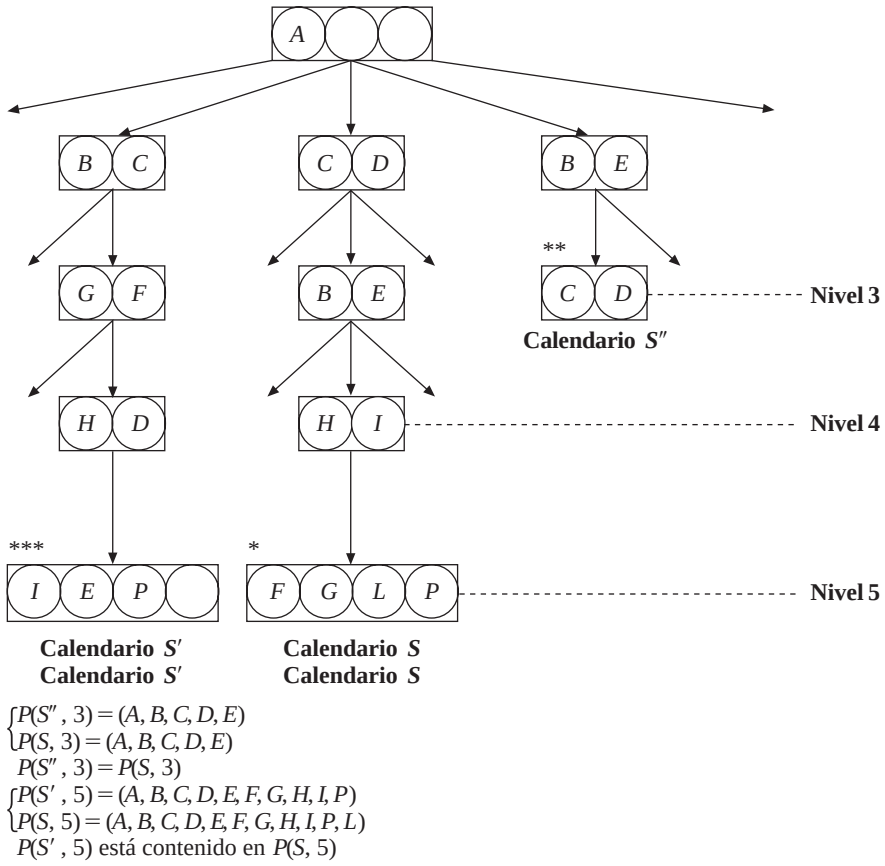
REGLA 3: Maximización del número de trabajos procesados

Sea $P(S, i)$ un conjunto de trabajos ya procesados en la calendarización *S* desarrollada parcial o completamente, desde el periodo 1 hasta el periodo *i*. La regla 3 puede plantearse como sigue:

Un calendario S no es una solución óptima si P(S, i) está contenido en P(S′, i) para algún otro calendario S′. La regla 3 es evidentemente correcta. En la figura 5-32

se muestra la forma en que es posible usar la regla 3 para eliminar algunos nodos del árbol solución. Para la figura 5-32, se afirma que los calendarios del nodo "***" y de "**" no pueden ser mejores que el calendario de "*". Esto se debe al hecho de que $P(S'', 3) = P(S, 3)$ y a que $P(S', 5)$ está contenido en $P(S, 5)$.

FIGURA 5-32 Efecto de trabajos procesados.



Regla 4: Estrategia de procesadores ociosos acumulados

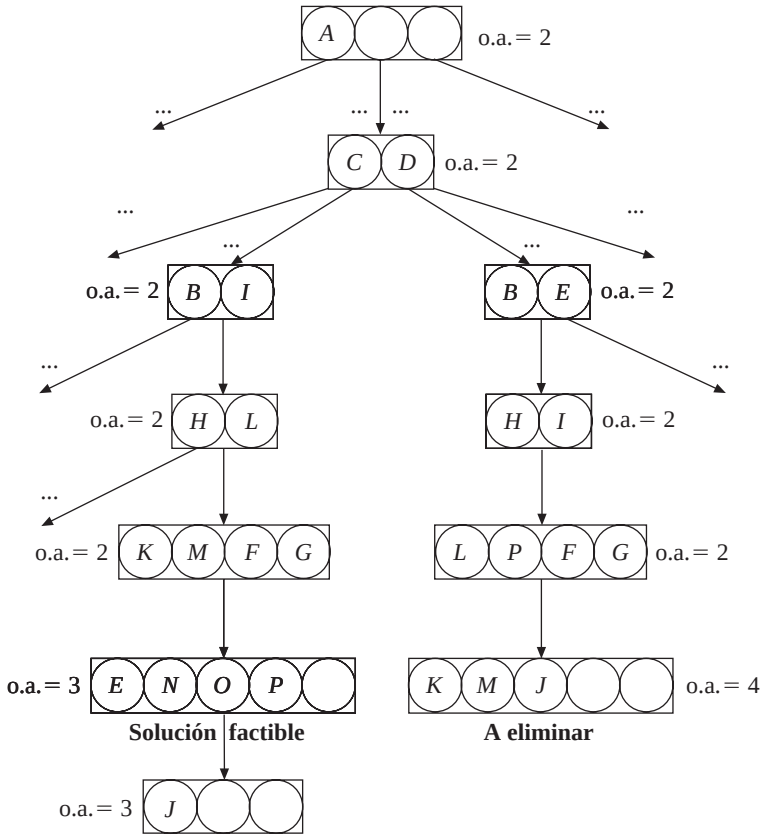
La regla 4 puede plantearse como sigue:

Cualquier calendario parcialmente desarrollado con el número de procesadores ociosos acumulados mayor que el de un calendario factible no puede ser mejor que esta solución factible, y en consecuencia es posible eliminarlo.

La regla 4 muestra que si ya se ha encontrado una solución factible, entonces es posible usar el número total de procesadores ociosos para eliminar otros nodos del árbol solución.

Considere la figura 5-33. El número de procesadores ociosos acumulados del nodo (*K*, *M*, *J*) del árbol solución es 4, que es mayor que el de la solución factible actual. Así, el nodo (*K*, *M*, *J*, *, *) se acota al aplicar la regla 4.

FIGURA 5-33 Efecto de los procesadores ociosos acumulados.



5-10 ALGORITMO *A**

Este algoritmo constituye una buena estrategia de árbol de búsqueda bastante favorecida por investigadores dedicados a la inteligencia artificial. Es muy lamentable que a menudo sea ignorado por los investigadores que se ocupan del campo de los algoritmos.

Primero se abordará la filosofía fundamental que subyace a este algoritmo. La mejor forma de hacer lo anterior es con la estrategia de ramificar-y-acotar.

Observe que con esta estrategia el primer esfuerzo es tener la certeza de que no es necesario investigar con mayor profundidad muchas soluciones porque no conducirán a soluciones óptimas. Así, los trucos principales en la estrategia de ramificar-y-acotar se encuentran en el acotamiento.

El método del algoritmo A^* recalca otro punto de vista. *Indica que en ciertas situaciones, una solución factible que se haya obtenido debe ser óptima.* Así, es posible detenerse. Por supuesto, es de esperar que esta terminación ocurra en la primera etapa.

El algoritmo A^* suele aplicarse a problemas de optimización. Utiliza la estrategia de primero el mejor (primero el menor costo). El elemento crítico del algoritmo A^* es la función de costos. Considere la figura 5-34. La tarea consiste en encontrar la ruta más corta de *S* a *T*. Suponga que para resolver el problema se aplica algún algoritmo de árbol de búsqueda. La primera etapa del árbol solución es la que se muestra en la figura 5-35. Considere el nodo *A* en la figura 5-35. Este nodo está asociado con la decisión de seleccionar V_1 (borde *a*). El costo en que se incurre al seleccionar V_1 es por lo menos 2 porque el costo del borde *a* es 2. Sea $g(n)$ la longitud de la ruta que va de la raíz del árbol decisión al nodo *n*. *Sea* $h^*(n)$ *la longitud de la ruta óptima que va del nodo n a un nodo meta*. *Entonces el costo del nodo es* $f^*(n) = g(n) + h^*(n)$. Para el árbol en la figura 5-35, $g(A) = 2$, $g(B) = 4$ y $g(C) = 3$. El problema es: ¿cuál es el valor de $h^*(n)$? Observe que hay muchas rutas que empiezan en V_1 y terminan en *T*.

A continuación se enumeran algunas de esas rutas:

$V_1 \rightarrow V_4 \rightarrow T$ (longitud de la ruta = 5)
$V_1 \rightarrow V_2 \rightarrow V_4 \rightarrow T$ (longitud de la ruta = 8)
$V_1 \rightarrow V_2 \rightarrow V_5 \rightarrow T$ (longitud de la ruta = 10)
$V_1 \rightarrow V_4 \rightarrow V_5 \rightarrow T.$ (longitud de la ruta = 8)

FIGURA 5-34 Gráfica para ilustrar el algoritmo A^*.

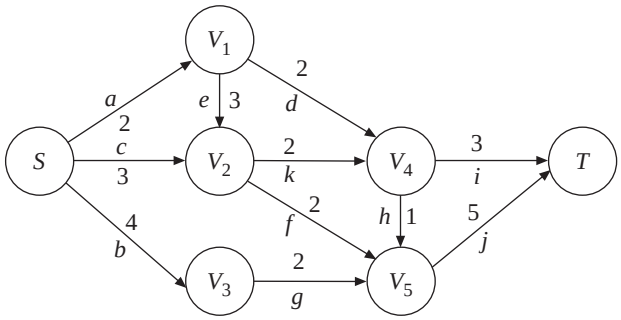
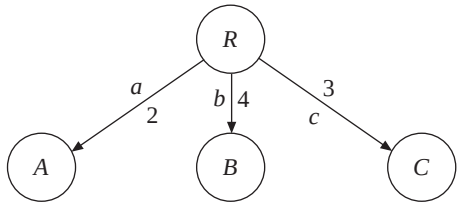


FIGURA 5-35 El primer nivel de un árbol solución.



Si se empieza desde V_1, la ruta óptima es $V_1 \rightarrow V_4 \rightarrow T$, cuya longitud es 5. Así, $h^*(A)$ es igual a 5, que es la ruta de longitud $V_1 \rightarrow V_4 \rightarrow T$. Debido a que, en términos generales, $h^*(n)$ es desconocida, también suele desconocerse $f^*(n)$. El algoritmo A^* supone que sigue siendo posible determinar $h^*(n)$. Considere nuevamente el nodo A de la figura 5-35. Este nodo corresponde a V_1 en la figura 5-34. Empezando desde V_1, hay dos rutas posibles, ya sea a V_2 o a V_4. La más corta es pasar por V_4, cuyo costo es 2. Esto significa que empezar desde el nodo A, la ruta más corta a partir de A tiene una longitud por lo menos igual a 2. En consecuencia, $h^*(A)$ es por lo menos igual a 2. *Ahora* $h(n)$ *denota una estimación de* $h^*(n)$.

Aunque hay muchas formas para estimar $h^*(n)$, *el algoritmo* A^* *estipula que siempre debe usarse* $h(n) \leq h^*(n)$. *Es decir, siempre debe usarse una estimación más bien conservadora de* $h^*(n)$. *Esto significa que en el algoritmo* A^* *siempre se usa* $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = f^*(n)$ *como función de costos*. Este concepto se aclarará después de la presentación de un ejemplo completo.

Finalmente, se presenta una propiedad muy importante del algoritmo A^*. Observe que este algoritmo usa la regla de primero el mejor, lo cual significa que de entre los nodos desarrollados, como siguiente nodo a expandir se escoge el nodo con menor costo. El algoritmo A^* posee la siguiente regla de terminación: *si un nodo seleccionado también es un nodo meta, entonces este nodo representa una solución óptima y el proceso puede terminar.*

A continuación se explicará por qué la regla anterior es correcta. Sea t el nodo meta seleccionado. Sea n un nodo que ya se ha desarrollado.

1. Debido a que el algoritmo A^* usa la regla del menor costo, se tiene $f(t) \leq f(n)$ para toda n.
2. Debido a que el algoritmo A^* usa una estimación conservadora de $h^*(n)$, se tiene $f(t) \leq f(n) \leq f^*(n)$ para toda n.
3. Sin embargo, una de las $f^*(n)$ tiene que ser una solución óptima. De hecho, sea $f^*(s)$ el valor de una solución óptima. Entonces se tiene

   $f(t) \leq f^*(s)$.

4. La afirmación 3 indica que en cualquier momento, para cualquier nodo a expandir, su función de costo no es mayor que el valor de una solución óptima. Debido a que t es un nodo meta, se tiene $h(t) = 0$ y

   $$f(t) = g(t) + h(t) = g(t)$$

   debido a que $h(t)$ es cero. En consecuencia,

   $$f(t) = g(t) \leq f^*(s).$$

5. No obstante, $f(t) = g(t)$ es el valor de una solución factible. En consecuencia, por definición $g(t)$ no puede ser menor que $f^*(s)$. Esto significa que $g(t) = f^*(s)$.

Ya se ha descrito el algoritmo A^*. A continuación se resumen las reglas fundamentales de este algoritmo como sigue:

1. El algoritmo A^* usa la regla de primero el mejor (o primero el menor costo). En otras palabras, de entre todos los nodos desarrollados, el siguiente a expandir es el de menor costo.
2. En el algoritmo A^*, la función de costo $f(n)$ se define como sigue:

   $$f(n) = g(n) + h(n)$$

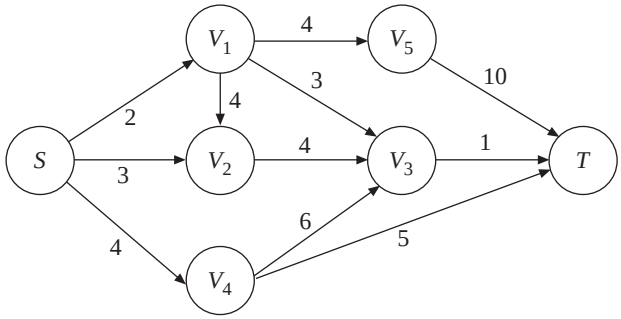
   donde $g(n)$ es la longitud de la ruta que va de la raíz del árbol a n y $h(n)$ es la estimación de $h^*(n)$, que es la longitud de la ruta óptima desde n hasta algún nodo meta.
3. $h(n) \leq h^*(n)$ para toda n.
4. El algoritmo A^* se detiene si y sólo si el nodo seleccionado también es un nodo meta. Entonces regresa una solución óptima.

El algoritmo A^* puede, por supuesto, usarse con otras reglas de árbol de búsqueda. Por ejemplo, considere la figura 5-36. Para esta gráfica, si el problema es encontrar una ruta más corta de S a T, entonces se observa que de S a V_3 hay dos rutas:

$$S \xrightarrow{2} V_1 \xrightarrow{3} V_3$$
$$S \xrightarrow{3} V_2 \xrightarrow{4} V_3$$

Debido a que la longitud de la segunda ruta es mayor que la de la primera, la segunda ruta debe ignorarse, ya que nunca producirá una solución óptima.

FIGURA 5-36 Gráfica que ilustra la regla de dominancia.



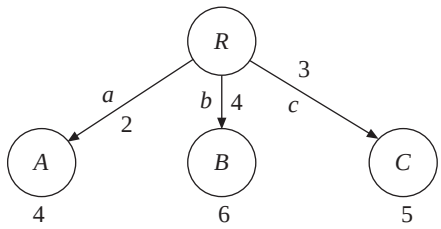
Esta regla se denomina de dominancia, que se usa en programación dinámica. Este tipo de programación se presentará después en este libro. Quizás el lector observe que si se usa la regla de dominancia, entonces también es posible ignorar la ruta

$S \rightarrow V_1 \rightarrow V_2$.

De manera semejante, todas las reglas de acotamiento que se usan en la estrategia de ramificar-y-acotar también pueden aplicarse en el algoritmo A^*.

A continuación se demostrará cómo funciona el algoritmo A^* para encontrar una ruta más corta en la gráfica de la figura 5-34. Los pasos 1 a 7 indican todo el proceso.

Paso 1. Expandir *R*

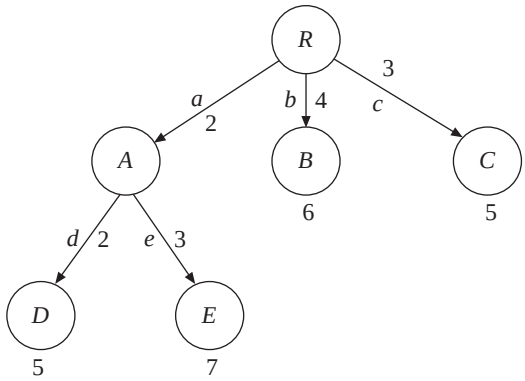


$g(A) = 2$ $\quad h(A) = \text{mín}\{2, 3\} = 2$ $\quad f(A) = 2 + 2 = 4$

$g(B) = 4$ $\quad h(B) = \text{mín}\{2\} = 2$ $\quad f(B) = 4 + 2 = 6$

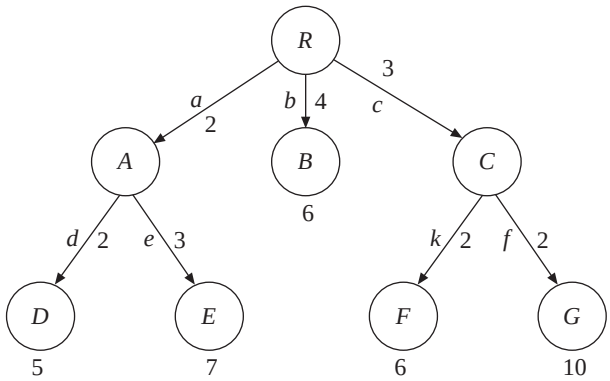
$g(C) = 3$ $\quad h(C) = \text{mín}\{2, 2\} = 2$ $\quad f(C) = 3 + 2 = 5$

Paso 2. Expandir *A*



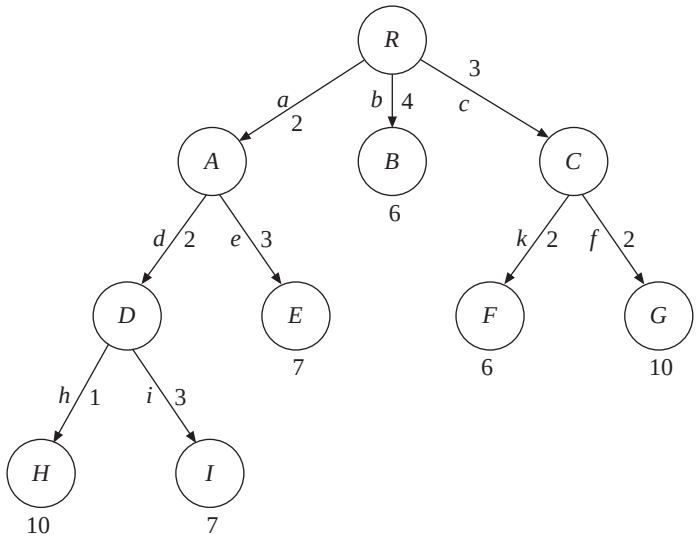
$g(D) = 2 + 2 = 4$ $h(D) = \text{mín}\{3, 1\} = 1$ $f(D) = 4 + 1 = 5$
$g(E) = 2 + 3 = 5$ $h(E) = \text{mín}\{2, 2\} = 2$ $f(E) = 5 + 2 = 7$

Paso 3. Expandir *C*



$g(F) = 3 + 2 = 5$ $h(F) = \text{mín}\{3, 1\} = 1$ $f(F) = 5 + 1 = 6$
$g(G) = 3 + 2 = 5$ $h(G) = \text{mín}\{5\} = 5$ $f(G) = 5 + 5 = 10$

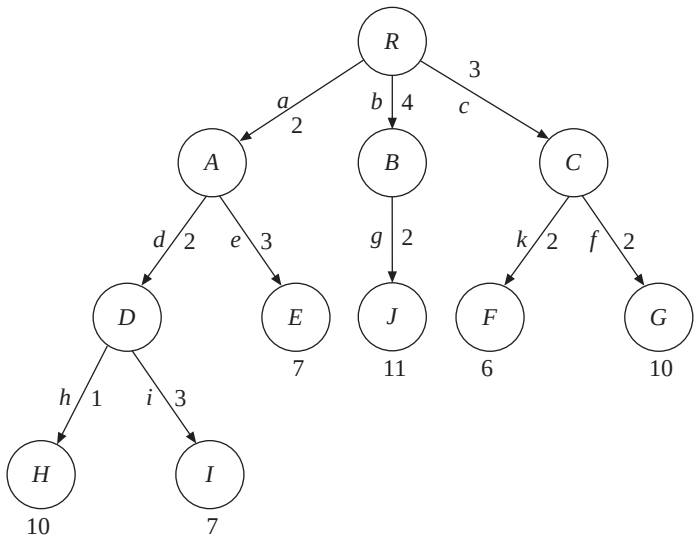
Paso 4. Expandir *D*



$g(H) = 2 + 2 + 1 = 5$ $h(H) = \text{mín}\{5\} = 5$ $f(H) = 5 + 5 = 10$

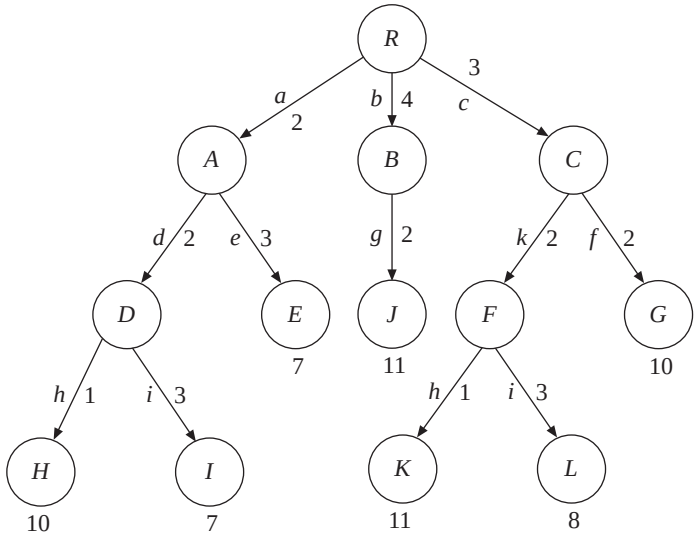
$g(I) = 2 + 2 + 3 = 7$ $h(I) = 0$ $f(I) = 7 + 0 = 7$

Paso 5. Expandir *B*



$g(J) = 4 + 2 = 6$ $h(J) = \text{mín}\{5\} = 5$ $f(J) = 6 + 5 = 11$

Paso 6. Expandir *F*



$g(K) = 3 + 2 + 1 = 6 \quad h(K) = \text{mín}\{5\} = 5 \quad f(K) = 6 + 5 = 11$

$g(L) = 3 + 2 + 3 = 8 \quad h(L) = 0 \quad f(L) = 8 + 0 = 8$

Paso 7. Expandir *I*

Debido a que *I* es un nodo meta, el procedimiento llega a un alto y como solución óptima se regresa

$$S \xrightarrow[2]{a} V_1 \xrightarrow[2]{d} V_4 \xrightarrow[3]{i} T.$$

Quizá sea importante plantear la siguiente pregunta: *¿el algoritmo A^* puede considerarse como un tipo especial de la estrategia de ramificar-y-acotar donde la función de costo esté diseñada de manera ingeniosa?* La respuesta es "Sí". Observe que cuando se termina el algoritmo A^*, esencialmente se afirma que todos los demás nodos desarrollados ahora están acotados simultáneamente por esta solución factible que se ha encontrado. Por ejemplo, considere el paso 6 del ejemplo anterior. El nodo *I* corresponde a una solución factible con costo 7. Esto significa que para este caso del problema se ha encontrado una cota superior, que es 7. Sin embargo, los costos de todos los demás nodos son mayores que, o iguales a 7. Estos costos también son cotas inferiores. Éste es el motivo por el que es posible terminar el proceso al acotar todos los otros nodos con esta cota superior.

Observe que la estrategia de ramificar-y-acotar es una estrategia general. No especifica la función de costo ni la regla para seleccionar un nodo a expandir. *El algoritmo A^* especifica la función de costo $f(n)$, que en realidad es una cota inferior de este nodo. El algoritmo A^* también especifica que la regla para seleccionar al nodo es la regla del menor costo. Siempre que se llega a una meta, se encuentra una cota superior.* Si esta meta se selecciona después para ser desarrollada, debido a que se aplica la regla del menor costo, entonces esta cota superior debe ser menor que o igual a todos los costos de los otros nodos. De nuevo, como el costo de cada nodo es una cota inferior de este nodo, esta cota superior es menor que o igual a todas las demás cotas inferiores. En consecuencia, esta cota superior debe ser el costo de una solución óptima.

5-11 UN PROBLEMA DE DIRECCIÓN DE CANALES RESUELTO CON UN ALGORITMO A^* ESPECIALIZADO

En esta sección se presentará un problema de dirección de canales que surge de un sistema muy grande de diseño asistido por computadora integrado. Se demostrará cómo este problema puede resolverse con mucha desenvoltura aplicando un algoritmo A^* especializado. Es necesario mencionar que en este algoritmo *tan pronto como se encuentra un nodo meta, es posible detener el proceso y obtener una solución óptima.* Observe que esto no se cumple en los algoritmos A^* ordinarios porque en este tipo de algoritmos no es posible terminar el algoritmo cuando se encuentra un nodo meta. El algoritmo sólo puede terminar cuando se selecciona un nodo para ser desarrollado.

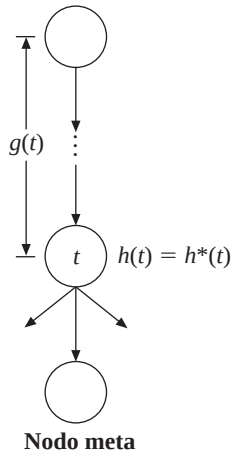
El algoritmo A^* especializado puede explicarse al considerar la figura 5-37. En este algoritmo, siempre que se selecciona un nodo t y produce un nodo meta como su sucesor inmediato, entonces $h(t) = h^*(t) = g(meta) - g(t)$. Con esta condición,

$$f(meta) = g(meta) + h(meta) = g(t) + h^*(t) + 0 = f^*(t)$$
$$g(meta) = f(meta) = f^*(t).$$

Observe que el nodo a expandir elegido es t. En consecuencia, $f^*(t) \leq f(n)$ si n es un nodo desarrollado porque se aplica la estrategia primero el mejor. Por definición, $f(n) \leq f^*(n)$. En consecuencia, $g(meta) = f^*(t) \leq f^*(n)$ y el primer nodo meta que se encuentra debe ser una solución óptima.

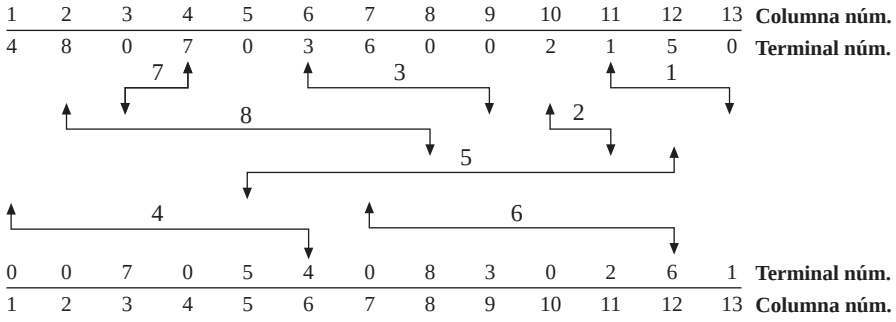
Como es de esperar, la función $h(n)$ debe diseñarse con sumo cuidado; en caso contrario, no se obtiene ese buen resultado. Será sorprendente para el lector que para el problema de dirección de canales que se define en esta sección, $h(n)$ puede diseñarse fácilmente de modo que cuando se aplica el algoritmo A^*, el primer nodo meta encontrado, o para plantearlo en otros términos, el primer nodo hoja, representa una solución óptima.

FIGURA 5-37 Situación especial en que se aplica el algoritmo A^*.



A continuación se describe el problema de dirección de canales. Considere la figura 5-38. En esta figura se tiene un canal y dos renglones de terminales, uno arriba y otro abajo. También hay un conjunto de redes identificadas de 1 a 8. Por ejemplo, la red 7 conecta la terminal 4 de arriba con la terminal 3 de abajo. De manera semejante, la red 8 conecta la terminal 2 de arriba con la terminal 8 de abajo. Una vez que se conectan estas terminales, no se permiten las conexiones ilegales descritas en la figura 5-39.

FIGURA 5-38 Especificación de canales.



Hay muchas formas de conectar terminales. Dos de ellas se muestran en las figuras 5-40 y 5-41, respectivamente. Para la disposición en la figura 5-40 hay siete pistas (tracks), mientras que para la de la figura 5-41 sólo hay cuatro pistas. De hecho, la disposición en la figura 5-41 tiene el número mínimo de pistas.

FIGURA 5-39 Conexiones ilegales.

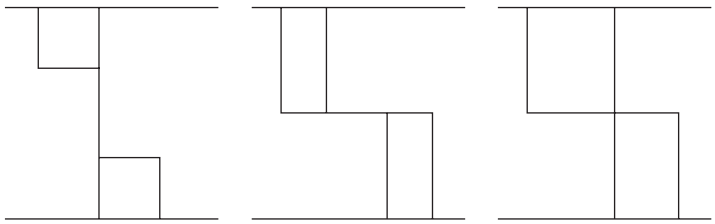


FIGURA 5-40 Una disposición factible.

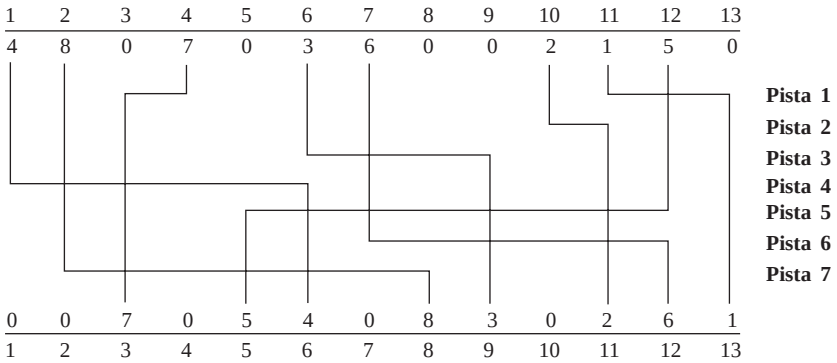
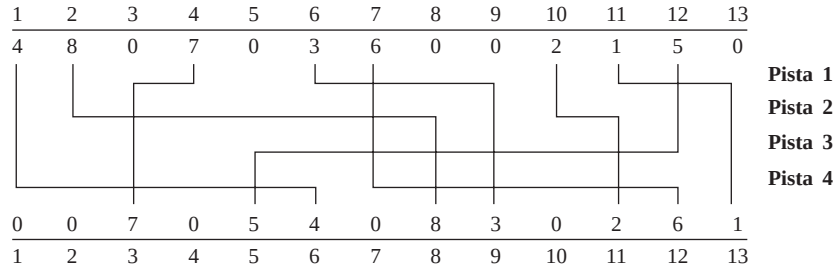


FIGURA 5-41 Una disposición óptima.

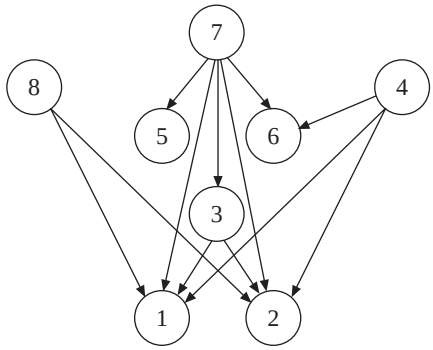


El problema de dirección de canales consiste en encontrar una disposición que minimice el número de pistas. Se ha demostrado que éste es un problema NP-difícil. A fin de diseñar un algoritmo A^* para resolver este problema, primero se observa que las redes deben respetar dos tipos de restricciones: restricciones horizontales y restricciones verticales.

Restricciones horizontales

Estas restricciones pueden explicarse al considerar la gráfica que se muestra en la figura 5-42. Al consultar la figura 5-38 se observa que la red 7, por ejemplo, debe conectarse a la izquierda de las redes 3, 5 y 6 si éstas se encuentran en la misma pista. De manera semejante, la red 8 debe conectarse a la izquierda de las redes 1 y 2. Estas relaciones se resumen en la siguiente gráfica.

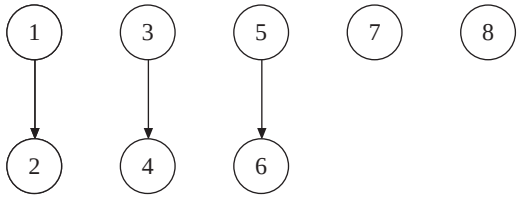
FIGURA 5-42 Gráfica de restricciones horizontales.



Restricciones verticales

Estas restricciones también pueden resumirse en la gráfica que se muestra en la figura 5-43.

FIGURA 5-43 Gráfica de restricciones verticales.



De nuevo, se considerará la figura 5-38. Se observa que la red 1 debe realizarse antes que la red 2, ya que comparten la misma terminal 11. De manera semejante, la red 3 debe conectarse antes que la red 4.

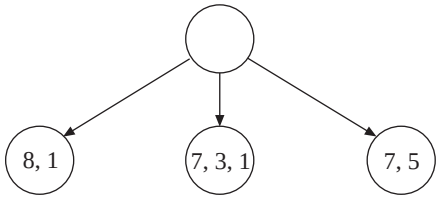
Debido a que en este libro sólo se está interesado en cómo aplicar el algoritmo A^*, no se abordarán los detalles para resolver el problema de dirección de canales. No se explicarán en detalle muchos pasos porque son irrelevantes para entender los algoritmos A^*.

Debido a la existencia de la gráfica de restricciones verticales, que esencialmente define un ordenamiento parcial, sólo aquellas redes sin predecesores en la gráfica de restricciones verticales pueden asignarse a alguna pista. Por ejemplo, inicialmente, sólo las redes 1, 3, 5, 7 y 8 pueden asignarse a una pista. Suponga que ya se han asignado 1 y 3. Entonces, luego es posible asignar 2 y 4.

Mientras la gráfica de restricciones verticales proporciona información sobre cuáles redes pueden asignarse, la gráfica de restricciones horizontales informa cuáles redes pueden asignarse a una pista. Como el interés principal es presentar el algoritmo A^*, no el método para resolver este problema de dirección de canales, se presentarán algunas operaciones sólo conceptualmente, sin presentar la teoría detrás de ellas. Se considerará la figura 5-43. Con base en esta figura se observa que es posible asignar las redes 1, 3, 5, 7 y 8. Al consultar la figura 5-42, se observa que de entre las redes 1, 3, 5, 7 y 8 hay tres círculos máximos (o cliques): {1, 8}, {1, 3, 7} y {5, 7}. (Un círculo de una gráfica es una subgráfica en la que cada par de vértices está unido. Un círculo máximo es un círculo cuyo tamaño no puede incrementarse.) Puede demostrarse que cada círculo máximo puede asignarse a una pista. El lector puede comprobar esto consultando la figura 5-38.

Al aplicar estas reglas, es posible usar un método de árbol de búsqueda para resolver el problema de dirección de canales, y el primer nivel del árbol se muestra en la figura 5-44.

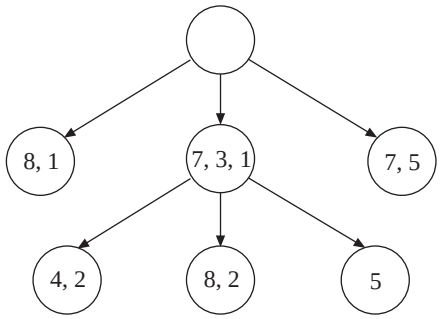
FIGURA 5-44 Primer nivel de un árbol para resolver un problema de dirección de canales.



Para expandir aún más este árbol, se considerará el nodo (7, 3, 1). Al consultar la gráfica de restricciones verticales que se muestra en la figura 5-43, después de que se han asignado las redes 1, 3 y 7, las redes que pueden asignarse son las redes 2, 4, 5 y 8. De manera semejante, si las redes 1 y 8 ya están asignadas, entonces es posible asignar las redes 2, 3, 5 y 7. Para el nodo (7, 3, 1), el siguiente conjunto de redes que puede asignarse es {2, 4, 5, 8}. Al consultar nuevamente la gráfica de restricciones horizontales que se

muestra en la figura 5-42, se observa que entre 2, 4, 5 y 8 hay tres círculos maximales; a saber, {4, 2}, {8, 2} y {5}. Entonces, si sólo se desarrolla el nodo {7, 3, 1}, las soluciones se vuelven las que se muestra en la figura 5-45.

FIGURA 5-45 El árbol de la figura 5-44 aún más desarrollado.

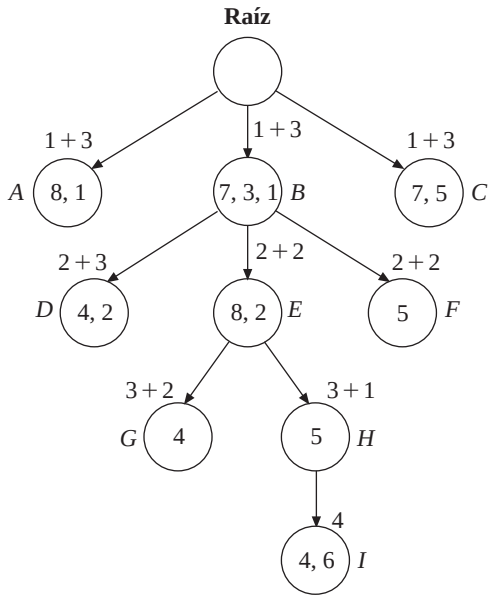


Una pregunta crítica es: ¿cuál es la función de costo de este método de árbol de búsqueda para resolver el problema de dirección de canales? Observe que para el algoritmo A^* se requieren dos funciones de costo: $g(n)$ *y* $h(n)$; $g(n)$ *puede definirse simplemente. Debido a que el nivel del árbol corresponde exactamente con el número de pistas, entonces* $g(n)$ *puede definirse de manera conveniente como el nivel del árbol. En cuanto a* $h(n)$*, puede calcularse usando el número mínimo de pistas que deben usarse después de que se ha asignado cierto número de pistas.*

Se considerará la figura 5-38. Se observa que para cada terminal es posible trazar una recta vertical. Si esta recta vertical corta k rectas horizontales, entonces el número mínimo de pistas que se requieren es k. Por ejemplo, en la terminal 3, el número mínimo de pistas es 3 mientras que en la terminal 7, el número mínimo de pistas es 4. Esto se denomina funciones densidad locales en la terminal i. La función densidad de todo el problema es la función densidad local maximal. Para el problema que se muestra en la figura 5-38, la función densidad es 4. La función densidad puede modificarse después de que algunas redes se han asignado a pistas. Por ejemplo, después de que las pistas 7, 3 y 1 se han asignado, la densidad se vuelve 3. Luego, esta función densidad se usa como $h(n)$. En la figura 5-46 se muestra cómo funciona el algoritmo A^* con nuestra función de costo.

Es fácil ver que para el algoritmo A^*, $h(t) = h^*(t) = 1 = g(meta) - g(t)$ porque para completar la disposición se requiere por lo menos una pista. En consecuencia, el nodo 1 debe representar una solución óptima.

FIGURA 5-46 Árbol de solución parcial para el problema de dirección de canales aplicando el algoritmo A^*.



Este ejemplo muestra que el algoritmo A^* constituye una muy buena estrategia para resolver algunos problemas combinatorios explosivos en el supuesto de que sea posible diseñar una buena función de costo.

5-12 EL PROBLEMA DE DECODIFICACIÓN DE UN BLOQUE LINEAL RESUELTO CON EL ALGORITMO *A**

Suponga que se usan códigos binarios para enviar ocho números enteros, de 0 a 7. Para cada número se requieren tres bits. Por ejemplo, 0 se envía como 000, 4 como 100 y 7 como 111. El problema es que si hay algún error, la señal recibida se decodifica erróneamente. Por ejemplo, para 100, si se envía y recibe como 000, se provocará un gran error.

Se supondrá que en vez de usar tres bits para codificar los números se usan, por ejemplo, seis dígitos. Considere los códigos en la tabla 5-13. El código derecho se denomina palabra código. Cada número en su forma de número binario se envía en la forma de su código correspondiente. Es decir, en lugar de enviar 100, ahora se envía 100110 y en vez de enviar 101, se envía 101101. Ahora es evidente la ventaja. Un vector recibido se decodifica en una palabra código cuya distancia Hamming es la más

TABLA 5-13 Palabras código.

000	000000
100	100110
010	010101
001	001011
110	110011
101	101101
011	011110
111	111000

pequeña de entre todas las palabras código. Suponga que la palabra código 000000 se envía como 000001. Fácilmente se ve que la distancia Hamming entre 000001 y 000000 es la más corta de todas las distancias Hamming entre 000001 y los ocho códigos. Así, el proceso de decodificación decodifica 000001 como 000000. En otras palabras, alargando el número de dígitos es posible tolerar más errores.

En este libro no se analizará cómo se generan los códigos. Este tema puede consultarse en libros sobre teoría de la codificación. Simplemente se supone que las palabras código ya existen. Y nuestra tarea es realizar la decodificación. En nuestro caso, el método de codificación se denomina código por bloque lineal.

En la práctica, 1 y 0 no se envían directamente. En esta sección se supondrá que 1 (0) se envía realmente como –1 (1). Es decir, 110110 se envía como $(-1, -1, 1, -1, -1, 1)$. La distancia entre un vector recibido $r = (r_1, r_2, \ldots, r_n)$ y una palabra código $(c_1, c_2, \ldots, c_n)$ es

$$d_E(r, c) = \sum_{i=1}^{n} (r_i - (-1)^{c_i})^2.$$

Suponga que se tiene $r = (-2, -2, -2, -1, -1, 0)$ y $c = 111000$. Entonces la distancia entre r y c es

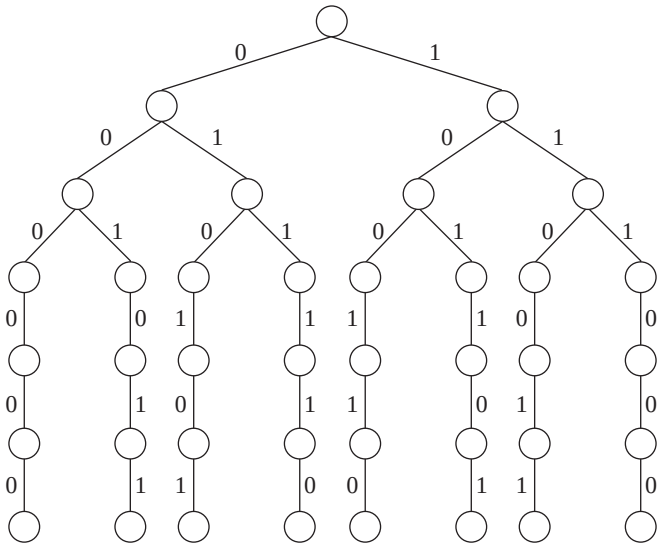
$$\begin{aligned} d_E(r, c) &= (-2 - (-1)^1)^2 + (-2 - (-1)^1)^2 + (-2 - (-1)^1)^2 \\ &\quad + (-1 - (-1)^0)^2 + (-1 - (-1)^0)^2 + (0 - (-1)^0)^2 \\ &= 12. \end{aligned}$$

Para decodificar un vector recibido, simplemente se calculan las distancias entre este vector y todas las palabras código y este vector recibido se decodifica como la palabra código particular con la distancia más pequeña. Observe que en la práctica, el

número de palabras código es más de 10^7, que es exageradamente grande. Cualquier búsqueda exhaustiva de todas las palabras código es imposible.

Lo que se hace es utilizar un árbol de código para representar todas las palabras código. Por ejemplo, para nuestro caso, todas las palabras código de la tabla 5-13 se representan con el árbol de la figura 5-47.

FIGURA 5-47 Árbol código.



La decodificación de un vector recibido, que es encontrar una palabra código lo más cercana a esta palabra recibida, ahora se vuelve un problema de árbol de búsqueda. Este problema puede describirse formalmente como sigue: encontrar la ruta de la raíz del árbol de código a un nodo meta de modo que el costo de la ruta sea mínimo de entre todas las rutas de la raíz a un nodo meta, donde el costo de una ruta es la suma de los costos de las ramas en la ruta. El costo de la rama de un nodo al nivel $t-1$ al nivel t es $(r_t - (-1)^{c_t})^2$. Se demostrará que el algoritmo A^* es un método efectivo para resolver el problema.

Sea –1 el nivel de la raíz del árbol de código. Sea n un nodo en el nivel t. La función $g(n)$ se define como

$$g(n) = \sum_{i=0}^{t} (r_i - (-1)^{c_i})^2$$

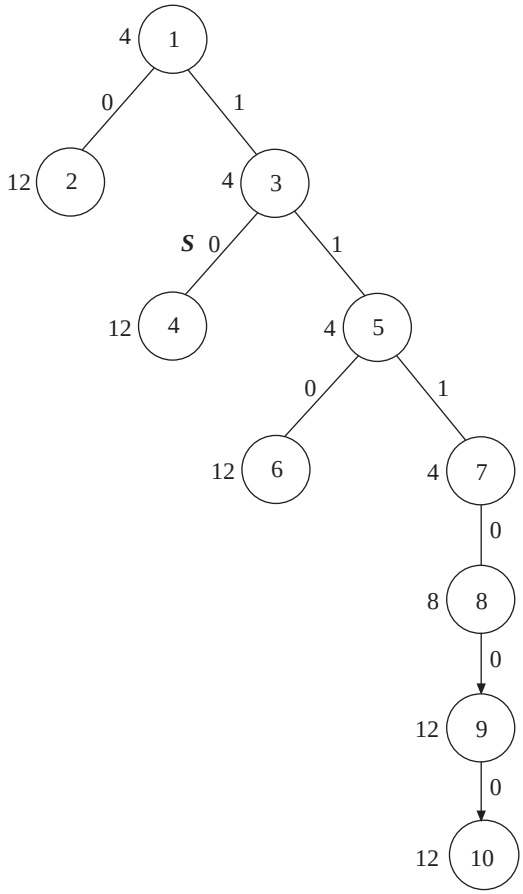
donde $(c_1, c_2, \ldots, c_t)$ son las etiquetas de las ramas asociadas con la ruta de la raíz al nodo n. La heurística $h(n)$ se define como

$$h(n) = \sum_{i=t+1}^{n-1} (|r_i| - 1)^2.$$

Puede verse fácilmente que $h(n) \leq h^*(n)$ para todo nodo n.

Considere el árbol de código en la figura 5-47. Sea $(-2, -2, -2, -1, -1, 0)$ el vector recibido. El proceso de decodificación usando el algoritmo A^* se ilustra en la figura 5-48.

FIGURA 5-48 Desarrollo de un árbol solución.



Al principio de la decodificación, se desarrolla el nodo raíz y se calculan los dos nuevos nodos generados, que son los nodos 2 y 3. El valor de $f(2)$ se calcula como sigue:

$$\begin{aligned} f(2) &= (-2 - (-1)^0)^2 + h(2) \\ &= 9 + \sum_{i=1}^{5} (|r_i| - 1)^2 \\ &= 9 + 1 + 1 + 0 + 0 + 1 \\ &= 12. \end{aligned}$$

El valor de $f(3)$ puede calcularse de la misma forma. El nodo 10 es un nodo meta. Cuando se selecciona para expandirlo, el proceso termina y se encuentra que la palabra código más cercana es 111000.

5-13 LOS RESULTADOS EXPERIMENTALES

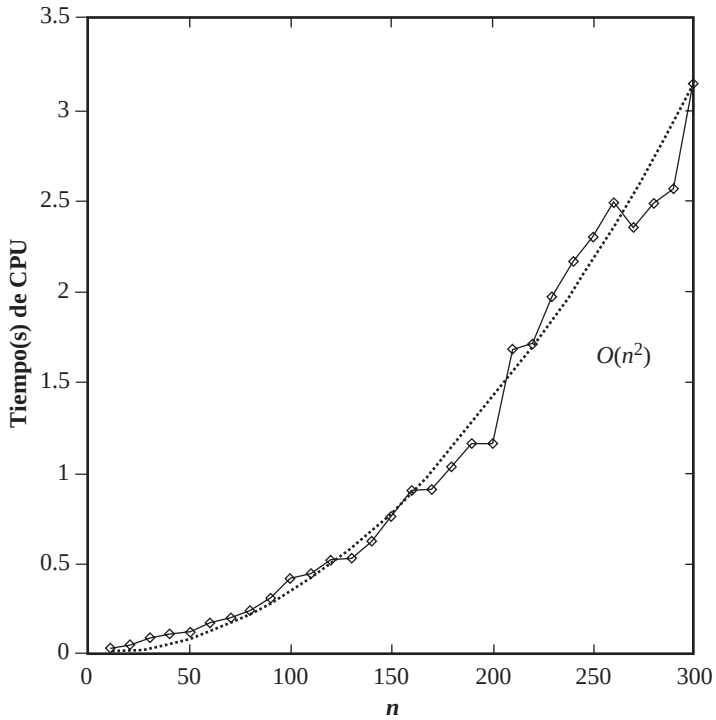
En nuestra opinión, muchos problemas NP-completos pueden resolverse de manera eficiente, en el sentido de casos promedio, mediante técnicas de árboles de búsqueda. Considere el problema 0/1 de la mochila, que es un problema NP-completo. El método de ramificar-y-acotar se implementó para resolver este problema en una PC IBM. El objetivo era probar si el desempeño en el caso promedio es exponencial o no. En la tabla 5-14 se resumen los resultados de la prueba. Para cada n, el número de elementos, se utilizó un generador de números aleatorios para obtener cinco conjuntos de datos. En cada conjunto de datos, los valores tanto de las P_i como de las W_i varían entre 1 y 50. El entero M se determina como sigue. Sea W la suma de todos los pesos. Entonces M se hace igual a $W/4$, $(W/4) + 20$, $(W/4) + 40$, …. Por supuesto, M no debe exceder a W. Luego, para cada n se encuentra el tiempo promedio necesario para resolver todos los conjuntos de instancias del problema.

Resulta más bien evidente que el desempeño en el caso promedio del algoritmo que resuelve el problema 0/1 de la mochila con base en la estrategia de ramificar-y-acotar está lejos de ser exponencial. De hecho, la figura 5-49 muestra que es casi $O(n^2)$. El lector debe sentirse alentado por estos resultados experimentales. No se debe temer a los problemas NP-completos. Hay algoritmos eficientes para resolver muchos de ellos, en casos promedio.

TABLA 5-14 Resultados de prueba al usar la estrategia de ramificar-y-acotar para resolver el problema 0/1 de la mochila.

n	Tiempo promedio
10	0.025
20	0.050
30	0.090
40	0.110
50	0.120
60	0.169
70	0.198
80	0.239
90	0.306
100	0.415
110	0.441
120	0.522
130	0.531
140	0.625
150	0.762
160	0.902
170	0.910
180	1.037
190	1.157
200	1.167
210	1.679
220	1.712
230	1.972
240	2.167
250	2.302
260	2.495
270	2.354
280	2.492
290	2.572
300	3.145

FIGURA 5-49 Resultados experimentales del problema 0/1 de la mochila resuelto con la estrategia de ramificar-y-acotar.



5-14 NOTAS Y REFERENCIAS

Las técnicas de búsqueda de primero en profundidad, de primero anchura y de primero el costo mínimo pueden encontrarse en Horowitz y Sahni (1976) y Knuth (1973). Para algoritmos A^*, consulte Nilsson (1980) y Perl (1984). En Lawler y Wood (1966) y Mitten (1970) pueden encontrarse excelentes repasos de las estrategias de ramificar-y-acotar.

El problema de asignación de personal fue analizado primero por Ramanan, Deogun y Liu (1984). Liang (1985) demostró que este problema se puede resolver con el método de ramificar-y-acotar. Para aplicar la estrategia de ramificar-y-acotar a fin de resolver el problema del agente viajero, consulte la obra de Lawler, Lenstra, Rinnooy Kan y Shmoys (1985) y la de Little, Murty, Sweeney y Karel (1963). La aplicación de la estrategia de ramificar-y-acotar para resolver el problema del agente viajero apareció en Yang, Wang y Lee (1989). Wang y Lee (1990) señalaron que el algoritmo A^* es una buena técnica para resolver el problema de dirección de canales. Aplicaciones de los algoritmos A^* a problemas de código en bloque lineal pueden encontrarse en Ekroot y

Dolinar (1996); Han, Hartmann y Chen (1993); y Han, Hartmann y Mehrotra (1998). En Hu y Tucker (1971) pueden consultarse trabajos relacionados.

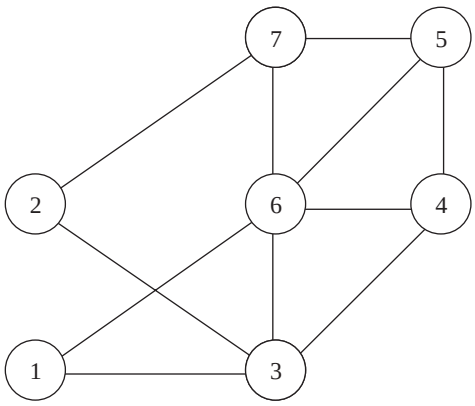
5-15 Bibliografía adicional

Las técnicas de árboles de búsqueda son muy naturales y fáciles de aplicar. Para el análisis de casos promedio con algoritmos de árboles de búsqueda, se recomiendan las siguientes obras: Brown y Purdom (1981); Huyn, Dechter y Pearl (1980); Karp y Pearl (1983); y Purdom y Brown (1985). Para algoritmos de ramificar-y-acotar, se recomienda Boffey y Green (1983); Hariri y Potts (1983); Ibaraki (1977); Sen y Sherali (1985); Smith (1984); y Wah y Yu (1985). Para algoritmos A^*, se recomienda Bagchi y Mahanti (1983); Dechter y Pearl (1985); Nau, Kumar y Kanal (1984); Pearl (1983), y Srimani (1989).

Para conocer resultados recientes, consulte a Ben-Asher, Farchi y Newman (1999); Devroye (2002); Devroye y Robson (1995); Gallant, Marier y Storer (1980); Giancarlo y Grossi (1997); Kirschenhofer, Prodinger y Szpankowski (1994); Kou, Markowsky y Berman (1981); Lai y Wood (1998); Lew y Mahmoud (1992); Louchard, Szpankowski y Tang (1999); Lovasz, Naor, Newman y Wigderson (1995), y Meleis (2001).

Ejercicios

5.1 Considere la gráfica siguiente. Aplique alguna técnica de árbol de búsqueda para encontrar un ciclo Hamiltoniano.



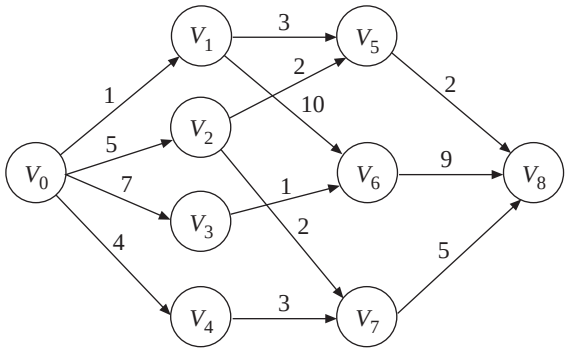
5.2 Resuelva el problema del rompecabezas de 8 piezas con el siguiente estado inicial.

2	3	
8	1	4
7	5	6

Observe que la meta final es

1	2	3
8		4
7	6	5

5.3 Aplique la estrategia de ramificar-y-acotar para encontrar la ruta más corta de v_0 a v_8.

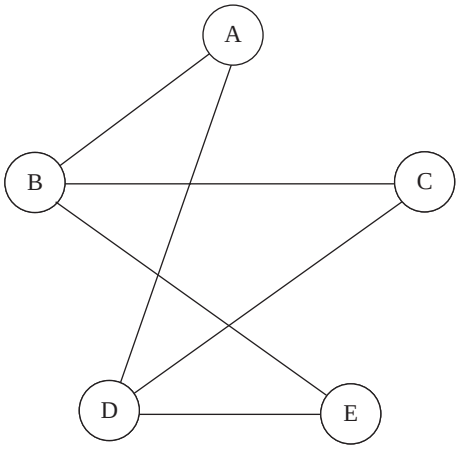


5.4 Aplique la estrategia de ramificar-y-acotar para resolver el problema del agente viajero caracterizado por la siguiente matriz de distancias.

i \ *j*	**1**	**2**	**3**	**4**	**5**
1	∞	5	61	34	12
2	57	∞	43	20	7
3	39	42	∞	8	21
4	6	50	42	∞	8
5	41	26	10	35	∞

5.5 Resuelva la siguiente suma del problema de subconjunto. $S = \{7, 1, 4, 6, 14, 25, 5, 8\}$ y $M = 18$. Encuentre una adición cuyos elementos sumen 18. Aplique la estrategia de ramificar-y-acotar.

5.6 Aplique alguna técnica de árbol de búsqueda para resolver el problema de cubierta de vértices de la siguiente gráfica.



5.7 Determine la satisfacibilidad de las siguientes fórmulas booleanas aplicando una técnica de árbol de búsqueda.

a) $-X_1 \vee X_2 \vee X_3$
$X_1 \vee X_3$
X_2

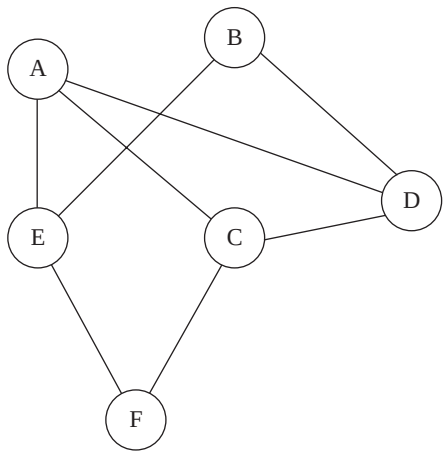
b) $-X_1 \vee X_2 \vee X_3$
$X_1 \vee X_2$
$-X_2 \vee X_3$
$-X_3$

c) $X_1 \vee X_2 \vee X_3$
$-X_1 \vee -X_2 \vee -X_3$

d) $X_1 \vee X_2$
$-X_2 \vee X_3$
$-X_3$

5.8 Considere la gráfica en el problema 5.6. ¿Es 2-coloreable? Conteste esta pregunta aplicando alguna técnica de árbol de búsqueda.

5.9 Considere la gráfica siguiente. Aplique una técnica de árbol de búsqueda para demostrar que esta gráfica contiene un 3-círculo.



5.10 Diseñe un experimento basado en la estrategia de ramificar-y-acotar para probar el desempeño en el caso promedio del algoritmo que resuelve el problema del agente viajero.

capítulo

6

La estrategia prune-and-search

En este capítulo se abordará una estrategia elegante para el diseño de algoritmos, denominada prune-and-search. Este método puede aplicarse para resolver muchos problemas, especialmente de optimización, y suele proporcionar algoritmos eficientes para resolver estos problemas. Por ejemplo, cuando se fija la dimensión, puede usarse la estrategia prune-and-search para resolver en tiempo lineal el problema de programación lineal con n restricciones. En los siguientes apartados, primero se presentarán los pasos generales de la estrategia prune-and-search y luego se analizarán algunos problemas que pueden resolverse de manera eficiente con esta estrategia.

6-1 El método general

La estrategia prune-and-search siempre consta de varias iteraciones. En cada iteración, siempre se poda (o elimina) una fracción de los datos de entrada, por ejemplo f, y luego se invoca de manera recurrente el mismo algoritmo a fin de resolver el problema para los datos restantes. Luego de p iteraciones, el tamaño de los datos de entrada es q, que es tan pequeño que el problema puede resolverse directamente en algún tiempo constante c'. El análisis de la complejidad temporal de este tipo de algoritmos es como sigue: suponga que el tiempo necesario para ejecutar la prune-and-search en cada iteración es $O(n^k)$ para alguna constante k, y que el tiempo de ejecución del peor caso del algoritmo prune-and-search es $T(n)$. Entonces

$$T(n) = T((1 - f)n) + O(n^k).$$

Se tiene

$$\begin{aligned} T(n) &\leq T((1-f)n) + cn^k \qquad \text{para } n \text{ suficientemente grande.} \\ &\leq T((1-f)^2 n) + cn^k + c(1-f)^k n^k \\ &\quad \vdots \end{aligned}$$

$$\leq c' + cn^k + c(1 - f)^k n^k + c(1 - f)^{2k} n^k + \cdots + c(1 - f)^{pk} n^k$$
$$= c' + cn^k(1 + (1 - f)^k + (1 - f)^{2k} + \cdots + (1 - f)^{pk}).$$

Debido a que $1 - f < 1$, cuando $n \to \infty$,

$$T(n) = O(n^k).$$

La fórmula anterior indica que la complejidad temporal de todo el proceso prune-and-search es del mismo orden que la complejidad temporal de prune-and-search en cada iteración.

6-2 EL PROBLEMA DE SELECCIÓN

En este problema se tienen n elementos y se solicita determinar el k-ésimo menor elemento. Una solución posible de este problema es ordenar los n elementos y ubicar el k-ésimo elemento en la lista ordenada. La complejidad temporal en el peor caso de este método es tiempo $O(n \log n)$ porque la complejidad temporal del ordenamiento es $O(n \log n)$ y ésta es la dominante. En esta sección se mostrará que el problema de selección puede resolverse en tiempo lineal con la estrategia prune-and-search. El problema de la mediana, que consiste en encontrar el $\lceil n/2 \rceil$-ésimo menor elemento, es un caso especial del problema de selección. Así, el problema de la mediana también puede resolverse en tiempo lineal.

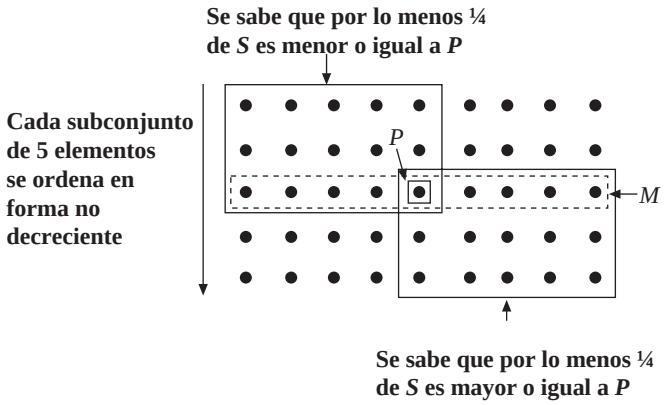
La idea fundamental de este algoritmo de selección prune-and-search en tiempo lineal es determinar una fracción de elementos que no contenga al k-ésimo elemento y descartar la fracción de elementos durante cada iteración. Con base en el apartado 6-1, se sabe que para tener un algoritmo $O(n)$ es necesario contar con un método que en cada iteración pode una fracción de elementos en tiempo $O(n)$. Se mostrará que en cada iteración, el tiempo necesario para la búsqueda es insignificante.

Sea S el conjunto de los datos de entrada. Sea p un elemento de S, conjunto que puede dividirse en tres subconjuntos S_1, S_2 y S_3 tales que S_1 contiene a todos los elementos menores que p, S_2 contiene a todos los elementos iguales a p y S_3 contiene a todos los elementos mayores que p. Si el tamaño de S_1 es mayor que k, entonces puede tenerse la certeza de que el k-ésimo menor elemento de S está en S_1, y es posible podar S_2 y S_3 durante la siguiente iteración. En caso contrario, si el número de elementos en S_1 y S_2 es mayor que k, entonces p es el k-ésimo menor elemento de S. Si no se cumple ninguna de las condiciones anteriores, entonces el k-ésimo menor elemento de S debe ser mayor que p. Por lo tanto, es posible descartar S_1 y S_2 y empezar la siguiente iteración, que selecciona el $(k - |S_1| - |S_2|)$-ésimo menor elemento de S_3.

La cuestión clave es cómo seleccionar p de modo que siempre sea posible descartar una fracción de S, sin importar si durante cada iteración se poda S_1, S_2 o S_3; p puede

seleccionarse como sigue: antes que todo, los n elementos se dividen en $\lceil n/5 \rceil$ subconjuntos de cinco elementos cada uno, sumando algunos elementos ∞ dummy, en caso de ser necesario, para completar el último subconjunto. Luego se ordena cada uno de los subconjuntos de cinco elementos. Se selecciona la mediana de cada subconjunto para formar una nueva secuencia $M = \{m_1, m_2, \ldots, m_{\lceil n/5 \rceil}\}$ y se hace que p sea la mediana de M. Como se muestra en la figura 6-1, por lo menos la cuarta parte de los elementos en S son menores que o iguales a p y por lo menos la cuarta parte de los elementos en S son mayores que o iguales a p.

FIGURA 6-1 Poda de puntos en el procedimiento de selección.



Entonces, si p se escoge de esta manera, siempre es posible podar por lo menos $n/4$ elementos de S durante cada iteración. Ahora es posible plantear el algoritmo.

Algoritmo 6-1 □ Un algoritmo de prune-and-search para encontrar el *k*-ésimo menor elemento

Input: Un conjunto S de n elementos.
Output: El k-ésimo menor elemento de S.
Paso 1. Si $|S| \leq 5$, resolver el problema aplicando cualquier método de fuerza bruta.
Paso 2. Divide S en $\lceil n/5 \rceil$ subconjuntos. Cada subconjunto contiene cinco elementos. Sumar algunos elementos dummy ∞ al último subconjunto si n no es un múltiplo neto de 5.
Paso 3. Ordenar cada subconjunto de elementos.
Paso 4. Recursivamente, encontrar el elemento p que es la mediana de las medianas de los $\lceil n/5 \rceil$ subconjuntos.

Paso 5. Partir S en S_1, S_2 y S_3, que contienen los elementos menores que, iguales a y mayores que p, respectivamente.

Paso 6. Si $|S_1| \geq k$, entonces se descartan S_2 y S_3 y se resuelve el problema que selecciona el k-ésimo menor elemento de S_1 durante la siguiente iteración; si $|S_1| + |S_2| \geq k$, entonces p es el k-ésimo menor elemento de S; en caso contrario, sea $k' = k - |S_1| - |S_2|$. Resolver el problema que selecciona el k'-ésimo menor elemento de S_3 durante la siguiente iteración.

Sea $T(n)$ la complejidad temporal del algoritmo que acaba de plantearse para seleccionar el k-ésimo elemento de S. Debido a que cada subconjunto contiene un número constante de elementos, para cada uno de éstos el ordenamiento requiere un tiempo constante. Así, los pasos 2, 3 y 5 pueden ejecutarse en tiempo $O(n)$ y el paso 4 requiere tiempo $T(\lceil n/5 \rceil)$ si el mismo algoritmo se usa de manera recurrente para encontrar la mediana de los $\lceil n/5 \rceil$ elementos. Debido a que durante cada iteración siempre se podan por lo menos $n/4$ elementos, el problema restante en el paso 6 contiene a lo sumo $3n/4$ elementos, de modo que es posible realizarlo en tiempo $T(3n/4)$. Así,

$$T(n) = T(3n/4) + T(n/5) + O(n).$$

Let $T(n) = a_0 + a_1 n + a_2 n^2 + \cdots$, $a_1 \neq 0$.

Se tiene

$$T\left(\frac{3}{4}n\right) = a_0 + \frac{3}{4}a_1 n + \frac{9}{16}a_2 n^2 + \cdots$$

$$T\left(\frac{1}{5}n\right) = a_0 + \frac{1}{5}a_1 n + \frac{1}{25}a_2 n^2 + \cdots$$

$$T\left(\frac{3}{4}n + \frac{1}{5}n\right) = T\left(\frac{19}{20}n\right) = a_0 + \frac{19}{20}a_1 n + \frac{361}{400}a_2 n^2 + \cdots$$

Así,

$$T\left(\frac{3}{4}n\right) + T\left(\frac{1}{5}n\right) \leq a_0 + T\left(\frac{19}{20}n\right).$$

En consecuencia,

$$\begin{aligned} T(n) &= T\left(\frac{3}{4}n\right) + T\left(\frac{1}{5}n\right) + O(n) \\ &\leq T\left(\frac{19}{20}n\right) + cn. \end{aligned}$$

Cuando a esta desigualdad se aplica la fórmula que se obtuvo en el apartado 6-1, el resultado es

$T(n) = O(n)$.

Así, se tiene un algoritmo de tiempo lineal para el peor caso para resolver el problema de selección con base en la estrategia prune-and-search.

6-3 Programación lineal con dos variables

La complejidad computacional de la programación lineal ha constituido un tema de gran interés para los científicos expertos en computación durante mucho tiempo. Aunque un algoritmo brillante fue propuesto por Khachian, quien demostró que el problema de programación lineal puede resolverse en tiempo polinomial, es de interés teórico porque la constante implicada es demasiado grande. Megiddo y Dyer propusieron de manera independiente una estrategia prune-and-search para resolver el problema de programación lineal con un número fijo de variables en tiempo $O(n)$, donde n es el número de restricciones.

En esta sección se describirán sus elegantes técnicas para resolver el problema de programación lineal con dos variables. A continuación se describe el problema especial de programación lineal con dos variables:

Minimizar $ax + by$

Sujeta a $a_i x + b_i y \geq c_i, \quad i = 1, 2, \ldots, n.$

La idea básica de la estrategia prune-and-search para resolver el problema de programación lineal con dos variables es que siempre hay algunas restricciones que no tienen nada que ver con la solución y en consecuencia pueden podarse. En el método prune-and-search, después de cada iteración se poda una fracción de las restricciones. Luego de varias iteraciones, el número de restricciones es tan pequeño que el problema de programación lineal puede resolverse en algún tiempo constante.

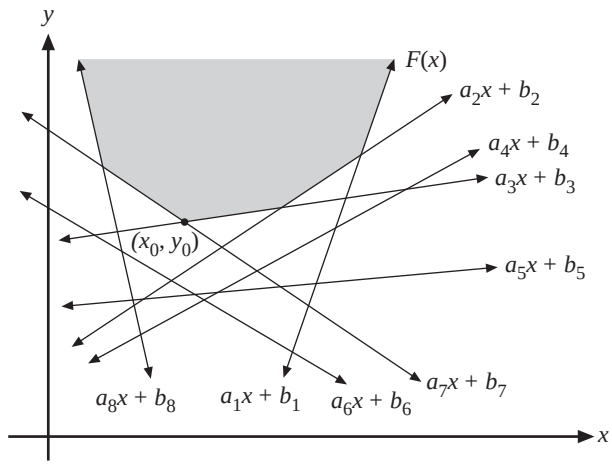
Con el objeto de simplificar la exposición, en seguida se describirá el método prune-and-search para resolver un problema simplificado de programación lineal con dos variables:

Minimizar y

Sujeta a $y \geq a_i x + b_i, \quad i = 1, 2, \ldots, n.$

Considere la figura 6-2. Ahí hay ocho restricciones en total y (x_0, y_0) es la solución óptima.

FIGURA 6-2 Ejemplo del problema especial de programación lineal con dos variables.



Debido a que

$$y \geq a_i x + b_i \quad \text{para toda } i,$$

se sabe que esta solución óptima debe estar en la frontera que rodea la región factible, como se muestra en la figura 6-2. Para toda x, la frontera $F(x)$ debe asumir el mayor valor de entre todas las n restricciones. Es decir,

$$F(x) = \max_{1 \leq i \leq n} \{a_i x + b_i\}$$

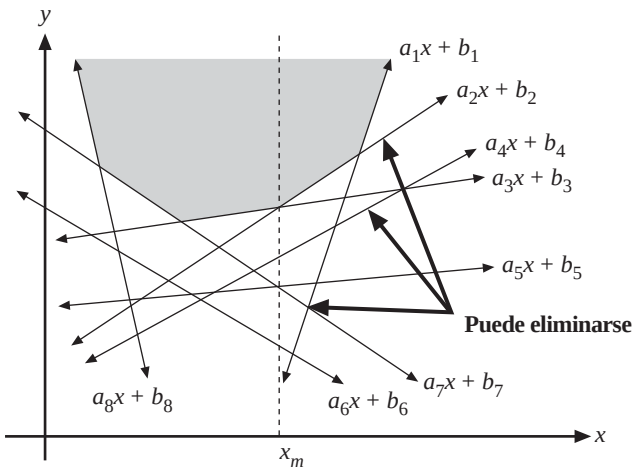
donde $a_i x + b_i$ es una restricción de entrada. La solución óptima x_0 satisface la siguiente ecuación:

$$F(x_0) = \min_{-\infty \leq x \leq \infty} F(x).$$

Ahora se supondrá lo siguiente:

1. Se ha escogido un punto x_m, como se muestra en la figura 6-3.
2. Se comprende que $x_0 \leq x_m$.

FIGURA 6-3 Restricciones que pueden eliminarse en el problema de programación lineal con dos variables.



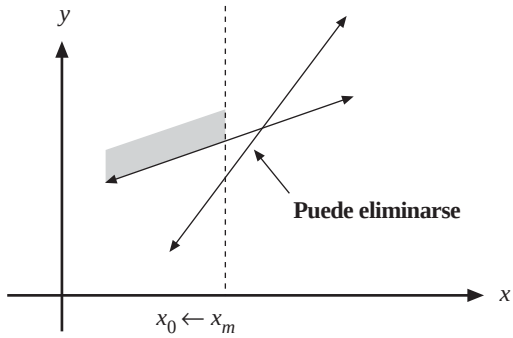
En este caso, considere

$$y = a_1x + b_1,$$

y $\quad y = a_2x + b_2.$

La intersección de estas dos rectas se encuentra a la derecha de x_m. De estas dos rectas, una es menor que la otra para $x < x_m$. Es posible eliminar esta restricción porque no puede formar parte de la frontera, como se ilustra en la figura 6-4.

FIGURA 6-4 Ilustración de por qué es posible eliminar una restricción.



Entonces, si se selecciona x_m y se sabe que $x_0 < x_m$, entonces $a_1x + b_1$ puede eliminarse.

De manera semejante, debido a que la intersección de $a_3x + b_3$ y $a_4x + b_4$ está a la derecha de x_m, $x_0 < x_m$ y $a_4x + b_4 < a_3x + b_3$ para $x \leq x_m$, es posible eliminar la restricción $a_4x + b_4$.

En general, si existe una x_m así, tal que la solución óptima $x_0 < x_m$ y la intersección de dos restricciones de entrada $a_1x + b_1$ y $a_2x + b_2$ esté a la derecha de x_m, entonces una de estas restricciones siempre es menor que la otra para $x < x_m$. En consecuencia, esta restricción puede eliminarse porque no afecta la búsqueda de la solución óptima.

El lector debe observar que la eliminación de la restricción $a_1x + b_1$ sólo puede hacerse después que se sabe que $x_0 < x_m$. En otras palabras, suponga que se está ejecutando un proceso de búsqueda, y que ésta se lleva a cabo a lo largo de la dirección de x_m a x_0 sobre la frontera. Entonces es posible eliminar $a_1x + b_1$ porque donde ocurre que $x < x_m$, $a_1x + b_1$ no contribuye al resultado de la búsqueda. Debe observarse que $a_1x + b_1$ forma parte de la frontera de la región factible. Pero esto ocurre para $x > x_m$.

Si $x_0 > x_m$, entonces para cada par de restricciones cuya intersección esté a la izquierda de x_m, hay una restricción que es menor que la otra para $x > x_m$ y esta restricción puede eliminarse sin afectar la solución.

Aún es necesario contestar las dos preguntas siguientes:

1. Suponga que se conoce x_m. ¿Cómo se conoce la dirección de búsqueda? Es decir, ¿cómo se sabe si $x_0 < x_m$ o $x_0 > x_m$?
2. ¿Cómo se escoge x_m?

Se contestará la primera pregunta. Suponga que se escoge x_m. Sea

$$y_m = F(x_m) = \max_{1 \leq i \leq n} \{a_i x_m + b_i\}.$$

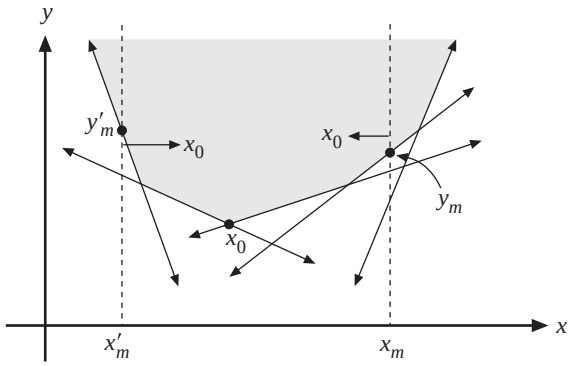
Resulta evidente que (x_m, y_m) es un punto sobre la frontera de la región factible. Hay dos posibilidades:

Caso 1. y_m sólo está en una restricción.

Caso 2. y_m está en la intersección de varias restricciones.

Para el caso 1, simplemente se comprueba la pendiente g de esta restricción. Si $g > 0$, entonces $x_0 < x_m$. Si $g < 0$, entonces $x_0 > x_m$. Esto se ilustra en la figura 6-5.

FIGURA 6-5 Casos en que x_m sólo está en una restricción.



Para el caso 2, se calcula lo siguiente:

$$g_{\text{máx}} = \max_{1 \le i \le n} \{a_i : a_i x_m + b_i = F(x_m)\}$$

$$g_{\text{mín}} = \min_{1 \le i \le n} \{a_i : a_i x_m + b_i = F(x_m)\}.$$

En otras palabras, de todas las restricciones que se cortan entre sí en (x_m, y_m), sean $g_{\text{mín}}$ y $g_{\text{máx}}$ las pendientes mínima y máxima de estas restricciones, respectivamente. Así, hay tres posibilidades:

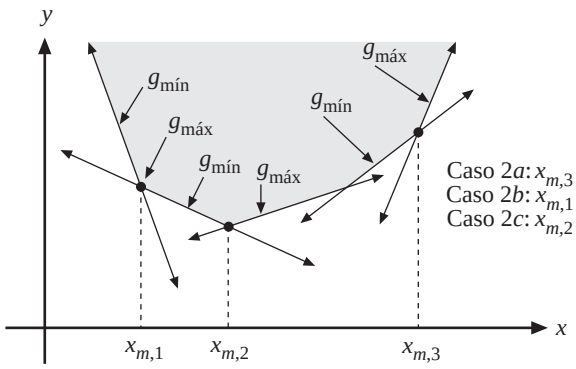
1. Caso 2*a*. $g_{\text{mín}} > 0$ y $g_{\text{máx}} > 0$. En este caso, $x_0 < x_m$.
2. Caso 2*b*. $g_{\text{mín}} < 0$ y $g_{\text{máx}} < 0$. En este caso, $x_0 > x_m$.
3. Caso 2*c*. $g_{\text{mín}} < 0$ y $g_{\text{máx}} > 0$. En este caso, (x_m, y_m) es la solución óptima.

Estos tres casos se muestran en la figura 6-6.

Se ha contestado la primera pregunta. Ahora se responderá la segunda: ¿cómo se escoge x_m? *La x_m debe escogerse de modo que la mitad de las intersecciones entre rectas pareadas en el par escogido estén a la derecha de x_m y la mitad de este tipo de intersecciones esté a la izquierda de x_m. Esto significa que para n restricciones, arbitrariamente pueden agruparse en n/2 pares. Para cada par, se encuentra su intersección. De estas n/2 intersecciones, sea x_m la mediana de sus coordenadas x.*

En el algoritmo 6-2 se proporcionará a detalle el algoritmo basado en la estrategia prune-and-search para encontrar una solución óptima del problema de programación lineal especial.

FIGURA 6-6 Casos en que x_m está en la intersección de varias restricciones.



Algoritmo 6-2 □ Algoritmo de prune-and-search para resolver un problema de programación lineal especial

Input: Restricciones: $S : a_i x + b_i, \quad i = 1, 2, \ldots, n.$

Output: El valor de x_0 tal que y se minimiza en x_0 sujeta a $y \geq a_i x + b_i, \quad i = 1, 2, \ldots, n.$

Paso 1. Si S no contiene más de dos restricciones, resolver este problema con un método de fuerza bruta.

Paso 2. Dividir S en $n/2$ pares de restricciones. Para cada par de restricciones $a_i x + b_i$ y $a_j x + b_j$, buscar la intersección p_{ij} de éstas y denote su abscisa por x_{ij}.

Paso 3. Entre las x_{ij} (a lo sumo $n/2$ de ellas) encontrar la mediana x_m.

Paso 4. Determinar $y_m = F(x_m) = \max\limits_{1 \leq i \leq n} \{a_i x_m + b_i\}$

$$g_{\text{mín}} = \min_{1 \leq i \leq n} \{a_i : a_i x_m + b_i = F(x_m)\}$$

$$g_{\text{máx}} = \max_{1 \leq i \leq n} \{a_i : a_i x_m + b_i = F(x_m)\}.$$

Paso 5. Caso 5*a*: Si $g_{\text{mín}}$ y $g_{\text{máx}}$ no tienen el mismo signo, y_m es la solución y salir.
Caso 5*b*: De otra forma, $x_0 < x_m$, si $g_{\text{mín}} > 0$, y $x_0 > x_m$, si $g_{\text{mín}} < 0$.

Paso 6. Caso 6*a*: Si $x_0 < x_m$, para cada par de restricciones cuya intersección de la coordenada x es mayor que x_m, podar la restricción que siempre es menor que la otra para $x \leq x_m$.

Caso 6*b*: Si $x_0 > x_m$, para cada par de restricciones cuya intersección de la coordenada x es menor que x_m, entonces se debe podar la restricción que siempre es menor que la otra para $x \geq x_m$.
Hacer S que denota al resto de las restricciones. Ir a paso 2.

La complejidad del algoritmo anterior se analiza como sigue: debido a que la intersección de dos rectas puede encontrarse en tiempo constante, el paso 2 puede ejecutarse en tiempo $O(n)$. La mediana puede encontrarse en tiempo $O(n)$, como se muestra en el apartado 6-2. El paso 4 puede ejecutarse al explorar linealmente todas estas restricciones. Por lo tanto, los pasos 4 y 5 pueden completarse en tiempo $O(n)$. El paso 6 también puede efectuarse en tiempo $O(n)$ al explorar todos los pares de intersecciones.

Debido a que se usa la mediana de las intersecciones escogidas, la mitad de ellas están a la derecha de x_m. En total hay $\lfloor n/2 \rfloor$ intersecciones en total. Para cada intersección, se poda una restricción. Así, en cada iteración se podan $\lfloor n/4 \rfloor$ restricciones. Con base en el resultado del apartado 6-1, la complejidad temporal del algoritmo anterior es $O(n)$.

Ahora se regresará al problema de programación lineal original con dos variables, que se define como sigue:

$$\begin{aligned} &\text{Minimizar } Z = ax + by \\ &\text{Sujeta a } \quad a_i x + b_i y \geq c_i \ (i = 1, 2, \ldots, n). \end{aligned}$$

A continuación se presenta un ejemplo típico:

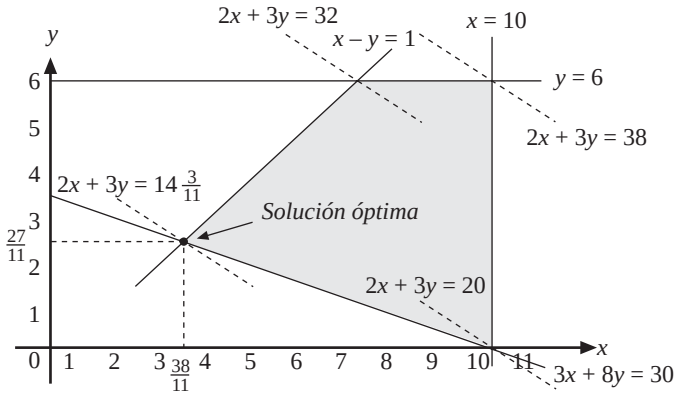
$$\begin{aligned} &\text{Minimizar } Z = 2x + 3y \\ &\text{Sujeta a } \quad x \leq 10 \\ &\qquad\qquad\ \ y \leq 6 \\ &\qquad\qquad\ \ x - y \geq 1 \\ &\qquad\qquad\ \ 3x + 8y \geq 30. \end{aligned}$$

Las restricciones forman una región factible, que se muestra en la figura 6-7. En esta figura, cada línea discontinua se representa como

$$Z = 2x + 3y.$$

Es posible imaginar que estas rectas forman una secuencia de rectas paralelas. La solución óptima se encuentra en (38/11, 27/11), donde la primera recta discontinua corta la región factible.

FIGURA 6-7 Problema general de programación lineal con dos variables.



El problema de programación lineal general con dos variables anterior puede transformarse en un problema de minimización sólo del valor *y*. El problema original es

Minimizar $Z = ax + by$
Sujeta a $a_i x + b_i y \geq c_i$ $(i = 1, 2, \ldots, n)$.
Ahora se hace $x' = x$
$y' = ax + by$.

Luego, este problema se convierte en

Minimizar y'
Sujeta a $a_i' x' + b_i' y' \geq c_i'$ $(i = 1, 2, \ldots, n)$
donde $a_i' = a_i - b_i a/b$
$b_i' = b_i/b$
y $c_i' = c_i$.

En consecuencia, puede afirmarse que un problema general con dos variables puede transformarse, después de *n* pasos, en el siguiente problema.

Minimizar y
Sujeta a $a_i x + b_i y \geq c_i$ $(i = 1, 2, \ldots, n)$.

El lector debe observar que hay una diferencia entre el problema anterior y el problema especial de programación lineal con dos variables. Este problema se definió como

Minimizar y
Sujeto a $y \geq a_i x + b_i$ $(i = 1, 2, \ldots, n)$.

Pero, en el caso general, hay tres clases de restricciones:

$$y \geq a_i x + b_i$$
$$y \leq a_i x + b_i$$
y $a \leq x \leq b$.

Considere las cuatro restricciones en la figura 6-7. Son

$$x \leq 10$$
$$y \leq 6$$
$$x - y \geq 1$$
$$3x + 8y \geq 30.$$

Las ecuaciones anteriores pueden volver a escribirse como

$$x \leq 10$$
$$y \leq 6$$
$$y \leq x - 1$$
$$y \geq \frac{-3}{8} x + \frac{30}{8}$$

Es posible agrupar las restricciones con coeficientes positivos (negativos) para la variable y en el conjunto I_1 (I_2). Así, el problema original se vuelve

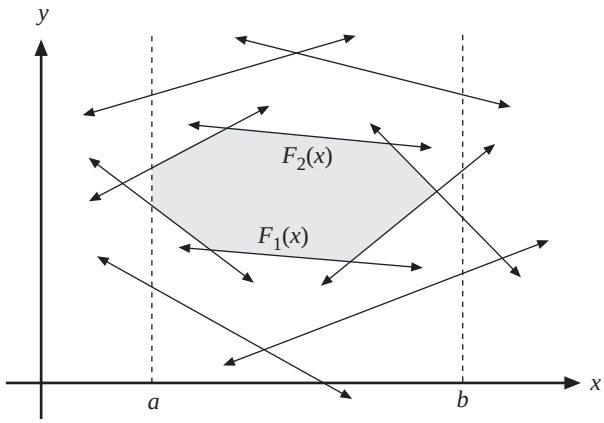
Minimizar y
Sujeto a $y \geq a_i x + b_i$ $(i \in I_1)$
$y \leq a_i x + b_i$ $(i \in I_2)$
$a \leq x \leq b$.

Se definirán dos funciones:

$$F_1(x) = \text{máx}\{a_i x + b_i : i \in I_1\}$$
$$F_2(x) = \text{mín}\{a_i x + b_i : i \in I_2\}.$$

$F_1(x)$ es una función lineal convexa por piezas, $F_2(x)$ es una función lineal cóncava por piezas y estas dos funciones junto con la restricción $a \leq x \leq b$ definen la región factible del problema de programación lineal que se muestra en la figura 6-8.

FIGURA 6-8 Una región factible del problema de programación lineal con dos variables.



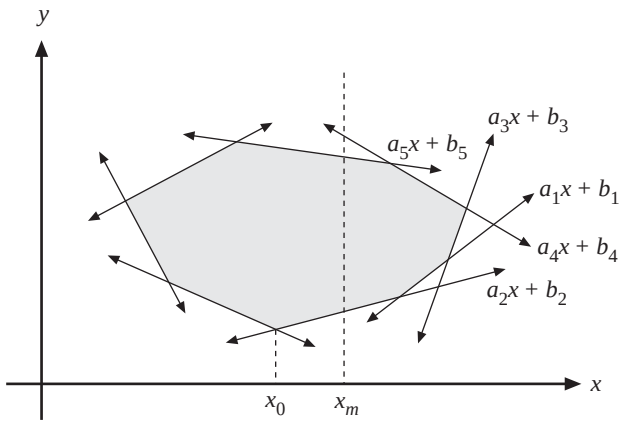
Así, el problema original de programación lineal con dos variables puede transformase todavía más en

$$\begin{aligned} &\text{Minimizar } F_1(x) \\ &\text{Sujeto a } \quad F_1(x) \leq F_2(x) \\ &\qquad\qquad a \leq x \leq b. \end{aligned}$$

De nuevo, es conveniente indicar aquí que algunas restricciones pueden podarse si se conoce la dirección de búsqueda. Esta razón es exactamente la misma que se utilizó en el análisis del problema especial de programación lineal con dos variables. Considere la figura 6-9. Si se sabe que la solución óptima x_0 está a la izquierda de x_m, entonces $a_1x + b_1$ puede eliminarse sin afectar la solución porque $a_1x + b_1 < a_2x + b_2$ para $x < x_m$. Este hecho se conoce debido a que la intersección de $a_1x + b_1$ y $a_2x + b_2$ está a la derecha de x_m. De manera semejante, $a_4x + b_4$ puede eliminarse porque $a_4x + b_4 > a_5x + b_5$ para $x < x_m$.

Ahora, el lector observará que la cuestión más importante en el algoritmo prune-and-search para resolver el problema de programación lineal con dos variables es decidir si x_0 (la solución óptima) está a la izquierda, o derecha, de x_m. La solución de

FIGURA 6-9 Poda de las restricciones para el problema general de programación lineal con dos variables.



x_m puede encontrarse de manera semejante a como se hizo en el problema especial de programación lineal con dos variables.

En general, dado un punto x_m, $a \leq x_m \leq b$, es necesario decidir lo siguiente:

1. ¿x_m es factible?
2. Si x_m es factible, es necesario decidir si la solución óptima x_0 está a la izquierda, o derecha, de x_m. También puede ocurrir que x_m en sí sea la solución óptima.
3. Si x_m no es factible, es necesario decidir si existe alguna solución factible. En caso afirmativo, es necesario decidir a qué lado de x_m se encuentra esta solución óptima.

A continuación se describirá el proceso de decisión. Considere $F(x) = F_1(x) - F_2(x)$. Resulta evidente que x_m es factible si y sólo si $F(x_m) \leq 0$. Para decidir de qué lado está la solución óptima, se definirá lo siguiente:

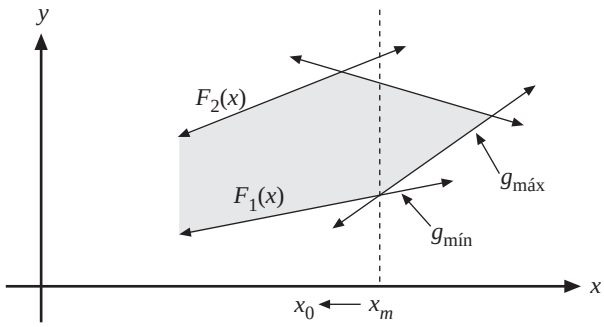
$$g_{\text{mín}} = \text{mín}\{a_i : i \in I_1,\ a_i x_m + b_i = F_1(x_m)\}$$
$$g_{\text{máx}} = \text{máx}\{a_i : i \in I_1,\ a_i x_m + b_i = F_1(x_m)\}$$
$$h_{\text{mín}} = \text{mín}\{a_i : i \in I_2,\ a_i x_m + b_i = F_2(x_m)\}$$
$$h_{\text{máx}} = \text{máx}\{a_i : i \in I_2,\ a_i x_m + b_i = F_2(x_m)\}.$$

Se consideran los casos siguientes:

Caso 1. $F(x_m) \leq 0$.
Esto significa que x_m es factible.

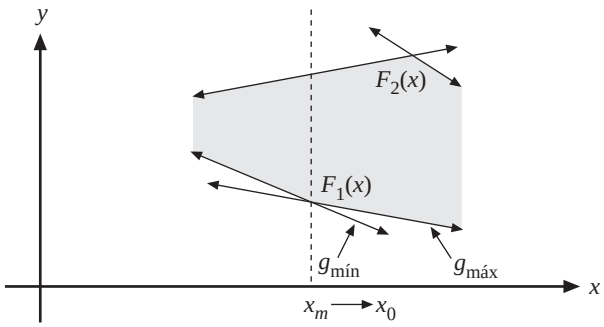
1. Si $g_{\text{mín}} > 0$ y $g_{\text{máx}} > 0$, entonces $x_0 < x_m$, lo cual se muestra en la figura 6-10.

FIGURA 6-10 Caso en que $g_{\text{mín}} > 0$ y $g_{\text{máx}} > 0$.



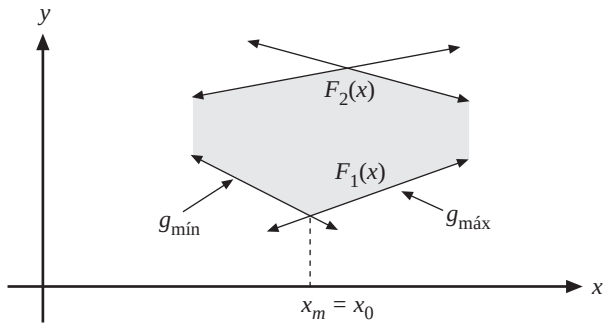
2. Si $g_{\text{máx}} < 0$ y $g_{\text{mín}} < 0$, entonces $x_0 > x_m$, lo cual se muestra en la figura 6-11.

FIGURA 6-11 Caso en que $g_{\text{máx}} < 0$ y $g_{\text{mín}} < 0$.



3. Si $g_{\text{mín}} < 0$ y $g_{\text{máx}} > 0$, entonces x_m es la solución óptima, lo cual se muestra en la figura 6-12.

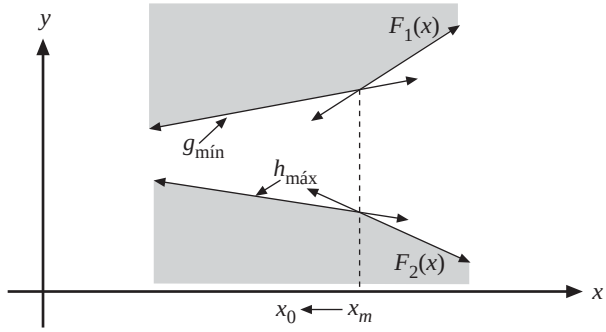
FIGURA 6-12 Caso en que $g_{\text{mín}} < 0$ y $g_{\text{máx}} > 0$.



Caso 2. $F(x_m) > 0$.
Esto significa que x_m no es factible.

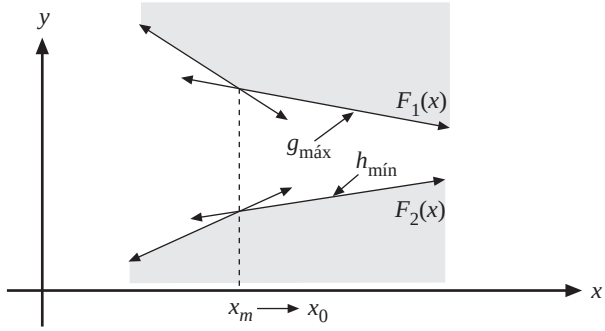
1. Si $g_{\text{mín}} > h_{\text{máx}}$, entonces $x_0 < x_m$, lo cual se muestra en la figura 6-13.

FIGURA 6-13 Caso en que $g_{\text{mín}} > h_{\text{máx}}$.



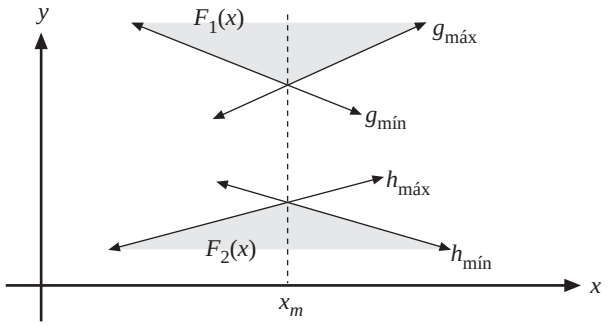
2. Si $g_{\text{máx}} < h_{\text{mín}}$, entonces $x_0 > x_m$, lo cual se muestra en la figura 6-14.

FIGURA 6-14 Caso en que $g_{máx} < h_{mín}$.



3. Si ($g_{mín} \leq h_{máx}$) y ($g_{máx} \geq h_{mín}$), entonces no existe solución factible porque $F(x)$ alcanza su mínimo en x_m. Esto se muestra en la figura 6-15.

FIGURA 6-15 Caso en que ($g_{mín} \leq h_{máx}$) y ($g_{máx} \geq h_{mín}$).



El procedimiento de decisión anterior se resume en el siguiente procedimiento.

Procedimiento 6-1 □ Un procedimiento usado en el algoritmo 6-3

Input: x_m, el valor de la coordenada x de un punto.

I_1: $y \geq a_i x + b_i$, $i = 1, 2, \ldots, n_1'$

I_2: $y \leq a_i x + b_i$, $i = n_1' + 1, n'_1 + 2, \ldots, n'$

$a \leq x \leq b$.

Output: La decisión de si tiene sentido continuar la búsqueda de x_m, y en caso afirmativo, la dirección de búsqueda.

Paso 1. $F_1(x) = \text{máx}\{a_i x + b_i : i \in I_1\}$

$F_2(x) = \text{mín}\{a_i x + b_i : i \in I_2\}$

$F(x) = F_1(x) - F_2(x)$.

Paso 2. $g_{\text{mín}} = \text{mín}\{a_i : i \in I_1, a_i x_m + b_i = F_1(x_m)\}$

$g_{\text{máx}} = \text{máx}\{a_i : i \in I_1, a_i x_m + b_i = F_1(x_m)\}$

$h_{\text{mín}} = \text{mín}\{a_i : i \in I_2, a_i x_m + b_i = F_2(x_m)\}$

$h_{\text{máx}} = \text{máx}\{a_i : i \in I_2, a_i x_m + b_i = F_2(x_m)\}$.

Paso 3. **Caso 1:** $F(x_m) \leq 0$.

a) Si $g_{\text{mín}} > 0$ y $g_{\text{máx}} > 0$, reportar "$x_0 < x_m$" y salir.

b) Si $g_{\text{mín}} < 0$ y $g_{\text{máx}} < 0$, reportar "$x_0 > x_m$" y salir.

c) Si $g_{\text{mín}} < 0$ y $g_{\text{máx}} > 0$, reportar "x_m es la solución óptima" y salir.

Caso 2: $F(x_m) > 0$.

a) Si $g_{\text{mín}} > h_{\text{máx}}$, reportar "$x_0 < x_m$" y salir.

b) Si $g_{\text{máx}} < h_{\text{mín}}$, reportar "$x_0 > x_m$" y salir.

c) Si ($g_{\text{mín}} \leq h_{\text{máx}}$) y ($g_{\text{máx}} \geq h_{\text{mín}}$), reportar "no existe solución factible" y salir.

A continuación se presenta el algoritmo prune-and-search para resolver el problema de programación lineal con dos variables.

Algoritmo 6-3 □ Un algoritmo prune-and-search para resolver el problema de programación lineal con dos variables

Input: I_1: $y \geq a_i x + b_i$, $i = 1, 2, \ldots, n_1$.

I_2: $y \leq a_i x + b_i$, $i = n_1 + 1, n_1 + 2, \ldots, n$.

$a \leq x \leq b$.

Output: El valor de x_0 tal que y se minimice en x_0 sujeta a

$y \geq a_i x + b_i$, $i = 1, 2, \ldots, n_1$.

$y \leq a_i x + b_i$, $i = n_1 + 1, n_1 + 2, \ldots, n$.

$a \leq x \leq b$.

Paso 1. Si no hay más que dos restricciones en I_1 e I_2, entonces resolver este problema con un método de fuerza bruta.

Paso 2. Arreglar las restricciones en I_1 en pares ajenos; hacer lo mismo con las restricciones en I_2. Para cada par, si $a_i x + b_i$ es paralela a $a_j x + b_j$, delete $a_i x + b_i$ si $b_i < b_j$ para $i, j \in I_1$ o $b_i > b_j$ for $i, j \in I_2$. De otra forma, encontrar la intersección p_{ij} de $y = a_i x + b_i$ y $a_j x + b_j$. Hacer x_{ij} la coordenada x de p_{ij}.

Paso 3. Busca la mediana x_m de las x_{ij}.

Paso 4. Aplicar el procedimiento 6-1 a x_m.
Si x_m es óptima, reportar x_m y salir.
Si no existe una solución factible, reportar este hecho y salir.

Paso 5. Si $x_0 > x_m$
Para toda $x_{ij} < x_m$ e $i, j \in I_1$, podar la restricción $y \geq a_i x + b_i$ si $a_i < a_j$;
de otra forma, podar la restricción $y \geq a_j x + b_j$.
Para toda $x_{ij} < x_m$ e $i, j \in I_2$, podar la restricción $y \leq a_i x + b_i$ si $a_i > a_j$;
de otra forma, podar la restricción $y \leq a_j x + b_j$.
Si $x_0 < x_m$
Para toda $x_{ij} > x_m$ e $i, j \in I_1$, podar la restricción $y \geq a_i x + b_i$ si $a_i > a_j$;
de otra forma, podar la restricción $y \geq a_j x + b_j$.
Para toda $x_{ij} > x_m$ e $i, j \in I_2$, podar la restricción $y \leq a_i x + b_i$ si $a_i < a_j$;
de otra forma, podar la restricción $y \leq a_j x + b_j$.

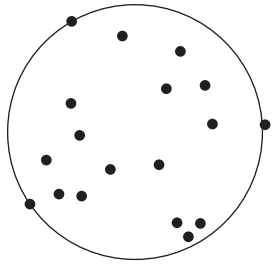
Paso 6. Ir a paso 1.

Debido a que es posible podar $\lfloor n/4 \rfloor$ restricciones por cada iteración, y como cada paso en el algoritmo 6-3 requiere tiempo $O(n)$, resulta fácil ver que este algoritmo es de orden $O(n)$.

6-4 EL PROBLEMA CON UN CENTRO

Este problema se define como sigue: se tiene un conjunto de n puntos planos y la tarea es encontrar el círculo más pequeño que cubre estos n puntos. En la figura 6-16 se muestra un ejemplo típico.

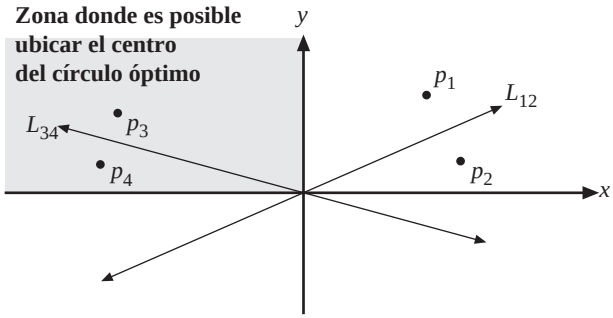
FIGURA 6-16 El problema con un centro.



En este apartado se presentará una estrategia para resolver el problema con un centro basada en la estrategia prune-and-search. Para este problema, los datos de entrada son estos n puntos. Durante cada etapa se poda $^1/_{16}$ de los puntos que no afectan la solución. El algoritmo de prune-and-search para el problema con un centro se lleva a cabo sólo en tiempo lineal, ya que para cada etapa se requiere de tiempo lineal.

A continuación se ilustrará el algoritmo de prune-and-search a partir de la figura 6-17. En esta figura, L_{12} y L_{34} son bisectrices de segmentos que unen p_1 y p_2, y p_3 y p_4, respectivamente. Suponga también que el centro del círculo óptimo debe estar en el área sombreada; entonces se considera L_{12}. Debido a que L_{12} no corta esta área, debe haber un punto que esté más próximo a este centro óptimo que el otro. En nuestro caso, p_1 está más cerca del centro óptimo que p_2. Así, es posible eliminar p_1 porque no afecta la solución.

FIGURA 6-17 Posible poda de puntos en el problema con un centro.



Esencialmente, es necesario conocer la región donde se localiza el centro óptimo y debe tenerse la certeza de que algunos puntos están más cerca de este centro óptimo que los otros. En consecuencia, estos puntos más próximos pueden eliminarse.

Antes de abordar el algoritmo general para el problema con un centro, primero se presentará un algoritmo para *el problema con un centro restringido, donde el centro se restringe a estar en una recta*. Luego, este algoritmo se usará como una subrutina llamada por el algoritmo general de un centro. Se supondrá, entonces, que la recta es $y = y'$.

Algoritmo 6-4 □ Un algoritmo para resolver el problema con un centro restringido

Input: n puntos y una recta $y = y'$.

Output: El centro restringido sobre la recta $y = y'$.

Paso 1. Si n no es mayor que 2, resolver este problema con un método de fuerza bruta.

Paso 2. Formar pares ajenos de puntos (p_1, p_2), (p_3, p_4), …, (p_{n-1}, p_n). Si el número de puntos es impar, entonces dejar que el último par sea (p_n, p_1).

Paso 3. Para cada par de puntos (p_i, p_{i+1}), buscar el punto $x_{i,i+1}$ sobre la recta $y = y'$ de modo que $d(p_i, x_{i,i+1}) = d(p_{i+1}, x_{i,i+1})$.

Paso 4. Buscar la mediana de los $\lceil n/2 \rceil$ $x_{i,i+1}$. Nombrar como x_m.

Paso 5. Calcular la distancia entre p_i y x_m para toda i. Sea p_j el punto más alejado de x_m. Sea x_j la proyección de p_j sobre $y = y'$. Si x_j está a la izquierda (derecha) de x_m, entonces la solución óptima, x^*, debe estar a la izquierda (derecha) de x_m.

Paso 6. Si $x^* < x_m$ (lo cual se ilustra en la figura 6-18)

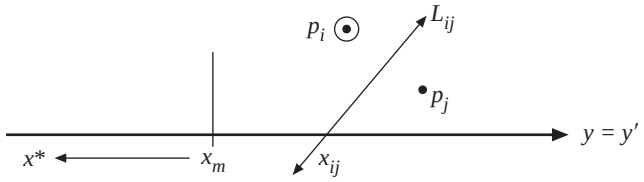
Para toda $x_{i,i+1} > x_m$, podar el punto p_i si p_i está más próximo a x_m que el punto p_{i+1}; de otra forma, podar el punto p_{i+1}.

Si $x^* > x_m$

Para toda $x_{i,i+1} < x_m$, podar el punto p_i si p_i está más próximo a x_m que el punto p_{i+1}; de otra forma, podar el punto p_{i+1}.

Paso 7. Ir a paso 1.

FIGURA 6-18 Poda de los puntos en el problema con un centro restringido.



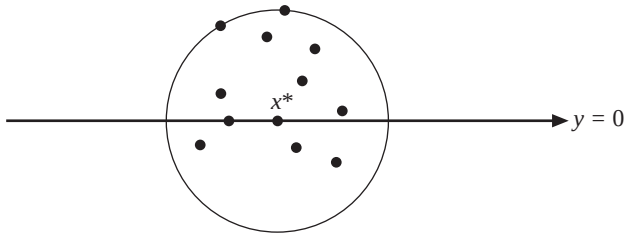
Debido a que hay $\lfloor n/4 \rfloor$ $x_{i,i+1}$ a la izquierda (derecha) de x_m, es posible podar $\lfloor n/4 \rfloor$ puntos por cada iteración y algoritmo. Cada iteración requiere tiempo $O(n)$. Por lo tanto, la complejidad temporal de este algoritmo es

$$\begin{aligned} T(n) &= T(3n/4) + O(n) \\ &= O(n) \end{aligned}$$

Aquí es necesario recalcar que el algoritmo con un centro restringido se usará en el algoritmo principal para encontrar una solución óptima. Los puntos podados en el proceso de ejecución de este problema con un centro restringido seguirán siendo utilizados por el algoritmo principal.

A continuación se considerará un problema más complicado. Imagine que se tiene un conjunto de puntos y una recta $y = 0$, como se muestra en la figura 6-19. Al usar el algoritmo con un centro restringido es posible determinar la ubicación exacta de x^* en esta recta. En realidad, al usar esta información, puede lograrse más. Sea (x_s, y_s) el centro del círculo óptimo que contiene todos los puntos. Ahora es posible determinar si $y_s > 0$, $y_s < 0$ o $y_s = 0$. Por la misma razón, también es posible determinar si $x_s > 0$, $x_s < 0$ o $x_s = 0$.

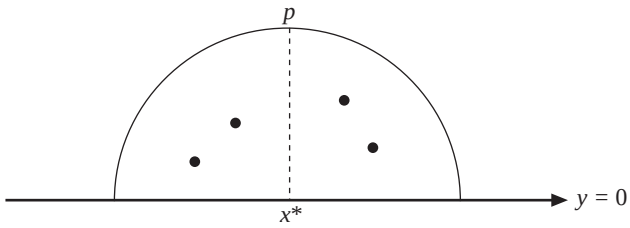
FIGURA 6-19 Solución de un problema con un centro restringido para el problema con un centro.



Sea I el conjunto de puntos más alejados de $(x^*, 0)$. Hay dos casos posibles.

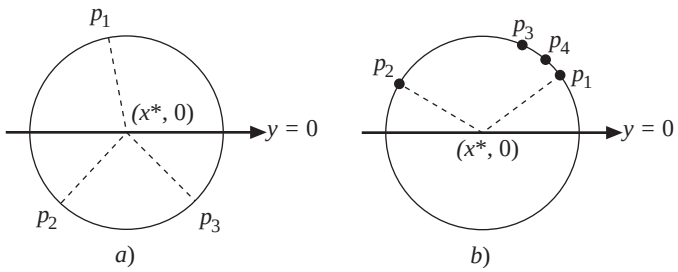
Caso 1. I contiene un punto; por ejemplo, p.
En este caso, la abscisa de p debe ser igual a x.* Suponga lo contrario. Entonces es posible desplazar x^* hacia p a lo largo de la recta $y = 0$. Esto contradice la hipótesis de que $(x^*, 0)$ es una solución óptima sobre la recta $y = 0$. En consecuencia, si p es el único punto más alejado de $(x^*, 0)$, su abscisa debe ser igual a x^*, lo cual se muestra en la figura 6-20. *Se concluye que y_s tiene el mismo signo que el de la ordenada de p.*

FIGURA 6-20 Caso en que *I* sólo contiene un punto.



Caso 2. *I* contiene más de un punto.
De todos los puntos en el conjunto I, se encuentra el menor arco que genera todos los puntos de I. Sean p_1 y p_2 *los dos puntos extremos de este arco. Si el arco es de grado mayor que o igual a 180°, entonces* $y_s = 0$. *En caso contrario, sean* y_i *la ordenada de* p_i y $y_c = (y_1 + y_2)/2$. *Entonces puede demostrarse que el signo de* y_s *es igual al de* y_c. Estas situaciones se ilustran en las figuras 6-21*a*) y 6-21*b*), respectivamente, y se analizan a continuación.

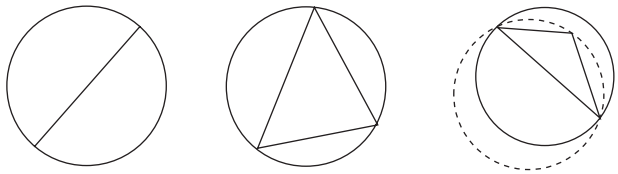
FIGURA 6-21 Casos en que *I* contiene más de un punto.



A continuación se considerará el primer caso; a saber, cuando el menor arco que genera todos los puntos más alejados es de grado mayor o igual a 180°. Observe que un menor círculo que contiene un conjunto de puntos se define ya sea por dos o tres puntos de este conjunto, lo cual se muestra en la figura 6-22*a*) y 6-22*b*). Tres puntos definen la frontera de un menor círculo que encierra a todos los tres puntos, si y sólo si no forman un ángulo obtuso (consulte la figura 6-22*b*); en caso contrario, este círculo puede sustituirse por el círculo cuyo diámetro es el lado más grande de este triángulo (consulte la figura 6-22*c*). En nuestro caso, debido a que el arco que genera todos los puntos más alejados es de grado mayor que o igual a 180°, debe haber por lo menos tres de estos puntos más alejados que además no forman un triángulo obtuso. *En otras*

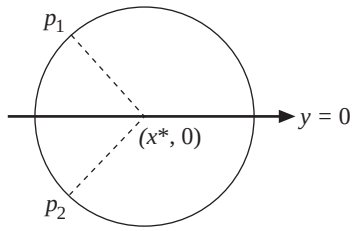
palabras, el círculo más pequeño actual ya es óptimo. En consecuencia, puede concluirse que $y_s = 0$.

FIGURA 6-22 Dos o tres puntos que definen el menor círculo que cubre todos los puntos.



Para el segundo caso, el arco que genera todos los puntos más alejados es de grado menor que 180^o*, primero se demostrará que las abscisas de los puntos extremos* p_1 y p_2 *deben tener signos opuestos*. Suponga lo contrario, como se muestra en la figura 6-23. Entonces es posible desplazar x^* hacia la dirección en que se encuentran p_1 y p_2. Esto es imposible porque x^* es el centro del círculo óptimo sobre $y = 0$.

FIGURA 6-23 Dirección de x^* donde el grado es menor que 180^o.



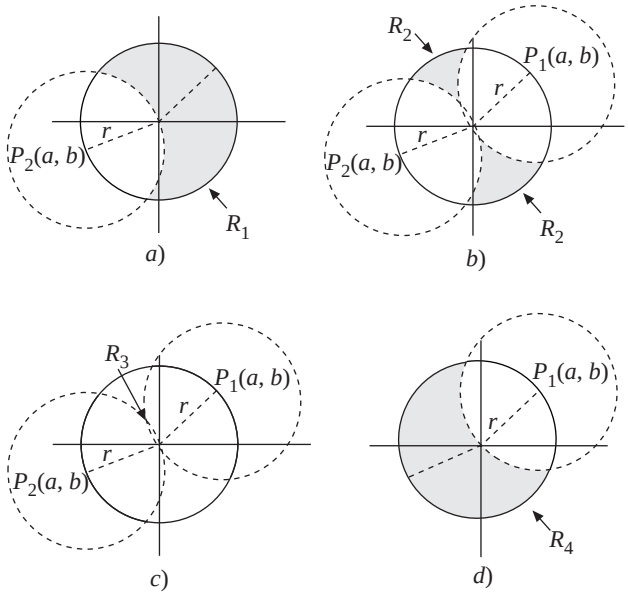
Sean $p_1 = (a, b)$ y $p_2 = (c, d)$. Por lo tanto, puede suponerse que

$$a > x^*,\ b > 0$$

y $$c < x^*,\ d < 0$$

como se muestra en la figura 6-24. En esta figura hay tres círculos centrados en $(x^*, 0)$, (a, b) y (c, d), respectivamente. Estos tres círculos forman cuatro regiones: R_1, R_2, R_3 y R_4. Sea r el radio del círculo actual. Se indican las tres observaciones siguientes:

FIGURA 6-24 Dirección de x^* donde el grado es mayor que 180°.



1. En R_2, la distancia entre cualquier punto x en esta región y p_1, o p_2, es mayor que r. En consecuencia, el centro del círculo óptimo no puede estar en R_2.
2. En R_1, la distancia entre cualquier punto x en esta región y p_2 es mayor que r. Por lo tanto, el centro del círculo óptimo no puede estar en R_1.
3. Con un razonamiento semejante, es fácil ver que el centro del círculo óptimo no puede estar en R_4.

Con base en las tres observaciones anteriores, se concluye que el centro óptimo debe estar en la región R_3. En consecuencia, el signo de y_s debe ser el signo de $(b + d)/2 = (y_1 + y_2)/2$.

El procedimiento anterior puede resumirse de la siguiente manera.

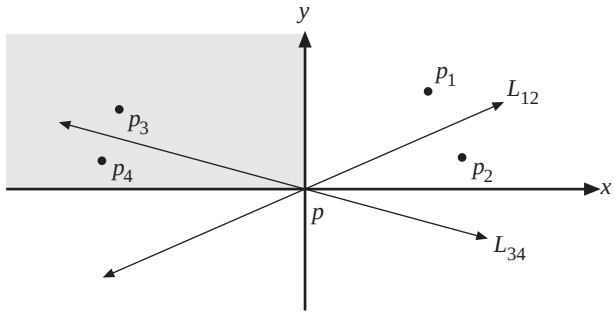
Procedimiento 6-2 □ Procedimiento usado en el algoritmo 6-5

Input: Un conjunto S de puntos.
Una recta $y = y^*$.
(x^*, y^*): las soluciones del problema con un centro restringido para S dando por resultado que la solución está en la recta $y = y^*$.

Output: Ya sea que $y_s > y^*$, $y_s < y^*$ o $y_s = y^*$, donde (x_s, y_s) es la solución óptima del problema con un centro para S.
Paso 1. Encontrar I, que es el conjunto de puntos más alejados de (x^*, y^*).
Paso 2. **Caso 1:** I contiene sólo un punto $p = (x_p, y_p)$.
Si $y_p > y^*$, reportar "$y_s > y^*$" y salir.
Si $y_p < y^*$, reportar "$y_s < y^*$" y salir.
Caso 2: I contiene más de un punto. En I, buscar $p_1 = (x_1, y_1)$ y $p_2 = (x_2, y_2)$, que son los dos puntos extremos que forman el menor arco que genera todos los puntos en I.
Caso 2.1: El grado del arco formado por p_1 y p_2 es mayor o igual a 180°.
Reportar $y_s = y^*$ y salir.
Caso 2.2: El grado del arco formado por p_1 y p_2 es menor que 180°.
Hacer $y_c = (y_1 + y_2)/2$.
Si $y_c > y^*$, reportar $y_s > y^*$ y salir.
Si $y_c < y^*$, reportar $y_s < y^*$ y salir.

El algoritmo con un centro restringido puede usarse ahora para podar puntos. Antes de proporcionar el algoritmo preciso, se considerará la figura 6-25. Para los pares (p_1, p_2) y (p_3, p_4) se trazan las bisectrices L_{12} y L_{34}, de los segmentos de recta $\overline{p_1p_2}$ y $\overline{p_3p_4}$, respectivamente. Sea que L_{12} y L_{34} se cortan en un punto; por ejemplo, p. El eje x se rota de modo que la pendiente de L_{34} sea negativa y la pendiente de L_{12} sea positiva. El origen del sistema de coordenadas se desplaza a p. Primero se aplica el algoritmo con un centro restringido requiriendo que el centro esté ubicado en $y = 0$. Después que

FIGURA 6-25 Rotación idónea de las coordenadas.



se encuentra este centro restringido, el procedimiento 6-2 se aplica después para descubrir que para las soluciones óptimas, donde está implicada la dirección y, es necesario moverse hacia arriba. Este proceso se repite aplicando nuevamente el algoritmo con un centro restringido a $x = 0$. Luego se descubre que donde está implicada la dirección x, es necesario moverse hacia la izquierda. En consecuencia, la ubicación óptima debe encontrarse en la región sombreada, que se ilustra en la figura 6-25. Debido a que hay una bisectriz que no corta la región sombreada, esta información puede usarse siempre para eliminar un punto de la consideración.

En nuestro caso, L_{12} no corta la región sombreada. El punto p_1 está del mismo lado que la región sombreada, lo que significa que en tanto el centro óptimo esté restringido a la región sombreada, el punto p_1 siempre está más cerca del centro que p_2. En consecuencia, es posible eliminar el punto p_1 porque es "dominado" por el punto p_2. En otras palabras, sólo debe considerarse el punto p_2.

A continuación se presenta el algoritmo prune-and-search para resolver el problema con un centro.

Algoritmo 6-5 □ Un algoritmo prune-and-search para resolver el problema con un centro

Input: Un conjunto $S = \{p_1, p_2, \ldots, p_n\}$ de n puntos.

Output: El círculo más pequeño que abarca a S.

Paso 1. Si S no contiene más de 16 puntos, resolver el problema con un método de fuerza bruta.

Paso 2. Formar pares de puntos ajenos, (p_1, p_2), (p_3, p_4), … , (p_{n-1}, p_n). Para cada par de puntos (p_i, p_{i+1}), buscar las bisectrices perpendiculares del segmento de recta $\overline{p_i p_{i+1}}$. Nombrar como $L_{i/2}$ tales bisectrices perpendiculares, para $i = 2, 4, \ldots, n$ y calcular sus pendientes. Sea s_k la pendiente de L_k, para $k = 1, 2, \ldots, n/2$.

Paso 3. Calcular la mediana de las s_k y denotar por s_m.

Paso 4. Rotar el sistema de coordenadas de modo que el eje x coincida con $y = s_m x$. Sea I^+ (I^-) el conjunto de L_k con pendientes positivas (negativas). (El tamaño de cada uno es $n/4$.)

Paso 5. Construir pares ajenos de las rectas (L_{i+}, L_{i-}) para $i = 1, 2, \ldots, n/4$, donde $L_{i+} \in I^+$ y $L_{i-} \in I^-$. Buscar la intersección de cada par y denotar por (a_i, b_i) para $i = 1, 2, \ldots, n/4$.

Paso 6. Buscar la mediana de las b_i y denotar por y^*. Utilizar la subrutina de un centro restringido a S, requiriendo que el centro del círculo esté en $y = y^*$. Hacer (x', y^*) la solución de este problema con un centro restringido.

Paso 7. Utilizar el procedimiento 6-2, usando como parámetros a S y (x', y^*). Si $y_s = y^*$, reportar "el círculo con centro en (x', y^*) encontrado en el paso 6 es la solución óptima" y salir.

De otra forma, reporte $y_s > y^*$ o $y_s < y^*$.

Paso 8. Buscar la mediana de las a_i y denote por x^*. Utilizar el algoritmo con un centro restringido a S, requiriendo que el centro del círculo esté en $x = x^*$. Sea (x^*, y') la solución de este problema con 1 centro restringido.

Paso 9. Utilizar el procedimiento 6-2, usando como parámetros a S y (x^*, y'). Si $x_s = x^*$, reportar "el círculo con centro en (x^*, y') encontrado en el paso 6 es la solución óptima" y salir.

De otra forma, reporte $x_s > x^*$ o $x_s < x^*$.

Paso 10. Caso 1: $x_s > x^*$ y $y_s > y^*$.

Buscar todas las (a_i, b_i) tales que $a_i < x^*$ y $b_i < y^*$. Hacer (a_i, b_i) la intersección de L_{i+} y L_{i-} (consulte el paso 5). Sea L_{i-} la bisectriz de p_j y p_k. Si p_j (p_k) está más cerca de (x^*, y^*) que p_k (p_j), podar p_j (p_k).

Caso 2: $x_s < x^*$ y $y_s > y^*$.

Buscar todas las (a_i, b_i) tales que $a_i > x^*$ y $b_i < y^*$. Hacer (a_i, b_i) la intersección de L_{i+} y L_{i-}. Sea L_{i+} la bisectriz de p_j y p_k. Si p_j (p_k) está más cerca de (x^*, y^*) que p_k (p_j), podar p_j (p_k).

Caso 3: $x_s < x^*$ y $y_s < y^*$.

Buscar todas las (a_i, b_i) tales que $a_i > x^*$ y $b_i > y^*$. Hacer (a_i, b_i) la intersección de L_{i+} y L_{i-}. Hacer L_{i-} la bisectriz de p_j y p_k. Si p_j (p_k) está más cerca de (x^*, y^*) que p_k (p_j), podar p_j (p_k).

Caso 4: $x_s > x^*$ y $y_s < y^*$.

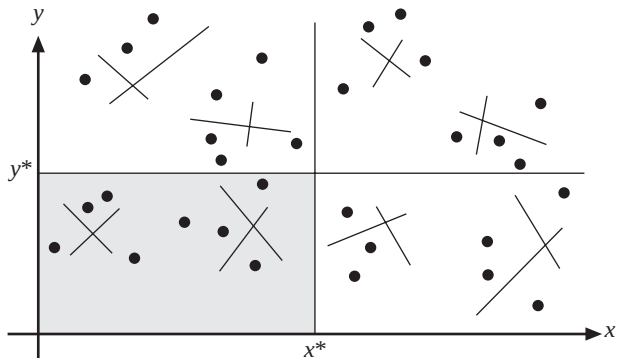
Buscar todas las (a_i, b_i) tales que $a_i < x^*$ y $b_i > y^*$. Hacer (a_i, b_i) la intersección de L_{i+} y L_{i-}. Sea L_{i+} la bisectriz de p_j y p_k. Si p_j (p_k) está más cerca de (x^*, y^*) que p_k (p_j), podar p_j (p_k).

Paso 11. Hacer S el conjunto integrado por los puntos restantes. Ir a paso 1.

El análisis del algoritmo 6-5 es como sigue. Primero, se supone que hay $n = (16)^k$ puntos para alguna k. En el paso 2 se formaron $n/2$ bisectrices. Después del paso 4, $n/4$ bisectrices tienen pendiente positiva y $n/4$, negativa. Así, en total hay $n/4$ intersecciones formadas en el paso 5. Debido a que x^* (y^*) es la mediana de las a_i (b_i), hay $(n/4)/4 = n/16$ (a_i, b_i) para cada caso en el paso 10, lo cual se muestra en la figura 6-26, donde

se supone que la solución óptima está en la región sombreada. Entonces, para cada par de intersecciones en la región donde $x > x^*$ y $y > y^*$, es posible podar el punto que está abajo de la recta con pendiente negativa. Para cada una de tales (a_i, b_i), se poda exactamente un punto. Por consiguiente, en cada intersección se podan $n/16$ puntos.

FIGURA 6-26 Los pares de puntos ajenos y sus pendientes.



Resulta fácil darse cuenta de que cada paso del algoritmo 6-5 requiere tiempo $O(n)$. Así, la complejidad temporal total del algoritmo 6-5 es $O(n)$.

6-5 LOS RESULTADOS EXPERIMENTALES

Para demostrar la eficacia del método prune-and-search se implementó el algoritmo prune-and-search para resolver el problema con un centro. Hay otra forma de resolver este problema: observe que toda solución de él debe pasar por lo menos por tres puntos. En consecuencia, puede intentarse cualquier combinación posible de éstos. Es decir, si se seleccionan tres puntos, se construye un círculo y se observa si los cubre. De no hacerlo, se ignora. En caso afirmativo, se registra su radio. Una vez que se han examinado todos los círculos así, es posible encontrar el más pequeño. Así, éste es un método $O(n^3)$, mientras el método prune-and-search es un algoritmo $O(n)$.

Ambos métodos se implementaron en una IBM PC. El método directo se identificó como la búsqueda exhaustiva. Todos los datos fueron generados aleatoriamente. Los resultados de prueba se presentan en la tabla 6-1.

TABLA 6-1 Resultados de prueba.

Número de puntos en la muestra	Búsqueda exhaustiva (segundos)	Prune-and-search (segundos)
17	1	1
30	8	11
50	28	32
100	296	83
150	1 080	130
200	2 408	185
250	2 713	232
300	–	257

Resulta evidente que la estrategia prune-and-search es muy eficiente en este caso. Cuando n es 150, la búsqueda exhaustiva ya es mucho peor que el método prune-and-search. Cuando n es 300, la búsqueda exhaustiva requiere tanto tiempo para resolver el problema que éste prácticamente jamás se resuelve aplicando este método. Por otra parte, si se aplica el algoritmo prune-and-search, para resolver el problema con un centro se requieren 257 segundos. Los resultados no son sorprendentes, ya que la complejidad de un método es $O(n^3)$, mientras la del otro es $O(n)$.

6-6 Notas y referencias

El método para resolver el problema de selección ha sido clasificado tradicionalmente como una estrategia divide y vencerás. Consideramos que es más correcto clasificarlo como un método prune-and-search. Para un análisis más detallado, consulte las obras de Blum, Floyd, Pratt, Rivest y Tarjan (1972) y la de Floyd y Rivest (1975), con una exposición más detallada. En Hyafil (1976) puede encontrarse un análisis de la cota inferior para el problema de selección. En muchos textos también aparece el método ingenioso para el problema de selección (Brassard y Bratley, 1988; Horowitz y Sahni, 1978; Kronsjo, 1987).

El ingenioso método presentado en el apartado 6-3 fue descubierto de manera independiente por Dyer (1984) y Megiddo (1983). Megiddo (1984) extendió los resultados a dimensiones superiores. El hecho de que el problema con un centro puede resolverse con el método prune-and-search también fue demostrado por Megiddo (1983). Así se refutó la conjetura de Shamos y Hoey (1975).

6-7 BIBLIOGRAFÍA ADICIONAL

El método prune-and-search es relativamente nuevo, por lo que en comparación con otros campos en esta área hay menos artículos publicados. Para investigaciones adicionales recomendamos los siguientes artículos: Avis, Bose, Shermer, Snoeyink, Toussaint y Zhu (1996); Bhattacharya, Jadhav, Mukhopadhyay y Robert (1994); Chou y Chung (1994); Imai (1993); Imai, Kato y Yamamoto (1989); Imai, Lee y Yang (1992); Jadhav y Mukhopadhyyay (1993); Megiddo (1983); Megiddo (1984); Megiddo (1985); y Shreesh, Asish y Binay (1996).

Para conocer algunos artículos muy interesantes de reciente aparición, consulte Eiter y Veith (2002); ElGindy, Everett y Toussaint (1993); Kirpatrick y Snoeyink (1993); Kirpatrick y Snoeyink (1995), y Santis y Persiano (1994).

Ejercicios

6.1 Asegúrese de comprender las diferencias y semejanzas entre divide y vencerás y prune-and-search. Observe que las fórmulas de recurrencia pueden parecerse bastante.

6.2 Escriba un programa para implementar el algoritmo de selección que se presentó en este capítulo. Otro método para encontrar el *k*-ésimo número más grande es ejecutar el quick sort y seleccionar el *k*-ésimo número más grande del arreglo ordenado. También implemente este método. Compare los dos programas. Explique sus resultados de prueba.

6.3 Se dice que dos conjuntos de puntos A y B en R^d son linealmente separables si existe un hiperplano de dimensión $(d - 1)$ tal que A y B están en lados opuestos de éste. Demuestre que el problema de separabilidad lineal es un problema de programación lineal.

6.4 Con base en los resultados del ejercicio 6.3, demuestre que el problema de separabilidad lineal puede resolverse en tiempo $O(n)$ en dos y tres dimensiones.

6.5 Lea el teorema 3.3 de la obra de Horowitz y Sahni (1978) para el análisis del caso promedio del algoritmo de selección basado en el método prune-and-search.

capítulo

7

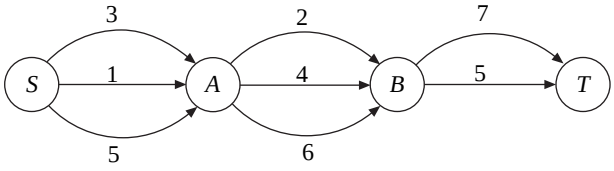
Programación dinámica

La estrategia de programación dinámica es una técnica muy útil para resolver muchos problemas de optimización combinatorios. Antes de presentar el método de programación dinámica, quizá convenga reconsiderar un caso típico donde puede aplicarse el método codicioso (*greedy*). Luego se mostrará un caso para el que el método codicioso no funciona, mientras que el método de programación dinámica sí lo hace.

Se considerará la figura 7-1, que muestra una gráfica de multietapas. Suponga que se quiere encontrar la ruta más corta de *S* a *T*. En este caso, la ruta más corta puede encontrarse mediante la siguiente línea de razonamiento:

1. Debido a que se sabe que la ruta más corta debe pasar por el vértice *A*, es necesario encontrar una ruta más corta de *S* a *A*. El costo de esta ruta es 1.
2. Debido a que se sabe que la ruta más corta debe pasar por *B*, se encuentra una ruta más corta de *A* a *B*. El costo de esta ruta es 2.
3. De manera semejante, se encuentra una ruta más corta de *B* a *T* cuyo costo es 5.

FIGURA 7-1 Un caso en que funciona el método codicioso.



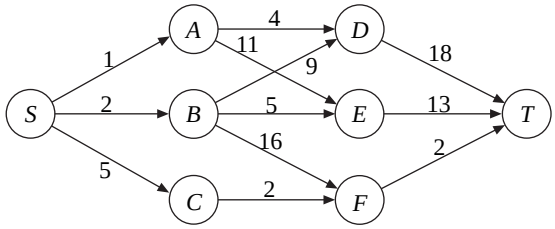
Así, el costo total de la ruta más corta de *S* a *T* es $1 + 2 + 5 = 8$. Además, este problema de ruta más corta en realidad se resolvió aplicando un método codicioso.

¿Por qué es posible usar el método codicioso para resolver ese problema? Esto es posible porque definitivamente se sabe que la solución debe constar de una subruta de *S* a *A*, de una subruta de *A* a *B*, etc. En consecuencia, nuestra estrategia para resolver el

problema primero encuentra una subruta más corta de *S* a *A*, luego una subruta más corta de *A* a *B* y así sucesivamente.

A continuación se presentará un caso en que no funciona el método codicioso. Considere la figura 7-2. De nuevo, se quiere encontrar una ruta más corta de *S* a *T*. Sin embargo, esta vez no se sabe por cuál de los vértices *A*, *B* y *C* debe pasar la ruta más corta. Si para resolver este problema se usa el método codicioso, se escoge el vértice *A* porque el costo asociado con el borde de *S* a *A* es el menor. Después de que se escoge *A*, se elige *D*. El costo de esta ruta es $1 + 4 + 18 = 23$. Ésta no es la ruta más corta porque la ruta más corta es $S \rightarrow C \rightarrow F \rightarrow T$, cuyo costo total es $5 + 2 + 2 = 9$.

FIGURA 7-2 Un caso en que no funciona el método codicioso.

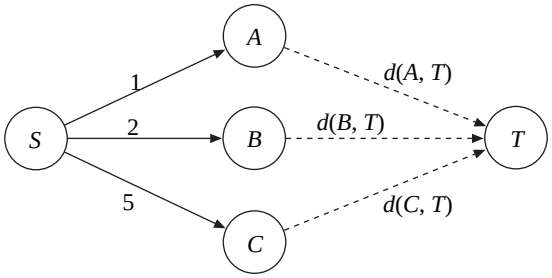


$$S \xrightarrow{5} C \xrightarrow{2} F \xrightarrow{2} T$$

Como ya se indicó, debe seleccionarse *C*, en vez de *A*. El método de programación dinámica produce esta solución mediante la siguiente línea de razonamiento.

1. Se sabe que es necesario empezar en *S* y pasar por *A*, *B* o *C*. Esto se ilustra en la figura 7-3.

FIGURA 7-3 Un paso en el proceso del método de programación dinámica.



En consecuencia, la longitud de la ruta más corta de S a T se determina con la fórmula siguiente:

$$d(S, T) = \text{mín}\{1 + d(A, T), 2 + d(B, T), 5 + d(C, T)\} \quad \textbf{(7.1)}$$

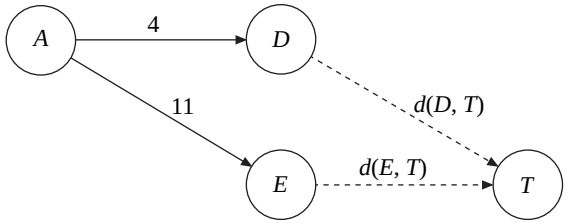
donde $d(X, T)$ denota la longitud de la ruta más corta de X a T. La ruta más corta de S a T se conoce tan pronto como se encuentran las rutas más cortas de A, B y C a T, respectivamente.

2. Para encontrar las rutas más cortas de A, B y C a T, de nuevo puede usarse el método anterior. Para el vértice A, se tiene una subgráfica que se muestra en la figura 7-4. Es decir, $d(A, T) = \text{mín}\{4 + d(D, T), 11 + d(E, T)\}$. Debido a que $d(D, T) = 18$ y $d(E, T)$ 13, se tiene

$$\begin{aligned} d(A, T) &= \text{mín}\{4 + 18, 11 + 13\} \\ &= \text{mín}\{22, 24\} \\ &= 22. \end{aligned}$$

Por supuesto, es necesario registrar el hecho de que esta ruta más corta va de A a D y luego a T.

FIGURA 7-4 Un paso en el proceso del método de programación dinámica.

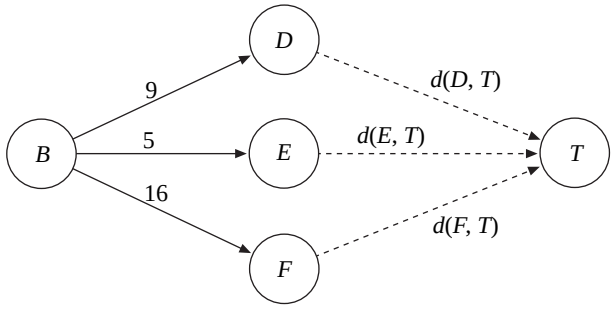


De manera semejante, para B, el paso se muestra en la figura 7-5.

$$\begin{aligned} d(B, T) &= \text{mín}\{9 + d(D, T), 5 + d(E, T), 16 + d(F, T)\} \\ &= \text{mín}\{9 + 18, 5 + 13, 16 + 2\} \\ &= \text{mín}\{27, 18, 18\} \\ &= 18. \end{aligned}$$

Por último, fácilmente se encuentra que $d(C, T)$ es 4.

FIGURA 7-5 Un paso en el proceso del método de programación dinámica.



3. Una vez que se han encontrado $d(A, T)$, $d(B, T)$ y $d(C, T)$ es posible encontrar $d(S, T)$ usando la fórmula (7.1).

$$\begin{aligned} d(S, T) &= \text{mín}\{1 + 22, 2 + 18, 5 + 4\} \\ &= \text{mín}\{23, 20, 9\} \\ &= 9. \end{aligned}$$

El principio fundamental de la programación dinámica consiste en descomponer un problema en subproblemas, cada uno de los cuales se resuelve aplicando de manera recurrente el mismo método. El método puede resumirse como sigue:

1. Para encontrar la ruta más corta de *S* a *T*, se encuentran las rutas más cortas de *A*, *B* y *C* a *T*.
2. *a*) Para encontrar la ruta más corta de *A* a *T*, se encuentran las rutas más cortas de *D* y *E* a *T*.
   b) Para encontrar la ruta más corta de *B* a *T*, se encuentran las rutas más cortas de *D*, *E* y *F* a *T*.
   c) Para encontrar la ruta más corta de *C* a *T*, se encuentra la ruta más corta de *F* a *T*.
3. Las rutas más cortas de *D*, *E* y *F* a *T* pueden encontrarse fácilmente.
4. Una vez que se han encontrado las rutas más cortas de *D*, *E* y *F* a *T*, es posible encontrar las rutas más cortas de *A*, *B* y *C* a *T*.
5. Una vez que se han encontrado las rutas más cortas de *A*, *B* y *C* a *T*, es posible encontrar la ruta más corta de *S* a *T*.

La línea de razonamiento anterior es realmente un *razonamiento hacia atrás* (*backward reasoning*). A continuación se mostrará que el problema también puede resolverse razonando hacia atrás. La ruta más corta se encontrará como sigue:

1. Primero se encuentran $d(S, A)$, $d(S, B)$ y $d(S, C)$.

$$\begin{aligned} d(S, A) &= 1 \\ d(S, B) &= 2 \\ d(S, C) &= 5. \end{aligned}$$

2. Luego se determinan $d(S, D)$, $d(S, E)$ y $d(S, F)$ como sigue:

$$\begin{aligned} d(S, D) &= \text{mín}\{d(A, D) + d(S, A), d(B, D) + d(S, B)\} \\ &= \text{mín}\{4 + 1, 9 + 2\} \\ &= \text{mín}\{5, 11\} \\ &= 5 \\ d(S, E) &= \text{mín}\{d(A, E) + d(S, A), d(B, E) + d(S, B)\} \\ &= \text{mín}\{11 + 1, 5 + 2\} \\ &= \text{mín}\{12, 7\} \\ &= 7 \\ d(S, F) &= \text{mín}\{d(B, F) + d(S, B), d(C, F) + d(S, C)\} \\ &= \text{mín}\{16 + 2, 2 + 5\} \\ &= \text{mín}\{18, 7\} \\ &= 7. \end{aligned}$$

3. Ahora es posible determinar la distancia más corta de S a T como sigue:

$$\begin{aligned} d(S, T) &= \text{mín}\{d(D, T) + d(S, D), d(E, T) + d(S, E), d(F, T) + d(S, F)\} \\ &= \text{mín}\{18 + 5, 13 + 7, 2 + 7\} \\ &= \text{mín}\{23, 20, 9\} \\ &= 9. \end{aligned}$$

El hecho de que el método de programación dinámica ahorra cálculos se explica al considerar nuevamente la figura 7-2. En esta figura, hay una solución como sigue:

$S \rightarrow B \rightarrow D \rightarrow T$.

La longitud de esta solución; a saber,

$d(S, B) + d(B, D) + d(D, T)$

jamás se calcula si se aplica el método de programación dinámica con razonamiento hacia atrás. Al aplicar el método con razonamiento hacia atrás, se encuentra

$$d(B, E) + d(E, T) < d(B, D) + d(D, T).$$

Es decir, no es necesario considerar la solución:

$$S \rightarrow B \rightarrow D \rightarrow T$$

porque se sabe que la longitud de

$$B \rightarrow E \rightarrow T$$

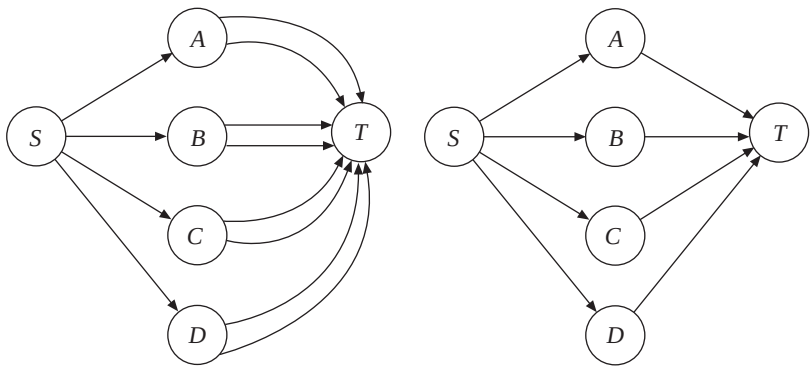
es menor que la longitud de

$$B \rightarrow D \rightarrow T.$$

Así, el método de programación dinámica, como el método *branch-and-bound*, ayuda a evitar una búsqueda exhaustiva en el espacio solución.

Puede afirmarse que el método de programación dinámica es un método de eliminación por etapas porque después de que se considera una etapa, se eliminan muchas subsoluciones. Por ejemplo, considere la figura 7-6*a*). Originalmente hay ocho soluciones. Si se usa el método de programación dinámica, el número de soluciones se reduce a cuatro, que se muestran en la figura 7-6*b*).

FIGURA 7-6 Ejemplo que ilustra la eliminación de la solución en el método de programación dinámica.



a) *b*)

La programación dinámica se basa en un concepto denominado principio de optimalidad. Suponga que al resolver un problema se requiere hacer una serie de decisiones $D_1, D_2, \ldots, D_n$. *Si esta serie es óptima, entonces las k últimas decisiones,* $1 \leq k \leq n$, *deben ser óptimas*. Este principio se ilustrará varias veces en las siguientes secciones.

Al aplicar el método de programación dinámica se tienen dos ventajas. La primera, como ya se explicó, es que pueden eliminarse soluciones y también ahorrarse cálculos. La otra ventaja de la programación dinámica es que ayuda a resolver el problema etapa por etapa de manera sistemática.

Si alguien que resuelve un problema no conoce en absoluto la programación dinámica, quizá tenga que resolver el problema examinando todas las posibles soluciones combinatorias, lo cual puede consumir tiempo excesivamente. Si se aplica el método de programación dinámica, el problema puede repentinamente convertirse en un problema de multietapas, con lo cual puede resolverse de manera muy sistemática. Esto se clarificará después.

7-1 EL PROBLEMA DE ASIGNACIÓN DE RECURSOS

Este problema se define como sigue: Se tienen m recursos y n proyectos. Se obtiene una ganancia $P(i, j)$ si al proyecto i se asignan j, $0 \leq j \leq m$ recursos. El problema consiste en encontrar una asignación de recursos que maximice la ganancia total.

Suponga que hay cuatro proyectos y tres recursos, y que la matriz de ganancias P es la que se muestra en la tabla 7-1 con $P(i, 0) = 0$ para toda i.

TABLA 7-1 Matriz de ganancias.

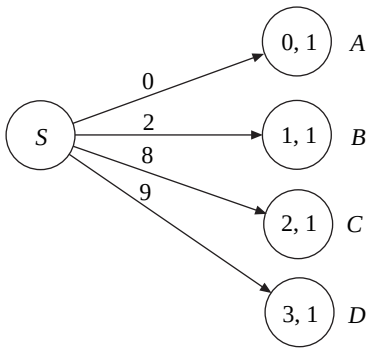
Recurso / Proyecto	1	2	3
1	2	8	9
2	5	6	7
3	4	4	4
4	2	4	5

Para resolver este problema se toma una serie de decisiones. En cada decisión, se determina el número de recursos a asignar al proyecto i. Es decir que, sin pérdida de generalidad, es necesario decidir lo siguiente:

1. ¿Cuántos recursos deben asignarse al proyecto 1?
2. ¿Cuántos recursos deben asignarse al proyecto 2?
   ⋮

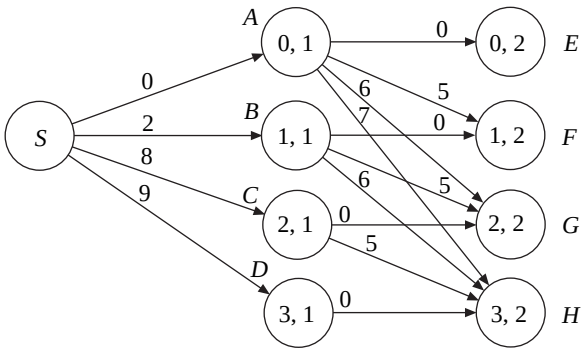
Resulta evidente que el enfoque del método codicioso no funciona en este caso. Sin embargo, es posible resolver el problema usando el método de programación dinámica. Si se aplica este método, es posible suponer que cada decisión origina un nuevo estado. Sea (i, j) el estado alcanzado donde se han asignado i recursos a los proyectos 1, 2, …, j. Así, inicialmente, sólo se tienen cuatro estados descritos, que se muestran en la figura 7-7.

FIGURA 7-7 Decisiones de la primera etapa de un problema de asignación de recursos.



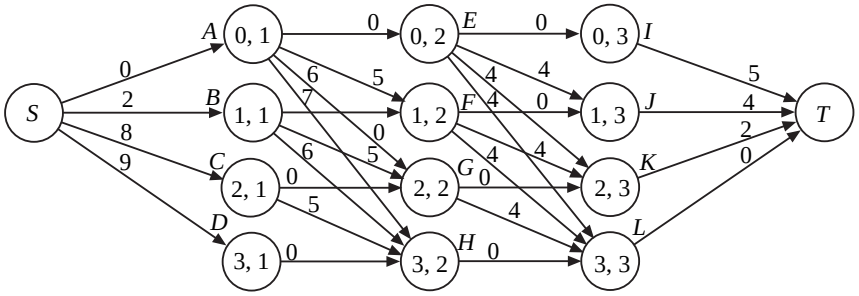
Después de haber asignado i recursos al proyecto 1, cuando mucho es posible asignar $(3 - i)$ recursos al proyecto 2. Así, las decisiones de la segunda etapa están relacionadas con las decisiones de la primera etapa, que se muestran en la figura 7-8.

FIGURA 7-8 Decisiones de las dos primeras etapas de un problema de asignación de recursos.



Finalmente, en la figura 7-9 se muestra cómo es posible describir todo el problema.

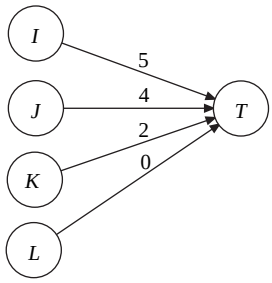
FIGURA 7-9 El problema de asignación de recursos descrito como una gráfica multietapas.



Para resolver el problema de asignación de recursos, basta encontrar la ruta más larga de *S* a *T*. El método de razonamiento hacia atrás puede aplicarse como sigue:

1. Las rutas más largas de *I*, *J*, *K* y *L* a *T* se muestran en la figura 7-10.

FIGURA 7-10 Las rutas más largas de *I*, *J*, *K* y *L* a *T*.

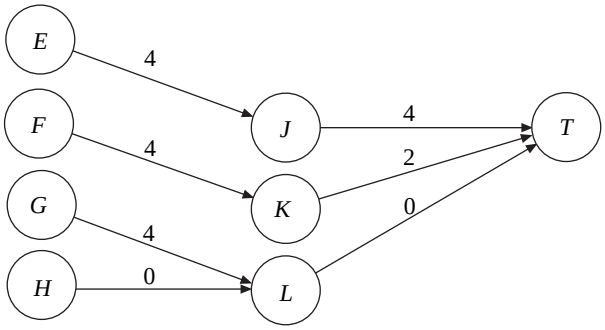


2. Una vez que se han obtenido las rutas más largas de *I*, *J*, *K* y *L* a *T*, las rutas más largas de *E*, *F*, *G* y *H* a *T* pueden obtenerse fácilmente. Por ejemplo, la ruta más larga de *E* a *T* se determina como sigue:

$$\begin{aligned} d(E, T) &= \text{máx}\{d(E, I) + d(I, T), d(E, J) + d(J, T), \\ &\qquad d(E, K) + d(K, T), d(E, L) + d(L, T)\} \\ &= \text{máx}\{0 + 5, 4 + 4, 4 + 2, 4 + 0\} \\ &= \text{máx}\{5, 8, 6, 4\} \\ &= 8. \end{aligned}$$

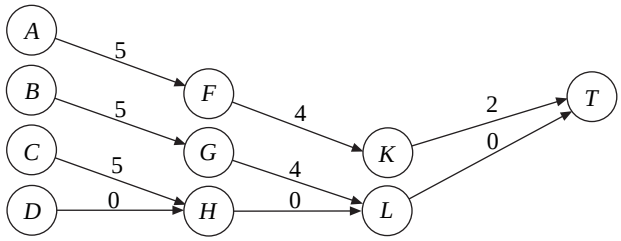
Los resultados se resumen en la figura 7-11.

FIGURA 7-11 Las rutas más largas de *E*, *F*, *G* y *H* a *T*.



3. Las rutas más largas de *A*, *B*, *C* y *D* a *T*, respectivamente, pueden obtenerse fácilmente aplicando el mismo método y se muestran en la figura 7-12.

FIGURA 7-12 Las rutas más largas de *A*, *B*, *C* y *D* a *T*.



4. Finalmente, la ruta más larga de *S* a *T* se obtiene como sigue:

$$\begin{aligned} d(S, T) &= \text{máx}\{d(S, A) + d(A, T), d(S, B) + d(B, T), \\ &\qquad d(S, C) + d(C, T), d(S, D) + d(D, T)\} \\ &= \text{máx}\{0 + 11, 2 + 9, 8 + 5, 9 + 0\} \\ &= \text{máx}\{11, 11, 13, 9\} \\ &= 13. \end{aligned}$$

La ruta más larga es

$$S \to C \to H \to L \to T.$$

Lo anterior corresponde a

2 recursos asignados al proyecto 1,
1 recurso asignado al proyecto 2,
0 recursos asignados al proyecto 3,
y 0 recursos asignados al proyecto 4,

7-2 EL PROBLEMA DE LA SUBSECUENCIA COMÚN MÁS LARGA

Considere una cadena $A = a\ b\ a\ a\ d\ e$. Una subsecuencia A (también llamada subcadena) se obtiene eliminando 0 o más símbolos (no necesariamente consecutivos) de A. Por ejemplo, todas las siguientes cadenas son subsecuencias de *A:*

a
b
d
a b
a a
b d
a e
a b a
a a a
a d e
b a d e
a a a d
a b a d e

Una subsecuencia común entre A y B se define como una subsecuencia de ambas cadenas. Por ejemplo, considere $A = a\ b\ a\ a\ d\ e\ c$ y $B = c\ a\ a\ c\ e\ d\ c$. Todas las siguientes cadenas son subsecuencias comunes de A y B*:*

a
d
c
a d
d c
a a
a c
a a c
a a d
a a e c

El problema de la subsecuencia común más larga consiste en encontrar una subsecuencia común más larga entre dos cadenas. Muchos problemas son variantes del problema de la subsecuencia común más larga. Por ejemplo, el problema de identificación de lenguaje puede considerarse como un problema de la subsecuencia común más larga.

Para un novato en la solución de problemas, el problema de la subsecuencia común más larga de ninguna manera es fácil de resolver. Un método directo para resolverlo consiste en encontrar todas las subsecuencias comunes mediante una búsqueda exhaustiva. Por supuesto, debe empezarse con la más larga posible. La longitud de la subsecuencia común más larga entre dos cadenas no puede ser mayor que la longitud de la cadena más corta. En consecuencia, debe empezarse con la cadena más corta.

Por ejemplo, sean $A = a\ b\ a\ a\ d\ e\ c$ y $B = b\ a\ f\ c$. Puede intentar determinarse si $b\ a\ f\ c$ es una subsecuencia común. Puede verse que no lo es. Luego se intenta con subsecuencias escogidas de B cuya longitud es tres; a saber, $a\ f\ c$, $b\ f\ c$, $b\ a\ c$ y $b\ a\ f$. Debido a que $b\ a\ c$ es una subsecuencia común, también debe ser una subsecuencia común más larga, ya que se empezó con la más larga posible.

Este método directo consume demasiado tiempo porque un gran número de subsecuencias de B deben corresponder con un gran número de subsecuencias de A. Así, se trata de un procedimiento exponencial.

Afortunadamente, para resolver este problema de la subsecuencia común más larga puede aplicarse el método de programación dinámica. A continuación se modificará ligeramente el problema original. En vez de encontrar la subsecuencia común más larga, se intentará encontrar la *longitud* de la subsecuencia común más larga. En realidad, al rastrear el procedimiento para encontrar la longitud de la subsecuencia común más larga, fácilmente puede encontrarse la subsecuencia común más larga.

Considere dos cadenas $A = a_1a_2 \ldots a_m$ y $B = b_1b_2 \ldots b_n$. Se prestará atención a los dos últimos símbolos: a_m y b_n. Hay dos posibilidades:

Caso 1: $a_m = b_n$. En este caso, la subsecuencia común más larga debe contener a a_m. Simplemente basta encontrar la subsecuencia común más larga de $a_1a_2 \ldots a_{m-1}$ y $b_1b_2 \ldots b_{n-1}$.

Caso 2: $a_m \neq b_n$. En este caso, puede hacerse corresponder $a_1a_2 \ldots a_m$ con $b_1b_2 \ldots b_{n-1}$ y también $a_1a_2 \ldots a_{m-1}$ con $b_1b_2 \ldots b_n$. Con lo se obtiene una subsecuencia común mayor, ésta será la subsecuencia común más larga.

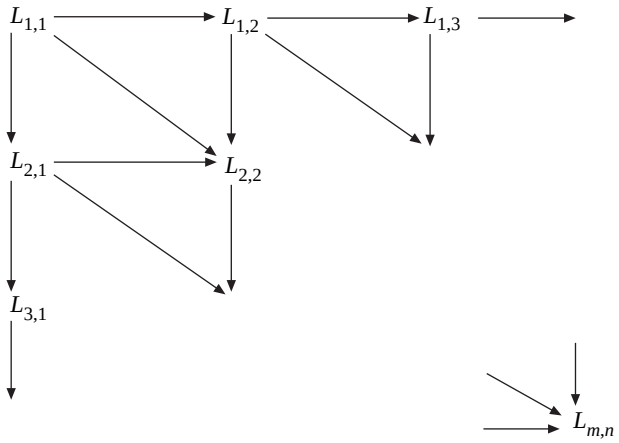
Sea $L_{i,j}$ la longitud de la subsecuencia común más larga de $a_1a_2 \ldots a_i$ con $b_1b_2 \ldots b_j$. $L_{i,j}$ puede encontrarse aplicando la siguiente fórmula recurrente:

$$L_{i,j} = \begin{cases} L_{i-1,j-1} + 1 & \text{si } a_i = b_j \\ \max\{L_{i-1,j}, L_{i,j-1}\} & \text{si } a_i \neq b_j \end{cases}$$

$L_{0,0} = L_{0,j} = L_{i,0} = 0$, para $1 \leq i \leq m$ y $1 \leq j \leq n$.

La fórmula anterior muestra que primero debe encontrarse $L_{1,1}$. Después de encontrar $L_{1,1}$ es posible encontrar $L_{1,2}$ y $L_{2,1}$. Una vez que se obtienen $L_{1,1}$, $L_{1,2}$ y $L_{2,1}$, pueden encontrarse $L_{1,3}$, $L_{2,2}$ y $L_{3,1}$. Todo el procedimiento puede representarse en la figura 7-13.

FIGURA 7-13 El método de programación dinámica para resolver el problema de la subsecuencia común más larga.



Para implementar de manera secuencial el algoritmo es posible calcular el valor de cada $L_{i,j}$ mediante la secuencia: $L_{1,1}$, $L_{1,2}$, $L_{1,3}$, …, $L_{1,n}$, $L_{2,1}$, …, $L_{2,n}$, $L_{3,1}$, …, $L_{3,n}$, …, $L_{m,n}$. Debido a que para calcular cada $L_{i,j}$ se requiere tiempo constante, la complejidad temporal del algoritmo es $O(mn)$.

A continuación, el método de programación dinámica se ilustra con un ejemplo. Sean $A = a\ b\ c\ d$ y $B = c\ b\ d$. En este caso se tiene

$a_1 = a$
$a_2 = b$
$a_3 = c$
$a_4 = d$
$b_1 = c$
$b_2 = b$
y $b_3 = d$.

La longitud de la subsecuencia común más larga se encuentra paso por paso, lo cual se ilustra en la tabla 7-2.

TABLA 7-2 Los valores de los $L_{i,j}$

b_j \ a_i			**a**	**b**	**c**	**d**
		0	**1**	**2**	**3**	**4**
	0	0	0	0	0	0
c	1	0	0	0	1	1
b	2	0	0	1	1	1
d	3	0	0	1	1	2

$L_{i,j}$

7-3 EL PROBLEMA DE ALINEACIÓN DE 2 SECUENCIAS

Sean $A = a_1a_2 \ldots a_m$ y $B = b_1b_2 \ldots b_n$ dos secuencias sobre un conjunto alfabeto Σ. Una alineación de secuencias de A y B es una matriz $2 \times k$ de M ($k \geq m, n$) de caracteres sobre $\Sigma \bigcup \{-\}$ tal que ninguna columna de M consta completamente de guiones, y los resultados que se obtienen al eliminar todos los guiones en el primer renglón y en el segundo renglón de M son iguales a A y B, respectivamente. Por ejemplo, si $A = a\ b\ c$ y $B = c\ b\ d$, una alineación posible de éstas sería

$$\begin{matrix} a & b & c & - & d \\ - & - & c & b & d \end{matrix}$$

Otra alineación posible de las dos secuencias es

$$\begin{matrix} a & b & c & d \\ c & b & - & d. \end{matrix}$$

Puede verse que la segunda alineación parece mejor que la primera. Con precisión, es necesario definir una función de puntaje que pueda usarse para medir el desempeño de una alineación. A continuación se presentará una función así. Debe entenderse que es posible definir otros tipos de funciones de puntaje.

Sea $f(x, y)$ el puntaje por alinear x con y. Suponga que tanto x como y son caracteres. Entonces $f(x, y) = 2$ si x y y son iguales y $f(x, y) = 1$ si x y y no son iguales. Si x o y es "$-$", entonces $f(x, y) = -1$.

El puntaje de una alineación es la suma total de los puntajes de las columnas. Así, el puntaje de la siguiente alineación es $-1 - 1 + 2 - 1 + 2 = 1$.

$a - b\ c\ d$
$-\ c\ b - d$

El puntaje de la siguiente alineación es $1 + 2 - 1 + 2 = 4$.

$a\ b\ c\ d$
$c\ b - d$

El problema de alineación de 2 secuencias para A y B consiste en encontrar una alineación óptima con el puntaje máximo. Resulta fácil plantear una fórmula de recurrencia semejante a la que se utilizó para encontrar la secuencia común más larga. Sea $A_{i,j}$ el puntaje de la alineación óptima entre $a_1a_2\ldots\ a_i$ y $b_1b_2\ldots\ b_j$, donde $1 \leq i \leq m$ y $1 \leq j \leq n$. Entonces, $A_{i,j}$ puede expresarse como sigue:

$$A_{0,0} = 0$$
$$A_{i,0} = i \cdot f(a_i, -)$$
$$A_{0,j} = j \cdot f(-, b_j)$$
$$A_{i,j} = \text{máx}\begin{cases} A_{i-1,j} + f(a_i, -) \\ A_{i-1,j-1} + f(a_i, b_j) \\ A_{i,j-1} + f(-, b_j) \end{cases}$$

Quizá sea necesario explicar el significado de la fórmula de recurrencia anterior:

$A_{0,0}$ es para la condición inicial.
$A_{i,0}$ significa que todas las $a_1, a_2, \ldots, a_i$ están alineadas con "–".
$A_{0,j}$ significa que todas las $b_1, b_2, \ldots, b_j$ están alineadas con "–".
$A_{i-1,j} + f(a_i, -)$ significa que a_i está alineada con "–" y se requiere encontrar una alineación óptima entre $a_1, a_2, \ldots, a_{i-1}$ y $b_1, b_2, \ldots, b_j$.
$A_{i-1,j-1} + f(a_i, b_j)$ significa que a_i está alineada con b_j y se requiere encontrar una alineación óptima entre $a_1, a_2, \ldots, a_{i-1}$ y $b_1, b_2, \ldots, b_{j-1}$.
$A_{i,j-1} + f(-, b_j)$ significa que b_j está alineada con "–" y se requiere encontrar una alineación óptima entre $a_1, a_2, \ldots, a_i$ y $b_1, b_2, \ldots, b_{j-1}$.

Las $A_{i,j}$ para $A = a\ b\ d\ a\ d$ y $B = b\ a\ c\ d$ usando la fórmula de recurrencia anterior se enumeran en la tabla 7-3.

TABLA 7-3 Los valores de las $A_{i,j}$ para $A = a\ b\ d\ a\ d$ y $B = b\ a\ c\ d$.

a_i / b_j			**a**	**b**	**d**	**a**	**d**
		0	**1**	**2**	**3**	**4**	**5**
	0	0	−1	−2	−3	−4	−5
b	1	−1	1	1	0	−1	−2
a	2	−2	1	2	2	2	1
c	3	−3	0	2	3	3	3
d	4	−4	−1	1	4	4	5

En la tabla 7-3 se registró cómo se obtuvo cada $A_{i,j}$. Una flecha de (a_i, b_j) a (a_{i-1}, b_{j-1}) significa que a_i corresponde a b_j. Una flecha de (a_i, b_j) a (a_{i-1}, b_j) significa que a_i corresponde a "–", y una flecha de (a_i, b_j) a (a_i, b_{j-1}) significa que b_j corresponde a "–". Con base en las flechas de la tabla, es posible rastrear y encontrar que la alineación óptima es:

a b d a d
– b a c d

Se considerará otro ejemplo. Sean $A = a\ b\ c\ d$ y $B = c\ b\ d$ las dos secuencias. En la tabla 7-4 se muestra cómo se obtuvieron las $A_{i,j}$.

TABLA 7-4 Los valores de las $A_{i,j}$ para $A = a\ b\ c\ d$ y $B = c\ b\ d$.

			1	**2**	**3**	**4**
		–	a	b	c	d
	–	0	–1	–2	–3	–4
1	a	–1	1	0	0	–1
2	b	–2	0	3	2	1
3	d	–3	–1	2	4	4

Con base en las flechas de la tabla 7-4, es posible realizar un rastreo y encontrar que la alineación óptima es

a b c d
c b – d

La alineación de secuencias puede considerarse como un método para medir la semejanza de dos secuencias. A continuación se presentará un concepto, denominado distancia de edición, que también se usa bastante a menudo para medir la semejanza entre dos secuencias.

Se considerarán dos secuencias $A = a_1a_2 \ldots a_m$ y $B = b_1b_2 \ldots b_n$. *A* puede transformarse en *B* si se ejecutan las tres siguientes operaciones de edición: eliminar un carácter de *A*, insertar un carácter en *A* y sustituir un carácter en *A* por otro carácter. Por ejemplo, sean *A* = *GTAAHTY* y *B* = *TAHHYC*. *A* puede transformarse en *B* si se realizan las siguientes operaciones:

1. Eliminar el primer carácter *G* de *A*. La secuencia *A* se convierte en *A* = *TAAHTY*.
2. Sustituir el tercer carácter de *A*; a saber, *A* por *H*. La secuencia *A* se convierte en *A* = *TAHHTY*.
3. Eliminar el quinto carácter de *A*; a saber, *T*. La secuencia *A* se convierte en *A* = *TAHHY*.
4. Insertar *C* después del último carácter de *A*. La secuencia *A* se convierte en *A* = *TAHHYC*, que es idéntica a *B*.

Con cada operación puede asociarse un costo. La distancia de edición es el costo mínimo asociado con las operaciones de edición necesarias para transformar la secuencia *A* en la secuencia *B*. Si para cada operación el costo es uno, la distancia de edición se convierte en el número mínimo de operaciones de edición necesarias para transformar *A* en *B*. En el ejemplo anterior, si el costo de cada operación es 1 y la distancia de edición entre *A* y *B* es 4, se requieren por lo menos cuatro operaciones de edición.

Resulta evidente que la distancia de edición puede encontrarse con el método de programación dinámica. Fácilmente puede plantearse una fórmula de recurrencia semejante a la que se utilizó para encontrar la secuencia común más larga o una alineación óptima entre dos secuencias. Sean α, β y γ los costos de inserción, eliminación y sustitución, respectivamente. Sea $A_{i,j}$ la distancia de edición entre $a_1a_2 \ldots a_i$ y $b_1b_2 \ldots b_j$. Entonces, $A_{i,j}$ puede expresarse como sigue:

$$A_{0,0} = 0$$

$$A_{i,0} = i\beta$$
$$A_{0,j} = j\alpha$$
$$A_{i,j} = \begin{cases} A_{i-1,j-1} & \text{si } a_i = b_j \\ \text{mín} \begin{cases} A_{i-1,j} + \beta \\ A_{i-1,j-1} + \gamma \\ A_{i,j-1} + \alpha \end{cases} & \text{en caso contrario} \end{cases}$$

En realidad, resulta fácil ver que el problema para encontrar la distancia de edición es equivalente al problema de la alineación óptima. No es la intención presentar una demostración formal aquí, ya que ésta puede percibirse fácilmente a partir de la semejanza de las fórmulas de recurrencia respectivas. En vez de ello, para ilustrar esta cuestión se usará el ejemplo presentado.

Considere otra vez $A = GTAAHTY$ y $B = TAHHYC$. Una alineación óptima produciría lo siguiente:

$G\ T\ A\ A\ H\ T\ Y\ -$
$-\ T\ A\ H\ H\ -\ Y\ C$

Un análisis de la alineación anterior muestra la equivalencia entre las operaciones de edición y las operaciones de alineación como sigue:

1. (a_i, b_j) en la determinación de la alineación es equivalente a la operación de sustitución en la determinación de la distancia de edición. En este caso, a_i se sustituye por b_j.
2. $(a_i, -)$ en la determinación de la alineación es equivalente a eliminar a_i en A en la determinación de la distancia de edición.
3. $(-, b_j)$ en la determinación de la alineación es equivalente a insertar b_j en A en la determinación de la distancia de edición.

El lector puede usar estas reglas y la alineación óptima ya encontrada para producir las cuatro operaciones de edición. Para dos secuencias dadas $A = a_1a_2 \ldots a_m$ y $B = b_1b_2 \ldots b_n$, para registrar $A_{i,j}$ se requiere una tabla con $(n + 1)(m + 1)$ elementos. Es decir, para encontrar una alineación óptima para las dos secuencias $A = a_1a_2 \ldots a_m$ y $B = b_1b_2 \ldots b_n$ se requiere tiempo $O(nm)$.

7-4 PROBLEMA DE APAREAMIENTO DEL MÁXIMO PAR DE BASES DE ARN

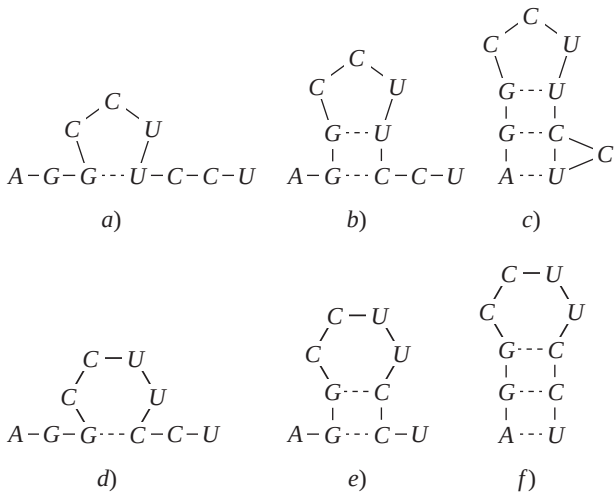
El ácido ribonucleico (ARN) es una cadena simple de nucleótidos (bases), adenina (A), guanina (G), citosina (C) y uracil (U). La secuencia de las bases A, G, C y U se deno-

mina *estructura primaria* de un ARN. En el ARN, *G* y *C* pueden formar un par de bases *G*≡*C* mediante un triple enlace de hidrógeno. *A* y *U* pueden formar un par de bases *A*=*U* mediante un doble enlace de hidrógeno, y *G* y *U* pueden formar una par de bases *G*-*U* mediante un enlace simple de hidrógeno. Debido a estos enlaces de hidrógeno, la estructura primaria de un ARN puede plegarse sobre sí misma para formar su estructura secundaria. Por ejemplo, suponga que se tiene la siguiente secuencia de ARN.

A–G–G–C–C–U–U–C–C–U

Así, esta estructura puede plegarse sobre sí misma para formar muchas estructuras secundarias posibles. En la figura 7-14 se muestran seis estructuras secundarias posibles de esta secuencia. En la naturaleza, no obstante, sólo existe una secuencia secundaria correspondiente a una secuencia de ARN. ¿Cuál es la estructura secundaria verdadera de una secuencia de ARN?

FIGURA 7-14 Seis estructuras secundarias posibles de la secuencia de ARN *A–G–G–C–C–U–U–C–C–U* (las líneas discontinuas indican los enlaces de hidrógeno).



Según la hipótesis de la termodinámica, la estructura secundaria verdadera de una secuencia de ARN es la que tiene energía libre mínima, lo cual se explicará después. En la naturaleza sólo puede existir la estructura estable, y esta estructura debe ser la que tenga energía libre mínima. En una estructura secundaria de ARN, los pares básicos aumentan la estabilidad estructural, y las bases no apareadas la disminuyen. Los pares básicos de los tipos *G*≡*C* y *A*=*U* (denominados pares básicos *Watson-Crick*)

son más estables que los del tipo *G–U* (denominados pares básicos *wobble*). Según estos factores, puede encontrarse que la estructura secundaria de la figura 7-14*f*) es la estructura secundaria verdadera de la secuencia *A–G–G–C–C–U–U–C–C–U*.

Una secuencia de ARN se representa por una cadena de n caracteres $R = r_1 r_2 \ldots r_n$, donde $r_i \in \{A, C, G, U\}$. Típicamente, n puede variar desde 20 hasta 2 000. Una *estructura secundaria* de R es un conjunto S de pares básicos (r_i, r_j) donde $1 \leq i < j \leq n$ tal que se cumplen las condiciones siguientes.

1. $j - i > t$, donde t es una constante positiva pequeña. Típicamente, $t = 3$.
2. Si (r_i, r_j) y (r_k, r_l) son dos pares básicos en S, entonces ocurre una de las siguientes posibilidades
   a) $i = k$ y $j = l$; es decir, (r_i, r_j) y (r_k, r_l) son el mismo par de bases.
   b) $i < j < k < l$; es decir, (r_i, r_j) precede a (r_k, r_l), o bien
   c) $i < k < l < j$; es decir, (r_i, r_j) incluye a (r_k, r_l).

La primera condición implica que la secuencia de ARN no se pliega demasiado acusadamente sobre sí misma. La segunda condición significa que cada nucleótido puede formar parte a lo sumo en un par de bases, y garantiza que la estructura secundaria no contiene un seudonudo. Dos pares básicos (r_i, r_j) y (r_k, r_l) se denominan *seudonudo* si $i < k < j < l$. Los seudonudos ocurren en moléculas de ARN, aunque su exclusión simplifica el problema.

Recuerde que el objetivo de la predicción de la estructura secundaria es encontrar una estructura secundaria que tenga la energía libre mínima. Por lo tanto, es necesario contar con un método para calcular la energía libre de una estructura secundaria S. Debido a que la formación de pares básicos proporciona efectos estabilizadores a la energía libre estructural, el método más simple para medir la energía libre de S consiste en asignar energía a cada par de bases de S y luego la energía libre de S es la suma de las energías de todos los pares básicos. Debido a los diferentes enlaces de hidrógeno, a las energías de los pares básicos suelen asignarse valores distintos. Por ejemplo, los valores razonables para *A≡U*, *G≡C* y *G–U* son -3, -2 y -1, respectivamente. Otros valores posibles podrían ser que todas las energías de todos los pares básicos sean iguales. En este caso, el problema se convierte en encontrar una estructura secundaria con el número máximo de pares básicos. Esta versión del problema de la predicción de la estructura secundaria también se denomina *problema de apareamiento del máximo par de bases de RNA* (maximum base pair matching problem), ya que una estructura secundaria puede considerarse como un apareamiento. A continuación se presentará un algoritmo de programación dinámica para resolver este problema y que se define como sigue: dada una secuencia $R = r_1 r_2 \ldots r_n$ de RNA, encontrar una estructura secundaria de RNA que tenga el número máximo de pares básicos.

Sea $S_{i,j}$ la estructura secundaria del número máximo de pares básicos en la subcadena $R_{i,j} = r_i r_{i+1} \ldots r_j$. El número de pares básicos apareados en $S_{i,j}$ se denota por $M_{i,j}$. Observe que no es posible aparear entre sí dos bases r_i, r_j, donde $1 \leq i < j \leq n$. Los pares básicos admisibles que se consideran aquí son pares básicos Watson-Crick (es decir, $A \equiv U$ y $G = C$) y pares básicos wobble (es decir, G–U). Sea $WW = \{(A, U), (U, A), (G, C), (C, G), (G, U), (U, G)\}$. Luego, se usa una función $\rho(r_i, r_j)$ para indicar si dos bases r_i y r_j cualesquiera pueden constituir un par de bases legal:

$$\rho(r_i, r_j) = \begin{cases} 1 & \text{si } (r_i, r_j) \in WW \\ 0 & \text{en caso contrario} \end{cases}$$

Por definición, se sabe que una secuencia de RNA no se pliega de manera muy pronunciada sobre sí misma. Es decir, si $j - i \leq 3$, entonces r_i y r_j no pueden ser un par de bases de $S_{i,j}$. Entonces, se hace $M_{i,j} = 0$ si $j - i \leq 3$.

Para calcular $M_{i,j}$, donde $j - i > 3$, se consideran los casos siguientes desde el punto de vista de las r_j.

Caso 1: En la solución óptima, r_j no está apareada con ninguna otra base. En este caso, se encuentra una solución óptima para $r_i r_{i+1} \ldots r_{j-1}$ y $M_{i,j} = M_{i,j-1}$ (consulte la figura 7-15).

Caso 2: En la solución óptima, r_j está apareada con r_i. En este caso, se encuentra una solución óptima para $r_{i+1} r_{i+2} \ldots r_{j-1}$ y $M_{i,j} = 1 + M_{i+1,j-1}$ (consulte la figura 7-16).

Caso 3: En la solución óptima, r_j está apareada con alguna r_k, donde $i + 1 \leq k \leq j - 4$. En este caso, se encuentran las soluciones óptimas para $r_i r_{i+1} \ldots r_{k-1}$ y $r_{k+1} r_{k+1} \ldots r_{j-1}$, $M_{i,j} = 1 + M_{i,k-1} + M_{k+1,j-1}$ (consulte la figura 7-17).

Debido a que se quiere encontrar la k entre $i + 1$ y $j + 4$ tal que $M_{i,j}$ sea el máximo, se tiene

FIGURA 7-15 Ilustración del caso 1.

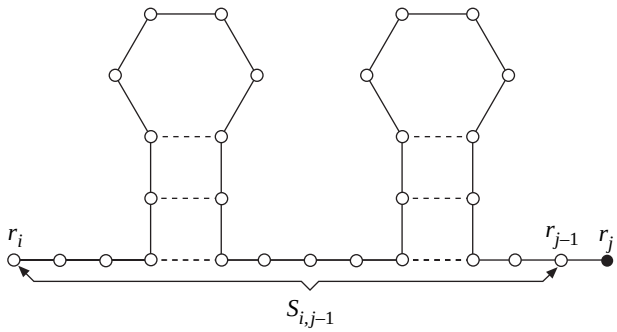


FIGURA 7-16 Ilustración del caso 2.

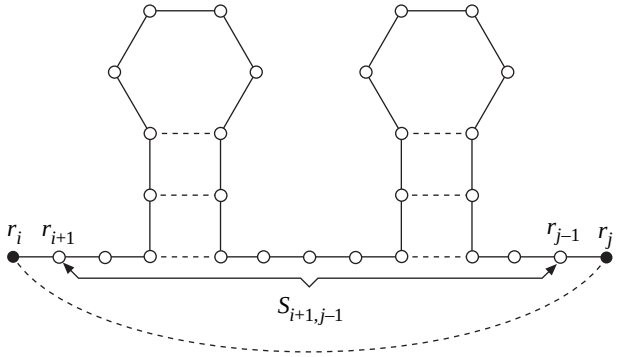
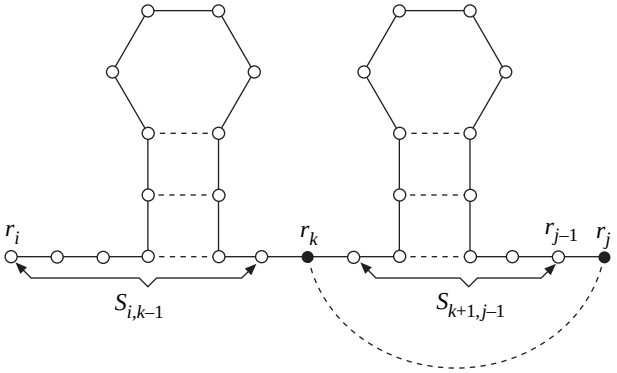


FIGURA 7-17 Ilustración del caso 3.



$$M_{i,j} = \max_{i+1 \le k \le j-4} \{1 + M_{i,k-1} + M_{k+1,j-1}\}$$

En resumen, para calcular $M_{i,j}$ se tiene la siguiente fórmula de recurrencia

Si $j - i \leqq 3$, entonces $M_{i,j} = 0$.

$$\text{Si } j - i > 3, \text{ entonces } M_{i,j} = \max \begin{cases} M_{i,j-1} \\ (1 + M_{i+1,j-1}) \times \rho(r_i, r_j) \\ \max\limits_{i+1 \le k \le j-4} \{1 + M_{i,k-1} + M_{k+1,j-1}\} \times \rho(r_k, r_j) \end{cases}$$

Según la fórmula en la página anterior, el algoritmo 7-1 para calcular $M_{1,n}$ puede diseñarse aplicando la técnica de programación dinámica. En la tabla 7-5 se ilustra el cálculo de $M_{1,n}$, donde $1 \leq i < j \leq 10$, para una secuencia de ARN $R_{1,10} = A–G–G–C–C–U–U–C–C–U$. Como resultado, puede encontrarse que el número máximo de pares básicos en $S_{1,10}$ es 3, ya que $M_{1,10} = 3$.

TABLA 7-5 Cálculo del número máximo de pares básicos de una secuencia de ARN *A–G–G–C–C–U–U–C–C–U*.

i \ *j*	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	0	0	0	0	0	1	2	2	2	3
2	–	0	0	0	0	1	1	2	2	2
3	–	–	0	0	0	0	1	1	1	1
4	–	–	–	0	0	0	0	0	0	0
5	–	–	–	–	0	0	0	0	0	0
6	–	–	–	–	–	0	0	0	0	0
7	–	–	–	–	–	–	0	0	0	0
8	–	–	–	–	–	–	–	0	0	0
9	–	–	–	–	–	–	–	–	0	0
10	–	–	–	–	–	–	–	–	–	0

Algoritmo 7-1 □ Un algoritmo de apareamiento del máximo par de bases de RNA

Input: Una secuencia de ARN $R = r_1 r_2 \ldots r_n$.

Output: Encontrar una estructura secundaria de RNA que tenga el número máximo de pares básicos.

Paso 1: /* Cálculo de la función $\rho(r_i, r_j)$ para $1 \leq i < j \leq n$ */
$WW = \{(A, U), (U, A), (G, C), (C, G), (G, U), (U, G)\}$;
for $i = 1$ **to** n **do**
  for $j = i$ **to** n **do**
    if $(r_i, r_j) \in WW$ **then** $\rho(r_i, r_j) = 1$; **else** $\rho(r_i, r_j) = 0$;

```
        end for
    end for
Paso 2: /* Inicialización de M_{i,j} para j − i ≤ 3 */
    for i = 1 to n do
        for j = i to i + 3 do
            if j ≦ n then M_{i,j} = 0;
        end for
    end for
Paso 3: /* Cálculo de M_{i,j} para j − i > 3 */
    for h = 4 to n − 1 do
        for i = 1 to n − h do
            j = i + h;
            caso1 = M_{i,j−1};
            caso2 = (1 + M_{i+1,j−1}) × ρ(r_i, r_j);
            caso3 = M_{i,j} máx_{i+1≤k≤j−4} {(1 + M_{i,k−1} + M_{k+1,j−1}) × ρ(r_k, r_j)};
            M_{i,j} = máx{caso1, caso2, caso3};
        end for
    end for
```

A continuación se ilustra todo el procedimiento en detalle.

r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}
A	G	G	C	C	U	U	C	C	U

1. $i = 1, j = 5, \rho(r_1, r_5) = \rho(A, C) = 0$

$$M_{1,5} = \text{máx}\begin{cases} M_{1,4} \\ (1 + M_{2,4}) \times \rho(r_1, r_5) \end{cases}$$
$$= \text{máx}\{0, 0\} = 0.$$

2. $i = 2, j = 6, \rho(r_2, r_6) = \rho(G, U) = 1$

$$M_{2,6} = \text{máx}\begin{cases} M_{2,5} \\ (1 + M_{3,5}) \times \rho(r_2, r_6) \end{cases}$$
$$= \text{máx}\{0, (1 + 0) \times 1\} = \text{máx}\{0, 1\} = 1.$$

r_2 coincide con r_6.

3. $i = 3, j = 7, \rho(r_3, r_7) = \rho(G, U) = 1$

$$M_{3,7} = \text{máx}\begin{cases} M_{3,6} \\ (1 + M_{4,6}) \times \rho(r_3, r_7) \end{cases}$$
$$= \text{máx}\{0, (1 + 0) \times 1\} = \text{máx}\{0, 1\} = 1.$$

r_3 coincide con r_7.

4. $i = 4, j = 8, \rho(r_4, r_8) = \rho(C, C) = 0$

$$M_{4,8} = \text{máx}\begin{cases} M_{4,7} \\ (1 + M_{5,7}) \times \rho(r_4, r_8) \end{cases}$$
$$= \text{máx}\{0, (1 + 0) \times 0\} = 0.$$

5. $i = 5, j = 9, \rho(r_5, r_9) = \rho(C, C) = 0$

$$M_{5,9} = \text{máx}\begin{cases} M_{5,8} \\ (1 + M_{6,8}) \times \rho(r_5, r_9) \end{cases}$$
$$= \text{máx}\{0, (1 + 0) \times 0\} = \text{máx}\{0, 0\} = 0.$$

6. $i = 6, j = 10, \rho(r_6, r_{10}) = \rho(U, U) = 0$

$$M_{6,10} = \text{máx}\begin{cases} M_{6,9} \\ (1 + M_{7,9}) \times \rho(r_6, r_{10}) \end{cases}$$
$$= \text{máx}\{0, (1 + 0) \times 0\} = \text{máx}\{0, 0\} = 0.$$

7. $i = 1, j = 6, \rho(r_1, r_6) = \rho(A, U) = 1$

$$M_{1,6} = \text{máx}\begin{cases} M_{1,5} \\ (1 + M_{2,5}) \times \rho(r_1, r_6) \\ (1 + M_{1,1} + M_{3,5}) \times \rho(r_2, r_6) \end{cases}$$
$$= \text{máx}\{0, (1 + 0) \times 1, (1 + 0 + 0) \times 1\}$$
$$= \text{máx}\{0, 1, 1\} = 1.$$

r_1 coincide con r_6.

8. $i = 2, j = 7, \rho(r_2, r_7) = \rho(G, U) = 1$

$$M_{2,7} = \text{máx}\begin{cases} M_{2,6} \\ (1 + M_{3,6}) \times \rho(r_2, r_7) \\ (1 + M_{2,2} + M_{4,6}) \times \rho(r_3, r_7) \end{cases}$$
$$= \text{máx}\{1, (1 + 0) \times 1, (1 + 0 + 0) \times 1\}$$
$$= \text{máx}\{1, 1, 1\} = 1.$$

9. $i = 3, j = 8, \rho(r_3, r_8) = \rho(G, C) = 1$

$$M_{3,8} = \text{máx}\begin{cases} M_{3,7} \\ (1 + M_{4,7}) \times \rho(r_3, r_8) \\ (1 + M_{3,3} + M_{5,7}) \times \rho(r_4, r_8) \end{cases}$$
$$= \text{máx}\{1, (1 + 0) \times 1, (1 + 0 + 0) \times 0\}$$
$$= \text{máx}\{1, 1, 0\} = 1.$$

r_3 coincide con r_8.

10. $i = 4, j = 9, \rho(r_4, r_9) = \rho(C, U) = 0$

$$M_{4,9} = \text{máx}\begin{cases} M_{4,8} \\ (1 + M_{5,8}) \times \rho(r_4, r_9) \\ (1 + M_{4,4} + M_{6,8}) \times \rho(r_5, r_9) \end{cases}$$
$$= \text{máx}\{0, (1 + 0) \times 0, (1 + 0 + 0) \times 0\}$$
$$= \text{máx}\{0, 0, 0\} = 0.$$

11. $i = 5, j = 10, \rho(r_5, r_{10}) = \rho(C, U) = 0$

$$M_{5,10} = \text{máx}\begin{cases} M_{5,9} \\ (1 + M_{6,9}) \times \rho(r_5, r_{10}) \\ (1 + M_{5,5} + M_{7,9}) \times \rho(r_6, r_{10}) \end{cases}$$
$$= \text{máx}\{0, (1 + 0) \times 0, (1 + 0 + 0) \times 0\}$$
$$= \text{máx}\{0, 0, 0\} = 0.$$

12. $i = 1, j = 7, \rho(r_1, r_7) = \rho(A, U) = 1$

$$M_{1,7} = \text{máx}\begin{cases} M_{1,6} \\ (1 + M_{2,6}) \times \rho(r_1, r_7) \\ (1 + M_{1,1} + M_{3,6}) \times \rho(r_2, r_7) \\ (1 + M_{1,2} + M_{4,6}) \times \rho(r_3, r_7) \end{cases}$$

$$= \text{máx}\{1, (1 + 1) \times 1, (1 + 0 + 0) \times 1, (1 + 0 + 0) \times 1\}$$
$$= \text{máx}\{1, 2, 1, 1\} = 2.$$

13. $i = 2, j = 8, \rho(r_2, r_8) = \rho(G, C) = 1$

$$M_{2,8} = \text{máx}\begin{cases} M_{2,7} \\ (1 + M_{3,7}) \times \rho(r_2, r_8) \\ (1 + M_{2,2} + M_{4,7}) \times \rho(r_3, r_8) \\ (1 + M_{2,3} + M_{5,7}) \times \rho(r_4, r_8) \end{cases}$$

$$= \text{máx}\{1, (1 + 1) \times 1, (1 + 0 + 0) \times 1, (1 + 0 + 0) \times 0\}$$
$$= \text{máx}\{1, 2, 1, 0\} = 2.$$

r_2 coincide con r_8; r_3 coincide con r_7.

14. $i = 3, j = 9, \rho(r_3, r_9) = \rho(G, C) = 1$

$$M_{3,9} = \text{máx}\begin{cases} M_{3,8} \\ (1 + M_{4,8}) \times \rho(r_3, r_9) \\ (1 + M_{3,3} + M_{5,8}) \times \rho(r_4, r_9) \\ (1 + M_{3,4} + M_{6,8}) \times \rho(r_5, r_9) \end{cases}$$

$$= \text{máx}\{1, (1 + 0) \times 1, (1 + 0 + 0) \times 0, (1 + 0 + 0) \times 0\}$$
$$= \text{máx}\{1, 1, 0, 0\} = 1.$$

r_3 coincide con r_9.

15. $i = 4, j = 10, \rho(r_4, r_{10}) = \rho(C, U) = 0$

$$M_{4,10} = \text{máx}\begin{cases} M_{4,9} \\ (1 + M_{5,9}) \times \rho(r_4, r_{10}) \\ (1 + M_{4,4} + M_{6,9}) \times \rho(r_5, r_{10}) \\ (1 + M_{4,5} + M_{7,9}) \times \rho(r_6, r_{10}) \end{cases}$$

$$= \text{máx}\{0, (1 + 0) \times 0, (1 + 0 + 0) \times 0, (1 + 0 + 0) \times 0\}$$
$$= \text{máx}\{0, 0, 0, 0\} = 0.$$

16. $i = 1, j = 8, \rho(r_1, r_8) = \rho(A, C) = 0$

$$M_{1,8} = \text{máx}\begin{cases} M_{1,7} \\ (1 + M_{2,7}) \times \rho(r_1, r_8) \\ (1 + M_{1,1} + M_{3,7}) \times \rho(r_2, r_8) \\ (1 + M_{1,2} + M_{4,7}) \times \rho(r_3, r_8) \\ (1 + M_{1,3} + M_{5,7}) \times \rho(r_4, r_8) \end{cases}$$

$$= \text{máx}\{2, (1 + 1) \times 0, (1 + 0 + 1) \times 1, (1 + 0 + 0) \times 1, (1 + 0 + 0) \times 0\}$$

$$= \text{máx}\{2, 0, 2, 1, 0\} = 2.$$

r_1 coincide con r_7; r_2 coincide con r_6.

17. $i = 2, j = 9, \rho(r_2, r_9) = \rho(G, C) = 1$

$$M_{2,9} = \text{máx}\begin{cases} M_{2,8} \\ (1 + M_{3,8}) \times \rho(r_2, r_9) \\ (1 + M_{2,2} + M_{4,8}) \times \rho(r_3, r_9) \\ (1 + M_{2,3} + M_{5,8}) \times \rho(r_4, r_9) \\ (1 + M_{2,4} + M_{6,8}) \times \rho(r_5, r_9) \end{cases}$$

$$= \text{máx}\{2, (1 + 1) \times 1, (1 + 0 + 0) \times 1, (1 + 0 + 0) \times 0, (1 + 0 + 0) \times 0\}$$

$$= \text{máx}\{2, 2, 1, 0, 0\} = 2.$$

r_2 coincide con r_9; r_3 coincide con r_8.

18. $i = 3, j = 10, \rho(r_3, r_{10}) = \rho(G, U) = 1$

$$M_{3,10} = \text{máx}\begin{cases} M_{3,9} \\ (1 + M_{4,9}) \times \rho(r_3, r_{10}) \\ (1 + M_{3,3} + M_{5,9}) \times \rho(r_4, r_{10}) \\ (1 + M_{3,4} + M_{6,9}) \times \rho(r_5, r_{10}) \\ (1 + M_{3,5} + M_{7,9}) \times \rho(r_6, r_{10}) \end{cases}$$

$$= \text{máx}\{1, (1 + 0) \times 1, (1 + 0 + 0) \times 0, (1 + 0 + 0) \times 0, (1 + 0 + 0) \times 0\}$$

$$= \text{máx}\{1, 1, 0, 0, 0\} = 1.$$

r_3 coincide con r_{10}.

19. $i = 1, j = 9, \rho(r_1, r_9) = \rho(A, C) = 0$

$$M_{1,9} = \text{máx}\begin{cases} M_{1,8} \\ (1 + M_{2,8}) \times \rho(r_1, r_9) \\ (1 + M_{1,1} + M_{3,8}) \times \rho(r_2, r_9) \\ (1 + M_{1,2} + M_{4,8}) \times \rho(r_3, r_9) \\ (1 + M_{1,3} + M_{5,8}) \times \rho(r_4, r_9) \\ (1 + M_{1,4} + M_{6,8}) \times \rho(r_5, r_9) \end{cases}$$

$$= \text{máx}\{2, (1 + 2) \times 0, (1 + 0 + 1) \times 1, (1 + 0 + 0) \times 1, (1 + 0 + 0) \times 0, (1 + 0 + 0) \times 0\}$$
$$= \text{máx}\{2, 0, 2, 1, 0, 0\} = 2.$$

r_1 coincide con r_7; r_2 coincide con r_6.

20. $i = 2, j = 10, \rho(r_2, r_{10}) = \rho(G, U) = 1$

$$M_{2,10} = \text{máx}\begin{cases} M_{2,9} \\ (1 + M_{3,9}) \times \rho(r_2, r_{10}) \\ (1 + M_{2,2} + M_{4,9}) \times \rho(r_3, r_{10}) \\ (1 + M_{2,3} + M_{5,9}) \times \rho(r_4, r_{10}) \\ (1 + M_{2,4} + M_{6,9}) \times \rho(r_5, r_{10}) \\ (1 + M_{2,5} + M_{7,9}) \times \rho(r_6, r_{10}) \end{cases}$$

$$= \text{máx}\{2, 2, 1, 0, 0, 0\} = 2.$$

r_2 coincide con r_{10}; r_3 coincide con r_9.

21. $i = 1, j = 10, \rho(r_1, r_{10}) = \rho(A, U) = 1$

$$M_{1,10} = \text{máx}\begin{cases} M_{1,9} \\ (1 + M_{2,9}) \times \rho(r_1, r_{10}) \\ (1 + M_{1,1} + M_{3,9}) \times \rho(r_2, r_{10}) \\ (1 + M_{1,2} + M_{4,9}) \times \rho(r_3, r_{10}) \\ (1 + M_{1,3} + M_{5,9}) \times \rho(r_4, r_{10}) \\ (1 + M_{1,4} + M_{6,9}) \times \rho(r_5, r_{10}) \\ (1 + M_{1,5} + M_{7,9}) \times \rho(r_6, r_{10}) \end{cases}$$

$$= \text{máx}\{2, 3, 2, 1, 0, 0, 0\} = 3.$$

r_1 coincide con r_{10}; r_2 coincide con r_9; r_3 coincide con r_8.

A continuación se analizará la complejidad temporal del algoritmo 7-1. Resulta evidente que el costo del paso 1 es $O(n)$. Para el paso 2, hay cuando mucho $\sum_{i=1}^{n} \sum_{j=i+4}^{n}$ iteraciones y cada iteración cuesta tiempo $O(j - i)$. Por lo tanto, el costo del paso 2 es

$$\sum_{i=1}^{n} \sum_{j=i+4}^{n} O(j-i) = O(n^3)$$

Así, la complejidad temporal del algoritmo 7-1 es $O(n^3)$.

7-5 PROBLEMA 0/1 DE LA MOCHILA

Este problema se analizó en el capítulo 6. En esta sección se demostrará que este problema puede resolverse fácilmente aplicando el método de programación dinámica. El problema 0/1 de la mochila se define como sigue: se tienen n objetos y una mochila. El objeto i pesa W_i y la mochila tiene una capacidad M. Si el objeto i se coloca en la mochila, se obtiene una ganancia P_i. El problema 0/1 de la mochila consiste en maximizar la ganancia total con la restricción de que el peso total de todos los objetos seleccionados sea cuando mucho M.

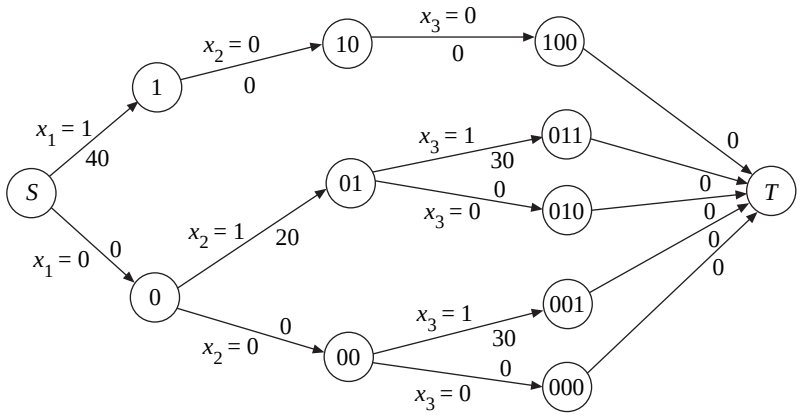
Hay una secuencia de acciones a emprender. Sea X_i la variable que denota si el objeto i se selecciona o no. Es decir, se hace que $X_i = 1$ si el objeto i se escoge y 0 en caso contrario. Si a X_1 se asigna 1 (el objeto 1 es escogido), entonces el problema restante se convierte en un problema 0/1 de la mochila modificado, donde M se vuelve $M - W_1$. En general, después de que se hace una secuencia de decisiones representadas por $X_1, X_2, \ldots, X_i$, el problema se reduce a uno que implica las decisiones $X_{i+1}, X_{i+2}, \ldots, X_n$ y $M' = M - \sum_{j=1}^{i} X_j W_j$. Así, no importa cuáles sean las decisiones $X_1, X_2, \ldots, X_i$, el resto de las decisiones $X_{i+1}, X_{i+2}, \ldots, X_n$ deben ser óptimas con respecto al nuevo problema de la mochila. A continuación se demostrará que el problema 0/1 de la mochila puede representarse mediante un problema de gráfica multietapas. Se supondrá que se tienen tres objetos y una mochila con capacidad 10. Los pesos y las ganancias de estos objetos se muestran en la tabla 7-6.

TABLA 7-6 Pesos y ganancias de tres objetos.

i	W_i	P_i
1	10	40
2	3	20
3	5	30

El método de programación dinámica para resolver este problema 0/1 de la mochila puede ilustrarse en la figura 7-18.

FIGURA 7-18 Método de programación dinámica para resolver el problema 0/1 de la mochila.



En cada nodo se tiene una etiqueta que especifica las decisiones que ya se han tomado hasta este nodo. Por ejemplo, 011 significa $X_1 = 0$, $X_2 = 1$ y $X_3 = 1$. En este caso se tiene interés en la ruta más larga, y es fácil darse cuenta de que ésta es

$$S \rightarrow 0 \rightarrow 01 \rightarrow 011 \rightarrow T$$

que corresponde a

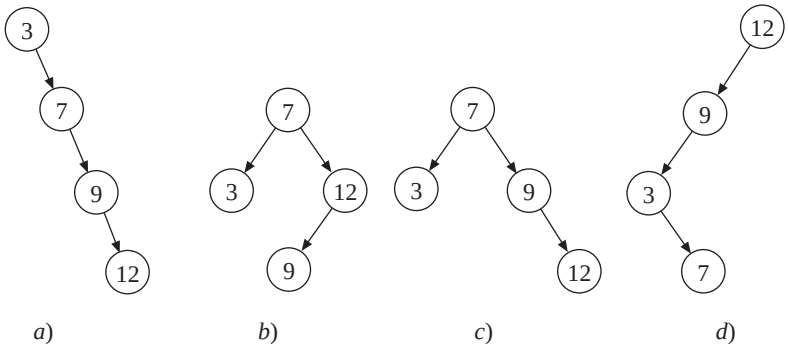
$$X_1 = 0,$$
$$X_2 = 1$$
y $$X_3 = 1$$

con el costo total igual a $20 + 30 = 50$.

7-6 EL PROBLEMA DEL ÁRBOL BINARIO ÓPTIMO

El árbol binario es quizás una de las estructuras de datos más conocidas. Dados n identificadores $a_1 < a_2 < a_3 \cdots < a_n$, pueden tenerse muchos árboles binarios distintos. Por ejemplo, se supondrá que se tienen cuatro identificadores: 3, 7, 9 y 12. En la figura 7-19 se enumeran cuatro árboles binarios distintos para este conjunto de datos.

FIGURA 7-19 Cuatro árboles binarios distintos para el mismo conjunto de datos.



Para un árbol binario dado, los identificadores almacenados cerca de la raíz del árbol pueden buscarse de manera más bien rápida, mientras que para recuperar datos almacenados lejos de la raíz se requieren varios pasos. A cada identificador a_i se asociará una probabilidad P_i, que es la probabilidad de que este identificador sea buscado. Para un identificador que no está almacenado en el árbol, también se supondrá que hay una probabilidad asociada con el identificador. Los datos se separan en $n + 1$ clases de equivalencia. Q_i denota la probabilidad de que X sea buscado donde $a_i < X < a_{i+1}$, en el supuesto de que a_0 es $-\infty$ y a_{n+1} es $+\infty$. Las probabilidades cumplen la siguiente ecuación:

$$\sum_{i=1}^{n} P_i + \sum_{i=0}^{n} Q_i = 1.$$

Resulta fácil determinar el número de pasos necesarios para realizar una búsqueda exitosa. Sea $L(a_i)$ el nivel del árbol binario donde se almacena a_i. Entonces la recuperación de a_i requiere $L(a_i)$ pasos si se hace que la raíz del árbol esté en el nivel 1. Para búsquedas infructuosas, la mejor forma de entenderlas es agregando nodos externos al árbol binario. Se considerará el caso en que se tienen identificadores 4, 5, 8, 10, 11, 12 y 14. Para este conjunto de datos, toda la región se separa como sigue:

$$\begin{array}{cccccccc} & 4, & 5, & 8, & 10, & 11, & 12, & 14 \\ \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\ E_0 & E_1 & E_2 & E_3 & E_4 & E_5 & E_6 & E_7 \end{array}$$

Suponga que se tiene un árbol binario como se muestra en la figura 7-20. En este árbol binario hay algunos nodos que no tienen dos nodos descendientes. En estos nodos siempre ocurren búsquedas infructuosas. A todos estos nodos es posible agregar nodos ficticios, trazados como cuadrados, como se muestra en la figura 7-21.

FIGURA 7-20 Un árbol binario.

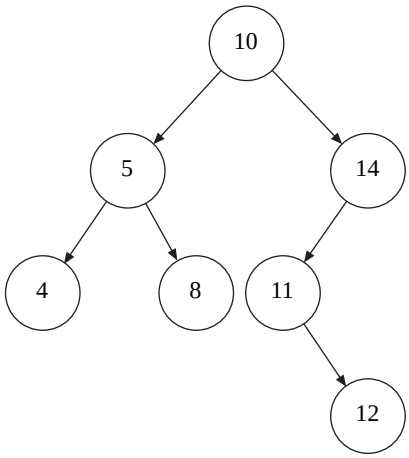
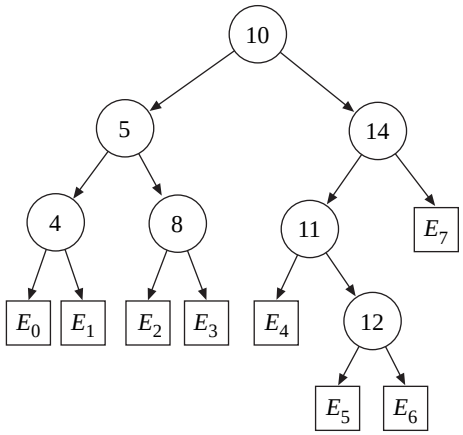


FIGURA 7-21 Un árbol binario con nodos externos agregados.



Todos los demás nodos se denominan nodos internos. Los agregados se denominan nodos externos. Las búsquedas infructuosas siempre terminan en esos nodos externos, mientras que las búsquedas exitosas terminan en nodos internos.

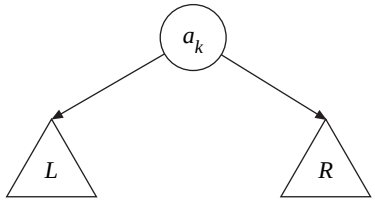
Después de que se agregan los nodos externos, el costo esperado de un árbol binario puede expresarse como sigue:

$$\sum_{i=1}^{n} P_i L(a_i) + \sum_{i=0}^{n} Q_i (L(E_i) - 1).$$

Un árbol binario óptimo es un árbol donde se ha minimizado el costo anterior. Debe observarse que se consumiría un tiempo excesivo si el problema del árbol binario óptimo se resuelve analizando exhaustivamente todos los árboles binarios posibles. Dados n identificadores, el número de árboles binarios distintos que es posible construir a partir de estos n identificadores es $\frac{1}{n+1}\binom{2n}{n}$, que es aproximadamente igual a $O(4^n/n^{3/2})$.

¿Por qué es posible aplicar el método de programación dinámica para resolver el problema del árbol binario óptimo? Un primer paso crítico para encontrar un árbol binario óptimo consiste en seleccionar un identificador, por ejemplo a_k, como raíz del árbol. Después de que se selecciona a_k, todos los identificadores menores que a_k constituyen el descendiente izquierdo de a_k, y todos los identificadores mayores que a_k constituyen el descendiente derecho, lo cual se ilustra en la figura 7-22.

FIGURA 7-22 Un árbol binario después de que a_k se selecciona como la raíz.

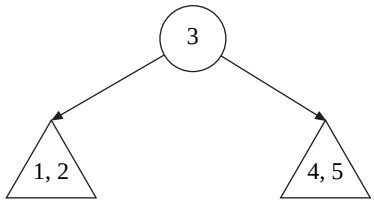


No es fácil determinar cuál a_k debe elegirse, por lo que es necesario analizar todas las a_k posibles. No obstante, una vez que se ha seleccionado a_k, se observa que ambos subárboles L y R también deben ser óptimos. Es decir, una vez que se ha seleccionado a_k, la tarea se reduce a encontrar árboles binarios óptimos para identificadores menores que a_k e identificadores mayores que a_k. El hecho de que este problema puede resolverse de manera recurrente indica que es posible aplicar el método de programación dinámica.

¿Por qué es idóneo el método de programación dinámica para encontrar un árbol binario óptimo? Antes que nada, proporciona una forma de pensamiento sistemático. Para ilustrar este hecho, suponga que hay cinco identificadores: 1, 2, 3, 4 y 5. Suponga que como raíz del árbol se selecciona a 3. Luego, como se muestra en la figura 7-23, el subárbol izquierdo contiene a 1 y 2 y el subárbol derecho contiene a 4 y 5.

El siguiente paso consiste en encontrar árboles binarios óptimos para 1 y 2 y para 4 y 5. Para el árbol binario que contiene a 1 y 2, sólo hay dos opciones para la raíz: 1 o 2. Situaciones semejantes ocurren para el árbol binario que contiene a 4 y 5. En otras palabras, el árbol binario óptimo con 3 como raíz es determinado por los cuatro subárboles siguientes:

FIGURA 7-23 Árbol binario con cierto identificador seleccionado como la raíz.



a) Un subárbol que contiene a 1 y 2 con 1 como la raíz.
b) Un subárbol que contiene a 1 y 2 con 2 como la raíz.
c) Un subárbol que contiene a 4 y 5 con 4 como la raíz.
d) Un subárbol que contiene a 4 y 5 con 5 como la raíz.

El subárbol izquierdo es *a*) o *b*) y el subárbol derecho es *c*) o *d*).

A continuación se analizará el caso en que se selecciona a 2 como la raíz. En este caso, el subárbol izquierdo sólo contiene un identificador; por ejemplo, 1. El subárbol derecho contiene tres identificadores: 3, 4 y 5. Para este subárbol derecho es necesario considerar tres subcasos: seleccionar 3 como la raíz, seleccionar 4 como la raíz y seleccionar 5 como la raíz. Se prestará atención especial al caso en que como la raíz se selecciona a 3. Si se selecciona 3, entonces su subárbol derecho contiene a 4 y 5. Es decir, de nuevo es necesario analizar, en tanto esté implicado un subárbol que contiene a 4 y 5, si debe seleccionarse 4 o 5 como la raíz. Así, se concluye que para encontrar un árbol binario óptimo cuya raíz sea 2, es necesario examinar, entre otros subárboles, los dos árboles siguientes:

e) Un subárbol que contenga a 4 y 5 con 4 como la raíz.
f) Un subárbol que contenga a 4 y 5 con 5 como la raíz.

Resulta obvio que *c*) es equivalente a *e*) y que *d*) es equivalente a *f*). Esto significa que el trabajo realizado para encontrar un árbol binario óptimo con 3 como la raíz también es de utilidad para encontrar un árbol binario óptimo con 2 como la raíz. Con un razonamiento semejante puede verse que en la búsqueda de un árbol binario óptimo con 4 como la raíz es necesario encontrar un árbol binario óptimo que contenga a 1 y 2, con 1 o 2 como la raíz. De nuevo, como ya se indicó, este trabajo es necesario para encontrar un árbol binario óptimo con 3 como la raíz.

En resumen, se observa que dados $a_1, a_2, \ldots, a_n$, el trabajo necesario para encontrar un árbol binario óptimo con, por ejemplo, a_i, como la raíz, quizá también sea necesario para encontrar un árbol binario óptimo con, por ejemplo, a_j, como la raíz. Este principio puede aplicarse a todos los subárboles a todos los niveles. En consecuencia, la programación dinámica debe resolver el problema del árbol binario óptimo trabajando de abajo arriba. Es decir, se empieza por construir árboles binarios óptimos pequeños. Usando estos árboles binarios óptimos pequeños, se construyen árboles binarios óptimos cada vez más grandes. El objetivo se alcanza cuando se encuentra un árbol binario óptimo que contiene a todos los identificadores.

Especificando. Suponga que es necesario encontrar un árbol binario óptimo para cuatro identificadores: 1, 2, 3 y 4. En lo que sigue, se usará la notación $(a_k, a_i \rightarrow a_j)$ para denotar un árbol binario óptimo que contiene los identificadores a_i a a_j y cuya raíz es a_k. También se usará $(a_i \rightarrow a_j)$ para denotar un árbol binario óptimo que contiene los identificadores a_i a a_j. El proceso de programación dinámica para encontrar un árbol binario óptimo que contiene a 1, 2, 3 y 4 se describe a continuación:

1. Se empieza por encontrar lo siguiente:
   $(1, 1 \rightarrow 2)$
   $(2, 1 \rightarrow 2)$
   $(2, 2 \rightarrow 3)$
   $(3, 2 \rightarrow 3)$
   $(3, 3 \rightarrow 4)$
   $(4, 3 \rightarrow 4)$.

2. Al usar los resultados anteriores es posible determinar
   $(1 \rightarrow 2)$ (Determinado por $(1, 1 \rightarrow 2)$ y $(2, 1 \rightarrow 2)$)
   $(2 \rightarrow 3)$
   $(3 \rightarrow 4)$.

3. Luego se encuentra
   $(1, 1 \rightarrow 3)$ (Determinado por $(2 \rightarrow 3)$)
   $(2, 1 \rightarrow 3)$
   $(3, 1 \rightarrow 3)$
   $(2, 2 \rightarrow 4)$
   $(3, 2 \rightarrow 4)$
   $(4, 2 \rightarrow 4)$.

4. Al usar los resultados anteriores es posible determinar
   $(1 \rightarrow 3)$ (Determinado por $(1, 1 \rightarrow 3)$, $(2, 1 \rightarrow 3)$ y $(3, 1 \rightarrow 3)$)
   $(2 \rightarrow 4)$.

5. Luego se encuentra
   $(1, 1 \rightarrow 4)$ (Determinado por $(2 \rightarrow 4)$)
   $(2, 1 \rightarrow 4)$
   $(3, 1 \rightarrow 4)$
   $(4, 1 \rightarrow 4)$.
6. Finalmente, es posible determinar
   $(1 \rightarrow 4)$
   porque está determinado por
   $(1, 1 \rightarrow 4)$
   $(2, 1 \rightarrow 4)$
   $(3, 1 \rightarrow 4)$
   $(4, 1 \rightarrow 4)$.

Cualquiera de los cuatro árboles binarios anteriores con el costo mínimo es óptimo.

El lector puede darse cuenta de que la programación dinámica constituye una manera eficiente para resolver el problema del árbol binario óptimo. No sólo ofrece una forma de pensamiento sistemática, sino que también evita cálculos redundantes.

Una vez que se han descrito los principios básicos para resolver el problema del árbol binario óptimo aplicando el método de programación dinámica, ahora es posible proporcionar el mecanismo preciso. Debido a que la búsqueda de un árbol binario óptimo consiste en encontrar muchos subárboles binarios óptimos, simplemente se considera el caso general de encontrar un subárbol arbitrario. Suponga que se tiene una secuencia de identificadores, $a_1, a_2, \ldots, a_n$ y a_k y que a_k es uno de ellos. Si a_k se selecciona como la raíz del árbol binario, entonces el subárbol izquierdo (derecho) contiene a todos los identificadores $a_1, \ldots, a_{k-1}(a_{k+1}, \ldots, a_n)$. Además, la recuperación de a_k requiere de un paso y todas las otras búsquedas exitosas requieren un paso más que los pasos necesarios para buscar ya sea en el subárbol izquierdo, o derecho. Esto también es cierto para todas las búsquedas infructuosas.

Sea $C(i, j)$ el costo de un árbol binario óptimo que contiene desde a_i hasta a_j. Entonces, el árbol binario óptimo cuya raíz es a_k tiene el costo siguiente

$$C(1, n) = \min_{1 \le k \le n} \left\{ P_k + \left[Q_0 + \sum_{i=1}^{k-1} (P_i + Q_i) + C(1, k-1) \right] + \left[Q_k + \sum_{i=k+1}^{n} (P_i + Q_i) + C(k+1, n) \right] \right\},$$

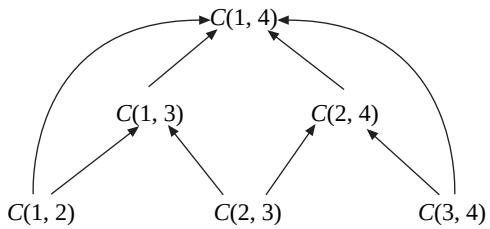
donde P_k es el costo de búsqueda de la raíz y los términos segundo y tercero son los costos de búsqueda para el subárbol izquierdo y el subárbol derecho, respectivamente. La fórmula anterior puede generalizarse para obtener cualquier $C(i, j)$ como sigue:

$$C(i, j) = \min_{i \le k \le j} \left\{ P_k + \left[Q_{i-1} + \sum_{l=i}^{k-1} (P_l + Q_l) + C(i, k-1) \right] + \left[Q_k + \sum_{l=k+1}^{j} (P_l + Q_l) + C(k+1, j) \right] \right\}$$

$$= \min_{i \le k \le j} \left\{ C(i, k-1) + C(k+1, j) + \sum_{l=i}^{j} (P_l + Q_l) + Q_{i-1} \right\}.$$

Por ejemplo, si se tienen cuatro identificadores a_1, a_2, a_3 y a_4, el objetivo es encontrar $C(1, 4)$. Debido a que para la raíz hay cuatro opciones posibles, es necesario calcular el costo de cuatro subárboles óptimos, $C(2, 4)$ (a_1 como la raíz), $C(3, 4)$ (a_2 como la raíz), $C(1, 2)$ (a_3 como la raíz) y $C(1, 3)$ (a_4 como la raíz). Para calcular $C(2, 4)$, se requieren $C(3, 4)$ (a_2 como la raíz) y $C(2, 3)$ (a_4 como la raíz), y para calcular $C(1, 3)$, se requieren $C(2, 3)$ (a_1 como la raíz) y $C(1, 2)$ (a_3 como la raíz). Sus relaciones pueden ilustrarse como se muestra en la figura 7−24.

FIGURA 7-24 Relaciones de cálculo de subárboles.



En general, dados n identificadores para calcular $C(1, n)$, puede procederse calculando primero todos los $C(i, j)$ con $j - i = 1$. Luego, pueden calcularse todos los $C(i, j)$ con $j - i = 2$, luego todos los $C(i, j)$ con $j - i = 3$ y así sucesivamente. A continuación se analiza la complejidad de este procedimiento. Este procedimiento de evaluación requiere el cálculo de $C(i, j)$ para $j - i = 1, 2, \ldots, n - 1$. Cuando $j - i = m$, hay que calcular $(n - m)$ $C(i, j)$. El cálculo de cada uno de estos $C(i, j)$ requiere encontrar el mínimo de m cantidades. Así, cada $C(i, j)$ con $j - i = m$ puede calcularse en tiempo $O(m)$. El tiempo total para calcular todos los $C(i, j)$ con $j - i = m$ es $O(mn - m^2)$. En

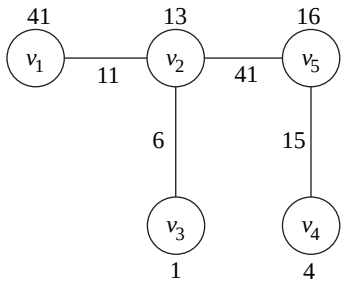
consecuencia, la complejidad temporal total de este método de programación dinámica para resolver el problema del árbol binario óptimo es $O(\sum_{1 \le m \le n} (nm - m^2)) = O(n^3)$.

7-7 EL PROBLEMA PONDERADO DE DOMINACIÓN PERFECTA EN ÁRBOLES

En este problema se tiene una gráfica $G = (V, E)$ donde cada vértice $v \in V$ tiene un costo $c(v)$ y cada arista $e \in E$ también tiene un costo $c(e)$. Un conjunto dominante perfecto de G es un subconjunto D de vértices de V tal que todo vértice que no está en D es adyacente a exactamente un vértice en D. El costo de un conjunto dominante perfecto D incluye todos los costos de los vértices en D y el costo de $c(u, v)$ si v no pertenece a D, u pertenece a D y (u, v) es un arista en E. El problema de dominación perfecta consiste en encontrar un conjunto dominante perfecto con un costo mínimo.

Considere la figura 7-25. Hay muchos conjuntos dominantes perfectos.

FIGURA 7-25 Una gráfica que ilustra el problema ponderado de dominación perfecta.



Por ejemplo, $D_1 = \{v_1, v_2, v_5\}$ es un conjunto dominante perfecto. El costo de D_1 es

$$c(v_1) + c(v_2) + c(v_5) + c(v_3, v_2) + c(v_4, v_5)$$
$$= 41 + 13 + 16 + 6 + 15$$
$$= 91.$$

Otro ejemplo es $D_2 = \{v_2, v_5\}$. El costo de D_2 es

$$c(v_2) + c(v_5) + c(v_1, v_2) + c(v_3, v_2) + c(v_4, v_5)$$
$$= 13 + 16 + 11 + 6 + 15$$
$$= 61.$$

Finalmente, sea D_3 el conjunto $\{v_2, v_3, v_4, v_5\}$. El costo de D_3 es

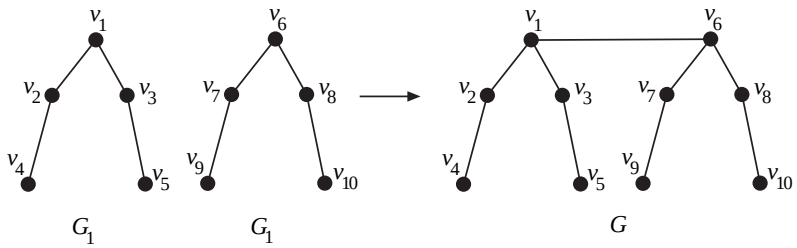
$$c(v_2) + c(v_3) + c(v_4) + c(v_5) + c(v_1, v_2)$$
$$= 13 + 1 + 4 + 16 + 11$$
$$= 45.$$

Puede demostrarse que D_3 es un conjunto dominante perfecto con costo mínimo.

El problema ponderado de dominación perfecta es NP-difícil para gráficas bipartitas y gráficas cordales. Sin embargo, se demostrará que al aplicar el método de programación dinámica es posible resolver el problema ponderado de dominación perfecta en árboles.

La tarea fundamental al aplicar la estrategia de programación dinámica consiste en fusionar dos soluciones factibles en una nueva solución factible. A continuación, la fusión se ilustrará con un ejemplo. Considere la figura 7-27. Ahí hay dos gráficas: G_1 y G_2, que serán fusionadas en G al unir v_1 de G_1 con v_6 de G_2.

FIGURA 7-26 Un ejemplo que ilustra el esquema de fusión al resolver el problema ponderado de dominación perfecta.



Debido a que G_1 y G_2 se combinan al unir v_1 de G_1 con v_6 de G_2, puede considerarse que G_1 está arraigado en v_1 y que G_2 está arraigado en v_6. A continuación se considerarán seis conjuntos dominantes perfectos:

$$D_{11} = \{v_1, v_2, v_3\},\ D_{21} = \{v_6, v_7, v_8\}$$
$$D_{12} = \{v_3, v_4\},\ D_{22} = \{v_7, v_{10}\}$$
$$D_{13} = \{v_4, v_5\},\ D_{23} = \{v_9, v_{10}\}.$$

$D_{11}(D_{21})$ es un conjunto dominante perfecto de $G_1(G_2)$ a condición de que la raíz de $G_1(G_2)$, $v_1(v_6)$ esté incluida en él.

$D_{12}(D_{22})$ es un conjunto dominante perfecto de $G_1(G_2)$ a condición de que la raíz de $G_1(G_2)$, $v_1(v_6)$ no esté incluida en él.

$D_{13}(D_{23})$ es un conjunto dominante perfecto de $G_1 - \{v_1\}(G_2 - \{v_6\})$ a condición de que ninguno de los vecinos v_1 (v_6) esté incluido en él.

Ahora es posible producir conjuntos dominantes perfectos para G al combinar los conjuntos dominantes perfectos anteriores para G_1 y G_2.

1. $D_{11} \bigcup D_{21} = \{v_1, v_2, v_3, v_6, v_7, v_8\}$ es un conjunto dominante perfecto de G. Su costo es la suma de los costos de D_{11} y D_{21}. Este conjunto dominante perfecto incluye tanto a v_1 como a v_6.
2. $D_{11} \bigcup D_{23} = \{v_1, v_2, v_3, v_9, v_{10}\}$ es un conjunto dominante perfecto de G. Su costo es la suma de los costos de D_{11} y D_{23} más el costo de la arista que une v_1 con v_6. Este conjunto dominante perfecto incluye a v_1 y no incluye a v_6.
3. $D_{12} \bigcup D_{22} = \{v_3, v_4, v_7, v_{10}\}$ es un conjunto dominante perfecto de G. Su costo es la suma de los costos de D_{12} y D_{22}. Este conjunto dominante perfecto no incluye a v_1 ni a v_6.
4. $D_{13} \bigcup D_{21} = \{v_4, v_5, v_6, v_7, v_8\}$ es un conjunto dominante perfecto de G. Su costo es la suma de los costos de D_{13} y D_{21} más el costo de la arista que une v_1 con v_6. Este conjunto dominante perfecto no incluye a v_1 pero sí incluye a v_6.

El análisis anterior describe la base de la estrategia de aplicación del método de programación dinámica para resolver el problema ponderado de dominación perfecta.

En lo que sigue, primero se supondrá que un árbol está arraigado en cierto vértice y al combinar las dos gráficas, se unen las dos raíces.

Se define lo siguiente:

$D_1(G, u)$: un conjunto dominante perfecto óptimo para la gráfica G a condición de que el conjunto dominante perfecto incluya a u. El costo de $D_1(G, u)$ se denota por $\delta_1(G, u)$.

$D_2(G, u)$: un conjunto dominante perfecto óptimo para la gráfica G a condición de que el conjunto dominante perfecto no incluya a u. El costo de $D_2(G, u)$ se denota por $\delta_2(G, u)$.

$D_3(G, u)$: un conjunto dominante perfecto óptimo para la gráfica $G - \{u\}$ a condición de que el conjunto dominante perfecto de $G - \{u\}$ no incluya ningún vértice vecino de u. El costo de $D_3(G, u)$ se denota por $\delta_3(G, u)$.

Dadas dos gráficas G_1 y G_2 arraigadas en u y v respectivamente, sea G la gráfica que se obtiene al unir u y v. Así, resulta evidente que se tendrán las reglas siguientes:

REGLA 1: Tanto $D_1(G_1, u) \bigcup D_1(G_2, v)$ como $D_1(G_1, u) \bigcup D_3(G_2, v)$ son conjuntos dominantes perfectos de G que incluyen a u.

REGLA 1.1: El costo de $D_1(G_1, u) \bigcup D_1(G_2, v)$ es $\delta_1(G_1, u) + \delta_1(G_2, v)$.
REGLA 1.2: El costo de $D_1(G_1, u) \bigcup D_3(G_2, v)$ es $\delta_1(G_1, u) + \delta_3(G_2, v) + c(u, v)$.

REGLA 2: Tanto $D_2(G_1, u) \bigcup D_2(G_2, v)$ como $D_3(G_1, u) \bigcup D_1(G_2, v)$ son conjuntos dominantes perfectos de G que no incluyen a u.

REGLA 2.1: El costo de $D_2(G_1, u) \bigcup D_2(G_2, v)$ es $\delta_2(G_1, u) + \delta_2(G_2, v)$.
REGLA 2.2: El costo de $D_3(G_1, u) \bigcup D_1(G_2, v)$ es $\delta_3(G_1, u) + \delta_1(G_2, v) + c(u, v)$.

El principio fundamental de aplicar la estrategia de la programación dinámica para encontrar un conjunto dominante perfecto óptimo de una gráfica puede resumirse como sigue:

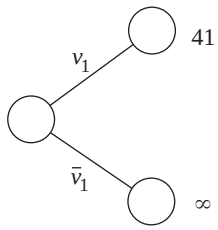
Si una gráfica G puede descomponerse en dos subgráficas G_1 y G_2 de modo que G puede reconstruirse uniendo u de G_1 y v de G_2, entonces un conjunto dominante perfecto de G, denotado por $D(G)$, puede encontrarse como sigue:

1. Si $\delta_1(G_1, u) + \delta_1(G_2, v)$ es menor que $\delta_1(G_1, u) + \delta_3(G_2, v) + c(u, v)$, entonces se hace que $D_1(G, u)$ sea $D_1(G_1, u) \bigcup D_1(G_2, v)$ y $\delta_1(G, u)$ sea $\delta_1(G_1, u) + \delta_1(G_2, v)$. En caso contrario, se hace que $D_1(G, u)$ sea $D_1(G_1, u) \bigcup D_3(G_2, v)$ y $\delta_1(G, u)$ sea $\delta_1(G_1, u) + \delta_3(G_2, v) + c(u, v)$.
2. Si $\delta_2(G_1, u) + \delta_2(G_2, v)$ es menor que $\delta_3(G_1, u) + \delta_1(G_2, v) + c(u, v)$, entonces se hace que $D_2(G, u)$ sea $D_2(G_1, u) \bigcup D_2(G_2, v)$ y $\delta_2(G, u)$ sea $\delta_2(G_1, u) + \delta_2(G_2, v)$. En caso contrario, se hace que $D_2(G, u)$ sea $D_3(G_1, u) \bigcup D_1(G_2, v)$ y $\delta_2(G, u)$ sea $\delta_3(G_1, u) + \delta_1(G_2, v) + c(u, v)$.
3. $D_3(G, u) = D_3(G_1, u) \bigcup D_2(G_2, v)$.
   $\delta_3(G, u) = \delta_3(G_1, u) + \delta_2(G_2, v)$.
4. Si $\delta_1(G, u)$ es menor que $\delta_2(G, u)$, se hace que $D(G)$ sea $D_1(G, u)$. En caso contrario, se hace que $D(G)$ sea $D_2(G, u)$.

Las reglas anteriores son para gráficas generales. En los párrafos siguientes, se abordarán los árboles. Para árboles se usará un algoritmo especial que empieza a partir de los nodos hoja y trabaja hacia la parte interna del árbol. Antes de presentar el algoritmo especial para árboles, el algoritmo se ilustrará con un ejemplo completo. Considere nuevamente la figura 7-25. Se empezará sólo desde el nodo hoja v_1. Sólo hay dos

casos: el conjunto dominante perfecto contiene a v_1 o no contiene a v_1, como se muestra en la figura 7-27.

FIGURA 7-27 Cálculo del conjunto dominante perfecto que implica a v_1.



Resulta fácil ver que

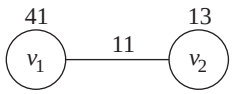
$$D_1(\{v_1\}, v_1) = \{v_1\},$$
$$D_2(\{v_1\}, v_1) \text{ no existe},$$
$$D_3(\{v_1\}, v_1) = \phi,$$
$$\delta_1(\{v_1\}, v_1) = c(v_1) = 41,$$
$$\delta_2(\{v_1\}, v_1) = \infty,$$
y $$\delta_3(\{v_1\}, v_1) = 0.$$

A continuación se considerará el subárbol que sólo contiene a v_2. De nuevo, es posible ver lo siguiente:

$$D_1(\{v_2\}, v_2) = \{v_2\},$$
$$D_2(\{v_2\}, v_2) \text{ no existe},$$
$$D_3(\{v_2\}, v_2) = \phi,$$
$$\delta_1(\{v_2\}, v_2) = c(v_2) = 13,$$
$$\delta_2(\{v_2\}, v_2) = \infty,$$
y $$\delta_3(\{v_2\}, v_2) = 0.$$

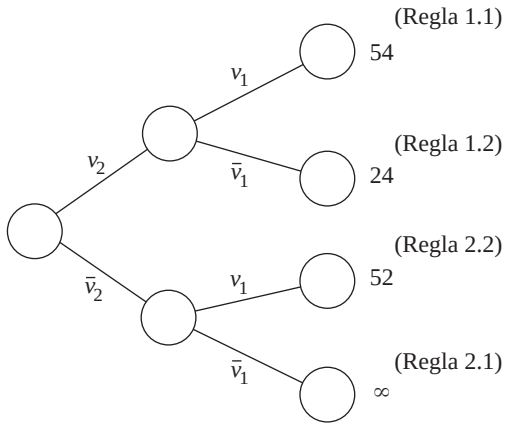
Considere el subárbol que contiene a v_1 y v_2, que se muestra en la figura 7-28.

FIGURA 7-28 Subárbol que contiene a v_1 y v_2.



El cálculo del conjunto dominante perfecto para $\{v_1, v_2\}$, mediante programación dinámica, se ilustra en la figura 7-29.

FIGURA 7-29 Cálculo del conjunto dominante perfecto del subárbol que contiene a v_1 y v_2.



Debido a que mín(54, 24) = 24, se tiene

$D_1(\{v_1, v_2\}, v_2) = \{v_2\}$,
$\delta_1(\{v_1, v_2\}, v_2) = 24$.

Debido a que mín(∞, 52) = 52 se tiene

$D_2(\{v_2, v_1\}, v_2) = \{v_1\}$,
$\delta_2(\{v_2, v_1\}, v_2) = 52$.

Además,

$D_3(\{v_1, v_2\}, v_2)$ no existe,
$\delta_3(\{v_1, v_2\}, v_2) = \infty$.

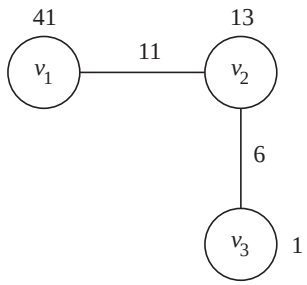
Considere el árbol que sólo contiene a v_3. Resulta evidente que

$D_1(\{v_3\}, v_3) = \{v_3\}$,
$D_2(\{v_3\}, v_3)$ no existe,

$D_3(\{v_3\}, v_3) = \phi$,
$\delta_1(\{v_3\}, v_3) = c(v_3) = 1$,
$\delta_2(\{v_3\}, v_3) = \infty$,
y $\delta_3(\{v_3\}, v_3) = 0$.

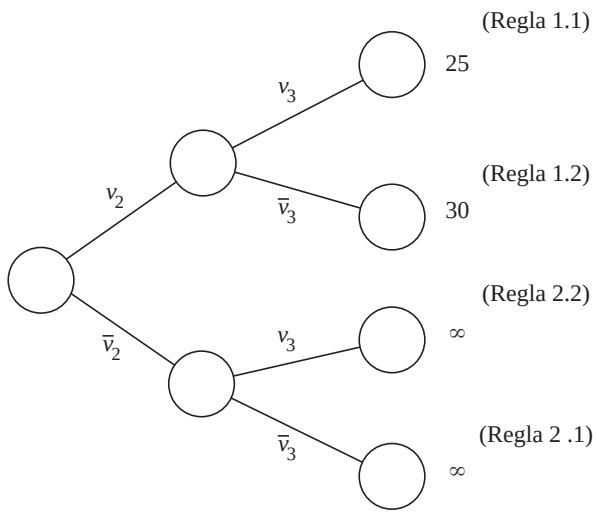
Al subárbol que contiene a v_1 y v_2 se agrega v_3, lo cual se muestra en la figura 7-30.

FIGURA 7-30 Subárbol que contiene a v_1, v_2 y v_3.



El cálculo del conjunto dominante perfecto para este subárbol se ilustra en la figura 7-31.

FIGURA 7-31 Cálculo del conjunto dominante perfecto del subárbol que contiene a v_1, v_2 y v_3.

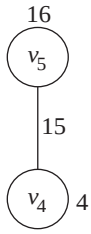


Con base en el cálculo, se tiene

$$D_1(\{v_1, v_2, v_3\}, v_2) = \{v_2, v_3\},$$
$$D_2(\{v_1, v_2, v_3\}, v_2) \text{ no existe},$$
$$D_3(\{v_1, v_2, v_3\}, v_2) \text{ no existe},$$
$$\delta_1(\{v_1, v_2, v_3\}, v_2) = c(v_2) + c(v_3) + c(v_1, v_2) = 13 + 1 + 11 = 25,$$
$$\delta_2(\{v_1, v_2, v_3\}, v_2) = \infty,$$
y $$\delta_3(\{v_1, v_2, v_3\}, v_2) = \infty.$$

Considere el subárbol que contiene a v_5 y v_4, que se muestra en la figura 7-32.

FIGURA 7-32 Subárbol que contiene a v_5 y v_4.

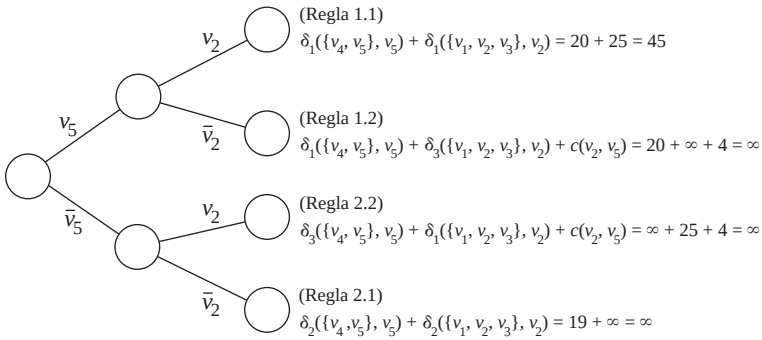


Fácilmente puede verse que lo siguiente es cierto.

$$D_1(\{v_5, v_4\}, v_5) = \{v_5, v_4\},$$
$$D_2(\{v_5, v_4\}, v_5) = \{v_4\},$$
$$D_3(\{v_5, v_4\}, v_5) \text{ no existe},$$
$$\delta_1(\{v_5, v_4\}, v_5) = c(v_5) + c(v_4) = 16 + 4 = 20,$$
$$\delta_2(\{v_5, v_4\}, v_5) = c(v_4) + c(v_5, v_4) = 4 + 15 = 19,$$
y $$\delta_3(\{v_5, v_4\}, v_5) = \infty.$$

Ahora se considerará el subárbol que contiene a v_1, v_2 y v_3, así como el subárbol que contiene a v_4 y v_5. Es decir, ahora se considerará todo el árbol. El cálculo de este conjunto dominante perfecto se muestra en la figura 7-33.

FIGURA 7-33 Cálculo del conjunto dominante perfecto de todo el árbol que se muestra en la figura 7-25.



Puede concluirse que el conjunto dominante perfecto con costo mínimo es $\{v_2, v_3, v_4, v_5\}$, cuyo costo es 45.

En el siguiente algoritmo, sea $TP(u)$ el árbol parcialmente construido en el proceso y que está arraigado en u.

Algoritmo 7-2 □ Un algoritmo para resolver el problema ponderado de dominación perfecta en árboles

Input: Un árbol $T = (V, E)$ donde todo vértice $v \in V$ tiene un costo $c(v)$ y toda arista $e \in E$ tiene un costo $c(e)$.

Output: Un conjunto dominante perfecto $D(T)$ de T con costo mínimo.

Paso 1: Para todo vértice $v \in V$, hacer

$TP(v) = \{v\}$

$D_1(TP(v), v) = TP(v)$

$\delta_1(TP(v), v) = c(v)$

$D_2(TP(v), v)$ no existe

$\delta_2(TP(v), v) = \infty$

$D_3(TP(v), v) = \phi$

$\delta_3(TP(v), v) = 0$.

Paso 2: $T' = T$.

Paso 3: Mientras T' tiene más de un vértice, hacer:

escoger una hoja v de T' que sea adyacente a un único vértice u en T'.

Hacer $TP'(u) = TP(u) \bigcup TP(v) \bigcup \{u, v\}$.

Si $(\delta_1(TP(u), u) + \delta_1(TP(v), v)) < (\delta_1(TP(u), u) + \delta_3(TP(v), v) + c(u, v))$

$D_1(TP'(u), u) = D_1(TP(u), u) \bigcup D_1(TP(v), v)$

$\delta_1(TP'(u), u) = \delta_1(TP(u), u) + \delta_1(TP(v), v)$.

En caso contrario,

$D_1(TP'(u), u) = D_1(TP(u), u) \bigcup D_3(TP(v), v)$

$\delta_1(TP'(u), u) = \delta_1(TP(u), u) + \delta_3(TP(v), v) + c(u, v)$.

Si $(\delta_2(TP(u), u) + \delta_2(TP(v), v)) < (\delta_3(TP(u), u) + \delta_1(TP(v), v) + c(u, v))$

$D_2(TP'(u), u) = D_2(TP(u), u) \bigcup D_2(TP(v), v)$

$\delta_2(TP'(u), u) = \delta_2(TP(u), u) + \delta_2(TP(v), v)$.

En caso contrario,

$D_2(TP'(u), u) = D_3(TP(u), u) \bigcup D_1(TP(v), v)$

$\delta_2(TP'(u), u) = \delta_3(TP(u), u) + \delta_1(TP(v), v) + c(u, v)$

$D_3(TP'(u), u) = D_3(TP(u), u) \bigcup D_2(TP(v), v)$

$\delta_3(TP'(u), u) = \delta_3(TP(u), u) + \delta_2(TP(v), v)$

$TP(u) = TP'(u)$

$T' = T' - v$;

end while.

Paso 4: Si $\delta_1(TP(u), u) < \delta_2(TP(u), u)$

Hacer $D(T) = D_1(TP(u), u)$.

En caso contrario,

hacer $D(T) = D_2(TP(u), u)$.

Regresar $D(T)$ como el conjunto dominante perfecto mínimo de $T(V, E)$.

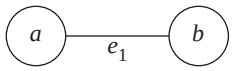
7-8 EL PROBLEMA PONDERADO DE BÚSQUEDA DE BORDES EN UNA GRÁFICA EN UN SOLO PASO

En este problema se tiene una gráfica no dirigida simple $G = (V, E)$ donde cada vértice $v \in V$ está asociado con un peso $wt(v)$. Todos los bordes de G tienen la misma longitud. Se supone que en alguna arista de G está escondido un fugitivo que puede moverse a cualquier velocidad. En cada arista de G, para buscar al fugitivo se asigna un buscador de aristas. El buscador siempre empieza desde un vértice. El costo de recorrer una arista (u, v) se define como $wt(u)$ si el buscador empieza desde u y como $wt(v)$ si empieza desde v. Suponga que todos los buscadores de aristas se desplazan a la misma velocidad. El problema ponderado de búsqueda de aristas en una gráfica simple en un solo paso consiste en disponer las direcciones de búsqueda de los buscadores de aristas de modo que el fugitivo sea atrapado en un paso y se minimice el número de buscadores utilizados.

Resulta evidente que si hay m aristas, se requieren por lo menos m buscadores de aristas. Debido a que el fugitivo puede moverse tan rápido como los buscadores de aristas y cada uno de éstos se desplaza hacia delante, el equipo de buscadores puede capturar al fugitivo en un solo paso si éste no puede escabullirse a través de ningún vértice ya explorado y esconderse detrás de los buscadores de aristas.

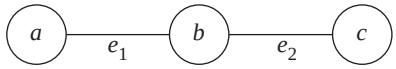
Se considerará la figura 7-34. En este caso sólo se requiere un buscador. No importa dónde esté ubicado inicialmente el buscador, el fugitivo no puede escapar. Suponga que el buscador se encuentra en a y busca en dirección de b. Cuando mucho, el fugitivo puede llegar a b. Pero entonces el buscador también llegará a b y lo capturará. De manera semejante, si un buscador está inicialmente en b, es posible alcanzar el mismo objetivo. Es decir, es posible atrapar al fugitivo en un paso.

FIGURA 7-34 Caso en que no se requieren buscadores adicionales.



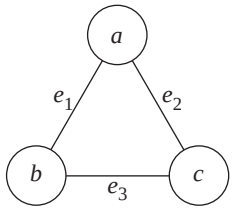
Ahora, considere la figura 7-35. Al colocar buscadores en a y c también es posible atrapar al fugitivo en un paso.

FIGURA 7-35 Otro caso en que no se requieren buscadores adicionales.



Ahora considere la figura 7-36.

FIGURA 7-36 Caso en que se requieren buscadores adicionales.



En este caso, si sólo se usan tres buscadores, entonces sin importar cómo se dispongan las direcciones de búsqueda, es posible que el fugitivo evite ser capturado. Por ejemplo, considere el siguiente plan de búsqueda:

1. El buscador en la arista e_1 busca de a a b.
2. El buscador en la arista e_2 busca de a a c.
3. El buscador en la arista e_3 busca de b a c.

Entonces, si el fugitivo está originalmente en e_1, puede pasar desapercibido si se desplaza a e_3.

Suponga que en el vértice b se coloca un buscador adicional y que se aplica el mismo plan de búsqueda ya presentado. Entonces sin importar dónde estaba originalmente el fugitivo, será atrapado. Esto demuestra que en algunos casos se requieren buscadores adicionales.

Un plan de búsqueda en un solo paso consiste en disponer las direcciones de búsqueda de los buscadores de bordes y determinar el número mínimo de buscadores adicionales requeridos. El costo de un plan de búsqueda en un solo paso es la suma de los costos de todos los buscadores. El problema ponderado de búsqueda de bordes en una gráfica en un solo paso consiste en encontrar un plan de búsqueda en un solo paso factible con costo mínimo. Para el ejemplo anterior, el costo de este plan de búsqueda es $2wt(a) + wt(b) + wt(b)$.

El problema ponderado de búsqueda de bordes en una gráfica en un solo paso es NP-difícil para gráficas generales. Sin embargo, se demostrará que al aplicar la estrategia de la programación dinámica, el problema ponderado de búsqueda de bordes en una gráfica en un solo paso en árboles puede resolverse en tiempo polinomial.

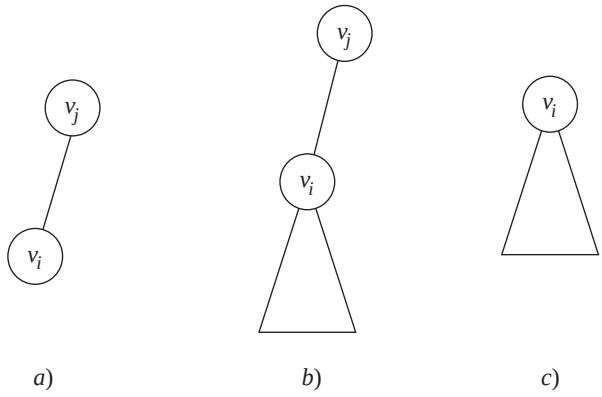
Nuestro método de programación dinámica para resolver este problema se basa en varias reglas básicas que se explicarán a continuación.

Primero se definen algunas notaciones.

Sea v_i el vértice de un árbol.

Caso 1: v_i es un nodo hoja. Entonces $T(v_i)$ denota a v_i y a su nodo padre, lo cual se muestra en la figura 7-37*a*).

FIGURA 7-37 Definición de $T(v_i)$.



Caso 2: v_i es un nodo interno, pero no el nodo raíz. Así, (v_i) denota al árbol que implica a v_i, al nodo padre v_j y a todos los nodos descendientes de v_i, lo cual se muestra en la figura 7-37*b*).

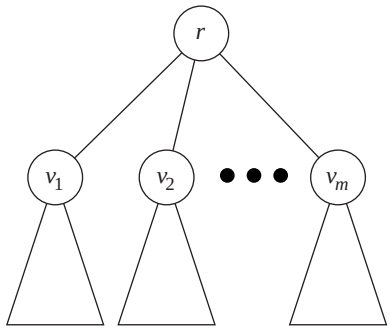
Caso 3: v_i es la raíz de un árbol T. Entonces $T(v_i)$ denota a T, como se muestra en la figura 7-37*c*).

Además, sea v_j el nodo padre de v_i. Entonces $C(T(v_i), v_i, v_j)$ ($C(T(v_i), v_j, v_i)$) denota el costo de un plan de búsqueda óptimo en un solo paso donde la dirección de búsqueda de (v_j, v_i) es de v_i a v_j (de v_j a v_i).

Regla 1: Sea r la raíz de un árbol. Sea $C(T(r), r)(C(T(r), \overline{r}))$ el costo de un plan de búsqueda óptimo en un solo paso para T con (sin) un buscador adicional ubicado en r. Así, el costo de un plan de búsqueda óptimo en un solo paso para T, denotado por $C(T(r))$, es mín$\{C(T(r), r), C(T(r), \overline{r})\}$.

Regla 2: Sea r la raíz de un árbol donde $v_1, v_2, \ldots, v_m$ son descendientes de r (consulte la figura 7-38). Si en r no hay un guardia adicional, entonces todas las direcciones de búsqueda de (r, v_1), (r, v_2), …, (r, v_m) van de r a $v_1, v_2, \ldots, v_m$ o todas van de $v_1, v_2, \ldots, v_m$ a r. Si en r hay un guardia adicional, entonces la dirección de búsqueda de cada (r, v_i) puede determinarse de manera independiente.

FIGURA 7-38 Ilustración de la regla 2.



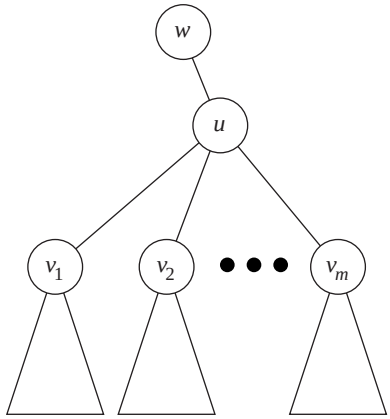
Es decir,

$$C(T(r), \bar{r}) = \text{mín}\left\{\sum_{i=1}^{m} C(T(v_i), r, v_i), \sum_{i=1}^{m} C(T(v_i), v_i, r)\right\}$$

$$C(T(r), r) = wt(r) + \sum_{i=1}^{m} \text{mín}\{C(T(v_i), r, v_i), C(T(v_i), v_i, r)\}.$$

Regla 3: Sean u un nodo interno, w su nodo padre y $v_1, v_2, \ldots, v_m$ sus nodos descendientes (figura 7-39).

FIGURA 7-39 Ilustración de la regla 3.



Si en u no hay un guardia adicional, entonces si la dirección de búsqueda de (w, u) es de w a u (de u a w), todas las direcciones de búsqueda de (u, v_1), (u, v_2), …, (u, v_m) son de $v_1, v_2, \ldots, v_m$ a u (de u a $v_1, v_2, \ldots, v_m$). Si en u hay un guardia adicional, entonces las direcciones de búsqueda de (u, v_j) pueden determinarse de manera independiente. Sea $C(T(u), w, u, \bar{u})$ ($C(T(u), w, u, u)$) el costo de un plan de búsqueda óptimo en un solo paso donde la dirección de búsqueda de la arista (u, w) es de w a u y en el vértice u no hay un guardia adicional (en el vértice u hay un guardia adicional). De manera semejante, sea $C(T(u), u, w, \bar{u})$ ($C(T(u), u, w, u)$ el costo de un plan de búsqueda óptimo en un solo paso donde la dirección de búsqueda de la arista (u, w) es de u a w y en el vértice u no hay un guardia adicional (en el vértice u hay un guardia adicional). Entonces se tienen las fórmulas siguientes.

$$C(T(u), w, u, \bar{u}) = wt(w) + \sum_{i=1}^{m} C(T(v_i), v_i, u),$$

$$C(T(u), w, u, u) = wt(w) + wt(u) + \sum_{i=1}^{m} \min\{C(T(v_i), v_i, u), C(T(v_i), u, v_i)\},$$

$$C(T(u), u, w, \bar{u}) = wt(u) + \sum_{i=1}^{m} C(T(v_i), u, v_i),$$

y $$C(T(u), u, w, u) = 2wt(u) + \sum_{i=1}^{m} \min\{C(T(v_i), v_i, u), C(T(v_i), u, v_i)\}.$$

Entonces,

$$C(T(u), w, u) = \min\{C(T(u), w, u, \bar{u}), C(T(u), w, u, u)\},$$
$$C(T(u), u, w) = \min\{C(T(u), u, w, \bar{u}), C(T(u), u, w, u)\}.$$

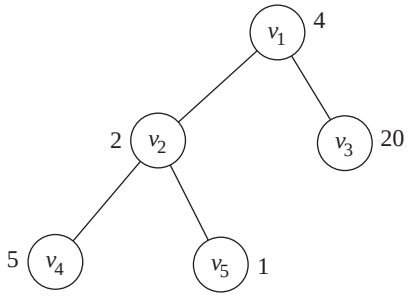
Finalmente, se tiene una regla que concierne a las condiciones en la frontera.

Regla 4: Si u es un nodo hoja y w es su nodo padre, entonces $C(T(u), w, u) = wt(w)$ y $C(T(u), u, w) = wt(u)$.

Nuestro método de programación dinámica resuelve el problema ponderado de búsqueda de bordes en una gráfica en un solo paso al trabajar desde los nodos hoja y gradualmente hacerlo hacia la raíz del árbol. Si un nodo es una hoja, entonces se aplica la regla 4. Si se trata de un nodo interno, pero no es la raíz, entonces se aplica la regla 3. Si es el nodo raíz, entonces primero se aplica la regla 2 y luego la regla 1.

El método se ilustrará con un ejemplo. Considere el árbol ponderado en la figura 7-40.

FIGURA 7-40 Árbol que ilustra el método de programación dinámica.



El método de programación dinámica puede trabajar como sigue:

Paso 1: Se escoge v_3, que es un nodo hoja. Se aplica la regla 4. El nodo padre de v_3 es v_1.

$C(T(v_3), v_1, v_3) = wt(v_1) = 4$

$C(T(v_3), v_3, v_1) = wt(v_3) = 20.$

Paso 2: Se escoge v_4, que es un nodo hoja. Se aplica la regla 4. El nodo padre de v_4 es v_2.

$C(T(v_4), v_2, v_4) = wt(v_2) = 2$

$C(T(v_4), v_4, v_2) = wt(v_4) = 5.$

Paso 3: Se escoge v_5, que es un nodo hoja. Se aplica la regla 4. El nodo padre de v_5 es v_2.

$C(T(v_5), v_2, v_5) = wt(v_2) = 2$

$C(T(v_5), v_5, v_2) = wt(v_5) = 1.$

Paso 4: Se escoge v_2, que es un nodo interno, pero no la raíz. Se aplica la regla 3. El nodo padre de v_2 es v_1. Los nodos descendientes de v_2 son v_4 y v_5.

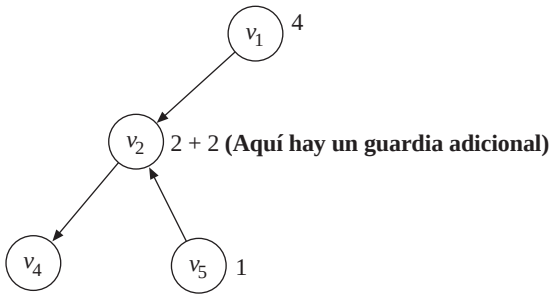
$$C(T(v_2), v_1, v_2) = \text{mín}\{C(T(v_2), v_1, v_2, \overline{v_2}), C(T(v_2), v_1, v_2, v_2)\}$$

$$\begin{aligned} C(T(v_2), v_1, v_2, \overline{v_2}) &= wt(v_1) + C(T(v_4), v_4, v_2) + C(T(v_5), v_5, v_2) \\ &= 4 + 5 + 1 \\ &= 10 \end{aligned}$$

$$
\begin{aligned}
&C(T(v_2), v_1, v_2, \overline{v_2})\\
&= wt(v_1) + wt(v_2) + \text{mín}\{C(T(v_4), v_2, v_4), C(T(v_4), v_4, v_2)\}\\
&\quad + \text{mín}\{C(T(v_5), v_2, v_5), C(T(v_5), v_5, v_2)\}\\
&= 4 + 2 + \text{mín}\{2, 5\} + \text{mín}\{2, 1\}\\
&= 6 + 2 + 1\\
&= 9\\
&C(T(v_2), v_1, v_2) = \text{mín}\{10, 9\} = 9.
\end{aligned}
$$

Observe que el cálculo anterior indica que si la dirección de búsqueda de (v_1, v_2) es de v_1 a v_2, entonces el plan de búsqueda óptimo es el que se muestra en la figura 7-41. En v_2 hay un guardia adicional.

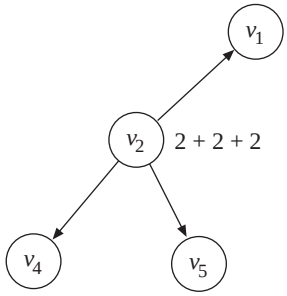
FIGURA 7-41 Plan de búsqueda en un solo paso que implica a v_2.



$$
\begin{aligned}
&C(T(v_2), v_2, v_1) = \text{mín}\{C(T(v_2), v_2, v_1, \overline{v_2}), C(T(v_2), v_2, v_1, v_2)\}\\
&C(T(v_2), v_2, v_1, \overline{v_2}) = wt(v_2) + C(T(v_4), v_2, v_4) + C(T(v_5), v_2, v_5)\\
&\qquad\qquad = 2 + 2 + 2\\
&\qquad\qquad = 6\\
&C(T(v_2), v_2, v_1, v_2)\\
&= wt(v_2) + wt(v_2) + \text{mín}\{C(T(v_4), v_2, v_4), C(T(v_4), v_4, v_2)\}\\
&\quad + \text{mín}\{C(T(v_5), v_2, v_5), C(T(v_5), v_5, v_2)\}\\
&= 2 + 2 + \text{mín}\{2, 5\} + \text{mín}\{2, 1\}\\
&= 4 + 2 + 1\\
&= 7\\
&C(T(v_2), v_2, v_1) = \text{mín}\{6, 7\} = 6.
\end{aligned}
$$

El cálculo anterior indica que si el plan de búsqueda de (v_1, v_2) es de v_2 a v_1, entonces el plan de búsqueda óptimo en un solo paso es el que se muestra en la figura 7-42.

FIGURA 7-42 Otro plan de búsqueda en un solo paso que implica a v_2.

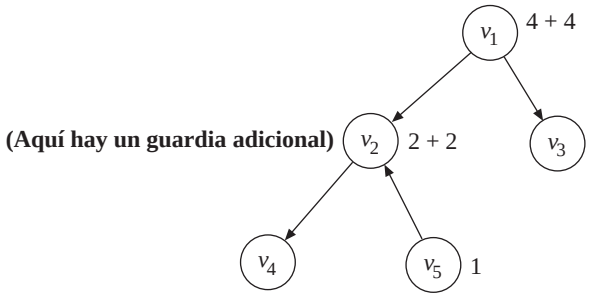


Paso 5. Se escoge v_1. Debido a que v_1 es la raíz, se aplica la regla 2.

$$\begin{aligned} C(T(v_1), \overline{v_1}) &= \text{mín}\{C(T(v_2), v_1, v_2) + C(T(v_3), v_1, v_3), \\ &\qquad C(T(v_2), v_2, v_1) + C(T(v_3), v_3, v_1)\} \\ &= \text{mín}\{9 + 4, 6 + 20\} \\ &= \text{mín}\{13, 26\} \\ &= 13. \end{aligned}$$

Al hacer un rastreo, se comprende que si en v_1 no hay un guardia adicional, entonces el plan de búsqueda óptimo en un solo paso es el que se muestra en la figura 7-43.

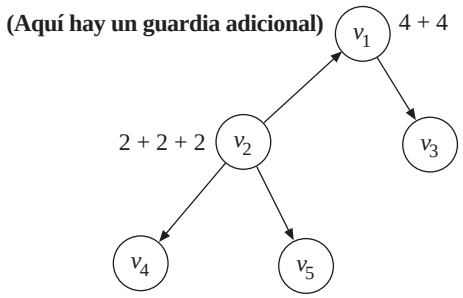
FIGURA 7-43 Plan de búsqueda en un solo paso que implica a v_1.



$$\begin{aligned} C(T(v_1), v_1) &= wt(v_1) + \text{mín}\{C(T(v_2), v_1, v_2), C(T(v_2), v_2, v_1)\} \\ &\quad + \text{mín}\{C(T(v_3), v_1, v_3), C(T(v_3), v_3, v_1)\} \\ &= 4 + \text{mín}\{9, 6\} + \text{mín}\{4, 20\} \\ &= 4 + 6 + 4 \\ &= 14. \end{aligned}$$

Al hacer un rastreo, se comprende que si en v_1 hay un guardia adicional, entonces el plan de búsqueda óptimo en un solo paso es el que se muestra en la figura 7-44.

FIGURA 7-44 Otro plan de búsqueda en un solo paso que implica a v_1.



Finalmente, se aplica la regla 1.

$$\begin{aligned} C(T(v_1)) &= \text{mín}\{C(T(v_1), v_1), C(T(v_1), \overline{v_1})\} \\ &= \text{mín}\{14, 13\} \\ &= 13. \end{aligned}$$

Esto significa que en v_1 no debe haber ningún guardia adicional y que el plan de búsqueda óptimo es el que se muestra en la figura 7-43.

El número de operaciones sobre cada vértice es dos; una para calcular el costo de búsqueda mínimo y otra para determinar las direcciones de búsqueda en los bordes. Así, el número total de operaciones es $O(n)$, donde n es el número de nodos del árbol. Debido a que para cada operación se requiere tiempo constante, este algoritmo es lineal.

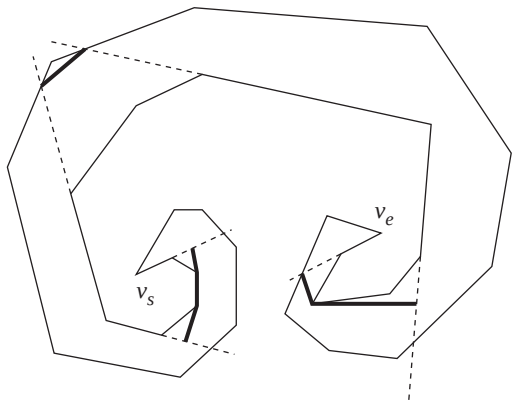
7-9 EL PROBLEMA DE RUTAS DE *m*-VIGILANTES PARA POLÍGONOS DE 1 ESPIRAL RESUELTO CON EL MÉTODO DE PROGRAMACIÓN DINÁMICA

Este problema se define como sigue. Se tienen un polígono simple y un entero $m \geq 1$ y se requiere encontrar las rutas para m-vigilantes, denominadas las rutas de m-vigilan-

tes, de modo que todo punto del polígono sea visto al menos por un vigilante desde alguna posición en su ruta. El objetivo es minimizar la suma de las longitudes de las rutas. Se ha demostrado que el problema de rutas para *m*-vigilantes para polígonos de 1 espiral es NP-difícil.

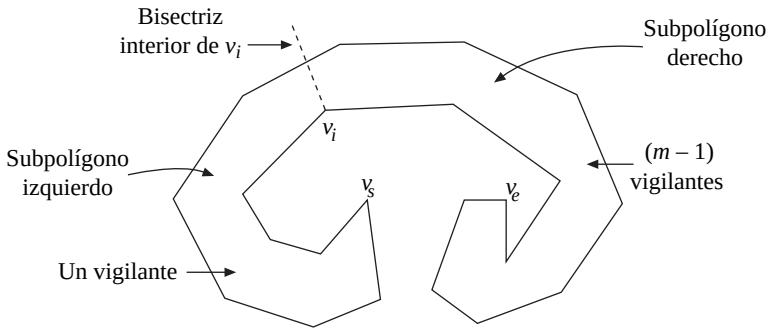
En esta sección se demostrará que el problema de rutas de *m*-vigilantes para polígonos de 1 espiral puede resolverse con el método de programación dinámica. Recuerde que en el capítulo 3 los polígonos de 1 espiral se definieron como un polígono simple cuya frontera puede separarse en una cadena reflex y una cadena convexa. Al atravesar la frontera de un polígono de 1 espiral, los vértices inicial y final de la cadena reflex se denominan v_s y v_e, respectivamente. En la figura 7-45 se muestra una solución del problema de rutas de 3 vigilantes para el polígono de 1 espiral.

FIGURA 7-45 Una solución del problema de rutas de 3 vigilantes para un polígono de 1 espiral.



La idea fundamental es que un polígono de 1 espiral puede separarse en dos partes por medio de una bisectriz interior que parte de un vértice de la cadena reflex. Considere la figura 7-46. Una bisectriz interior que pasa por v_i corta la cadena convexa y separa en dos partes el polígono de 1 espiral, siendo ambas partes de 1 espiral o una de 1 espiral y una convexa. El subpolígono que incluye a v_s se denomina polígono izquierdo y el otro se denomina polígono derecho con respecto a v_i. El vértice v_i se denomina punto de corte. Por conveniencia para el análisis ulterior, la bisectriz interior del primer vértice v_s (el último vértice v_e) en la cadena reflex se define como el primer (último) borde de la cadena convexa.

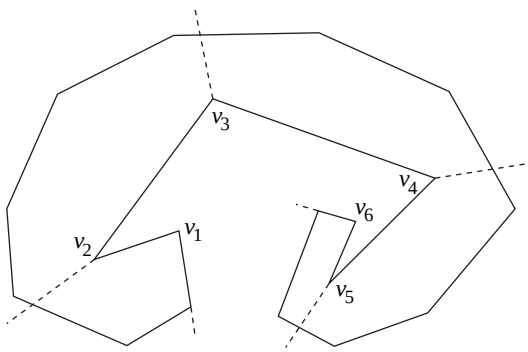
FIGURA 7-46 Idea básica para resolver el problema de rutas de *m*-vigilantes.



Los *m* vigilantes pueden distribuirse en los dos polígonos como sigue: un vigilante para el subpolígono izquierdo y $(m - 1)$ vigilantes para el subpolígono derecho. Se supondrá que se sabe cómo resolver el problema de ruta de 1 vigilante. Entonces el problema de rutas de $(m - 1)$ vigilantes puede resolverse de manera recurrente. Es decir, el subpolígono derecho de nuevo se separa en dos partes y se distribuye un vigilante para la izquierda y $(m - 2)$ vigilantes para la derecha. El problema de rutas de *m* vigilantes puede resolverse al intentar exhaustivamente todos los puntos de corte y escogiendo el que tenga la menor suma de las longitudes de las rutas. A continuación se demostrará por qué este método es de programación dinámica.

Considere la figura 7-47 y suponga que se resolverá un problema de rutas de 3 vigilantes. El método de programación dinámica resuelve este problema de rutas de 3 vigilantes como sigue. Sea $SP(v_i, v_j)$ el subpolígono acotado entre bisectrices interiores de v_i y v_j.

FIGURA 7-47 Polígono de 1 espiral con seis vértices en la cadena reflex.



1. Encontrar las siguientes soluciones de ruta de 1 vigilante.

   Solución 1: La solución de ruta de 1 vigilante para $SP(v_1, v_2)$.
   Solución 2: La solución de ruta de 1 vigilante para $SP(v_1, v_3)$.
   Solución 3: La solución de ruta de 1 vigilante para $SP(v_1, v_4)$.
   Solución 4: La solución de ruta de 1 vigilante para $SP(v_2, v_3)$.
   Solución 5: La solución de ruta de 1 vigilante para $SP(v_2, v_4)$.
   Solución 6: La solución de ruta de 1 vigilante para $SP(v_2, v_5)$.
   Solución 7: La solución de ruta de 1 vigilante para $SP(v_3, v_4)$.
   Solución 8: La solución de ruta de 1 vigilante para $SP(v_3, v_5)$.
   Solución 9: La solución de ruta de 1 vigilante para $SP(v_3, v_6)$.
   Solución 10: La solución de ruta de 1 vigilante para $SP(v_4, v_5)$.
   Solución 11: La solución de ruta de 1 vigilante para $SP(v_4, v_6)$.
   Solución 12: La solución de ruta de 1 vigilante para $SP(v_5, v_6)$.

2. Encontrar las siguientes soluciones de ruta de 2 vigilantes.

   Solución 13: La solución de rutas de 2 vigilantes para $SP(v_2, v_6)$.
      Primero se obtienen las siguientes soluciones:
      Solución 13-1: Combinar la solución 4 y la solución 9.
      Solución 13-2: Combinar la solución 5 y la solución 11.
      Solución 13-3: Combinar la solución 6 y la solución 12.
      Como solución 13, se selecciona la de menor longitud entre la solución 13-1, la solución 13-2 y la solución 13-3.
   Solución 14: La solución de rutas de 2 vigilantes para $SP(v_3, v_6)$.
      Primero se obtienen las siguientes soluciones:
      Solución 14-1: Combinar la solución 7 y la solución 11.
      Solución 14-2: Combinar la solución 8 y la solución 12.
      Como solución 14, se selecciona la de menor longitud entre la solución 14-1 y la solución 14-2.
   Solución 15: La solución de rutas de 2 vigilantes para $SP(v_4, v_6)$.
   Ésta puede obtenerse al combinar la solución 10 y la solución 12.

3. Encontrar la solución de ruta de 3 vigilantes del problema original determinando las siguientes soluciones:

   Solución 16-1: Combinar la solución 1 y la solución 13.
   Solución 16-2: Combinar la solución 2 y la solución 14.
   Solución 16-3: Combinar la solución 3 y la solución 15.

Se selecciona la de menor longitud entre la solución 16-1, la solución 16-2 o la solución 16-3.

El análisis muestra que el problema de rutas de *m* vigilantes puede resolverse con el método de programación dinámica. Observe que el espíritu del método de programación dinámica es que empieza resolviendo problemas básicos y gradualmente resuelve problemas cada vez más complicados al combinar los subproblemas ya resueltos. Primero se resuelven todos los problemas relevantes de rutas de 1 vigilante, luego los problemas de rutas de 2 vigilantes y así sucesivamente.

Sea $OWR_k(v_i, v_j)$ la longitud de las rutas óptimas de *k* vigilantes para el subpolígono $SP(v_i, v_j)$. La ruta óptima de *m* vigilantes $OWR_m(v_s, v_e)$ puede obtenerse con las siguientes fórmulas:

$$OWR_m(v_s, v_e) = \min_{s+1 \le i \le e-1} \{OWR_1(v_s, v_i) + OWR_{m-1}(v_i, v_e)\},$$
$$OWR_k(v_i, v_e) = \min_{i+1 \le j \le e-1} \{OWR_1(v_i, v_j) + OWR_{k-1}(v_j, v_e)\},$$
para $2 \le k \le m - 1, s + 1 \le i \le e - 1$.

Así como para el problema de ruta de 1 vigilante, aquí tampoco se abordarán los detalles porque el objetivo principal es sólo demostrar que este problema puede resolverse con el método de programación dinámica. En la figura 7-48 se muestra una solución típica de un problema de ruta de 1 vigilante, y en la figura 7-49 se muestran casos especiales.

FIGURA 7-48 Problema típico de ruta de 1 vigilante p, v_a, $C[v_a, v_b]$, v_b, r_1 en un polígono de 1 espiral.

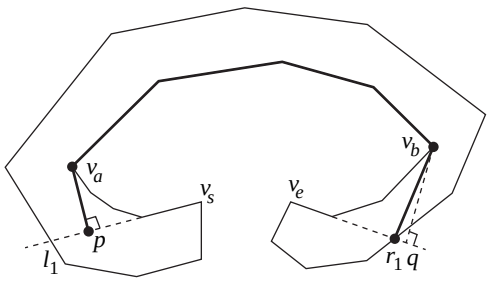
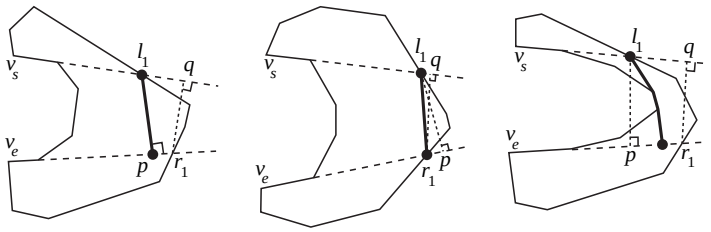


FIGURA 7-49 Casos especiales del problema de ruta de 1 vigilante en un polígono de 1 espiral.



a) Uno de los dos, $\overline{l_1 p}$ o $\overline{r_1 q}$ está dentro del polígono.

b) Ambos p o q están fuera del polígono.

c) Uno de los dos, $\overline{l_1 p}$ o $\overline{r_1 q}$ corta la cadena reflex.

7-10 LOS RESULTADOS EXPERIMENTALES

Para demostrar la potencia del método de programación dinámica, éste se implementó para resolver en una computadora el problema de la subsecuencia común más larga. En la misma computadora también se programó el método directo. Los resultados experimentales se muestran en la tabla 7-7. Los resultados indican claramente la superioridad del método de programación dinámica.

TABLA 7-7 Resultados experimentales.

Longitud de la cadena	Tiempo de CPU en milisegundos	
	Programación dinámica	Método exhaustivo
4	< 1	20
6	< 2	172
8	< 2	2 204
10	< 10	32 952
12	< 14	493 456

7-11 NOTAS Y REFERENCIAS

Puede afirmarse que el término programación dinámica fue acuñado por Bellman (1962). Muchos autores han escrito libros sobre este tema: Nemhauser (1966); Dreyfus y Law (1977), y Denardo (1982).

En 1962, Bellman realizó una revisión de la aplicación de la programación dinámica para resolver problemas combinatorios (1962). El planteamiento del método de programación dinámica para el problema de asignación de recursos y de programación apareció en Lawler y Moore (1969); Sahni (1976), y Horowitz y Sahni (1978). El planteamiento de programación dinámica para el problema del agente viajero se debe a Held y Karp (1962) y Bellman (1962). La aplicación de la programación dinámica al problema de la subsecuencia común más larga fue propuesta por Hirschberg (1975). El método de programación dinámica para resolver el problema 0/1 de la mochila puede encontrarse en Nemhauser y Ullman (1969) y Horowitz y Sahni (1974). La construcción de árboles de búsqueda binarios óptimos usando el método de programación dinámica puede consultarse en Gilbert y Moore (1959), Knuth (1971) y Knuth (1973). La aplicación del método de programación dinámica al problema ponderado del conjunto de dominación perfecta en árboles se encuentra en Yen y Lee (1990), y la aplicación para resolver el problema de búsqueda en un solo paso puede consultarse en Hsiao, Tang y Chang (1993). El algoritmo de alineación de 2 secuencias puede encontrarse en Neddleman y Wunsch (1970), mientras que el algoritmo para la predicción de la estructura secundaria en ARN en Waterman y Smith (1978).

Otras aplicaciones importantes de la programación dinámica, que no se mencionan en este libro, incluyen multiplicación matricial en cadena: Goodbole (1973); Hu y Shing (1982), y Hu y Shing (1984); todas las rutas más cortas: Floyd (1962); análisis sintáctico de lenguajes libre de contexto: Younger (1967), y/o gráficas en serie-paralelo: Simon y Lee (1971) y decodificación de Viterbi: Viterbi (1967) y Omura (1969).

La programación dinámica puede usarse junto con la técnica branch-and-bound. Consulte la obra de Morin y Marsten (1976). También puede aplicarse para resolver un tipo especial de problema de partición. Consulte la sección 4.2 del libro de Garey y Johnson (1979). Este concepto fue usado por Hsu y Nemhauser (1979).

7-12 BIBLIOGRAFÍA ADICIONAL

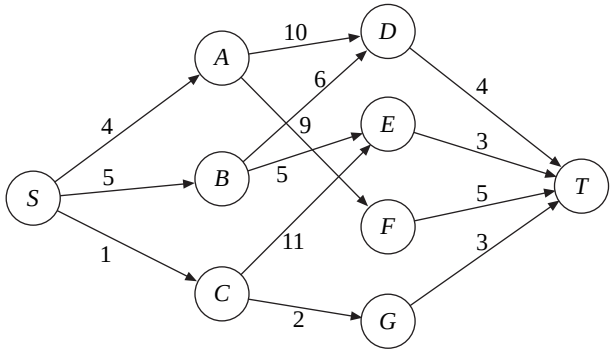
Debido a que la programación dinámica es tan sencilla e ingeniosa, durante mucho tiempo ha constituido un tema de investigación para teóricos y para prácticos en ciencias de computación. Para investigación adicional, recomendamos los artículos siguientes: Akiyoshi y Takeaki (1997); Auletta, Parente y Persiano (1996); Baker (1994);

Bodlaender (1993); Chen, Kuo y Sheu (1988); Chung, Makedon, Sudborough y Turner (1985); Even, Itai y Shamir (1976); Farber y Keil (1985); Fonlupt y Nachef (1993); Gotlieb (1981); Gotlieb y Wood (1981); Hirschberg y Larmore (1987); Horowitz y Sahni (1974); Huo y Chang (1994); Johnson y Burrus (1983); Kantabutra (1994); Kao y Queyranne (1982); Kilpelainen y Mannila (1995); Kryazhimskiy y Savinov (1995); Liang (1994); Meijer y Rappaport (1992); Morin y Marsten (1976); Ozden (1988); Park (1991); Peng, Stephens y Yesha (1993); Perl (1984); Pevzner (1992); Rosenthal (1982); Sekhon (1982); Tidball y Atman (1996); Tsai y Hsu (1993); Tsai y Lee (1997); Yannakakis (1985); Yen y Lee (1990); Yen y Lee (1994), y Yen y Tang (1995).

A continuación se presenta una lista de resultados nuevos e interesantes: Aho, Ganapathi y Tjang (1989); Akutsu (1996); Alpert y Kahng (1995); Amini, Weymouth y Jain (1990); Baker y Giancarlo (2002); Bandelloni, Tucci y Rinaldi (1994); Barbu (1991); Brown y Whitney (1994); Charalambous (1997); Chen, Chern y Jang (1990); Cormen (1999); Culberson y Rudnicki (1989); Delcoigne y Hansen (1975); Eppstein, Galil, Giancarlo e Italiano (1990); Eppstein, Galil, Giancarlo e Italiano (1992(a)); Eppstein, Galil, Giancarlo e Italiano (1992(b)); Erdmann (1993); Farach y Thorup (1997); Fischel-Ghodsian, Mathiowitz y Smith (1990); Geiger, Gupta, Costa y Vlontzos (1995); Gelfand y Roytberg (1993); Galil y Park (1992); Hanson (1991); Haussmann y Suo (1995); Hein (1989); Hell, Shamir y Sharan (2001); Hirosawa, Hoshida, Ishikawa y Toya (1993); Holmes y Durbin (1998); Huang, Liu y Viswanathan (1994); Huang y Waterman (1992); Ibaraki y Nakamura (1994); Karoui y Quenez (1995); Klein (1995); Kostreva y Wiecek (1993); Lewandowski, Condon y Bach (1996); Liao y Shoemaker (1991); Lin, Fan y Lee (1993); Lin, Chen, Jiang y Wen (2002); Littman, Cassandra y Kaelbling (1996); Martin y Talley (1995); Merlet y Zerubia (1996); Miller y Teng (1999); Mohamed y Gader (1996); Moor (1994); Motta y Rampazzo (1996); Myoupo (1992); Ney (1984); Ney (1991); Nuyts, Suetens, Oosterlinck, Roo y Mortelmans (1991); Ohta y Kanade (1985); Ouyang y Shahidehpour (1992); Pearson y Miller (1992); Rivas y Eddy (1999); Sakoe y Chiba (1978); Schmidt (1998); Snyder y Stormo (1993); Sutton (1990); Tataru (1992); Tatman y Shachter (1990); Tatsuya (2000); Vintsyuk (1968); Von Haeselerm, Blum, Simpson, Strum y Waterman (1992); Waterman y Smith (1986); Wu (1996); Xu (1990), y Zuker (1989).

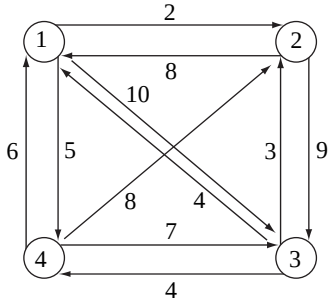
Ejercicios

7.1 Considere la gráfica siguiente. Aplique el método de programación dinámica y encuentre la ruta más corta de *S* a *T*.



7.2 Para la gráfica que se muestra en la figura 7-1, resuelva el mismo problema aplicando el método branch-and-bound. Para este problema, ¿cuál método es mejor (programación dinámica o branch-and-bound)? ¿Por qué?

7.3 Para la gráfica que se muestra a continuación, resuelva el problema del agente viajero aplicando el método branch-and-bound. Compárelo con el método de programación dinámica.



	1	2	3	4
1	∞	2	10	5
2	8	∞	9	∞
3	4	3	∞	4
4	6	8	7	∞

7.4 Para la tabla siguiente, encuentre una asignación de recursos óptima que maximice la ganancia total para estos tres proyectos y cuatro recursos.

recurso / proyecto	1	2	3	4
1	3	7	10	12
2	1	2	6	9
3	2	4	8	9

7.5 Aplique el método de programación dinámica para resolver el siguiente problema de programación lineal.

Maximizar $x_0 = 8x_1 + 7x_2$
sujeta a
$2x_1 + x_2 \leq 8$
$5x_1 + 2x_2 \leq 15$
donde x_1 y x_2 son enteros no negativos.

7.6 Encuentre una subsecuencia común más larga de

$S_1 = a\ a\ b\ c\ d\ a\ e\ f$

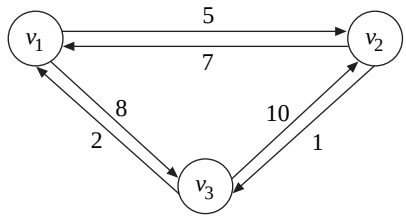
y $S_2 = b\ e\ a\ d\ f$.

7.7 En general, el problema de partición es NP-completo. Sin embargo, bajo algunas restricciones, un tipo especial del problema de partición es un problema polinomial porque puede resolverse con programación dinámica. Consulte la sección 4-2 del libro de Garey y Johnson (1979).

7.8 Encuentre un árbol binario óptimo para $a_1, a_2, \ldots, a_6$, si los identificadores, en orden, tienen probabilidades 0.2, 0.1, 0.15, 0.2, 0.3 y 0.05, respectivamente y todos los demás identificadores tienen probabilidad cero.

7.9 Considere la gráfica siguiente. Resuelva el problema de las rutas más cortas de todos los pares de la gráfica. Este problema consiste en encontrar la ruta más corta que hay entre cada par de vértices. Consulte la

sección 5-3 del libro de Horowitz y Sahni (1978) o la sección 5-4 de la obra de Brassard y Bratley (1988).



7.10 Sean f una función real de x y $y = (y_1, y_2, \ldots, y_k)$. Se dice que f puede descomponerse en f_1 y f_2 si f es separable ($f(x, y) = f_1(x), f_2(y)$) y si, además, la función es monótona no decreciente con respecto a su segundo argumento. Demuestre que si f puede descomponerse con $f(x, y) = (f_1(x), f_2(y))$, entonces

$$\underset{(x,y)}{Ópt}\{f(x, y)\} = Ópt\{f_1(x, Ópt\{f_2(y)\})\} \quad (Ópt = \text{mín o máx})$$

(Consulte la sección 9-2 de [Minoux 1986].)

7.11 El algoritmo de Floyd, que puede encontrarse fácilmente en muchos libros de texto, es para encontrar las rutas más cortas de todos los pares en una gráfica ponderada. Proporcione un ejemplo para explicar el algoritmo.

7.12 Escriba un algoritmo de programación dinámica que resuelva el problema de la subsecuencia creciente más larga.

7.13 Dadas dos secuencias S_1 y S_2 en un conjunto alfabeto Σ y una función de puntaje f: $\Sigma \times \Sigma \rightarrow \Re$, el problema de asignación local consiste en encontrar una subsecuencia S_1' de S_1 y una subsecuencia S_2' de S_2 tales que el puntaje obtenido al alinear S_1' y S_2' sea el más alto, de entre todas las posibles subsecuencias de S_1 y S_2. Aplique la estrategia de programación dinámica para diseñar un algoritmo de tiempo $O(nm)$ o mejor para resolver este problema, donde n y m denotan las longitudes de S_1 y S_2, respectivamente.

capítulo

8

Teoría de los problemas NP-completos

Quizás esta teoría sea una de las más interesantes en ciencias de la computación. El investigador más importante en este campo, el profesor S. A. Cook de la Universidad de Toronto fue galardonado con el Premio Turing por sus aportes en esta área de investigación. Sin duda, la teoría de los problemas NP-completos constituye una de las teorías más apasionantes y desconcertantes entre los numerosos hallazgos del campo de las ciencias de la computación. El teorema principal, ahora denominado teorema de Cook, es tal vez el más citado.

En este libro no sólo se presentará la aplicación del teorema de Cook, sino que también se intentará explicar cuál es su significado.

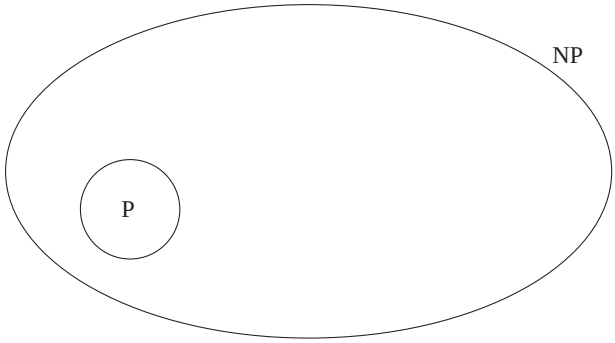
8-1 Análisis informal de la teoría de los problemas NP-completos

La importancia de esta teoría reside en que identifica un amplio tipo de problemas difíciles. Por éstos entendemos aquellos cuya cota inferior parece ser del orden de una función exponencial. En otras palabras, la teoría de los problemas NP-completos ha identificado un extenso tipo de problemas, que en apariencia carecen de algoritmos de tiempo polinomial, con objeto de resolverlos.

En términos generales, se puede decir que la teoría de los problemas NP-completos primeramente apunta a los numerosos problemas NP (polinomial no-determinístico). (La definición formal de NP aparece en el apartado 8-4.) No todos los NP son difíciles; muchos son sencillos. Por ejemplo, el problema de búsqueda es un sencillo problema NP. Es posible resolverlo por medio de algoritmos cuyas complejidades temporales son polinomiales. Otro ejemplo es el problema del árbol de expansión mínimo. De nuevo, este problema se resuelve mediante un algoritmo polinomial. Todos ellos reciben la etiqueta de problemas P (polinomiales).

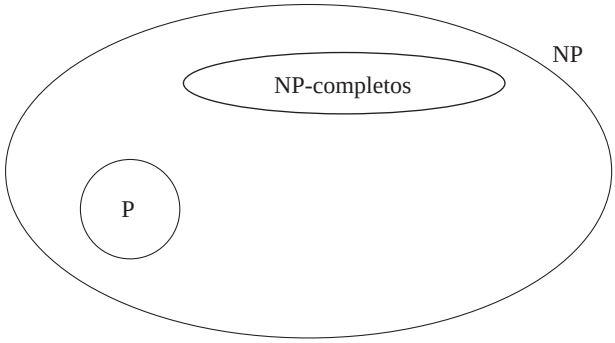
Como el conjunto de problemas NP contiene numerosos problemas P, puede trazarse una figura que muestra su relación, como en la figura 8-1.

FIGURA 8-1 Conjunto de problemas NP.



Además, se ha demostrado que existe también una extensa clase de problemas NP-completos dentro del conjunto de problemas NP, lo cual se muestra en la figura 8-2.

FIGURA 8-2 Problemas NP que incluyen tanto problemas P como problemas NP-completos.



El conjunto de problemas NP-completos conocidos es muy grande y sigue aumentando. Incluye muchos problemas famosos, como el problema de satisfactibilidad, el problema del agente viajero y el problema del empaque en contenedores (bin packing). Todos estos problemas poseen una característica común: *hasta ahora, en el peor caso, ningún problema NP-completo puede resolverse con un algoritmo polinomial. En otras palabras, hasta la fecha, en el peor caso, el mejor algoritmo para resolver cualquier problema NP-completo tiene complejidad exponencial*. Es importante recalcar que la teoría de los problemas NP-completos siempre se refiere a los peores casos.

Es posible que un problema NP-completo sea resuelto aplicando un algoritmo con complejidad temporal polinomial en su caso promedio. En el resto del capítulo, siem-

pre que se hable de complejidad temporal, se entiende que es en el peor caso, a menos que específicamente se indique lo contrario.

Identificar un conjunto de problemas, que hasta la fecha no pueden resolverse con ningún algoritmo polinomial, no es muy interesante. Por la definición de la teoría de problemas NP-completos, se cumple lo siguiente:

> *Si un problema NP-completo puede resolverse en tiempo polinomial, entonces todos los problemas NP pueden resolverse en tiempo polinomial. O bien, si cualquier problema NP-completo puede resolverse en tiempo polinomial, entonces* $NP = P$.

Así que, la teoría de los problemas NP-completos indica que todo problema NP-completo es semejante a un pilar decisivo. Si se derrumba un pilar, todo el edificio se viene abajo. O bien, para plantearlo de otra forma, un problema NP-completo es como un general: si se rinde ante el enemigo, todo su ejército también se rinde.

Debido a que es muy improbable que todos los problemas NP puedan resolverse con algoritmos polinomiales, en consecuencia es muy improbable que cualquier problema NP-completo pueda resolverse con cualquier algoritmo polinomial.

Conviene recalcar que la teoría de los problemas NP-completos no sostiene que éstos jamás puedan resolverse con algoritmos polinomiales. Simplemente afirma que todos los problemas NP-completos pueden resolverse en un número polinomial de pasos. En cierto modo, desalienta el intento de encontrar algoritmos polinomiales para resolver dichos problemas.

8-2 LOS PROBLEMAS DE DECISIÓN

La mayor parte de los problemas considerados en este libro pueden dividirse en dos categorías: problemas de optimización y problemas de decisión.

Considere el problema del agente viajero, que se define así: dado un conjunto de puntos, encontrar el recorrido más corto que empiece en cualquier punto v_0. Resulta evidente que este problema es un problema de optimización.

Un problema de decisión es un problema cuya solución es simplemente "sí" o "no". Para el problema del agente viajero hay un problema de decisión correspondiente que se define a continuación: dado un conjunto de puntos, ¿existe algún recorrido, empezando en cualquier punto v_0, cuya longitud total sea menor que una constante c dada?

Observe que el problema del agente viajero es más difícil que el problema de decisión del agente viajero. Si es posible resolver el problema del agente viajero, entonces se sabrá que el recorrido más corto es igual a algún valor, por ejemplo, a. Si $a < c$, entonces la respuesta al problema de decisión del agente viajero es "sí"; en caso contrario, "no". Por lo tanto, puede afirmarse que *si es posible resolver el problema del*

agente viajero, entonces se puede resolver el problema de decisión del agente viajero, pero no al contrario. En consecuencia, se concluye que el problema del agente viajero es más difícil que el problema de decisión del agente viajero.

A continuación se considerará otro ejemplo: el problema 0/1 de la mochila que se presentó en el capítulo 5 y se define como sigue:

Dados M, W_i y P_i, $W_i > 0$, $P_i > 0$, $1 \leq i \leq n$, $M > 0$, encontrar las x_i tales que $x_i = 1$ o 0, y $\sum_{i=1}^{n} P_i x_i$ se maximice, sujeta a $\sum_{i=1}^{n} W_i x_i \leq M$.

Resulta evidente que este problema 0/1 de la mochila es un problema de optimización. También tiene un problema de decisión correspondiente que se define de la siguiente manera:

Dados los M, R, W_i y P_i, $M > 0$, $R > 0$, $W_i > 0$, $P_i > 0$, $1 \leq i \leq n$, determinar si existen los x_i, $x_i = 1$ o 0, tales que $\sum_{i=1}^{n} P_i x_i \geq R$ y $\sum_{i=1}^{n} W_i x_i \leq M$.

Puede demostrarse fácilmente que el problema 0/1 de la mochila es más difícil que el problema de decisión 0/1 de la mochila.

En general, los problemas de optimización son más difíciles de resolver que sus problemas correspondientes de decisión. En consecuencia, hasta donde tiene que ver la cuestión de si un problema puede resolverse con un algoritmo polinomial, simplemente pueden considerarse sólo problemas de decisión. Si el problema de decisión del agente viajero no puede resolverse con algoritmos polinomiales, se concluye que el problema del agente viajero tampoco. *Al estudiar problemas NP, sólo se analizarán problemas de decisión.*

En la siguiente sección se abordará el problema de satisfactibilidad, que es uno de los problemas de decisión más famosos.

8-3 EL PROBLEMA DE SATISFACTIBILIDAD

Éste es un problema importante porque fue el primer problema NP-completo que se descubrió.

Se considerará la siguiente fórmula lógica:

$$\begin{aligned} & x_1 \vee x_2 \vee x_3 \\ \& \quad & -x_1 \\ \& \quad & -x_2. \end{aligned}$$

La siguiente asignación hace verdadera la fórmula

$x_1 \leftarrow F$
$x_2 \leftarrow F$
$x_3 \leftarrow T.$

En seguida, se usará la notación $(-x_1, -x_2, x_3)$ para representar $\{x_1 \leftarrow F, x_2 \leftarrow F, x_3 \leftarrow T\}$. *Si una asignación hace verdadera una fórmula, se dirá que esta asignación satisface la fórmula; en caso contrario, no la satisface.*

Si por lo menos hay una asignación que satisface una fórmula, entonces se dice que esta fórmula es satisfactible; en caso contrario, es insatisfactible.

Una típica fórmula insatisfactible es

$$\begin{array}{ll} & x_1 \\ \& & -x_1. \end{array}$$

Otra fórmula insatisfactible es

$$\begin{array}{lllll} & x_1 & \vee & x_2 \\ \& & x_1 & \vee & -x_2 \\ \& & -x_1 & \vee & x_2 \\ \& & -x_1 & \vee & -x_2. \end{array}$$

El problema de satisfactibilidad se define como sigue: dada una fórmula booleana, determinar si esta fórmula es satisfactible o no.

Más adelante en este apartado se analizarán algunos métodos para resolver el problema de satisfactibilidad. Primero se necesitan algunas definiciones.

Definición

Una literal es x_i o $-x_i$, donde x_i es una variable booleana.

Definición

Una cláusula es una disyunción de literales. Se entiende que ninguna cláusula contiene simultáneamente una literal y su negación.

Definición

Una fórmula está en su forma normal conjuntiva si está en forma de c_1 & c_2 & … & c_m donde cada c_i, $1 \leq i \leq m$, es una cláusula.

Es bien conocido que toda fórmula booleana puede transformarse en la forma normal conjuntiva. En consecuencia, se supone que todas las fórmulas ya están en forma normal conjuntiva.

Definición

Una fórmula G es una consecuencia lógica de una fórmula F si y sólo si siempre que F es verdadera, G es verdadera. En otras palabras, toda asignación que satisface a F también satisface a G.

Por ejemplo,

$$\begin{aligned} & -x_1 \vee x_2 && \textbf{(1)} \\ \& \quad & x_1 && \textbf{(2)} \\ \& \quad & x_3 && \textbf{(3)} \end{aligned}$$

es una fórmula en forma normal conjuntiva. La única asignación que satisface la fórmula anterior es (x_1, x_2, x_3). El lector puede percatarse fácilmente de que la fórmula x_2 es una consecuencia lógica de la fórmula anterior. Dadas dos cláusulas

$$\begin{aligned} & c_1\colon\ L_1 \vee L_2 \ldots \vee L_j \\ \text{y} \quad & c_2\colon\ -L_1 \vee L'_2 \ldots \vee L'_k, \end{aligned}$$

puede deducirse una cláusula

$$L_2 \vee \ldots \vee L_j \vee L'_2 \vee \ldots \vee L'_k$$

como consecuencia lógica de c_1 & c_2 si la cláusula

$$L_2 \vee \ldots \vee L_j \vee L'_2 \vee \ldots \vee L'_k$$

no contiene ningún par de literales que sean complementarias entre sí.

Por ejemplo, considere las cláusulas siguientes:

c_1: $-x_1 \vee x_2$
c_2: $x_1 \vee x_3$.

Entonces

c_3: $x_2 \vee x_3$

es una consecuencia lógica de c_1 & c_2.

La regla de inferencia anterior se denomina *principio de resolución*, y la cláusula c_3 generada al aplicar este principio de resolución a c_1 y c_2 se denomina *resolvente* de c_1 y c_2.

Se considerará otro ejemplo:

c_1: $-x_1 \vee -x_2 \vee x_3$
c_2: $x_1 \vee x_4$.

Entonces

c_3: $-x_2 \vee x_3 \vee x_4$

es una resolvente de c_1 y c_2. También es, por supuesto, una consecuencia lógica de c_1 & c_2.

Considere las dos cláusulas siguientes:

c_1: x_1
c_2: $-x_1$.

Entonces la resolvente es una cláusula especial porque no contiene ninguna literal y se denota por

$c_3 = \square$

que es una cláusula vacía.

Si de un conjunto de cláusulas es posible deducir una cláusula vacía, entonces este conjunto de cláusulas debe ser insatisfactible. Considere el siguiente conjunto de cláusulas:

$$x_1 \vee x_2 \quad \textbf{(1)}$$
$$x_1 \vee -x_2 \quad \textbf{(2)}$$
$$-x_1 \vee x_2 \quad \textbf{(3)}$$
$$-x_1 \vee -x_2. \quad \textbf{(4)}$$

Una cláusula vacía puede obtenerse como sigue:

(1) & (2)	x_1	**(5)**
(3) & (4)	$-x_1$	**(6)**
(5) & (6)	$\square$.	**(7)**

Debido a que (7) es una cláusula vacía, puede concluirse que (1) & (2) & (3) & (4) es insatisfactible.

Dado un conjunto de cláusulas, el principio de resolución puede aplicarse de manera repetida para deducir nuevas cláusulas. Éstas se agregan al conjunto de cláusulas original y se les aplica de nuevo el principio de resolución. Este proceso termina si se genera una cláusula vacía o ya no es posible deducir nuevas cláusulas. *Si se deduce una cláusula vacía, este conjunto de cláusulas es insatisfactible. Si no es posible deducir nuevas cláusulas cuando se termina el proceso, este conjunto de cláusulas es satisfactible*.

Se considerará un conjunto de cláusulas satisfactible:

	$-x_1 \vee -x_2 \vee x_3$	**(1)**
	x_1	**(2)**
	x_2	**(3)**
(1) & (2)	$-x_2 \vee x_3$	**(4)**
(4) & (3)	x_3	**(5)**
(1) & (3)	$-x_1 \vee x_3$.	**(6)**

Ahora se observa que ya no es posible deducir ninguna cláusula nueva a partir de las cláusulas (1) a (6). En consecuencia, puede concluirse que este conjunto de cláusulas es satisfactible.

Si el conjunto de cláusulas anterior se modifica al agregarle $-x_3$, se obtiene un conjunto de cláusulas insatisfactible:

	$-x_1 \vee -x_2 \vee x_3$	**(1)**
	x_1	**(2)**
	x_2	**(3)**
	$-x_3$	**(4)**
(1) & (2)	$-x_2 \vee x_3$	**(5)**
(5) & (4)	$-x_2$	**(6)**
(6) & (3)	$\square$.	**(7)**

La propiedad de ser insatisfactible puede establecerse ahora porque se ha deducido una cláusula vacía.

Para adentrarse en análisis más teóricos del principio de resolución, consulte algún libro sobre demostración mecánica de teoremas.

En el análisis anterior se demostró que el problema de satisfactibilidad puede considerarse como un problema de deducción. En otras palabras, de manera constante se está buscando una inconsistencia. Se concluye que el conjunto de cláusulas es insatisfactible si es posible deducir una inconsistencia, y satisfactible en caso contrario. Nuestro método parece no tener ninguna relación con encontrar asignaciones que cumplan la fórmula. En realidad, a continuación se demostrará que el método de deducción (o inferencia) es equivalente al método de determinación de asignaciones. Es decir, la deducción de una cláusula vacía en realidad es equivalente al fracaso en encontrar alguna asignación que cumpla todas las cláusulas. Por el contrario, el fracaso en deducir una cláusula vacía equivale a encontrar, por lo menos, una asignación que cumpla todas las cláusulas.

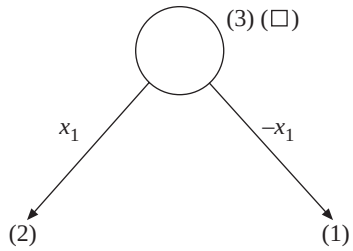
Se empezará con un ejemplo muy sencillo:

x_1 **(1)**

$-x_1$. **(2)**

Debido a que el conjunto de cláusulas proporcionado sólo contiene una variable; a saber, x_1, es posible comenzar a construir un árbol semántico, que se muestra en la figura 8-3.

FIGURA 8-3 Árbol semántico.



La rama izquierda significa que la asignación contiene a x_1 (lo cual significa $x_1 \leftarrow T$) y la rama derecha significa que la asignación contiene a $-x_1$ (lo cual significa $x_1 \leftarrow F$. Puede verse que la asignación izquierda; a saber, x_1, hace falsa la cláusula 2. En consecuencia, esta rama se termina con (2). De manera semejante, la rama derecha se termina con (1). En la figura 8-3 se indica que la cláusula 1 debe contener a x_1 y que la cláusula 2 debe contener a $-x_1$. A estas cláusulas se aplica el principio de resolución y se deduce una nueva cláusula, vacía, como sigue:

(1) & (2) □. **(3)**

La ecuación (3) puede colocarse junto al nodo padre, como se muestra en la figura 8-3. Considere el siguiente conjunto de cláusulas:

$-x_1 \vee -x_2 \vee x_3$ **(1)**

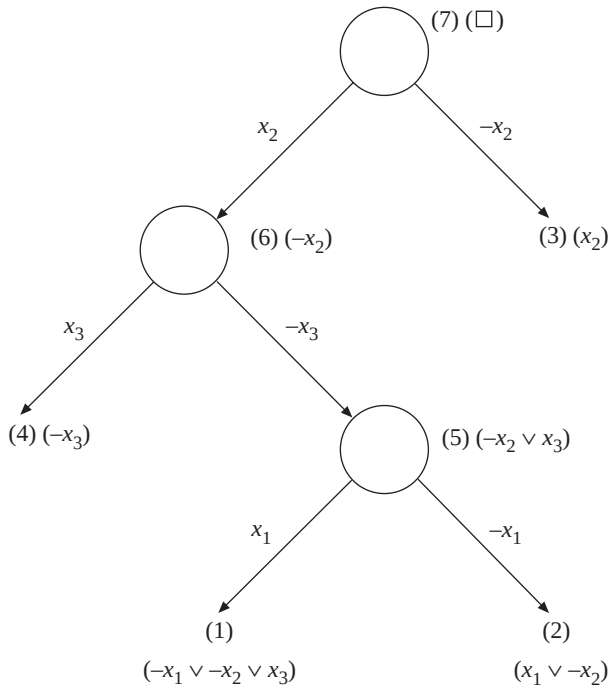
$x_1 \vee -x_2$ **(2)**

x_2 **(3)**

$-x_3$. **(4)**

Es posible construir el árbol semántico que se observa en la figura 8-4. En esa figura, cada ruta de la raíz del árbol a un nodo terminal representa una clase de asignaciones. Por ejemplo, cada asignación debe contener a x_2 o a $-x_2$. La primera rama derecha se identifica como $-x_2$. Esto representa a todas las asignaciones que contienen a $-x_2$ (hay cuatro asignaciones así). Debido a que la cláusula (3) sólo contiene a x_2, es falsada por toda asignación que contenga a $-x_2$. Así, la primera rama derecha es terminada por (3).

FIGURA 8-4 Árbol semántico.



Considere la ruta que contiene a x_2, $-x_3$ y x_1. Esta asignación hace falsa la cláusula (1). De manera semejante, la ruta que contiene a x_2, $-x_3$ y $-x_1$ representa una asignación que hace falsa la cláusula (2).

Considere los nodos terminales indicados por (1) y (2) respectivamente. Debido a que las ramas que conducen a ellos están identificadas por x_1 y $-x_1$, respectivamente, la cláusula (1) debe contener a $-x_1$ y la cláusula (2) debe contener x_1. Cuando el principio de resolución se aplica a las cláusulas (1) y (2), puede deducirse la cláusula $-x_2 \vee x_3$. Ésta puede identificarse como cláusula (5) y asociarse con el nodo padre, como se muestra en la figura 8-4. Siguiendo el mismo razonamiento, el principio de resolución puede aplicarse a las cláusulas (5) y (4) para obtener la cláusula (6). Las cláusulas (6) y (3) son contradictorias entre sí, por lo que se deduce una cláusula vacía. Todo esto se muestra en la figura 8-4. Todo el proceso de deducción se detalla como sigue:

(1) & (2) $-x_2 \vee x_3$ **(5)**

(4) & (5) $-x_2$ **(6)**

(6) & (3) $\square$. **(7)**

En general, dado un conjunto de cláusulas que representa una fórmula booleana, un árbol semántico puede construirse según las reglas siguientes:

1. A partir de cada nodo interno del árbol semántico hay dos ramas que se bifurcan hacia fuera. Una se identifica por x_i y la otra, por $-x_i$, donde x_i es una variable que aparece en el conjunto de cláusulas.
2. Un nodo es eliminado tan pronto como la asignación correspondiente a las literales, que aparecen en la ruta que va de la raíz del árbol a este nodo, hace falsa una cláusula (j) en el conjunto. Este nodo se señala como un nodo terminal y la cláusula (j) se asocia con este nodo terminal.
3. Ninguna ruta en el árbol semántico puede contener un par complementario, de modo que cada asignación es consistente.

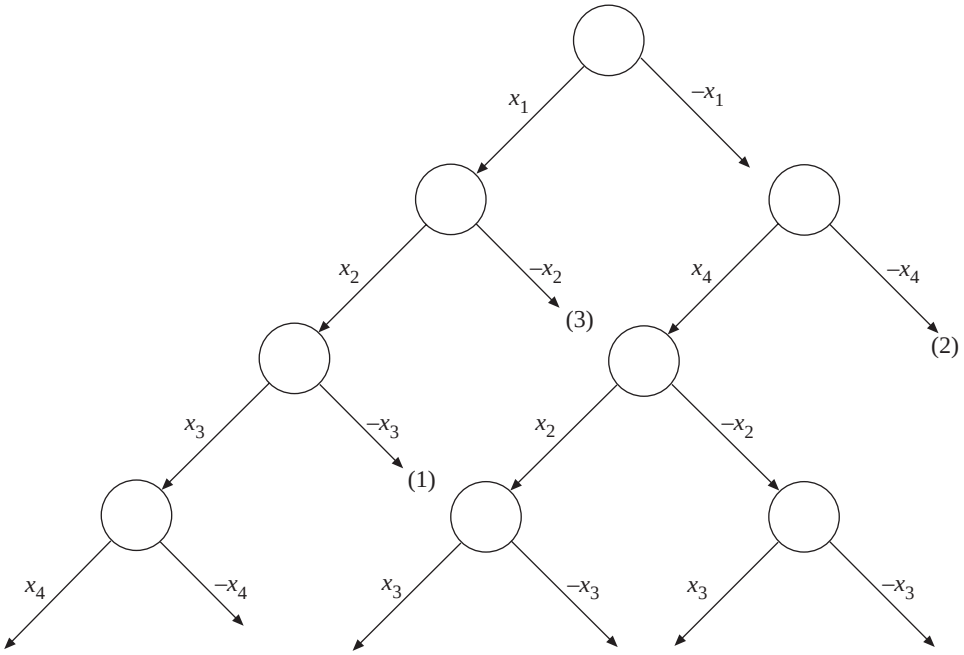
Resulta evidente que todo árbol semántico es finito. *Si cada terminal se asocia con una cláusula, entonces no existe ninguna asignación que cumpla todas las cláusulas. Esto significa que este conjunto de cláusulas es insatisfactible. En caso contrario, existe por lo menos una asignación que satisface todas las cláusulas y este conjunto es satisfactible.*

Considere el siguiente conjunto de cláusulas:

$$-x_1 \vee -x_2 \vee x_3 \quad \textbf{(1)}$$
$$x_1 \vee x_4 \quad \textbf{(2)}$$
$$x_2 \vee -x_1. \quad \textbf{(3)}$$

Es posible construir un árbol semántico como se muestra en la figura 8-5.

FIGURA 8-5 Árbol semántico.



Con base en el árbol semántico anterior, se concluye que este conjunto de cláusulas es satisfactible. Todas las siguientes asignaciones satisfacen la fórmula:

(x_1, x_2, x_3, x_4),
$(x_1, x_2, x_3, -x_4)$,
$(-x_1, -x_2, x_3, x_4)$,
$(-x_1, -x_2, -x_3, x_4)$,
$(-x_1, x_2, x_3, x_4)$,
y $(-x_1, x_2, -x_3, x_4)$.

Si un conjunto de cláusulas es insatisfactible, entonces todo árbol semántico corresponde a la deducción de una cláusula vacía aplicando el principio de resolución. Esta deducción se extrae del árbol semántico como sigue:

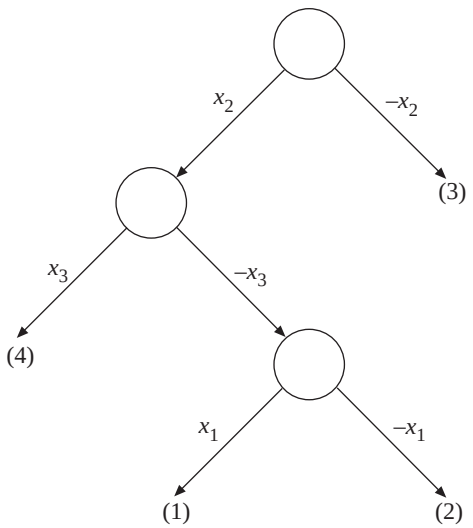
1. Considere un nodo interno cuyos descendientes sean nodos terminales. Sean c_i y c_j, respectivamente, las cláusulas asociadas con ellos. El principio de resolución se aplica a estas dos cláusulas y la resolvente se asocia con este nodo padre. Los nodos descendientes se eliminan completamente. Por lo tanto, el nodo interno original se convierte en un nodo terminal.
2. El paso anterior se repite hasta que el árbol queda vacío y se deduce una cláusula vacía.

La idea anterior se explicará con otro ejemplo. Considere el siguiente conjunto de cláusulas:

$$-x_1 \vee -x_2 \vee x_3 \quad \textbf{(1)}$$
$$x_1 \vee x_3 \quad \textbf{(2)}$$
$$x_2 \quad \textbf{(3)}$$
$$-x_3. \quad \textbf{(4)}$$

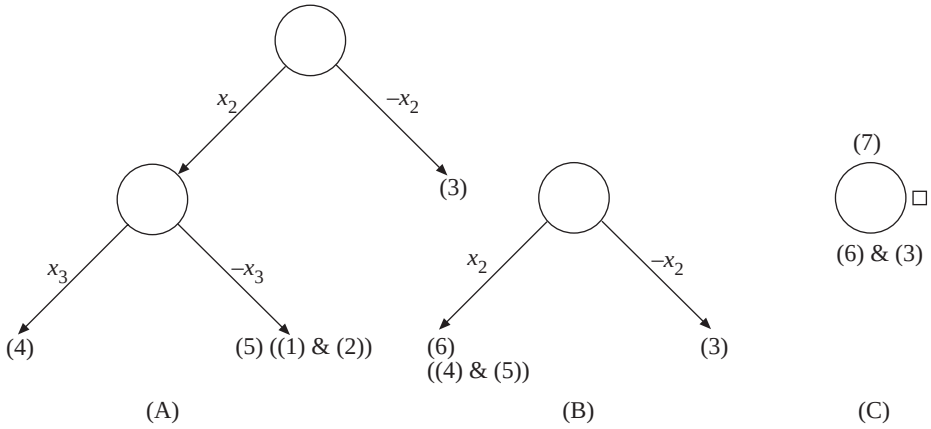
Luego se construye un árbol semántico, que se muestra en la figura 8-6.

FIGURA 8-6 Árbol semántico.



El árbol semántico de la figura 8-6 puede colapsarse gradualmente, lo cual se muestra en la figura 8-7.

FIGURA 8-7 Colapso del árbol semántico de la figura 8-6.



La deducción es como sigue:

(1) & (2) $-x_2 \vee x_3$ **(5)**

(5) & (4) $-x_2$ **(6)**

(6) & (3) □. **(7)**

Lo que se ha demostrado es que incluso cuando se aplica el método de deducción, en realidad se están encontrando asignaciones que satisfacen todas las cláusulas. Si hay n variables, entonces hay 2^n asignaciones posibles. Hasta el momento, para el mejor algoritmo disponible, en los peores casos, es necesario examinar un número exponencial de asignaciones posibles antes de poder hacer cualquier conclusión.

¿Hay alguna posibilidad de resolver el problema de satisfactibilidad en tiempo polinomial? La teoría de los problemas NP-completos no descarta dicha posibilidad. Sin embargo, afirma lo siguiente: *si el problema de satisfactibilidad puede resolverse en un número polinomial de pasos, entonces todos los problemas NP pueden resolverse en un número polinomial de pasos*.

Todavía no se han definido los problemas NP, que se analizarán en el siguiente apartado.

8-4 LOS PROBLEMAS NP

La notación NP significa polinomial no determinístico. Primero se definirá un algoritmo no determinístico como sigue: *un algoritmo no determinístico es un algoritmo que consta de dos fases: suponer y comprobar. Además, se sobreentiende que un algoritmo no determinístico siempre hace una suposición correcta*.

Por ejemplo, dado el problema de satisfactibilidad con una fórmula booleana particular, al principio un algoritmo no determinístico supone una asignación y luego comprueba si esa asignación satisface la fórmula o no. Un concepto importante que debe observarse aquí, es que por medio de suposiciones siempre se obtiene una solución correcta. En otras palabras, si la fórmula es satisfactible, entonces un algoritmo no determinístico siempre hace una suposición correcta y obtiene una asignación que satisface esta fórmula.

Considere el problema de decisión del agente viajero. Un algoritmo no determinístico siempre supone un recorrido y comprueba si es menor que la constante c.

Quizás el lector se sienta escandalizado por esta noción de algoritmo no determinístico porque físicamente es imposible tener tal algoritmo. ¿Cómo puede hacerse una suposición correcta siempre?

En realidad, los algoritmos no determinísticos no existen y nunca existirán. El concepto de algoritmo no determinístico es de utilidad únicamente porque más adelante ayuda a definir una clase de problemas, llamados problemas NP.

Si la complejidad temporal de la etapa de comprobación de un algoritmo no determinístico es polinomial, entonces este algoritmo no determinístico se denomina algoritmo polinomial no-determinístico. Si un problema de decisión puede resolverse con un algoritmo polinomial no-determinístico, entonces este problema se denomina problema polinomial no-determinístico (NP, para abreviar).

Con base en la definición anterior, es posible concluir que *todo problema que puede resolverse en tiempo polinomial (con algoritmos deterministicos, por supuesto) debe ser un problema polinomial no-determinístico*. Casos típicos son los problemas de búsqueda, de fusión, de ordenamiento y del árbol de expansión mínima. Aquí se recuerda al lector que se está hablando de problemas de decisión. La búsqueda es un problema de decisión; evidentemente, el ordenamiento no lo es. Pero siempre es posible crear un problema de decisión a partir del problema de ordenamiento. Este problema consiste en ordenar $a_1, a_2, \ldots, a_n$ de manera creciente o decreciente. Un problema de decisión puede construirse como sigue: dados $a_1, a_2, \ldots, a_n$ y C, determinar si existe una permutación de a_i $(a_1', a_2', \ldots, a_n')$ tal que $|a_2' - a_1'| + |a_3' - a_2'| + \cdots + |a_n' - a_{n-1}'| < C$. Todos los problemas que pueden resolverse en tiempo polinomial se denominan problemas P.

El problema de satisfactibilidad y el problema de decisión del agente viajero son problemas NP porque la complejidad temporal de la etapa de comprobación de los dos problemas es polinomial. De hecho, la mayor parte de los problemas resolubles en que puede pensarse son problemas NP.

Un famoso problema de decisión que no es NP es el problema de detención, que se define como sigue: dado un programa arbitrario con datos de entrada arbitrarios, ¿el programa termina o no? Otro problema es el problema de satisfactibilidad del cálculo de predicados de primer orden. Estos dos problemas se denominan problemas indecidibles.

Los problemas indecidibles no pueden resolverse por suposición y comprobación. Aunque son problemas de decisión, de alguna manera no es posible resolverlos mediante el análisis exhaustivo del espacio solución. El lector debe observar que en lógica booleana (también conocida como lógica proposicional o simbólica), una asignación es caracterizada por una eneada. Pero, para el cálculo de predicados de primer orden, una asignación no está acotada. Puede ser de longitud infinita. Debido a ello, el cálculo de predicados de primer orden no es un problema NP. Basta recordar al lector que los problemas indecidibles son aún más difíciles que los problemas NP.

A continuación se especificará la situación. Para el problema de satisfactibilidad y el problema de decisión del agente viajero, el número de soluciones es finito. Para el problema de satisfactibilidad hay 2^n asignaciones posibles y para el problema de decisión del agente viajero hay $(n - 1)!$ recorridos posibles. En consecuencia, aunque estos problemas son difíciles, por lo menos tienen algunas cotas superiores. Por ejemplo, para resolver el problema de satisfactibilidad cuando menos puede aplicarse un algoritmo con complejidad temporal $O(2^n)$.

Sin embargo, para los problemas indecidibles no existe tal cota superior. Puede demostrarse que las cotas superiores nunca existen. Intuitivamente, puede afirmarse que es posible dejar que el programa corra un millón de años, por ejemplo, y aun así no es posible extraer ninguna conclusión porque puede que dicho programa se detenga en el paso siguiente. De manera semejante, para el problema de satisfactibilidad del cálculo de predicados de primer orden, se tiene la misma situación. Suponga que después de ejecutar el programa durante mucho tiempo, aún no se ha producido una cláusula vacía. No obstante, sigue siendo posible que la próxima cláusula que se genere sea vacía.

8-5 EL TEOREMA DE COOK

En este apartado se presenta el teorema de Cook. Sólo se proporciona una demostración informal porque la formal es muy complicada. Este teorema puede plantearse como sigue.

Teorema de Cook

NP = P si y sólo si el problema de satisfactibilidad es un problema P.

La demostración de este teorema consta de dos partes. La primera parte es "Si NP = P, entonces el problema de satisfactibilidad es un problema P". Esta parte es obvia porque el problema de satisfactibilidad es un problema NP. La segunda parte es "Si el problema de satisfactibilidad es un problema P, entonces NP = P". Ésta es una parte crucial del teorema de Cook, que se trabajará en el resto del apartado.

A continuación se explicará la esencia principal del teorema de Cook. Suponga que se tiene un problema NP denominado *A*, el cual es muy difícil de resolver. En vez de resolverlo directamente, se crea otro problema *A′* y al resolver éste se obtiene la solución de *A*. Es importante observar aquí que todo problema es un problema de decisión. Nuestro método es como sigue:

1. *Debido a que el problema **A** es un problema NP, debe existir algún algoritmo **B** NP que resuelva este problema. Un algoritmo NP es un algoritmo polinomial no-determinístico. Es imposible que un algoritmo así exista físicamente, de modo que no puede usarse. Sin embargo, como se verá, **B** puede seguir usándose conceptualmente en los pasos siguientes.**
2. *Se construirá una fórmula booleana **C** correspondiente a **B** tal que **C** sea satisfactible si y sólo si el algoritmo no determinístico **B** termina exitosamente y regresa como respuesta "sí". Si **C** es insatisfactible, entonces el algoritmo **B** termina infructuosamente y regresa como respuesta "no".*

En este momento se observa que cuando se menciona un problema, se entiende una de las instancias de un problema. Es decir, se entiende un problema con una entrada particular. En caso contrario, no puede afirmarse que el algoritmo termina.

Este apartado trata sobre los elementos esenciales del teorema de Cook. Más adelante se hará el análisis sobre cómo se construye *C*.

3. *Después de construir la fórmula **C**, temporalmente nos olvidaremos del problema original **A** y del algoritmo no determinístico **B**. Se intentará determinar si **C** es satisfactible o no. En caso de ser satisfactible, entonces se dice que la respuesta del problema **A** es "sí"; en caso contrario, la respuesta es "no". Esto puede hacerse debido a la propiedad de la fórmula **C** planteada en el paso (2). Es decir, **C** es satisfactible si y sólo si **B** termina exitosamente.*

---

* Recuérdese que el algoritmo no determinístico es sólo un concepto abstracto, no una posibilidad real. Se utiliza para el desarrollo y explicación de la teoría, no para la elaboración de programas. *(N del RT)*

El método anterior parece sugerir que sólo debe prestarse atención al problema de satisfactibilidad. Por ejemplo, nunca es necesario saber cómo resolver el problema de decisión del agente viajero; basta saber cómo determinar si la fórmula booleana correspondiente a este problema es satisfactible o no. Sin embargo, ahí hay un problema serio y grande. Si el problema de satisfactibilidad es difícil de resolver, entonces el problema del agente viajero original sigue siendo difícil de resolver. Ésta es la esencia del teorema de Cook. Indica que si el problema de satisfactibilidad puede resolverse en un número polinomial de pasos, entonces todo problema NP puede resolverse en un número polinomial de pasos, esencialmente debido al método anterior.

El lector notará que el método anterior es válido si y sólo si siempre es posible construir una fórmula booleana C a partir de un algoritmo no determinístico B tal que C es satisfactible si y sólo si B termina exitosamente. El hecho de que lo anterior es posible se ilustrará con algunos ejemplos.

Ejemplo 8-1 Una fórmula booleana para el problema de búsqueda (Caso 1)

Considere el problema de búsqueda. Se tiene un conjunto $S = \{x(1), x(2), \ldots, x(n)\}$ de n números y se quiere determinar si en S existe un número que sea igual, por ejemplo, a 7. Para simplificar el análisis, se supone que $n = 2$, $x(1) = 7$ y $x(2) \neq 7$.

El algoritmo no determinístico es como sigue:

$i = choice(1, 2)$
Si $x(i) = 7$, entonces *ÉXITO*
si no, *FRACASO*.

La fórmula booleana correspondiente al algoritmo no determinístico anterior es como sigue:

$$
\begin{array}{llll}
 & i = 1 & \vee & i = 2 \\
\& & i = 1 & \rightarrow & i \neq 2 \\
\& & i = 2 & \rightarrow & i \neq 1 \\
\& & x(1) = 7 \;\&\; i = 1 & \rightarrow & \textit{ÉXITO} \\
\& & x(2) = 7 \;\&\; i = 2 & \rightarrow & \textit{ÉXITO} \\
\& & x(1) \neq 7 \;\&\; i = 1 & \rightarrow & \textit{FRACASO} \\
\& & x(2) \neq 7 \;\&\; i = 2 & \rightarrow & \textit{FRACASO} \\
\& & \textit{FRACASO} & \rightarrow & \textit{ÉXITO}
\end{array}
$$

& *ÉXITO* (garantiza una terminación exitosa)
& $x(1) = 7$ (datos de entrada)
& $x(2) \neq 7$. (datos de entrada)

Para facilitar el análisis, la fórmula anterior se transformará en su forma normal conjuntiva:

$i = 1$	v	$i = 2$				**(1)**
$i \neq 1$	v	$i \neq 2$				**(2)**
$x(1) \neq 7$	v	$i \neq 1$	v	*ÉXITO*		**(3)**
$x(2) \neq 7$	v	$i \neq 2$	v	*ÉXITO*		**(4)**
$x(1) = 7$	v	$i \neq 1$	v	*FRACASO*		**(5)**
$x(2) = 7$	v	$i \neq 2$	v	*FRACASO*		**(6)**
$-$*FRACASO*	v	$-$*ÉXITO*				**(7)**
ÉXITO						**(8)**
$x(1) = 7$						**(9)**
$x(2) \neq 7$.						**(10)**

Las cláusulas del conjunto anterior están conectadas por "&", lo cual se omite aquí. Son satisfactibles en tanto las siguientes asignaciones cumplan todas las cláusulas:

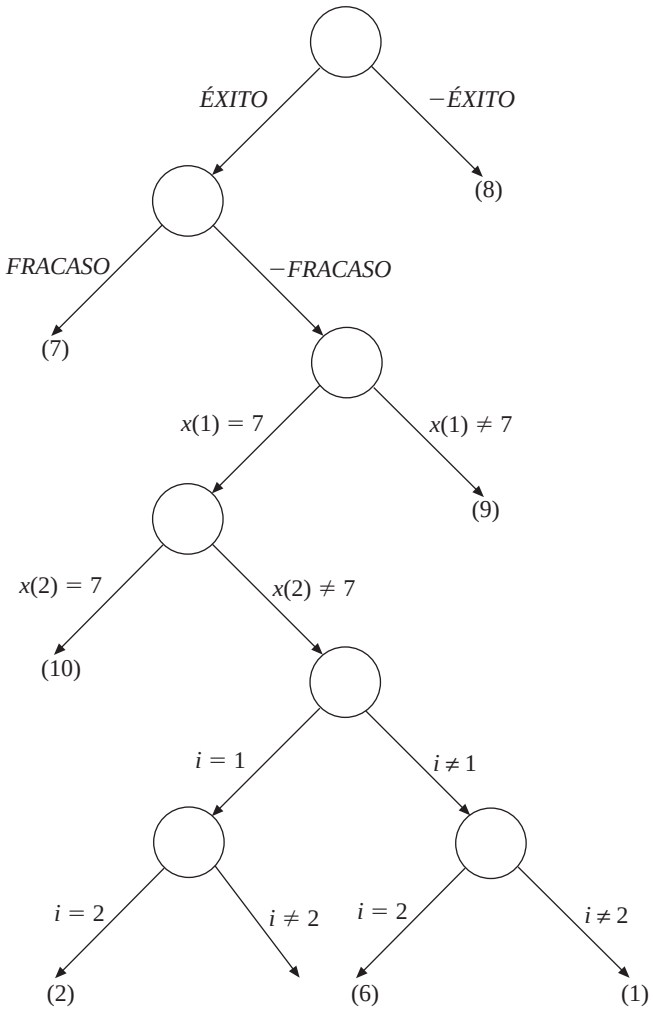
$i = 1$	que satisface a	(1)
$i \neq 2$	que satisface a	(2), (4) y (6)
ÉXITO	que satisface a	(3), (4) y (8)
$-$*FRACASO*	que satisface a	(7)
$x(1) = 7$	que satisface a	(5) y (9)
$x(2) \neq 7$	que satisface a	(4) y (10).

Como puede verse, ahora se satisfacen todas las cláusulas. La asignación anterior que las satisface puede encontrarse al construir un árbol semántico, que se muestra en la figura 8-8.

Con base en el árbol semántico puede verse que la única asignación que satisface a todas las cláusulas es la que ya se había proporcionado.

Se ha demostrado que la fórmula es satisfactible. ¿Por qué es posible afirmar que el algoritmo no determinístico de búsqueda terminará exitosamente? Esto se debe al

FIGURA 8-8 Un árbol semántico.



hecho de que esta fórmula describe la ejecución del algoritmo no determinístico y a que hay una cláusula especial; a saber, *ÉXITO*, que insiste en que se quiere que el algoritmo termine exitosamente.

La demostración de la satisfactibilidad del conjunto de cláusulas anterior no sólo indica que el algoritmo terminará exitosamente y regresará "sí", sino también la razón por la que la respuesta es "sí", que puede encontrarse en la asignación. En la asignación hay una literal,

$$i = 1$$

que constituye nuestra solución. Es decir, no sólo se sabe que la búsqueda será coronada con la respuesta "sí"; también se sabe que la búsqueda tendrá éxito porque $x(1) = 7$.

Aquí se recalca que un problema de búsqueda se ha transformado exitosamente en un problema de satisfactibilidad. De todos modos, no hay que emocionarse por esta transformación, ya que el problema de satisfactibilidad sigue siendo, hasta la fecha, difícil de resolver.

Ejemplo 8-2 Una fórmula booleana para el problema de búsqueda (Caso 2)

En el ejemplo 8-1 se presentó un caso en que un algoritmo no determinístico termina exitosamente. En esta ocasión se presentará un caso en que un algoritmo no determinístico termina infructuosamente. En esta situación, la fórmula booleana correspondiente es insatisfactible.

Como ejemplo seguirá usándose el problema de búsqueda. Para simplificar el análisis, se supone que $n = 2$ y que ninguno de los dos números es igual a 7. La fórmula booleana contiene el siguiente conjunto de cláusulas:

$i = 1$	v	$i = 2$				**(1)**
$i \neq 1$	v	$i \neq 2$				**(2)**
$x(1) \neq 7$	v	$i \neq 1$	v	*ÉXITO*		**(3)**
$x(2) \neq 7$	v	$i \neq 2$	v	*ÉXITO*		**(4)**
$x(1) = 7$	v	$i \neq 1$	v	*FRACASO*		**(5)**
$x(2) = 7$	v	$i \neq 2$	v	*FRACASO*		**(6)**
ÉXITO						**(7)**
$-$*ÉXITO*	v	$-$*FRACASO*				**(8)**
$x(1) \neq 7$						**(9)**
$x(2) \neq 7$.						**(10)**

El conjunto de cláusulas anterior es insatisfactible, lo cual puede demostrarse fácilmente aplicando el principio de resolución.

(9) & (5)	$i \neq 1$	v	*FRACASO*		**(11)**
(10) & (6)	$i \neq 2$	v	*FRACASO*		**(12)**
(7) & (8)	$-$*FRACASO*				**(13)**
(13) & (11)	$i \neq 1$				**(14)**

(13) & (12)	$i \neq 2$	**(15)**
(14) & (1)	$i = 2$	**(16)**
(15) & (16)	$\square$.	**(17)**

Tal vez sea interesante observar que la deducción anterior de una cláusula vacía puede traducirse en una demostración similar al idioma español:

1. Debido a que $x(1) \neq 7$ e $i = 1$ implican *FRACASO* y $x(1)$ en efecto no es igual a 7, se tiene
   $i = 1$ implica *FRACASO*. **(11)**
2. De manera semejante, se tiene
   $i = 2$ implica *FRACASO*. **(12)**
3. Debido a que se insiste en *ÉXITO*, se tiene
   –FRACASO. **(13)**
4. En consecuencia, i no puede ser 1 ni 2. **(14) y (15)**
5. Sin embargo, i es 1 o 2. Si i no es 1, debe ser 2.
6. Se deriva una contradicción.

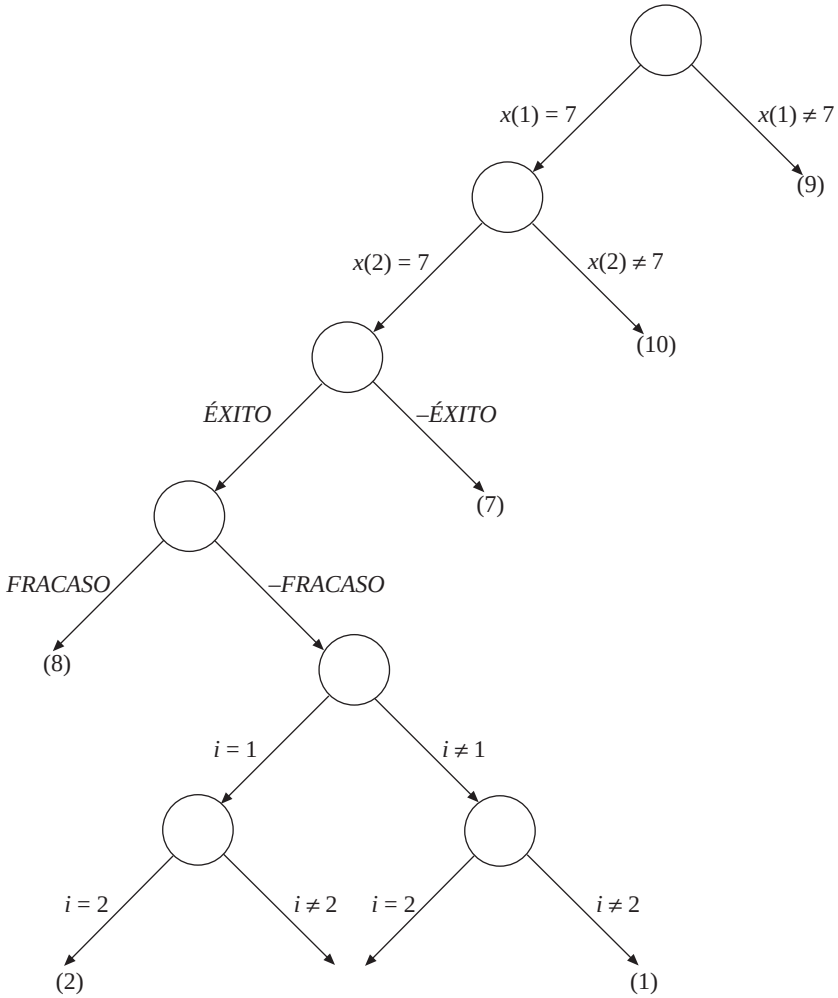
Ejemplo 8-3 Una fórmula booleana para el problema de búsqueda (Caso 3)

Nuevamente se modificará el ejemplo 8-1 de modo que ambos números sean iguales a 7. En este caso, se tiene el siguiente conjunto de cláusulas:

$i = 1$	v	$i = 2$			**(1)**
$i \neq 1$	v	$i \neq 2$			**(2)**
$x(1) \neq 7$	v	$i \neq 1$	v	*ÉXITO*	**(3)**
$x(2) \neq 7$	v	$i \neq 2$	v	*ÉXITO*	**(4)**
$x(1) = 7$	v	$i \neq 1$	v	*FRACASO*	**(5)**
$x(2) = 7$	v	$i \neq 2$	v	*FRACASO*	**(6)**
ÉXITO					**(7)**
$-$*ÉXITO*	v	$-$*FRACASO*			**(8)**
$x(1) = 7$					**(9)**
$x(2) = 7$.					**(10)**

Luego se construye el árbol semántico que se muestra en la figura 8-9.

FIGURA 8-9 Un árbol semántico.



En el árbol semántico anterior puede verse que hay dos asignaciones que satisfacen a todas las cláusulas del conjunto anterior. En una asignación $i = 1$, y en la otra, $i = 2$.

Ejemplo 8-4 Una fórmula booleana para el problema de satisfactibilidad (Caso 1)

En este ejemplo se demostrará que nuestra idea puede aplicarse al problema de satisfactibilidad. Es decir, que para un problema de satisfactibilidad es posible construir una

fórmula booleana de modo que el problema de satisfactibilidad original se resuelve con una respuesta "sí" si y sólo si la fórmula booleana construida es satisfactible.

Se considerará el siguiente conjunto de cláusulas:

$$x_1 \quad \textbf{(1)}$$
$$-x_2. \quad \textbf{(2)}$$

Se intentará determinar si el conjunto de cláusulas anterior es satisfactible o no. Un algoritmo no determinístico para resolver este problema es como sigue:

Hacer $i = 1, 2$
    $x_i = elegir(T, F)$

Si x_1 y x_2 satisfacen las cláusulas 1 y 2, entonces *ÉXITO*; en caso contrario, *FRACASO*.

Se mostrará cómo es posible transformar el algoritmo en una fórmula booleana. Antes que todo, se sabe que para que el algoritmo no determinístico termine con *ÉXITO*, es necesario que las cláusulas 1 y 2 sean verdaderas. Así,

$$\left.\begin{array}{lcll} -ÉXITO & \vee & c_1 = T & \textbf{(1)} \\ -ÉXITO & \vee & c_2 = T & \textbf{(2)} \end{array}\right\} (ÉXITO \rightarrow c_1 = T \;\&\; c_2 = T)$$
$$-c_1 = T \quad \vee \quad x_1 = T \quad \textbf{(3)} \qquad (c_1 = T \rightarrow x_1 = T)$$
$$-c_2 = T \quad \vee \quad x_2 = F \quad \textbf{(4)} \qquad (c_2 = T \rightarrow x_2 = F)$$
$$x_1 = T \quad \vee \quad x_1 = F \quad \textbf{(5)}$$
$$x_2 = T \quad \vee \quad x_2 = F \quad \textbf{(6)}$$
$$x_1 \neq T \quad \vee \quad x_1 \neq F \quad \textbf{(7)}$$
$$x_2 \neq T \quad \vee \quad x_2 \neq F \quad \textbf{(8)}$$
$$ÉXITO. \quad \textbf{(9)}$$

Resulta fácil ver que la siguiente asignación satisface a todas las cláusulas.

$c_1 = T$	que satisface a	(1)
$c_2 = T$	que satisface a	(2)
$x_1 = T$	que satisface a	(3) y (5)
$x_2 = F$	que satisface a	(4) y (6)
$x_1 \neq F$	que satisface a	(7)
$x_2 \neq T$	que satisface a	(8)
ÉXITO	que satisface a	(9).

Así, el conjunto de cláusulas es satisfactible.

Ejemplo 8-5 Una fórmula booleana para el problema de satisfactibilidad (Caso 2)

En el ejemplo 8-4 se demostró que la fórmula construida es satisfactible porque la fórmula original es satisfactible. Si se empieza con un conjunto de cláusulas insatisfactibles, la fórmula correspondiente también debe ser insatisfactible. Este hecho se demostrará con el siguiente ejemplo.

Considere el siguiente conjunto de cláusulas:

$$x_1 \quad (1)$$
$$-x_1. \quad (2)$$

Para el conjunto de cláusulas dado es posible construir la siguiente fórmula booleana:

$$-\textit{ÉXITO} \vee c_1 = T \quad (1)$$
$$-\textit{ÉXITO} \vee c_2 = T \quad (2)$$
$$-c_1 = T \vee x_1 = T \quad (3)$$
$$-c_2 = T \vee x_1 = F \quad (4)$$
$$x_1 = T \vee x_1 = F \quad (5)$$
$$x_1 \neq T \vee x_1 \neq F \quad (6)$$
$$\textit{ÉXITO}. \quad (7)$$

El hecho de que el conjunto de cláusulas anterior es insatisfactible puede demostrarse aplicando el principio de resolución:

(1) & (7) $c_1 = T$ **(8)**
(2) & (7) $c_2 = T$ **(9)**
(8) & (3) $x_1 = T$ **(10)**
(9) & (4) $x_1 = F$ **(11)**
(10) & (6) $x_1 \neq F$ **(12)**
(11) & (12) □. **(13)**

Cuando un algoritmo no determinístico se transforma en una fórmula booleana, debe tenerse cuidado en que la fórmula booleana correspondiente no contenga un número exponencial de cláusulas; en caso contrario, esta transformación carecería de

sentido. Por ejemplo, suponga que un problema de satisfactibilidad que contiene n variables se transforma en un conjunto de cláusulas que contiene 2^n cláusulas. Entonces, el procedimiento de transformación en sí es un proceso exponencial.

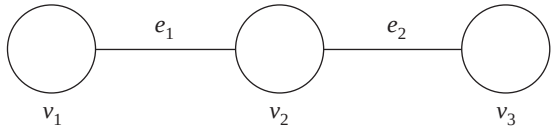
Para recalcar esta cuestión se considerará otro problema (consulte el ejemplo 8-6).

Ejemplo 8-6 El problema de decisión de la cubierta de nodos

Dada una gráfica $G = (V, E)$, un conjunto S de nodos en V se denomina cubierta de nodos de G si toda arista incide en algún nodo en S.

Considere la gráfica en la figura 8-10. Para esta gráfica, $S = \{v_2\}$ es una cubierta de nodos, ya que toda arista incide en el nodo v_2.

FIGURA 8-10 Una gráfica.



El problema de decisión de la cubierta de nodos es el siguiente: dados una gráfica $G = (V, E)$ y un entero positivo k, determinar si existe una cubierta de nodos S de G tal que $|S| \leq k$.

El problema anterior puede resolverse con un algoritmo no determinístico. Sin embargo, este algoritmo no puede sólo intentar todos los subconjuntos de V porque el número de tales subconjuntos es exponencial. No obstante, es posible usar el siguiente algoritmo polinomial no-determinístico: sean $|V| = n$ y $|E| = m$.

```
Begin
    i_1 = elegir ({1, 2, ... , n})
    i_2 = elegir ({1, 2, ... , n} − {i_1})
      ⋮
    i_k = elegir ({1, 2, ... , n} − {i_1, i_2, ... , i_{k−1}}).
    For j ≔ 1 to m do
        Begin
            Si e_j no incide en v_{i_t}(1 ≤ t ≤ k)
            entonces FRACASO; detener
        End
    ÉXITO
End
```

Se considera la figura 8-10 y se supone que $k = 1$. En este caso se tiene el siguiente conjunto de cláusulas, donde $v_i \in e_j$ significa que e_j incide en v_i.

$i_1 = 1$	v	$i_1 = 2$	v	$i_1 = 3$	**(1)**
$i_1 \neq 1$	v	$v_1 \in e_1$	v	*FRACASO*	**(2)**
$i_1 \neq 1$	v	$v_1 \in e_2$	v	*FRACASO*	**(3)**
$i_1 \neq 2$	v	$v_2 \in e_1$	v	*FRACASO*	**(4)**
$i_1 \neq 2$	v	$v_2 \in e_2$	v	*FRACASO*	**(5)**
$i_1 \neq 3$	v	$v_3 \in e_1$	v	*FRACASO*	**(6)**
$i_1 \neq 3$	v	$v_3 \in e_2$	v	*FRACASO*	**(7)**
$v_1 \in e_1$					**(8)**
$v_2 \in e_1$					**(9)**
$v_2 \in e_2$					**(10)**
$v_3 \in e_2$					**(11)**
ÉXITO					**(12)**
$-$*ÉXITO*	v	$-$*FRACASO*.			**(13)**

Puede verse que la siguiente asignación satisface el conjunto de cláusulas anterior:

$i_1 = 2$	que satisface a	(1)
$v_1 \in e_1$	que satisface a	(2) y (8)
$v_2 \in e_1$	que satisface a	(4) y (9)
$v_2 \in e_2$	que satisface a	(5) y (10)
$v_3 \in e_2$	que satisface a	(7) y (11)
ÉXITO	que satisface a	(12)
$-$*FRACASO*	que satisface a	(13)
$i_1 \neq 1$	que satisface a	(3)
$i_1 \neq 3$	que satisface a	(6)

Debido a que el conjunto de cláusulas es satisfactible, se concluye que la respuesta al problema de decisión de la cubierta de nodos es SÍ. La solución consiste en seleccionar el nodo v_2.

Con base en la descripción informal anterior, ahora es posible comprender el significado del teorema de Cook. *Para todo problema* ***A*** *NP, es posible transformar el*

*algoritmo **B** NP correspondiente a este problema NP en una fórmula booleana **C** tal que **C** es satisfactible si y sólo si **B** termina exitosamente y regresa la respuesta "sí". Además, para completar esta transformación se requiere un número polinomial de pasos. Por lo tanto, si es posible determinar la satisfactibilidad de una fórmula booleana **C** en un número polinomial de pasos, definitivamente puede afirmarse que la respuesta al problema **A** es "sí" o "no". O bien, de manera equivalente, es posible decir que si el problema de satisfactibilidad puede resolverse en un número polinomial de pasos, entonces todo problema NP puede resolverse en un número polinomial de pasos. Para plantearlo de otra forma: si el problema de satisfactibilidad está en P, entonces NP = P.*

Es importante observar que el teorema de Cook es válido con una restricción: se requiere un número polinomial de pasos para transformar un problema NP en una fórmula booleana correspondiente. Si para construir esta fórmula booleana se requiere un número exponencial de pasos, entonces no es posible establecer el teorema de Cook.

Otra cuestión importante merece la pena destacar: aunque es posible construir una fórmula booleana que describa el problema original, sigue siendo imposible resolver fácilmente el problema original porque la satisfactibilidad de la fórmula booleana no puede determinarse fácilmente. Observe que al demostrar que una fórmula es satisfactible, se encuentra una asignación que satisface esta fórmula. Este trabajo es equivalente a encontrar una solución del problema original. El algoritmo no determinístico ignora de manera irresponsable el tiempo necesario para encontrar esta solución mientras afirma que siempre hace una suposición correcta. Un algoritmo determinista para resolver el problema de satisfactibilidad no puede ignorar este tiempo necesario para encontrar una asignación. El teorema de Cook indica que si es posible encontrar una asignación que satisface una fórmula booleana en tiempo polinomial, entonces en realidad es posible adivinar una solución en tiempo polinomial. Desafortunadamente, hasta la fecha, no ha sido posible encontrar una asignación en tiempo polinomial. En consecuencia, no es posible hacer una suposición correcta en tiempo polinomial.

El teorema de Cook advierte que de todos los problemas NP, el de satisfactibilidad es el más difícil porque si es posible resolverlo en tiempo polinomial, entonces todos los problemas NP pueden resolverse en tiempo polinomial. Pero, el problema de satisfactibilidad, ¿es el único problema en NP con esta propiedad? Se verá que hay una clase de problemas que son equivalentes entre sí en el sentido de que si cualquiera de ellos puede resolverse en tiempo polinomial, entonces todos los problemas NP pueden resolverse en tiempo polinomial. Estos problemas se denominan clase de problemas NP-completos, que se estudiarán en el siguiente apartado.

8-6 PROBLEMAS NP-COMPLETOS

Definición

Sean A_1 y A_2 dos problemas. A_1 se reduce a A_2 (lo cual se escribe $A_1 \propto A_2$) si y sólo si A_1 puede resolverse en tiempo polinomial usando un algoritmo de tiempo polinomial que resuelve A_2.

Con base en la definición anterior, puede afirmarse que si $A_1 \propto A_2$, y si hay un algoritmo de tiempo polinomial que resuelve A_2, entonces hay un algoritmo de tiempo polinomial que resuelve A_1.

Al usar el teorema de Cook, puede decirse que *todo problema NP se reduce al problema de satisfactibilidad*, ya que este problema NP siempre puede solucionarse al resolver primero el problema de satisfactibilidad de la fórmula booleana correspondiente.

Ejemplo 8-7 El problema de optimización de una eneada

Se considerará el siguiente problema: dado un entero positivo C, $C > 1$ y un entero positivo n. El problema consiste en determinar si existen enteros positivos $c_1, c_2, \ldots, c_n$ tales que $\prod_{i=1}^{n} c_i = C$ y $\sum_{i=1}^{n} c_i$ son minimizados. Este problema se denomina problema de optimización de una eneada.

También se considerará el famoso problema del número primo, que consiste en determinar si un entero positivo C es primo o no. Resulta evidente que la siguiente relación es verdadera: el problema del número primo $\propto$ problema de optimización de una eneada. La razón resulta obvia. Después de resolver el problema de optimización de una eneada, se examina la solución $c_1, c_2, \ldots, c_n$; C es un número primo si y sólo si hay exactamente un c_i distinto de 1 y todos los otros c_i son iguales a 1. Este proceso de revisión sólo requiere n pasos, por lo que se trata de un proceso polinomial. En resumen, si el problema de optimización de una eneada puede resolverse en tiempo polinomial, entonces el problema del número primo puede ser resuelto en tiempo polinomial.

Hasta ahora todavía no es posible resolver el problema de optimización de una eneada con ningún algoritmo de tiempo polinomial.

Ejemplo 8-8 El problema del empaque en contenedores y el problema de asignación de cubetas

Considere el problema de decisión de empaque y el problema de decisión de asignación de cubetas.

El problema de decisión de empaque en contenedores se define como sigue: se tiene un conjunto de n objetos que deben colocarse en B contenedores (o recipientes). La capacidad de cada recipiente es C y cada objeto requiere c_i unidades de capacidad. El problema de decisión de empaque consiste en determinar si estos n objetos pueden dividirse en k grupos, $1 \leq k \leq B$, tales que cada grupo de objetos pueda colocarse en un contenedor.

Por ejemplo, sean $(c_1, c_2, c_3, c_4) = (1, 4, 7, 4)$, $C = 8$ y $B = 2$. Así, los objetos pueden dividirse en dos grupos. Los objetos 1 y 3 en un grupo, y los objetos 2 y 4 en otro grupo.

En cambio, si $(c_1, c_2, c_3, c_4) = (1, 4, 8, 4)$, $C = 8$ y $B = 2$, entonces no hay forma de dividir los objetos en dos grupos o en un grupo de modo que cada grupo de objetos pueda colocarse en un contenedor sin exceder la capacidad de éste.

El problema de decisión de asignación de cubetas se define como sigue: se tienen n registros, todos diferenciados por una clave. Esta clave asume h valores distintos: v_1, v_2, …, v_h y hay n_i registros correspondientes a v_i. Es decir, $n_1 + n_2 + \cdots + n_h = n$. El problema de asignación de cubetas consiste en determinar si es posible colocar estos n registros en k cubetas de modo que registros con el mismo v_i estén en un cubo y ninguna cubeta contenga más de C registros.

Por ejemplo, considere que la clave asume los valores a, b, c y d, $(n_a, n_b, n_c, n_d) = (1, 4, 2, 3)$, $k = 2$ y $C = 5$. Así, los registros pueden colocarse en dos cubetas como sigue:

Cubeta 1	**Cubeta 2**
a	*c*
b	*c*
b	*d*
b	*d*
b	*d*

Si $(n_a, n_b, n_c, n_d) = (2, 4, 2, 2)$, entonces no hay forma de asignar los registros en cubetas sin exceder la capacidad de cada cubeta y mantener los registros con la misma clave en la misma cubeta.

El problema de decisión de asignación de cubetas es un problema interesante. Si los registros se almacenan en discos, entonces el problema de decisión de asignación de cubos está relacionado con el problema de minimizar el número de accesos a los discos. Ciertamente, si se desea minimizar el número total de accesos a los discos, entonces hasta donde sea posible es necesario colocar en la misma cubeta registros que tengan la misma clave.

Puede demostrarse fácilmente que el problema de empaque se reduce al problema de decisión de asignación de cubetas. Para cada problema de empaque es posible crear, en un número polinomial de pasos, un problema de decisión de asignación de cubetas correspondiente. En consecuencia, si existe un algoritmo polinomial para resolver el problema de decisión de asignación de cubetas, entonces el problema de empaque puede resolverse en tiempo polinomial.

Con base en la definición de "se reduce a", resulta fácil ver lo siguiente: *si* $A_1 \propto A_2$ *y* $A_2 \propto A_3$, *entonces* $A_1 \propto A_3$.

Una vez que se ha definido "se reduce a", es posible definir los problemas NP-completos.

Definición

Un problema A es NP-completo si $A \in$ NP y todo problema NP se reduce a A.

Con base en la definición anterior, se sabe que si ***A*** *es un problema NP-completo y* ***A*** *puede resolverse en tiempo polinomial, entonces todo problema NP puede resolverse en tiempo polinomial. Resulta evidente que el problema de satisfactibilidad es un problema NP-completo debido al teorema de Cook.*

Por definición, si cualquier problema NP-completo puede resolverse en tiempo polinomial, entonces NP = P.

El problema de satisfactibilidad fue el primer problema NP-completo que se descubrió. Después, R. Karp mostró 21 problemas NP-completos. Estos problemas NP-completos incluyen la cubierta de vértices (o nodos), la disposición de arco de retroalimentación, el ciclo Hamiltoniano, etc. Karp fue galardonado con el Premio Turing en 1985.

Para demostrar que un problema A es NP-completo, no es necesario demostrar que todos los problemas NP se reducen a A. Esto es lo que hizo Cook cuando demostró la teoría de NP-completo del problema de satisfactibilidad. Actualmente, basta aplicar la propiedad transitiva de "reducir a". *Si* A_1 *es un problema NP-completo,* A_2 *es un problema NP y puede demostrarse que* $A_1 \propto A_2$, *entonces* A_2 *es un problema NP-completo*. El razonamiento es más bien directo. Si A_1 es un problema NP-completo, entonces todos los problemas NP se reducen a A. Si $A \propto B$, entonces todos los problemas NP se reducen a B debido a la propiedad transitiva de "reducir a". En consecuencia, B debe ser NP-completo.

En el análisis anterior se hicieron las siguientes afirmaciones:

1. De todos los problemas NP, el de satisfactibilidad es el más difícil.

2. Cuando se demuestra que un problema es NP-completo, a menudo se intenta demostrar que el problema de satisfactibilidad se reduce a *A*. Así, parece que *A* es más difícil que el problema de satisfactibilidad.

Para ver que no hay inconsistencia en estas afirmaciones, se observa que todo problema NP se reduce al problema de satisfactibilidad. Por lo tanto, si se tiene interés en un problema *A* que es NP, entonces ciertamente *A* se reduce al problema de satisfactibilidad. No obstante, es necesario recalcar aquí que, afirmar que un problema *A* se reduce al problema de satisfactibilidad no es significativo en absoluto porque sólo significa que el problema de satisfactibilidad es más difícil que *A*, lo cual es un hecho bien establecido. Si se ha demostrado exitosamente que el problema de satisfactibilidad se reduce a *A*, entonces *A* es incluso más difícil que el problema de satisfactibilidad, lo cual es una afirmación bastante significativa. Observe que $A \propto$ al problema de satisfactibilidad y que el problema de satisfactibilidad $\propto A$. Por consiguiente, en cuanto concierne al grado de dificultad, *A* es equivalente al problema de satisfactibilidad.

Los argumentos anteriores pueden extenderse a todos los problemas NP-completos. *Si **A** es un problema NP-completo, entonces por definición todo problema NP, por ejemplo **B**, se reduce a **A**. Si además se demuestra que **B** es NP-completo al probar que $\boldsymbol{A} \propto \boldsymbol{B}$, entonces **A** y **B** son equivalentes entre sí. En resumen, todos los problemas NP-completos constituyen una clase de equivalencia.*

Observe que los problemas NP se han restringido a problemas de decisión. Ahora es posible extender el concepto de NP-completo a problemas de optimización al definir "dificultad-NP". *Un problema **A** es NP-difícil (NP-hard) si todo problema NP se reduce a **A***. (*Observe que **A** no necesariamente es un problema NP. De hecho, **A** puede ser un problema de optimización.*) *Así, un problema es NP-completo si **A** es NP-difícil y **A** es NP*. De esta manera, un problema de optimización es NP-difícil si su problema de decisión correspondiente es NP-completo. Por ejemplo, el problema del agente viajero es NP-difícil.

8-7 EJEMPLOS DE DEMOSTRACIÓN DE NP-COMPLETO

En este apartado se mostrará que muchos problemas son NP-completos. Nos gustaría recordar al lector que cuando se desea demostrar que un problema *A* es NP-completo, suele hacerse en dos pasos:

1. Primero se demuestra que *A* es un problema NP.
2. Luego se demuestra que algún problema NP-completo se reduce a *A*.

Muchos lectores cometen el error al demostrar que *A* se reduce a un problema NP-completo. Esto carece de sentido absolutamente, ya que por definición todo problema NP se reduce a todo problema NP-completo.

Se observa que hasta ahora sólo se ha aceptado que el problema de satisfactibilidad es NP-completo. Para producir más problemas NP-completos es necesario empezar con el problema de satisfactibilidad. Es decir, debe intentar demostrarse que el problema de satisfactibilidad se reduce al problema en que se tiene interés.

Ejemplo 8-9 El problema de 3-satisfactibilidad

Este problema es semejante al problema de satisfactibilidad, aunque es más restringido: toda cláusula contiene exactamente tres literales (variables).

Resulta evidente que el problema de 3-satisfactibilidad es un problema NP. Para probar que es un problema NP-completo, es necesario demostrar que el problema de satisfactibilidad se reduce al problema de 3-satisfactibilidad. Se demostrará que *para toda fórmula booleana normal F_1, puede crearse otra fórmula booleana F_2, en la que toda cláusula contiene exactamente tres literales, de modo que F_1 es satisfactible si y sólo si F_2 es satisfactible*.

Se empezará con un ejemplo. Considere el siguiente conjunto de cláusulas:

$$x_1 \quad \vee \quad x_2 \qquad \textbf{(1)}$$
$$-x_1. \qquad \textbf{(2)}$$

El conjunto anterior puede extenderse de modo que cada cláusula contenga tres literales:

$$x_1 \quad \vee \quad x_2 \quad \vee \quad y_1 \qquad \textbf{(1)}'$$
$$-x_1 \quad \vee \quad y_2 \quad \vee \quad y_3. \qquad \textbf{(2)}'$$

Puede verse que (1) & (2) es satisfactible y que (1)′ & (2)′ también es satisfactible. Sin embargo, el método anterior de agregar algunas literales nuevas puede originar problemas, como puede verse en el siguiente caso:

$$x_1 \qquad \textbf{(1)}$$
$$-x_1. \qquad \textbf{(2)}$$

(1) & (2) es insatisfactible. Si a estas dos cláusulas se agregan arbitrariamente algunas literales nuevas:

$$x_1 \quad \vee \quad y_1 \quad \vee \quad y_2 \qquad (1)'$$
$$-x_1 \quad \vee \quad y_3 \quad \vee \quad y_4. \qquad (2)'$$

$(1)'$ & $(2)'$ se convierte en una fórmula satisfactible.

El análisis anterior muestra que no es posible agregar arbitrariamente nuevas literales a una fórmula sin afectar su satisfactibilidad. Lo que puede hacerse es añadir nuevas cláusulas que por sí solas son insatisfactibles respecto al conjunto original de cláusulas. Si el conjunto original de cláusulas es satisfactible, por supuesto que el nuevo conjunto sigue siendo satisfactible. Si el conjunto original de cláusulas es insatisfactible, el nuevo conjunto de cláusulas es insatisfactible.

Si el conjunto original de cláusulas sólo contiene una literal, es posible agregar el siguiente conjunto de cláusulas:

$$y_1 \quad \vee \quad y_2$$
$$-y_1 \quad \vee \quad y_2$$
$$y_1 \quad \vee \quad -y_2$$
$$-y_1 \quad \vee \quad -y_2.$$

Por ejemplo, suponga que la cláusula original es x_1; las cláusulas recientemente creadas son

$$x_1 \quad \vee \quad y_1 \quad \vee \quad y_2$$
$$x_1 \quad \vee \quad -y_1 \quad \vee \quad y_2$$
$$x_1 \quad \vee \quad y_1 \quad \vee \quad -y_2$$
$$x_1 \quad \vee \quad -y_1 \quad \vee \quad -y_2.$$

Si la cláusula original contiene dos literales, puede agregarse

$$y_1$$
$$-y_1.$$

Por ejemplo, suponga que la cláusula original es

$$x_1 \quad \vee \quad x_2.$$

Las cláusulas recientemente creadas son

$$x_1 \quad \vee \quad x_2 \quad \vee \quad y_1$$
$$x_1 \quad \vee \quad x_2 \quad \vee \quad -y_1.$$

Considere el siguiente conjunto de cláusulas insatisfactibles:

$$x_1 \quad \textbf{(1)}$$
$$-x_1. \quad \textbf{(2)}$$

Ahora se tiene

$$\begin{array}{ccccc} x_1 & \vee & y_1 & \vee & y_2 \\ x_1 & \vee & -y_1 & \vee & y_2 \\ x_1 & \vee & y_1 & \vee & -y_2 \\ x_1 & \vee & -y_1 & \vee & -y_2 \\ -x_1 & \vee & y_3 & \vee & y_4 \\ -x_1 & \vee & -y_3 & \vee & y_4 \\ -x_1 & \vee & y_3 & \vee & -y_4 \\ -x_1 & \vee & -y_3 & \vee & -y_4. \end{array}$$

Este nuevo conjunto de cláusulas sigue siendo insatisfactible.

Si una cláusula contiene más de tres literales, esta cláusula puede separarse en un conjunto de nuevas cláusulas agregando el siguiente conjunto de cláusulas insatisfactibles:

$$\begin{array}{ccc} y_1 & & \\ -y_1 & \vee & y_2 \\ -y_2 & \vee & y_3 \\ & \vdots & \\ -y_{i-1} & \vee & y_i \\ -y_i. & & \end{array}$$

Considere la siguiente cláusula:

$$x_1 \vee -x_2 \vee x_3 \vee x_4 \vee -x_5.$$

Es posible agregar nuevas variables para obtener nuevas cláusulas que contienen exactamente tres literales:

$$\begin{array}{ccccc} x_1 & \vee & -x_2 & \vee & y_1 \\ x_3 & \vee & -y_1 & \vee & y_2 \\ x_4 & \vee & -x_5 & \vee & -y_2. \end{array}$$

Las reglas para transformar una cláusula en un conjunto de cláusulas que contienen exactamente tres literales pueden resumirse como sigue. (En lo que sigue, todas las y_i representan nuevas variables.)

1. Si la cláusula contiene exactamente una literal L_1, entonces se obtienen las cuatro cláusulas siguientes:

$$\begin{array}{lllll} L_1 & \vee & y_1 & \vee & y_2 \\ L_1 & \vee & -y_1 & \vee & y_2 \\ L_1 & \vee & y_1 & \vee & -y_2 \\ L_1 & \vee & -y_1 & \vee & -y_2. \end{array}$$

2. Si la cláusula contiene dos literales L_1 y L_2, entonces se obtienen las dos cláusulas siguientes:

$$\begin{array}{lllll} L_1 & \vee & L_2 & \vee & y_1 \\ L_1 & \vee & L_2 & \vee & -y_1. \end{array}$$

3. Si la cláusula contiene tres literales, no se hace nada.
4. Si la cláusula contiene más de tres literales, se obtienen nuevas cláusulas como sigue: sean las literales $L_1, L_2, \ldots, L_k$. Las nuevas cláusulas son

$$\begin{array}{lllll} L_1 & \vee & L_2 & \vee & y_1 \\ L_3 & \vee & -y_1 & \vee & y_2 \\ \vdots & & & & \\ L_{k-1} & \vee & L_k & \vee & -y_{k-3}. \end{array}$$

Considere el siguiente conjunto de cláusulas:

$$\begin{array}{lllllllll} x_1 & \vee & x_2 & & & & & & \\ -x_3 & & & & & & & & \\ x_1 & \vee & -x_2 & \vee & x_3 & \vee & -x_4 & \vee & x_5. \end{array}$$

Luego se obtiene el nuevo conjunto de cláusulas:

$$\begin{array}{lllll} x_1 & \vee & x_2 & \vee & y_1 \\ x_1 & \vee & x_2 & \vee & -y_1 \\ -x_3 & \vee & y_2 & \vee & y_3 \\ -x_3 & \vee & -y_2 & \vee & y_3 \\ -x_3 & \vee & y_2 & \vee & -y_3 \\ -x_3 & \vee & -y_2 & \vee & -y_3 \\ x_1 & \vee & -x_2 & \vee & y_4 \\ x_3 & \vee & -y_4 & \vee & y_5 \\ -x_4 & \vee & x_5 & \vee & -y_5. \end{array}$$

Como ya se indicó, la transformación anterior debe preservar la propiedad de satisfactibilidad del conjunto de cláusulas original. Es decir, sea S el conjunto de cláusulas original. Sea S' el conjunto de cláusulas transformadas, donde cada cláusula contiene tres literales. Entonces S' es satisfactible si y sólo si S es satisfactible. Esto se demostrará a continuación.

1. Parte 1: si S es satisfactible, entonces S' es satisfactible. Sea I una asignación que satisface a S. Entonces, resulta evidente que I satisface a todas las cláusulas obtenidas a partir de cláusulas que contienen no más de tres literales. Sean C una cláusula en S que contiene más de tres literales y $T(C)$ el conjunto de cláusulas en S' relacionadas con C. Por ejemplo, para

$$C = x_1 \vee x_2 \vee -x_3 \vee \ x_4 \vee -x_5$$

$$T(C) = \begin{cases} x_1 \vee x_2 \vee y_1 \\ -x_3 \vee -y_1 \vee y_2 \\ x_4 \vee -x_5 \vee -y_2. \end{cases}$$

I satisface a C, como se supuso. Si I también satisface a $T(C)$, ya se ha terminado. En caso contrario, I debe satisfacer por lo menos a un subconjunto de $T(C)$. A continuación se explicará cómo I puede ampliarse a I' de modo que I' cumpla todas las cláusulas en $T(C)$. Esto puede explicarse como sigue: sea C_i una cláusula en $T(C)$ que es satisfecha por I. A la última literal de la cláusula C_i se asigna el valor falsa. Esta asignación satisface otra cláusula C_j en $T(C)$. Si se satisface toda cláusula en $T(C)$, ya se ha terminado. En caso contrario, a la última literal de la cláusula C_j se asigna el valor falsa. Este proceso puede repetirse hasta que se satisfacen todas las cláusulas.

Considere $C = x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5$.

$T(C)$ es:

$$x_1 \vee x_2 \vee y_1 \quad \textbf{(1)}$$
$$x_3 \vee -y_1 \vee y_2 \quad \textbf{(2)}$$
$$x_4 \vee x_5 \vee -y_2. \quad \textbf{(3)}$$

Suponga que $I = \{x_1\}$. I satisface a (1). A y_1 de (1) se asigna el valor de falsa. Así, también se satisfacen $I' = \{x_1, -y_1\}$ y (2). Luego, a y_2 de (2) se asigna el valor de falsa. Finalmente, esto hace que se cumpla (3). Entonces, $I' = \{x_1, -y_1, -y_2\}$ satisface todas las cláusulas.

2. Parte 2: si S' es satisfactible, entonces S es satisfactible. Observe que para S', las nuevas cláusulas añadidas son insatisfactibles por sí mismas. En consecuencia, si S' es satisfactible, la asignación que satisface a S' no puede contener sólo las y_i. Debe asignar valores de verdad a algunas x_i, las variables originales. Considere un conjunto S_j de cláusulas en S' que se obtienen de manera correspondiente a una cláusula c_j en S. Debido a que cualquier cláusula que satisface a S_j debe satisfacer por lo menos a una literal de C_j, esta asignación hace falsa a C_j. En consecuencia, si S' es satisfactible, entonces S es satisfactible.

Debido a que se requiere tiempo polinomial para transformar un conjunto de cláusulas arbitrario S en un conjunto de cláusulas S' donde cada cláusula contiene tres literales, y como S' es satisfactible si y sólo si S es satisfactible, se concluye que si es posible resolver el problema de 3-satisfactibilidad en tiempo polinomial, entonces el problema de satisfactibilidad puede resolverse en tiempo polinomial. Así, el problema de satisfactibilidad se reduce al problema de 3-satisfactibilidad y éste es NP-completo.

En el ejemplo anterior se demostró que el problema de 3-satisfactibilidad es NP-completo. Quizá muchos lectores estén algo desconcertados en este momento debido al siguiente razonamiento erróneo: el problema de 3-satisfactibilidad es un caso especial del problema de satisfactibilidad. Debido a que el problema de satisfactibilidad es un problema NP-completo, entonces el problema de 3-satisfactibilidad debe ser automáticamente un problema NP-completo.

Observe que el grado de dificultad de un caso especial de un problema general no puede deducirse examinando el problema general. Considere el problema de satisfactibilidad. Es NP-completo, aunque es completamente posible que una versión especial del problema de satisfactibilidad no sea NP-completo. Considere el caso en que cada cláusula sólo contiene literales positivas. Es decir, no aparece ningún signo negativo. Un ejemplo típico es el siguiente:

$$x_1 \vee x_2 \vee x_3$$
$$x_1 \vee x_4$$
$$x_4 \vee x_5$$
$$x_6.$$

En este caso, es posible obtener fácilmente una asignación satisfactoria al asignar el valor verdadero a cada variable. Así, esta clase especial de problemas de satisfactibilidad no es NP-completa.

En general, si un problema es NP-completo, sus casos especiales pueden ser NP-completos, o no. Por otra parte, si un caso especial de un problema es NP-completo, entonces este problema es NP-completo.

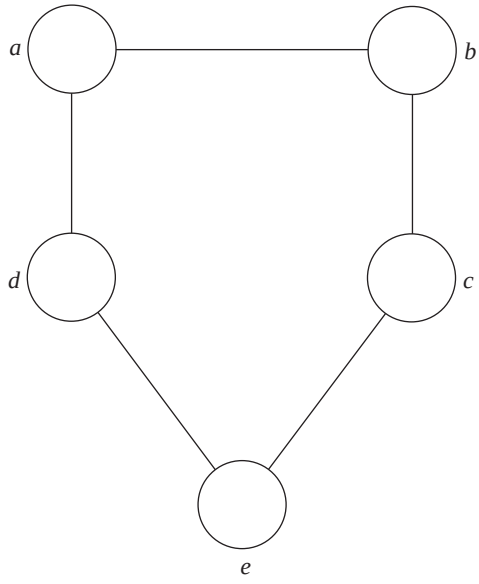
Ejemplo 8-10 El problema de decisión de coloración de una gráfica

En este ejemplo se demostrará que el problema de decisión de coloración de una gráfica es NP-completo. Este problema se define como sigue: se tiene una gráfica $G = (V, E)$. A cada vértice se asocia un color de modo que si dos vértices están unidos por una arista, entonces a estos dos vértices deben asociarse colores diferentes. El problema de decisión de coloración de una gráfica consiste en determinar si para colorear sus vértices pueden usarse k colores.

Considere la figura 8-11. En esta figura, para colorear la gráfica pueden usarse tres colores como sigue:

$a \leftarrow 1,\ b \leftarrow 2,\ c \leftarrow 1,\ d \leftarrow 2,\ e \leftarrow 3.$

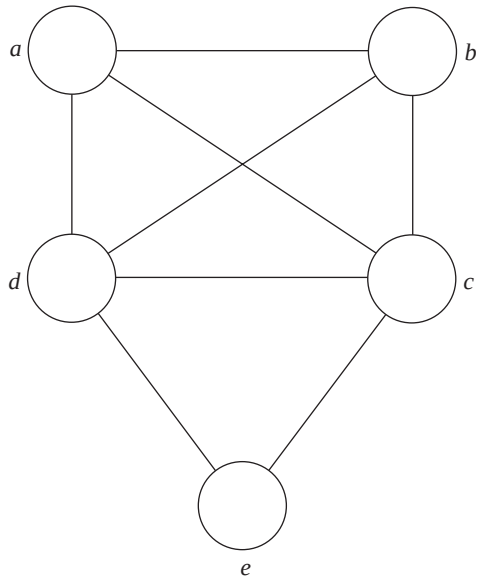
FIGURA 8-11 Una gráfica 3-coloreable.



Considere la figura 8-12. Resulta fácil ver que para colorear la gráfica se requieren cuatro colores.

Para demostrar que el problema de decisión de coloración de una gráfica es NP-completo, se requiere un problema NP-completo y demostrar que este problema NP-completo se reduce al problema de decisión de coloración de una gráfica. En este caso, se usará un problema semejante al problema de 3-satisfactibilidad analizado en el ejemplo 8-9. Este problema es un problema de satisfactibilidad donde cada cláusula

FIGURA 8-12 Una gráfica 4-coloración.



contiene cuando mucho tres literales. Debido al análisis en el ejemplo 8-9, es fácil darse cuenta que este problema de satisfactibilidad con tres literales, a lo sumo por cláusula, también es NP-completo. Para todo problema de satisfactibilidad común, siempre es posible transformarlo en un problema de satisfactibilidad cuando mucho con tres literales por cláusula sin afectar la satisfactibilidad original; basta separar las cláusulas con más de tres literales en cláusulas con exactamente tres literales.

A continuación se demostrará que el problema de satisfactibilidad con a lo más tres literales por cláusula se reduce al problema de decisión de coloración de una gráfica. Básicamente, se demostrará que para todo problema de satisfactibilidad con a lo más tres literales por cláusula es posible construir una gráfica correspondiente de modo que la fórmula booleana original es satisfactible si y sólo si la gráfica construida puede colorearse usando $n + 1$ colores, donde n es el número de variables que aparecen en la fórmula booleana.

Sean $x_1, x_2, \ldots, x_n$ las variables en la fórmula booleana F, donde $n \geq 4$. Si $n < 4$, entonces n es una constante y el problema de satisfactibilidad puede determinarse fácilmente. Sean $C_1, C_2, \ldots, C_r$ las cláusulas donde cada cláusula contiene a lo más tres literales.

La gráfica G correspondiente a la fórmula booleana se construye según las reglas siguientes:

1. Los vértices de la gráfica G son $x_1, x_2, \ldots, x_n, -x_1, -x_2, \ldots, -x_n, y_1, y_2, \ldots, y_n, C_1, C_2, \ldots, C_r$.
2. Las aristas de la gráfica G se forman siguiendo las reglas siguientes:
   a) Hay una arista entre toda x_i y toda $-x_i$, para $1 \leq i \leq n$.
   b) Hay una arista entre toda y_i y toda y_j, si $i \neq j$, $1 \leq i, j \leq n$.
   c) Hay una arista entre toda y_i y toda x_j, si $i \neq j$, $1 \leq i, j \leq n$.
   d) Hay una arista entre toda y_i y toda $-x_j$, si $i \neq j$, $1 \leq i, j \leq n$.
   e) Hay una arista entre toda x_i y toda C_j, si $x_i \notin C_j$, $1 \leq i \leq n$, $1 \leq j \leq r$.
   f) Hay una arista entre toda $-x_i$ y toda C_j, si $-x_i \notin C_j$, $1 \leq i \leq n$, $1 \leq j \leq r$.

Se demostrará que F es satisfactible si y sólo si G es $n + 1$ coloreable. La demostración consta de dos partes:

1. Si F es satisfactible, entonces G es $n + 1$ coloreable.
2. Si G es $n + 1$ coloreable, entonces F es satisfactible.

Se demostrará la primera parte. Ahora se supone que F es satisfactible. En este caso, puede escogerse cualquier asignación A que cumpla a F y colorear los vértices como sigue:

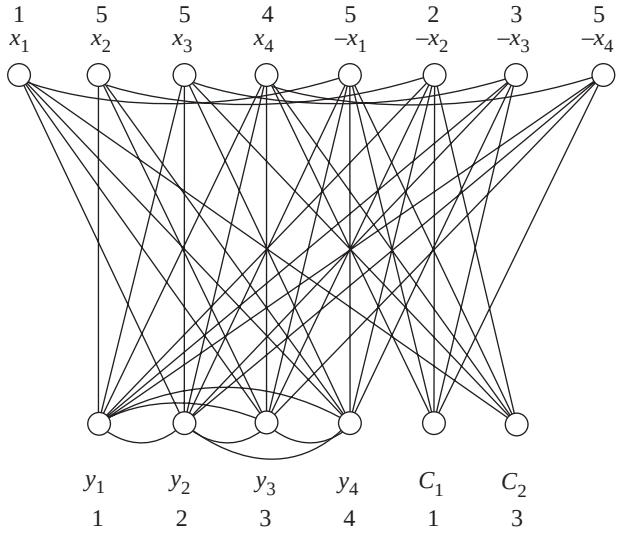
1. Para todas las y_i, y_i se colorea con el color i.
2. Para todas las x_i y las $-x_i$, a x_i y a $-x_i$ los colores se asignan como sigue: si a x_i en A se asigna V, entonces x_i se colorea con el color i y $-x_i$ se colorea con $n + 1$; en caso contrario, x_i se colorea con el color $n + 1$ y $-x_i$ se colorea con i.
3. Para toda C_j, se encuentra una literal L_i en C_j que es verdadera en A. Debido a que A satisface toda cláusula, tal L_i existe para todas las cláusulas. Asignar a C_i el mismo color que esta literal. Es decir, si L_i es x_i, a C_j se asigna el mismo color que x_i; en caso contrario, a C_j se asigna el mismo color que $-x_i$.

Considere el siguiente conjunto de cláusulas:

$$x_1 \vee x_2 \vee x_3 \quad \textbf{(1)}$$
$$-x_3 \vee -x_4 \vee x_2. \quad \textbf{(2)}$$

La gráfica se muestra en la figura 8-13.

FIGURA 8-13 Gráfica construida para el problema de decisión de coloración de una gráfica.



Sea $A = (x_1, -x_2, -x_3, x_4)$. Entonces las x_i se asignan como sigue:

$x_1 \leftarrow 1, \qquad -x_1 \leftarrow 5$
$x_2 \leftarrow 5, \qquad -x_2 \leftarrow 2$
$x_3 \leftarrow 5, \qquad -x_3 \leftarrow 3$
$x_4 \leftarrow 4, \qquad -x_4 \leftarrow 5.$

Las C_i se colorean como sigue:

$C_1 \leftarrow 1.$ (x_1 en A satisface C_1.)
$C_2 \leftarrow 3.$ ($-x_3$ en A satisface C_2.)

Para demostrar que esta coloración es válida puede razonarse como sigue:

1. Toda y_i está unida con toda y_j si $i \neq j$. En consecuencia, no es posible asignar el mismo color a dos y_i. Esto se hace en la medida en que a y_i se asigna el color i.
2. Toda x_i está unida con toda $-x_i$. En consecuencia, x_i y $-x_i$ no pueden colorearse igual. Esto se hace porque no se asigna el mismo color a x_i y $-x_i$. Además, según nuestras reglas ninguna y_j tiene el mismo color que x_i o $-x_i$.

3. Considere las C_j. Suponga que L_i aparece en C_j, que L_i es verdadera en A y que C_j se colorea igual que L_i. De todas las x_i y $-x_i$, sólo la x_i, o $-x_i$ particular, que es igual a L_i, tiene el mismo color que C_j. Pero L_i no está unida con C_j porque L_i aparece en C_j. En consecuencia, ninguna C_j tiene el mismo color que ninguna x_i, o $-x_i$, que está unida con C_j.

Con base en el análisis anterior, puede concluirse que si F es satisfactible, entonces G es $n + 1$ coloreable.

A continuación se demostrará la otra parte. Si G es $n + 1$ coloreable, entonces F debe ser satisfactible. El razonamiento es como sigue:

1. Sin pérdida de generalidad, puede suponerse que y_i está coloreada con el color i.
2. Debido a que x_i está unida con $-x_i$, a x_i y $-x_i$ no puede asignarse el mismo color. Debido a que x_i y $-x_i$ están unidas con y_j si $i \neq j$, entonces ocurre una de dos: a x_i se asigna el color i y a $-x_i$ se asigna el color $n + 1$ o a $-x_i$ se asigna el color i y a x_i se asigna el color $n + 1$.
3. Debido a que C_j contiene a lo más tres literales y $n \geq 4$, por lo menos hay una i tal que ni x_i ni $-x_i$ aparecen en C_j. Por consiguiente, toda C_j debe estar unida a por lo menos un vértice coloreado con el color $n + 1$. En consecuencia, ninguna C_j está coloreada con el color $n + 1$.
4. Para toda C_j, $1 \leq j \leq r$, si a C_j se asigna el color i y a x_i se asigna el color i, entonces en la asignación A asigna el valor verdadero a x_i. Si a C_j se asigna el color i y a $-x_i$ se asigna el color i, entonces en la asignación A asigna el valor falso a x_i.
5. Observe que si a C_j se asigna el color i, entonces no debe unirse con x_i o $-x_i$, cualquiera a la que se asigne el color i. Esto significa que a la literal a que se asigna el color i debe aparecer en C_j. En A, a esta literal particular se asigna el valor verdadero y, por lo tanto, satisface a C_j. En consecuencia, A satisface todas las cláusulas. F debe ser satisfactible porque existe por lo menos una asignación que satisface todas las cláusulas.

En el análisis anterior se ha demostrado que para todo conjunto de cláusulas donde cada cláusula contiene a lo sumo tres literales es posible construir una gráfica tal que el conjunto de cláusulas original con n variables es satisfactible si y sólo si la gráfica construida correspondiente es $n + 1$ coloreable. Además, es fácil demostrar que la construcción de la gráfica requiere un número polinomial de pasos. En consecuencia, *el problema de satisfactibilidad con cuando mucho tres literales por cláusula se reduce al problema de decisión de coloración de una gráfica y éste es NP-completo.*

A continuación, en este apartado se demostrará que un problema de disposición discreto VLSI es NP-completo. Para hacerlo, es necesario probar que algunos otros problemas son NP-completos.

Ejemplo 8-11 El problema de la cubierta exacta

Sea una familia de conjuntos $F = \{S_1, S_2, \ldots, S_k\}$ y un conjunto S de elementos $\{u_1, u_2, \ldots, u_n\}$, tales que $\bigcup_{S_i \in F} S_i$. El problema de la cubierta exacta consiste en determinar si existe un subconjunto $T \subseteq F$ de conjuntos ajenos por pares tal que

$$\bigcup_{S_i \in T} S_i = \{u_1, u_2, \ldots, u_n\} = \bigcup_{S_i \in F} S_i.$$

Por ejemplo, suponga que $F = \{(a_3, a_1), (a_2, a_4), (a_2, a_3)\}$. Entonces $T = \{(a_3, a_1), (a_2, a_4)\}$ es una cubierta exacta de F. Observe que todo par de conjuntos en T debe ser ajeno. Si $F = \{(a_3, a_1), (a_4, a_3), (a_2, a_3)\}$, entonces no existe cubierta exacta.

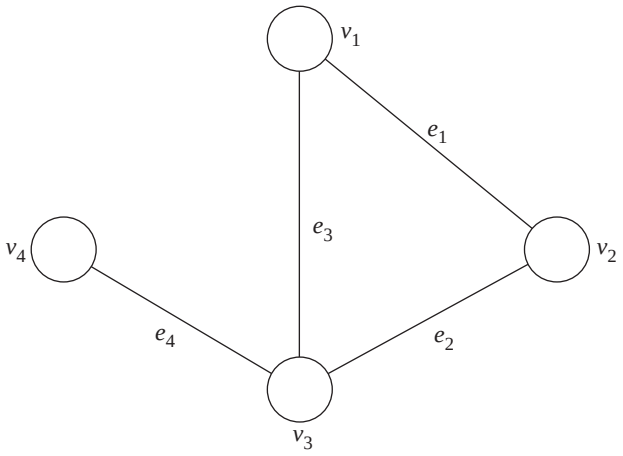
A continuación se intentará demostrar que este problema de la cubierta exacta es NP-completo, al reducir el problema de la coloración de una gráfica a este problema de la cubierta exacta. El problema del coloreado de vértices se presentó en el ejemplo 8-10.

Sean $V = \{v_1, v_2, \ldots, v_n\}$ el conjunto de vértices de la gráfica dada en el problema del coloreado de vértices y $E = \{e_1, e_2, \ldots, e_m\}$ el conjunto de aristas. Junto con el entero k, este caso del problema del coloreado de vértices se transforma en un caso del problema de la cubierta exacta $S = \{v_1, v_2, \ldots, v_n, E_{11}, E_{12}, \ldots, E_{1k}, E_{21}, E_{22}, \ldots, E_{2k}, \ldots, E_{m1}, E_{m2}, \ldots, E_{mk}\}$, donde $E_{i1}, E_{i2}, \ldots, E_{ik}$ corresponde a e_i, $1 \leq i \leq m$, y una familia F de subconjuntos $F = \{C_{11}, C_{12}, \ldots, C_{1k}, C_{21}, C_{22}, \ldots, C_{2k}, \ldots, C_{n1}, C_{n2}, \ldots, C_{nk}, D_{11}, D_{12}, \ldots, D_{1k}, D_{21}, D_{22}, \ldots, D_{2k}, \ldots, D_{m1}, D_{m2}, \ldots, D_{mk}\}$. Cada C_{ij} y D_{ij} están determinados según la regla siguiente:

1. Si la arista e_i tiene a los vértices v_a y v_b como sus terminales extremas, entonces ambas C_{ad} y C_{bd} contienen a E_{id} para $d = 1, 2, \ldots, k$.
2. $D_{ij} = \{E_{ij}\}$ para toda i y j.
3. C_{ij} contiene a v_i para $j = 1, 2, \ldots, k$.

Se proporcionará un ejemplo. Considere la figura 8-14, que muestra una gráfica.

FIGURA 8-14 Gráfica que ilustra la transformación de un problema de coloreado de vértices en un problema de cubierta exacta.



En este caso, $n = 4$ y $m = 4$. Suponga que $k = 3$. En consecuencia, se tiene $S = \{v_1, v_2, v_3, v_4, E_{11}, E_{12}, E_{13}, E_{21}, E_{22}, E_{23}, E_{31}, E_{32}, E_{33}, E_{41}, E_{42}, E_{43}\}$ y $F = \{C_{11}, C_{12}, C_{13}, C_{21}, C_{22}, C_{23}, C_{31}, C_{32}, C_{33}, C_{41}, C_{42}, C_{43}, D_{11}, D_{12}, D_{13}, D_{21}, D_{22}, D_{23}, D_{31}, D_{32}, D_{33}, D_{41}, D_{42}, D_{43}\}$. Cada D_{ij} contiene exactamente una E_{ij}. Para C_{ij}, su contenido se ilustrará con un ejemplo. Considere a e_1, que está unido por v_1 y v_2. Esto significa que C_{11} y C_{21} contienen ambos a E_{11}. De manera semejante, C_{12} y C_{22} contienen ambos a E_{12}. También, C_{13} y C_{23} contienen ambos a E_{13}.

Toda la familia F de conjuntos se construye como sigue:

$C_{11} = \{E_{11}, E_{31}, v_1\}$,
$C_{12} = \{E_{12}, E_{32}, v_1\}$,
$C_{13} = \{E_{13}, E_{33}, v_1\}$,
$C_{21} = \{E_{11}, E_{21}, v_2\}$,
$C_{22} = \{E_{12}, E_{22}, v_2\}$,
$C_{23} = \{E_{13}, E_{23}, v_2\}$,
$C_{31} = \{E_{21}, E_{31}, E_{41}, v_3\}$,
$C_{32} = \{E_{22}, E_{32}, E_{42}, v_3\}$,
$C_{33} = \{E_{23}, E_{33}, E_{43}, v_3\}$,
$C_{41} = \{E_{41}, v_4\}$,
$C_{42} = \{E_{42}, v_4\}$,
$C_{43} = \{E_{43}, v_4\}$,

$$D_{11} = \{E_{11}\},\ D_{12} = \{E_{12}\},\ D_{13} = \{E_{13}\},$$
$$D_{21} = \{E_{21}\},\ D_{22} = \{E_{22}\},\ D_{23} = \{E_{23}\},$$
$$D_{31} = \{E_{31}\},\ D_{32} = \{E_{32}\},\ D_{33} = \{E_{33}\},$$
$$D_{41} = \{E_{41}\},\ D_{42} = \{E_{42}\},\ D_{43} = \{E_{43}\}.$$

Se omitirá una demostración formal del hecho que el problema de coloreado de vértices tiene un k coloreado si y sólo si el problema de la cubierta exacta construido tiene una solución. Mientras tanto, la validez de esta transformación sólo se demostrará a través de un ejemplo.

Para la gráfica de la figura 8-14 existe un 3-coloreado. Puede hacerse que v_1, v_2, v_3 y v_4 estén coloreados como 1, 2, 3 y 1, respectivamente. En este caso, como la cubierta se escogen C_{11}, C_{22}, C_{33}, C_{41}, D_{13}, D_{21}, D_{32} y D_{42}. Primero puede verse fácilmente que son ajenos por pares. Además, el conjunto S está cubierto exactamente por estos conjuntos. Por ejemplo, v_1, v_2, v_3 y v_4 están cubiertos por C_{11}, C_{22}, C_{33} y C_{41} respectivamente. E_{11} está cubierto por C_{11}, E_{12} está cubierto por C_{22} y E_{13} está cubierto por D_{13}.

La demostración formal de esta reducibilidad se deja como ejercicio.

Ejemplo 8-12 El problema de la suma de subconjuntos

Este problema se define como sigue: sean un conjunto de números $A = \{a_1, a_2, \ldots, a_n\}$ y una constante C. Determinar si existe un subconjunto A' de A tal que la suma de los elementos de A' sea C.

Por ejemplo, sean $A = \{7, 5, 19, 1, 12, 8, 14\}$ y C igual a 21. Entonces este problema de la suma de subconjuntos tiene una solución; a saber, $A' = \{7, 14\}$. Si C es igual a 11, el problema de la suma de subconjuntos no tiene solución.

Puede demostrarse que este problema es NP-completo al reducir el problema de la cubierta exacta a este problema de la suma de subconjuntos. Dados un caso del problema de la cubierta exacta $F = \{S_1, S_2, \ldots, S_n\}$ y un conjunto $S = \bigcup_{S_i \in F} S_i = \{u_1, u_2, \ldots, u_m\}$, un caso del problema de la suma de subconjuntos correspondiente se construye según la siguiente regla: el caso del problema de la suma de subconjuntos contiene un subconjunto $A = \{a_1, a_2, \ldots, a_n\}$ donde

$$a_j = \sum_{1 \le i \le m} e_{ji}(n+1)^{i-1} \text{ donde } e_{ji} = 1 \text{ si } u_i \in S_j \text{ y } e_{ji} = 0 \text{ en caso contrario.}$$
$$C = \sum_{0 \le i \le m-1} (n+1)^i = ((n+1)^m - 1)/n.$$

De nuevo, la demostración formal de la validez de esta transformación se deja como ejercicio.

Ejemplo 8-13 El problema de decisión de partición

Este problema se define como sigue: se tiene $A = \{a_1, a_2, \ldots, a_n\}$, donde cada a_i es un entero positivo. El problema de partición consiste en determinar si existe una partición $A = \{A_1, A_2\}$ tal que

$$\sum_{a_i \in A_1} a_i = \sum_{a_i \in A_2} a_i .$$

Por ejemplo, sea $A = \{1, 3, 8, 4, 10\}$. A puede separarse en dos subconjuntos $\{1, 8, 4\}$ y $\{3, 10\}$, y es fácil comprobar que la suma de los elementos en el primer subconjunto es igual a la suma de los elementos en el segundo subconjunto.

Puede demostrarse que es NP-completo al reducir el problema de la suma de subconjuntos a este problema. La forma en que es posible hacer lo anterior se deja como ejercicio.

Ejemplo 8-14 El problema de decisión de empaque en contenedores

Este problema, que ya se presentó, puede describirse como sigue: se tiene un conjunto de n artículos, cada uno de tamaño c_i que es un entero positivo. También se tienen enteros positivos B y C que son el número de contenedores y la capacidad del contenedor, respectivamente. Se requiere determinar si es posible asignar artículos a k contenedores, $1 \leq k \leq B$, de modo que la suma de las c_i sobre todos los artículos asignados a cada contenedor no exceda C.

Puede establecerse que este problema es NP-completo al reducir el problema de partición a éste. Suponga que un caso del problema de partición tiene $A = \{a_1, a_2, \ldots, a_n\}$. El problema de decisión de empaque en contenedores correspondiente puede definirse al hacer $B = 2$, $c_i = a_i$, $1 \leq i \leq n$ y $C = \sum_{1 \leq i \leq n} a_i/2$. Resulta evidente que el problema de decisión de empaque en contenedores tiene una solución con k igual a 2 si y sólo si existe una partición para A.

Ejemplo 8-15 El problema de disposición discreta VLSI

En este problema se tiene un conjunto S de n rectángulos y la tarea es determinar si es posible colocar estos rectángulos, con ciertas restricciones, en un rectángulo mayor de

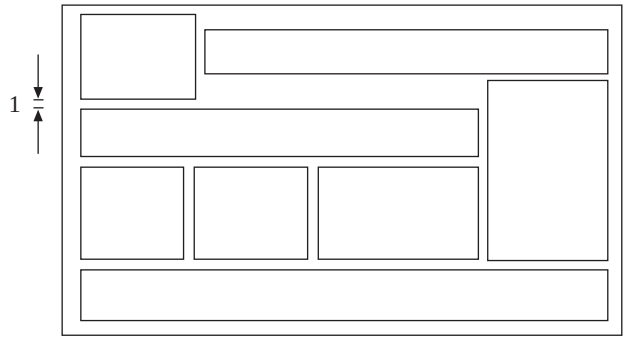
área específica. Formalmente hablando, se cuenta con un conjunto de n rectángulos y un entero A. Para $1 \leq i \leq n$, cada rectángulo r_i tiene dimensiones h_i y w_i, que son enteros positivos. El problema de disposición discreta VLSI consiste en determinar si existe una colocación de rectángulos sobre el plano de modo que:

1. Cada frontera sea paralela a uno de los ejes del sistema de coordenadas.
2. Los vértices de los rectángulos estén ubicados en puntos enteros del plano.
3. Los rectángulos no se traslapen.
4. Las fronteras de dos rectángulos estén separadas al menos por una distancia unitaria.
5. En el plano haya un rectángulo que circunscriba los rectángulos colocados, tenga fronteras paralelas a los ejes y su área sea cuando mucho igual a A. Se permite que la frontera de este rectángulo contenga fronteras de los rectángulos colocados.

Considere la figura 8-15, que muestra una colocación exitosa. Puede establecerse que este problema de disposición discreta VLSI es NP-completo al demostrar que el problema de decisión de empaque en contenedores puede reducirse al problema de disposición discreta VLSI. En un problema de decisión de empaque en contenedores, se tienen n artículos, cada uno de tamaño c_i. El número de contenedores es B y la capacidad de cada contenedor es C. Para este problema de decisión de empaque en contenedores, se construye un problema de disposición discreta VLSI como sigue:

1. Para todo c_i se tiene un rectángulo correspondiente r_i de altura $h_i = 1$ y ancho $w_i = (2B + 1)c_i - 1$.
2. Además, debe tenerse otro rectángulo de ancho $w = (2B + 1)C - 1$ y altura $h = 2Bw + 1$.
3. El área del rectángulo limitante es $A = w(h + 2B)$.

FIGURA 8-15 Una colocación exitosa.



Primero se demostrará que si el problema de decisión de empaque en contenedores tiene una solución, entonces el problema de disposición discreta VLSI construido también tiene una solución. Suponga que para el contenedor *i*, los tamaños de los artículos ahí almacenados son d_{i1}, d_{i2}, …, d_{ij_i}. Nuestra colocación dispondrá los rectángulos correspondientes en un renglón de tal modo que la altura de este renglón se mantenga en 1, lo cual se muestra en la figura 8-16. Si se usan *k* contenedores, habrá *k* renglones de rectángulos, donde la altura de cada uno es 1. Luego, el rectángulo de ancho *w* y altura *h* se coloca exactamente abajo de estos renglones como se muestra en la figura 8-17.

FIGURA 8-16 Un renglón particular de la colocación.

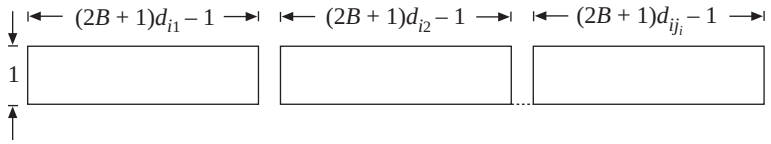
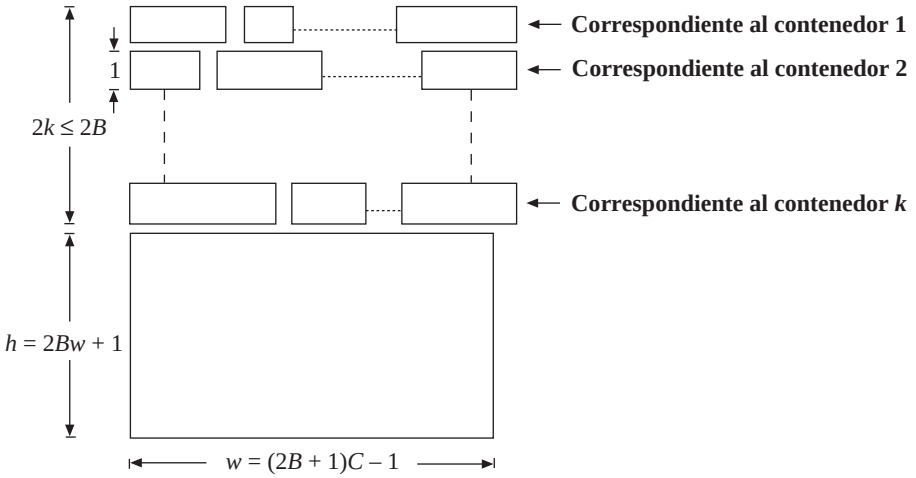


FIGURA 8-17 Una colocación de $n + 1$ rectángulos.



Para cada renglón i con j_i rectángulos, el ancho puede encontrarse como sigue:

$$\begin{aligned}
&\sum_{a=1}^{j_i} ((2B + 1)d_{i_a} - 1) + (j_i - 1) \\
&= (2B + 1)\left(\sum_{a=1}^{j_i} d_{i_a}\right) - j_i + j_i - 1 \\
&\leq (2B + 1)C - 1 \\
&= w.
\end{aligned}$$

Así, el ancho de estos $n + 1$ rectángulos es menor que o igual a w y la altura correspondiente es menor que $(2k + h) \leq (2B + h)$. Entonces, el área total de la colocación de estos $n + 1$ rectángulos es menor que

$$(2B + h)w = A.$$

Esto significa que si el problema de decisión de empaque en contenedores tiene una solución, entonces los $n + 1$ rectángulos correspondientes pueden colocarse en un rectángulo de tamaño A.

Ahora se demostrará la otra parte. Suponga que los $n + 1$ rectángulos se han colocado exitosamente en un rectángulo de tamaño $A = w(h + 2B)$. Ahora se demostrará que el problema original de decisión de empaque en contenedores tiene una solución.

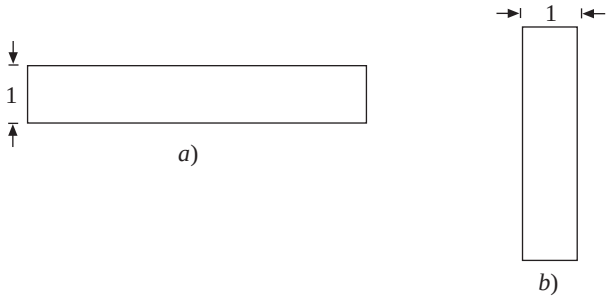
El razonamiento básico es que esta colocación exitosa no puede ser arbitraria; debe cumplir algunas restricciones. Las restricciones son las siguientes:

1. El ancho de la colocación debe ser menor que $w + 1$. Suponga lo contrario. Debido a que se tiene un rectángulo de altura h, el área total es mayor que

$$\begin{aligned}
&h(w + 1) \\
&= hw + h \\
&= hw + 2Bw + 1 \\
&= w(h + 2B) + 1 \\
&= A + 1 > A,
\end{aligned}$$

   lo cual es imposible.
2. Cada rectángulo r_i debe colocarse de modo que su contribución a la altura total sea 1. En otras palabras, el rectángulo r_i debe colocarse como se muestra en la figura 8-18*a*); no puede colocarse como se observa en la figura 8-18*b*).

FIGURA 8-18 Posibles formas de colocar r_i.



Suponga lo contrario. Entonces el área total debe ser mayor que

$$h + (2B + 1)c_i - 1 + 1$$
$$= h + (2B + 1)c_i$$

para alguna i. En este caso, el área total se vuelve mayor que

$$(h + (2B + 1)c_i)w$$
$$= hw + 2Bc_iw + wc_i$$
$$= (2Bc_i + h)w + wc_i$$
$$\geq A + wc_i$$
$$> A$$

lo cual es imposible.

3. El número total de renglones ocupados por los n rectángulos no puede ser mayor que B. Si es mayor, entonces el área total se hace mayor que

$$w(h + 2B) = A,$$

lo cual es imposible.

Con base en los argumentos anteriores, es fácil demostrar que

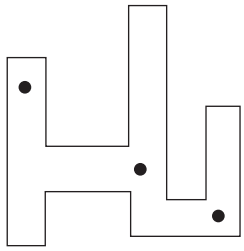
$$\sum_{k=1}^{j_i} d_{i_k} \leq C.$$

En otras palabras, los artículos correspondientes al renglón i pueden colocarse en el contenedor i sin exceder la capacidad de éste. Por consiguiente, se ha completado la demostración.

Ejemplo 8-16 El problema de la galería de arte para polígonos simples

Este problema, que se mencionó en el capítulo 3, se define como sigue: se tiene una galería de arte y se trata de colocar el número mínimo de guardias en ella de modo que todo punto de la galería sea visible para por lo menos un guardia. Se supone que la galería de arte es representada por un polígono simple. Así, el problema de la galería de arte también puede plantearse como sigue: *dado un polígono simple, colocar el número mínimo de guardias en éste de modo que todo punto del polígono simple sea visible por lo menos para un guardia.* Por ejemplo, considere el polígono simple en la figura 8-19. En este caso se requieren por lo menos tres guardias.

FIGURA 8-19 Un polígono simple y el número mínimo de guardias para i_t.



Si se estipula que cada guardia puede colocarse sólo en un vértice del polígono simple, entonces esta versión especial del problema de la galería de arte se denomina problema de colocación de guardias en un número mínimo de vértices. A continuación se demostrará que este problema es NP-difícil. La dificultad-NP del problema de la galería de arte puede demostrarse de manera semejante. Primero se define el *problema de decisión de guardias en vértices como sigue: dados un polígono simple* $\boldsymbol{P}$ *con* $\boldsymbol{n}$ *vértices y un entero positivo* $\boldsymbol{K} < \boldsymbol{n}$*, se pide determinar si existe un subconjunto* $\boldsymbol{T} \subseteq \boldsymbol{V}$ *con* $|\boldsymbol{T}| \leq \boldsymbol{K}$ *de modo que al colocar un guardia en cada vértice en* $\boldsymbol{T}$ *se obtiene que todo punto en* $\boldsymbol{P}$ *sea visible por lo menos para un guardia colocado en* $\boldsymbol{T}$.

Resulta fácil ver que el problema de decisión de guardias en vértices es NP, ya que un algoritmo no determinístico sólo requiere adivinar un subconjunto $V' \subseteq V$ de K vértices y comprobar en tiempo polinomial si colocar un guardia en cada vértice en V' permite que todo punto en el polígono sea visible para ese guardia. Para demostrar que este problema de decisión es NP-completo se usará el problema de 3-satisfactibilidad (3-SAT), que es NP-completo.

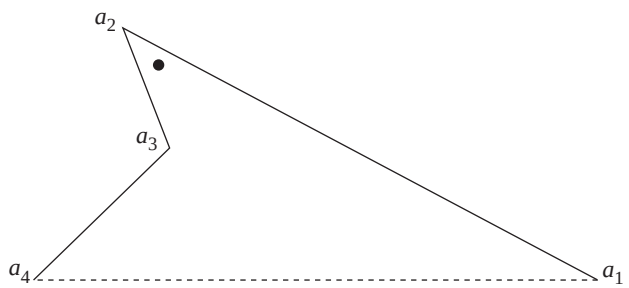
Se demostrará que el problema 3-SAT es polinomialmente reducible al problema de decisión de guardias en vértices. Primero se definen puntos distinguidos como si-

gue: *dos puntos diferentes en un polígono son distinguidos si no pueden ser visibles desde ningún punto de este polígono.* Esta definición es necesaria para análisis ulteriores. Para construir un polígono simple a partir de un caso dado del problema 3-SAT, primero se definen los polígonos básicos correspondientes a literales, cláusulas y variables, respectivamente. Estos polígonos básicos se combinarán en el polígono simple transformado final.

Polígonos de literales

Para cada literal se construye un polígono, lo cual se muestra en la figura 8-20. El signo punto "•" indica un punto distinguido porque, como se demostrará después, estos polígonos de literales se dispondrán de modo que en el polígono simple final, ningún punto en ellos pueda ver ambos. Resulta fácil percatarse que sólo a_1 y a_3 pueden ver este polígono de literales completo. En un caso del problema 3-SAT, por cada cláusula hay tres literales. Por lo tanto, en cada polígono correspondiente a una cláusula, habrá tres de estos polígonos de literales.

FIGURA 8-20 Patrón de un subpolígono de literales.

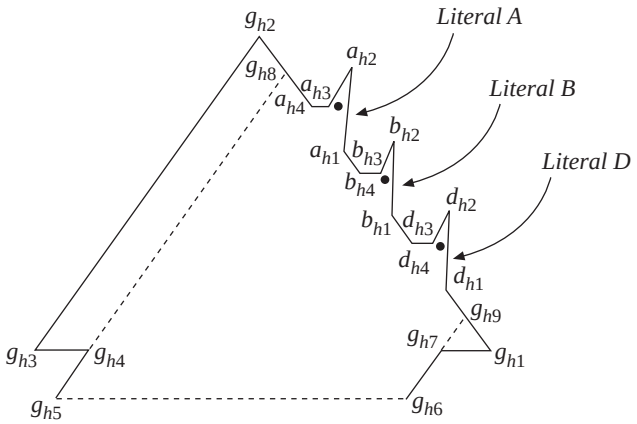


Polígonos de cláusulas

Para cada cláusula $C_i = A \vee B \vee D$, se construirá un polígono de cláusulas, que se muestra en la figura 8-21. Sea $(a_1, a_2, \ldots, a_n)$ que denota el hecho de que $a_1, a_2, \ldots, a_n$ son colineales. Entonces, en la figura 8-21, se tiene (g_{h8}, g_{h4}, g_{h5}), $(g_{h3}, g_{h4}, g_{h7}, g_{h1})$, $(g_{h2}, g_{h8}, a_{h4}, a_{h1}, b_{h4}, b_{h1}, d_{h4}, d_{h1}, g_{h9}, g_{h1})$ y (g_{h9}, g_{h7}, g_{h6}). Además, $|(g_{h2}, g_{h8})| = |(g_{h8}, a_{h4})|$ y $|(d_{h1}, g_{h9})| = |(g_{h9}, g_{h1})|$ donde $|(u, v)|$ denota la longitud del segmento de recta (u, v).

Un polígono de cláusulas posee algunas propiedades importantes. Antes que todo, es fácil darse cuenta que ningún g_{hi}, $i = 1, 2, \ldots, 7$, puede ver completo ningún polígono de literales. Así, es necesario colocar guardias dentro de éstos. Observe que ningún

FIGURA 8-21 Conjunción de cláusulas $C_h = A \vee B \vee D$.



guardia ubicado en ningún polígono de literales puede ver completos los otros dos polígonos de literales. En consecuencia, se requieren tres guardias, uno ubicado en cada polígono de literales. Sin embargo, en estos polígonos de literales hay ciertos vértices donde no es posible colocar un guardia. A continuación, se demostrará que sólo siete combinaciones escogidas de $\{a_{h1}, a_{h3}, b_{h1}, b_{h3}, d_{h1}, d_{h3}\}$ son candidatas para colocar guardias. El razonamiento es como sigue:

1. Ninguno de a_{h4}, b_{h4} y d_{h4} puede ver completo a ninguno de los tres polígonos de literales. Por lo tanto, no pueden ser candidatos.
2. Considere el polígono de literales para la literal *A*. Si se escoge a_{h2}, entonces sin importar cómo se ubiquen los guardias en los polígonos de literales para las literales *B* y *D*, $\Delta a_{h1}a_{h3}a_{h4}$ no será visible para ningún guardia. Por lo tanto, debe descartarse a_{h2}. Es decir, sólo se tienen que escoger estos guardias de $\{a_{h1}, a_{h3}, b_{h1}, b_{h3}, d_{h1}, d_{h3}\}$.
3. No deben escogerse dos vértices del mismo polígono de literales. Así, sólo hay ocho combinaciones posibles de los vértices escogidos de $\{a_{h1}, a_{h3}, b_{h1}, b_{h3}, d_{h1}, d_{h3}\}$. Pero $\{a_{h3}, b_{h3}, d_{h3}\}$ debe excluirse porque no puede ver completo a $\Delta g_{h1}g_{h2}g_{h3}$. Esto significa que sólo se tienen siete combinaciones posibles, como se resume a continuación.

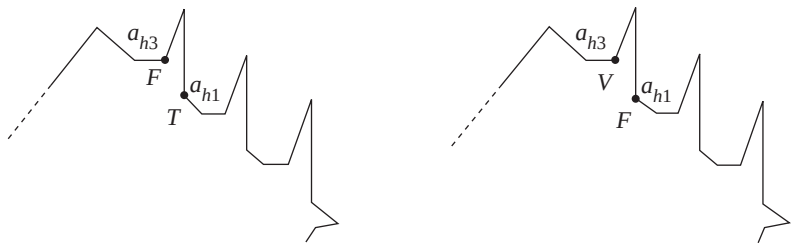
   Propiedad 1: Sólo siete combinaciones de vértices escogidos de $\{a_{h1}, a_{h3}, b_{h1}, b_{h3}, d_{h1}, d_{h3}\}$ pueden ver todo el polígono de cláusulas. Estas siete combinaciones de tales vértices son $\{a_{h1}, b_{h1}, d_{h1}\}$, $\{a_{h1}, b_{h1}, d_{h3}\}$, $\{a_{h1}, b_{h3}, d_{h1}\}$, $\{a_{h1}, b_{h3}, d_{h3}\}$, $\{a_{h3}, b_{h1}, d_{h1}\}$, $\{a_{h3}, b_{h1}, d_{h3}\}$ y $\{a_{h3}, b_{h3}, d_{h1}\}$.

Mecanismo 1 de etiquetado para polígonos de cláusulas

Observe que todo polígono de literales corresponde a una literal. Una literal puede aparecer positiva o negativa en una cláusula. No obstante, con base en el análisis anterior sobre la construcción de estos polígonos de literales, parece que no se toman en cuenta los signos de las literales. Esto es desconcertante. En realidad, como se verá después, el signo de una literal determina la etiqueta de algunos vértices de este polígono de literales. Entonces, la asignación del valor de verdad a esta variable determina dónde deben ubicarse los guardias al tomar en cuenta el etiquetado de vértices.

Considere la literal *A*. Los otros casos son semejantes. Observe que para la literal *A*, es necesario ubicar un guardia ya sea en a_{h3} o en a_{h3}. Si *A* es positiva (negativa), entonces el vértice $a_{h1}(a_{h3})$ se identifica como *V* y el vértice $a_{h1}(a_{h1})$ como falso. Consulte la figura 8-22. Si a esta variable se asigna el valor verdadera (falsa), entonces en el vértice identificado como *V*(*F*) se ubica un guardia. Esto significa que el vértice $a_{h1}(a_{h3})$ representa una asignación verdadera (falsa) para la literal *A*.*

FIGURA 8-22 Mecanismo 1 de etiquetado para conjunciones de cláusulas.



a) **literal** $A = u_i$ *b*) **literal** $A = -u_i$

A continuación se considerará un ejemplo para explicar la relación que existe entre el mecanismo de etiquetado, la asignación del valor de verdad y finalmente la ubicación de los guardias. Considere el caso en que $C_h = u_1 \vee -u_2 \vee u_3$. El etiquetado para este caso se muestra en la figura 8-23.

Considere una asignación satisfactible, por ejemplo

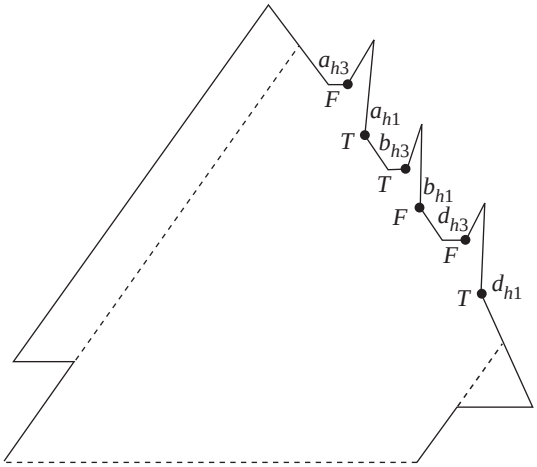
$u_1 \leftarrow T,\ u_2 \leftarrow F,\ u_3 \leftarrow T.$

En este caso, los guardias se ubicarán en a_{h1}, b_{h1} y d_{h1}. Los guardias así ubicados no pueden ver todo el polígono de cláusulas. Considere otra asignación satisfactible, por ejemplo

$u_1 \leftarrow T,\ u_2 \leftarrow T,\ u_3 \leftarrow F.$

* Los autores usan el mismo texto para definir el caso positivo y el caso negativo. *(N. del R.T.)*

FIGURA 8-23 Ejemplo del mecanismo 1 de etiquetado para conjunciones de cláusulas $C_h = u_1 \vee -u_2 \vee u_3$.



En este caso, los guardias se ubicarán en a_{h1}, b_{h3} y d_{h3}. De nuevo, los guardias así ubicados pueden ver todo el polígono de cláusulas. Finalmente, considere la siguiente asignación insatisfactible:

$u_1 \leftarrow F$, $u_2 \leftarrow T$, $u_3 \leftarrow F$.

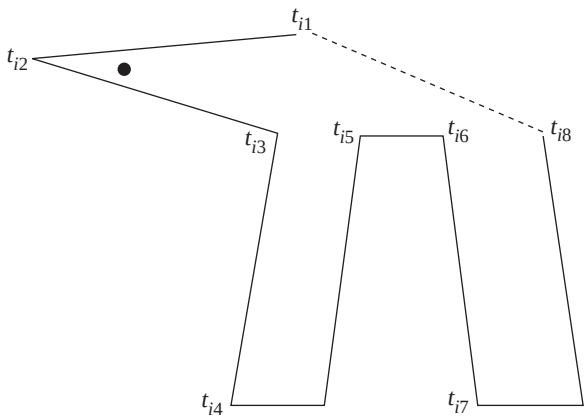
En este caso, los guardias se ubicarán en a_{h3}, b_{h3} y d_{h3}. Los guardias así ubicados no pueden ver todo el polígono de cláusulas. Al analizar cuidadosamente el mecanismo 1 de etiquetado puede verse que de las ocho combinaciones posibles de vértices de $\{a_{h1}, a_{h3}, b_{h1}, b_{h3}, d_{h1}, d_{h3}\}$, sólo $\{a_{h3}, b_{h3}, d_{h3}\}$ corresponde a una asignación insatisfactoria de valores de verdad a variables que aparecen en la cláusula. En consecuencia, se tiene la siguiente propiedad:

Propiedad 2: *El polígono de cláusulas de una cláusula C_h puede ser visible desde tres vértices en los polígonos de literales si y sólo si los valores de verdad correspondientes a las identificaciones de los vértices constituyen una asignación satisfactoria de la cláusula C_h.*

Polígonos de variables

Para cada variable se construye un polígono de variables, que se muestra en la figura 8-24. Se tiene (t_{i3}, t_{i5}, t_{i6}, t_{i8}). En $\Delta t_{i1}t_{i2}t_{i3}$ existe un punto distinguido. Observe que $\Delta t_{i1}t_{i2}t_{i3}$ sólo puede ser visible desde t_{i1}, t_{i2}, t_{i3}, t_{i5}, t_{i6} y t_{i8}.

FIGURA 8-24 Patrón de variables para u_i.



Se han descrito los polígonos de cláusulas y los polígonos de variables. A continuación se describe cómo es posible fusionarlos en el polígono de un caso del problema.

Construcción del polígono de una instancia del problema

Paso 1. Colocación de los polígonos de variables y los polígonos de cláusulas juntos.

Los polígonos de variables y los polígonos de cláusulas se colocan juntos como se muestra en la figura 8-25. En esta figura se tienen los hechos siguientes:

1. El vértice w puede visualizar los n polígonos de variables, excepto a $\Delta t_{i1}t_{i2}t_{i3}$, donde $i = 1, 2, 3, \ldots, n$.
2. $(w, g_{15}, g_{16}, g_{25}, g_{26}, \ldots, g_{m5}, g_{m6})$ y $(t_{11}, g_{h5}, g_{h4}, g_{h8})$ son satisfechos donde $h = 1, 2, \ldots, m$.

Paso 2. Ampliación de los polígonos de variables mediante picos.

Suponga que la variable u_i aparece en la cláusula C_h. Si u_i aparece positivamente en la cláusula C_h, entonces los dos picos $\overline{pq}$ y $\overline{rs}$ que se muestran en la figura 8-26*a*) son tales que se satisfacen (a_{h1}, t_{i5}, p, q) y (a_{h3}, t_{i8}, r, s). La situación en que u_i aparece negativamente en la cláusula C_h se muestra en la figura 8-26*b*).

Paso 3. Sustitución de picos por polígonos.

Debido a que los picos añadidos $\overline{pq}$ y $\overline{rs}$ son segmentos de recta, después de completar el paso 2 podría no tenerse un polígono simple. En consecuencia, es necesario sustituir los picos por polígonos como sigue. El caso se explicará en la figura 8-26*a*), ya que el de la figura 8-26*b*) es muy semejante. En

FIGURA 8-25 Fusión de patrones de variables y conjunciones de cláusulas.

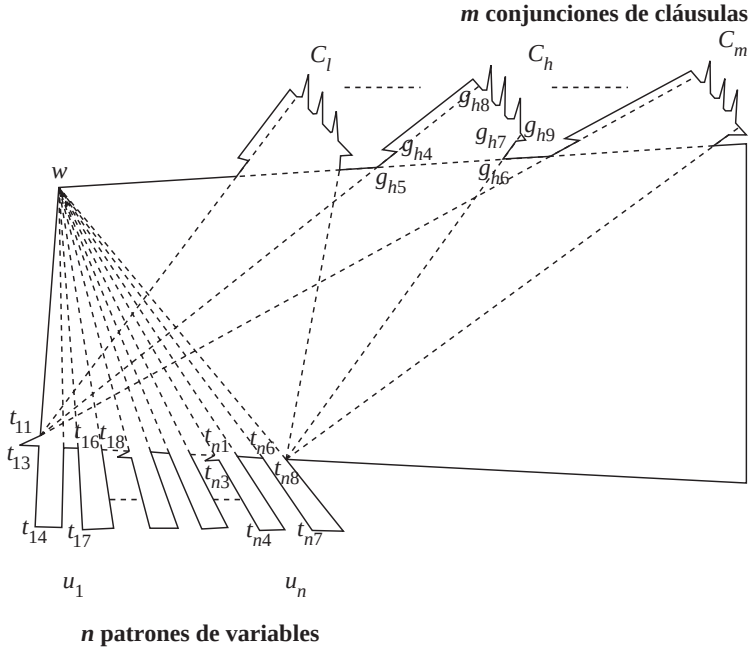
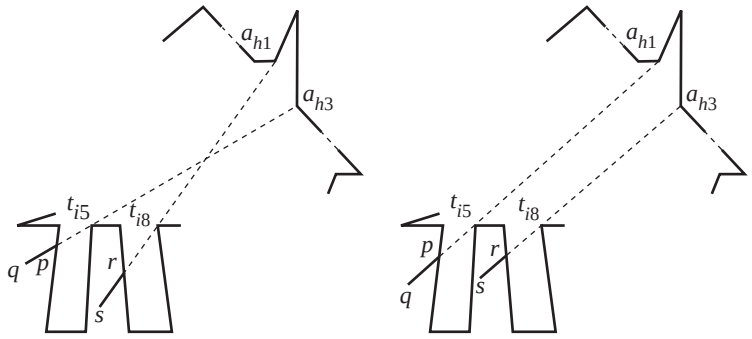


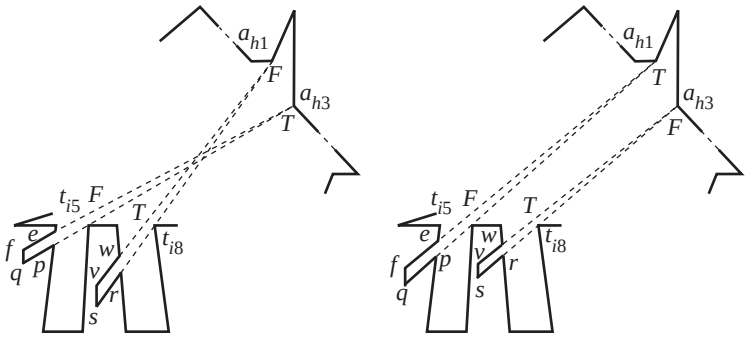
FIGURA 8-26 Ampliación de picos.



a) *b*)

la figura 8-26*a*), $\overline{pq}$ y $\overline{rs}$ se sustituyen por las áreas sombreadas que se muestran en la figura 8-27*a*). En la figura 8-27*b*) se tiene (a_{h1}, t_{i5}, *p*, *q*), (a_{h1}, *e*, *f*), (a_{h3}, t_{i8}, *r*, *s*) y (a_{h3}, *u*, *v*). Es decir, se tiene $\Delta a_{h1}qf$ y $\Delta a_{h3}sv$. Los polígonos obtenidos en la figura 8-27 se denominan polígonos de comprobación de consistencia. El significado de comprobación de consistencia se explicará después.

FIGURA 8-27 Sustitución de cada pico por una pequeña región denominada patrón de comprobación de consistencia.



a) *b*)

Mecanismo 2 de etiquetado para polígonos de variables

Para el polígono de variables u_i, los vértices t_{i5} y t_{i8} siempre se identifican como *F* y *T*, respectivamente, como se muestra en la figura 8-27. Si el valor asignado a u_i es verdadero (falso), entonces en el vértice identificado por *T*(*F)* se ubica un guardia.

A continuación se volverá a considerar la figura 8-27*a*). En este caso, $u_i \in C_h$. Observe que hay dos formas de colocar un guardia en el polígono de literales y un guardia en el polígono de variables.

$$F = (u_1 \vee u_2 \vee -u_3) \wedge (-u_1 \vee -u_2 \vee u_3).$$

Se trata de $\{a_{h1}, t_{i8}\}$ y $\{a_{h3}, t_{i5}\}$. El primer caso corresponde al hecho de asignar un valor verdadero a u_i y el segundo, al hecho de asignar un valor falso a u_i.

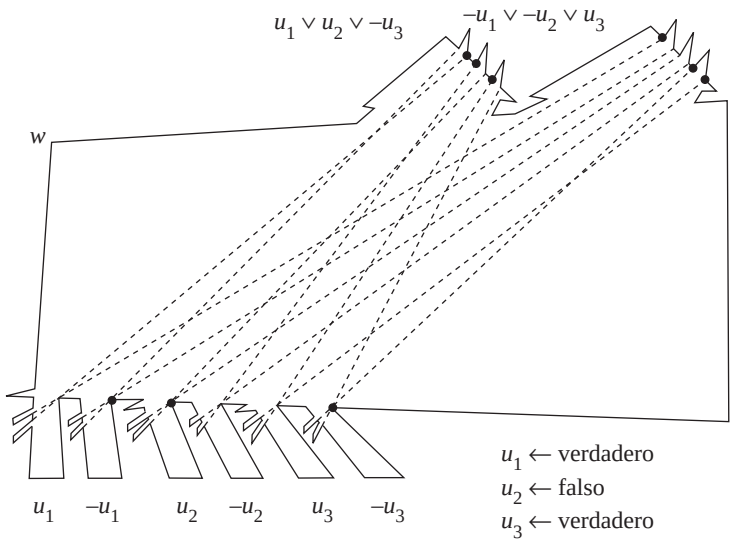
En la figura 8-28, a partir de la fórmula simple 3-SAT $F = (u_1 \vee u_2 \vee -u_3) \wedge (-u_1 \vee -u_2 \vee u_3)$ se construyó un polígono simple completo. Los guardias están representados por puntos. En este caso, el número mínimo de guardias es 10. Según los mecanismos 1 y 2 de identificación, este patrón de colocación de los guardias corresponde a

$$u_1 \leftarrow T,\ u_2 \leftarrow F,\ u_3 \leftarrow T.$$

Los resultados principales se muestran a continuación.

Afirmación 1: Dado un conjunto *S* de cláusulas, si *S* es satisfactible, entonces el número mínimo de vértices necesarios para ver todo el polígono de un caso del problema de *S* es $K = 3m + n + 1$.

FIGURA 8-28 Ejemplo de un polígono simple construido a partir de la fórmula 3-SAT.



Lo anterior puede razonarse como sigue: si S es satisfactible, entonces existe una asignación de valor verdadero a las variables que aparecen en S de modo que toda cláusula C_h es verdadera bajo esta asignación. Si el valor que se asigna a u_i es verdadero, entonces se coloca un guardia en t_{i8} del polígono de variables correspondiente para u_i y se coloca un guardia ya sea en a_{h1} o en a_{h3} del polígono de literales correspondiente dependiendo de si u_i está en C_h o si $-u_i$ está en C_h, respectivamente. Si el valor que se asigna a u_i es falso, entonces se coloca un guardia en t_{i5} del polígono de variables correspondiente para u_i y se coloca un guardia ya sea en a_{h1} o en a_{h3} del polígono de literales correspondiente dependiendo de si $-u_i$ o u_i está en C_h, respectivamente. Así, para los polígonos definidos por los polígonos de comprobación de consistencia y por los polígonos de literales, es necesario ubicar $3m + n$ guardias para verlos según los mecanismos 1 y 2 de identificación. Para los rectángulos restantes definidos por los polígonos de variables, para verlos sólo se requiere un guardia ubicado en el vértice w. Así, se tiene $K = 3m + n + 1$.

Afirmación 2: Dado un conjunto S de cláusulas, si el número mínimo de guardias es $K = 3m + n + 1$, entonces S es satisfactible.

Esta parte de la demostración es mucho más complicada. Primero, se afirma que debe escogerse w; en caso contrario, por lo menos se requieren $3m + 2n$ vértices, ya que un guardia debe estar ubicado en cada uno de los $2n$ rectángulos en los n polígonos de variables. Así, en lo que sigue, sólo se consideran los $K - 1 = 3m + n$ vértices restantes.

En el polígono simple construido, cada uno de los $3m$ polígonos de literales posee un punto distinguido y cada uno de los n polígonos de variables también posee uno. Así, en el polígono simple hay $3m + n$ puntos distinguidos. Se sabe que ningún vértice que ve un punto distinguido puede ver a cualquier otro punto distinguido. Así, se requieren por lo menos $3m + n$ vértices para ver los $3m + n$ puntos distinguidos.

Estos $3m + n$ puntos no pueden escogerse de manera arbitraria; en caso contrario, no pueden ver el resto del polígono simple. Más adelante, se demostrará que en realidad deben escogerse de modo que correspondan a una asignación de valores verdaderos a variables que satisfacen a S. Este patrón de ubicación de guardias se denomina consistente y se define como sigue:

Sea u_i una variable. Sea u que aparece positiva o negativamente en $C_1, C_2, \ldots, C_a$. Sean $x_1, x_2, \ldots, x_a$ los vértices escogidos de los polígonos de literales correspondientes a u que aparece en $C_1, C_2, \ldots, C_a$. Sea $\boldsymbol{X_i}$ igual a $\{x_1, x_2, \ldots, x_a\}$. Entonces el polígono de variables de u_i es consistente con respecto a $\boldsymbol{X_i}$ si se cumplen las siguientes condiciones:

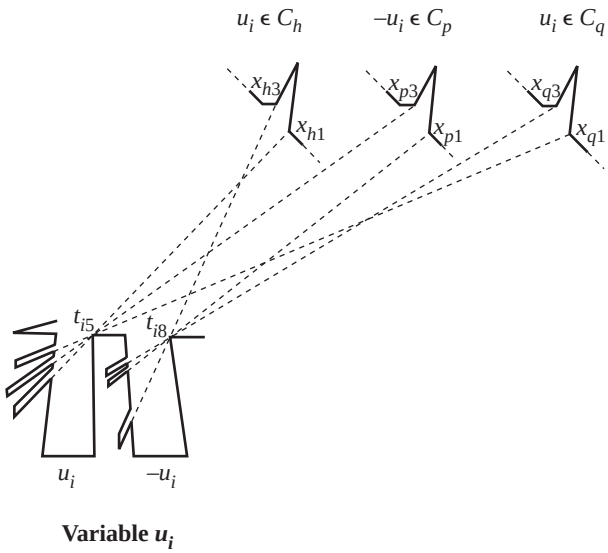
1. *Todos los polígonos de comprobación de consistencia unidos con uno de sus dos rectángulos pueden ser visibles desde los vértices en $\boldsymbol{X_i}$.*
2. *Los polígonos unidos con el otro rectángulo no pueden ser visibles en absoluto desde el mismo conjunto en $\boldsymbol{X_i}$.*

El polígono de variables de $\boldsymbol{u_i}$ es inconsistente con respecto a $\boldsymbol{X_i}$ en caso contrario.

Por ejemplo, considere la figura 8-29. Suponga que la variable u_i aparece en las cláusulas C_h, C_p y C_q y que aparece positivamente (u_i), negativamente ($-u_i$) y positivamente (u_i), respectivamente. Si se escogen los vértices x_{h1}, y_{p3} y z_{q1}, entonces el polígono de variables es consistente con respecto a este conjunto de vértices, ya que el rectángulo izquierdo puede ser visto desde estas tres variables y el rectángulo derecho no puede ser visto en absoluto desde estas tres variables. Este conjunto de vértices corresponde a u_i, a la cual se ha asignado un valor verdadero. De manera semejante, es consistente con respecto a x_{h3}, y_{p1} y z_{p3}. Es inconsistente con respecto a x_{h3}, y_{p1} y z_{q1}, lo cual corresponde a asignar un valor falso a u_i. *Puede verse que cuando un polígono de variables para la variable $\boldsymbol{u_i}$ es consistente con respecto a $\boldsymbol{X_i}$, entonces $\boldsymbol{X_i}$ corresponde a una cierta asignación del valor verdadero a u_i y si es inconsistente con $\boldsymbol{X_i}$, entonces $\boldsymbol{X_i}$ no corresponde a ninguna asignación de un valor verdadero a $\boldsymbol{u_i}$.*

Se afirma que todos los polígonos de variables deben ser consistentes con respecto al conjunto de vértices en los polígonos de literales si sólo se tienen $K = 3m + n$ vértices. En cualquier polígono de variables L_i para U_i, *el número de polígonos de comprobación de consistencia depende del número de cláusulas en que* aparece u_i. Sea q este número. Debido a que el número de polígonos de comprobación de consistencia unidos con el

FIGURA 8-29 Ilustración del concepto de consistencia.



rectángulo izquierdo es igual al número de polígonos de comprobación de consistencia unidos con el rectángulo derecho de L_i, entonces el número total de polígonos de comprobación de consistencia en L_i es $2q$; q de ellos están unidos a un rectángulo, ya sea izquierdo o derecho. Si es izquierdo (derecho), se coloca un guardia en $t_{i8}(t_{i5})$. Por lo tanto, sólo se requieren $3m + n + 1$ guardias. Si L_i es inconsistente, entonces algunos de los polígonos de comprobación de consistencia unidos con el rectángulo izquierdo y algunos de los polígonos de comprobación de consistencia unidos con el rectángulo derecho no pueden verse desde cualquier vértice. Entonces, se requieren por lo menos $3m + n + 2$ vértices. Por lo tanto, $K = 3m + n + 1$ requiere que todos los polígonos de variables sean consistentes con respecto a los vértices escogidos de los polígonos de literales.

Debido a que toda variable u_i es consistente con X_i, los polígonos de comprobación de consistencia unidos con uno de sus rectángulos pueden verse desde los vértices en X_i. Para el otro rectángulo, los polígonos de comprobación de consistencia pueden verse ya sea desde t_{i8} o desde t_{i5}. Además, t_{i8} o t_{i5} también pueden ver el punto distinguido del polígono de variables. Esto significa que se requiere exactamente un vértice para ver cada polígono de variables y que se requieren n vértices para ver los n polígonos de variables. Por lo tanto, se tiene que todos los m polígonos de cláusulas, que no pueden ser vistos desde w y cualquier vértice escogido de los polígonos de variables, deben ser vistos desde $3m + n + 1 - (n + 1) = 3m$ vértices escogidos de los polígonos de literales. Según la propiedad 2, tres vértices por cláusula corresponden a una asignación verdadera a variables que aparecen en esta cláusula. En consecuencia, C es satisfactible.

8-8 EL PROBLEMA DE 2-SATISFACTIBILIDAD

En este capítulo se demostró que el problema de satisfactibilidad es NP-completo. También se afirmó que aunque un problema sea NP-completo, su variante no necesariamente lo es. Para recalcar esta cuestión, se demostrará que el problema 2-satisfactibilidad está en *P*, lo cual es muy sorprendente.

El problema 2-satisfactibilidad (2-SAT en forma abreviada) es un caso especial del problema de satisfactibilidad, donde en cada cláusula hay exactamente dos literales.

Dado un caso del problema 2-SAT, es posible construir una gráfica dirigida, denominada gráfica de asignación, de este caso del problema. La gráfica de asignación se construye como sigue: si $x_1, x_2, \ldots, x_n$ son variables en el caso del problema, entonces los nodos de la gráfica de asignación son $x_1, x_2, \ldots, x_n$, y $-x_1, -x_2, \ldots, -x$. Si hay una cláusula en la forma de $L_1 \vee L_2$ donde L_1 y L_2 son literales, entonces en la gráfica de asignación hay una arista que va de $-L_1$ a L_2 y una arista que va de $-L_2$ a L_1. Debido a que toda arista está relacionada con una y sólo una cláusula, cada arista se identifica con la cláusula correspondiente.

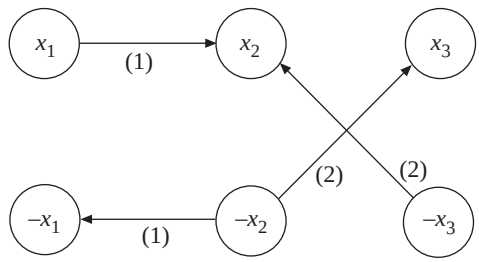
Considere el siguiente problema 2-SAT:

$$-x_1 \quad \vee \quad x_2 \qquad \textbf{(1)}$$

$$x_2 \quad \vee \quad x_3. \qquad \textbf{(2)}$$

Ahora se construye la gráfica de asignación del problema 2-SAT anterior, que se muestra en la figura 8-30.

FIGURA 8-30 Una gráfica de asignación.



Considere la cláusula (1), que es

$$-x_1 \quad \vee \quad x_2.$$

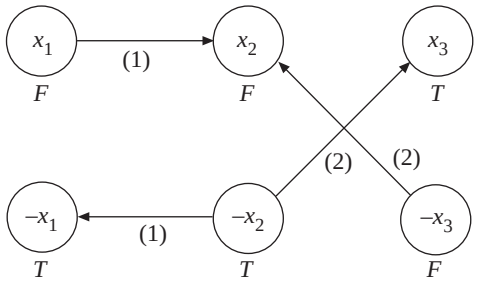
Suponga que se asigna x_1T; entonces para hacer verdadera la cláusula es necesario asignar x_2T. Esto se indica en la gráfica de asignación con la arista que va de x_1 a x_2. Por otra parte, suponga que se asigna $-x_2T$; entonces es necesario asignar $-x_1T$. De nuevo, esto se indica con la arista que va de $-x_2$ a $-x_1$. Esto explica por qué hay dos aristas para cada cláusula. Estas dos aristas son simétricas. Es decir, si hay una arista del nodo x al nodo y, entonces hay un nodo de $-y$ a $-x$. Cada arista corresponde a una asignación posible que satisface a esta cláusula.

Con base en el razonamiento anterior, se tienen las reglas siguientes que asignan valores de verdad a los nodos de la gráfica de asignación.

1. Si a un nodo A se asigna T, entonces a todos los nodos alcanzables desde el nodo A debe asignarse T.
2. Si a un nodo A se asigna T, entonces al nodo $-A$ simultáneamente debe asignarse F.

Por ejemplo, considere la gráfica en la figura 8-30. Si se asigna $-x_2T$, debe asignarse tanto $-x_1$ como x_3T. Simultáneamente, a todos los otros nodos debe asignarse F, lo cual se ilustra en la figura 8-31. Es posible que haya más de una forma de asignar valores de verdad a una gráfica de asignación para satisfacer todas las cláusulas. Por ejemplo, para la gráfica en la figura 8-30, es posible asignar $-x_3$, x_2T y x_1F. Puede verse fácilmente que estas dos asignaciones, a saber, $(-x_1, -x_2, x_3)$ y $(-x_1, x_2, -x_3)$, satisfacen las cláusulas.

FIGURA 8-31 Asignación de valores de verdad a una gráfica de asignación.



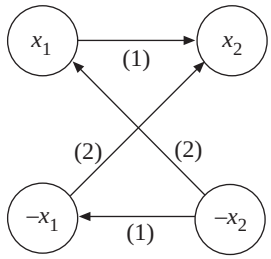
Debido a que hay una arista incidente en un nodo A si y sólo si la variable A aparece en una cláusula, *siempre que a un nodo* ***A*** *se asigne* ***T****, se satisfacen todas las cláusulas correspondientes a estas aristas incidentes en* ***A***. Por ejemplo, suponga que al nodo x_2 en la figura 8-30 se asigna T. Debido a que las aristas que inciden en el nodo x_2 corresponden a las cláusulas 1 y 2, ahora se satisfacen estas dos cláusulas.

Se considerará otro ejemplo. Suponga que las cláusulas son

$$-x_1 \quad \vee \quad x_2 \qquad (1)$$
$$x_1 \quad \vee \quad x_2. \qquad (2)$$

La gráfica de asignación correspondiente al conjunto de cláusulas anterior se muestra en la figura 8-32. En este caso no es posible asignar T a $-x_2$. En caso de hacerlo, se presentan dos problemas:

FIGURA 8-32 Una gráfica de asignación.



1. Tanto a x_1 como a $-x_1$ se asigna T porque ambos x_1 y $-x_1$ son alcanzables desde $-x_2$. Esto no está permitido.
2. Ya que a $-x_2$ se asigna T, entonces a x_1 se asigna T. Esto provocará que finalmente a x_2 se asigne T. De nuevo, esto no está permitido porque no es posible asignar T a ambos $-x_2$ y x_2.

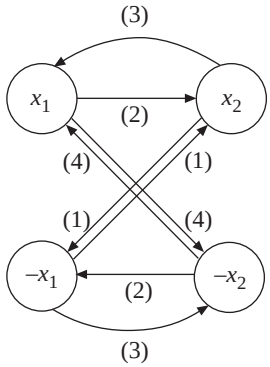
Sin embargo, es posible asignar x_2T. Esta asignación satisface ambas cláusulas 1 y 2, como se indica en la figura 8-32. En cuanto a x_1, puede asignársele cualquier valor. En cuanto concierne a la satisfactibilidad, después de asignar x_2T, la asignación de x_1 ya no importa.

Considere el siguiente ejemplo:

$$x_1 \quad \vee \quad x_2 \qquad (1)$$
$$-x_1 \quad \vee \quad x_2 \qquad (2)$$
$$x_1 \quad \vee \quad -x_2 \qquad (3)$$
$$-x_1 \quad \vee \quad -x_2. \qquad (4)$$

La gráfica de asignación correspondiente se muestra en la figura 8-33.

FIGURA 8-33 Gráfica de asignación correspondiente a un conjunto insatisfactible de cláusulas.



En este caso puede verse fácilmente que no hay ninguna forma de asignar valores de verdad a nodos en la gráfica. Toda asignación conduce a una contradicción. Suponga que se asigna x_1T. Debido a que hay una ruta de x_1 a $-x_1$, esto provoca que a $-x_1$ también se asigne T. Si se asigna $-x_1T$, esto obliga a asignar x_1T, ya que también hay una ruta de $-x_1$ a x_1. Entonces, simplemente no es posible asignar ningún valor de verdad a x_1.

La imposibilidad de asignar valores de verdad a los nodos en esta gráfica de ninguna manera es fortuita. Tal imposibilidad se debe a que este conjunto de cláusulas es insatisfactible.

En general, *un conjunto de cláusulas 2-SAT es satisfactible si y sólo si su gráfica de asignación no contiene simultáneamente una ruta de x a –x y una ruta de –x a x.* El lector puede ver fácilmente que en la gráfica en la figura 8-33 hay tales rutas, y que en ninguna de las gráficas en las figuras 8-31 y 8-32 se observa alguna de estas rutas.

Suponga que en la gráfica de asignación, para alguna x hay una ruta x a $-x$ y una ruta de $-x$ a x. Resulta evidente que no es posible asignar ningún valor de verdad a x, por lo que las cláusulas de entrada deben ser insatisfactibles.

Suponga que en la gráfica de asignación, para toda x no coexisten una ruta x a $-x$ y una ruta de $-x$ a x. Entonces se demostrará que el conjunto de cláusulas de entrada debe ser satisfactible. El razonamiento es que en esta situación es posible asignar exitosamente valores de verdad a los nodos de la gráfica de asignación. A continuación se proporciona un algoritmo que puede usarse para encontrar tal asignación.

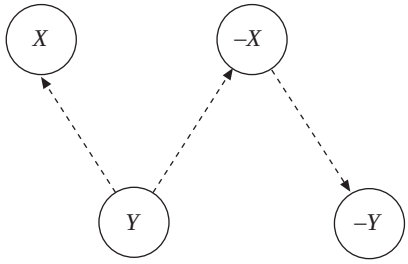
Paso 1. En la gráfica de asignación se escoge un nodo A al que no se haya asignado ningún valor de verdad y no conduzca a $-A$.

Paso 2. Al nodo A y a todos los nodos alcanzables desde A se asigna un valor verdadero. Para todo nodo x al que se haya asignado T, a $-x$ se asigna F.

Paso 3. Si hay algún nodo al que aún no se haya asignado un valor de verdad, ir al paso 1; en caso contrario, salir.

Todavía queda una cuestión que debe abordarse con extremo cuidado. En el algoritmo anterior, ¿la asignación puede conducir a una situación en que tanto a x como a $-x$ se asigna T, para alguna x? Esto es imposible. Si un nodo y conduce a un nodo x, entonces por la simetría de la gráfica de asignación debe haber una arista que va de $-x$ a $-y$. Así, y es tal nodo que conduce a $-y$, lo cual se muestra en la figura 8-34, por lo que no se escoge en el paso 1.

FIGURA 8-34 Simetría de las aristas en la gráfica de asignación.



Debido a que para toda x las rutas de x a $-x$ y de $-x$ a x no coexisten, entonces siempre hay un nodo que sea posible escoger en el algoritmo. En otras palabras, el algoritmo siempre producirá una asignación que satisface al conjunto de cláusulas de entrada. Por lo tanto, el conjunto de cláusulas es satisfactible.

Se ha demostrado que para determinar la satisfactibilidad de un caso del problema 2-SAT basta determinar si su gráfica de asignación correspondiente contiene una ruta de x a $-x$ y una ruta y de $-x$ a x para alguna x. Esto es equivalente a determinar si en la gráfica de asignación existe un ciclo de la forma $x \rightarrow \cdots \rightarrow -x \rightarrow \cdots \rightarrow x$. Hasta el momento, no se ha analizado cómo diseñar el algoritmo. Después será muy fácil que el lector diseñe un algoritmo polinomial para determinar si en la gráfica de asignación existe un ciclo así.

En conclusión, el problema 2-SAT es un problema P.

8-9 NOTAS Y REFERENCIAS

Quizás el concepto del NP-completo es uno de los más difíciles de entender en ciencias de computación. El artículo más importante relacionado con este concepto fue redacta-

do por Cook en 1971, donde demuestra la importancia del problema de satisfactibilidad. En 1972, Karp demostró que muchos problemas combinatorios son NP-completos (Karp, 1972). Estos dos artículos han sido considerados hitos en este tema. A partir de entonces, se ha demostrado que muchos problemas son NP-completos. Hay tantos que David Johnson de AT&T mantiene una base de datos. Johnson también es colaborador regular del *Journal of Algorithms* sobre el tema del NP-completo. Sus artículos se titulan "The NP-completeness Column: An Ongoing Guide". Estos artículos son muy importantes e interesantes para los especialistas en computación interesados en algoritmos.

El mejor libro dedicado absolutamente a la teoría de los problemas NP-completos es el de Garey y Johnson (1979). Ahí se narra la historia del desarrollo del NP-completo y siempre puede utilizarse como enciclopedia para el tema. En este libro no se proporcionó una demostración formal del teorema de Cook, lo cual fue una decisión dolorosa. Descubrimos que sí era posible proporcionar una demostración formal. No obstante, sería muy difícil que el lector comprendiera su verdadero significado. En vez de ello, proporcionamos ejemplos para ilustrar cómo un algoritmo no determinístico puede transformarse en una fórmula booleana de modo que el algoritmo no determinístico termine con un "sí" si y sólo si la fórmula booleana transformada es satisfactible. En el artículo original de Cook (Cook, 1971) pueden consultarse demostraciones formales del teorema de Cook. También es posible consultar a Horowitz y Sahni (1978); Garey y Johnson (1979), y Mandrioli y Ghezzi (1987).

Observe que en la teoría de los problemas NP-completos sólo aplica en el análisis del peor caso. El que un problema sea NP-completo no debe desalentar a nadie en su intento por desarrollar algoritmos eficientes que resuelvan el problema en casos promedio. Por ejemplo, hay muchos algoritmos desarrollados para resolver el problema de satisfactibilidad. Muchos de ellos se basan en el principio de resolución que se analizó en este capítulo. El principio de resolución fue inventado por Robinson (Robinson, 1965) y analizado por Chang y Lee (1973). Recientemente se ha descubierto que varios algoritmos para resolver el problema de satisfactibilidad son polinomiales para casos promedio (Chao, 1985; Chao y Franco, 1986; Franco, 1984; Hu, Tang y Lee, 1992; Purdom y Brown, 1985).

Otro famoso problema NP-completo es el problema del agente viajero, del que puede establecerse su teoría de problema NP-completo reduciendo el problema del ciclo Hamiltoniano (Karp, 1972). Hay todo un libro dedicado a algoritmos para resolver el problema del agente viajero (Lawler, Lenstra, Rinnooy Kan y Shmoys, 1985). El problema de la galería de arte para polígonos simples también es NP-completo (Lee y Lin, 1986). También hay un libro dedicado por completo a teoremas y algoritmos sobre este problema (O'Rourke, 1987).

En (Cook, 1971) se descubrió que el problema de 3-satisfactibilidad es NP-completo. Que el problema de coloración de una gráfica es NP-completo fue establecido

por Karp (1972). Para la teoría de los problemas NP-completos de suma de subconjuntos, partición, empaque en contenedores, consultar a Horowitz y Sahni (1978). El problema de disposición discreta VLSI fue probado como NP-completo por Lapaugh (1980).

El problema de 2-satisfactibilidad fue probado como un problema de tiempo polinomial por Cook (1971) y Even, Itai y Shamir (1976). Este problema es también discutido por Papadimitiriou (1994).

Existen todavía muchos problemas famosos los cuales se sospecha que son NP-completos. Garey y Johnson (1978) dan una lista de ellos. Uno de estos problemas, denominado problema de programación lineal, fue descubierto como un problema polinomial por Khachian (1979). Consultar a Papadimitriou y Steglitz (1982).

Finalmente, existe otro concepto, llamado "promedio-completo". Si un problema es promedio-completo, entonces es difícil aun en los casos promedio. El problema de los azulejos (Berger, 1966) fue probado como promedio-completo por Levin (1986). Levin (1986) es probablemente el artículo más corto jamás publicado, solo una y media páginas. Pero es uno de los artículos publicados más difíciles de asimilar. Si el lector fracasa en entenderlo, no se desanime. Un gran número de científicos en computación no lo comprende.

8-10 Bibliografía adicional

La teoría de los problemas NP-completos es uno de los temas más emocionantes de investigación. Muchos problemas aparentemente sencillos han sido descubiertos como NP-completos. Los autores aconsejamos al lector leer los artículos de Johnson en el "Journal of Algorithms" sobre los avances en esta área. Los autores recomiendan los siguientes artículos; la mayoría de ellos fueron publicados recientemente y no pueden ser encontrados en ningún libro de texto: Ahuja (1988); Bodlaender (1988); Boppana, Hastad y Zachos (1987); Cai y Meyer (1987); Chang y Du (1988); Chin y Ntafos (1988); Cheng y Krishnamoorthy (1994); Colbourn, Kocay y Stinson (1986); Du y Book (1989); Du y Leung (1988); Fellows y Langston (1988); Flajolet y Prodiuger (1986); Friesen y Langston (1986); Homer (1986); Homer y Long (1987); Huynh (1986); Jonson, Yanakakis y Papadimitriou (1988); Kirousis y Papadimitriou (1988); Krivanek y Moravek (1986); Levin (1986); Megido y Supowit (1984); Monien y Sudborough (1988); Murgolo (1987); Papadimitriou (1981); Ramanan, Deogun y Lin (1984); Wu y Sahni (1988); Sarrafzadeh (1987); Tang Buehrer y Lee (1985); Valiant y Vazirani (1986); Wang y Kuo (1988) y Yannakakis (1989).

Para resultados más recientes, consultar: Agrawal, Kayak y Saxena (2004); Berger y Leighton (1998); Bodlaender, Fellows y Hallet (1994); Boros, Crama, Hammer y Saks (1994); Caprara (1997a); Caprara (1997b); Caprara (1999); Feder y Motwani

(2002); Ferreira, de Souza y Wakabayashi (2002); Foulds y Grahan (1982); Galil, Haber y Yung (1989); Goldberg, Goldberg y Paterson (2001); Goldstein y Waterman (1987); Grove (1995); Hochbaum (1997); Holyer (1981); Kannan y Warnow (1994); Kearney, Hayward y Meijer (1997); Lathrop (1994); Lent y Mahmoud (1996); Lipton (1995); Lyngso y Pedersen (2000); Ma, Li y Zhang (2000); Maier y Storer (1977); Pe'er y Shamir (1998); Pierce y Winfree (2002); Storer (1977); Thomassen (1997); Unger y Moult (1993) y Wareham (1995).

Ejercicios

8.1 Determine si las siguientes afirmaciones son correctas o no.

1. Si un problema es NP-completo, entonces no es posible resolverlo con ningún algoritmo en los peores casos.
2. Si un problema es NP-completo, entonces no se ha descubierto ningún algoritmo polinomial para resolverlo en los peores casos.
3. Si un problema es NP-completo, entonces es improbable encontrar en el futuro un algoritmo polinomial para resolverlo en los peores casos.
4. Si un problema es NP-completo, entonces es improbable encontrar un algoritmo polinomial para resolverlo en los casos promedio.
5. Si puede demostrarse que la cota inferior de un problema NP-completo es exponencial, entonces se ha demostrado que NP $\neq$ P.

8.2 Determine la satisfactibilidad de cada uno de los siguientes conjuntos de cláusulas.

1. $-X_1 \vee -X_2 \vee X_3$
   X_1
   $X_2 \vee X_3$
   $-X_3$
2. $X_1 \vee X_2 \vee X_3$
   $-X_1 \vee X_2 \vee X_3$
   $X_1 \vee -X_2 \vee X_3$
   $X_1 \vee X_2 \vee -X_3$
   $-X_1 \vee -X_2 \vee X_3$

$$
\begin{array}{llllll}
 & X_1 & \vee & -X_2 & \vee & -X_3 \\
 & -X_1 & \vee & X_2 & \vee & -X_3 \\
 & -X_1 & \vee & -X_2 & \vee & -X_3 \\
3. & -X_1 & \vee & X_2 & \vee & X_3 \\
 & X_1 & \vee & X_2 & & \\
 & X_3 & & & & \\
4. & X_1 & \vee & X_2 & \vee & X_3 \\
 & X_1 & & & & \\
 & X_2 & & & & \\
5. & -X_1 & \vee & X_2 & & \\
 & -X_2 & \vee & X_3 & & \\
 & -X_3 & & & &
\end{array}
$$

8.3 Todos sabemos cómo demostrar que un problema es NP-completo. ¿Cómo puede demostrarse que un problema no es NP-completo?

8.4 Termine la demostración del NP-completo del problema de la cubierta exacta según se describió en este capítulo.

8.5 Termine la demostración del NP-completo del problema de la suma de subconjuntos según se describió en este capítulo.

8.6 Considere el siguiente problema. Dadas dos variables de entrada a y b, regresar "**SÍ**" si $a > b$ y "**NO**" en caso contrario. Diseñe un algoritmo polinomial no-determinístico para resolver este problema. Transfórmelo en una fórmula booleana de modo que el algoritmo regrese "**SÍ**" si y sólo si la fórmula booleana es satisfactible.

8.7 Problema de decisión del círculo maximal: un círculo maximal es una subgráfica completa maximal de una gráfica. El tamaño de un círculo maximal es el número de vértices que hay en él. El problema de decisión del círculo consiste en determinar si existe o no un círculo maximal de tamaño por lo menos igual a k para alguna k en una gráfica. Demuestre que el problema de decisión del círculo maximal es NP-completo, reduciendo el problema de satisfactibilidad a él.

8.8 Problema de decisión de la cubierta de vértices: un conjunto S de vértices de una gráfica es una cubierta de vértices de esta gráfica si y sólo si

todos las aristas de la gráfica inciden en por lo menos un vértice en *S*. El problema de decisión de la cubierta de vértices consiste en determinar si una gráfica tiene una cubierta de vértices que en el peor de los casos tenga *k* vértices. Demuestre que el problema de decisión de la cubierta de vértices es NP-completo.

8.9 Problema de decisión del agente viajero: demuestre que el problema de decisión del agente viajero es NP-completo, probando que el problema de decisión del ciclo Hamiltoniano se reduce polinomialmente a él. La definición del problema de decisión del ciclo Hamiltoniano puede consultarse en casi todos los libros de texto sobre algoritmos.

8.10 Problema de decisión del conjunto independiente: dados una gráfica *G* y un entero *k*, el problema de decisión del conjunto independiente consiste en determinar si existe un conjunto *S* de *k* vértices de modo que ningún par de vértices en *S* esté unido por una arista. Demuestre que el problema de decisión del conjunto independiente es NP-completo.

8.11 Problema de decisión del agente viajero en un cuello de botella: dados una gráfica *G* y un número *M*, este problema consiste en determinar si en esta gráfica existe un ciclo Hamiltoniano de modo que la arista más larga de este ciclo sea menor que *M*. Demuestre que el problema de decisión del agente viajero en un cuello de botella es NP-completo.

8.12 Demuestre que 3-coloreado $\propto$ 4-coloreado $\propto$ *k*-coloreado.

8.13 Problema de satisfactibilidad de una cláusula monótona: una fórmula es monótona si cada cláusula de la fórmula contiene sólo variables positivas o sólo variables negativas. Por ejemplo

$$F = (X_1 \vee X_2) \,\&\, (-X_3) \,\&\, (-X_2 \vee -X_3)$$

es una fórmula monótona. Demuestre que el problema de decidir si una fórmula monótona es satisfactible o no es NP-completo.

8.14 Lea el teorema 15.7 del libro de Papadimitriou y Steiglitz (1982) sobre si el problema de apareamiento 3 dimensional es NP-completo.

capítulo

9

Algoritmos de aproximación

En capítulos previos, todo algoritmo presentado proporciona soluciones óptimas. Resulta evidente que lo anterior tiene un precio: el tiempo consumido para producir esas soluciones óptimas llega a ser muy largo. Si el problema es un NP-completo, entonces nuestros algoritmos que producen soluciones óptimas podrían consumir tiempo de una manera muy poco práctica.

Una solución alternativa consiste en usar fórmulas heurísticas. La palabra "heurística" puede interpretarse como "conjetura hecha con cierta base". Así, un algoritmo heurístico es un algoritmo basado en conjeturas con cierto fundamento. Debido a que un algoritmo heurístico "adivina", no hay garantía de que produzca soluciones óptimas. De hecho, no lo hace.

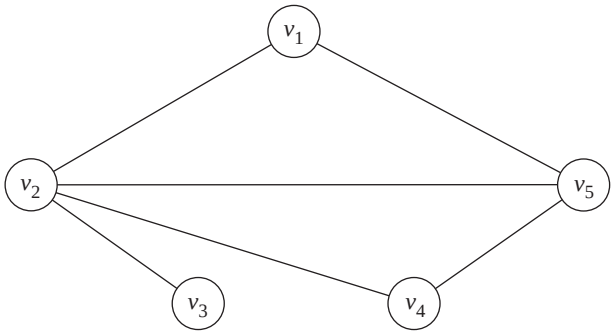
En este capítulo se presentarán varios algoritmos heurísticos. Aunque este tipo de algoritmos no garantiza soluciones óptimas, sí garantiza que los errores provocados por sus soluciones no óptimas pueden estimarse de antemano. Todos estos algoritmos se denominan algoritmos de aproximación.

9-1 Un algoritmo de aproximación para el problema de cubierta de nodos

Dada una gráfica $G = (V, E)$, un conjunto S de nodos en V se denomina cubierta de nodos de G si todo nodo incide en algún nodo en S. El tamaño de una cubierta de nodos S es el número de nodos en S. El problema de la cubierta de nodos consiste en encontrar una cubierta de nodos S de $G = (V, E)$ de tamaño mínimo. Por ejemplo, $\{v_1, v_2, v_4\}$ es una cubierta de nodos de la gráfica que se muestra en la figura 9-1, y $\{v_2, v_5\}$ es una cubierta de nodos de tamaño mínimo.

Se ha demostrado que el problema de la cubierta de nodos es NP-difícil. Para resolver este problema se introduce un algoritmo de aproximación, el algoritmo 9-1.

FIGURA 9-1 Una gráfica de ejemplo.



Algoritmo 9-1 □ Un algoritmo de aproximación para el problema de la cobertura de vértices

Input: Una gráfica $G = (V, E)$.
Output: Una cobertura de vértices S de G.
Paso 1. Hacer $S = \phi$ y $E' = E$.
Paso 2. Mientras $E' \neq \phi$
Elegir arbitrariamente una arista (a, b) en E'.
Hacer $S = S \cup \{a, b\}$ y E'' el conjunto de aristas en E' incidentes en a o b. Sea $E' = E' - E''$.
end while (fin de ciclo)
Paso 3. Output S.

Se considerará la gráfica de la figura 9-1. En el paso 1 del algoritmo 9-1, $S = \phi$ y $E' = \{(v_1, v_2), (v_1, v_5), (v_2, v_3), (v_2, v_4), (v_2, v_5), (v_4, v_5)\}$. Debido a que $E' = \phi$, se ejecuta el ciclo en el paso 2. Suponga que se escoge la arista (v_2, v_3) de E'. Entonces se tiene $S = \{v_2, v_3\}$ y $E' = \{(v_1, v_5), (v_4, v_5)\}$ en el final del ciclo. Luego, se ejecuta el segundo ciclo en el paso 2 porque $E' = \phi$. Suponga que se escoge la arista (v_1, v_5) de E'. Entonces se tiene $S = \{v_1, v_2, v_3, v_5\}$ y $E' = \phi$ en el final del segundo ciclo. Finalmente, como una cobertura de vértices para G se reporta $\{v_1, v_2, v_3, v_5\}$.

La complejidad temporal del algoritmo anterior es $O(|E|)$. Se demostrará que el tamaño de una cobertura de vértices para G es a lo más dos veces el tamaño mínimo de una cobertura de vértices de G. Sea M^* el tamaño mínimo de una cobertura de vértices de G. Sea M el tamaño de una cobertura de vértices de G encontrada con el algoritmo 9-1. Sea L el número total de aristas escogidas en el algoritmo. Debido a que la arista

se escoge en cada ciclo del paso 2 y hay dos vértices incidentes por una arista, se tiene $M = 2L$. Como ningún par de aristas escogidos en el paso 2 comparte ningún vértice final, se requieren por lo menos L nodos para cubrir estos L aristas. Así se tiene $L \leq M^*$. Finalmente, $M = 2L \leq 2M^*$.

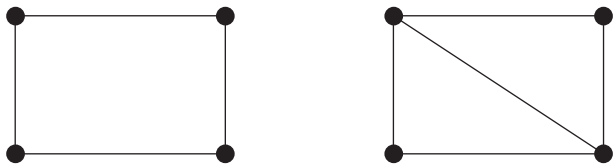
9-2 UN ALGORITMO DE APROXIMACIÓN PARA EL PROBLEMA DEL AGENTE VIAJERO VERSIÓN EUCLIDIANA

El problema euclidiano del agente viajero (PAVE) consiste en encontrar la ruta cerrada más corta que pase por un conjunto S de n puntos en el plano. Ya se ha demostrado que es NP-difícil. Así, es muy improbable que exista un algoritmo eficiente para el peor caso del problema euclidiano del agente viajero.

En este apartado se describirá un algoritmo de aproximación para el problema euclidiano del agente viajero, que puede encontrar un recorrido inferior a 3/2 del óptimo. Es decir, si la longitud del recorrido óptimo es L, entonces la longitud de este recorrido aproximado no es mayor que $(3/2)L$.

La idea básica de este esquema de aproximación es construir un ciclo euleriano del conjunto de puntos y luego usarlo para encontrar un recorrido aproximado. *Un ciclo euleriano de una gráfica es un ciclo en la gráfica en el que todo vértice se visita no necesariamente una vez, y cada arista aparece exactamente una vez. Una gráfica se denomina euleriana si tiene un ciclo euleriano. Euler demostró que una gráfica G es euleriana si y sólo si G es conexa y todos los vértices de G tienen grados pares.* Considere la figura 9-2. Puede verse fácilmente que la gráfica en la figura 9-2*a*) es euleriana y que la de la figura 9-2*b*) no lo es. Más tarde se demostrará que después de encontrar un ciclo euleriano es posible encontrar un ciclo hamiltoniano a partir de aquél simplemente recorriendo y evitando todos los vértices previamente visitados.

FIGURA 9-2 Gráficas con y sin ciclos eulerianos.

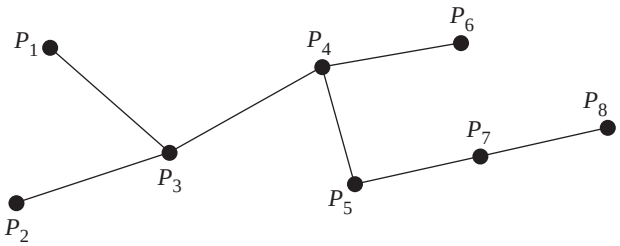


a) *b*)

Entonces, si es posible construir una gráfica conexa G del conjunto de puntos S (es decir, usar los puntos en S como los vértices de G) de modo que el grado de cada vértice de G sea par, entonces puede construirse un ciclo euleriano y después encontrar un ciclo hamiltoniano para aproximar el recorrido óptimo del agente viajero.

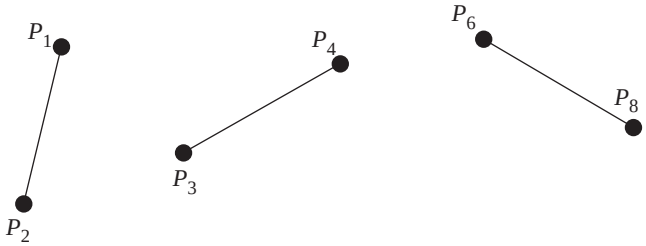
El esquema de aproximación consta de tres pasos: primero, se construye un árbol de expansión mínimo de S. Por ejemplo, considere la figura 9-3. Para el conjunto de puntos $S_1 = \{P_1, P_2, \ldots, P_8\}$ con ocho puntos, en la figura 9-3 se muestra un árbol de expansión mínimo de S_1.

FIGURA 9-3 Árbol de expansión mínimo de ocho puntos.



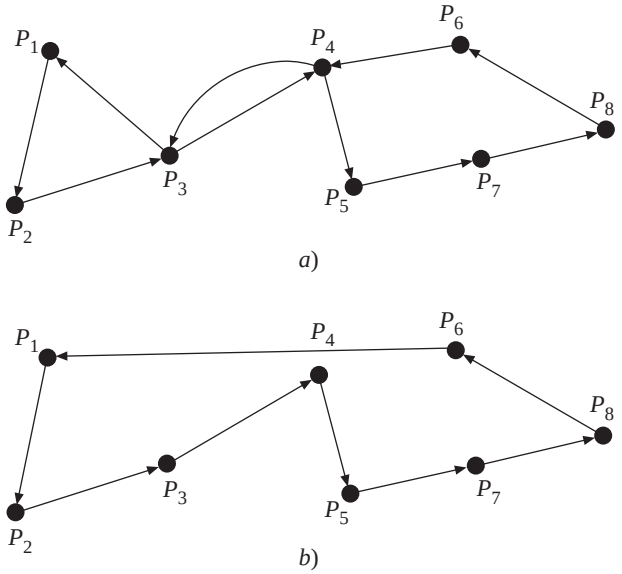
Aunque el árbol de expansión mínimo es conexo, puede contener vértices de grado impar. De hecho, puede demostrarse que en cualquier árbol, siempre hay un número par de tales vértices. Por ejemplo, el grado de cada vértice en $\{P_1, P_2, P_3, P_4, P_6, P_8\}$ en la figura 9-3 es impar. El segundo paso consiste en construir una gráfica euleriana; es decir, deben agregarse algunas aristas adicionales para que todos los vértices sean de grado par. El problema se convierte en ¿cómo encontrar este conjunto de aristas? En este algoritmo se utiliza el apareamiento ponderado euclidiano mínimo para el conjunto de vértices con grados impares. *Dado un conjunto de puntos en el plano, el problema del apareamiento ponderado euclidiano mínimo (minimum Euclidean weighted matching) consiste en unir los puntos en pares por segmentos de recta de modo que la longitud total sea mínima.* Si es posible encontrar un apareamiento ponderado euclidiano mínimo sobre los vértices con grados impares y aumentar este conjunto de aristas de apareamiento al conjunto de aristas del árbol de expansión mínimo, entonces la gráfica resultante es euleriana. Por ejemplo, el apareamiento ponderado euclidiano mínimo para los vértices P_1, P_2, P_3, P_4, P_6 y P_8 se ilustra en la figura 9-4.

FIGURA 9-4 Apareamiento ponderado mínimo de seis vértices.



Después de que los vértices en la figura 9-4 se agregan al árbol de expansión mínimo en la figura 9-3, todos los vértices son de grado par. El tercer paso del algoritmo consiste en construir un ciclo euleriano a partir de la gráfica resultante y luego obtener un ciclo hamiltoniano a partir de este ciclo euleriano. El ciclo euleriano del conjunto de ocho puntos es P_1–P_2–P_3–P_4–P_5–P_7–P_8–P_6–P_4–P_3–P_1 (observe que P_3 y P_4 se visitan dos veces) y se muestra en la figura 9-5*a*). Al evitar P_4 y P_3 y unir directamente P_6 con P_1 se obtiene un ciclo hamiltoniano. Por lo tanto, el recorrido aproximado resultante es P_1–P_2–P_3–P_4–P_5–P_7–P_8–P_6–P_1, que se muestra en la figura 9-5*b*).

FIGURA 9-5 Ciclo euleriano y el recorrido aproximado resultante.



Algoritmo 9-2 □ Un algoritmo de aproximación para el PEAV

Input: Un conjunto S de n puntos en el plano.
Output: Un recorrido aproximado de S del agente de ventas.
Paso 1. Buscar un árbol de expansión mínima T de S.
Paso 2 Buscar un apareamiento ponderado euclidiano mínimo M sobre el conjunto de vértices de grados impares en T. Hacer $G = M \bigcup T$.
Paso 3. Buscar un ciclo euleriano de G y luego recorrerlo para encontrar un ciclo hamiltoniano como un recorrido aproximado del PAVE evitando todos los vértices previamente visitados.

El análisis de la complejidad temporal de este algoritmo de aproximación es fácil. El paso 1 puede ejecutarse en tiempo $O(n \log n)$. El apareamiento ponderado euclidiano mínimo puede encontrarse en tiempo $O(n^3)$, el ciclo euleriano puede construirse en tiempo lineal y después en tiempo lineal puede obtenerse un ciclo hamiltoniano a partir de aquél. Así, la complejidad temporal del algoritmo anterior es $O(n^3)$. Debido a que el objetivo principal de este análisis es presentar algoritmos de aproximación, no se proporcionarán detalles de algoritmos para encontrar el apareamiento mínimo y el ciclo euleriano.

Sea L un recorrido óptimo en S del agente viajero. Si de L se quita una arista, se obtiene una ruta L_p que también es un árbol de expansión de S. Sea T un árbol de expansión mínimo de S. Entonces longitud(T) $\leq$ longitud(L_p) $\leq$ longitud(L) porque T es un árbol de expansión mínimo de S. Sea $\{i_1, i_2, \ldots, i_{2m}\}$ el conjunto de vértices de grado impar en T, donde los índices de los vértices en el conjunto están dispuestos en el mismo orden que tienen en el recorrido óptimo L del agente viajero. Considere los dos apareamientos de los vértices de grado impar $M_1 = \{[i_1, i_2], [i_3, i_4], \ldots, [i_{2m-1}, i_{2m}]\}$ y $M_2 = \{[i_2, i_3], [i_4, i_5], \ldots, [i_{2m}, i_1]\}$. Es fácil ver que longitud(L) $\geq$ longitud(M_1) + longitud(M_2) porque las aristas satisfacen la desigualdad del triángulo. Sea M el apareamiento ponderado euclidiano mínimo óptimo de $\{i_1, i_2, \ldots, i_{2m}\}$. Así, la longitud del más corto de estos dos apareamientos es mayor que longitud(M). Así, longitud(M) $\leq$ longitud(L)/2. Debido a que $G = T \cup M$, longitud(G) = longitud(T) + longitud(M) $\leq$ longitud(L) + longitud(L)/2 $\leq$ 3/2 (longitud(L)).

Debido a que nuestra solución, que es un ciclo hamiltoniano, es más corta que la longitud de G, se tiene un algoritmo de aproximación para el problema euclidiano del agente viajero que puede producir en tiempo $O(n^3)$ un recorrido aproximado a menos de 3/2 de un recorrido óptimo.

9-3 ALGORITMO DE APROXIMACIÓN PARA UN PROBLEMA ESPECIAL DE CUELLO DE BOTELLA DEL AGENTE VIAJERO

El problema del agente viajero también puede definirse en gráficas: en este caso se tiene una gráfica y se trata de encontrar el recorrido cerrado más corto, que empieza en algún vértice, visita los otros vértices en la gráfica y regresa al vértice inicial. Se ha demostrado que este problema es NP-difícil.

En este apartado se considera otro tipo de problema del agente viajero. Sigue teniéndose interés en un recorrido cerrado. No obstante, no se desea minimizar la longitud de todo el recorrido; en vez de ello, se quiere minimizar la longitud de la arista más larga de este recorrido. En otras palabras, se quiere encontrar un recorrido cuya

arista más larga sea la más corta posible. Entonces, éste es un problema mini-máx, y este tipo de problema del agente viajero se denomina problema de cuello de botella del agente viajero porque pertenece a una clase de problemas de cuello de botella, a menudo relacionados con los de la ciencia del transporte. De hecho, es evidente que muchos de los problemas que se han analizado pueden modificarse para convertirlos en problemas de cuello de botella. Por ejemplo, el problema del árbol de expansión mínimo cuenta con una versión de cuello de botella. En esta versión, se debe encontrar un árbol de expansión cuya arista más larga sea la más corta.

De nuevo, como en los dos apartados previos, se proporcionará un algoritmo de aproximación. Antes de facilitar este algoritmo, la hipótesis se plantea como sigue:

1. *La gráfica es una gráfica completa. Es decir, entre cada par de vértices hay una arista.*
2. *Todas las aristas cumplen con la desigualdad del triángulo*. Así, el problema de cuello de botella del agente viajero es especial. Puede demostrarse que este problema especial de cuello de botella del agente viajero es NP-difícil.

A fin de preparar al lector para nuestro siguiente algoritmo de aproximación, primero se propone un algoritmo que no es polinomial. Este algoritmo produce una solución óptima para el problema de cuello de botella del agente viajero. Luego, este algoritmo se modifica para obtener un algoritmo de aproximación.

Primero se ordenan las aristas de la gráfica $G = (V, E)$ en una secuencia no decreciente $|e_1| \leq |e_2| \ldots \leq |e_m|$. Sea $G(e_i)$ la gráfica que se obtiene a partir de $G = (V, E)$ al eliminar todas las aristas más largas que e_i. Considere $G(e_1)$. Si $G(e_1)$ contiene un ciclo hamiltoniano, entonces este ciclo debe ser la solución óptima que se busca. En caso contrario, considere $G(e_2)$. Este proceso se repite hasta que alguna $G(e_1)$ contiene un ciclo hamiltoniano. Luego, este ciclo se regresa como la solución óptima del problema especial de cuello de botella del agente viajero.

Para ilustrar la idea anterior puede usarse un ejemplo. Considere la figura 9-6. Para encontrar un ciclo hamiltoniano con la arista más larga mínima, para empezar puede empezar a intentarse con $G(AC)$. La longitud de AC es 12. Por consiguiente, $G(AC)$ contiene sólo aristas menores o iguales a 12, lo cual se muestra en la figura 9-7. En $G(AC)$ no hay ningún ciclo hamiltoniano. Luego se intenta con $G(BD)$ porque BD es la siguiente arista más larga. $G(BD)$ se muestra en la figura 9-8. En $G(BD)$ hay un ciclo hamiltoniano; a saber, A–B–D–C–E–F–G–A. Ahora puede concluirse que se ha encontrado una solución óptima y que el valor de ésta es 13.

Nuestro algoritmo no puede ser polinomial porque el problema de encontrar un ciclo hamiltoniano es un problema NP-difícil. A continuación se presentará un algoritmo de aproximación. Éste aún mantiene el espíritu del algoritmo anterior.

FIGURA 9-6 Gráfica completa.

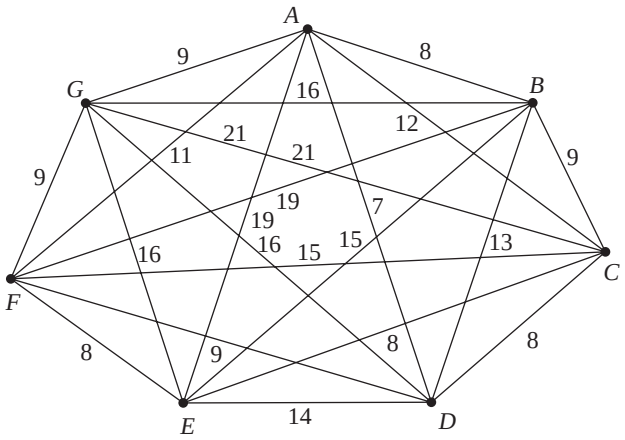
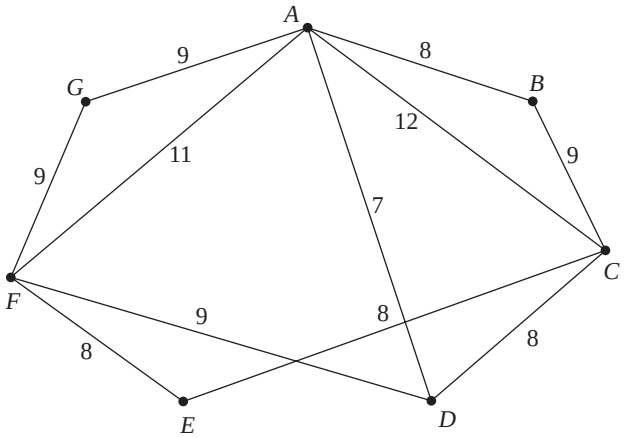


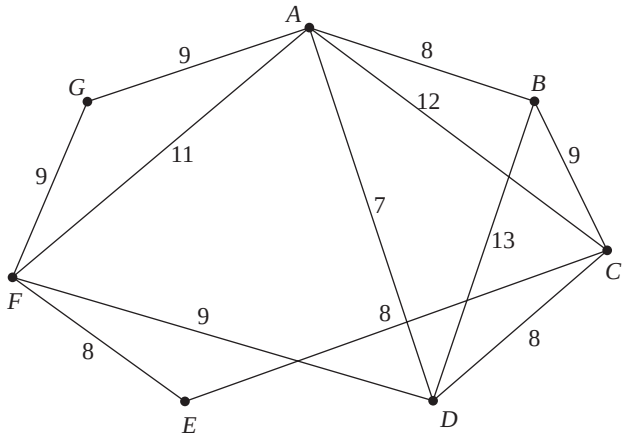
FIGURA 9-7 *G*(*AC*) de la gráfica de la figura 9-6.



Antes de presentar este algoritmo de aproximación se introducirá el concepto de biconectividad. *Una gráfica es biconexa si y sólo si todo par de sus vértices pertenecen por lo menos a un ciclo simple común.*

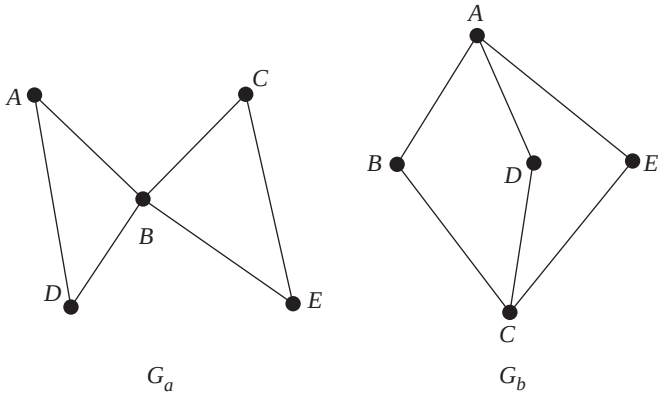
Por ejemplo, en la figura 9-9, G_b es biconexa mientras que G_a, no. G_a no es biconexa porque los vértices *A* y *C* no están en ningún ciclo simple. Por otra parte, en G_b, todo par de vértices pertenece por lo menos a un ciclo común simple. A continuación, se define la potencia de una gráfica. *Si* $G = (V, E)$ *es una gráfica arbitraria y t es un*

FIGURA 9-8 *G(BD)* de la gráfica de la figura 9-6.



entero positivo, entonces sea $G^t = (V, E^t)$, *la t-ésima potencia de G, donde hay una arista (u, v) en* G^t *siempre que haya una ruta de u a v con a lo sumo t aristas en G.*

FIGURA 9-9 Ejemplos que ilustran la biconectividad.



Las gráficas biconexas poseen una propiedad muy importante. Esto es, *si una gráfica G es biconexa, entonces* G^2 *tiene un ciclo hamiltoniano*. Considere la figura 9-10. Resulta evidente que la gráfica *G* de esta figura es biconexa. Por ejemplo, los vértices *D* y *F* están en el mismo ciclo *D–C–F–E–D*. Ahora se considerará G^2 como se muestra en la figura 9-11. En la gráfica de esa figura hay un ciclo hamiltoniano, que es *A–B–C–D–F–E–G–A*.

FIGURA 9-10 Una gráfica biconexa.

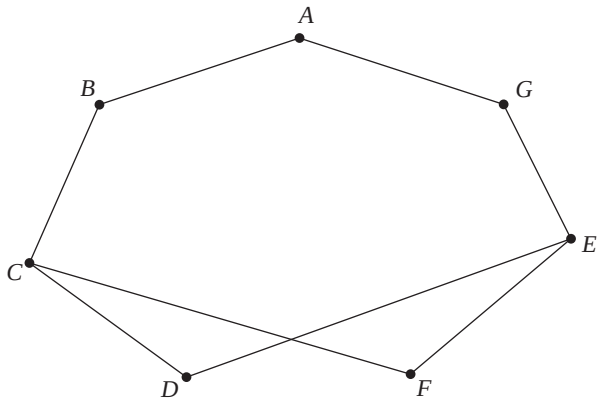
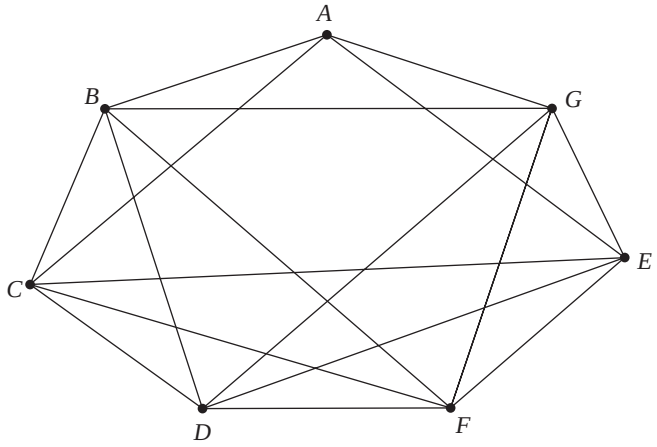


FIGURA 9-11 G^2 de la gráfica de la figura 9-10.



Con base en la propiedad de que para toda gráfica biconexa G, G^2 tiene un ciclo hamiltoniano, es posible usar el método siguiente para obtener una solución aproximada al problema de cuello de botella del agente viajero. *Como ya se hizo, las aristas de G se ordenan en una secuencia no decreciente:* $|e_1| \leq |e_2| \ldots \leq |e_m|$. *Sea* $G(e_i)$ *que sólo contiene las aristas cuya longitud es menor o igual a la longitud de* e_i. *Para empezar se considera* $G(e_1)$. *Si* $G(e_1)$ *es biconexa, se construye* $G(e_1)^2$. *En caso contrario, se considera* $G(e_2)$. *Este proceso se repite hasta que se encuentra la primera i en que* $G(e_i)$ *es biconexa. Luego se construye* $G(e_i)^2$. *Esta gráfica debe contener un ciclo hamiltoniano que puede usarse como la solución por aproximación.*

El algoritmo de aproximación se plantea formalmente como sigue:

Algoritmo 9-3 □ Un algoritmo de aproximación para resolver el problema especial de cuello de botella del agente viajero

Input: Una gráfica completa $G = (V, E)$ donde todas las aristas satisfacen la desigualdad del triángulo.

Output: Un recorrido en G cuya arista más larga no es mayor que el doble del valor de una solución óptima del problema especial de cuello de botella del agente viajero de G.

Paso 1. Ordenar las aristas en $|e_1| \leq |e_2| \leq \ldots \leq |e_m|$.

Paso 2. $i := 1$.

Paso 3. Si $G(e_i)$ es biconexa, construir $G(e_i)^2$; buscar un ciclo hamiltoniano en $G(e_i)$ y regresar esto como output; en caso contrario, ir a paso 4.

Paso 4. $i := i + 1$. Ir a paso 3.

A continuación se ilustrará este algoritmo con un ejemplo. Considere la figura 9-12, que muestra $G(FE)$ ($|FE| = 8$) donde G es la gráfica de la figura 9-6. $G(FE)$ no es biconexa. Ahora se considera $G(FG)$ porque $|FG| = 9$. $G(FG)$ es biconexa, lo cual se muestra en la figura 9-13. Luego se construye $G(FG)^2$, que se muestra en la figura 9-14. En esta gráfica hay un ciclo hamiltoniano que es *A–G–E–F–D–C–B–A*. La longitud de la arista más larga de este ciclo es 16. Debe observarse que ésta no es una solución óptima, ya que el valor de una solución óptima para el problema de cuello de botella del agente viajero de la gráfica en la figura 9-6 es 13, que es menor que 16.

FIGURA 9-12 $G(FE)$ de la gráfica de la figura 9-6.

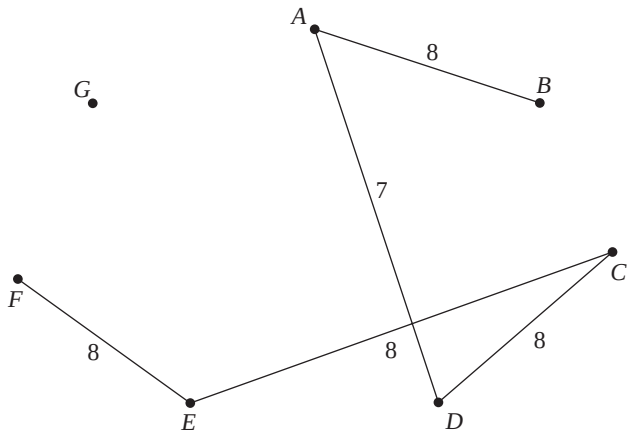


FIGURA 9-13 $G(FG)$ de la gráfica en la figura 9-6.

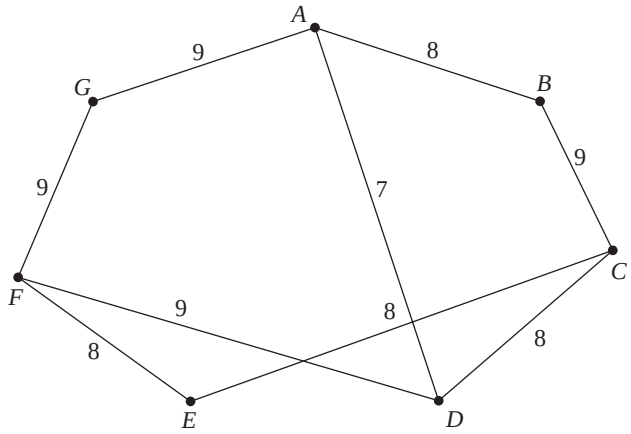
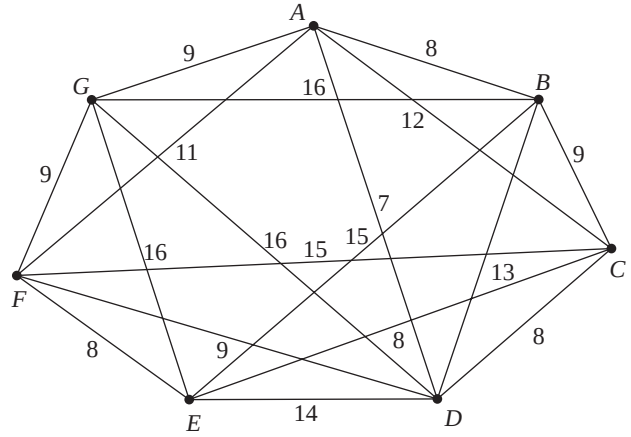


FIGURA 9-14 $G(FG)^2$.



A continuación se analizarán tres preguntas relacionadas con el algoritmo de aproximación:

1. ¿Cuál es la complejidad temporal de nuestro algoritmo de aproximación?
2. ¿Cuál es la cota de la solución aproximada?
3. Esta cota, ¿es óptima en cierto sentido?

Se contestará la primera pregunta. Se observa que la determinación de biconectividad puede resolverse con algoritmos polinomiales.

Además, si G es biconexa, entonces existe un algoritmo polinomial para resolver el problema del ciclo hamiltoniano de G^2. Así, nuestro algoritmo de aproximación es polinomial.

Ahora se intentará contestar la segunda pregunta: ¿cuál es la cota de la solución aproximada? Considere el siguiente problema: una subgráfica de aristas de $G = (V, E)$ es una subgráfica de G con conjunto de vértices V. De todas las subgráficas de aristas de G, encontrar una subgráfica de aristas que sea biconexa y la arista más larga se minimice. Resulta evidente que $G(e_i)$ encontrado en el algoritmo de aproximación antes mencionado, que es biconexa, debe ser una solución de este problema. Sea e_{op} la arista más larga de una solución óptima del problema especial de cuello de botella del agente viajero. Obviamente, $|e_i| \leq |e_{op}|$ porque cualquier solución del problema del agente viajero es biconexa. Una vez que se encuentra $G(e_i)$, se construye $G(e_i)^2$. La longitud de la arista más larga de $G(e_i)^2$ no puede ser mayor que $2|e_i|$ porque todas las aristas cumplen la desigualdad del triángulo. *En consecuencia, el valor de la solución que se obtiene con el algoritmo de aproximación no es mayor que* $2|e_i| \leq 2|e_{op}|$. *Es decir, aunque nuestro algoritmo de aproximación no produce una solución óptima, el valor de esta solución sigue siendo acotado por dos veces la cota de una solución óptima*.

A continuación comprobaremos nuestra afirmación contra el ejemplo proporcionado. En el ejemplo anterior, $|e_{op}| = 13$. La solución producida por el algoritmo de aproximación es 16. Debido a que $16 \leq 2 \cdot 13 = 26$, en este caso nuestra afirmación es correcta.

Finalmente, se contestará la tercera pregunta: la cota de la solución óptima, ¿es óptima? Es decir, ¿puede tenerse otro algoritmo de aproximación polinomial que produzca una solución aproximada cuya cota sea menor que dos? Por ejemplo, ¿es posible que haya un algoritmo de aproximación polinomial que produzca una solución aproximada cuya arista más larga sea menor o igual a 1.5 veces el de una solución óptima para todos los casos del problema?

A continuación se demostrará que lo anterior es probablemente imposible porque si existe un algoritmo de aproximación así, que es polinomial, entonces este algoritmo puede usarse para resolver un problema NP-completo. Es decir, si hay un algoritmo de aproximación polinomial que produzca una cota menor que dos, entonces NP = P. El problema particular NP-completo que usaremos es el problema de decisión del ciclo hamiltoniano.

Suponga que existe un algoritmo polinomial A para el problema especial de cuello de botella del agente viajero de modo que A garantiza producir una solución menor que $2v^*$ para toda gráfica completa G_c cuyas aristas cumplen la desigualdad del triángulo donde v^* es el valor de una solución óptima del problema especial de cuello de botella del agente viajero de G_c. Para una gráfica arbitraria $G = (V, E)$, es posible transformar G en una gráfica completa G_c y definir los valores de las aristas de G_c como

$c_{ij} = 1$ si $(i, j) \in E$

$c_{ij} = 2$ en caso contrario.

Es claro que la definición anterior de c_{ij} cumple la desigualdad del triángulo. Ahora se intentará resolver el problema especial de cuello de botella del agente viajero de G_c. Resulta evidente que

$v^* = 1$ si G es hamiltoniano

y $v^* = 2$ en caso contrario.

En forma equivalente, $v^* = 1$ si y sólo si G es hamiltoniano.

En otras palabras, si es posible resolver el problema especial de cuello de botella del agente viajero de G_c, entonces es posible resolver el problema de decisión del ciclo hamiltoniano de G. Simplemente se comprueba el valor de v. G es hamiltoniana si y sólo si* $v^* = 1$. A continuación se demostrará que el algoritmo de aproximación puede usarse para resolver el problema del ciclo hamiltoniano, a pesar de que *A* es sólo un algoritmo de aproximación del problema de cuello de botella de agente viajero.

Como hipótesis, *A* puede producir una solución de aproximada cuyo valor es menor que $2v^*$. Considere el caso en que *G* es hamiltoniana. Entonces el algoritmo de aproximación *A* produce una solución aproximada cuyo valor V_A es menor que $2v^* = 2$.

Esto es, si G es hamiltoniana, entonces $V_A = 1$ *porque sólo hay aristas con dos tipos de pesos,* 1 *y* 2. *En consecuencia, se afirma que el algoritmo A puede usarse para resolver el problema de decisión del ciclo hamiltoniano de G. Si A es polinomial, entonces NP = P porque el problema de decisión del ciclo hamiltoniano es NP completo*.

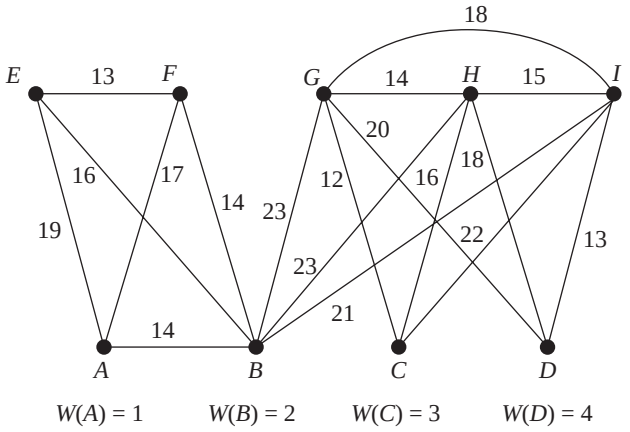
El razonamiento anterior indica que es bastante improbable que exista un algoritmo así.

9-4 UN ALGORITMO DE APROXIMACIÓN PARA UN PROBLEMA PONDERADO ESPECIAL DEL CUELLO DE BOTELLA DEL *K*-ÉSIMO PROVEEDOR

Este problema es otro ejemplo de nuestro algoritmo de aproximación. En este problema se tiene una gráfica completa cuya totalidad de aristas satisface la desigualdad del triángulo. El peso de una arista puede considerarse como una distancia. Todos los vértices están separados en un conjunto proveedor V_{prov} *y un conjunto consumidor* V_{cons}. *Además, cada vértice proveedor i tiene un peso* w_i *que indica el costo de construcción de un centro de suministro en este vértice. El problema consiste en escoger un conjunto de proveedores cuyo peso total sea cuando mucho igual a k, de modo que la distancia más larga entre proveedores y consumidores se minimice.*

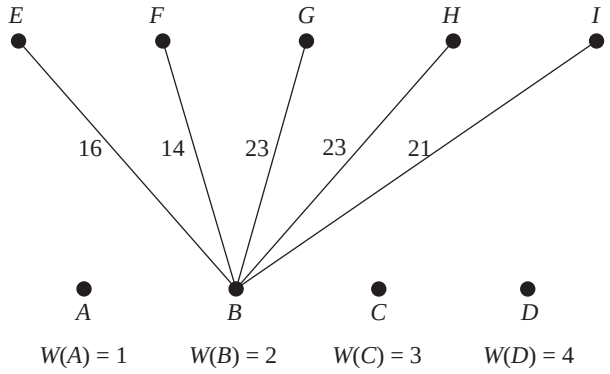
Se considerará la figura 9-15, que tiene nueve vértices, $V = \{A, B, C, \ldots, I\}$, $V_{prov} = \{A, B, C, D\}$ y $V_{const} = \{E, F, G, H, I\}$. Los costos de los proveedores *A*, *B*, *C* y *D* son 1, 2, 3 y 4, respectivamente. Observe que la gráfica que se muestra en la figura 9-15 debe ser una gráfica completa. Todas las aristas que faltan en la figura 9-15 tienen costos mayores que el costo máximo que se muestra en la figura 9-15, aunque todas las aristas de costos satisfacen la desigualdad del triángulo.

FIGURA 9-15 Caso de un problema del *k*-proveedor ponderado.



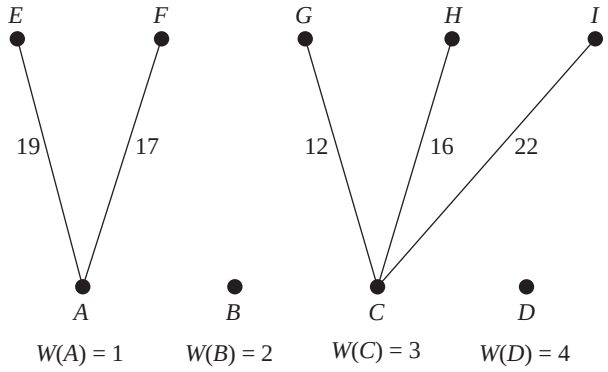
Suponga que $k = 5$. En la figura 9-16 se muestran varias soluciones factibles. La solución óptima se muestra en la figura 9-16*d*) y el valor de esta solución es 20.

FIGURA 9-16 Soluciones factibles para el caso del problema en la figura 9-15.

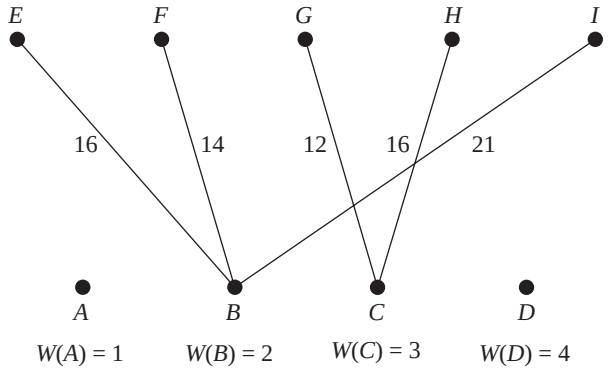


a) Máximo = 23

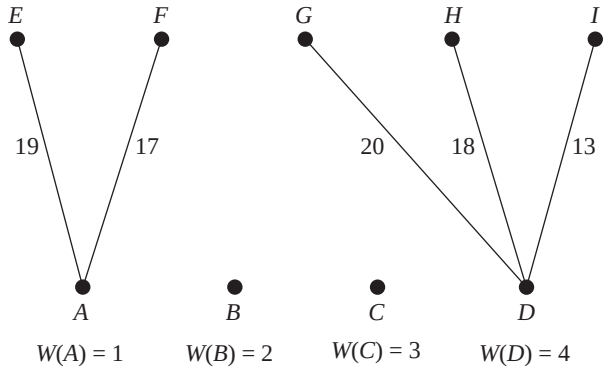
FIGURA 9-16 (*Continuación.*)



b) Máximo = 22



c) Máximo = 21



d) Máximo = 20

Nuestro algoritmo de aproximación nuevamente es semejante al algoritmo de aproximación propuesto en el apartado 9-3, que se usó para resolver el problema especial de cuello de botella del agente viajero. *Para resolver este problema especial ponderado del cuello de botella del k-ésimo proveedor, las aristas se ordenan en una secuencia no decreciente:* $|e_1| \leq |e_2| \ldots \leq |e_m|$. *Como se hizo antes, sea* $G(e_i)$ *la gráfica que se obtiene a partir de* $G(V, E)$ *al eliminar todas las aristas que son más largas que* e_i. *De modo ideal, nuestro algoritmo debe probar* $G(e_1)$, $G(e_2)$,…, *y así sucesivamente hasta que encuentra el primer* $G(e_i)$ *tal que* $G(e_i)$ *contenga una solución factible para el problema ponderado del k-ésimo proveedor. Esta solución factible debe ser óptima. Desafortunadamente, como resulta fácil imaginar, determinar si una gráfica contiene una solución factible para el problema ponderado del k-ésimo proveedor es un problema NP-completo.* En consecuencia, es necesario modificar este método para obtener un algoritmo de aproximación.

En nuestro algoritmo de aproximación se utilizará un procedimiento de prueba que cumple las siguientes propiedades:

1. Si el procedimiento regresa "SÍ", entonces se obtiene una solución inducida como solución del algoritmo de aproximación.
2. Si el procedimiento regresa "NO", entonces se tiene la certeza de que $G(e_i)$ no contiene ninguna solución factible.

Si el procedimiento de prueba no cumple las condiciones anteriores, y las aristas de la gráfica satisfacen la desigualdad del triángulo, entonces resulta fácil demostrar que el algoritmo de aproximación produce una solución aproximada cuyo valor es menor o igual a tres veces el valor de una solución óptima.

Sean V_{op} y V_{ap} las soluciones óptima y aproximada, respectivamente. Debido a la propiedad (2), $|e_i| \leq V_{op}$.

Además, debido a la propiedad (1) y por razones que serán explicadas después,

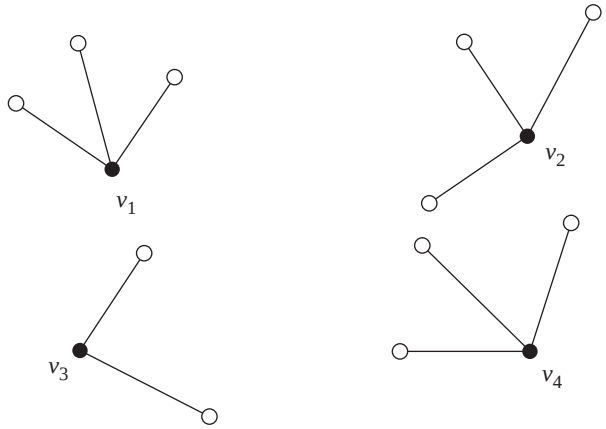
$$V_{ap} \leq 3|e_i|.$$

En consecuencia, se tiene

$$V_{ap} \leq 3V_{op}.$$

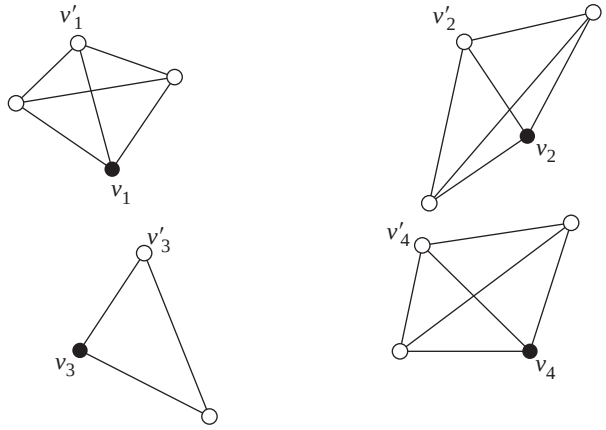
A continuación se explicará cómo funciona el procedimiento de prueba. Imagine cualquier $G(e_i)$ que contiene una solución factible. Entonces todo vértice consumidor debe estar unido por lo menos a un vértice proveedor, lo cual se ilustra en la figura 9-17, donde v_1, v_2, v_3 y v_4 son los vértices proveedores y todos los demás vértices son vértices consumidores. Para todo v_i hay varios vértices consumidores unidos a éste. Considere $G(e_i)^2$ de la gráfica $G(e_i)$ en la figura 9-17.

FIGURA 9-17 Un $G(e_i)$ que contiene una solución factible.



$G(e_i)^2$ se muestra en la figura 9-18. Debido a que $G(e_i)$ contiene una solución factible y cada vértice consumidor está unido por lo menos con un vértice proveedor, entonces $G(e_i)^2$ contiene varios circuitos. En un circuito, todo par de vértices está conectado.

FIGURA 9-18 Un G^2 de la gráfica en la figura 9-17.



De cada circuito es posible seleccionar un vértice consumidor. El resultado es un conjunto independiente maximal. *Un conjunto independiente de una gráfica $G(V, E)$ es un subconjunto $V' \subset V$ tal que cada arista e de E incide a lo sumo en un vértice en V'. Un conjunto independiente maximal de una gráfica es un conjunto independiente que no está contenido propiamente en ningún conjunto independiente*. Como se muestra

en la figura 9-18, $\{v'_1, v'_2, v'_3, v'_4\}$ es un conjunto independiente maximal. Después de que se encuentra un conjunto independiente maximal, para cada vértice consumidor v'_i, se encuentra un vértice proveedor unido a éste con peso mínimo. Puede demostrarse que este conjunto proveedor creado al revés es una solución factible en $G(e_i)^3$.

El procedimiento de prueba (en $G(e_i)$ y $G(e_i)^2$) consta de los pasos siguientes:

1. Si hay un vértice v en V_{cons}, que no sea adyacente a ningún vértice proveedor en $G(e_i)$, regresa "NO".
2. En caso contrario, se encuentra un conjunto independiente maximal S en V_{cons} de $G(e_i)^2$.
3. Para todo vértice v' en S, se encuentra un vértice proveedor adyacente v_i con peso mínimo en $G(e_i)$. Sea S' el conjunto de estos vértices.
4. Si el peso total de $w(v_i)$ en S' es menor o igual a k, regresa "SÍ". En caso contrario, regresa "NO".

Si el resultado de la prueba es "NO", resulta evidente que $G(e_i)$ no puede contener ninguna solución factible. Si el resultado de la prueba es "SÍ", puede usarse el conjunto proveedor inducido como la solución de nuestro algoritmo de aproximación. Así, nuestro algoritmo de aproximación puede plantearse como sigue en el algoritmo 9-4.

Algoritmo 9-4 □ Un algoritmo de aproximación para resolver el problema especial de cuello de botella del *k*-ésimo proveedor

Input: Una gráfica completa $G = (V, E)$, donde $V = V_{cons} \cup V_{prov}$ y $V_{cons} \cap V_{prov} = \phi$. Hay un peso $w(v_i)$ asociado con cada v_i en V_{prov}. Todas las aristas de G cumplen con la desigualdad del triángulo.

Output: Una solución aproximada para el problema especial de cuello de botella del k-ésimo proveedor. El valor de esta solución aproximada no es mayor que tres veces el valor de una solución óptima.

Paso 1. Las aristas en E se ordenan de modo que $|e_1| \leq |e_2| \leq \ldots \leq |e_m|$ donde $m = \binom{n}{2}$.

Paso 2. $i := 0$.

Paso 3. $i := i + 1$.

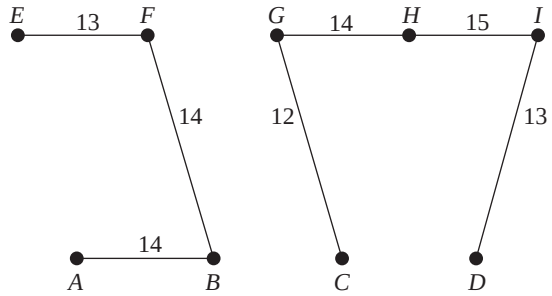
Paso 4. Si hay algún vértice (que sea un vértice consumidor) en V_{cons} que no sea adyacente a ningún vértice proveedor en $G(e_i)$ ir a paso 3.

Paso 5. Buscar un conjunto independiente maximal S en V_{cons} de $G(e_i)^2$. Para todo vértice v'_j en S buscar un vértice proveedor v_j que sea vecino de v'_j en $G(e_i)$ de peso mínimo. Sea $S' = \bigcup_j v_j$.

Paso 6. Si $\Sigma w(v_j) > k$, donde $v_j \in S'$, entonces ir a paso 3.
Paso 7. Output S' como el conjunto proveedor.
Paso 8. Para todo vértice v_j en v_{cons}, sea DIST(j) la distancia más corta entre v_j y S'. Hacer $V_{ap} = \max_j (\text{DIST}(j))$. Output V_{ap}.

A continuación ilustraremos nuestro algoritmo con un ejemplo. Considere la gráfica en la figura 9-15. En la figura 9-19 se muestra $G(HI)$ de la gráfica en la figura 9-15. Debido a que H no es adyacente a ningún proveedor en $G(HI)$, la respuesta a la prueba es "NO".

FIGURA 9-19 $G(H)$ de la gráfica en la figura 9-15.



En la figura 9-20 se muestra $G(HC)$ porque la siguiente arista más corta es HC. En la figura 9-21 se muestra $G(HC)^2$. Suponga que en V_{cons} de $G(HC)^2$ se selecciona $S = \{E, G\}$. Entonces $S' = \{B, C\}$. $W(B) + W(C) = 5 \leq k = 5$. Por lo tanto, $S' = \{B, C\}$, es nuestra solución. A partir de $S' = \{B, C\}$, se encontrará el vecino más cercano para cada vértice consumidor. El resultado se muestra en la figura 9-22. Como se muestra en la figura 9-22, $DIST(E) = 16$, $DIST(F) = 14$, $DIST(G) = 12$, $DIST(H) = 16$ y $DIST(I) = 21$. En consecuencia, $V_{ap} = 21$.

FIGURA 9-20 $G(HC)$ de la gráfica en la figura 9-15.

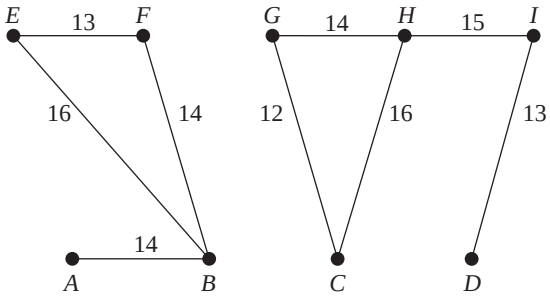


FIGURA 9-21 $G(HC)^2$.

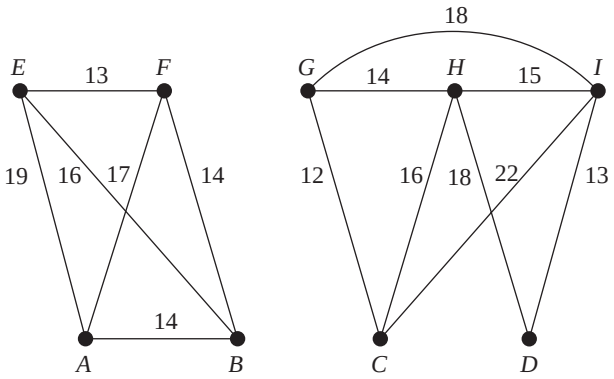
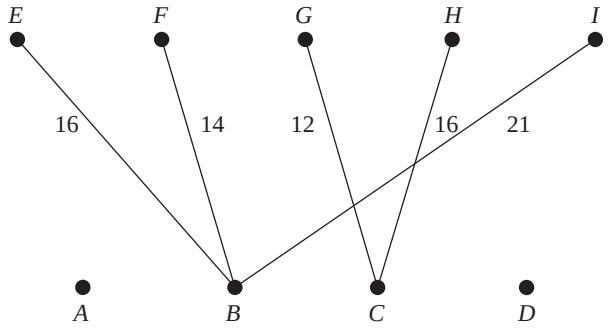


FIGURA 9-22 Solución obtenida a partir de $G(HC)^2$.



Como se hizo antes, es necesario responder las tres preguntas siguientes:

1. ¿Cuál es la complejidad temporal del algoritmo de aproximación?
2. ¿Cuál es la cota del algoritmo de aproximación?
3. ¿Esta cota es óptima?

Para contestar la primera pregunta, simplemente se afirma que el algoritmo de aproximación está dominado por los pasos necesarios para encontrar un conjunto independiente maximal. Éste es un algoritmo polinomial. Por lo tanto, el algoritmo de aproximación es un algoritmo polinomial.

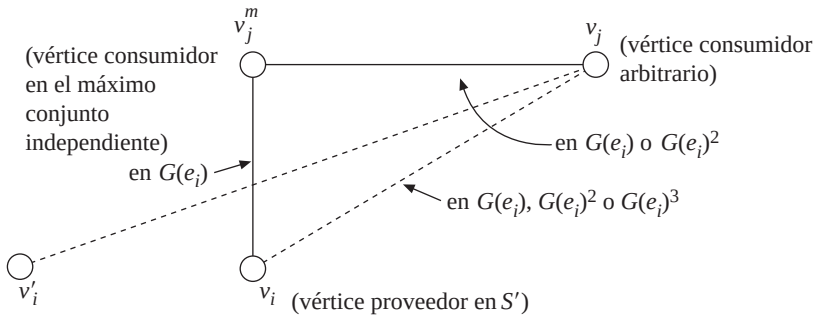
A continuación se responde la segunda pregunta. ¿Cuál es la cota del valor de la solución obtenida usando el algoritmo de aproximación? Sea e_i el e_i particular para el que el algoritmo de aproximación proporciona una solución. Sea V_{op} el valor de una solución óptima para el problema especial del cuello de botella del k-ésimo proveedor.

Resulta fácil ver que $|e_i| \leq V_{op}$. Además se observa que V_{ap}, el valor de nuestra solución aproximada, se obtiene en el paso 8 del algoritmo de aproximación. Para todo vértice consumidor v_j en S se encuentra el vecino más cercano en el conjunto original de vértices proveedores. Estos vecinos más cercanos constituyen S'. Debe observarse que la arista que une v_j con su vecino más cercano, que está en S', no necesariamente es una arista en $G(e_i)^2$. Por ejemplo, considere la figura 9-22. *BI* no es una arista en $G(HC)^2$. Se supondrá que como vértice proveedor para *I* se ha seleccionado *C* en vez de *B*. Debe observarse que *CI* puede no ser una arista de $G(HC)^2$. Sin embargo, $V_{ap} = |BI|$ debe ser menor o igual a $|CI|$. Es decir, $|CI|$ es una cota superior de $|BI|$. En consecuencia, es importante investigar a qué $G(HC)^d$ pertenece *CI*. A continuación se demostrará que *CI* pertenece a $G(HC)$, $G(HC)^2$ o $G(HC)^3$.

El razonamiento es como sigue: para todo v_j consumidor que no pertenece al conjunto independiente maximal seleccionado en nuestro algoritmo, sea v_j^m el vértice del conjunto independiente maximal que está unido directamente a v_j. Observe que v_j^m debe existir; en caso contrario, el conjunto independiente maximal no es un conjunto independiente maximal. v_j^m debe ser un vértice consumidor y estar unido directamente con un vértice proveedor v_i en S', determinado en el paso 5 del algoritmo de aproximación, lo cual se ilustra en la figura 9-23. Luego, la arista (v_j, v_j^m) debe estar ya sea en $G(e_i)$ o en $G(e_i)^2$. Además, (v_j^m, v_i) debe estar en $G(e_i)$, como se estipuló en el paso 5. Así, la arista (v_i, v_j) debe estar en $G(e_i)$, $G(e_i)^2$ o $G(e_i)^3$. En otras palabras, la arista (v_i, v_j) está en $G(e_i)^d$ si $d \geq 3$. Al final del algoritmo de aproximación, v_j puede estar unido con algún v_i' que no es v_i. (v_i', v_j) debe ser más corto que (v_i, v_j) en caso contrario, v_i' no se hubiese seleccionado. Por lo tanto,

$$V_{ap} = |(v_i', v_j)| \leq |(v_i, v_j)| \leq 3|e_i| \leq 3V_{op}.$$

FIGURA 9-23 Ilustración que explica el límite del algoritmo de aproximación para el problema especial de cuello de botella de *k*-ésimo proveedor.



La última pregunta está relacionada con la optimalidad de este factor de acotamiento 3. Se demostrará que es bastante improbable poder disminuir este factor de acotamiento. En caso de ser posible disminuirlo, entonces NP = P. Esencialmente, se demostrará que si existe un algoritmo de aproximación polinomial *D* capaz de producir una solución de aproximación del problema especial de cuello de botella del *k*-ésimo proveedor de cuyo valor esté garantizada que es menor que $a \cdot V_{op}$, donde $a < 3$ y V_{op} es el valor de la solución óptima, entonces P = NP.

Para probar esto, se usará el problema del hitting set (conjunto afectado), que es NP-completo. Es posible demostrar que si el algoritmo de aproximación *D* mencionado existe, entonces puede usarse para resolver el problema hitting set. Esto implica que P = NP. El problema hitting set se define como sigue: se tiene un conjunto finito *S* y una colección *C* de subconjuntos de *S* y un entero positivo *k*. La pregunta es: ¿existe un subconjunto $S' \subseteq S$ donde $|S'| \leq k$ y $S' \cap S_i \neq \phi$ para cualquier S_i en *C*?

Por ejemplo, sea

$$S = \{1, 2, 3, 4, 5\}$$
$$C = \{\{1, 2, 3\}, \{4, 5\}, \{1, 2, 4\}, \{3, 5\}, \{4, 5\}, \{3, 4\}\}$$
y $$k = 3.$$

Entonces $S' = \{1, 3, 4\}$ tiene la propiedad de que $|S'| = 3 \leq k$ y $S' \cap S_i \neq \phi$ para toda S_i en *C*.

Para fines ilustrativos, un problema hitting set se denominará *k*-hitting set para recalcar el parámetro *k*.

Dado un caso del problema *k*-hitting set, puede transformarse en un caso del problema especial de cuello de botella del *k*-ésimo proveedor. Suponga que $S = \{a_1, a_2, \ldots, a_n\}$. Luego, sea $V_{prov} = \{v_1, v_2, \ldots, v_n\}$, donde cada v_i corresponde a a_i. Suponga que $C = \{S_1, S_2, \ldots, S_m\}$. Luego, sea $V_{cust} = \{v_{n+1}, v_{n+2}, \ldots, v_{n+m}\}$, donde v_{n+1} corresponde a S_i. El peso de cada vértice proveedor se define como igual a 1. La distancia entre v_i y v_j se define como sigue:

d_{ij} = 2 si $\{v_i, v_j\} \subseteq V_{prov}$ o bien, $\{v_i, v_j\} \subseteq V_{cons}$
= 3 si $v_i \in V_{prov}$, $v_j \in V_{cons}$ y $a_i \subseteq S_j$
= 1 si $v_i \in V_{prov}$, $v_j \in V_{cons}$ y $a_i \in S_j$.

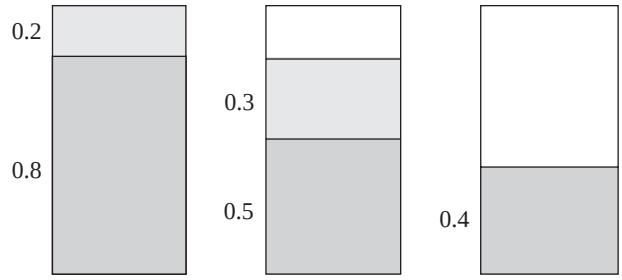
La demostración de que estas distancias cumplen la desigualdad del triángulo es directa, y *el problema del k-ésimo proveedor tiene una solución de costo 1 si y sólo si existe un k-hitting set S′*. Además, sino hay *k*-hitting set, el costo de un proveedor óptimo debe ser 3.

En consecuencia, si un algoritmo de aproximación garantiza una solución aproximada V_{ap}, donde $V_{ap} < 3V_{op}$, entonces $V_{ap} < 3$ si y sólo si $V_{op} = 1$ y hay un k-hitting set. Es decir, este algoritmo de aproximación polinomial puede usarse para resolver el problema del k-hitting set, que es NP-completo. Esto implica que NP = P.

9-5 UN ALGORITMO DE APROXIMACIÓN PARA EL PROBLEMA DE EMPAQUE EN CONTENEDORES

Dados n artículos en la lista $L = \{a_i \mid 1 \leq i \leq n \text{ y } 0 < a_i \leq 1\}$ *que deben colocarse en contenedores de capacidad unitaria, el problema de empaque en un contenedor consiste en determinar el número mínimo de contenedores necesarios para acomodar todos los n artículos*. Si los artículos de distintos tamaños se consideran como las longitudes del tiempo de ejecución de diferentes trabajos en un procesador estándar, entonces el problema se convierte en el problema de usar el número mínimo de procesadores que pueden terminar todos los trabajos en un tiempo fijo. Por ejemplo, sea $L = \{0.3, 0.5, 0.8, 0.2, 0.4\}$. Para empacar estos tres artículos se requieren por lo menos tres contenedores, lo cual se muestra en la figura 9-24.

FIGURA 9-24 Un ejemplo del problema de empaque en contenedores.



Debido a que el problema de partición puede transformarse fácilmente en este problema, el problema de empaque en contenedores es NP-difícil. A continuación se presentará un algoritmo de aproximación para el problema de empaque en contenedores.

El algoritmo de aproximación se denomina algoritmo de primer ajuste (first-fit). Los contenedores necesarios se identifican con los índices 1, 2,…, m, y sea B_1 el primer contenedor de capacidad unitaria. El algoritmo de primer ajuste coloca los artículos en los contenedores, uno por uno, en orden de índice creciente. Para empacar el artículo a_i, el algoritmo de primer ajuste siempre coloca el artículo en el contenedor de índice más bajo, para la cual la suma de los tamaños de los artículos que ya están en ese contenedor no exceda la unidad menos el tamaño de a_i.*

* O lo que es lo mismo, si todavía hay lugar en el contenedor para el artículo a_i. (*N. del R.T.*)

El algoritmo de primer ajuste es muy intuitivo y resulta fácil de implementar. El problema restante es cómo este algoritmo aproxima una solución óptima del problema de empaque en contenedores.

El tamaño de un artículo a_i se denota por $S(a_i)$. Debido a que la capacidad de cada contenedor es 1, el tamaño de una solución óptima de un caso I del problema, denotada por $OPT(I)$, es mayor que el tope o igual al tope de la sumatoria de los tamaños de todos los artículos. Es decir,

$$OPT(I) \geq \left\lceil \sum_{i=1}^{n} S(a_i) \right\rceil.$$

Sea $FF(I)$ el número de contenedores necesarios en el algoritmo de primer ajuste. $FF(I) = m$. Una cota superior de $FF(I)$ puede obtenerse como sigue: se toma cualquier contenedor B_i y se hace que $C(B_i)$ sea la suma de los tamaños de los a_j empacados en el contenedor B_i. Así, se tiene

$$C(B_i) + C(B_{i+1}) > 1;$$

en caso contrario, los artículos en B_{i+1} deben colocarse en B_i. Al sumar todos los m contenedores no vacíos, se tiene

$$C(B_1) + C(B_2) + \cdots + C(B_m) > m/2.$$

Esto significa que

$$FF(I) = m < 2\left\lceil \sum_{i=1}^{m} C(B_i) \right\rceil = 2\left\lceil \sum_{i=1}^{n} S(a_i) \right\rceil.$$

Por lo tanto, puede concluirse que

$$FF(I) < 2OPT(I).$$

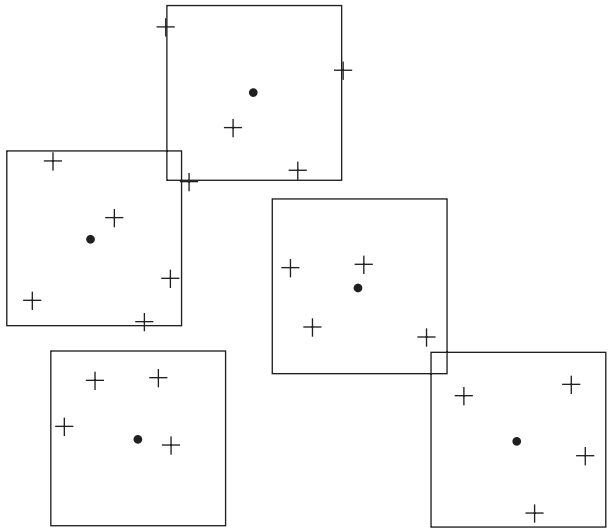
9-6 UN ALGORITMO DE APROXIMACIÓN POLINOMIAL PARA EL PROBLEMA RECTILÍNEO DE m-CENTROS

Dado un conjunto de n puntos en el plano, el problema rectilíneo de m-centros consiste en encontrar m cuadrados rectilíneos que cubran todos esos n puntos de modo que la longitud del lado máximo de estos cuadrados se minimice. Por cuadrado rectilíneo se entiende un cuadrado con lados perpendiculares o paralelos al eje x del plano euclidiano. Debido a que los cuadrados rectilíneos menores pueden agrandarse en una solución

para el problema rectilíneo del *m*-centros sin afectar nuestro objetivo, se supone que todos los *m* cuadrados rectilíneos de una solución tienen la misma longitud. A la longitud del lado se le denomina el tamaño de la solución.

En la figura 9-25 se proporciona un caso del problema rectilíneo de 5-centros. Todos los puntos indicados con "+" deben cubrirse con los cinco cuadrados rectilíneos. Los cinco cuadrados rectilíneos, que se muestran en esta figura, constituyen una solución óptima.

FIGURA 9-25 Ejemplo del problema rectilíneo de 5-centros.



Se ha demostrado que el problema rectilíneo de *m*-centros es NP-difícil. Además, se ha demostrado que cualquier algoritmo de aproximación polinomial que resuelva el problema tiene una tasa de error ≥ 2, a menos que NP = P. En esta sección se presentará una solución aproximada con una tasa de error exactamente igual a 2.

Observe que para cualquier solución óptima del problema rectilíneo de *m*-centros en $P = \{p_1, p_2, \ldots, p_n\}$, el tamaño de la solución debe ser igual a una de las distancias rectilíneas de entre los puntos de entrada. Es decir, el tamaño de una solución óptima debe ser igual a una de los $L_\infty(p_i, p_j)$, $1 \leq i \leq j \leq n$, donde $L_\infty((x_1, y_1), (x_2, y_2)) = \text{máx}\{|x_1 - x_2|, |y_1 - y_2|\}$. Se menciona que un número *r* es factible si existen *m* cuadrados con longitud del lado *r* que cubren a todos los *n* puntos. Un conjunto de *m* cuadrados que cubren a todos los puntos de entrada se denomina solución factible. Un método directo para resolver el problema rectilíneo de *m*-centros es como sigue:

1. Calcular todas las $L_\infty(p_i, p_j)$, $1 \leq i < j \leq n$ posibles.
2. Ordenar las distancias encontradas anteriores. Denotarlas por $D[1] \leq D[2] \leq \cdots \leq D[n(n-1)/2]$.
3. Ejecutar una búsqueda binaria sobre $D[i]$, $i = 1, 2, \ldots, n(n-1)/2$ para encontrar el menor índice i_0 tal que $D[i_0]$ sea factible, así como una solución factible de tamaño $D[i_0]$. Para cada distancia $D[i]$ investigada, es necesario probar si $D[i]$ es factible.

Sólo hay $O(\log n)$ distancias a probar. No obstante, para toda $D[i]$, la prueba de si existen m cuadrados con longitud de lado $D[i]$ que cubran todos los n puntos sigue siendo un problema NP-completo. En consecuencia, como se esperaba, el algoritmo directo anterior está acotado a ser exponencial, al menos hasta el presente.

En esta sección se mostrará una subrutina de prueba "relajada" que se usará en nuestro algoritmo de aproximación. Dados un conjunto de n puntos en el plano, y dos números m y r como parámetros de entrada, esta subrutina relajada genera m cuadrados de longitud de lado $2r$ en vez de r. Luego prueba si los m cuadrados cubren estos n puntos. Si fracasa, devuelve "fracaso". Si tiene éxito, proporciona una solución factible de tamaño $2r$. Después se verá que "fracaso" garantiza que no hay m cuadrados de longitud de lado r que puedan cubrir estos n puntos.

Más tarde se proporcionarán los detalles de esta subrutina. Esta subrutina relajada se identifica por Test(m, P, r). Con esta subrutina de prueba relajada, nuestro algoritmo de aproximación es uno que sustituye la subrutina de prueba en el algoritmo óptimo directo de antes por la subrutina relajada.

Algoritmo 9-5 □ Un algoritmo de aproximación polinomial para el problema rectilíneo de *m*-centros

Input: Un conjunto P de n puntos, número de centros: m.

Output: $SQ[1], \ldots, SQ[m]$. Una solución factible del problema rectilíneo de m-centros de tamaño menor o igual al doble del tamaño de una solución óptima.

Paso 1. Calcular las distancias rectilíneas de todos los pares de dos puntos y se ordenan junto con 0 en una secuencia creciente $D[0] = 0, D[1], \ldots, D[n(n-1)/2]$.

Paso 2. $LEFT := 1$, $RIGHT := n(n-1)/2$.

Paso 3. $i := \lceil (RIGHT + LEFT)/2 \rceil$.

Paso 4. Si Test(m, P, $D[i]$) no es "fracaso", entonces
$RIGHT := i$
en otro caso
$LEFT := i$.
Paso 5. Si $RIGHT = LEFT + 1$ entonces
regresar Test(m, P, $D[RIGHT]$)
en otro caso
ir a paso 3.

A continuación, la subrutina de prueba relajada Test(m, P, r) se describe como sigue:

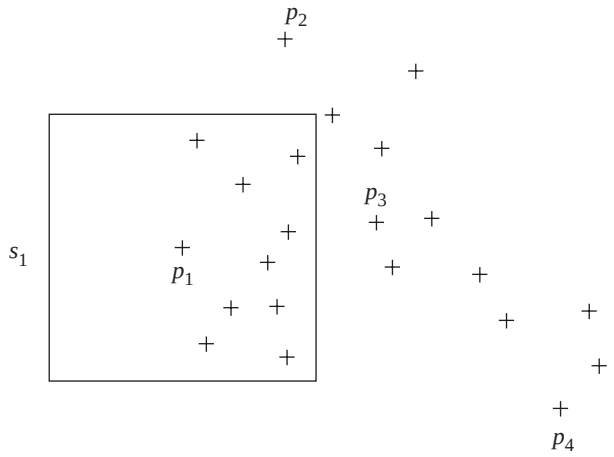
1. Encontrar el punto con el valor x mínimo en los puntos restantes que aún no se han cubierto y trazar un cuadrado con centro en este punto y cuya longitud de lado sea $2r$.
2. Agregar el cuadrado a la solución y eliminar todos los puntos ya cubiertos por el cuadrado.
3. Repetir m veces los pasos 1 y 2. Si se tiene éxito en encontrar m cuadrados para cubrir todos los puntos de entrada de esta forma, estos cuadrados se regresan como soluciones factibles; en caso contrario, se regresa "fracaso".

Algoritmo 9-6 □ Algoritmo Test(m, P, r)

Input: Conjunto de puntos P, número de centros: m, tamaño: r.
Output: "fracaso", o bien, $SQ[1], \ldots, SQ[m]$ m cuadrados de tamaño $2r$ que cubren a P.
Paso 1. $PS := P$
Paso 2. Para $i := 1$ to m do
If $PS \neq \phi$ entonces
p : el punto en PS con el menor valor x
$SQ[i] :=$ el cuadrado de tamaño $2r$ con centro en p
$PS := PS -$ {puntos cubiertos por $SQ[i]$}
en otro caso
$SQ[i] := SQ[i - 1]$.
Paso 3. Si $PS = \phi$ entonces
regresar $SQ[1], \ldots, SQ[m]$
en otro caso
regresar "fracaso".

Considere nuevamente el caso de un problema rectilíneo de 5 centros de la figura 9-25. Sea r la longitud del lado de los cuadrados de la figura 9-25. Al aplicar la subrutina de prueba relajada al caso con r, primero se encuentra que p_1 es el punto con el menor valor x, y luego se traza un cuadrado con centro en p_1 y longitud de lado $2r$, como se muestra en la figura 9-26. Luego se eliminan los puntos cubiertos por el cuadrado S_1. De los puntos restantes, se encuentra que el punto p_2 es el punto con menor valor x. De nuevo, se traza un cuadrado S_2 con centro en p_2 y longitud de lado $2r$, como se muestra en la figura 9-27 y se eliminan los puntos cubiertos por S_2. El proceso anterior se repite cinco veces. Finalmente, se llega a una solución factible mostrada en la figura 9-28. En este caso bastan cuatro cuadrados.

FIGURA 9-26 La primera aplicación de la subrutina de prueba relajada.



A continuación se demostrará que nuestro algoritmo de aproximación tiene una tasa de error igual a 2. La afirmación clave es: *si r es factible, entonces nuestra subrutina de prueba relajada* Test(m, P, r) *siempre regresa una solución factible de tamaño* $2r$. Suponga que la afirmación anterior es verdadera. Sea r^* el tamaño de una solución óptima. Debido a que r^* es factible, por la afirmación hecha, la subrutina Test(m, P, r^*) regresa una solución factible. Debido a que nuestro algoritmo de aproximación termina en el menor r tal que Test(m, P, r) regresa una solución factible de tamaño $2r$, se tiene $r \leq r^*$. En consecuencia, la solución factible encontrada es de tamaño $2r$, que es menor o igual a $2r^*$. En otras palabras, la tasa de error es 2.

A continuación se demostrará la afirmación hecha. Sean $S_1, S_2, \ldots, S_m$ una solución factible de tamaño r y $S_1, \ldots, S'_m$ los m cuadrados con longitud de lado generados en la subrutina Test(m, P, r).

FIGURA 9-27 La segunda aplicación de la subrutina de prueba relajada.

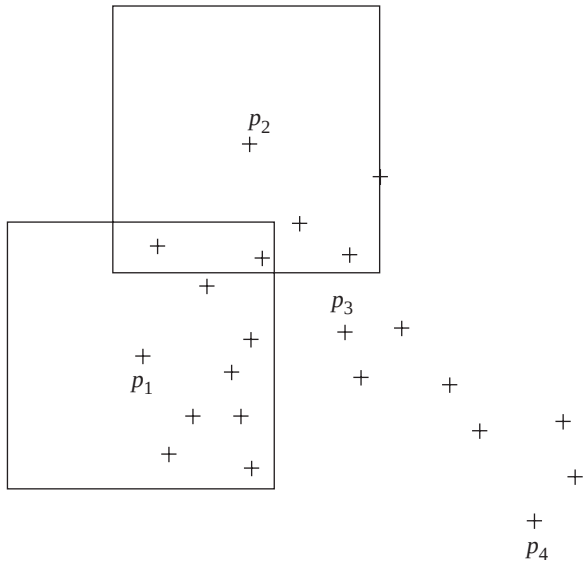
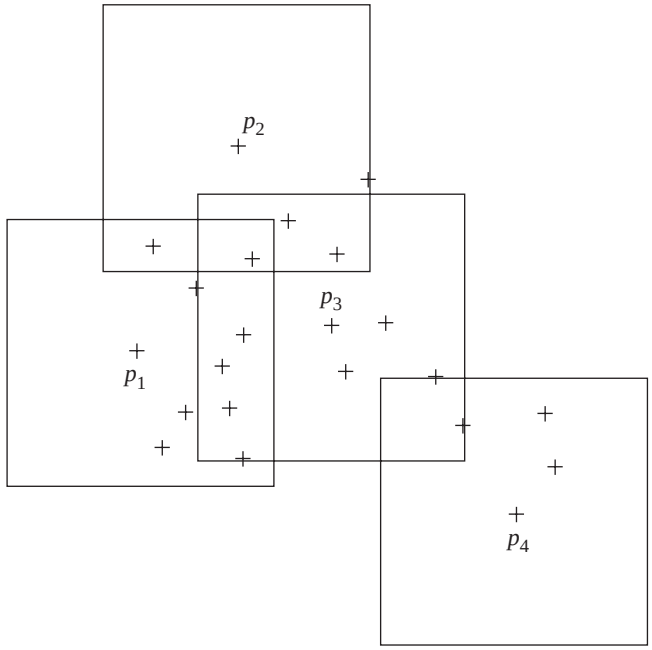


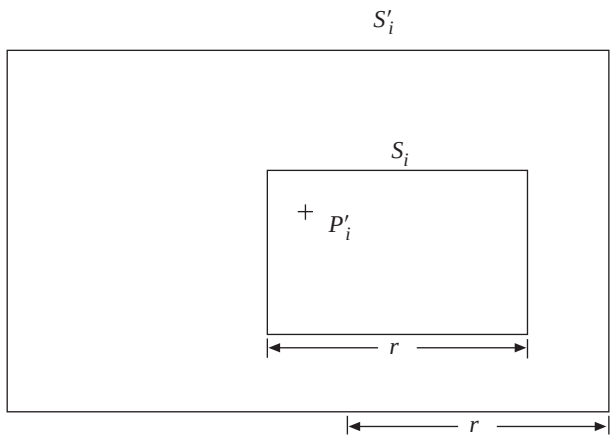
FIGURA 9-28 Una solución factible del problema rectilíneo de 4 centros.



En caso de que $S'_1, S'_2, \ldots, S'_m$ no sean distintos, $PS = \phi$ en el paso 2 de la subrutina Test. En otras palabras, en este caso, $S'_1, S'_2, \ldots, S'_m$ siempre cubre todos los puntos y constituye una solución factible. Luego, considere el caso en que $S'_1, S'_2, \ldots, S'_m$ son distintos. Sea S'_i el centro de p'_i para $i = 1, 2, \ldots, m$. Es decir, cualquier punto en S'_i es de distancia rectilínea menor o igual r desde p'_i. Por cierto, se escoge S'_i, p'_j no está cubierto por $S'_1, S'_2, \ldots, S'_{j-1}$. Así, $L_\infty(p'_i, p'_j) > r$ para todo $i \neq j$. Debido a que la longitud del lado en S_i es r para $i = 1, \ldots, m$, la distancia rectilínea entre dos puntos cualesquiera en S_i es menor o igual a r. Así, cualquier S_j contiene cuando mucho un p'_i, $i = 1, \ldots, m$. Por otra parte, debido a que $S_1, S_2, \ldots, S_m$ constituye una solución factible, la unión de $S_1, S_2, \ldots, S_m$ contiene $p'_1, \ldots, p'_m$. En consecuencia, p'_i pertenece a S_j distintos; es decir, $p'_i \in S_i$. Para $1 \leq i \leq m$, ya que el tamaño de S_i es r, $L_\infty(p, p'_i) \leq r$ para todo $p \in S_i$. Así, $S_i \subset S'_i$ (figura 9-29). En consecuencia, $S'_1, \ldots, S'_m$ contiene todos los puntos. Se concluye la afirmación hecha. Debido a la afirmación de marras, también puede tenerse la siguiente afirmación equivalente:

Si Test(m, P, r) no regresa una solución factible, entonces r no es factible.

FIGURA 9-29 La explicación de $S_i \subset S'_i$.



En el algoritmo 9-5, los pasos 1 y 2 requieren tiempo $O(n^2 \log n)$, y el paso 3 se ejecuta $O(\log n)$ veces. El ciclo de prueba del paso 3 al paso 5 requiere tiempo $O(mn)$. En consecuencia, se tiene un algoritmo de aproximación $O(n^2 \log n)$ con tasa de error 2 para el problema rectilíneo de m centros.

9-7 UN ALGORITMO DE APROXIMACIÓN PARA EL PROBLEMA DE ALINEACIÓN DE MÚLTIPLES SECUENCIAS

En el capítulo 7 sólo se estudió el problema de alineación de dos secuencias. Resulta natural extender este problema al problema de alineación de múltiples secuencias donde están implicadas más de dos secuencias. Considere el siguiente caso, en que están implicadas tres secuencias.

S_1 = ATTCGAT
S_2 = TTGAG
S_3 = ATGCT

A continuación se muestra una alineación bastante buena de estas tres secuencias.

S_1 = ATTCGAT
S_2 = –TT–GAG
S_3 = AT––GCT

Se observa que la alineación entre cada par de secuencias es muy buena.

El problema de alineación de múltiples secuencias es semejante al problema de alineación de dos secuencias. En vez de definir una función de puntaje fundamental $\sigma(x, y)$ es necesario definir una función de puntaje que implique tres variables. Se supondrá que hay, por ejemplo, tres secuencias. En vez de considerar a_i y b_j, ahora es necesario considerar el apareamiento de a_i, b_j y c_k. Es decir, debe definirse una $\sigma(x, y, z)$ y debe encontrarse $A(i, j, k)$. El problema es: una vez que se determina $A(i, j, k)$, debe considerarse lo siguiente.

$A(i - 1, j, k)$, $A(i, j - 1, k)$, $A(i, j, k - 1)$, $A(i - 1, j - 1, k)$
$A(i, j - 1, k - 1)$, $A(i - 1, j, k - 1)$, $A(i - 1, j - 1, k - 1)$

Considere que entre dos secuencias se ha definido una función de puntaje lineal. Dadas k secuencias de entrada, el problema de la suma de las alineaciones por pares de múltiples secuencias consiste en determinar una alineación de estas k secuencias que maximice la suma de puntajes de todas las alineaciones por pares entre ellas. Si k, el número de secuencias de entrada, es una variable, entonces se ha demostrado que este problema es NP-difícil. Así, no hay mucha esperanza en utilizar un algoritmo polinomial para resolver este problema de la suma de las alineaciones por pares de múltiples secuencias y en vez de tal algoritmo se requieren de algoritmos de aproximación. En

este apartado se presentará un algoritmo de aproximación, propuesto por Gusfield, para resolver el problema de la suma de las alineaciones por pares de múltiples secuencias.

Considere las dos secuencias siguientes:

$S_1 = \text{GCCAT}$
$S_2 = \text{GAT}.$

Una alineación posible entre estas dos secuencias es

$S_1' = \text{GCCAT}$
$S_2' = \text{G}--\text{AT}.$

Para esta alineación, hay tres apareamientos exactos y dos faltas de correlación. Se define $\sigma(x, y) = 0$ si $x = y$ y $\sigma(x, y) = 1$ si $x \neq y$. Para una alineación $S_1' = a_1', a_2', \ldots, a_n'$ y $S_2' \;\; b_1', b_2', \ldots, b_n'$, la distancia entre las dos secuencias inducida por la alineación se define como

$$\sum_{i=1}^{n} \sigma(a_i', b_i').$$

El lector puede ver fácilmente que esta función distancia, denotada por $d(S_i, S_j)$, posee las siguientes características:

1. $d(S_i, S_i) = 0$
2. $d(S_i, S_j) + d(S_i, S_k) \geq d(S_j, S_k)$

La segunda propiedad se denomina desigualdad del triángulo.

Otra cuestión que es necesario recalcar es que para una alineación de dos secuencias, el par $(-, -)$ nunca ocurre. Sin embargo, en una alineación de múltiples secuencias, es posible que haya un "$-$" apareado con un "$-$". Considere S_1, S_2 y S_3 como sigue:

$S_1 = \text{ACTC}$
$S_2 = \text{AC}$
$S_3 = \text{ATCG}.$

Primero se alinean S_1 y S_2 como sigue:

$S_1 = \text{ACTC}$
$S_2 \;\; \text{A}--\text{C}.$

Luego, suponga que S_3 y S_1 se alinean como sigue:

S_1 = ACTC–
S_3 = A–TCG.

Finalmente, las tres secuencias se alinean como se muestra:

S_1 = ACTC–
S_2 = A– –C–
S_3 = A–TCG.

Esta vez, "−" está apareado con "−" dos veces. Observe que cuando S_3 se alinea con S_1, "−" se agrega al S_1 ya alineado.

Dadas dos secuencias S_i y S_j, la distancia alineada mínima se denota por $D(S_i, S_j)$. Ahora se considerarán las cuatro secuencias siguientes. Primero se encuentra la secuencia con las distancias más cortas a todas las demás secuencias

S_1 = ATGCTC
S_2 = AGAGC
S_3 = TTCTG
S_4 = ATTGCATGC.

Las cuatro secuencias se alinean por pares.

S_1 = ATGCTC
$D(S_1, S_2) = 3$
S_2 = A–GAGC

S_1 = ATGCTC
$D(S_1, S_3) = 3$
S_3 = TT–CTG

S_1 = AT–GC–T–C
$D(S_1, S_4) = 3$
S_4 = ATTGCATGC

S_2 = AGAGC
$D(S_2, S_3) = 5$

S_3 = TTCTG

S_2 = A– –G–A–GC

$D(S_2, S_4) = 4$

S_4 = ATTGCATGC

S_3 = –TT–C–TG–

$D(S_3, S_4) = 4$

S_4 = ATTGCATGC.

Así, se tiene

$$D(S_1, S_2) + D(S_1, S_3) + D(S_1, S_4) = 3 + 3 + 3 = 9$$
$$D(S_2, S_1) + D(S_2, S_3) + D(S_2, S_4) = 3 + 5 + 4 = 12$$
$$D(S_3, S_1) + D(S_3, S_2) + D(S_3, S_4) = 3 + 5 + 4 = 12$$
$$D(S_4, S_1) + D(S_4, S_2) + D(S_4, S_3) = 3 + 4 + 4 = 11.$$

Puede verse que S_1 tiene las distancias más cortas a todas las demás secuencias. Puede afirmarse que S_1 es más semejante a otras. Esta secuencia se denomina centro de las secuencias. A continuación se definirá formalmente este concepto.

Dado un conjunto S de k secuencias, el *centro* de este conjunto de secuencias es la secuencia que minimiza

$$\sum_{X \in S \setminus \{S_i\}} D(S_i, X).$$

Hay $k(k - 1)/2$ pares de secuencias. Cada par puede alinearse usando el método de la programación dinámica. Puede verse que para encontrar un centro se requiere tiempo polinomial. Nuestro algoritmo de aproximación funciona como se describe en el algoritmo 9-7.

Algoritmo 9-7 □ Un algoritmo de 2-aproximación para encontrar una solución aproximada para el problema de la suma de las alineaciones por pares de múltiples secuencias

Input: k secuencias.
Output: Una alineación de las k secuencias con razón de desempeño no mayor que 2.

Paso 1: Buscar el centro de estas k secuencias. Sin pérdida de generalidad, puede suponerse que el centro es S_1.
Paso 2: Hacer $i = 2$.
Paso 3: Mientras $i \leq k$
    Buscar una alineación óptima entre S_i y S_1.
    Sumar espacios a las secuencias ya alineadas S_1, S_2,…, S_{i-1} en caso de ser necesario.
    $i = i + 1$
    End while (fin de ciclo)
Paso 4: Output la alineación final.

A continuación se analizarán las cuatro secuencias que acaban de estudiarse en los párrafos anteriores:

S_1 = ATGCTC
S_2 = AGAGC
S_3 = TTCTG
S_4 ATTGCATGC.

Como ya se demostró, S_1 es el centro. Luego, S_2 se alinea con S_1 como sigue:

S_1 = ATGCTC
S_2 = A−GAGC.

S_3 se agrega al alinear S_3 con S_1.

S_1 = ATGCTC
S_3 = −TTCTG.

Así, la alineación se convierte en:

S_1 = ATGCTC
S_2 = A–GAGC
S_3 = −TTCTG.

S_4 se agrega al alinear S_4 con S_1.

S_1 = AT–GC–T–C
S_4 = ATTGCATGC.

En esta ocasión, a S_1 alineado se han agregado espacios. Entonces, es necesario agregar espacios a S_2 y S_3 alineados. La alineación final es:

S_1 = AT–GC–T–C
S_2 = A– –GA–G–C
S_3 = –T–TC–T–G
S_4 = ATTGCATGC.

Como puede verse, éste es un algoritmo de aproximación típico, ya que todas las secuencias sólo se alinean con respecto a S_1. Sea $d(S_i, S_j)$ la distancia entre S_i y S_j inducida por este algoritmo de aproximación. Sea $App = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d(S_i, S_j)$. Sea $d^*(S_i, S_j)$ la distancia entre S_i y S_j inducida por un algoritmo de alineación múltiple óptimo. Sea $Opt = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d^*(S_i, S_j)$. A continuación se demostrará que $App \leq 2Opt$.

Antes de proporcionar la demostración formal, se observa que $d(S_1, S_i) = D(S_1, S_i)$. Esto puede verse fácilmente al analizar el ejemplo anterior.

S_1 = ATGCTC
S_2 = A–GAGC.

Así, $D(S_1, S_2) = 3$. Al final del algoritmo, S_1 y S_2 están alineados como sigue:

S_1 = AT–GC–T–C
S_2 = A– –GA–G–C.

Entonces $d(S_1, S_2) = 3 = D(S_1, S_2)$. Esta distancia no cambia porque $\sigma(-, -) = 0$.

La demostración de que $App \leq 2Opt$ es como sigue:

$$App = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d(S_i, S_j)$$

$$\leq \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d(S_i, S_1) + d(S_1, S_j) \qquad \text{(desigualdad del triángulo)}$$

$$= 2(k-1) \sum_{i=2}^{k} d(S_1, S_i) \qquad (d(S_1, S_i) = d(S_i, S_1)).$$

Debido a que $d(S_1, S_i) = D(S_1, S_i)$ para toda i, se tiene

$$App \leq 2(k-1) \sum_{i=2}^{k} D(S_1, S_i). \qquad \textbf{(9-1)}$$

A continuación se encontrará $Opt = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d^*(S_i, S_j)$. Primero, se observa que $D(S_i, S_j)$ es la distancia inducida por una alineación óptima de 2 secuencias. Así,

$$D(S_i, S_j) \leq d^*(S_i, S_j),$$

y

$$Opt = \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} d^*(S_i, S_j)$$

$$\geq \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq i}}^{k} D(S_i, S_j).$$

Sin embargo, observe que el centro es S_1. Así,

$$Opt \geq \sum_{i=1}^{k} \sum_{\substack{j=1 \\ j \neq 1}}^{k} D(S_i, S_j)$$

$$\geq \sum_{i=1}^{k} \sum_{j=2}^{k} D(S_1, S_j)$$

$$= k \sum_{j=2}^{k} D(S_1, S_j). \qquad \textbf{(9-2)}$$

Al considerar las ecuaciones (9-1) y (9-2) se tiene $App \leq 2Opt$.

9-8 ALGORITMO DE 2-APROXIMACIÓN PARA EL ORDENAMIENTO POR EL PROBLEMA DE TRANSPOSICIÓN

En este apartado se presentará el ordenamiento por algoritmo de transposición para comparar dos genomas. Un genoma puede verse simplemente como una secuencia de genes cuyo orden en la secuencia es sumamente importante. Así, en esta sección, cada gen se identifica con un entero y un genoma por una secuencia de enteros.

La comparación de dos genomas es importante porque proporciona información sobre cuán distintas son genéticamente estas especies. Si dos genomas son semejantes entre sí, están próximos genéticamente; en caso contrario, no. La cuestión consiste en cómo medir la semejanza entre dos genomas. Esencialmente, ésta se determina al medir la facilidad con que un genoma se transforma en otro mediante algunas operaciones. En este apartado se presentará este tipo de operación; a saber, la transposición.

Debido a que una secuencia de números se está transformando en otra secuencia, sin pérdida de generalidad siempre es posible suponer que la secuencia objetivo es 1, 2,…, *n*. Una transposición intercambia dos subcadenas adyacentes de cualquier longitud sin modificar el orden de las dos subcadenas. A continuación se presenta un ejemplo que describe tal operación:

Genoma *X*: 3 1 5 2 4 → Genoma *Y*: 3 2 4 1 5.

La semejanza entre dos secuencias puede medirse a través del número mínimo de operaciones necesarias para transformar una secuencia en otra. Debido a que la secuencia objetivo siempre es 1, 2,…, *n*, el problema puede considerarse como uno de ordenamiento. Sin embargo, se trata de un problema de ordenamiento con el que no estamos familiarizados. El problema de ordenamiento consiste en encontrar el número mínimo de una operación especificada necesaria para ordenar una secuencia. Así, el problema de ordenamiento es un problema de optimización.

Considere una secuencia 1 4 5 3 2. Esta secuencia puede ordenarse por transposición como sigue:

1 4 5 3 2
1 3 2 4 5
1 2 3 4 5.

Ahora se empieza a presentar el ordenamiento por transposición. Antes que todo, aunque se está hablando de ordenamiento, nuestra entrada es distinta de las entradas normales para el ordenamiento. Debe cumplir las siguientes condiciones.

1. La secuencia de entrada no puede contener dos números idénticos. Por ejemplo, no puede contener dos 5.
2. En la secuencia de entrada no puede haber ningún número negativo.
3. Si i y j aparecen en la secuencia e $i < k < j$, entonces k debe aparecer en la secuencia. Es decir, no se permite el caso en que aparezcan 5 y 7, pero no aparezca 6.

En resumen, la entrada puede definirse simplemente como una permutación de 1, 2,…, n. Es decir, el genoma de entrada se representa por una permutación $\pi = \pi_1 \pi_2 \cdots \pi_n$. Por razones que serán evidentes después, la permutación se extiende a fin de incluir $\pi_0 = 0$ y $\pi_{n+1} = n + 1$. Por ejemplo, una permutación de entrada típica es 0 2 4 1 3 5.

Para una permutación π, una transposición, denotada por $\rho(i, j, k)$ (definida por todas las $1 \leq i < j \leq n + 1$ y todas las $1 \leq k \leq n + 1$ tales que $k \notin [i, j]$), intercambia las subcadenas $\pi_i, \pi_{i+1}, \ldots, \pi_{j-1}$, y $\pi_j, \pi_{j+1}, \ldots, \pi_{k-1}$ en la permutación. Por ejemplo, $\rho(2, 4, 6)$ en la permutación 0 7 2 3 6 1 5 4 8 intercambia las subcadenas (2 3) y (6 1) y resulta en 0 7 6 1 2 3 5 4 8. Por ejemplo, una permutación típica de entrada es 0 2 4 1 3 5. Dadas una permutación π y una transposición ρ, la aplicación de ρ en π se denota por $\rho \cdot \pi$.

A continuación se define formalmente el problema de ordenar por transposición: dadas dos permutaciones π y σ, el problema de ordenar por transposición consiste en encontrar una serie de transposiciones $\rho_1, \rho_2, \ldots, \rho_t$ tales que $\rho_t \ldots \rho_2 \cdot \rho_1 \cdot \pi = \sigma$ y t sea mínima. Esta t se denomina distancia de transposición entre π y σ. Como ya se indicó, sin pérdida de generalidad puede suponerse que σ siempre en una permutación identidad en la forma de $(0, 1, 2, \ldots, n, n + 1)$. En lo que sigue, se presentará el algoritmo de 2-aproximación para ordenar por transposición, propuesto por Bafna y Pevzner en 1998. Observe que se desconoce la complejidad del problema de ordenamiento por transposición.

Para toda $0 \leq i \leq n$ en una permutación, hay un punto de *corte* entre π_i y π_{i+1} si $\pi_{i+1} \neq \pi_i + 1$. Por ejemplo, para una permutación 0 2 3 1 4 5, la permutación con puntos de corte agregados es 0, 2 3, 1, 4 5. Una permutación ordenada no contiene puntos de corte. Una permutación sin puntos de corte se denomina permutación identidad. En consecuencia, puede afirmarse que la tarea consiste en ordenar la permutación de entrada en una permutación identidad.

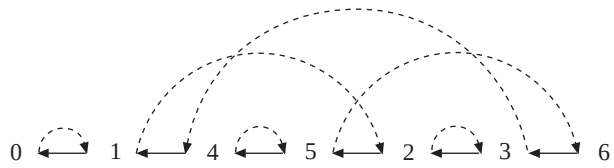
Debido a que la permutación identidad es una permutación sin puntos de corte, el ordenamiento de una permutación corresponde a disminuir el número de puntos de corte. Sea $d(\pi)$ el número mínimo de transposiciones necesarias para transformar π en una permutación identidad.

Para toda transposición, cuando mucho pueden disminuirse tres puntos de corte, y en consecuencia, una cota inferior trivial de $d(\pi)$ es

(el número de puntos de corte en π)/3.

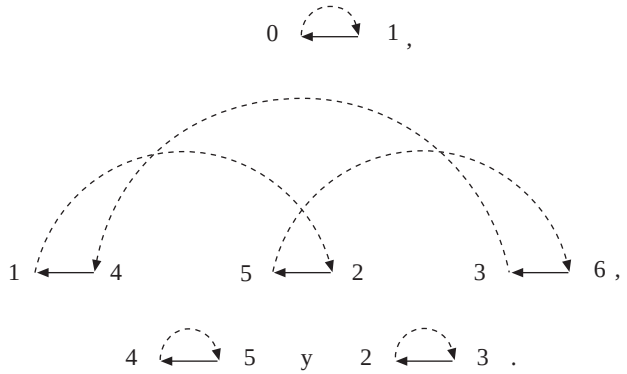
Una gráfica de ciclo de π, denotada por $G(\pi)$, es la gráfica dirigida con aristas coloreadas con conjunto de vértices $\{0, 1, 2, \ldots, n, n + 1\}$ y conjunto de aristas definido como sigue. Para toda $1 \leq i \leq n + 1$, las aristas grises están dirigidos de $i - 1$ a i y las aristas negras están dirigidas de π_i a π_{i-1}. Un ejemplo de gráfica de ciclos se muestra en la figura 9-30, donde los arcos punteados y continuos representan aristas grises y aristas negras, respectivamente.

FIGURA 9-30 Gráfica de ciclos de una permutación 0 1 4 5 2 3 6.



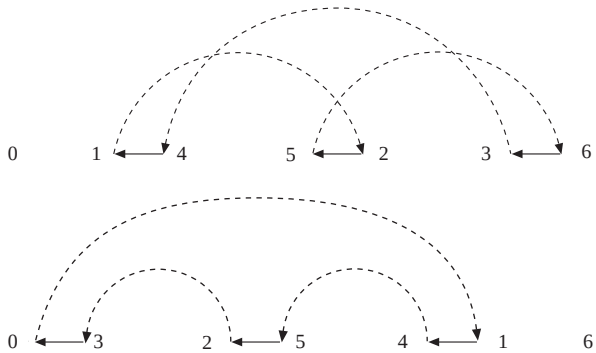
Un ciclo alterno de una gráfica coloreada es un ciclo donde cada par de aristas adyacentes es de colores distintos. Para cada vértice en $G(\pi)$, toda arista incidente está apareada de forma única con una arista de salida de color diferente. Por lo tanto, el conjunto de aristas de $G(\pi)$ puede descomponerse en ciclos alternos. En la figura 9-31 se muestra un ejemplo para la descomposición en ciclos alternos de la gráfica de ciclos alternos que se observa en la figura 9-30.

FIGURA 9-31 Descomposición de una gráfica de ciclos en ciclos alternos.

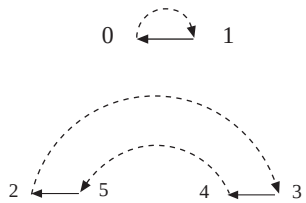


Para hacer referencia a un ciclo alterno que contiene k aristas negras se usan k-ciclos. Se dice que un k-ciclo es largo si $k > 2$, y corto en caso contrario. En la figura 9-32 se muestran dos ejemplos sobre ciclos alternos largos y cortos. En la figura 9-33 se muestra la gráfica de ciclos de una permutación identidad.

FIGURA 9-32 Ciclos alternos largos y cortos.

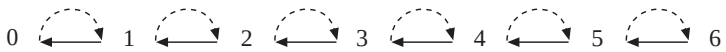


a) Ciclos alternos largos



b) Ciclos alternos cortos

FIGURA 9-33 Gráfica de ciclos de una gráfica de identidad.



Como se sugiere en la figura 9-33, en una permutación identidad, cada vértice x apunta a otro vértice y mediante una arista gris, y un vértice y apunta de vuelta a otro vértice x mediante una arista negra. Este tipo de gráficas de ciclos se denominan regulares. El ordenamiento por el problema de transposición consiste en transformar una gráfica de ciclos que no es regular en una gráfica de ciclos regular.

Debido a que siempre se trabaja con ciclos alternos, un ciclo alterno simplemente se denomina ciclo. En $G(\pi)$ hay cuando mucho $n + 1$ ciclos, y la única permutación con $n + 1$ ciclos es la permutación identidad.

El número de ciclos en $G(\pi)$ se denota como $c(\pi)$ para una permutación π. En consecuencia, el objetivo de ordenar π es incrementar el número de ciclos de $c(\pi)$ a $n + 1$. También, el cambio en el número de ciclos debido a la transposición ρ se denota como $\Delta c(\rho) = c(\rho\pi) - c(\pi)$ para una permutación π.

Se considerará la siguiente permutación:

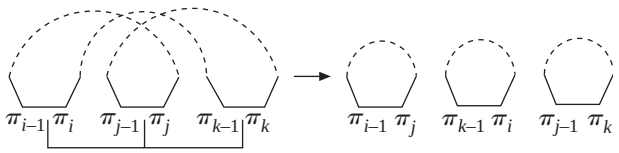
0 3 4 1 2 5.

Suponga que se realiza una transposición sobre las subcadenas (3 4) y (1 2), como ya se indicó, para obtener lo siguiente:

0 1 2 3 4 5

El lector puede ver fácilmente que la transposición anterior es ideal. La permutación original posee tres puntos de corte. Después de la transposición ya no hay tres puntos de corte. Esto puede explicarse usando el diagrama de ciclos alternos. Suponga que la transposición es $\rho(i, j, k)$ y suponga que los vértices correspondientes en $G(\pi)$ implican la permutación; a saber, π_{i-1}, π_i, π_{j-1}, π_j, π_{k-1}, π_k están en un ciclo que se muestra en la figura 9-34.

FIGURA 9-34 Caso especial de transposición con $\Delta c(\rho) = 2$.

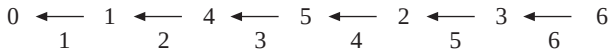


Como se muestra en la figura 9-34, este tipo particular de transposición incrementa por dos el número de ciclos. Así, se tiene una mejor cota inferior $d(\pi) = \dfrac{n+1-c(\pi)}{2}$. Cualquier algoritmo de ordenamiento por transposición que produce una distancia de transposición igual a esta cota inferior $\dfrac{n+1-c(\pi)}{2}$ debe ser un algoritmo óptimo. Hasta la fecha todavía no existe un algoritmo así. Por supuesto, hay una ligera probabilidad de que la cota inferior no sea demasiado alta. A continuación se obtendrá un algoritmo de 2-aproximación.

El caso en la figura 9-34 es bastante deseable porque se trata de una transposición que incrementa por dos el número de ciclos. Idealmente, se espera que en todo momento haya este tipo de ciclos en nuestra permutación. Desafortunadamente, no ocurre así. Como resultado, es necesario manejar los otros casos.

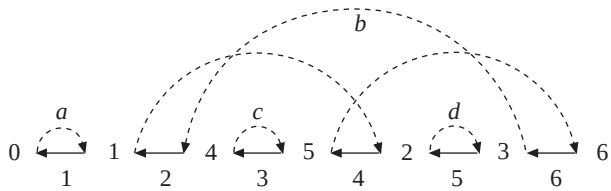
Primero se asigna un número de 1 a $n + 1$ a las aristas negras de $G(\pi)$, y se dice que una transposición ρ (i, j, k) actúa sobre las aristas negras i, j y k. En la figura 9-35 se muestra un ejemplo para asignación del número de las aristas negras.

FIGURA 9-35 Permutación con las aristas negras de $G(\pi)$ etiquetadas.



Observe que un ciclo puede representarse por $(i_1, \ldots, i_k)$ según el recorrido a las aristas negras de i_1 a i_k, donde i_1 es la arista negra más a la derecha en el ciclo. Por ejemplo, hay cuatro ciclos alternos en el $G(\pi)$ en la figura 9-36. La arista negra más a la derecha en el ciclo b es la arista negra 6, por lo que el ciclo b se representa como (6, 2, 4).

FIGURA 9-36 Una permutación con $G(\pi)$ que contiene cuatro ciclos.

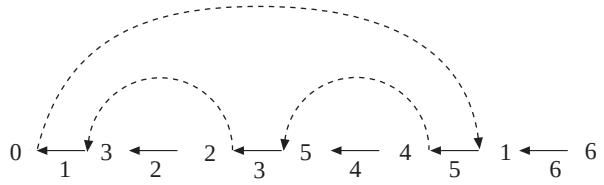


Hay dos clases distintas de ciclos: los ciclos no orientados y los ciclos orientados.* Para toda $k > 1$, un ciclo $C = (i_1, \ldots, i_k)$ es no orientado si $i_1, \ldots, i_k$ es una secuencia decreciente; C es orientado en caso contrario. En la figura 9-37 se muestran dos ejemplos sobre ciclos no orientados y ciclos orientados.

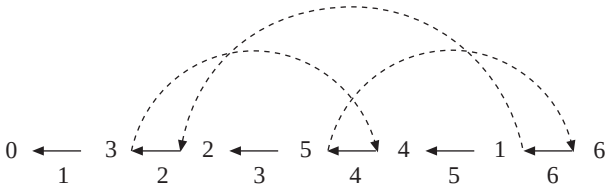
Una transposición ρ se denomina *movimiento x* si $\Delta c(\rho) = x$. Suponga que un ciclo $C = (i_1, \ldots, i_k)$ es un ciclo orientado. Puede demostrarse que existen un i_t en C, $i_t > i_{t-1}$, $3 \leq t \leq k$ y una transposición $\rho(i_{t-1}, i_t, i_1)$ tales que $\rho\pi$ crea un 1 ciclo que contiene a los vértices $\pi_{i_{t-1}-1}$ y π_{i_t} y otros ciclos. En consecuencia, ρ es una transposición de 2 movimientos. En conclusión, en todo ciclo orientado hay una transposición de 2 movimientos. En la figura 9-38 se muestra un ejemplo. En este ejemplo, la permutación de entrada es 0 4 5 1 6 3 2 7.

* El lector no debe confundir los ciclos orientados con los ciclos dirigidos. (*N. del R.T.*)

FIGURA 9-37 Ciclos orientados y no orientados.



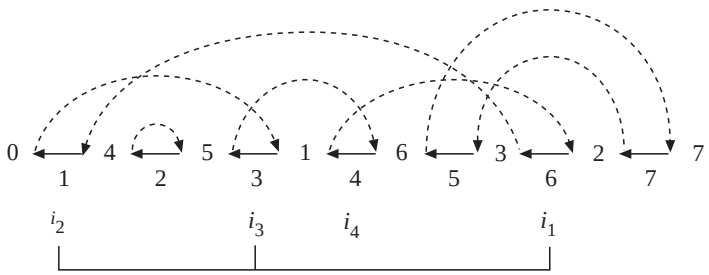
a) Ciclo no orientado (5, 3, 1)



b) Ciclo orientado (6, 2, 4)

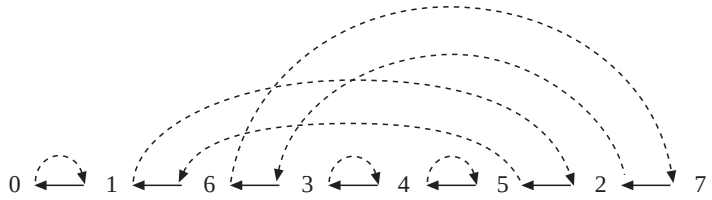
Como se muestra en la figura 9-38*a*), hay tres ciclos. De ellos, el ciclo (6, 1, 3, 4) es un ciclo orientado. Como se muestra en la figura 9-38*a*), hay una transposición $\rho(i_2, i_3, i_1) = \rho(1, 3, 6)$. Esta transposición corresponde a sustituir las subcadenas (4 5) y (1 6 3). Después de aplicar esta transposición, la permutación se convierte en 0 1 6 3 4 5 2 7 y por lo tanto hay cinco ciclos. Como puede verse en la figura 9-38*b*), el número de ciclos se incrementó por dos.

FIGURA 9-38 Ciclo orientado que permite un 2-movimiento.



a) Una permutación: 0 4 5 1 6 3 2 7, $t = 3$, un ciclo orientado (6, 1, 3, 4) y una transposición $\rho(i_2, i_3, i_1)$

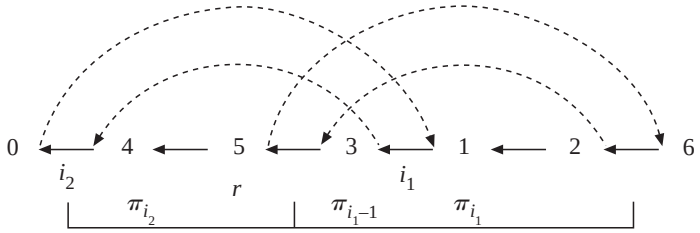
FIGURA 9-38 (*Continuación.*)



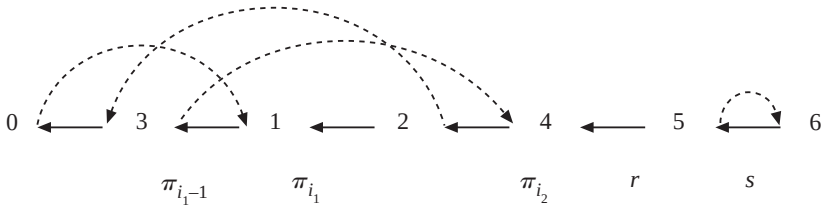
b) La gráfica de ciclos después de la transposición

Considere la figura 9-39*a*). Observe que en esta gráfica no hay ningún ciclo orientado. Puede verse fácilmente que debe haber más de un ciclo no orientado. Suponga que un ciclo $C = (i_1, i_2, \ldots, i_k)$ es un ciclo no orientado. Sea r la posición del elemento maximal de la permutación π en el intervalo $[i_2, i_1 - 1]$, y sea r la posición de $\pi_r + 1$ en π. Puede demostrarse que $s > i_1$ y que la transposición $\rho(r + 1, s, i_2)$ es un 0-movimiento que transforma un ciclo no orientado C en un ciclo orientado C' permitiendo un 2-movimiento. En consecuencia, en un ciclo no orientado existe un 0-movimiento seguido de un 2-movimiento sobre un ciclo no orientado. En la figura 9-39 se muestra un ejemplo.

FIGURA 9-39 Ciclo orientado que permite movimientos 0 y 2.



a) Una permutación: 0 4 5 3 1 2 6, un ciclo no orientado $C = (i_1, i_2)$, $r = 2$, $s = 6$, una transposición $\rho(3, 6, 1)$, $\pi_{i_1} = 1$, $\pi_{i_1-1} = 3$ y $\pi_{i_2} = 4$



b) La gráfica de ciclos después de la transposición

Con base en el análisis anterior, puede verse que para una permutación arbitraria π existe ya sea una permutación de 2-movimiento o de 0-movimiento seguida de una permutación de 2-movimiento. Por lo tanto, se ha obtenido una cota superior de la distancia de transposición. Es decir, $d(\pi) \leq \frac{n+1-c(\pi)}{2/2} = n+1-c(\pi)$ para ordenar por transposiciones. Así, se tiene un algoritmo capaz de producir una distancia de transposición no mayor que $n + 1 - c(\pi)$.

Debido a la cota inferior $d(\pi) \geq \frac{n+1-c(\pi)}{2}$, y a la cota superior $d(\pi) \leq n + 1 - c(\pi)$, hay un algoritmo de aproximación para ordenar por transposiciones con razón de desempeño igual a 2. Este algoritmo se presenta en el algoritmo 9-8.

Algoritmo 9-8 □ Un algoritmo de 2-aproximación para encontrar una solución por aproximación para el problema de ordenamiento por transposiciones

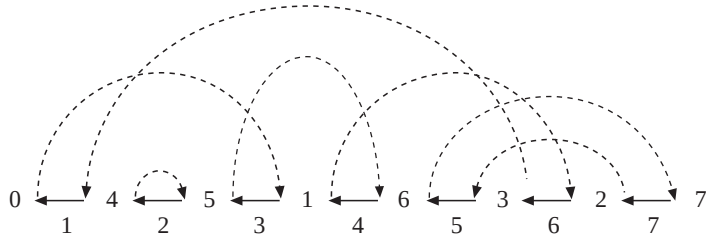
Input: Dos permutaciones π y σ.
Output: La distancia mínima entre dos permutaciones π y σ.
Paso 1: Reetiquetar dos permutaciones para ordenar la permutación π en la permutación identidad.
Paso 2: Construir la gráfica de ciclos $G(\pi)$ de la permutación π. Hacer la distancia $d(\pi) = 0$.
Paso 3: Mientras haya un ciclo orientado
    Realizar un 2-movimiento, $d(\pi) = d(\pi) + 1$
  Mientras hay un ciclo no orientado
    Realizar un 0-movimiento seguido de un 2-movimiento, $d(\pi) = d(\pi) + 2$
Paso 4: Output la distancia $d(\pi)$.

En la figura 9-40 se muestra un ejemplo de algoritmo 2-aproximación. El algoritmo es como sigue:

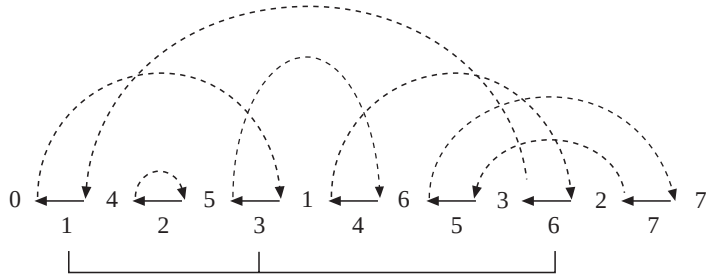
1. Debido a que hay un ciclo orientado (6, 1, 3, 4), sobre él se ejecuta una transposición ρ(6, 3, 1). El resultado se muestra en la figura 9-40*c*).
2. Como se muestra en la figura 9-40*b*), en la gráfica de ciclos no existe ningún ciclo orientado, y hay dos ciclos orientados (6, 2) y (7, 3). La transposición ρ(3, 7, 2) se ejecuta seguida de la ρ(6, 5, 2).

El resultado de la transposición $\rho(7, 3, 2)$ se muestra en la figura 9-40*d*), y el resultado que se obtiene al aplicar la transposición $\rho(6, 5, 2)$ se muestra en la figura 9-40*e*).

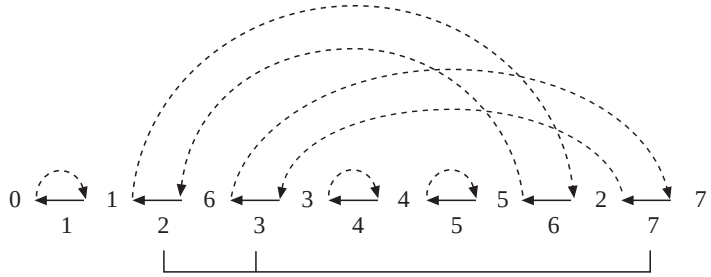
FIGURA 9-40 Un ejemplo de algoritmo de 2-aproximación.



a) La gráfica de ciclos de una permutación: 4 5 1 6 3 2, un ciclo orientado (6, 1, 3, 4), un ciclo (2) y un ciclo no orientado (7, 5)

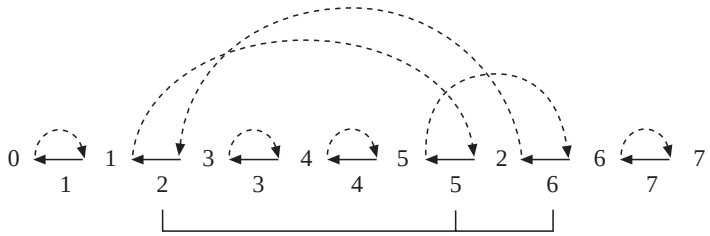


b) Una transposición $\rho(6, 3, 1)$ para el ciclo orientado (6, 1, 3, 4)



c) La gráfica de ciclos después de un 2-movimiento, dos ciclos no orientados (6, 2) y (7, 3), otros ciclos (1), (4) y (5) y una transposición $\rho(3, 7, 2)$ para el ciclo no orientado

FIGURA 9-40 (*Continuación.*)



d) La gráfica de ciclos después de *c*), un ciclo orientado (6, 2, 5), otros ciclos (1), (3), (4) y (7) y una transposición $\rho(6, 5, 2)$ que actúa sobre el ciclo orientado

e) La gráfica de ciclos final después de tres transposiciones, un 2-movimiento, un 0-movimiento y un 2-movimiento

Todo el proceso de aplicación de las transposiciones es como sigue:

0 4 5 1 6 3 2 7
0 1 6 3 4 5 2 7
0 1 3 4 5 2 6 7
0 1 2 3 4 5 6 7.

9-9 EL ESQUEMA DE APROXIMACIÓN EN TIEMPO POLINOMIAL

Para todo algoritmo de aproximación existe un error asociado. Por supuesto, lo mejor sería que este error fuese lo más pequeño posible. Es decir, sería deseable contar con una familia de algoritmos de aproximación tal que para todo error haya un algoritmo de aproximación correspondiente que alcance este error. Entonces, sin importar cuán pequeño se especifique el error, es posible alcanzarlo aplicando el algoritmo de aproximación correspondiente. Naturalmente, por este pequeño error se paga un precio porque la complejidad temporal del algoritmo con un error más pequeño debe ser mayor que para un algoritmo con un error más grande. Sería ideal que sin importar cuán pequeño sea el error, la complejidad temporal siguiera siendo polinomial. El análisis anterior conduce al concepto de esquema de aproximación en tiempo polinomial (PTAS, por su nombre en inglés: polynomial time approximation scheme).

Sean S_{OPT} el costo de una solución óptima y S_{APX} el costo de una solución aproximada. Ahora, la tasa de error se define como

$$\varepsilon = (S_{OPT} - S_{APX})S_{OPT}.$$

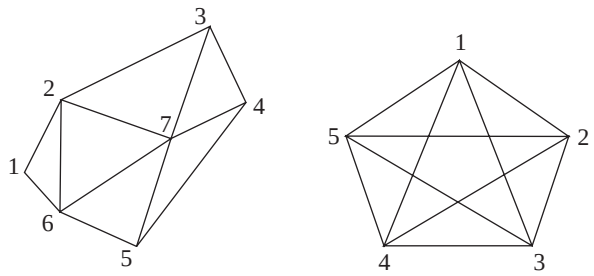
Un PTAS para un problema denota una familia de algoritmos de aproximación tal que para cada tasa de error especificada de antemano E existe un algoritmo de aproximación que conserva una complejidad temporal polinomial. Por ejemplo, suponga que la complejidad temporal de nuestro algoritmo es $O(n/E)$. Entonces, sin importar cuán pequeño pueda ser E, sigue teniéndose un algoritmo de aproximación de tiempo polinomial con respecto a E, porque E es una constante.

El PTAS para el problema del máximo conjunto independiente en gráficas planas

En este apartado se demostrará que para el problema del máximo conjunto independiente en gráficas planas hay un PTAS donde la complejidad temporal de cada algoritmo es $O(8^k kn)$, donde $k = \lceil 1/E \rceil - 1$.

Primero se definen las gráficas planas. *Se dice que una gráfica está incrustada* (embedded) *en una superficie S si puede trazarse sobre S de modo que sus aristas se intersectan sólo en los vértices. Una gráfica es plana si es posible incrustarla en un plano*. En la figura 9-41*a*) se muestra una gráfica plana, y la gráfica de la figura 9-41*b*) no es plana.

FIGURA 9-41 Gráficas.



a) Una gráfica plana *b*) Una gráfica que no es plana

El problema del conjunto independiente máximo sobre gráficas planas es NP-difícil. Así, los algoritmos de aproximación son aconsejables. Primero se definirán algunos términos. Una cara es una región determinada por un encaje plano. Las caras no acotadas se denominan caras externas y las otras, caras internas. Por ejemplo, para la cara en la figura 9-42, la cara $6 \rightarrow 7 \rightarrow 11 \rightarrow 12 \rightarrow 6$ es una cara interna, y la cara $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1$ es una cara externa. En una gráfica plana es posible asociar cada nodo con un nivel. En la figura 9-42 sólo hay una cara externa; a saber, $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1$. Así, los nodos 1, 2, 3, 4 y 5 son de nivel 1. A continuación, los nodos 6, 7, 11, 12, 8, 9, 10, 13 y 14 son de nivel 2, y los nodos 15, 16 y 17 son de nivel 3. Los niveles de los nodos pueden calcularse en tiempo lineal.

Una gráfica es outerplanar *(periplana) si no tiene nodos de nivel superior a k*. Por ejemplo, la figura 9-43 contiene una gráfica 2-*outerplanar*.

FIGURA 9-42 Una gráfica plana incrustada.

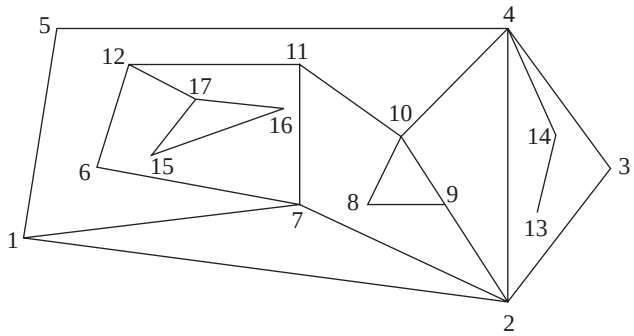
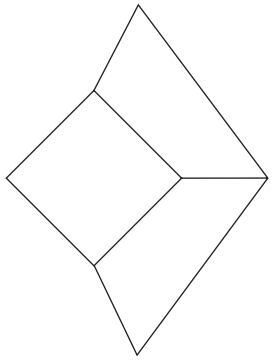


FIGURA 9-43 Ejemplo de una gráfica *outerplanar*.



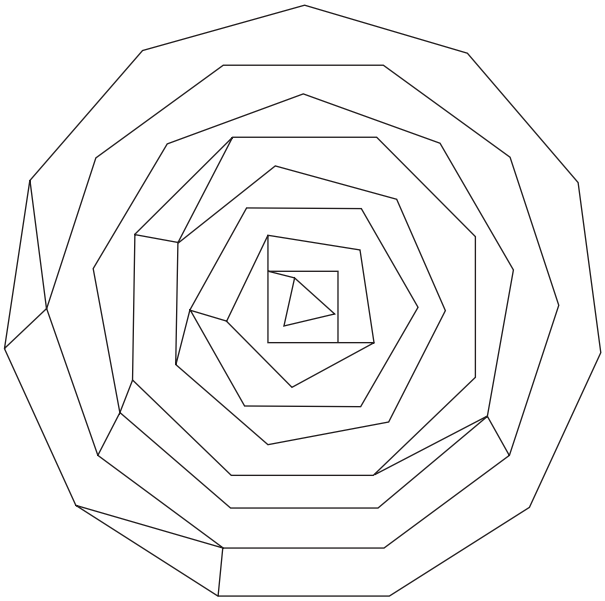
Para una gráfica *k-outerplanar*, una solución óptima para el problema del conjunto independiente máximo puede encontrarse en tiempo $O(8^k n)$ con el método de programación dinámica, donde n es el número de vértices. Aquí no se proporcionarán detalles de este método.

Dada una gráfica plana arbitraria G, es posible descomponerla en un conjunto de k gráficas *outerplanar*. Considere la figura 9-44. Suponga que k se iguala a 2. Luego, todos los nodos en los niveles 3, 6 y 9 se agrupan en la clase 3, los nodos en los niveles 1, 4 y 7 en la clase 1, y los nodos en los niveles 2, 5 y 8 en la clase 2, como se muestra en la tabla 9-1.

TABLA 9-1 Descomposición de los nodos.

Niveles	1	4	7	(clase 1)
Niveles	2	5	8	(clase 2)
Niveles	3	6	9	(clase 3)

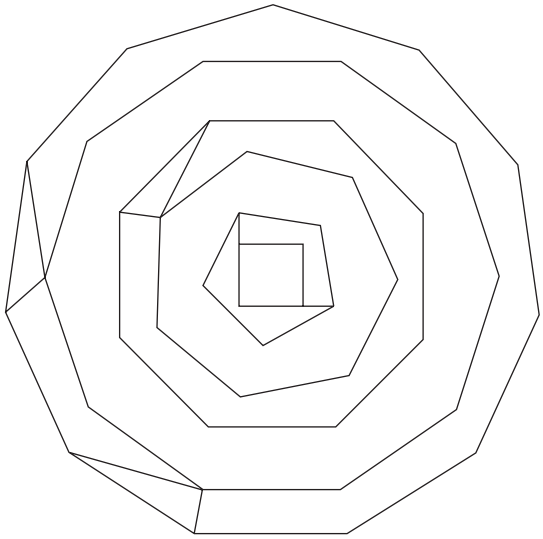
FIGURA 9-44 Gráfica plana con nueve niveles.



Si se eliminan todos los nodos en la clase 3, a saber, los nodos en los niveles 3, 6 y 9, se obtiene la gráfica resultante que se muestra en la figura 9-45. Resulta evidente que todas las subgráficas en la figura 9-45 son 2-*outerplanar*. Para cada una de estas gráficas es posible encontrar su conjunto independiente máximo en tiempo lineal. Además, la unión de estos conjuntos independientes máximos sigue siendo un conjunto independiente (no necesariamente máximo) para la gráfica plana original, de modo que puede funcionar como una solución óptima para ésta.

De manera semejante es posible eliminar los nodos en la clase 1; a saber, los nodos en los niveles 1, 4 y 7. De nuevo, la gráfica resultante también consta de un conjunto de gráficas 2-*outerplanar*. Al utilizar mecanismos semejantes es posible obtener un conjunto independiente máximo para nuestra gráfica plana original.

FIGURA 9-45 Gráfica obtenida al eliminar los nodos en los niveles 3, 6 y 9.



Luego, todos los niveles de los nodos en la clase 1 son congruentes con 1 (mod $k + 1$) y todos los niveles de los nodos en la clase 2 son congruentes con 2 (mod $k + 1$). Nuestro algoritmo de aproximación, basado en cierta k especificada de antemano, funciona como sigue:

Algoritmo 9-9 □ Un algoritmo de aproximación para resolver el problema del máximo conjunto independiente en gráficas planas

Paso 1. Para toda $i = 0, 1, \ldots, k$, hacer
- **(1.1)** Hacer G_i la gráfica obtenida al eliminar todos los nodos con niveles congruentes con i (mod $k + 1$). Todas las subgráficas restantes son *k-outerplanar*.
- **(1.2)** Para toda gráfica *k-outerplanar*, encontrar su conjunto independiente máximo. Hacer S_i la unión de estas soluciones.

Paso 2. Entre $S_0, S_1, \ldots, S_k$, escoger la S_j de tamaño máximo y hacer que sea la solución aproximada S_{APX}.

Resulta evidente que la complejidad temporal de nuestro algoritmo de aproximación es $O(8^k kn)$. A continuación se demostrará que la k es inversamente proporcional a

la tasa de error. Así, el problema del conjunto independiente máximo en gráficas planas tiene un PTAS.

Observe que todos los nodos se han dividido en $(k + 1)$ clases; cada clase corresponde a un nivel congruente con $i \pmod{k+1}$ para $i = 0, 1, \ldots, k$. Para todo conjunto independiente S, el número promedio de nodos en este conjunto para cada clase es $|S|/(k + 1)$ donde $|S|$ es el número de nodos en este conjunto independiente. Así, hay por lo menos una r tal que cuando mucho $\frac{1}{k+1}$ de vértices en S_{OPT} está en un nivel que es congruente con $r \pmod{k+1}$. Esto significa que la solución S_r obtenida al eliminar los nodos de S_{OPT} en la clase r tienen por lo menos $|S_{OPT}|\left(1-\frac{1}{k+1}\right)=|S_{OPT}|\frac{k}{k+1}$ nodos, ya que cuando mucho se han eliminado $|S_{OPT}|\frac{1}{k+1}$ nodos. En consecuencia,

$$|S_r| \geq |S_{OPT}|\frac{k}{k+1}.$$

Según nuestro algoritmo,

$$|S_{APX}| \geq |S_r| \geq |S_{OPT}|\frac{k}{k+1}$$

o bien,

$$\varepsilon = \frac{|S_{OPT} - S_{APX}|}{|S_{OPT}|} \leq \frac{1}{k+1}.$$

Entonces, si se hacet $k = \lceil 1/E \rceil - 1$, la fórmula anterior se convierte en

$$\varepsilon \leq \frac{1}{k+1} = \frac{1}{\lceil 1/E \rceil} \leq E.$$

Esto demuestra que para toda cota de error dada E se tiene una k correspondiente para garantizar que la solución aproximada difiere de la óptima por menos que esta tasa de error. Además, no importa cuán pequeño sea el error, siempre puede encontrarse un algoritmo para alcanzar dicho error con la complejidad temporal $O(8^k kn)$, que es polinomial con respecto a n.

El PTAS para el problema 0-1 de la mochila

El problema 0-1 de la mochila es NP-difícil y se estudió en el capítulo 5. De hecho, es sorprendente que para este problema exista un esquema de aproximación de tiempo polinomial.

El problema 0-1 de la mochila se define como sigue: se tienen n artículos. El i-ésimo artículo tiene ganancia p_i y peso w_i. Dado un entero M, el problema 0-1 de la mochila consiste en seleccionar un subconjunto de estos n artículos de modo que la suma de ganancias se maximice bajo la restricción de que la suma de pesos no excede a M. Formalmente, se quiere maximizar $\Sigma\delta_i p_i$ donde $\delta_i = 1$ o 0 de modo que $\Sigma\delta_i w_i \leq M$.

Se considerará un ejemplo, $M = 92$, $n = 8$ y los pesos y las ganancias se muestran en la tabla 9-2, donde los artículos están dispuestos según el orden no decreciente de p_i/w_i.

TABLA 9-2 Ganancias y pesos de ocho artículos.

i	1	2	3	4	5	6	7	8
p_i	90	61	50	33	29	23	15	13
w_i	33	30	25	17	15	12	10	9
p_i/w_i	2.72	2.03	2.0	1.94	1.93	1.91	1.5	1.44

Básicamente, el algoritmo de aproximación se basa en una tasa de error ε. Una vez que se proporciona este ε, se estima, a través de un cálculo elaborado, un umbral, denominado T. Con este umbral T, todos los artículos se dividen en dos subconjuntos: *BIG* y *SMALL*. *BIG* contiene todos los artículos cuyas ganancias son mayores que T y *SMALL*, los artículos cuyas ganancias son menores o iguales a T.

En nuestro caso, se encuentra que T es 46.8. Así, $BIG = \{1, 2, 3\}$ y $SMALL = \{4, 5, 6, 7, 8\}$.

Una vez que se obtienen *BIG* y *SMALL*, las ganancias de los artículos se normalizan en *BIG*. En nuestro caso, las ganancias normalizadas son

$$p'_1 = 9$$
$$p'_2 = 6$$
y $$p'_3 = 5.$$

A continuación se intentará enumerar todas las soluciones posibles para este caso del problema. Hay muchas, pero se considerarán dos de ellas:

Solución 1: Se seleccionan los artículos 1 y 2. La suma de las ganancias normalizadas es 15. La suma correspondiente de las ganancias originales es 90 + 61 = 151. La suma de pesos es 63.

Solución 2: Se seleccionan los artículos 1, 2 y 3. La suma de las ganancias normalizadas es 20. La suma correspondiente de las ganancias originales es 90 + 61 + 50 = 201. La suma de pesos es 88.

Como la suma de pesos es menor que $M = 92$, ahora es posible sumar artículos de *SMALL* a ambas soluciones. Esto puede hacerse aplicando el método codicioso.

Solución 1: Para la solución 1, es posible sumar los artículos 4 y 6. La suma de ganancias es 151 + 33 + 23 = 207.

Solución 2: Para la solución 2, no es posible sumar ningún artículo de *SMALL*. Así, la suma de ganancias es 201.

Por supuesto, hay muchas soluciones. Para cada una, se aplica el método codicioso y se obtiene una solución aproximada. Puede demostrarse que la solución 1 es la mayor. Así, la solución 1 se proporciona como la solución aproximada. Es importante observar que un algoritmo de aproximación debe ser un algoritmo polinomial. La parte codiciosa es evidentemente polinomial. La parte crítica es la parte del algoritmo que encuentra todas las soluciones factibles para los artículos en *BIG*. Una de estas soluciones factibles es una solución óptima para el problema 0-1 de la mochila con los artículos. Si esta parte del algoritmo es polinomial, ¿significa que se ha utilizado un algoritmo polinomial para resolver el problema 0-1 de la mochila?

Después se demostrará que para esta parte en realidad se está resolviendo una versión especial del problema 0-1 de la mochila. Después se demostrará que en este caso del problema, las ganancias se normalizaron en la medida en que la suma de las ganancias normalizadas fuera menor que $\lfloor(3/\varepsilon)^2\rfloor$, donde ε es la tasa de error. Es decir, la suma de ganancias normalizadas es menor que una constante. Una vez que se satisface esta condición, existe un algoritmo de tiempo polinomial que produce todas las soluciones factibles. Así, nuestro algoritmo de aproximación es polinomial.

Ya que se ha proporcionado el bosquejo al más alto nivel de nuestro algoritmo de aproximación, a continuación se proporcionarán los detalles del algoritmo. De nuevo, los detalles se describirán al aplicar el algoritmo a nuestros datos. Se hace que ε sea igual a 0.6.

Paso 1: Ordenar los artículos según la razón no decreciente ganancia a peso p_i/w_i (tabla 9-3).

TABLA 9-3 Artículos ordenados.

i	1	2	3	4	5	6	7	8
p_i	90	61	50	33	29	23	15	13
w_i	33	30	25	17	15	12	10	9
p_i/w_i	2.72	2.03	2.0	1.94	1.93	1.91	1.5	1.44

Paso 2: Calcular un número Q como sigue:
Encontrar la d más grande tal que
$W = w_1 + w_2 + \cdots + w_d \leq M$.
Si $d = n$ o $W = M$, entonces

Se hace $P_{APX} = p_1 + p_2 + \cdots + p_d$ e $ÍNDICES = \{1, 2, \ldots, d\}$ y se detiene el proceso.

En este caso, $P_{OPT} = P_{APX}$.

En caso contrario, se hace $Q = p_1 + p_2 + \cdots + p_d + p_{d+1}$.
Para nuestro caso, $d = 3$ y $Q = 90 + 61 + 50 + 33 = 234$.
¿Cuáles son las características de Q? A continuación se demostrará que

$$Q/2 \leq P_{OPT} \leq Q.$$

Observe que $p_1 + p_2 + \cdots + p_d \leq P_{OPT}$.
Debido a que $W_{d+1} \leq W$, en consecuencia p_{d+1} de suyo es una solución factible.

$$p_{d+1} \leq P_{OPT}.$$

En consecuencia, $Q = p_1 + p_2 + \cdots + p_d + p_{d+1} \leq 2P_{OPT}$

O bien $Q/2 \leq P_{OPT}$.

Debido a que P_{OPT} es una solución óptima y Q no lo es, se tiene $P_{OPT} \leq Q$.

Así, $Q/2 \leq P_{OPT} \leq Q$.

Después se demostrará que esto es fundamental para el análisis de error.

Paso 3: Calcular un factor de normalización δ como sigue:

$$\delta = Q(\varepsilon/3)^2.$$

En nuestro caso, $\delta = 234(0.6/3)^2 = 234(0.2)^2 = 9.36$
Luego se calcula un parámetro g:

$$g = \lfloor Q/\delta \rfloor = (3/\varepsilon)^2 = \lfloor (3/0.6)^2 \rfloor = 25.$$

Sea $T = Q(\varepsilon/3)$.
En nuestro caso, $T = 234(0.6/3) = 46.8$.

Paso 4:

Paso 4.1: Se hace que *SMALL* reúna todos los artículos cuyas ganancias son menores o iguales a *T*. Todos los otros artículos se reúnen en *BIG*.
En nuestro caso, $SMALL = \{4, 5, 6, 7, 8\}$
y $BIG = \{1, 2, 3\}$.

Paso 4.2: Para todos los artículos en *BIG*, se normalizan sus ganancias según la fórmula siguiente: $p'_i = \lfloor p_i/\delta \rfloor$.
En nuestro caso, $p'_1 = \lfloor 90/9.36 \rfloor = 9$
$p'_2 = \lfloor 61/9.36 \rfloor = 6$
$p'_3 = \lfloor 50/9.36 \rfloor = 5$.

Paso 4.3: Se inicializa una disposición *A* de tamaño *g*. Cada elemento de la disposición corresponde a una combinación de los p'_i. Cada elemento $A[i]$ consta de tres campos, *I*, *P*, *W*, que representan el índice de la combinación, la suma de ganancias y la suma de pesos, respectivamente.

Paso 4.4: Para cada artículo en *BIG*, se explora la tabla y se ejecutan las siguientes operaciones para cada elemento: para el elemento $A[j]$, si ya hay algo en $A[j]$ y sumar el artículo *i* a ello no se provoca que el peso total exceda el límite de capacidad *M*, entonces se comprueba el peso correspondiente a $A[j + p'_i]$, que corresponde a una combinación de sumar el artículo *i* a $A[j]$. Si en $A[j + p'_i]$ no hay nada o el peso $A[j + p'_i] \cdot W$ es mayor que $A[j] \cdot W + w_i$ (que es el peso que corresponde al sumar el artículo *i* con $A[j]$), entonces se actualiza el elemento con $A[j] \cdot I \cup \{i\}$, y se actualizan la ganancia y el peso correspondientes. Nuestro ejemplo es como sigue.

Cuando $i = 1$, $p_i' = 9$, consulte la tabla 9-4.

TABLA 9-4 La disposición A para $i = 1$, $p_i' = 9$.

p_i	I	P	W
0		0	0
1			
2			
3			
4			
5			
6			
7			
8			
9	1	90	33
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			

$i = 2$, $p_2' = 6$, consulte la tabla 9-5.

TABLA 9-5 La disposición A para $i = 2$, $p_2' = 6$.

p_i	I	P	W
0		0	0
1			
2			
3			
4			
5			
6	2	61	30
7			
8			
9	1	90	33
10			
11			
12			
13			
14			
15	1, 2	151	63
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			

$i = 3$, $p'_3 = 5$, consulte la tabla 9-6.

TABLA 9-6 La disposición A para $i = 3$, $p'_3 = 5$.

p_i	I	P	W
0		0	0
1			
2			
3			
4			
5	3	50	25
6	2	61	30
7			
8			
9	1	90	33
10			
11	2, 3	111	55
12			
13			
14	1, 3	140	58
15	1, 2	151	63
16			
17			
18			
19			
20	1, 2, 3	201	88
21			
22			
23			
24			
25			

Se analizará la tabla 9-6. Cada elemento no vacío se refiere a una solución factible. Por ejemplo, el elemento $A[j] = 5$ corresponde a la selección de un solo artículo; a saber, el artículo 3.

Su suma total de ganancias normalizadas es 5, su ganancia total es 50 y su peso total es 25. Para $j = 15$, se seleccionan los artículos 1 y 2. En este caso las ganancias normalizadas totales son 15, su ganancia total es 151 y su peso total es 63.

Paso 5. Para cada elemento de la disposición *A*, se aplica el método codicioso para sumar artículos en *SMALL* a su combinación correspondiente.

TABLA 9-7 La disposición final *A*.

p_i	*I*	*P*	*W*	FILL	Ganancia
0		0	0	4, 5, 6, 7, 8	113
1					
2					
3					
4					
5	3	50	25	4, 5, 6, 7, 8	163
6	2	61	30	4, 5, 6, 7	161
7					
8					
9	1	90	33	4, 5, 6, 7	190
10					
11	2, 3	111	55	4, 5	173
12					
13					
14	1, 3	140	58	4, 5	202
15	1, 2	151	63	4, 6	207
16					
17					
18					
19					
20	1, 2, 3	201	88		201
21					
22					
23					
24					
25					

De nuevo, se analizará el significado de la tabla 9-7. Para $j = 15$, es posible sumar los artículos 4 y 6. Así se obtiene una ganancia total de 207. Para $j = 20$, se seleccionan los artículos 1, 2 y 3 de *BIG*. En este caso, no es posible sumar ningún otro artículo de *SMALL* porque se excedería la restricción de peso. Así, el peso total es 201.

Paso 6. Como nuestra solución por aproximación se elige la ganancia más grande obtenida en el paso 5. En nuestro ejemplo, se elige el elemento correspondiente a $\Sigma p'_i = 15$, lo cual genera la ganancia total más grande, igual a 207.

El lector puede ver que el paso 4.4 es un paso de exploración exhaustiva. Se tienen estos artículos con ganancias normalizadas iguales a 9, 6 y 5. Los pesos correspondientes son 33, 30 y 25, respectivamente. Se inicia con el artículo cuya ganancia es 9. Debido a que su peso es menor que *M*, es posible seleccionarlo solo. El peso total es 33 y la ganancia normalizada total es 9. Entonces, es posible explorar el artículo cuya ganancia normalizada es 6. Es posible seleccionar solo este artículo. En caso de proceder así, la ganancia normalizada total es 6 y el peso total es 30. Si se escogen los dos artículos ya explorados, entonces la ganancia normalizada total es $9 + 6 = 15$ y el peso total es $33 + 30 = 63$. Así, se tienen cuatro soluciones factibles:

1. No seleccionar ningún artículo.
2. Seleccionar solo el artículo 1. Así se tendrán una ganancia normalizada total de 9 y un peso total de 33.
3. Seleccionar solo el artículo 2. Así se tendrán una ganancia normalizada total de 6 y un peso total de 30.
4. Seleccionar los dos artículos 1 y 2. La ganancia normalizada total es 15 y el peso total de 63.

Intuitivamente, este paso requiere tiempo exponencial: $2^{|BIG|}$. Se presenta una cuestión crítica: ¿por qué este proceso es polinomial?

A continuación se demostrará que el tamaño de la disposición *A* no es mayor que *g*.

Considere que el mayor elemento de *A* tiene ganancias normalizadas $p'_{i_1} + p'_{i_2} + \cdots + p'_{i_j}$. Como esto corresponde a una solución factible, se tiene $p_{i_1} + p_{i_2} + \cdots + p_{i_j} \leq P_{OPT} \leq Q$, donde, $p'_{i_j} = \lfloor p_{i_j}/\delta \rfloor$.

Considere $p'_{i_1} + p'_{i_2} + \cdots + p'_{i_j}$.

$$
\begin{aligned}
& p'_{i_1} + p'_{i_2} + \cdots + p'_{i_j} \\
&= \lfloor p_{i_1}/\delta \rfloor + \lfloor p_{i_2}/\delta \rfloor + \cdots + \lfloor p_{i_j}/\delta \rfloor \\
&\leq \lfloor (p_{i_1} + p_{i_2} + \cdots + p_{i_j})/\delta \rfloor \\
&\leq \lfloor Q/\delta \rfloor \\
&= g.
\end{aligned}
$$

Lo anterior explica por qué es suficiente contar con una disposición de tamaño g. Observe que g es una constante, que es independiente de n. La exploración requiere cuando mucho ng veces, y ésta es la razón de por qué el paso 4 es polinomial.

La complejidad temporal de este algoritmo de aproximación es como sigue:

Paso 1: $O(n \log n)$

Paso 2: $O(n)$

Paso 4.1 a 4.2: $O(n)$

Paso 4.3: $O(g)$

Paso 4.4: $O(ng)$

Paso 5: $O(ng)$

Paso 6: $O(n)$.

La complejidad temporal del algoritmo de aproximación es

$$O(n \log n) + O(n) + O(n) + O(g) + O(ng) + O(ng) + O(n)$$
$$= O(n \log n) + O(ng)$$
$$= O(n \log n) + O(n(3/\varepsilon)^2).$$

Así, nuestro algoritmo es un esquema de aproximación de tiempo polinomial. Su complejidad temporal es una función de ε, la tasa de error. Mientras más pequeña se establezca la tasa de error, más grande se vuelve la complejidad temporal. A pesar de ello, sigue siendo polinomial.

A continuación se llevará a cabo un análisis de error de nuestro algoritmo de aproximación. Primero, se requieren algunas afirmaciones:

Afirmación 1: $p'_i \geq 3/\varepsilon$, $\forall i \in BIG$.
Se sabe que $p'_i = \lfloor p_i/\delta \rfloor = \lfloor p_i/Q(\varepsilon/3)^2 \rfloor$,
pero $\forall i \in BIG$, $p_i > Q(\varepsilon/3)$,
en consecuencia, $p'_i \geq 3/\varepsilon$.

Afirmación 2: $p'_i\delta \leq p_i \leq p'_i\delta(1 + \varepsilon/3)$, $\forall i \in BIG$.
Observe que $p'_i = \lfloor p_i/\delta \rfloor$.
En consecuencia, $p'_i \leq p_i/\delta$.
Se tiene $p'_i\delta \leq p_i$.

Además, también se tiene $p_i/\delta \leq p'_i + 1 = p'_i(1 + 1/p'_i) \leq p'_i(1 + \varepsilon/3)$, ya que $p'_i \geq 3/\varepsilon$, por la afirmación 1.

Afirmación 3: Sean P_{OPT} la ganancia máxima de un problema 0/1 de la mochila, y $P_{Codicioso}$ la ganancia obtenida al aplicar el método codicioso al mismo problema. Entonces

$$P_{OPT} - P_{Codicioso} \leq \text{máx}\{p_i\}.$$

Demostración: Sea b el primer artículo no escogido por el algoritmo codicioso.
Por la elección de este algoritmo, $P_{Codicioso} + P_b \geq P_{OPT}$
Así, $P_{OPT} \leq P_{Codicioso} + P_b \leq P_{Codicioso} + \text{máx}\{p_i\}$,
De modo que $P_{OPT} - P_{Codicioso} \leq \text{máx}\{p_i\}$.

A continuación se abordará el análisis de error de nuestro algoritmo de aproximación. Primero se observa que P_{OPT} puede expresarse como sigue:

$$P_{OPT} = p_{i_1} + \cdots + p_{i_k} + \alpha,$$

donde $p_{i_1}, p_{i_2}, \ldots, p_{i_k}$ son artículos en *BIG* y α es la suma de las ganancias correspondientes a los artículos en *SMALL*.

Sean $c_{i_1}, \ldots, c_{i_k}$ los pesos asociados de $p_{i_1}, \ldots, p_{i_k}$. Considere $p'_{i_1} + \cdots + p'_{i_k}$. Debido a que $p'_{i_1} + \cdots + p'_{i_k}$ no es único, puede haber otro conjunto de artículos; a saber, los artículos $j_1, j_2, \ldots, j_h$, tales que $p'_{i_1} + \cdots + p'_{i_k} = p'_{j_1} + \cdots + p'_{j_h}$ y $c_{j_1} + \cdots + c_{j_h} \leq c_{i_1} + \cdots + c_{i_k}$. En otras palabras, nuestro algoritmo de aproximación escoge artículos $j_1, j_2, \ldots, j_h$, en vez de $i_1, i_2, \ldots, i_k$.

La suma $P_H = p_{j_1} + \cdots + p_{j_h} + \beta$ debió ser considerada por el algoritmo durante la ejecución del paso 6.

Resulta evidente que, por la elección de P_{APX},

$$P_{APX} \geq P_H.$$

En consecuencia,

$$P_{OPT} - P_{APX} \leq P_{OPT} - P_H.$$

Por la afirmación 2, se tiene

$$p_i\delta \leq p_i \leq p'_i\delta(1 + \varepsilon/3).$$

Al sustituir lo anterior en las fórmulas $P_{OPT} = p_{i_1} + \cdots + p_{i_k} + \alpha$ y $P_H = p_{j_1} + \cdots p_{j_h} + \beta$, se tiene

$$(p'_{i_1} + \cdots + p'_{i_k})\delta + \alpha \leq P_{OPT} \leq (p'_{i_1} + \cdots + p'_{i_k})\delta(1 + \varepsilon/3) + \alpha$$

y $$(p'_{j_1} + \cdots + p'_{j_h})\delta + \beta \leq P_H \leq (p'_{j_1} + \cdots + p'_{j_h})\delta(1 + \varepsilon/3) + \beta.$$

Debido a que $p'_{i_1} + \cdots + p'_{i_k} = p'_{j_1} + \cdots + p'_{j_h}$, se tiene

$$P_{OPT} \leq (p'_{i_1} + \cdots + p'_{i_k})\delta(1 + \varepsilon/3) + \alpha$$
y $$P_H \leq (p'_{i_1} + \cdots + p'_{i_k})\delta + \beta.$$

En consecuencia, se tiene

$$\begin{aligned}(P_{OPT} - P_H)/P_{OPT} &\leq ((p'_{i_1} + \cdots + p'_{i_k})\delta(\varepsilon/3) + \alpha - \beta)/P_{OPT} \\ &\leq \varepsilon/3 + (\alpha - \beta)/P_{OPT}. \end{aligned} \quad \textbf{(9–3)}$$

Además es necesario demostrar que

$$|(\alpha - \beta| \leq Q(\varepsilon/3).$$

Esto puede razonarse como sigue:

α es la suma de ganancias con respecto al problema 0/1 de la mochila definido sobre los artículos de *SMALL* con capacidad $M - (c_{i_1} + \cdots + c_{i_k})$.

β es la suma de ganancias obtenida con el método codicioso con respecto al problema 0/1 de la mochila definido sobre los artículos de *SMALL* con capacidad $M - (c_{j_1} + \cdots + c_{j_h})$. β' se definirá como la suma de ganancias obtenida con el método codicioso con respecto al problema 0/1 de la mochila definido sobre los artículos de *SMALL* con capacidad $M - (c_{i_1} + \cdots + c_{i_k})$.

Caso 1: $\alpha \geq \beta$
Debido a que se ha supuesto $c_{i_1} + \cdots + c_{i_k} \geq c_{j_1} + \cdots + c_{j_h}$,

$$M - (c_{i_1} + \cdots + c_{i_k}) \leq M - (c_{j_1} + \cdots + c_{j_h}).$$

Además, β y β' se obtienen con el método codicioso aplicado en *SMALL*, que se ordena de manera no decreciente. Por lo tanto, $M - (c_{i_1} + \cdots + c_{i_k}) \leq M - (c_{j_1} + \cdots + c_{j_h})$ que implica $\beta' \leq \beta$
En consecuencia,

$$\alpha - \beta \leq \alpha - \beta'.$$

Por la afirmación 3, también se tiene

$$\alpha - \beta \leq \text{máx}\{p_i \mid i \in SMALL\}.$$

Junto con la propiedad de que todo artículo en *SMALL* tiene ganancia menor o igual a $Q(\varepsilon/3)$, se obtiene

$$\alpha - \beta \leq Q(\varepsilon/3).$$

Caso 2: $\alpha \leq \beta$
Debido a que

$$(p'_{j_1} + \cdots + p'_{j_h})\delta + \beta \leq P_H \leq P_{OPT} \leq (p'_{i_1} + \cdots + p'_{i_k})\delta(1 + \varepsilon/3) + \alpha.$$

Por consiguiente,
$$\begin{aligned}\beta - \alpha &\leq (p'_{i_1} + \cdots + p'_{i_k})\delta(\varepsilon/3)\\ &\leq (p_{i_1} + \cdots + p_{i_k})(\varepsilon/3)\\ &\leq P_{OPT} \cdot \varepsilon/3\\ &\leq Q(\varepsilon/3).\end{aligned}$$

Al considerar el caso 1 y el caso 2 se tiene

$$|\alpha - \beta| \leq Q(\varepsilon/3). \qquad \textbf{(9-4)}$$

Al sustituir (9-4) en (9-3) se tiene

$$(P_{OPT} - P_H) \leq \varepsilon/3 + (\varepsilon/3)Q/P_{OPT}.$$

Por la elección de Q,

$$P_{OPT} \geq Q/2.$$

Junto con la desigualdad de que

$$P_{APX} \geq P_H,$$

se tiene

$$(P_{OPT} - P_{APX})/P_{OPT} \leq \varepsilon/3 + 2\varepsilon/3 = \varepsilon.$$

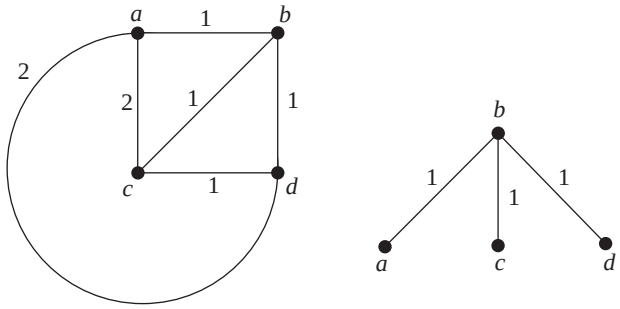
La ecuación anterior muestra que este algoritmo constituye un esquema de aproximación de tiempo polinomial.

9-10 UN ALGORITMO 2-APROXIMACIÓN PARA EL PROBLEMA DEL ÁRBOL DE EXPANSIÓN DE RUTA DE COSTO MÍNIMO

Este problema es semejante al problema del árbol de expansión mínimo, donde el costo total de todas las aristas del árbol de expansión debe minimizarse. En el problema del árbol de expansión de ruta de costo mínimo se tiene interés en el costo de ruta. Para dos nodos cualesquiera u y v en un árbol, entre ellos existe una ruta. El costo total de todas las aristas en esta ruta se denomina costo de ruta de este par de nodos. En nuestro caso, además se estipula que la gráfica dada es una gráfica completa y que todas las aristas de costos cumplen la desigualdad del triángulo. Así, el problema del árbol de expansión de ruta de costo mínimo se define como sigue: se tiene una gráfica completa G con aristas de costos que cumplen la desigualdad del triángulo, y el problema del árbol de expansión de ruta de costo mínimo consiste en encontrar un árbol de expansión de G cuya suma total de todos los pares de costos de ruta entre nodos se minimice.

Considere la figura 9-46. Un árbol de expansión de ruta de costo mínimo de la gráfica completa en la figura 9-46*a*) se muestra en la figura 9-46*b*).

FIGURA 9-46 Árbol de expansión de ruta de costo mínimo de una gráfica completa.



a) *b*)

Sea $RC(u, v)$ el costo de ruta entre u y v en un árbol. La suma total de todos los costos de ruta de este árbol es

$$2(RC(a,b) + RC(a,c) + RC(a,d) + RC(b,c) + RC(b,d) + RC(c,d))$$
$$2(1 + 2 + 2 + 1 + 1 + 2)$$
$$= 18,$$

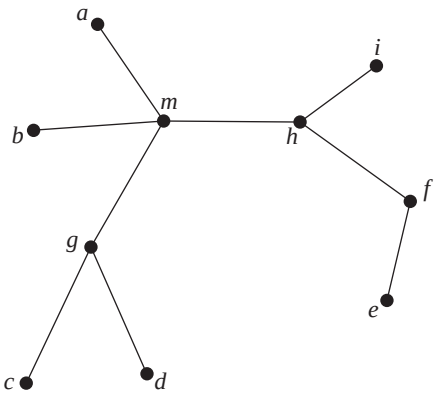
que es mínima de entre todos los árboles de expansión posibles.

El problema del árbol de expansión de ruta de costo mínimo es NP-difícil. En esta sección se proporcionará un algoritmo 2-aproximación para este problema. En la siguiente sección se presentará un PTAS para este problema.

Antes de presentar el algoritmo es necesario mencionar que cada par de nodos se cuenta dos veces. Es decir, se cuentan la ruta de u a v y la ruta de v a u. Esto se hace para simplificar el análisis.

El algoritmo 2-aproximación se basa en un concepto conocido como centroide. Un centroide de un árbol es un nodo cuya eliminación da por resultado subgráficas tales que cada subgráfica contiene no más de $n/2$ nodos, donde n es el número total de nodos del árbol. Considere la figura 9-47. El nodo m es un centroide.

FIGURA 9-47 El centroide de un árbol.

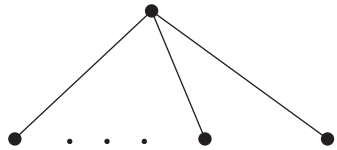


A continuación se considerará cualquier nodo v en un árbol arraigado por un centroide m de T. Por definición, todo subárbol de m contiene no más de $n/2$ nodos. Considere cualquier nodo v en T. El subárbol que contiene a v debe contener no más de $n/2$ nodos. Esto significa que hay por lo menos $n/2$ rutas entre algún nodo u y v que pasen por m. Por ejemplo, considere el nodo e del árbol en la figura 9-47. Sólo las rutas entre e y f, h e i no pasan por el centroide m. Las rutas de todos los demás nodos a e pasan por m. En consecuencia, en el costo de la ruta de este árbol, la longitud de la ruta de cualquier nodo u a m debe contarse $2(n/2) = n$ veces. Al usar $RC(u, v)$ como el costo de la ruta entre los nodos u y v, el costo del árbol $C(T)$ es

$$C(T) \geq n \sum_{u} RC(u, m). \qquad \textbf{(9-5)}$$

Suponga que el árbol T es un árbol de expansión de ruta de costo mínimo y que m es un centroide de T. Entonces la ecuación anterior sigue siendo válida. Luego, es posible usar m para obtener un algoritmo de aproximación. Una 1-estrella se definirá como un árbol que sólo tiene un nodo interno y todos los demás nodos son nodos hoja, lo cual se muestra en la figura 9-48.

FIGURA 9-48 Una 1-estrella.



Nuestro algoritmo de aproximación es para unir todos los nodos hacia m para formar una 1-estrella S, que también es un árbol de expansión. Sea $w(v, m)$ el peso de la arista entre v y w en la gráfica original G. La suma total de los costos de ruta de esta estrella, denotada por $C(S)$, es como sigue:

$$C(S) = (2n - 3)\sum_{v} w(v, m). \tag{9-6}$$

Pero $w(v, m)$ es menor que $RC(v, m)$ de T debido a la desigualdad del triángulo. Así, se tiene

$$C(S) = 2n\sum_{v} RC(v, m) \tag{9-7}$$

Esto significa que

$$C(S) \leq 2C(T). \tag{9-8}$$

Así, la 1-estrella S construida es una solución 2-aproximación para el problema del árbol de expansión de ruta de costo mínimo. Observe que se empieza a partir de un árbol de expansión de ruta de costo mínimo. Sin embargo, no se tiene ningún árbol de expansión de ruta de costo mínimo. En caso de contar con uno, no se necesitaría ningún algoritmo de aproximación para el problema.

¿Cómo puede encontrarse el centroide m de ese árbol sin el árbol? Observe que dada una gráfica completa G, sólo hay n 1-estrellas. Así, es posible realizar una búsqueda exhaustiva para construir todas las 1-estrellas posibles y como solución aproximada escoger la de costo mínimo. Así, una 1-estrella debe satisfacer nuestro requerimiento. La complejidad temporal de este algoritmo de aproximación es $O(n^2)$.

Quizá sea conveniente si el lector observa que acaba de demostrarse la existencia de una 1-estrella que satisface nuestro requerimiento. La búsqueda exhaustiva bien puede encontrar una 1-estrella cuyo costo de ruta sea menor que el recientemente analizado.

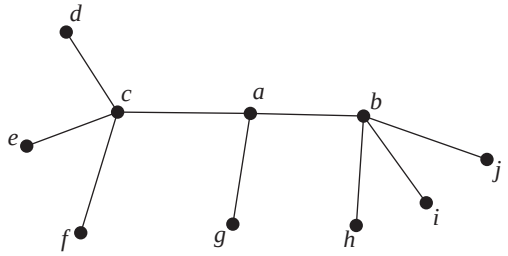
9-11 UN PTAS PARA EL PROBLEMA DEL ÁRBOL DE EXPANSIÓN DE RUTA DE COSTO MÍNIMO

En el apartado anterior se presentó un algoritmo 2-aproximación para el problema del árbol de expansión de ruta de costo mínimo. Nuestro algoritmo de aproximación se basa en la idea de que para toda gráfica G hay una 1-estrella cuyo costo de ruta es a lo sumo dos veces el costo de ruta del árbol de expansión de ruta de costo mínimo de la gráfica. Debido a que sólo hay n 1-estrellas, es posible encontrar una solución aproximada en un número polinomial de pasos.

En esta sección se presentará un PTAS para el problema del árbol de expansión de ruta de costo mínimo. Esencialmente, se construyen k-estrellas y mientras más grande es k, menor es el error.

Una k-estrella es un árbol con exactamente k nodos internos. En la figura 9-49 se muestra una 3-estrella.

FIGURA 9-49 Una 3-estrella.



En el resto de esta sección, *mcks*(G, k) denotará el costo de ruta de una ruta k-estrella de costo de ruta mínimo de una gráfica G. Sea *mrcst*(G) el costo de ruta de un árbol de expansión con costo de ruta mínimo de G. Entonces se demostrará lo siguiente:

$$mcks(G, k) \leq \left(\frac{k+3}{k+1}\right) mrcst(G). \tag{9-9}$$

Cuando $k = 1$, se está construyendo una 1-estrella y la ecuación anterior se convierte en:

$$mcks(G, 1) \leq 2mrcst(G) \tag{9-10}$$

que se demostró en el apartado previo.

La ecuación (9-9) demuestra que la tasa de error al usar una *k*-estrella para aproximar una solución óptima es

$$E = \frac{2}{k+1}. \tag{9-11}$$

Para $k = 2$, la tasa de error es 0.66 y para $k = 5$, la tasa de error se reduce a 0.33. Dada una cota de error, *k* puede seleccionarse por medio de la siguiente ecuación:

$$k = \left\lceil \frac{2}{E} - 1 \right\rceil. \tag{9-12}$$

Así, una *k* más grande significa una menor tasa de error. Para una cota de tasa de error dada especificada de antemano, es posible escoger una *k* correspondiente suficientemente grande para asegurar que el error inducido por esta *k*-estrella no excede la cota de tasa de error especificada de antemano. Puede demostrarse que la complejidad temporal requerida para encontrar una *k*-estrella de costo de ruta mínimo de una gráfica completa es $O(n^{2k})$. Para toda *k*, sin importar cuán grande sea, la complejidad temporal de nuestro algoritmo de aproximación sigue siendo polinomial. Es decir, se tiene un PTAS para el problema del árbol de expansión de ruta de costo mínimo.

Para encontrar una *k*-estrella de ruta de costo mínimo se requiere un concepto, denominado δ-separador, donde $0 < \delta \leq 1/2$. Dada una gráfica *G*, un δ-separador de *G* es una subgráfica mínima de *G* cuya eliminación da por resultado subgráficas, cada una de las cuales contiene no más de δn nodos. Para $\delta = 1/2$, el δ-separador sólo contiene un punto; a saber, el centroide, que se presentó en el apartado anterior. Puede demostrarse que hay una relación entre δ y *k* como sigue:

$$\delta = \frac{2}{k+3}, \tag{9-13}$$

o bien, al revés,

$$k = \frac{2}{\delta} - 3. \tag{9-14}$$

Al sustituir (9-14) en (9-9) se obtiene

$$mcks(G, k) \leq \left(\frac{1}{1-\delta}\right) mrcst(G). \tag{9-15}$$

Esencialmente, nuestro razonamiento es como sigue: una vez que se tiene una cota E para la tasa de error, se selecciona k mediante la ecuación (9-12) y luego se determina una δ mediante la ecuación (9-13). Mediante el δ-separador es posible determinar una k-estrella que satisface el requerimiento de la cota de error. Suponga que se especifica que E es 0.4. Entonces, usando la ecuación (9-12), se selecciona $k = (2/0.4) - 1 = 4$ y se encuentra $\delta = 2/(4 + 3) = 2/7 = 0.28$.

A continuación, primero se analizará el caso en que $k = 3$ para ilustrar el concepto fundamental del PTAS. En este caso, mediante la ecuación (9-13) se obtiene $\delta = 2/(3 + 3) = 1/3$.

Se supondrá que ya se ha encontrado un árbol de expansión T con costo de ruta mínimo. Sin pérdida de generalidad, puede suponerse que T está arraigado en su centroide m. Hay cuando mucho dos árboles que contienen más de $n/3$ nodos. Sean a y b los nodos más bajos con por lo menos $n/3$ nodos. Se ignoran los casos especiales en que $m = a$ o $m = b$. Estos dos casos pueden manejarse en forma semejante y producen la misma solución. Sea P la ruta en T que va del nodo a al nodo b. Esta ruta debe contener a m porque ninguno de los subárboles de m contiene más de $n/2$ nodos. Según nuestra definición, P es un (1/3)-separador. A continuación se demostrará cómo puede construirse una 3-estrella de costo de ruta mínimo con base en P. El costo de ruta de la 3-estrella no es mayor que

$$\frac{k+3}{k+1} = \frac{3+3}{3+1} = \frac{3}{2}$$

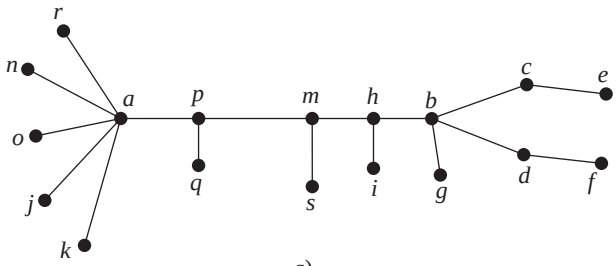
del costo de ruta de T.

Primero, todos los nodos se parten en V_a, V_b, V_m, V_{am} y V_{bm}. Sean V_a, V_b y V_m que constan de nodos cuyos ancestros más bajos en P son a, b y m, respectivamente. Sea $V_{am}(V_{bm})$ que consta de los nodos cuyos ancestros más bajos en P están entre a y m (entre b y m).

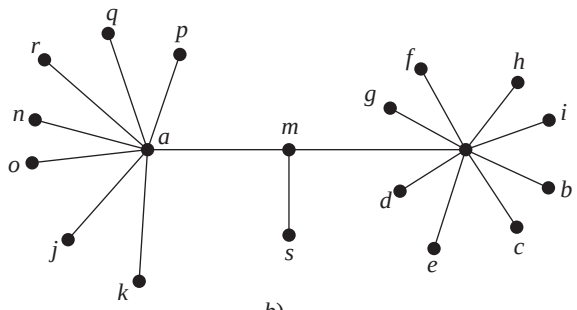
Luego, P se sustituye por una ruta con aristas (a, m) y (b, m). Para todo nodo v en V_a, V_b y V_m, v se une a a, b y m, respectivamente. Para todos los nodos en $V_{am}(V_{bm})$, una de dos: todos se unen a $a(b)$ o todos se unen a m. Así, hay cuatro 3-estrellas y se demostrará que una es la que se busca. Ahora se muestra un caso típico en la figura 9-50. En esta figura se observan las cuatro 3-estrellas.

Luego se intenta encontrar una fórmula para el costo de ruta del árbol de expansión de costo de ruta mínimo. Para cada nodo v en este árbol, hay una ruta que va de v a un nodo en P. Sea $dt(v, P)$ la longitud de la ruta que va de v a ese punto en P. En el costo total de ruta, esta longitud de ruta debe contarse por lo menos $2n/3$ veces porque P es un (1/3)-separador. Para toda arista de P, debido a que hay por lo menos $n/3$ nodos a cada lado de éste, la arista se cuenta por lo menos $(n/3)(2n/3)$ veces en el costo de ruta. Sea $w(P)$ la longitud total de la ruta de P. Entonces se tiene

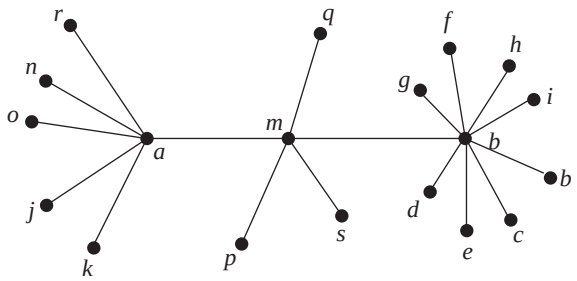
FIGURA 9-50 Un árbol y sus cuatro 3-estrellas.



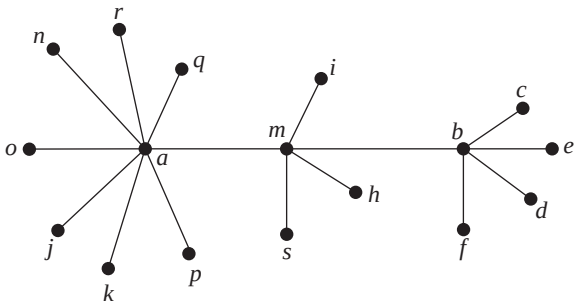
a)



b)

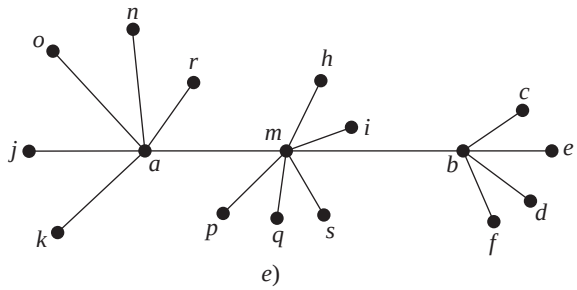


c)



d)

FIGURA 9-50 (*Continuación.*)



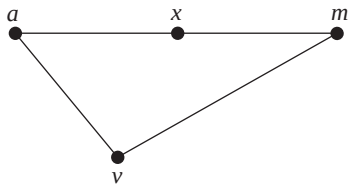
e)

$$mrcst(G) \geq (2n/3) \sum_{v \notin P} dt(v, P) + (2/9)n^2 w(P). \quad \textbf{(9-16)}$$

A continuación se analizará el costo de ruta de la 3-estrella que acaba de construirse.

1. Para toda arista (v, a), (v, b) y (v, m) en el costo de ruta final, la arista en cuestión se cuenta $(n - 1)$ veces. Para cada nodo v en $V_a \cup V_b \cup V_m$, lo único que puede decirse es que el peso de la arista no es mayor que $dt(v, P)$.
2. Para todo nodo en $V_{am} \cup V_{bm}$, puede decirse más. Observe que hay cuatro 3-estrellas posibles. Para todos los nodos en V_{am}, todos están unidos al nodo a o al nodo m. No se sabe cuál de los dos casos es mejor. Considere la figura 9-51. Suponga que originalmente v estaba unido a un nodo x entre a y m. Entonces se tienen las dos desigualdades siguientes:

FIGURA 9-51 Conexión de un nodo v a a o a m.



$$w(v, a) \leq w(v, x) + w(a, x). \quad \textbf{(9-17)}$$

$$w(v, m) \leq w(v, x) + w(x, m). \quad \textbf{(9-18)}$$

Debido a que $w(v,x) \leq dt(v,P)$ y $w(a,x) + w(x,m) \leq dt(a,m)$, al sumar (9-17) y (9-18), se tiene

$$w(v, a) + w(v, m) \le 2dt(v, P) + dt(a, m)$$

o $$\frac{w(v, a) + w(v, m)}{2} \le dt(v, P) + \frac{dt(a, m)}{2}. \quad \textbf{(9-19)}$$

Además, se tiene lo siguiente

$$\sum_v w(v, a) + \sum_v w(v, m) = \sum_v (w(v, a) + w(v, m)). \quad \textbf{(9-20)}$$

Así, una de dos, $\sum_v w(v, a)$ o $\sum_v w(v, m)$ es menor que $\sum_v \frac{w(v, a) + w(v, m)}{2}$.

En V_{am} o V_{bm} hay cuando mucho $n/6$ nodos. Sea v un nodo en $V_{am} \cup V_{bm}$. El razonamiento anterior conduce a la conclusión de que en una de las cuatro 3-estrellas, la suma de pesos de las aristas de todos los nodos en $V_{am} \cup V_{bm}$ no es mayor que lo siguiente:

$$\frac{n}{6} \sum_v \left(dt(v, P) + \frac{1}{2}(dt(a, m) + dt(b, m)) \right).$$

3. Para toda arista (a, m) o (b, m), éste se cuenta no más de $(n/2)(n/2) = (n^2/4)$ veces en el costo de ruta final.

En resumen, una de las cuatro 3-estrellas, su costo de ruta, denotado por *RC*(3-estrellas), satisface la siguiente ecuación:

$$RC(\text{3-estrella}, G)$$
$$\le (n-1) \sum_v dt(v, P) + \left(\frac{n}{12}\right)(dt(a, m) + dt(b, m)) + \left(\frac{1}{4}\right) n^2 w(P)$$
$$\le n \sum_v dt(v, P) + \left(\frac{1}{3}\right) n^2 w(P).$$

Al usar (9-16), se tiene

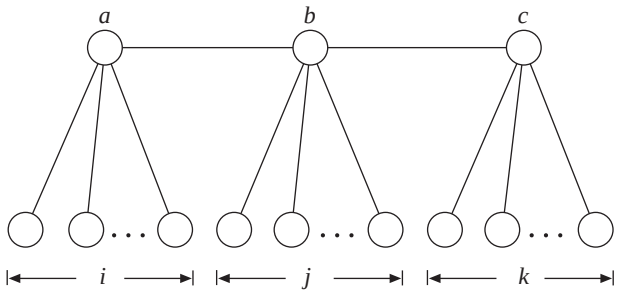
$$RC(\text{3-estrella}, G) \le \frac{3}{2}\, mrcst(G).$$

Así, se ha demostrado que para la tasa de error = 1/2 existe una 3-estrella cuyo costo de ruta no es mayor que 3/2 del costo de ruta de un árbol de expansión de costo de ruta mínimo.

Como se había mencionado, acaba de demostrarse la existencia de una 3-estrella. El problema es cómo encontrarla, lo cual se explicará a continuación.

Se supondrá que de entre todos los nodos se escogen tres, denotados por a, b y c. Estos tres nodos serán los únicos nodos internos de nuestra 3-estrella. Imagine que también se cuenta con una tríada (i, j, k), donde i, j y k son todos enteros positivos y $i + j + k = n - 3$. Esto significa que los nodos i se unirán a a, los nodos j se unirán a b y los nodos k se unirán a c, lo cual se muestra en la figura 9-52.

FIGURA 9-52 Una 3-estrella con $(i + j + k)$ nodos hoja.



La pregunta es: ¿cuáles nodos i deben unirse a a, cuáles nodos j deben unirse a b y cuáles nodos k deben unirse a c? Parar responder esta pregunta, es necesario resolver un problema de apareamiento en una gráfica bipartita. En una gráfica bipartita hay dos conjuntos de nodos, que se denotan por X y Y. En nuestro caso, la gráfica de entrada original es $G(V, E)$. Para una gráfica bipartita, $X = V - \{a, b, c\}$ y Y contiene i copias del nodo a, j copias del nodo b y k copias del nodo c. El peso de la arista entre un nodo en X y un nodo en Y puede encontrarse en la matriz de distancias de entrada original. El problema de apareamiento perfecto mínimo se resuelve en esta gráfica bipartita. Si un nodo v en X se aparea con un nodo y en Y, v se une a u en la 3-estrella. Esta 3-estrella es la mejor 3-estrella una vez que se han determinado los nodos a, b y c y los enteros i, j y k. El costo total de ruta de esta 3-estrella puede encontrarse más bien fácilmente. La arista entre un nodo hoja y un nodo interno se cuenta exactamente $(n - 1)$ veces. La arista entre a y b se cuenta $i(j + k)$ veces y la arista entre b y c se cuenta $k(i + j)$ veces. Cada arista entre un nodo hoja y un nodo interno se cuenta $(n - 1)$ veces.

Nuestro algoritmo para encontrar una 3-estrella cuyo costo total de ruta no sea superior a 3/2 de un árbol de ruta con costo mínimo se proporciona a continuación:

Algoritmo 9-10 □ Un algoritmo que produce una 3-estrella cuyo costo total de ruta no es superior a 3/2 del costo de un árbol de ruta de costo mínimo

Input: Una gráfica completa $G(V, E)$ con pesos en todas las aristas y donde todos los pesos satisfacen la desigualdad del triángulo.

Output: Una 3-estrella cuyo costo total de ruta no es superior a 3/2 del costo de un árbol de ruta de costo mínimo de G.

Hacer $RC = \infty$.
Para toda (a, b, c) donde a, b y c se seleccionan de V, do
  Para toda (i, j, k) donde $i + j + k = n - 3$ e i, j y k son todos enteros positivos, do
  Hacer $X = V - \{a, b, c\}$ y Y que contiene i copias de a, j copias de b y k copias de c.
  Realizar un apareamiento bipartita mínimo perfecto entre X y Y.
  Si un nodo v se une a a, b o c en el apareamiento, unir este nodo a a, b o c, respectivamente. Así se obtiene una 3-estrella.
  Calcular el costo total Z de esta 3-estrella.
  Si Z es menor que RC, sea $RC = Z$.
  Hacer 3-estrella obtenida se revela como la mejor 3-estrella.

La complejidad temporal del algoritmo anterior se analiza como sigue: hay $O(n^3)$ formas posibles de seleccionar a, b y c. Hay $O(n^2)$ formas posibles de seleccionar i, j y k. El problema del apareamiento bipartita mínimo perfecto puede resolverse en tiempo $O(n^3)$. En consecuencia, la complejidad temporal para encontrar una 3-estrella idónea es $O(n^8)$.

Una k-estrella deseable puede encontrarse de manera semejante. Aquí no se abordarán los detalles de esta afirmación. En general, se requiere tiempo $O(n^{2k+2})$ para encontrar una k-estrella cuyo costo total de ruta no sea mayor que $(k + 3)/(k + 1)$ del costo de ruta del árbol de costo de ruta mínimo. Observe que $k = \left\lceil \frac{2}{E} - 1 \right\rceil$. Así, no importa cuán pequeño sea E, siempre se cuenta con un algoritmo de aproximación polinomial que produce una k-estrella cuya tasa de error es menor que su cota de error. En resumen, para el problema del árbol de ruta de costo mínimo existe un PTAS.

9-12 LOS NPO-COMPLETOS

En el capítulo 8 se analizó el concepto del NP-completo. Si un problema de decisión es NP-completo, entonces es bastante improbable que pueda resolverse en tiempo polinomial.

En este apartado se estudiarán los problemas NPO-completos. Se demostrará que si un problema de optimización es NPO-completo, entonces es bastante improbable que exista un algoritmo de aproximación polinomial que produzca una solución aproximada de este problema con una tasa de error constante. En otras palabras, la clase de los problemas NPO-completos denota una clase de problemas de optimización que es improbable que cuente con buenos algoritmos de aproximación.

Primero, se define el concepto de problemas NPO. Recuerde que el conjunto NP está integrado por problemas de decisión. *El conjunto de problemas NPO (non-deterministic polynomial optimization) está integrado por problemas de optimización.* Debido a que ahora se está trabajando con problemas de optimización, se tiene interés en soluciones factibles, que son óptimas. Si un problema de optimización está en NPO, entonces una solución factible de este problema puede encontrarse en dos etapas; a saber, conjeturar y comprobar. Suponga que este problema tiene una solución factible. Entonces la etapa de conjetura siempre la localiza correctamente. Por supuesto, también puede suponerse que la comprobación requiere un número polinomial de pasos. Es importante observar que la etapa de conjetura sólo produce una solución factible, que no necesariamente es óptima.

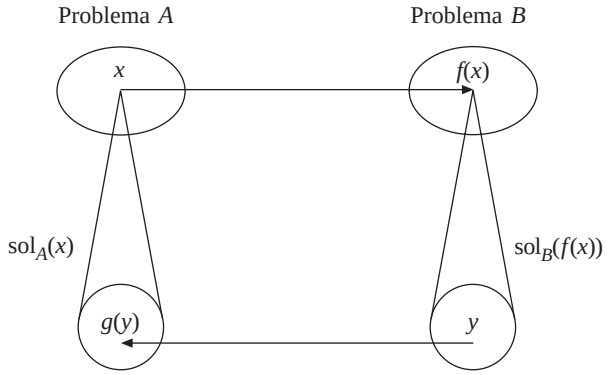
Por ejemplo, puede verse que el problema del agente viajero es un problema NPO porque se trata de un problema de optimización y siempre es posible conjeturar un recorrido y obtener su longitud. Por supuesto, no hay garantía de que este resultado obtenido por conjetura sea el óptimo.

A continuación se definirá qué es un NPO-completo. Primero, se define una reducción, denominada reducción estricta, que se ilustra en la figura 9-53 y se define como sigue:

Dados los problemas NPO A y B, (f, g) es una reducción estricta de A a B si

1. *Para todo caso x en A, $f(x)$ es un caso en B.*
2. *Para toda solución factible y de $f(x)$ en B, $g(y)$ es una solución factible en A.*
3. *El error absoluto de $g(y)$ con respecto a la x óptima es menor o igual al error absoluto de y con respecto al óptimo de $f(x)$ en B. Es decir,* $|g(y) - OPT_A(x)| \leq |y - OPT_B(f(x))|$.

FIGURA 9-53 El concepto de reducción estricta.



Una vez que se han definido el problema NPO y la reducción estricta, ahora es posible definir los problemas NPO-completos. *Un problema NPO es NPO-completo si todos los problemas NPO pueden reducirse estrictamente a él.*

Puede verse que si un problema *A* se reduce estrictamente al problema *B*, y *B* tiene un algoritmo de aproximación cuyo error relativo al óptimo es menor que ε, entonces puede usarse para construir un algoritmo de aproximación de *A* cuyo error se garantiza ser menor que ε. En consecuencia, si un problema de optimización *A* se reduce estrictamente a un problema de optimización *B*, entonces el hecho de que el problema *B* tenga un algoritmo de aproximación con una razón de desempeño constante implica que el problema *A* también tiene un algoritmo de aproximación con una razón de desempeño constante. Por lo tanto, si un problema NPO-completo tiene cualquier algoritmo de aproximación con una razón de desempeño constante, entonces todos los problemas NPO poseen algoritmos de aproximación con una razón de desempeño constante. Así, se ha establecido la dificultad del problema *A*.

En el capítulo 3 se demostró el primer problema NP-completo formal: el problema de satisfactibilidad. En este apartado se presentará el problema de satisfactibilidad ponderado y también se describirá por qué el problema de satisfactibilidad ponderado es NPO-completo.

El problema de satisfactibilidad ponderado mínimo (máximo) se define como sigue: se tiene una fórmula booleana (BF), donde cada variable x_i está asociada con un peso positivo $w(x_i)$. Nuestro objetivo es encontrar una asignación verdadera a las variables que cumpla BF y minimice (maximice)

$$\sum_{\substack{x_i \text{ es} \\ \text{verdadera}}} w(x_i).$$

Se omitirá la demostración de que el problema de satisfactibilidad ponderado es NPO-completo. Así como se hizo en el capítulo 8, aquí se proporcionarán muchos ejemplos para mostrar la forma en que los problemas de optimización pueden reducirse al problema ponderado de satisfactibilidad.

Por ejemplo, considere el problema de encontrar el máximo de dos números a_1 y a_2. En la reducción estricta para este caso, $f(x)$ es como sigue. Para a_1 (y a_2), se asocia $x_1(x_2)$ con a_1 (a_2) y $f(x)$ transforma a_1 (y a_2) en el caso de un problema del problema máximo ponderado de satisfactibilidad como sigue:

BF: $(x_1 \vee x_2) \wedge (-x_1 \vee -x_2)$.

Sea $w(x_i) = a_i$ para $i = 1, 2$. Es fácil ver que para toda asignación verdadera que cumpla BF, exactamente una variable es verdadera. Si $a_1(a_2)$ es máximo, entonces la asignación verdadera asigna el valor de verdad verdadero a $x_1(x_2)$.

La función $g(x)$ se define como:

$$g(x_1, -x_2) = a_1$$
$$g(-x_1, x_2) = a_2.$$

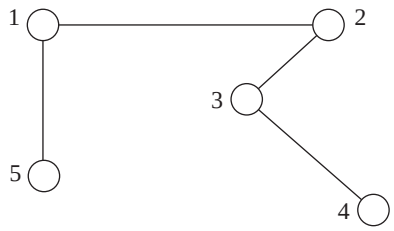
Puede verse que $f(x)$ y $g(x)$ constituyen una reducción estricta entre el problema de maximización y el problema de satisfactibilidad ponderado máximo.

Ejemplo: el problema del conjunto independiente máximo

Este problema se define como sigue: dada una gráfica $G = (V, E)$, encontrar $V' \subseteq V$ tal que V' sea independiente y $|V'|$ se maximice. Se dice que un conjunto S de vértices es "independiente" si para todo $u, v \in S$, la arista $(u, v) \notin E$.

Considere la gráfica en la figura 9-54.

FIGURA 9-54 Ejemplo de problema del conjunto independiente máximo.



En la reducción estricta para este caso, $f(x)$ es como sigue:

1. Para todo vértice i, que tiene vecinos $j_1, j_2, \ldots, j_k$, f los transforma en una cláusula $x_i \rightarrow (-x_{j_1} \& -x_{j_2} \& \cdots \& -x_{j_k})$.
2. $w(x_i) = 1$ para toda i.

El caso que se muestra en la figura 9-54 se transforma en

$$\begin{aligned}\phi = {} & (x_1 \rightarrow (-x_5 \& -x_2)) \\ & \& (x_2 \rightarrow (-x_1 \& -x_3)) \\ & \& (x_3 \rightarrow (-x_2 \& -x_4)) \\ & \& (x_4 \rightarrow (-x_3)) \\ & \& (x_5 \rightarrow (-x_1)).\end{aligned}$$

La función g se define como: $g(X) = \{i \mid x_i \in X$ y x_i es verdadera$\}$.

Puede verse que para cada asignación verdadera, X que satisface ϕ, $g(X)$ es un conjunto independiente. Así, se ha establecido la reducción estricta entre el conjunto independiente máximo y el problema ponderado máximo de satisfactibilidad.

Ejemplos que demuestran los NPO-completos

Con base en los ejemplos anteriores, el lector debería convencerse de que siempre es posible transformar exitosamente cualquier problema NPO en el problema de satisfactibilidad ponderado (WSAT, del inglés: weighted satisfiability). Esencialmente, lo anterior significa que el problema de satisfactibilidad ponderado es NPO-completo; es decir, que todos los problemas NPO pueden reducirse estrictamente a éste.

A continuación se demostrará que algunos problemas son NPO-completos. Nos gustaría recordar al lector que cuando se quiere demostrar que un problema A es NPO-completo, suele hacerse en dos pasos:

1. Primero se demuestra que A es un problema NPO.
2. Luego se demuestra que algún problema NPO-completo se reduce estrictamente a A.

Se observa que hasta el momento sólo se tiene un problema NPO-completo: el problema de satisfactibilidad ponderado. A fin de tener más problemas NPO-completos, es necesario empezar a partir del problema de satisfactibilidad ponderado. Es decir, debe intentar demostrarse que el problema de satisfactibilidad ponderado se reduce estrictamente al problema motivo de interés.

Ejemplo: programación entera cero-uno

El problema de programación entera cero-uno se define como sigue: dados una matriz entera $A\ \ m * n$, un m-vector entero b y un n-vector positivo c, se encontrará un n-vector cero-uno x que satisface $Ax \geq b$ y minimiza cx.

Ciertamente, resulta fácil ver que éste es un problema NPO. Para cualquier solución conjeturada no determinista X, en tiempo polinomial puede comprobarse si $Ax \geq b$, y la medida cx también puede calcularse en tiempo polinomial.

Antes de demostrar que el problema de programación entera cero/uno es NPO-completo, primero se presentará el problema de 3-satisfactibilidad máxima, o mínima (W3SAT). Este problema se define como sigue: dada una fórmula booleana ϕ, que consta de una conjunción de oraciones donde cada una de éstas contiene exactamente tres literales y un peso entero positivo $w(x_i)$, $i = 1, 2, 3$, asociado con x_i, encontrar una asignación $\tau(x_i)$ que satisface ϕ, en caso de existir, que maximice (minimice)

$$\sum_{\substack{\tau(x_i)\ es \\ verdadera}} w(x_i).$$

Sean $w(x_1) = 3$, $w(x_2) = 5$ y $w(x_3) = 2$. Considere la fórmula

$$(-x_1 \vee -x_2 \vee -x_3)$$
$$\&\,(-x_1 \vee -x_2 \vee -x_3)$$
$$\&\,(-x_1 \vee -x_2 \vee x_3)$$
$$\&\,(x_1 \vee x_2 \vee x_3).$$

Hay cuatro asignaciones que cumplen la fórmula anterior:

$(x_1, -x_2, x_3)$ con peso 5
$(x_1, -x_2, -x_3)$ con peso 3
$(-x_1, x_2, x_3)$ con peso 7
y $(-x_1, x_2, -x_3)$ con peso 5.

La asignación $(-x_1, x_2, x_3)((x_1, -x_2, -x_3))$ es la solución del problema W3SAT máximo (mínimo). Los problemas W3SAT máximo y mínimo son NPO-completos. Para demostrar que el problema de programación entera cero/uno es NPO-completo, se demostrará que el problema W3SAT mínimo puede reducirse estrictamente a él.

La función *f* que transforma el W3SAT mínimo en el problema de programación entera cero/uno es como sigue:

1. Cada variable *x* en el W3SAT mínimo corresponde a una variable *x* en el problema de programación entera cero/uno.
2. El valor 1 de la variable representa "TRUE" y el valor 0 representa "FALSE".
3. Cada cláusula en W3SAT se transforma en una desigualdad. El operador "or" se transforma en "+" y $-x$ se transforma en $1 - x$. Debido a que toda la cláusula debe ser verdadera, se tiene que debe ser "≥ 1".
4. Sea $w(x_i)$, $i = 1, 2, 3$ el peso de la variable x_i. Entonces se minimiza la siguiente función:

$$\sum_{i=1}^{3} w(x_i)x_i.$$

El problema W3SAT mínimo $(x_1 \vee x_2 \vee x_3) \& (-x_1 \vee -x_2 \vee x_3)$ con $w(x_1) = 3$, $w(x_2) = 5$, $w(x_3) = 5$ se transforma en el siguiente problema de programación entera cero/uno:

$$\begin{aligned} &\text{Minimizar} \quad 3x_1 + 5x_2 + 5x_3, \text{ donde } x_i = 0, 1, \\ &\text{sujeta a} \quad x_1 + x_2 + x_3 \geq 1,\ 1 - x_1 + 1 - x_2 + x_3 \geq 1. \end{aligned}$$

La asignación de verdad $(-x_1, -x_2, x_3)$ que es una solución del problema W3SAT mínimo corresponde al vector (0, 0, 1), que es una solución del problema de programación entera cero/uno. Así, se ha demostrado que el problema 3-satisfactibilidad ponderado se traduce estrictamente al problema de programación entera cero/uno. Por lo tanto, el problema de programación entera cero/uno es NPO-completo.

Ejemplo: los problemas de los ciclos hamiltonianos más corto y más largo en gráficas

Dada una gráfica, un ciclo hamiltoniano es un ciclo que atraviesa exactamente una vez todos los vértices de la gráfica. El problema del ciclo hamiltoniano más largo (más corto) en gráficas se define como sigue: dada una gráfica, encontrar el ciclo hamiltoniano más largo (más corto) de la gráfica. Se ha demostrado que ambos problemas son NP-difíciles. El problema del ciclo hamiltoniano más corto también se denomina problema del agente viajero. A continuación se demostrará que ambos problemas son NPO-completos. Resulta más bien evidente que ambos son problemas NPO. Lo NPO-completo puede establecerse al demostrar que el problema W3SAT máximo, o mínimo, se reduce estrictamente a aquél.

Primero, dado un caso ϕ del problema W3SAT máximo, se demostrará lo siguiente:

1. Existe una función f que transforma ϕ en una gráfica $f(\phi)$ de modo que el W3SAT es satisfactible si y sólo si $f(\phi)$ tiene un ciclo hamiltoniano.
2. Existe una función g que transforma un ciclo hamiltoniano C en la gráfica $f(\phi)$ en una asignación de verdad $g(C)$ para ϕ.
3. El costo de C es el mismo que el costo de $g(C)$.

La última propiedad asegura que el error absoluto de $g(C)$ con respecto a la solución óptima de ϕ es igual al error absoluto de C con respecto a la solución óptima de $f(\phi)$. Es decir, $|g(C) - OPT(\phi)| = |C - OPT(f(\phi))|$. Por lo tanto, (f, g) es una reducción estricta.

A continuación se ilustra la transformación de un caso del problema W3SAT en una gráfica.

1. Para toda cláusula C_i hay una componente C_i, como se muestra en la figura 9-55. Las aristas e_1, e_2 y e_3 corresponden a las tres literales en la cláusula C_i. Por ejemplo, si C_i contiene $-x_1$, x_2 y $-x_3$, entonces e_1, e_2 y e_3 corresponden a $-x_1$, x_2 y $-x_3$, respectivamente. Observe que una C_i-componente posee la propiedad de que, para cualquier ruta hamiltoniana de $c_{i,1}$ a $c_{i,4}$, por lo menos una de las aristas e_1, e_2, e_3 no es atravesada, lo cual se observa en la figura 9-56. Después se explicará el significado de arista no atravesada.

FIGURA 9-55 Una C-componente correspondiente a una 3-cláusula.

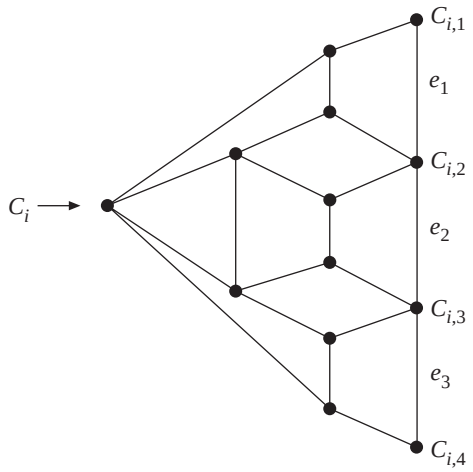
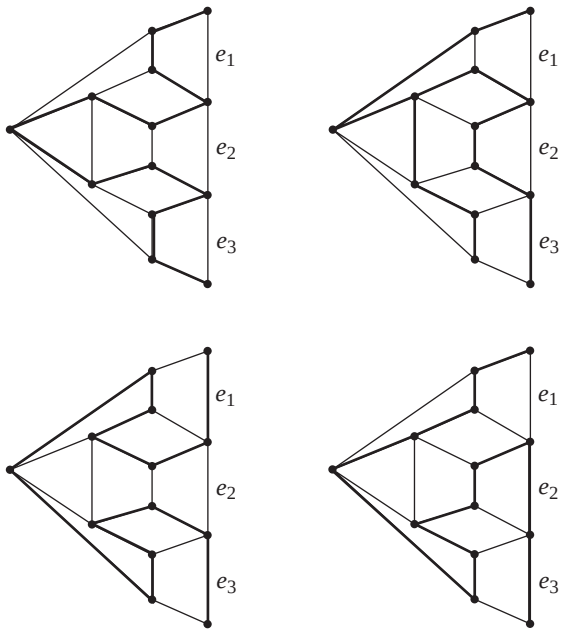
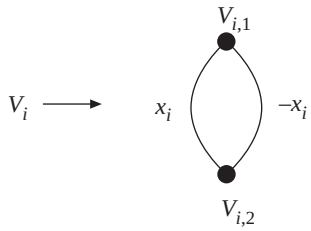


FIGURA 9-56 *C*-componente donde por lo menos no se atraviesa una de las aristas e_1, e_2 y e_3.



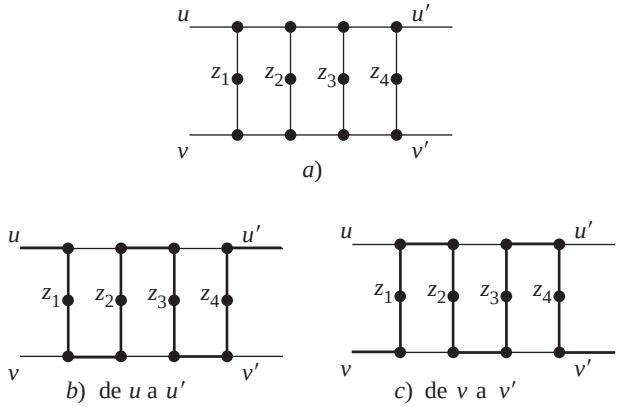
2. Para toda variable V_i, se tiene una V_i-componente, que se muestra en la figura 9-57. Para una ruta hamiltoniana, es posible atravesar ya sea x_i o $-x_i$.

FIGURA 9-57 Componente correspondiente a una variable.



3. Una componente *C* se une a una componente *V* mediante un dispositivo *H*, lo cual se muestra en la figura 9-58. Si una gráfica $G = (V, E)$ contiene un dispositivo $H = (W, F)$ de la figura 9-58*a*) de modo que ningún vértice en $V - W$ sea adyacente a ningún vértice en $W - \{u, v, u', v'\}$, entonces puede verse que cualquier ciclo hamiltoniano en *G* debe atravesar *H* como el que se muestra en la figura 9-58*b*) o como el que se muestra en la figura 9-58*c*).

FIGURA 9-58 Dispositivos.



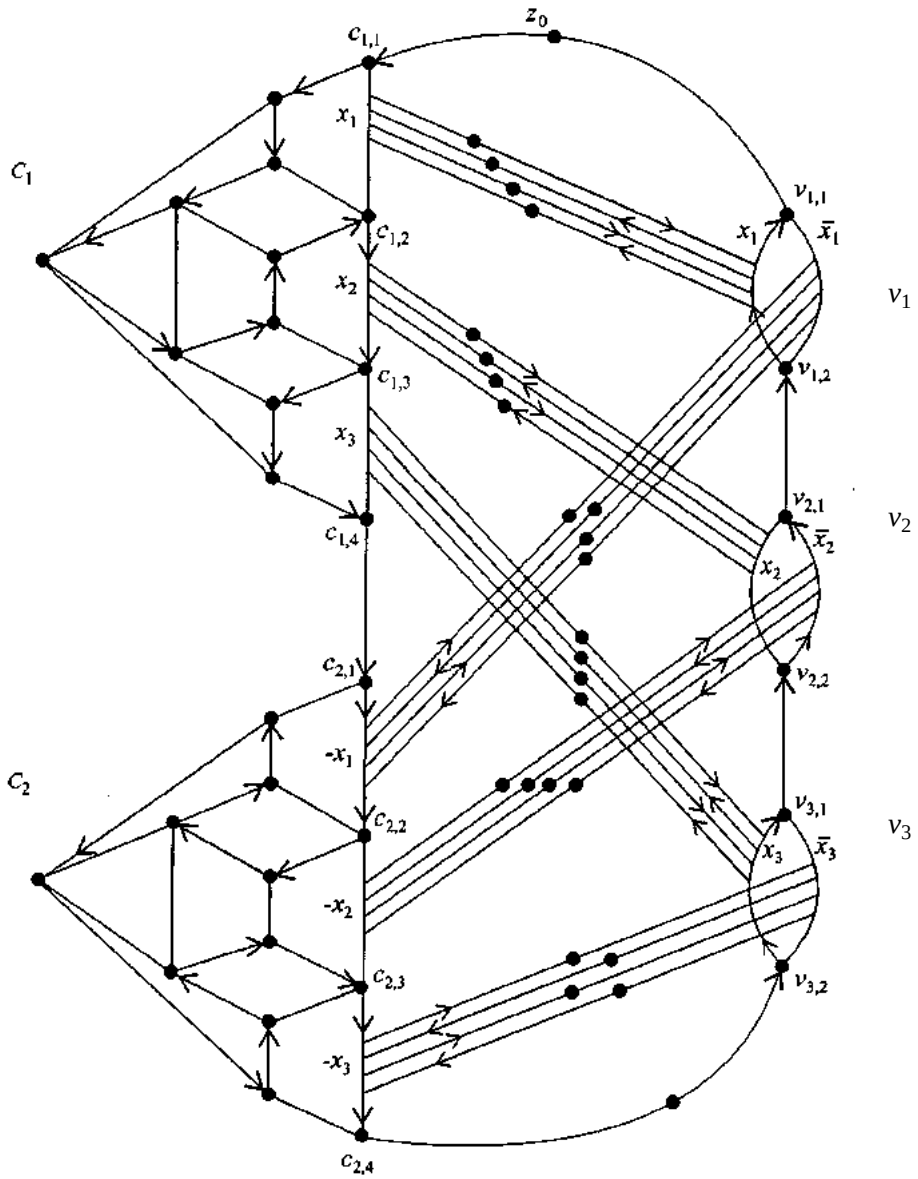
4. Para una fórmula booleana con oraciones $C_1, C_2, \ldots, C_m$, las componentes C correspondientes a estas oraciones estarán conectadas en serie con las aristas $(c_{i,4}, c_{i+1,1})$ para $i = 1, 2, \ldots, m-1$.
5. Para una fórmula booleana con variables $x_1, x_2, \ldots, x_n$, sus componentes V correspondientes están conectadas en serie con las aristas $(v_{i,2}, v_{i+1,1})$, $i = 1, 2, \ldots, n-1$.
6. Hay dos aristas especiales; a saber $(c_{1,1}, v_{1,1})$ y $(c_{m,4}, v_{n,2})$.
7. Si $x(-x)$ aparece en una cláusula C, entonces la arista izquierda (derecha) correspondiente a $x(-x)$ en la componente V de la variable x está unido a la arista correspondiente a la variable x en la componente C correspondiente a la cláusula C.

En la figura 9-59 se muestra un ejemplo de ciclo hamiltoniano. Observe que cualquier ciclo hamiltoniano debe contener la ruta de $v_{1,1}$ a $c_{1,1}$. Desde $c_{1,1}$ tiene dos alternativas, procediendo directamente hacia $c_{1,2}$ o en sentido inverso. En nuestro ejemplo, no atraviesa la arista correspondiente a x_1 en la componente C_1. A través de todo el ciclo, x_1 y x_3 en la componente C_1 y $-x_2$ en la componente C_2 no se atraviesan. Esto corresponde a la asignación $(x_1, -x_2, x_3)$.

En general, se tiene la regla siguiente: en un ciclo hamiltoniano, si no se atraviesa la arista en una componente C correspondiente a $x_i(-x_i)$, entonces asignar a $x_i(-x_i)$ el valor verdadero en la asignación correspondiente.

Según la regla anterior, si la arista correspondiente a x_1 en la cláusula C_1 no se atraviesa a un ciclo, entonces en la asignación se hace verdadera a x_1. Quizá el lector se pregunte: ¿acaso se atravesará la arista correspondiente a $-x_1$ en la cláusula C_2? Éste debe ser el caso. En caso contrario, hay una inconsistencia y no puede haber una asignación.

FIGURA 9-59 $(x_1 \vee x_2 \vee x_3)\,\mathrm{y}\,(-x_1 \vee -x_2 \vee -x_3)$ con la asignación $(-x_1, -x_2, x_3)$.



Que esto sea así puede verse al examinar la conexión de $-x_1$ en la componente V_1. La forma de realizar la conexión obliga a que se atraviese la arista correspondiente a $-x_1$. Puede demostrarse que para cualquier ciclo hamiltoniano en G, si la arista correspondiente a $x_i(-x_i)$ en alguna componente C_j no se atraviesa, entonces la aris-

ta correspondiente a $-x_i(x_i)$ se atraviesa en alguna componente C_k. Esto significa que todo ciclo hamiltoniano en G corresponde a una asignación para ϕ.

Aún queda por asignar pesos a las aristas de G. Se asigna $w(x_i)$ a la arista izquierda de la componente V_i, para $i = 1, 2, \ldots, n$, y 0 a todas las demás aristas. Puede demostrarse que la gráfica G tiene un ciclo hamiltoniano con peso W si y sólo si la fórmula booleana ϕ tiene una asignación verdadera con peso W.

Se ha demostrado que existe una reducción estricta desde un problema W3SAT máximo o mínimo, a un problema del ciclo hamiltoniano más largo o más corto, en gráficas. Debido a la NPO-completez de ambos problemas W3SAT, máximo y mínimo, ahora se ha establecido la NPO-completez de ambos problemas del ciclo hamiltoniano, más largo y más corto, en gráficas.

9-13 NOTAS Y REFERENCIAS

La NP-dificultad del problema euclidiano del agente viajero fue demostrada por Papadimitrou (1977). Rosenkrantz, Stearns y Lewis (1977) demostraron que el problema euclidiano del agente viajero puede aproximarse con longitudes menores que dos veces la longitud de un recorrido más corto. Christofides (1976) relacionó las árboles de expansión mínimos, los apareamientos y el problema del agente viajero para obtener un algoritmo de aproximación para el problema euclidiano del agente viajero, donde la longitud es menor que 3/2 de la longitud óptima. Las variantes del problema de cuello de botella que se analizaron en los apartados 9-3 y 9-4 aparecieron originalmente en Parker y Rardin (1984) y Hochbaum y Shmoys (1986), respectivamente. El algoritmo de aproximación de empaque en contenedores apareció en Johnson (1973). El algoritmo de aproximación para el problema rectilíneo de m-centros fue proporcionado por Ko, Lee y Chan (1990).

La terminología de los NPO-completos apareció por primera vez en Ausiello, Crescenzi y Protasi (1995), aunque este concepto fue introducido originalmente por Orponen y Mannila (1987). En Orponen y Mannila (1987) se demostró que todos los problemas siguientes: de programación entera cero-uno, del agente viajero y de 3-satisfactibilidad ponderado máximo son NPO-completos. La demostración del NPO-completo del problema de la ruta hamiltoniana más larga puede consultarse en Wu, Chang y Lee (1998).

El hecho de que del problema del agente viajero difícilmente puede tener algoritmos de aproximación aceptables también fue demostrado por Sahni y González (1976). Ellos demostraron que si el problema del agente viajero posee alguna aproximación de razón constante, entonces NP = P. Observe que esto es válido para casos generales. Se trata de buenos algoritmos de aproximación para casos generales.

El método ingenioso para encontrar PTAS para el problema 0/1 de la mochila fue descubierto por Ibarra y Kim (1975). Baker desarrolló PTAS para varios problemas en gráficas planas (Baker, 1994). El problema del agente viajero tiene PTAS si se restringe a ser plano (Grigni, Koutsoupias y Papadimitrou, 1995), o euclidiano (Arora, 1996). El esquema PTAS para el problema del árbol de expansión de ruta de costo mínimo fue proporcionado en Wu, Lancia, Bafna, Chao, Ravi y Tang (2000). Otro esquema PTAS interesante puede encontrarse en Wang y Gusfield (1997).

Hay otra clase de problemas, denotados por problemas MAX SNP-completos, propuestos por Papadimitrou y Yannakakis (1991). Esta clase de problemas puede expresarse en una forma sintáctica estricta (Fagin, 1974), y hay algoritmos de aproximación de razón constante para ellos. Se demostró que cualquier problema en esta clase puede tener un PTAS si y sólo si NP = P. Se demostró que a esta clase pertenecen varios problemas, como el MAX 3SAT, el de la métrica del agente viajero, el del árbol de expansión de hojas máximo y el del árbol etiquetado no ordenado. Para conocer más detalles, consulte las obras de Papadimitrou y Yannakakis (1991); Papadimitrou y Yannakakis (1992); Galbiati, Maffioli y Morzenti (1994), y Zhang y Jiang (1994).

9-14 BIBLIOGRAFÍA ADICIONAL

El algoritmo de aproximación sobre el ordenamiento por transposiciones puede consultarse en Bafna y Pevzner (1998). Para el problema de la alineación de secuencias múltiples, consulte Gusfield (1997). Tanto Horowitz y Sahni (1978) como Papapdimitrou y Steiglitz (1982) tienen excelentes análisis sobre los algoritmos de aproximación. Un repaso de este tema puede encontrarse en Garey y Johnson (1976). En años recientes se han publicado mucho algoritmos de aproximación. Varios de ellos están relacionados con los NP-completos. Recomendamos los siguientes artículos para profundizar sus conocimientos: Alon y Azar (1989); Baker y Coffman Jr. (1982); Bruno, Coffman y Sethi (1974); Cornuejols, Fisher y Nemhauser (1977); Friesen y Kuhl (1988); Hall y Hochbaum (1986); Hochbaum y Maass (1987); Hsu (1984); Johnson (1974); Johnson, Demars, Ullman, Garey y Graham (1974); Krarup y Pruzan (1986); Langston (1982); Larson (1984); Mehlhorn (1988); Moran (1981); Murgolo (1987); Nishiseki, Asano y Watanabe (1983); Raghavan (1988); Sahni (1977); Sahni y González (1976); Tarhio y Ukkonen (1988); Vaidya (1988), y Wu, Widmayer y Wong (1986).

Resultados más recientes pueden consultarse en Agarwal y Procopiuc (2000), Akustu y Halldorsson (1994); Akutsu y Miyano (1997); Akutsu, Arimura y Shimozono (2000); Aldous (1989); Amir y Farach (1995); Amir y Keselman (1997); Amir y Landau (1991); Arkin, Chiang, Mitchell, Skiena y Yang (1999); Armen y Stein (1994); Armen y Stein (1995); Armen y Stein (1996); Armen y Stein (1998); Arora, Lund, Motwani,

Sudan y Szegedy (1998); Arratia, Goldstein y Gordon (1989); Arratia, Martin, Reinert y Waterman (1996); Arya, Mount, Netanyahu, Silverman y Wu (1998); Avrim, Jiang, Li, Tromp y Yannakakis (1991); Baeza-Yates y Perleberg (1992); Baeza-Yates y Navarro (1999); Bafna y Pevzner (1996); Bafna, Berman y Fujito (1999); Bafna, Lawler y Pevzner (1997); Berman, Hannenhalli y Karpinki (2001); Blum (1994); Blum, Jiang, Li, Tromp y Yannakakis (1994); Bonizzoni, Vedova y Mauri (2001); Breen, Waterman y Zhang (1985); Breslauer, Jiang y Jiang (1997); Bridson y Tang (2001); Cary (2001); Chang y Lampe (1992); Chang y Lawler (1994); Chen y Miranda (2001); Chen (1975); Cristos y Drago (1998); Chu y La (2001); Clarkson (1994); Cobbs (1995); Czumaj, Gasieniec, Piotrow y Rytter (1994); Dinitz y Nutov (1999); Drake y Hougardy (2003); Esko (1990); Even, Naor y Zosin (2000); Feige y Krauthgamer (2002); Frieze y Kannan (1991); Galbiati, Maffioli y Morrzenti (1994); Galil y Park (1990); Goemans y Williamson (1995); Gonzalo (2001); Gusfield (1994); Hochbaum y Shmoys (1987); Ivanov (1984); Jain y Vazirani (2001); Jiang y Li (1995); Jiang, Kearney y Li (1998); Jiang, Kearney y Li (2001); Jian, Wang y Lawler (1996); Jonathan (1989); Jorma y Ukkonen (1988); Kannan y Warnow (1995); Karkkainen, Navarro y Ukkonen (2000); Kececioglu (1991); Kececioglu y Myers (1995a); Kececioglu y Myers (1995b); Kececioglu y Sankoff (1993); Kececioglu y Sankoff (1995); Kleinberg y Tardos (2002); Kolliopoulos y Stein (2002); Kortsarz y Peleg (1995); Krumke, Marathe y Ravi (2001); Landau y Schmidt (1993); Landau y Vishkin (1989); Leighton y Rao (1999); Maniezzo (1998); Mauri, Pavesi y Piccolboni (1999); Myers (1994); Myers y Miller (1989); Parida, Floratos y Rigoutsos (1999); Pe'er y Shamir (2000); Pevzner y Waterman (1995); Promel y Steger (2000); Raghavachari y Veerasamy (1999); Slavik (1997); Srinivasan (1999); Srinivasan y Teo (2001); Stewart (1999); Sweedyk (1995); Atrio y Ukkonen (1986); Atrio y Ukkonen (1988); Tong y Wong (2002); Trevisan (2001); Turner (1989); Ukkonen (1985a); Ukkonen (1985b); Ukkonen (1990); Ukkonen (1992); Vazirani (2001); Vyugin y V'yugin (2002); Wang y Gusfield (1997); Wang y Jiang (1994); Wang, Jiang y Gusfield (2000); Wang, Jiang y Lawler (1996); Wang, Zhang, Jeong y Shasha (1994); Waterman y Vingron (1994); Wright (1994); Wu y Manber (1992); Wu y Myers (1996), y Zelikovsky (1993).

Ejercicios

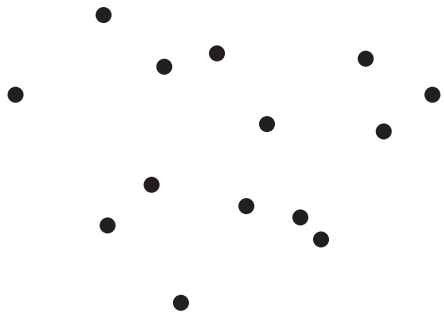
9.1 Un algoritmo de aproximación en línea sencillo para resolver el problema de empaque en contenedores consiste en colocar un objeto en el *i*-ésimo contenedor en tanto tenga disponibilidad y en el contenedor $(i + 1)$-ésimo en caso contrario. Este algoritmo se denomina algoritmo de siguiente ajuste o next-fit (algoritmo NF). Demuestre que el número de contenedores resultante de este algoritmo no es superior a dos veces el número de contenedores necesarios en una solución óptima.

9.2 Demuestre que si la secuencia de artículos a considerar es $1/2, 1/2n, 1/2, 1/2n, \ldots, 1/2$ (en total $4n - 1$ artículos), entonces el algoritmo NF en efecto utilizará casi el doble del número de contenedores que realmente se requieren.

9.3 Demuestre que no existe ningún algoritmo de aproximación polinomial para el problema del agente viajero de modo que el error provocado por el algoritmo de aproximación esté acotado a menos de $\varepsilon \cdot TSP$, donde ε es cualquier constante y TSP denota una solución óptima.

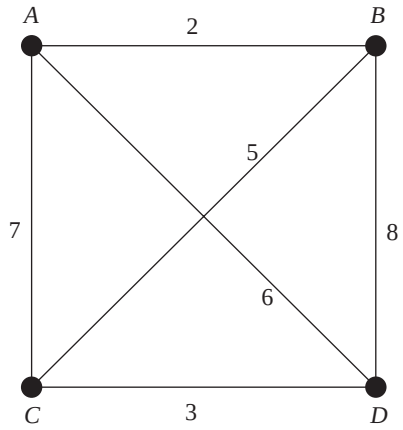
(**Sugerencia:** Demuestre que el problema del ciclo hamiltoniano puede reducirse a este problema.)

9.4 Aplique un algoritmo del casco convexo para encontrar un casco convexo aproximado del siguiente conjunto de puntos.

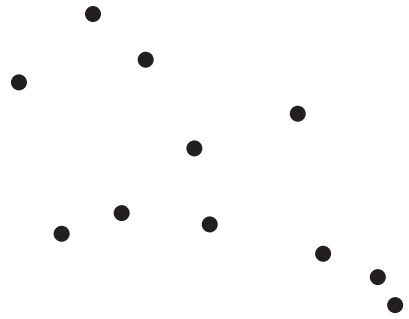


9.5 Sea un conjunto de puntos distribuidos densamente en un circuito. Aplique el algoritmo euclidiano de aproximación del agente viajero para encontrar un recorrido aproximado para este conjunto de puntos. ¿Este resultado es el óptimo?

9.6 Considere los cuatro puntos de un cuadrado como se muestra a continuación. Resuelva el problema de cuello de botella del agente viajero aproximadamente con el algoritmo presentado en este capítulo. ¿El resultado obtenido también es óptimo?



9.7 Aplique el algoritmo de aproximación para el problema rectilíneo de *m*-centros para encontrar una solución aproximada del problema rectilíneo de 2 centros para el siguiente conjunto de puntos.



9.8 Considere el problema de optimización de cuello de botella. Se tiene un conjunto de puntos en el plano y se solicita encontrar *k* puntos de modo que entre estos *k* puntos se maximice la distancia más corta. Wang y Kuo (1988) demostraron que este problema es NP-completo. Intente desarrollar un algoritmo de aproximación para resolver este problema.

9.9 Lea la sección 12.3 del libro de Horowitz y Sahni (1978) sobre algoritmos de aproximación para programar tareas independientes. Aplique la regla del tiempo de procesamiento más largo (LTP) al siguiente problema de calendarización: hay tres procesadores y siete tareas, donde los tiempos de las tareas son $(t_1, t_2, t_3, t_4, t_5, t_6, t_7) = (14, 12, 11, 9, 8, 7, 1)$.

9.10 Escriba un programa para implementar el algoritmo de aproximación para el problema del agente viajero. También escriba un programa para implementar el algoritmo branch-and-bound para encontrar una solución óptima del problema del agente viajero. Compare los resultados. Saque su conclusión. ¿Merece la pena usar este algoritmo de aproximación?

capítulo

10

Análisis amortizado

En este capítulo el interés lo constituye el análisis de la complejidad temporal de una secuencia de operaciones. Suponga que se considera una secuencia de operaciones OP_1, OP_2, …, OP_m y que se desea determinar el mayor tiempo posible que puede consumir esta secuencia de operaciones. Una reacción inmediata podría ser investigar la complejidad temporal del peor caso para toda OP_i. Sin embargo, no siempre es correcto afirmar que el mayor tiempo posible de OP_1, OP_2, …, OP_m es la suma de t_i, donde t_i es el tiempo utilizado por OP_i en el peor caso. La razón es más bien sencilla. No es de esperar que los peores casos ocurran tan a menudo.

Con más precisión, imagine que cada mes es posible ahorrar una vez dinero en el banco e ir a una tienda departamental para gastar el dinero mientras en el banco haya un fondo de ahorro. Ahora puede verse de inmediato que lo que se hace en un mes depende de lo que se hizo en los meses previos. Si en octubre se gasta dinero espléndidamente, debe tratar de ser muy austero en meses anteriores a dicho mes. Además, después de octubre, ya que se han gastado todos los ahorros, es necesario volver a ahorrar. Pasarán muchos meses antes de poder gastar espléndidamente de nuevo.

La situación anterior ilustra una cuestión. Las operaciones pueden estar interrelacionadas y no puede suponerse que son independientes entre sí.

La palabra “amortizado” significa que en el presente se ahorra para gastar en el futuro. ¿Por qué el término “amortizado” está relacionado con el análisis de algoritmos? Como se demostrará después, la palabra “amortizado” implica que en muchos casos las acciones que se realizan sobre nuestra estructura de datos pueden parecer como una pérdida de tiempo en ese momento. Sin embargo, esta acción sobre la estructura de datos puede rendir frutos después. Es decir, después de que se ejecutan estas acciones, más adelante las cosas pueden volverse mucho más fáciles. En la juventud se trabaja arduamente; los frutos se cosechan en la vejez.

10-1 UN EJEMPLO DE LA UTILIZACIÓN DE LA FUNCIÓN POTENCIAL

El potencial es un concepto bastante conocido. Para poder aprovechar la fuerza del agua, es necesario subirla, de modo que este "potencial" sea alto. Sólo cuando la energía potencial es alta puede usarse el agua. El dinero que se ahorra en el banco también puede considerarse potencial. Sólo es posible gastar después de que se ha ahorrado lo suficiente.

En esta sección, se mostrará cómo es posible usar el concepto de función potencial para hacer algo de análisis amortizado. Se considera una secuencia de operaciones OP_1, OP_2, …, OP_m donde cada OP_i consiste en varias extracciones de elementos de una pila y la inserción de un elemento a la misma pila. Se supone que para llevar a cabo cada inserción y cada extracción se requiere un tiempo unitario. Sea t_i el tiempo necesario para ejecutar OP_i. Entonces el tiempo total es

$$T = \sum_{i=1}^{m} t_i$$

y el tiempo promedio por operación es

$$t_{prom} = \sum_{i=1}^{m} t_i / m$$

y la tarea consiste en determinar t_{prom}.

El lector puede darse cuenta de que de ninguna manera es fácil encontrar t_{prom}. Antes de presentar un método que usa el concepto de función potencial para encontrar t_{prom}, se proporcionarán algunos ejemplos:

i	1	2	3	4	5	6	7	8
S_1	1 inserción	1 inserción	2 extracciones 1 inserción	1 inserción	1 inserción	1 inserción	2 extracciones 1 inserción	1 extracción 1 inserción
t_i	1	1	3	1	1	1	3	2

$t_{prom} = (1 + 1 + 3 + 1 + 1 + 1 + 3 + 2)/8 = 13/8 = 1.625$

i	1	2	3	4	5	6	7	8
S_2	1 inserción	1 extracción 1 inserción	1 inserción	1 inserción	1 inserción	1 inserción	5 extracciones 1 inserción	1 inserción
t_i	1	2	1	1	1	1	6	1

$t_{prom} = (1 + 2 + 1 + 1 + 1 + 1 + 6 + 1)/8 = 14/8 = 1.75$

i	1	2	3	4	5	6	7	8
S_3	1 inserción	1 inserción	1 inserción	1 inserción	4 extracciones 1 inserción	1 inserción	1 inserción	1 extracción 1 inserción
t_i	1	1	1	1	5	1	1	2

$t_{prom} = (1 + 1 + 1 + 1 + 5 + 1 + 1 + 2)/8 = 13/8 = 1.625$

El lector debe haber observado que $t_{prom} \leq 2$ para cada caso. De hecho, es difícil encontrar t_{prom}. No obstante, es posible demostrar que t_{prom} no tiene una cota superior igual a 2. Esto puede hacerse porque después de cada OP_i, no sólo se ha completado una operación, sino que se ha modificado el contenido de la pila. El número de elementos de la pila ha aumentado o disminuido. Si ha aumentado, entonces también ha aumentado la capacidad de poder ejecutar muchas extracciones. En cierto sentido, cuando a la pila se agrega un elemento, se incrementa el potencial, facilitando así la ejecución de muchas extracciones. Por otra parte, cuando de la pila se retira un elemento, el potencial disminuye de alguna manera.

Sea $a_i = t_i + \phi_i - \phi_{i-1}$, donde ϕ_i indica la función potencial de la pila después de OP_i. Así, $\phi_i - \phi_{i-1}$ es el cambio del potencial. Entonces,

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} t_i + \sum_{i=1}^{m} (\phi_1 - \phi_{i-1})$$
$$= \sum_{i=1}^{m} t_i + \phi_m - \phi_0.$$

Si $\phi_m - \phi_0 \geq 0$, entonces $\sum_{i=1}^{m} a_i$ representa una cota superior de $\sum_{i=1}^{m} t_i$. Es decir,

$$\sum_{i=1}^{m} t_i/m \leq \sum_{i=1}^{m} a_i/m \qquad \textbf{(1)}$$

Ahora, ϕ_i se define como *el número de elementos en la pila* inmediatamente después de la i-ésima operación. Resulta fácil ver que para este caso se tiene $\phi_m - \phi_0 \geq 0$. Se supone que antes de ejecutar OP_i en el pila hay k elementos y que OP_i consta de n extracciones y una inserción. Así,

$$\phi_{i-1} = k$$
$$\phi_i = k - n + 1$$

$$
\begin{aligned}
t_i &= n + 1 \\
a_i &= t_i + \phi_i - \phi_{i-1} \\
&= n + 1 + (k - n + 1) - k \\
&= 2.
\end{aligned}
$$

Con base en la ecuación anterior, puede demostrarse fácilmente que

$$\sum_{i=1}^{m} a_i/m = 2.$$

En consecuencia, por (1), $t_{prom} = \sum_{i=1}^{m} t_i/m \leq \sum_{i=1}^{m} a_i/m = 2$. Es decir, $t_{prom} \leq 2$.

En realidad, en este caso no es necesario usar la función potencial para obtener la cota superior igual a 2. Naturalmente que no. Para *m* operaciones, cuando mucho hay *m* inserciones y *m* extracciones porque el número de extracciones no puede ser mayor que el de inserciones. Así, a lo sumo puede haber 2*m* acciones y el tiempo promedio jamás puede superar a 2.

Aunque, para este caso sencillo, no es necesario usar la función potencial, en algunos otros casos sí es necesario usarla, como se verá en el siguiente apartado.

Finalmente, queremos recordar al lector que $a_i \leq 2$ no debe interpretarse como $t_i \leq 2$. El valor de t_i puede ser muy alto, mucho mayor que 2. No obstante, si t_i es alto en el presente, en el futuro será pequeño debido al hecho de que t_i sea alto ahora significa que el número de elementos en la pila se reducirá enormemente en el futuro. Para plantearlo de otra forma, si es alto, entonces el potencial disminuirá desmesuradamente. Esto es semejante al fenómeno natural. Si se usan grandes cantidades de agua, entonces el potencial de los recursos disminuye; no habrá agua para usarse en el futuro.

10-2 UN ANÁLISIS AMORTIZADO DE HEAPS SESGADOS

En un heap sesgado (skew heap), su operación básica se denomina "fusión" (meld). Considere la figura 10-1. Para fusionar los dos heaps sesgados, primero se fusionan las rutas derechas de estos dos heaps en una y luego las partes izquierdas se unen a los nodos en la ruta fusionada, lo cual se muestra en la figura 10-2. Después de este paso, se intercambian los hijos izquierdo y derecho de todo nodo en la ruta fusionada, excepto el más bajo.

En la figura 10-3 se muestran los efectos del intercambio. Como puede verse, el heap sesgado resultante tiende a poseer una ruta derecha más corta, aunque no hay ninguna garantía de que la ruta derecha sea la más corta. No obstante, observe que es muy fácil ejecutar la operación fusión, y produce un buen desempeño en el sentido amortizado.

FIGURA 10-1 Dos heaps sesgados.

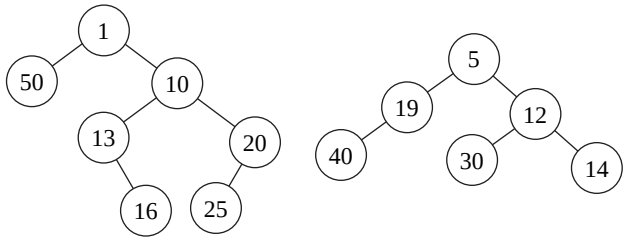


FIGURA 10-2 Fusión de los dos heaps sesgados de la figura 10-1.

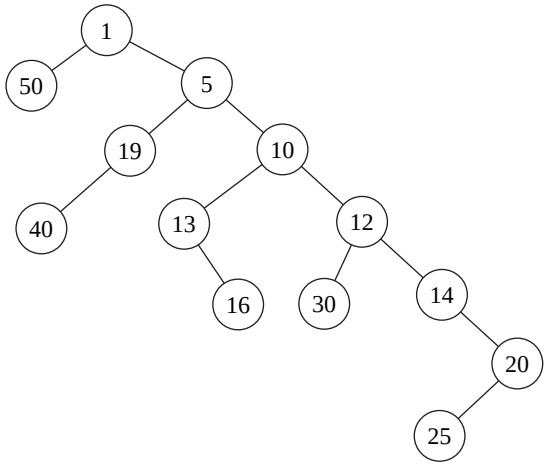
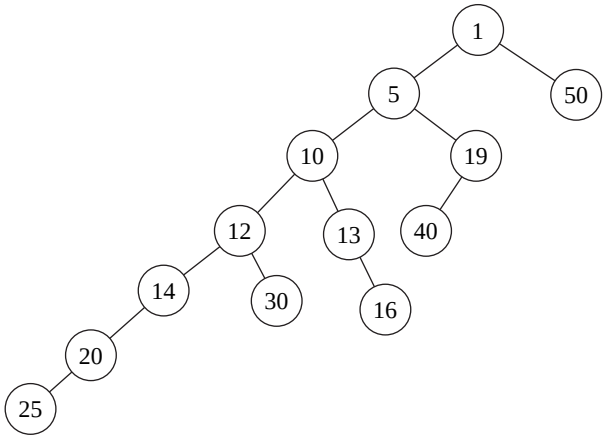


FIGURA 10-3 Intercambio de las rutas izquierda y derecha del heap sesgado de la figura 10-2.



Una estructura datos, como un heap sesgado, requiere muchas operaciones. En realidad, en un heap sesgado pueden ejecutarse las siguientes operaciones:

1. *find*-min(h): encontrar el mínimo de un heap sesgado h.
2. insert(x, h): inserta x en un heap sesgado h.
3. *delete*-min(h): elimina el mínimo de un heap sesgado h.
4. meld(h_1, h_2): fusiona dos heaps sesgados h_1 y h_2.

Las tres primeras operaciones pueden implementarse por fusión. Por ejemplo, para insertar x, x se considera como un heap sesgado con un solo elemento y estos dos heaps sesgados se fusionan. De manera semejante, cuando se elimina el mínimo, que siempre ocupa la raíz de los heaps sesgados, el heap sesgado se puede separar efectivamente en dos heaps sesgados. Luego, estos dos heaps sesgados de reciente creación se fusionan.

Debido a que todas las operaciones se basan en operaciones de fusión, una secuencia de operaciones que consta de eliminación, inserción y fusión puede considerarse como una secuencia de operaciones de fusión. En consecuencia, sólo es necesario analizar las operaciones de fusión. Antes de llevar a cabo el análisis, se observa que la operación de fusión posee una propiedad interesante. *Si se gasta mucho tiempo en la fusión, entonces debido al efecto de intercambio (swap), la ruta derecha se vuelve muy corta. Así, la operación de fusión no consumirá demasiado tiempo otra vez; el potencial ha desaparecido simplemente.*

Para efectuar un análisis amortizado, de nuevo se hace

$$a_i = t_i + \phi_i - \phi_{i-1}$$

donde t_i es el gasto de tiempo para OP_i, ϕ_i y ϕ_{i-1} son los potenciales antes y después de $OP_{i,}$ respectivamente. De nuevo, se tiene

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} t_i + \phi_m - \phi_0.$$

Si $\phi_m - \phi_0 \geq 0$, entonces $\sum_{i=1}^{m} a_i$ sirve como una cota superior del tiempo real gastado por m operaciones cualesquiera.

Ahora el problema consiste en definir una buena función potencial. Sea $wt(x)$ del nodo x el número de descendientes del nodo x, incluyendo a x. Sea un nodo pesado x un nodo no raíz con $wt(x) > wt(p(x))/2$, donde $p(x)$ es el nodo padre de x. Un nodo es un nodo ligero si no es un nodo pesado.

Considere el heap sesgado en la figura 10-2. Los pesos de los nodos se muestran en la tabla 10-1.

TABLA 10-1 Los nodos pesados de la figura 10-2.

Nodo	Peso	Pesado/ligero
1	13	
5	11	Pesado
10	8	Pesado
12	5	Pesado
13	2	Ligero
14	3	Pesado
16	1	Ligero
19	2	Ligero
20	2	Pesado
25	1	Ligero
30	1	Ligero
40	1	Ligero
50	1	Ligero

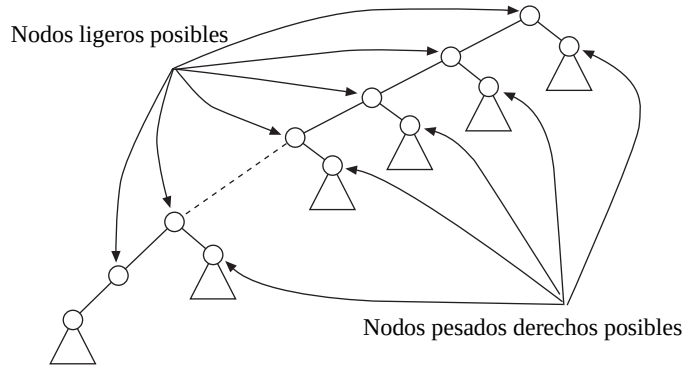
Se observa que cada nodo hoja debe ser un nodo ligero. También, si un nodo es ligero, no tiene muchos descendientes, al menos comparado con su hermano. Por ejemplo, los nodos 13 y 19 en la figura 10-2 son ambos nodos ligeros. Sus nodos hermanos son pesados. El árbol, o el heap sesgado, se encuentra orientado hacia la parte con muchos nodos pesados. Observe que en el heap sesgado de la figura 10-2, todos los nodos de la ruta derecha son pesados; se trata de una ruta larga. Por otra parte, considere la figura 10-3. La ruta derecha del heap sesgado de la figura 10-3 no es un nodo pesado; es una ruta corta.

Si un nodo es el hijo derecho (izquierdo) de otro nodo, entonces el nodo se denomina nodo derecho (izquierdo). Un nodo pesado derecho (izquierdo) es un nodo derecho (izquierdo) que es pesado. *El número de nodos pesados derechos de un heap sesgado puede usarse como función potencial*. Así, para el heap sesgado de la figura 10-2, el potencial es 5 y para el heap sesgado de la figura 10-3, el potencial es 0.

A continuación se investigará una cuestión muy importante. ¿Cuáles son los potenciales antes y después de la fusión?, o bien, ¿cuál es la diferencia entre el número de nodos pesados derechos antes y después de la fusión?

Primero se observa que para ser un nodo pesado, éste debe tener más descendientes que su nodo hermano. Así, puede verse fácilmente que *de los hijos de cualquier nodo, cuando mucho uno es pesado*. Con base en esta afirmación, en la figura 10-4 se ve que el número de nodos derechos pesados asociados con la ruta izquierda está relacionado con el número de nodos ligeros asociados con la ruta izquierda.

FIGURA 10-4 Nodos ligeros posibles y nodos pesados posibles.



Debido a que sólo el nodo hermano de un nodo ligero puede ser un nodo pesado asociado con la ruta izquierda, *el número de nodos pesados derechos asociados a la ruta izquierda siempre es menor o igual al número de nodos ligeros en la ruta izquierda de un heap sesgado*. Nuestro problema consiste en estimar el número de nodos ligeros que hay en una ruta.

Para estimar el número de nodos ligeros que hay en una ruta, se observa que *si f es un nodo ligero, entonces el peso de x debe ser menor* o *igual al semipeso de p*(x)*, donde p*(x) *denota el padre de x*. Si y es un descendiente de x y y es un nodo ligero, entonces el peso de y debe ser menor o igual a un cuarto del peso de $p(x)$. En otras palabras, considere la secuencia de nodos ligeros en una ruta que va de un nodo x a y. Puede verse fácilmente que el peso de estos nodos ligeros decrece rápidamente y además, *para cualquier ruta de un heap sesgado, el número de nodos ligeros en esta ruta debe ser menor que* $\lfloor \log_2 n \rfloor$. Esto significa que el número de nodos pesados derechos asociados con la ruta izquierda debe ser menor que $\lfloor \log_2 n \rfloor$.

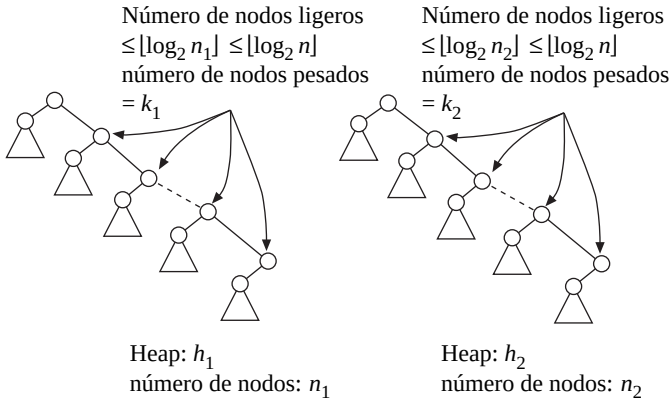
Observe que

$$a_i = t_i + \phi_i - \phi_{i-1}$$

donde t_i es el tiempo gastado por OP_i y ϕ_i denota el potencial después de la ejecución de OP_i.

El valor de t_i está relacionado con la longitud de la ruta derecha. Considere la figura 10-5. Sea $K_1(K_2)$ el número de nodos pesados de la ruta derecha en $h_1(h_2)$. Sea $n_1(n_2)$ el número de nodos en $h_1(h_2)$. Sea $L_1(L_2)$ igual al número de nodos ligeros más el número de nodos pesados en la ruta derecha en $h_1(h_2)$. Y t_i se determina por el número de nodos en la ruta fusionada, que es igual a $2 + L_1 + L_2$. (El "2" se refiere a las raíces de h_1 y h_2.)

FIGURA 10-5 Dos heaps sesgados.



$$
\begin{aligned}
\text{Así que } t_i &= 2 + L_1 + L_2 \\
&= 2 + \text{número de nodos ligeros de la ruta derecha de } h_1 \\
&\quad + \text{número de nodos pesados de la ruta derecha de } h_1 \\
&\quad + \text{número de nodos ligeros de la ruta derecha de } h_2 \\
&\quad + \text{número de nodos pesados de la ruta derecha de } h_2 \\
&\leq 2 + \lfloor \log_2 n_1 \rfloor + K_1 + \lfloor \log_2 n_2 \rfloor + K_2 \\
&\leq 2 + 2\lfloor \log_2 n \rfloor + K_1 + K_2
\end{aligned}
$$

donde $n = n_1 + n_2$.

A continuación se calcula el valor de a_i. *Se observa que después de la fusión,* $K_1 + K_2$ *nodos pesados derechos en la ruta derecha desaparecen y se crean menos de* $\lfloor \log_2 n \rfloor$ *nodos pesados derechos en la ruta derecha.* En consecuencia,

$$\phi_i - \phi_{i-1} \leq \lfloor \log_2 n \rfloor - K_1 - K_2, \text{ para } i = 1, 2, \ldots, m.$$

De esta manera, se tiene

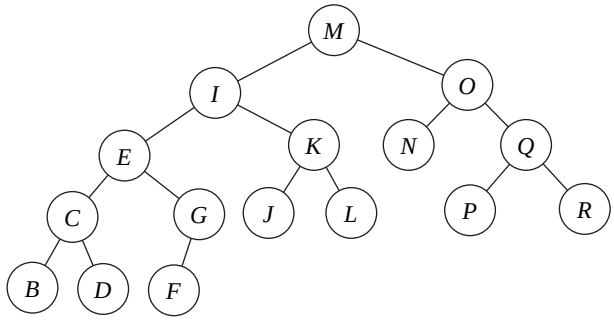
$$
\begin{aligned}
a_i &= t_i + \phi_i - \phi_{i-1} \\
&\leq 2 + 2\lfloor \log_2 n \rfloor + K_1 + K_2 + \lfloor \log_2 n \rfloor - K_1 - K_2 \\
&= 2 + 3\lfloor \log_2 n \rfloor.
\end{aligned}
$$

Por lo tanto, $a_i = O(\log_2 n)$ y $\Sigma t_i/m = O(\log_2 n)$.

10-3 ANÁLISIS AMORTIZADO DE ÁRBOLES-AVL

En este apartado se llevará a cabo un análisis amortizado de árboles-AVL. *Un árbol binario es un árbol-AVL si las alturas de los subárboles de cada nodo difieren por, cuando mucho, uno.* En la figura 10-6 se muestra un árbol-AVL. Como puede observarse, este árbol está equilibrado en el sentido de que para cada nodo, la altura de sus dos subárboles difiere a lo sumo por uno.*

FIGURA 10-6 Un árbol-AVL.



Para un subárbol T con v con su raíz, la altura de T, denotada por H(T), se define como la longitud de la ruta más larga de v a una hoja. Sea $L(v)(R(v))$ el subárbol izquierdo (derecho) del árbol con raíz v. Entonces para todo nodo v, su equilibrio de altura (height balance) $hb(v)$ se define como

$$hb(v) = H(R(v)) - H(L(v)).$$

Para un árbol-AVL, $hb(v)$ es igual a 0, $+1$ o -1.

Para el árbol-AVL de la figura 10-6, algunos de sus equilibrios de altura son como sigue:

$$hb(M) = -1 \qquad hb(I) = -1$$
$$hb(E) = 0 \qquad hb(C) = 0$$
$$hb(B) = 0 \qquad hb(O) = +1.$$

Debido a que un árbol-AVL es un árbol binario, a medida que se agrega un nuevo artículo al árbol, este nuevo dato se convierte en un nodo hoja. Este nodo hoja inducirá una ruta particular a lo largo de la cual todos los factores de equilibrio deben modificarse. Considere la figura 10-7, que muestra el árbol-AVL en la figura 10-6 con todos los equilibrios de altura. Suponga que se desea agregar el artículo *A*. El nuevo árbol se muestra en la figura 10-8. Como puede verse, el nuevo árbol ya no es un árbol-AVL,

* AVL fue inventado por Adelson-Velsky y Landis en 1962. *(N. del R.T.)*

por lo que es necesario volver a equilibrarlo. Al comparar la figura 10-8 con la figura 10-7, puede verse que sólo han cambiado los equilibrios de altura a lo largo de la ruta *M, I, E, C, B, A.*

FIGURA 10-7 Un árbol-AVL con equilibrios de altura etiquetados.

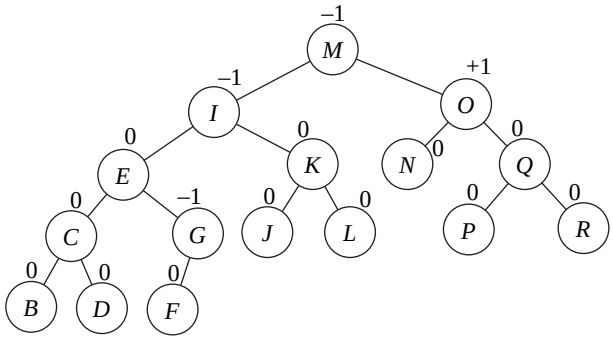
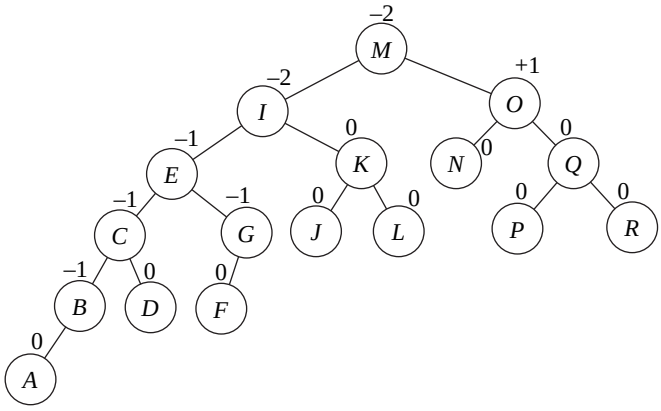


FIGURA 10-8 El nuevo árbol con *A* agregado.



Sea V_k el nuevo nodo hoja agregado a un árbol-AVL. Sea V_0, V_1, …, V_{k-1}, V_k la ruta de reciente creación donde V_0 es la raíz del árbol. Sea i la mínima y tal que $hb(V_i) = hb(V_{i+1}) = \cdots = hb(V_{k-1}) = 0$ antes de la inserción. El nodo V_{i-1} se denomina nodo crítico de la ruta si $i \geq 1$ y V_i, …, V_{k-1} se denomina ruta crítica inducida por V_k.

Considere la figura 10-7. Si el nodo que acaba de agregarse se une a *B*, entonces la ruta crítica es *E, C, B.* Al comparar la figura 10-7 con la figura 10-8, se observa que todos los factores de equilibrio de *E, C* y *B* han cambiado de 0 a −1. Suponga que se

agrega un nuevo nodo a, por ejemplo, la derecha del nodo *R*. Entonces los factores de equilibrio de *Q* y *R* cambian de 0 a +1.

Sea T_0 un árbol-AVL vacío. Se considera una secuencia de *m* inserciones en T_0. Sea X_1 el número total de factores de equilibrio que cambian de 0 a +1 o –1 durante estas *m* operaciones. Nuestro problema consiste en encontrar X_1. El siguiente análisis amortizado muestra que X_1 es menor o iguala a 2.618*m*, donde *m* es el número de elementos de datos en el árbol-AVL.

Sea L_i la longitud de la ruta crítica implicada en la *i*-ésima inserción. Entonces

$$X_1 = \sum_{i=1}^{m} L_i.$$

A continuación se demostrará que cuando se inserta un nuevo artículo, posiblemente haya tres casos.

Caso 1: No hay necesidad de volver a equilibrar. La altura del árbol no crece y sólo se requiere una "absorción" para formar un nuevo árbol.

Caso 2: Es necesario volver a equilibrar. Para crear un nuevo árbol equilibrado se requieren rotaciones simples o dobles.

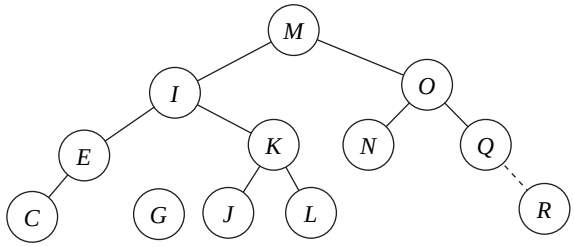
Caso 3: No hay necesidad de volver a equilibrar. Sin embargo, la altura del árbol crece. Basta registrar este aumento en la altura del árbol.

Antes de ilustrar estos tres casos, sea *Val*(*T*) *el número de nodos sin equilibrar que hay en T. Es decir, Val*(*T*) *es igual al número de nodos cuyos factores de equilibrio no son iguales a 0*.

Caso 1: Considere la figura 10-9. La línea punteada indica que se ha insertado un nuevo dato. Debido a que el árbol resultante sigue siendo un árbol-AVL, no es necesario volver a equilibrar. El árbol-AVL anterior simplemente absorbe este nuevo nodo. La altura del árbol tampoco cambia. En este caso, los factores de equilibrio de todos los nodos en la ruta crítica cambian de 0 a –1 o +1. Además, los factores de equilibrio del nodo crítico cambian de +1 o –1 a 0. Así,

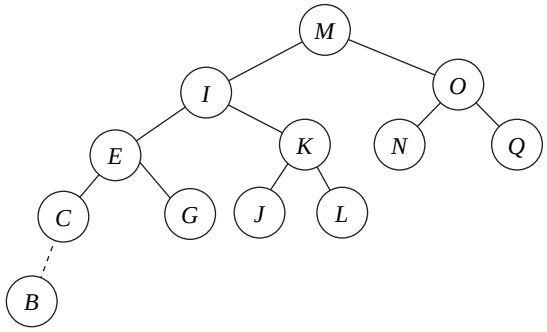
$$Val(T_i) = Val(T_{i-1}) + (L_i - 1).$$

FIGURA 10-9 Caso 1 después de una inserción.



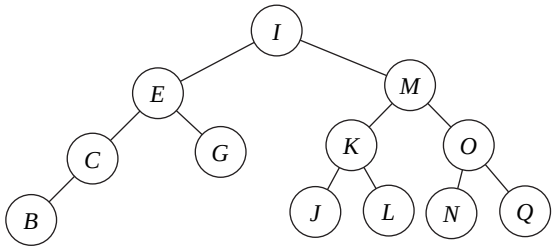
Caso 2: Considere la figura 10-10. En este caso, el árbol requiere un nuevo equilibrio.

FIGURA 10-10 Caso 2 después de una inserción.



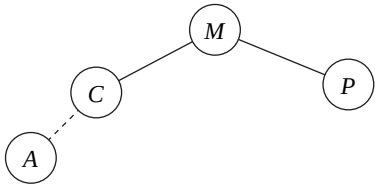
Se requieren ciertos tipos de rotaciones dobles y simples. Para el caso que se muestra en la figura 10-10, el árbol equilibrado se muestra en la figura 10-11. Es fácil ver que el factor de equilibrio del nodo crítico *M* ha cambiado de -1 a 0. Sin embargo, es necesario cambiar los factores de equilibrio de todos los nodos que están en la ruta crítica de 0 a $+1$ o -1, excepto el hijo del nodo crítico en la ruta crítica que también es un nodo equilibrado después de volver a equilibrar. Así, en este caso, $Val(T_i) = Val(T_{i-1}) + (L_i - 2)$.

FIGURA 10-11 El árbol de la figura 10-10 equilibrado.



Caso 3: Considere la figura 10-12. En este caso, no es necesario volver a equilibrar el árbol. Sin embargo, la altura del árbol crece.

FIGURA 10-12 Caso 3 después de una inserción.



Resulta fácil ver que

$$Val(T_i) = Val(T_{i-1}) + L_i.$$

Sean X_2 el número de absorciones necesarias en el caso 1, X_3 el número de rotaciones simples necesarias en el caso 2, X_4 el número de rotaciones dobles necesarias en el caso 2 y X_5 el número de aumentos de altura en el caso 3. Luego, usando la fórmula descrita en los tres casos anteriores, se tiene

$$Val(T_m) = Val(T_0) + \sum_{i=1}^{m} L_i - X_2 - 2(X_3 + X_4).$$

En la fórmula anterior, $-X_2$ y $-2(X_3 + X_4)$ corresponden al caso 1 y al caso 2, respectivamente. $\sum_{i=1}^{m} L_i$ se debe al término L_i en todas las tres fórmulas.

Debido a que $\sum_{i=1}^{m} L_i = X_1$ y $Val(T_0) = 0$, se tiene

$$\begin{aligned} X_1 &= Val(T_m) + X_2 + 2(X_3 + X_4) \\ &= Val(T_m) + (X_2 + X_3 + X_4 + X_5) + (X_3 + X_4 - X_5) \\ &= Val(T_m) + m + (X_3 + X_4 - X_5) \end{aligned}$$

debido a que $X_2 + X_3 + X_4 + X_5 = m$.

Así, resulta evidente que $X_3 + X_4 \leq m$ y $X_5 \geq 0$. En consecuencia,

$$\begin{aligned} X_1 &\leq Val(T_m) + m + m \\ &= Val(T_m) + 2m. \end{aligned}$$

En *The Art of Computer Programming*, vol. 3, Knuth demostró que

$$Val(T_m) \leq 0.618m.$$

En consecuencia, se tiene

$$\begin{aligned} X_1 &\leq Val(T_m) + 2m \\ &= 2.618m. \end{aligned}$$

10-4 ANÁLISIS AMORTIZADO DE MÉTODOS DE BÚSQUEDA SECUENCIAL BASADOS EN HEURÍSTICAS AUTOORGANIZADOS

La búsqueda secuencial es el tipo de búsqueda más sencillo que existe. Sin embargo, aún hay muchos métodos para mejorar el desempeño de la búsqueda secuencial. Uno de ellos es el denominado método autoorganizado.

Suponga que en un dormitorio hay varios estudiantes en exceso populares que constantemente reciben llamadas telefónicas. El operador, si es suficientemente inteligente, escribirá el número de habitación de estos estudiantes en la parte superior de la lista. De esta manera, las llamadas telefónicas serán contestadas rápidamente.

Un método autoorganizado utiliza esta idea. Mueve dinámicamente hacia arriba de la lista los artículos que son requeridos a menudo. Es decir, siempre que se completa una búsqueda, el artículo buscado se desplaza hacia arriba. Hay tres famosos métodos de búsqueda:

1. **Método transpuesto:** Una vez que se encuentra el artículo, se intercambia con el artículo que está enfrente. Así, se mueve un sitio hacia la parte superior de la lista.
2. **Método de mover al frente:** Una vez que se encuentra el artículo, éste se mueve a la parte superior de la lista.
3. **Método de conteo:** Una vez que se encuentra el artículo, se incrementa su conteo y se mueve hacia delante tan poco como se necesite para mantener la lista ordenada en forma decreciente.

Los métodos anteriores suelen denominarse métodos heurísticos. En consecuencia, estos tres métodos también se conocen como heurística transpuesta, heurística de mover al frente y heurística de conteo, respectivamente.

A continuación, estas heurísticas se ilustrarán con varios ejemplos. Siempre se supondrá que inicialmente se empieza con una lista vacía.

1. Heurística transpuesta (consulte la tabla 10-2).

TABLA 10-2 Heurística transpuesta.

Consulta	Secuencia
B	*B*
D	*D B*
A	*D A B*
D	*D A B*
D	*D A B*
C	*D A C B*
A	*A D C B*

2. Heurística de mover al frente (consulte la tabla 10-3).

TABLA 10-3 Heurística de mover al frente.

Consulta	Secuencia
B	*B*
D	*D B*
A	*A D B*
D	*D A B*
D	*D A B*
C	*C D A B*
A	*A C D B*

3. Heurística de conteo (consulte la tabla 10-4).

TABLA 10-4 Heurística de conteo.

Consulta	Secuencia
B	*B*
D	*B D*
A	*B D A*
D	*D B A*
D	*D B A*
A	*D A B*
C	*D A B C*
A	*D A B C*

Se tiene interés en el análisis amortizado de heurísticas distintas. Dada una secuencia de consultas (queries), el costo de esta secuencia *S* de consultas, bajo una heurística *R*, se define como el número total de comparaciones necesarias por esta secuencia si se utiliza la búsqueda secuencial bajo esta heurística *R*. Este costo se denota por $C_R(S)$. Suponga que hay *m* consultas. Sería deseable poder encontrar el costo como una función de *m*. Como se hizo antes, en vez de ello se encontrará una cota superior de este costo. Esta cota superior está relacionada con el costo si los artículos están ordenados de manera idónea y estática para esta secuencia. El costo en que se incurre cuando los artículos están ordenados de manera estática y óptima ahora se identifica por $C_O(S)$ para esta secuencia *S*. Por ejemplo, suponga que hay dos secuencias de consulta. Considere que el número que se está consultando de *B* es mayor que el de *A*. Entonces el ordenamiento estático óptimo para esta secuencia de consulta es *B A*.

En el resto de este apartado se comparará $C_R(S)$ con $C_O(S)$. El hecho de que estos costos pueden compararse se basa en una propiedad muy importante, denominada *propiedad de pares independientes*, que posee alguna heurística. Antes de presentar esta propiedad, las comparaciones entre palabras (intraword) se definirán como comparaciones realizadas entre elementos diferentes. Por lo tanto, las comparaciones entre palabras son comparaciones infructuosas. Las comparaciones entre palabras son comparaciones realizadas entre elementos iguales y por lo tanto son comparaciones exitosas. Ahora ya es posible presentar la propiedad de pares independientes.

Se considerará la heurística de mover al frente. Considere la siguiente secuencia de consulta: *C A C B C A*. Ahora la secuencia aparece como se muestra en la tabla 10-5.

TABLA 10-5 Heurística de mover al frente para la secuencia de consulta: *C A C B C A*.

Consulta	Secuencia	Comparación (*A*, *B*)
C	*C*	
A	*A C*	
C	*C A*	
B	*B C A*	√
C	*C B A*	
A	*A C B*	√

Luego, la atención se centra en la comparación entre *A* y *B*. El análisis anterior muestra que para esta secuencia, el número total de comparaciones realizadas entre *A* y *B* es 2.

Suponga que sólo se consideran las subsecuencias de la secuencia que consta de *A* y *B*. Entonces consulte la tabla 10-6. De nuevo, el número total de comparaciones realizadas entre *A* y *B* es 2, que es lo mismo que en la tabla 10-5.

TABLA 10-6 Heurística de mover al frente para la secuencia de consulta: *A B A*.

Consulta	Secuencia	Comparación (*A*, *B*)
A	*A*	
B	*B A*	√
A	*A B*	√

En otras palabras, para la heurística de mover al frente, el número total de comparaciones realizadas entre cualquier par de elementos *P* y *Q* sólo depende del ordenamiento relativo de las *P* y las *Q* en la secuencia de consulta y es independiente de otros elementos. En otras palabras, para cualquier secuencia *S* y todos los pares de *P* y *Q*, el número de comparaciones entre palabras de *P* y *Q* es exactamente el número de comparaciones entre palabras realizadas para las subsecuencias de *S* que sólo constan de *P* y *Q*. Esta propiedad se denomina propiedad de pares independientes.

Por ejemplo, se considerará nuevamente la secuencia de consulta *C A C B C A*. Hay tres comparaciones entre palabras distintas: (*A, B*), (*A, C*) y (*B, C*). Es posible considerarlas por separado y luego sumarlas, como se muestra en la tabla 10-7.

TABLA 10-7 Número de comparaciones entre palabras usando la heurística de mover al frente.

Consulta	Secuencia	*C*	*A*	*C*	*B*	*C*	*A*	
			0		1		1	(*A*, *B*)
		0	1	1		0	1	(*A*, *C*)
		0		0	1	1		(*B*, *C*)
		0	1	1	2	1	2	

El número total de comparaciones entre palabras es igual a $0 + 1 + 1 + 2 + 1 + 2 = 7$. Es fácil verificar que esto es correcto.

Sea $C_M(S)$ el costo de la heurística de mover al frente. A continuación se demostrará que $C_M(S) \leq 2C_O(S)$. En otras palabras, para cualquier secuencia de consulta, se sabe que para la heurística de mover al frente, el costo jamás puede exceder el doble del costo estático óptimo para esta secuencia de consulta.

Primero se demostrará que $C_M(S) \leq 2C_O(S)$ para *S* que consta de dos elementos distintos. Sea *S* que consta de *a A* y *b B*, donde $a < b$. Entonces, un ordenamiento estático óptimo es *B A*. Si se utiliza la heurística de mover al frente, entonces el número de comparaciones entre palabras es el número de cambios de *B A* a *A B* más el número de cambios de *A B* a *B A*. Así, este número no puede exceder a $2a$. Sean $Intra_M(S)$ e $Intra_O(S)$ el número total de comparaciones entre palabras bajo la heurística de mover al frente y el ordenamiento estático óptimo, respectivamente. Se tiene

$$Intra_M(S) \leq 2Intra_O(S)$$

para cualquier S que conste de dos elementos distintos.

Considere cualquier secuencia S que conste de más de dos elementos. Debido a la propiedad independiente por pares que posee la heurística de mover al frente, fácilmente puede verse que

$$Intra_M(S) \leq 2Intra_O(S)$$

Sean $Intra_M(S)$ e $Intra_O(S)$ los números de comparaciones entre palabras bajo la heurística de mover al frente y bajo el ordenamiento estático óptimo, respectivamente. Resulta evidente que se tiene

$$Inter_M(S) = Inter_O(S)$$

En consecuencia,

$$Inter_M(S) + Intra_M(S) \leq Inter_O(S) + 2Intra_O(S)$$
$$C_M(S) \leq 2C_O(S).$$

Puede plantearse la pregunta de si es posible restringir aún más el coeficiente 2. A continuación se demostrará que esto no es posible.

Considere la secuencia de consulta $S = (A\ B\ C\ D)^m$. El ordenamiento estático óptimo para esta secuencia es $A\ B\ C\ D$. El número total de comparaciones, $C_O(S)$, es

$$\underbrace{1 + 2 + 3 + 4 + 1 + 2 + 3 + 4 + \cdots}_{4m} = 10 \cdot \frac{4m}{4} = 10m.$$

Bajo la heurística de mover al frente el número total de comparaciones, $C_M(S)$ puede encontrarse como sigue:

$C_M(S) = 1$	A	A			
$+2$	B	B	A		
$+3$	C	C	B	A	
$+4$	D	D	C	B	A
$+4$	A	A	D	C	B
$+4$	B	B	A	D	C
$\vdots$	$\vdots$		$\vdots$		
$10 + 4(4(m-1))$					

En consecuencia, $C_M(S) = 10 + 4(4(m-1)) = 10 + 16m - 16 = 16m - 6$.

Si el número de elementos distintos se incrementa de 4 a k, se tiene

$$C_O(S) = \frac{k(k+1)}{2} m = \frac{m(k+1)k}{2}$$

y $$C_M(S) = \frac{k(k+1)}{2} + k(k(m-1)) = \frac{k(k+1)}{2} + k^2(m-1).$$

Así,

$$\frac{C_M(S)}{C_O(S)} = \frac{\dfrac{k(k+1)}{2} + k^2(m-1)}{\dfrac{k(k+1)}{2} \cdot m}$$

$$= \frac{1}{m} + 2\frac{k^2}{k(k+1)} \cdot \frac{m-1}{m} \to 2 \quad \text{cuando } k \to \infty \text{ y } m \to \infty.$$

Esto significa que el factor 2 no puede restringirse más.

El análisis anterior muestra que el costo en el peor caso de la heurística de mover al frente es $2C_O(S)$. El mismo razonamiento puede usarse para demostrar que $C_c(S) \leq 2C_O(S)$ cuando $C_c(S)$ es el costo de la heurística de conteo. Desafortunadamente, la heurística transpuesta no posee la propiedad de pares independientes. En consecuencia, no es posible tener una cota superior semejante para el costo de la heurística transpuesta. Que no hay propiedad de pares independientes puede ilustrarse al considerar nuevamente la misma secuencia de consulta, *C A C B C A*, bajo la heurística transpuesta.

Para la heurística transpuesta, si se consideran pares de elementos distintos de manera independiente, se tendrá la situación que se muestra en la tabla 10-8.

TABLA 10-8 Número de comparaciones entre palabras de la heurística transpuesta.

Consulta	Secuencia	*C*	*A*	*C*	*B*	*C*	*A*	
			0		1		1	(*A*, *B*)
		0	1	1		0	1	(*A*, *C*)
		0		0	1	1		(*B*, *C*)
		0	1	1	2	1	2	

El número de comparaciones entre palabras, con este tipo de cálculo, es 1 + 1 + 2 + 1 + 2 = 7. Sin embargo, esto no es correcto. La comparación entre palabras correctas se encuentra como sigue:

Secuencia de consulta	*C*	*A*	*C*	*B*	*C*	*A*
Ordenamiento de datos	C	*AC*	*CA*	*CBA*	*CBA*	*CAB*
Número de comparaciones entre palabras	0	1	1	2	0	2

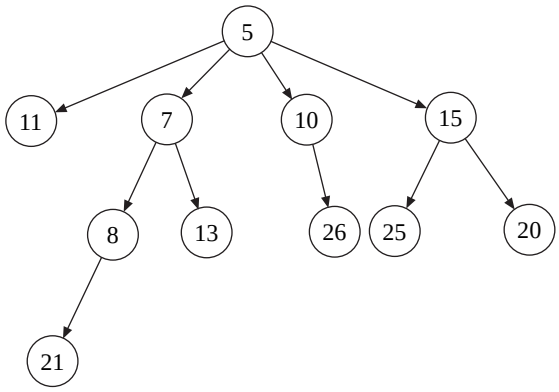
En realidad, el número total de comparaciones entre palabras es $1 + 1 + 2 + 2 = 6$. Esto demuestra que la propiedad de pares independientes no se cumple bajo la heurística transpuesta.

10-5 PAIRING HEAP Y SU ANÁLISIS AMORTIZADO

En este apartado se presenta el pairing heap y se lleva a cabo un análisis amortizado de su estructura de datos. Se hará evidente que el pairing heap puede degenerar de tal modo que será difícil eliminar el mínimo de él. No obstante, el análisis amortizado también demostrará que después que el apareamiento se deteriora hasta un nivel indeseable, puede recuperarse pronto.

En la figura 10-13 se muestra un ejemplo típico de pairing heap. La razón de esta denominación se explicará después de presentar la operación *delete minimum*.

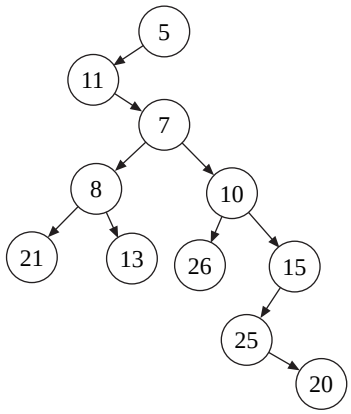
FIGURA 10-13 Un pairing heap.



Observe que, justo como en cualquier pairing heap, la clave del nodo padre es menor que la clave del nodo en sí. Por supuesto, es necesario utilizar una estructura de datos bien estructurada para implementar este heap. Para un pairing heap, se usa la técnica del árbol binario. La representación de árbol binario en la figura 10-13 ahora se muestra en la figura 10-14. Según se muestra en esta figura, cada nodo está unido al descendiente de la extrema izquierda y a su hermano derecho siguiente, en caso de te-

ner alguno. Por ejemplo, en la figura 10-13, el nodo 7 tiene al nodo 8 como su descendiente de la extrema izquierda y al nodo 10 como su hermano derecho siguiente. En consecuencia, el nodo 7 está unido tanto al nodo 8 como al nodo 10 en la representación de árbol binario que se muestra en la figura 10-14.

FIGURA 10-14 Representación de árbol binario del pairing heap que se muestra en la figura 10-13.



Para un pairing heap hay siete operaciones, que son las siguientes:

1. *make heap*(*h*): para crear un nuevo heap vacío denominado *h*.
2. *find min*(*h*): para encontrar el mínimo de un heap *h*.
3. *insert*(*x*, *h*): para insertar un elemento *x* en el heap *h*.
4. *delete min*(*h*): para eliminar el elemento mínimo del heap *h*.
5. *meld*(h_1, h_2): para crear un heap al unir dos heaps h_1 y h_2.
6. *decrease* (Δ, *x*, *h*): para disminuir el elemento *x* por el valor Δ en el heap *h*.
7. *delete*(*x*, *h*): para eliminar el elemento *x* del heap *h*.

Hay una operación básica interna, denominada link (h_1, h_2), que vincula (link) dos heaps en un nuevo heap. Se demostrará que muchas de las operaciones antes mencionadas se basan en la operación *link*. En la figura 10-15 se ilustra una típica operación *link*.

A continuación se ilustrará la forma en que las siete operaciones se implementan en pairing heaps.

1. *make heap*(*h*): Simplemente se asigna una ubicación de la memoria al nuevo heap.
2. *find min*(*h*): Se regresa la raíz del heap lo cual es trivial.

3. *insert*(*x*, *h*): Esto se hace en dos pasos. Primero se crea un nuevo árbol de un nodo y se vincula (link) con *h*. En las figuras 10-16 y 10-17 se muestra cómo se hace.

FIGURA 10-15 Ejemplo de vinculación con *link*(h_1, h_2).

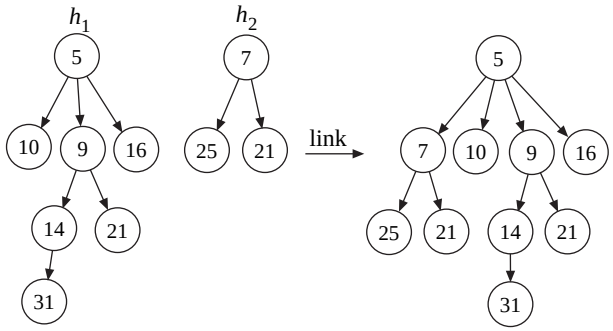


FIGURA 10-16 Ejemplo de inserción con *insert*(*x*, *h*).

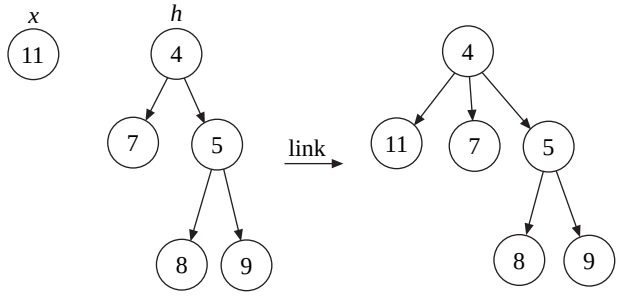
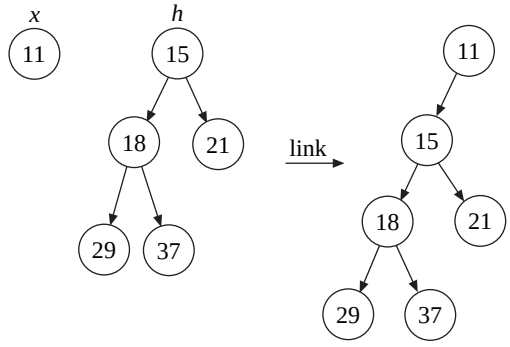


FIGURA 10-17 Otro ejemplo de inserción.



4. *meld*(h_1, h_2): Simplemente se regresa el heap formado al vincular h_1 y h_2.
5. *decrease*(Δ, x, h): Esta operación consta de tres pasos.
   Paso 1: Restar de x.
   Paso 2: Si x es la raíz, entonces terminar.
   Paso 3: Cortar la arista que une a x con su padre. Así se obtienen dos árboles. Vincular estos dos árboles resultantes.

Considere el caso *decrease*(3, 9, h), donde h se muestra en la figura 10-18. En la figura 10-19 se ilustra la forma en que se lleva a cabo esta operación.

FIGURA 10-18 Un pairing heap para ilustrar la operación decrease.

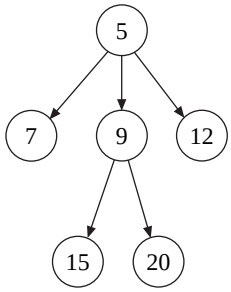
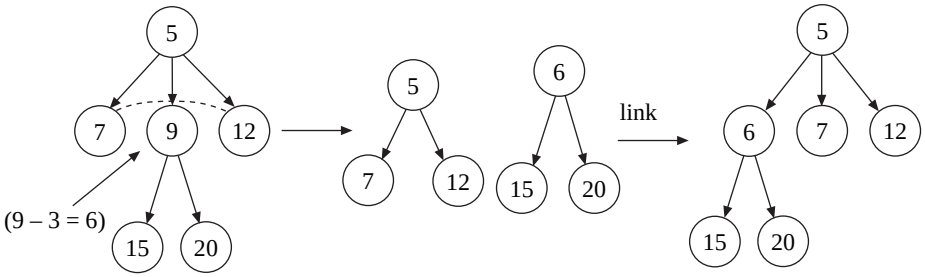


FIGURA 10-19 La operación decrease (3, 9, h) para el pairing heap de la figura 10-18.

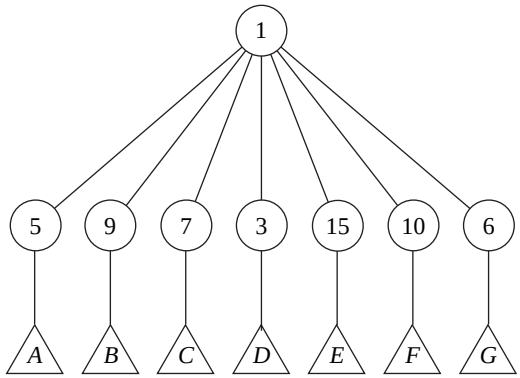


6. *delete*(x, h): En esta operación hay dos pasos:
   Paso 1: Si x es la raíz, entonces regresar (*delete* min(x))
   Paso 2: En caso contrario,
      Paso 2.1: Cortar la arista que une x y su padre.
      Paso 2.2: Realizar una *delete* min(x) en el árbol cuya raíz está en x.
      Paso 2.3: Vincular (link) los árboles resultantes.

Observe que dentro de *delete*(*x*, *h*) hay una operación *delete* min(*h*) que aún no se describe. El análisis de esta operación se pospuso deliberadamente porque es fundamental para el análisis amortizado del pairing heap. De hecho, es esta operación *delete* min(*h*) lo que hace del pairing heap una bella estructura de datos, en el sentido del análisis amortizado.

Antes de describir la operación *delete* min(*h*) se considerará el pairing heap de la figura 10-20. En la figura 10-21 se muestra la representación de árbol binario de este pairing heap. Como puede verse este árbol binario está bastante sesgado a la derecha.

FIGURA 10-20 Un pairing heap para ilustrar la operación delete min(*h*).



Una vez que se elimina el mínimo del heap; es decir, la raíz del árbol binario, es necesario reconstruir el heap. Este paso de reconstrucción debe realizar una función: encontrar el mínimo entre los elementos del segundo nivel del heap. De manera correspondiente a la representación de árbol binario, todos los elementos del segundo nivel están en una ruta, como se muestra en la figura 10-21. Si en el heap hay muchos elementos del segundo nivel, la ruta correspondiente es relativamente larga. En este caso, la operación para encontrar el mínimo requiere un tiempo relativamente largo, lo cual es indeseable.

Se demostrará que hay una elevada probabilidad de que el número de elementos en el segundo nivel del heap sea pequeña. En otras palabras, el árbol binario correspondiente estará equilibrado.

Se considerará nuevamente el pairing heap de la figura 10-20. La operación *delete* min(*h*) primero corta todos los vínculos que unen a la raíz del heap. Esto da por resultado siete heaps, que se muestran en la figura 10-22.

FIGURA 10-21 Representación de árbol binario del heap apareado que se muestra en la figura 10-19.

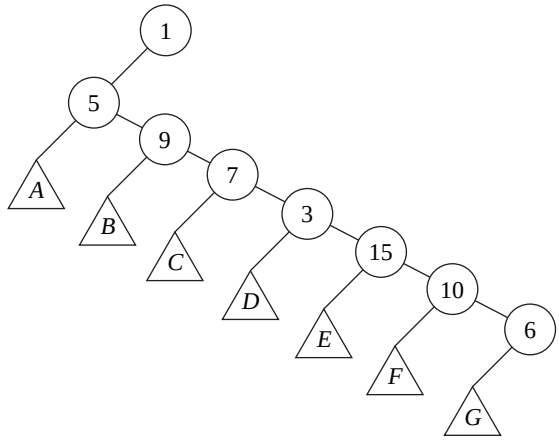
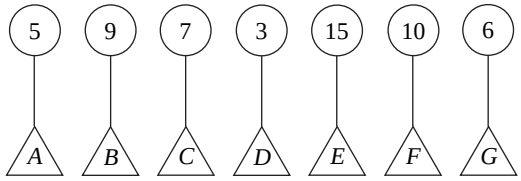
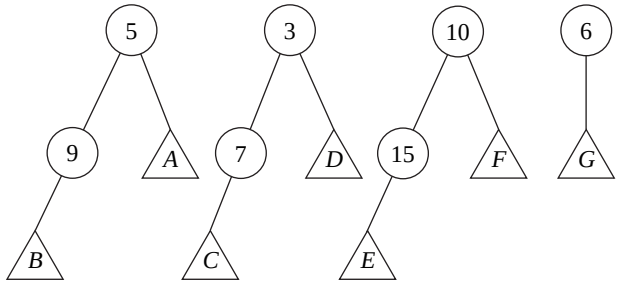


FIGURA 10-22 El primer paso de la operación delete min(*h*).



En el primer paso de la operación delete min(*h*), se fusionan los heaps resultantes en pares, el primero y el segundo, el tercero y el cuarto, etc., como se muestra en la figura 10-23.

FIGURA 10-23 El segundo paso de la operación delete min(*h*).

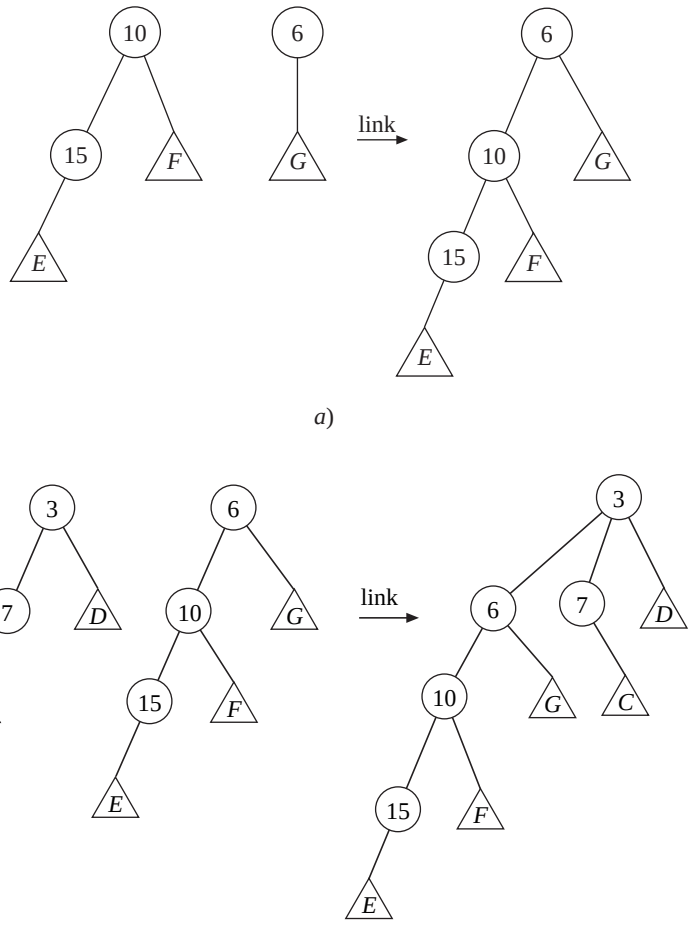


Después de la operación de fusión por pares, se vinculan los heaps apareados uno por uno hasta el último heap, empezando con el penúltimo heap, luego con el antepenúltimo, y así sucesivamente, como se muestra en la figura 10-24.

Los tres pasos de *delete* min(*h*) pueden implementarse directamente en la representación binaria del heap.

La representación de árbol binario del heap resultante en la figura 10-24 se muestra en la figura 10-25. Resulta evidente que este árbol binario está mucho más equilibrado que el de la figura 10-21.

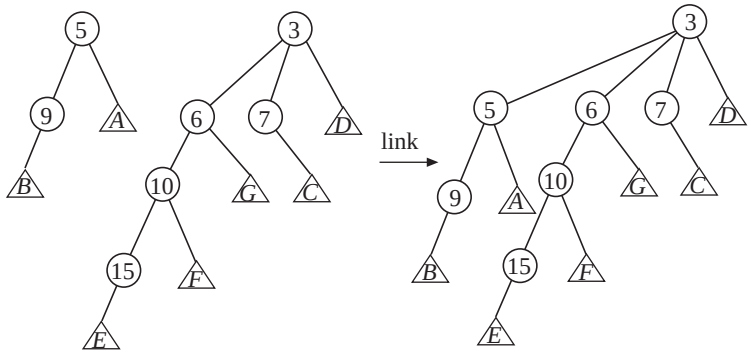
FIGURA 10-24 El tercer paso de la operación delete min(*h*).



a)

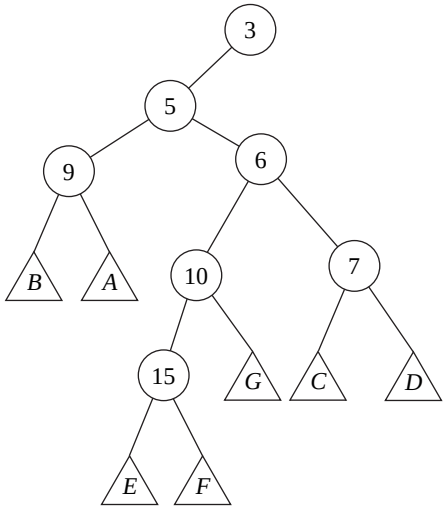
b)

FIGURA 10-24 (*continuación*)



c)

FIGURA 10-25 Representación de árbol binario del heap resultante que se muestra en la figura 10-24.



Una vez que se ha ilustrado la operación *delete* min(*h*), ahora es posible ilustrar la operación *delete*(*x*, *h*). Considere el heap apareado en la figura 10-26. Si $x = 6$, entonces después de eliminar *x* se obtiene el heap resultante que se muestra en la figura 10-27.

A continuación se presentará el análisis amortizado de las operaciones del heap apareado. Como ya se hizo, primero se definirá la función potencial.

Dado un nodo *x* de un árbol binario, sea $s(x)$ el número de nodos en su subárbol, incluyendo *x*. Sea el rango de *x*, denotado por $r(x)$, definido como $\log(s(x))$. El potencial de un árbol es la suma de los rangos de todos los nodos.

FIGURA 10-26 Pairing heap para ilustrar la operación delete(x, h).

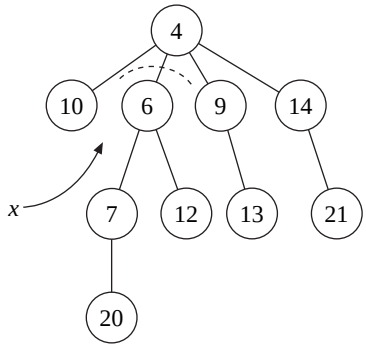
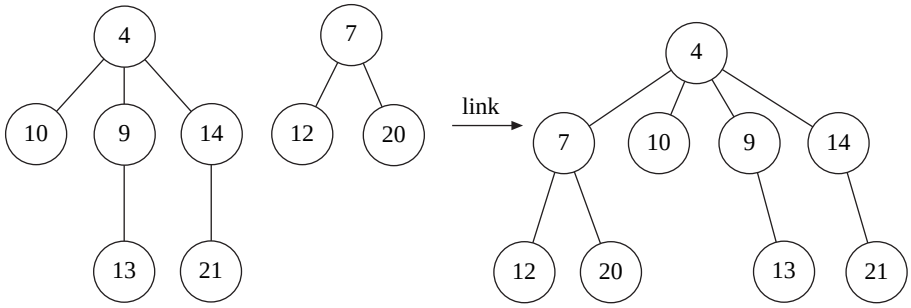


FIGURA 10-27 Resultado de eliminar 6 del pairing heap de la figura 10-26.



Por ejemplo, considere la figura 10-28, que muestra los dos potenciales de los dos árboles.

De las siete operaciones de los pairing heaps, hacer un heap y encontrar el mínimo no debe provocar ningún cambio en el potencial. Así como para *insert*, *meld* y *decrease*, puede verse fácilmente que el cambio de potencial es a lo sumo $\log(n + 1)$, que es causado por las operaciones internas de *link*. Así como para *delete* y *minimum*, la operación crítica es *delete minimum*. A continuación se presenta un ejemplo para ilustrar cómo cambia el potencial.

Considere la figura 10-29. El árbol binario del pairing heap de la figura 10-29*a*) se muestra en la figura 10-29*b*).

Sean r_a, r_b, r_c y r_d que representan los rangos de A, B, C y D, respectivamente. Sea S_a, S_b, S_c y S_d el número total de nodos en A, B, C y D, respectivamente. Entonces el potencial del árbol binario en la figura 10-29*b*) es

FIGURA 10-28 Potenciales de dos árboles binarios.

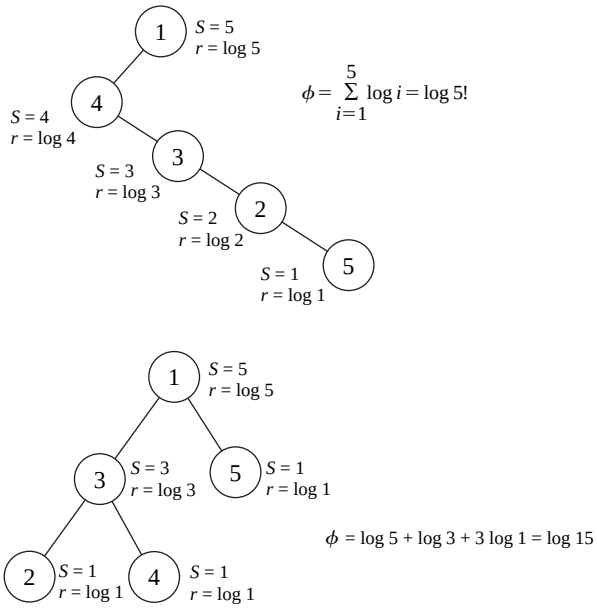
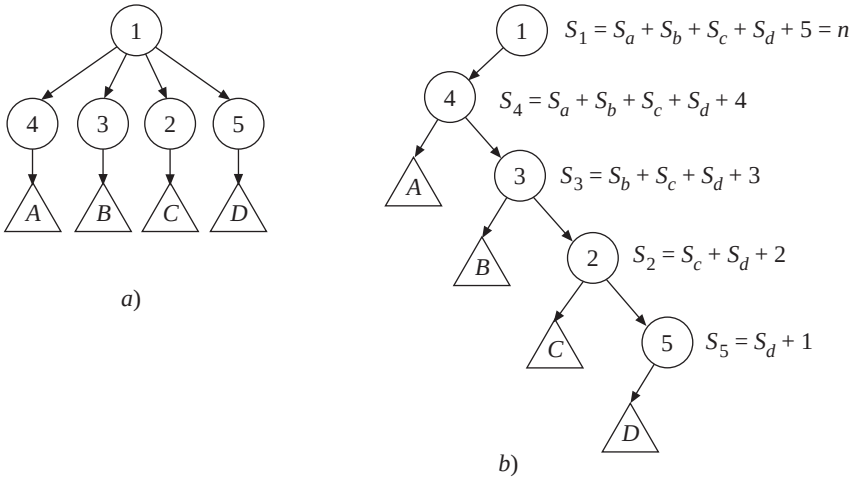


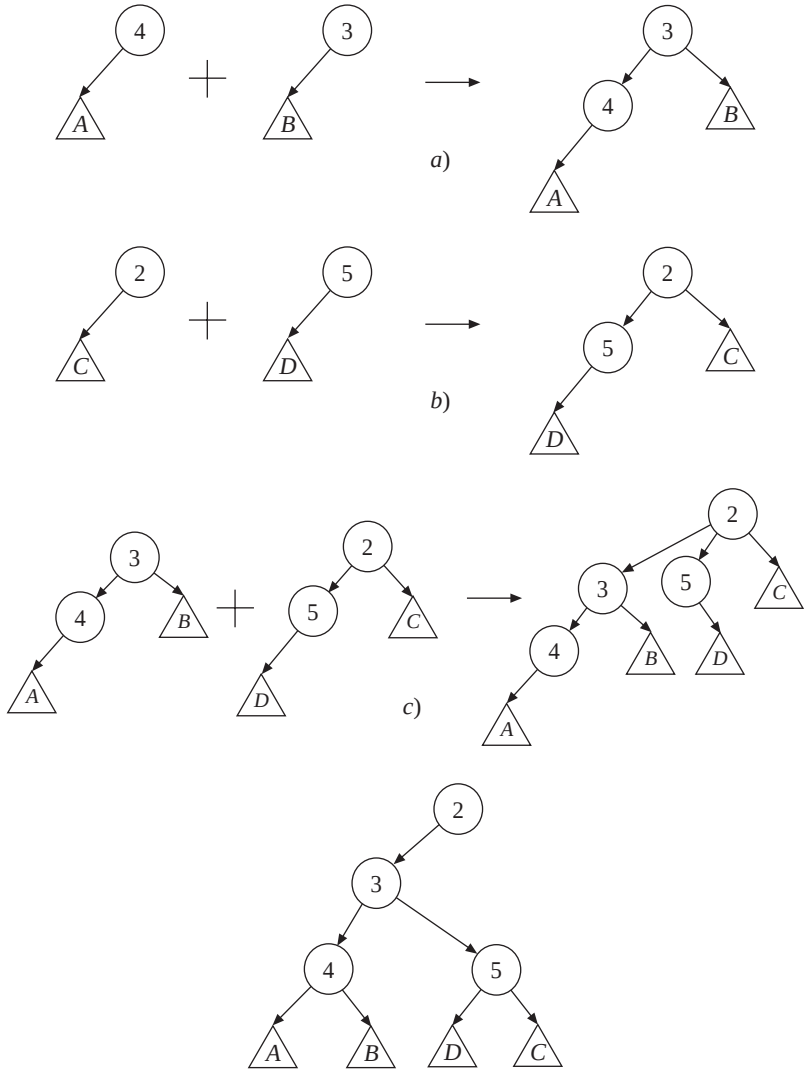
FIGURA 10-29 Un heap y su árbol binario para ilustrar el cambio de potencial.



$$\phi = r_a + r_b + r_c + r_d + \log (S_a + S_b + S_c + S_d + 5) + \log (S_a + S_b + S_c + S_d + 4) + \log (S_b + S_c + S_d + 3) + \log (S_c + S_d + 2) + \log (S_d + 1).$$

Después de que se elimina el mínimo; a saber, 1, hay una secuencia de operaciones de apareamiento, lo cual se muestra en la figura 10-30*a*) a *c*). El heap resultante se muestra en la figura 10-30*c*), y su representación de árbol binario de muestra en la figura 10-30*d*).

FIGURA 10-30 Secuencia de operaciones de apareamiento después de la operación de eliminación del mínimo en el heap de la figura 10-29 y la representación de árbol binario del heap resultante.



El nuevo potencial se cambia a

$$\begin{aligned}\phi' &= r_a + r_b + r_c + r_d + \log (S_a + S_b + 1) + \log (S_c + S_d + 1) \\ &\quad + \log (S_a + S_b + S_c + S_d + 3) + \log (n - 1) \\ &= r_a + r_b + r_c + r_d + \log (S_a + S_b + 1) + \log (S_c + S_d + 1) \\ &\quad + \log (S_a + S_b + S_c + S_d + 3) + \log (S_a + S_b + S_c + S_d + 4).\end{aligned}$$

En general, sea una secuencia de operaciones $OP_1, OP_2, \ldots, OP_q$ y cada OP_i puede ser una de las siete operaciones de apareamiento. Sea

$$a_i = t_i + \phi_i - \phi_{i-1}$$

donde ϕ_i y ϕ_{i-1} son los potenciales antes y después de OP_i, respectivamente, y t_i es el tiempo necesario para ejecutar OP_i. Luego, los análisis se centran en la operación *delete minimum*. Se obtendrá una cota superior amortizada para esta operación. Debe resultar evidente que las demás operaciones tienen la misma cota superior.

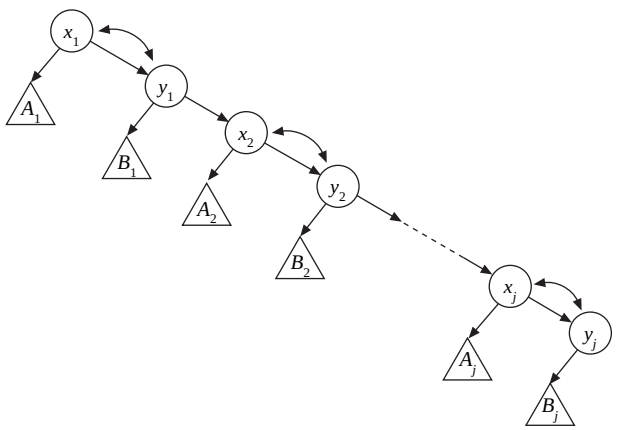
Observe que la operación *delete minimum* consta de las siguientes operaciones:

1. Eliminar la raíz.
2. Fusionar por pares.
3. Fusionar con el último heap uno por uno.

Para la primera operación, es fácil ver que el cambio máximo de potencial es $\Delta\phi_i = -\log n$.

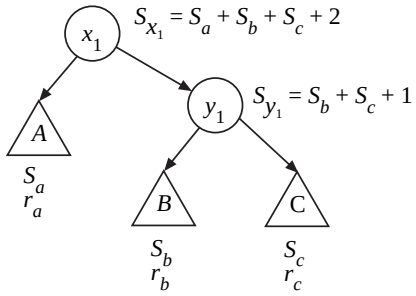
Luego, considere la fusión por pares. Originalmente, el árbol binario posee una cadena de nodos que se muestra en la figura 10-31.

FIGURA 10-31 Las operaciones de apareamiento.



A continuación se vuelve a trazar la figura 10-31, que se muestra en la figura 10-32.

FIGURA 10-32 Figura 10-31 vuelta a trazar.



Sean r_a, r_b y r_c que denotan los rangos de *A*, *B* y *C*, respectivamente. Sean S_a, S_b y S_c el número de nodos en *A*, *B* y *C*, respectivamente. Entonces el potencial antes de la primera fusión es

$$\phi_{antes} = r_a + r_b + r_c + \log (S_a + S_b + S_c + 2) + \log (S_b + S_c + 1).$$

La primera operación de fusión une dos heaps, mostrados en la figura 10-33. Hay dos posibilidades, dependiendo de las relaciones entre x_1 y y_1. Las representaciones de árboles binarios de los dos heaps se muestran en la figura 10-34. Para las dos representaciones posibles de árboles binarios, es necesario agregarles la parte restante, lo cual se muestra en la figura 10-35.

FIGURA 10-33 Los dos heaps posibles después de la primera fusión.

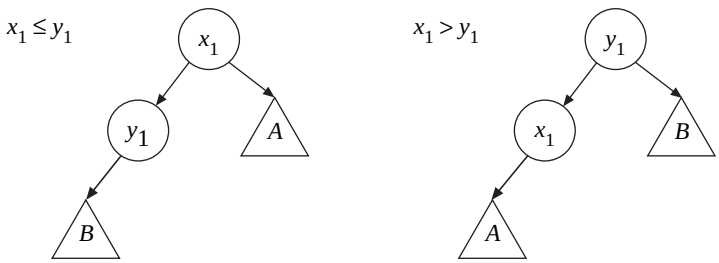


FIGURA 10-34 Representaciones de árboles binarios de los dos heaps de la figura 10-33.

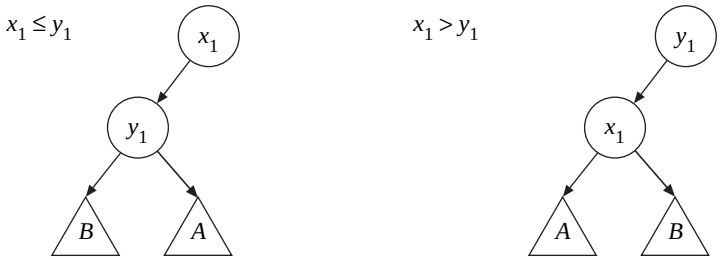
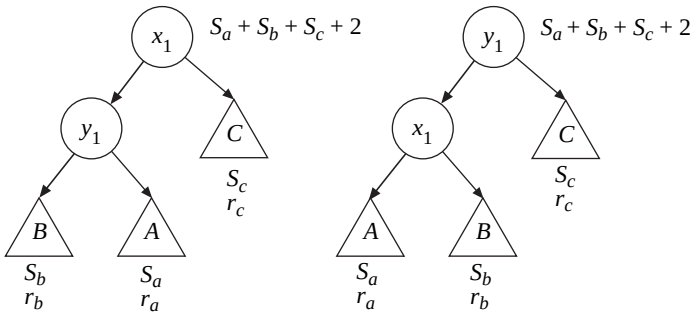


FIGURA 10-35 Árboles binarios resultantes después de la primera fusión.



Para los dos árboles binarios de la figura 10-35, el potencial es

$$\phi_{después} = r_a + r_b + r_c + \log (S_a + S_b + S_c + 2) + \log (S_a + S_b + 1).$$

En consecuencia, el cambio de potencial es

$$\begin{aligned} \Delta\phi_{apareamiento} &= \phi_{después} - \phi_{antes} \\ &= \log (S_a + S_b + 1) - \log (S_b + S_c + 1). \end{aligned}$$

Aunque lo anterior se refiere a la primera fusión, resulta evidente que $\Delta\phi_{apareamiento}$ puede referirse al apareamiento general.

Se demostrará que $\Delta\phi_{apareamiento} \leq 2 \log (S_x) \leq 2 \log (S_c) - 2$ donde S_x denota el número total de nodos en el subárbol con raíz en x antes del cambio. Para demostrar esto, se usará el hecho siguiente:

Si $p, q > 0$ y $p + q \leq 1$, entonces $\log p + \log q \leq -2$.

Es fácil demostrar la propiedad anterior. Luego, sea

$$p = \frac{S_a + S_b + 1}{S_a + S_b + S_c + 2}$$

y $$q = \frac{S_c}{S_a + S_b + S_c + 2}.$$

Entonces $$\log\left(\frac{S_a + S_b + 1}{S_a + S_b + S_c + 2}\right) + \log\left(\frac{S_c}{S_a + S_b + S_c + 2}\right) \leq -2,$$

es decir $(S_a + S_b + 1) - \log(S_c) \leq 2\log(S_a + S_b + S_c + 2) - 2\log(S_c) - 2$.

Pero $\log(S_c) < \log(S_b + S_c + 1)$.

En consecuencia,

$$\log(S_a + S_b + 1) - \log(S_b + S_c + 1) \leq 2\log(S_a + S_b + S_c + 2) - 2\log(S_c) - 2.$$

Así, se tiene

$$\begin{aligned}\Delta\phi_{apareamiento} &\leq 2\log(S_a + S_b + S_c + 2) - 2\log(S_c) - 2\\ &= 2\log(S_x) - 2\log(S_c) - 2.\end{aligned}$$

Para el último par, $S_c = 0$. En este caso,

$$\Delta\phi_{apareamiento} \leq 2\log(S_x).$$

Antes que todo, se observa que es posible suponer con seguridad que hay j pares. Si hay $2j + 1$ nodos, el último no se fusiona con otros y así no hay cambio de potencial debido al último nodo. El cambio total de la fusión por pares es

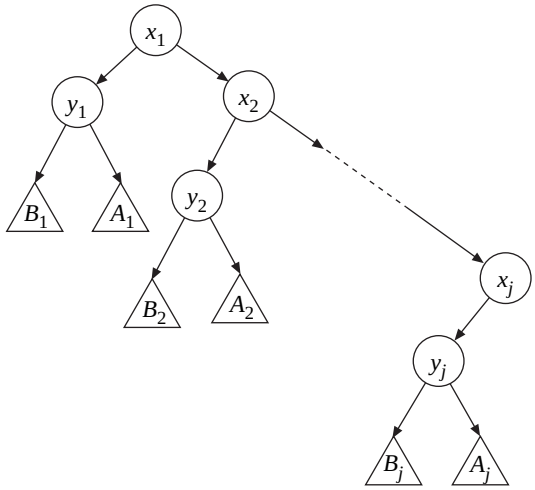
$$\begin{aligned}\Delta\phi_{apareamiento\ total} &= \sum_{i=1}^{j-1} (\text{el } i\text{-ésimo } \Delta\phi_{apareamiento}) + (\Delta\phi_{apareamiento} \text{ debido al } j\text{-ésimo par})\\ &\leq \sum_{i=1}^{j-1} (2\log(S_{x_i}) - 2\log(S_{x_{i+1}}) - 2) + 2\log(S_{x_j})\\ &= 2(\log(S_{x_1}) - \log(S_{x_2}) + \log(S_{x_2}) - \log(S_{x_3}) + \cdots\\ &\quad - \log(S_{x_j})) + 2\log(S_{x_j}) - 2(j-1)\\ &= 2\log(S_{x_1}) - 2(j-1)\\ &\leq 2\log n - 2j + 2.\end{aligned}$$

Así que, para el segundo paso de la operación *delete minimum*,

$$\Delta\phi_{apareamiento\ total} \leq 2 \log n - 2j + 2.$$

Por último, se analiza el paso final. Éste se lleva a cabo después de la operación de apareamiento anterior. Fusiona cada heap uno por uno con el último heap. En consecuencia, se tiene un árbol binario resultante de las operaciones de apareamiento, lo cual se ilustra en la figura 10-36. (Se supone que $x_i < y_i$ para toda i.)

FIGURA 10-36 El árbol binario después de las operaciones de apareamiento.



Considere el último par, que se muestra en la figura 10-37. El potencial

$$\begin{aligned}\phi_{antes} = {} & r_{A_{j-1}} + r_{B_{j-1}} + r_{A_j} + r_{B_j} + \log (S_{A_{j-1}} + S_{B_{j-1}} + 1) \\ & + \log (S_{A_j} + S_{B_j} + 1) + \log (S_{A_j} + S_{B_j} + 2) \\ & + \log (S_{A_{j-1}} + S_{B_{j-1}} + S_{A_j} + S_{B_j} + 4).\end{aligned}$$

Después de la fusión, dependiendo de si $x_{j-1} < x_j$ o $x_{j-1} \geq x_j$, el nuevo árbol binario se ve como el de la figura 10-38*a*) o el de la figura 10-38*b*).

Debido a que los nuevos potenciales son los mismos para ambos árboles binarios que se muestran en las figuras 10-38*a*) y 10-38*b*), en consecuencia se considerará sólo uno de los dos. Para el árbol binario que se muestra en la figura 10-38*a*), el nuevo potencial es

FIGURA 10-37 Un par de subárboles en el tercer paso de la operación delete minimum.

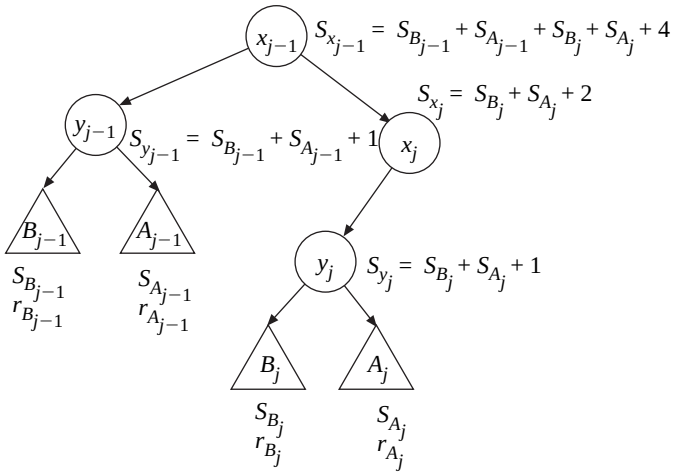
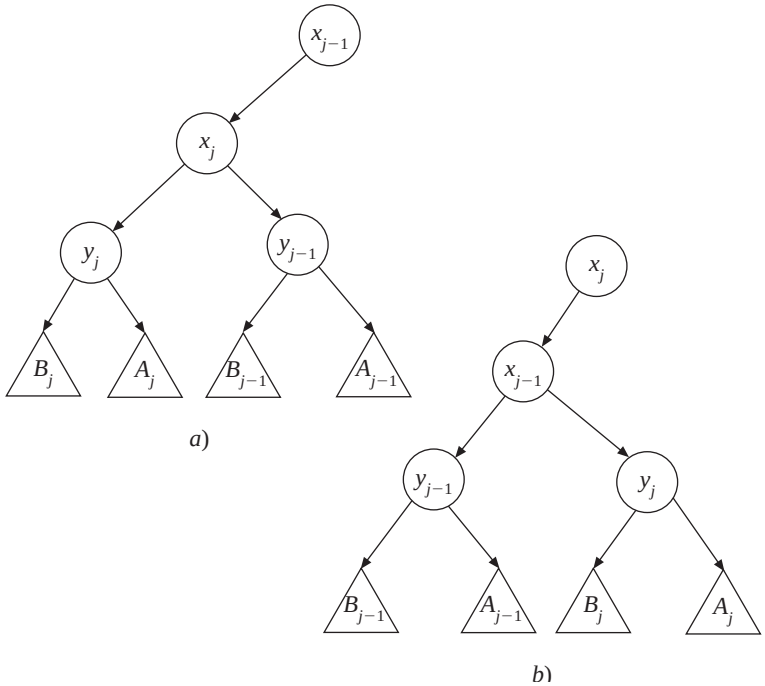


FIGURA 10-38 Resultado de la fusión de un par de subárboles en el tercer paso de la operación delete minimum.



$$\begin{aligned}\phi_{después} &= r_{A_{j-1}} + r_{B_{j-1}} + r_{A_j} + r_{B_j} + \log (S_{A_{j-1}} + S_{B_{j-1}} + 1) + \log (S_{A_j} + S_{B_j} + 1)\\ &\quad + \log (S_{A_{j-1}} + S_{B_{j-1}} + S_{A_j} + S_{B_j} + 3)\\ &\quad + \log (S_{A_{j-1}} + S_{B_{j-1}} + S_{A_j} + S_{B_j} + 4)\\ \Delta\phi &= \phi_{después} - \phi_{antes}\\ &= \log (S_{A_{j-1}} + S_{B_{j-1}} + S_{A_j} + S_{B_j} + 3) + \log (S_{A_j} + S_{B_j} + 2).\end{aligned}$$

$S_{A_{j-1}} + S_{B_{j-1}} + S_{A_j} + S_{B_j} + 4$ es el número total de nodos de todo el árbol y $S_{A_j} + S_{B_j} + 2$ es el número total de nodos del último árbol binario. Sea n_i el número de los nodos en el árbol que consta de x_i, y_i, A_i y B_i. Entonces el número de nodos en los subárboles son $n_1, n_2, \ldots, n_j$. El cambio total de potencial es

$$\begin{aligned}\Delta\phi_{\text{tercer paso}} &= \log (n_1 + n_2 + \cdots + n_j - 1) - \log (n_2 + n_3 + \cdots + n_j)\\ &\quad + \log (n_2 + n_3 + \cdots + n_j - 1) - \log (n_3 + n_4 + \cdots + n_j)\\ &\quad + \cdots + \log (n_{j-1} + n_j - 1) - \log (n_j)\\ &< \log (n - 2) - \log (n_j)\\ &< \log (n - 1).\end{aligned}$$

Ahora puede verse que para toda la operación *delete minimum*,

$$\begin{aligned}a_i &= t_i + \phi_i - \phi_{i-1}\\ &\leq 2j + 1 - \log n + (2 \log n - 2j + 2) + \log (n - 1)\\ &\leq 2 \log n + 3 = O(\log n).\end{aligned}$$

Aunque $O(\log n)$ es una cota superior para la operación *delete minimum*, resulta fácil ver que ésta también puede servir como cota superior para todas las otras operaciones. En consecuencia, puede concluirse que el tiempo amortizado para la operación de apareamiento de heaps es $O(\log n)$.

10-6 ANÁLISIS AMORTIZADO DE UN ALGORITMO PARA LA UNIÓN DE CONJUNTOS DISJUNTOS

En este apartado se analizará un algoritmo fácil de implementar. No obstante, su análisis amortizado revela un extraordinario tiempo de ejecución que es casi lineal. El algoritmo resuelve el problema de preservar una colección de conjuntos disjuntos bajo la operación de unión. Con mayor precisión, el problema consiste en realizar tres tipos de operaciones sobre conjuntos disjuntos: *makeset*, que crea un nuevo conjunto; *find*, que localiza el conjunto que contiene un elemento dado, y *link*, que combina dos conjuntos en uno. Como mecanismo de identificación de los conjuntos, se supondrá que el algoritmo preserva dentro de cada conjunto un elemento arbitrario pero único denominado

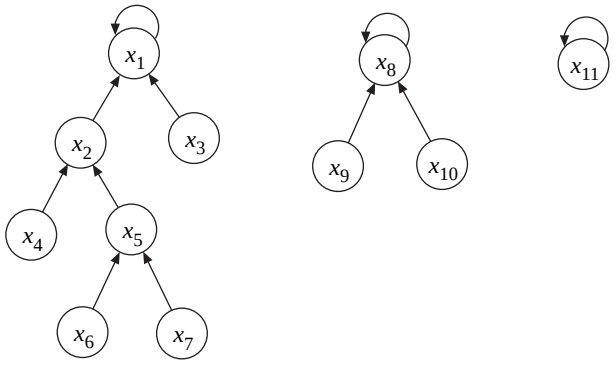
elemento *canónico* del conjunto. Se plantean tres operaciones de conjuntos como sigue:

1. *makeset*(x): crea un nuevo conjunto que contiene el elemento x, que antes no existía.
2. *find*(x): Regresa el elemento canónico del conjunto que contiene el elemento x.
3. *link*(x, y): Forma un nuevo conjunto que es la unión de los dos conjuntos cuyos elementos canónicos son x y y. Destruye los conjuntos anteriores. Selecciona y regresa un elemento canónico del nuevo conjunto. Esta operación supone que $x \neq y$.

Observe que en estas operaciones no hay eliminación.

Para resolver este problema, cada conjunto se representa mediante un árbol con raíz. Los nodos del árbol son los elementos del conjunto con el elemento canónico como la raíz del árbol. Cada nodo x posee un indicador que apunta hacia $p(x)$, su padre en el árbol; la raíz lo hace hacia sí misma. Para realizar *makeset*(x), $p(x)$ se define como x. Para realizar *find*(x), se siguen los indicadores padres desde a hasta la raíz del árbol que contiene a x y regresa el árbol. Para realizar link(x, y), $p(x)$ se define como y y como elemento canónico del nuevo conjunto se regresa y. Consulte la figura 10-39. La operación *find*(x_6) regresa x_1 y link(x_1, x_8) hace que x_1 apunte hacia x_8.

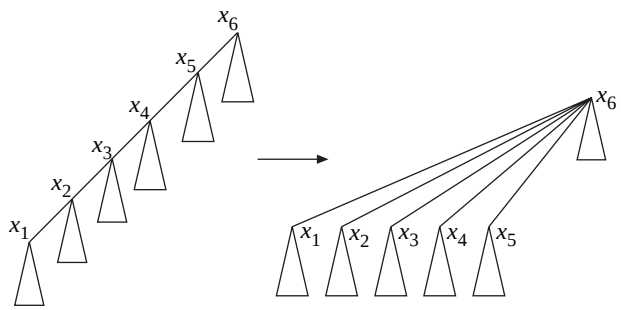
FIGURA 10-39 Representación de los conjuntos $\{x_1, x_2, \ldots, x_7\}$, $\{x_8, x_9, x_{10}\}$ y $\{x_{11}\}$.



Este algoritmo directo (naive) no es muy eficiente, ya que requiere tiempo $O(n)$ por operación *find* en el peor caso, donde n es el número total de elementos (operación *makeset*). Al agregar dos heurísticas al método, es posible mejorar mucho su desempeño. La primera, denominada compresión de ruta, modifica la estructura del árbol du-

rante una operación *find* al acercar más los nodos a la raíz: durante la ejecución de *find*(x), después de localizar la raíz r del árbol que contiene a x, se hace que todo nodo en la ruta de x a r apunte directamente a r (consulte la figura 10-40). La compresión de ruta incrementa el tiempo de una sola operación por un factor constante, aunque ahorra tiempo suficiente para ejecutar operaciones *find* posteriores.

FIGURA 10-40 Compresión de la ruta [x_1, x_2, x_3, x_4, x_5, x_6].



La segunda heurística, denominada unión por rango, preserva al árbol corto (o poco profundo). El rango de un nodo x, denotado por *rank*(x), se define como sigue:

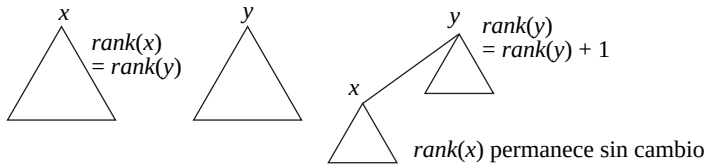
1. Cuando se ejecuta *makeset*(x), *rank*(x) se asigna a 0.
2. Cuando se realiza la operación link, sean x y y dos raíces. Hay dos casos:
   a) Caso 1: *rank*(x) = *rank*(y).
   En este caso se hace que x apunte hacia y, *rank*(x) se incrementa por uno y como elemento canónico se regresa y. (Consulte la figura 10-41*a*).) El rango de x permanece sin cambio.
   b) Caso 2: *rank*(x) < *rank*(y).
   En este caso se hace que x apunte hacia y, *rank*(x) se incrementa por uno y como elemento canónico se regresa y. Los rangos de x y y permanecen sin cambio. (Consulte la figura 10-41*b*).)
3. Cuando se ejecuta la heurística de compresión de ruta, ningún rango cambia.

A continuación se presentan algunas propiedades interesantes de los rangos según se han definido. Estas propiedades serán de utilidad cuando se analice el desempeño del algoritmo.

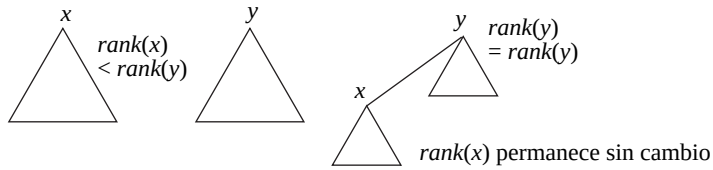
Propiedad 1. *Si* $x \neq p(x)$, entonces *rank*(x) < *rank*($p(x)$).

El hecho de que esta propiedad sea verdadera se debe a la heurística de unión por rango. Cuando se vinculan dos árboles, la nueva raíz posee el rango más alto.

FIGURA 10-41 Unión por rangos.



a) Raíces con rangos iguales



b) Raíces con rangos diferentes

Propiedad 2. Inicialmente, el valor de *rank*(x) es cero y se incrementa con el transcurso del tiempo hasta que deja de ser una raíz; posteriormente, *rank*(x) no cambia. El valor de *rank*($p(x)$) es una función no decreciente del tiempo.

Observe que cuando se inicializa x, al principio su rango es cero. Luego, en tanto x sigue siendo raíz, su rango jamás disminuye. En cuanto tiene un nodo padre, su rango permanece sin cambio por siempre, ya que la heurística de compresión de ruta no compensa el rango.

Considere la figura 10-42. Suponga que ésta es una ruta que recorre el algoritmo durante una operación *find*. Debido a la propiedad 1,

$$rank(x_1) < rank(x_2) \ldots < rank(x_r).$$

Después de la operación de compresión de ruta, esta parte del árbol se ve como se muestra en la figura 10-43. Luego, x_1 tiene un nuevo padre x_r y resulta evidente que este nuevo padre x_r tiene un rango mucho más alto que el padre previo de x_1. Después, x_r puede asociarse con algún otro nodo con rango aún más elevado. En consecuencia, es posible que x_1 puede más tarde tener otro nuevo nodo padre con rango aún más alto. Ésta es la razón por la que *rank*($p(x)$) jamás decrece.

Propiedad 3. *El número de nodos en un árbol con raíz en x es por lo menos* $2^{rank(x)}$.

Resulta evidente que la propiedad 3 se cumple para todo árbol cuando se inicializa. Ahora, suponga que la propiedad 3 es verdadera antes de la vinculación de dos árboles.

FIGURA 10-42 Una operación de encontrar una ruta.

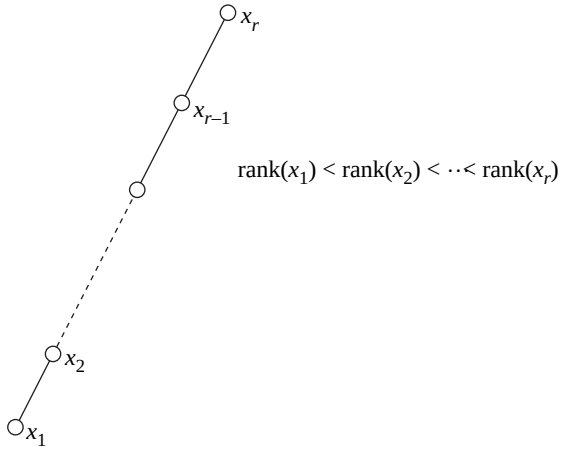
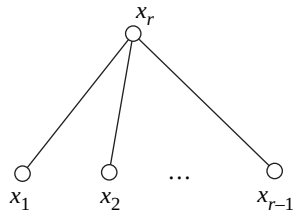


FIGURA 10-43 La operación de encontrar una ruta en la figura 10-42 después de la operación de compresión de ruta.



Sean x y y las raíces de los dos árboles. Si $rank(x) < rank(y)$, entonces la nueva raíz del árbol es y. El rango de y permanece sin cambio y el nuevo árbol tiene más nodos que antes. Así, la propiedad 3 se cumple para este nuevo árbol. Si $rank(x) = rank(y)$, entonces el nuevo árbol tiene por lo menos $2^{rank(x)} + 2^{rank(y)} = 2 \times 2^{rank(y)} = 2^{rank(y)+1}$ nodos. Sin embargo, el rango de la nueva raíz es $rank(y) + 1$. En consecuencia, la propiedad 3 se cumple nuevamente para este caso.

Propiedad 4. Para cualquier $k \geq 0$, *el número de nodos de rank k es cuando mucho* $n/2^k$, donde n es el número de elementos.

Es posible considerar que la raíz del árbol es un nodo con rango k. Por la propiedad 3, este nodo tiene por lo menos 2^k nodos. Así, no puede haber más de $n/2^k$ nodos con rango k; en caso contrario, habría más de k nodos, lo cual es imposible.

Propiedad 5. *El rango más alto es* $\log_2 n$.

El rango más alto se alcanza cuando sólo hay un árbol. En este caso, $2^k \leq n$ según indica la propiedad 3. Entonces, $k \leq \log_2 n$.

Nuestro objetivo es analizar el tiempo de ejecución de una secuencia compuesta de las tres operaciones con conjuntos. Resulta fácil ver que tanto *makeset*(x) como a continuación se analizará sólo el tiempo necesario para las operaciones *find*. Se usará m para denotar el número de operaciones *find* y n para denotar el número de elementos. Así, el número de operaciones *makeset* es n, el número de operaciones link es cuando mucho $n - 1$ y $m \geq n$. El análisis es difícil porque las compresiones de ruta modifican la estructura de los árboles de manera complicada. Sin embargo, mediante el análisis amortizado, es posible obtener la cota para el peor caso, $O(m\alpha(m, n))$, donde $\alpha(m, n)$ es una funcional inversa de la función de Ackermann. Para $i, j \geq 1$, la función de Ackermann $A(i, j)$ se define como

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

y la función inversa de Ackermann $\alpha(m, n)$ para $m, n \geq 1$ se define como

$$\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log_2 n\}.$$

La propiedad más importante de $A(i, j)$ es su crecimiento explosivo. Se analizarán unos cuantos:

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

Así, $\alpha(m, n) \leq 3$ para valores razonables de m/n y para $n < 2^{16} = 65\,536$. De hecho, para todos los efectos prácticos, $\alpha(m, n)$ es una constante no mayor que 4. Al final, se deducirá una cota superior de $O(m\alpha(m, n))$ sobre el tiempo total de ejecución de las m operaciones *find*.

Antes de analizar formalmente el algoritmo, brevemente se describirán las ideas fundamentales de nuestro análisis. Éste consta de los siguientes pasos:

1. Cada nodo de los árboles se asocia con un nivel i. Éste está relacionado con el rango. Se demostrará que hay tres niveles 1, 2, …, $\alpha(m, n) + 1$. Si un nodo tiene

un nivel alto, esto indica que el nodo en cuestión posee un gran número de hermanos, o que esta parte del árbol es más bien plana. Para el nivel i, los enteros se dividen en bloques (i, j). Esto se explicará después.

2. Mediante algún mecanismo, los nodos pueden dividirse en dos tipos: nodos crédito y nodos débito. Para cualquier ruta, el número de nodos crédito es cuando mucho una constante. Entonces, si una ruta es larga, siempre se debe a un gran número de nodos débito. Al revés, si una ruta es corta, mayoritariamente hay nodos crédito.
3. El tiempo total gastado por las operaciones *find* es la suma del número de nodos crédito atravesados por las operaciones *find* y el número de nodos débito atravesados por las operaciones *find*. Debido a que el número total de nodos crédito atravesados por las operaciones *find* está acotado por una constante, es importante encontrar una cota superior del número de nodos crédito atravesados por las operaciones *find*.
4. Para un nodo débito que pertenece al bloque (i, j), si es atravesado por el algoritmo de la operación *find* $b_{ij} - 1$ veces, donde b_{ij} se explicará después, el nivel de este nodo aumentará a $i + 1$.
5. Sea n_{ij} el número de nodos débito que hay en el bloque (i, j). El valor de $n_{ij}(b_{ij} - 1)$ es el número de unidades temporales gastadas al atravesar los nodos débito por las operaciones *find*, lo cual elevará todos los nodos en el bloque (i, j) al nivel $i + 1$.
6. Debido a que el nivel más alto es $\alpha(m, n) + 1$, $\sum_{i=1}^{\alpha(m,n)+1} \sum_{j \geq 0} n_{ij}(b_{ij} - 1)$ es la cota superior del número total de nodos débito atravesados por estas m operaciones *find*.

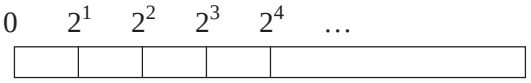
Para empezar, es necesario definir niveles. Nuevamente se revisará la función de Ackermann. Esta función esencialmente divide los enteros en bloques para niveles diferentes. A continuación se vuelve a escribir la función de Ackermann:

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

La función anterior será utilizada para dividir los enteros. Es decir, para el nivel i, los enteros se dividen en bloques definidos por $A(i, j)$. El bloque $(i, 0)$ contiene los enteros de 0 a $A(i, 1) - 1$ y el bloque (i, j) contiene los enteros de $A(i, j)$ a $A(i, j + 1) - 1$ para $j \geq 1$. Por ejemplo, para el nivel 1,

$$A(1, j) = 2^j.$$

Así, los bloques pueden ilustrarse como sigue:



Para el nivel 2,

$$A(2, 1) = 2^2$$
$$A(2, 2) = 2^4 = 16$$
$$A(2, 3) = 2^{16} = 65\,536.$$

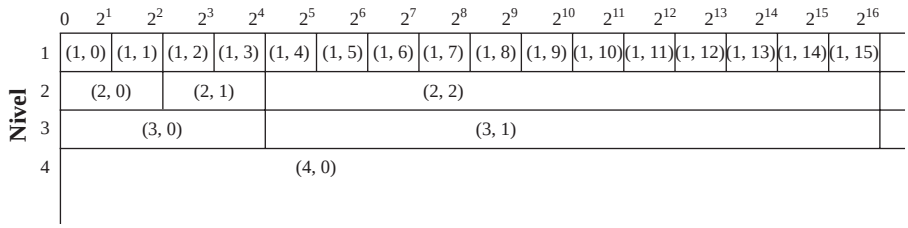
Para el nivel 3,

$$\begin{aligned} A(3, 1) &= 2^4 = 16 \\ A(3, 2) &= A(2, A(3, 1)) \\ &= A(2, 16) \\ &= A(1, A(2, 15)) \\ &= 2^{A(2, 15)}. \end{aligned}$$

Debido a que $A(2, 3)$ ya es un número bastante grande, $A(2, 15)$ es tan grande que para efectos prácticos, $2^{A(2, 15)}$ puede considerarse como infinito.

Al combinar estos tres niveles se tiene un diagrama que se muestra en la figura 10-44.

FIGURA 10-44 Diferentes niveles correspondientes a la función de Ackermann**.**



Considere que se tiene un entero $2^3 \leq i < 2^4$. Entonces al nivel 1, está dentro del bloque (1, 3); al nivel 2, está dentro del bloque (2, 1), y al nivel 3, está dentro del bloque (3, 0).

Según se definió antes, p(x) denota el padre de un nodo x. El nivel de x ahora se define como el nivel mínimo i tal que rank(x) y rank(p(x)) están en un bloque común de la partición al nivel i. Si $x = p(x)$, entonces el nivel de x es 0. Suponga que $rank(x)$ está en el boque (1, 1) y que $rank(p(x))$ está en el bloque (1, 3); entonces x está en el nivel 3. Por otra parte, si $rank(x)$ está en el boque (1, 3) y $rank(p(x))$ está en el bloque (1, 6), entonces x está en el nivel 4.

¿Cuán alto puede ser el nivel? Ciertamente, si el nivel es tan alto que el primer bloque de ese nivel incluye $\log_2 n$, entonces este nivel es suficientemente alto por una sencilla razón: $\log_2 n$ es el rango más grande posible debido a la propiedad 5. A continuación se analizará un ejemplo: $n = 2^{16}$, que es bastante grande. En este caso, $\log_2 n = 16$. Como se muestra en la figura 10-44, éste se incluye en el primer bloque del nivel 4. Al recordar que

$$\alpha(m, n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log_2 n\},$$

puede verse fácilmente que $A(\alpha(m, n), m/n) > \log_2 n$. En otras palabras, $\alpha(m, n) + 1$ es el nivel más alto al que pueden asociarse los nodos.

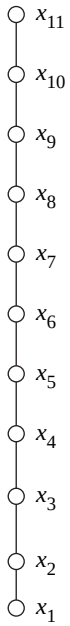
A continuación se considerará una operación *find* que implica una ruta $x_1, \ldots, x_r$. Un ejemplo típico se muestra en la figura 10-45. En esta figura, cada nodo tiene asociado un rango. Al usar $rank(x_i)$ y $rank(x_{i+1})$, es posible determinar el nivel de cada x_i.

Para toda i, $0 \leq i \leq \alpha(m, n) + 1$, el último nodo del nivel i en la ruta se considera un nodo crédito y los otros nodos, como nodos débito. Todos los nodos crédito y los nodos débito también se muestran en la figura 10-45. Según la definición de nodo crédito, *el número total de nodos crédito de cualquier ruta find está acotado por* $\alpha(m, n) + 2$. *Así, el número total de nodos crédito atravesados por el algoritmo está acotado por* $m(\alpha(m, n) + 2)$.

A continuación se analizará cómo el nivel de un nodo puede ser muy elevado. Por definición, el nivel de un nodo x es alto si la diferencia entre $rank(x)$ y $rank(p(x))$ es muy alta. En consecuencia, puede plantearse la siguiente pregunta: ¿Cómo puede el rango del nodo padre de x ser mucho más alto que el rango de x? Sólo hay dos operaciones: unión y compresión. Cuando se realiza una operación de unión, la diferencia entre el rango de x y el rango de $p(x)$ sólo puede ser uno.

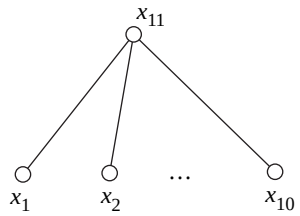
No obstante, cuando se realiza una operación de compresión, la diferencia entre el rango de un nodo x y el de su nodo padre puede ser muy marcada. Imagine que se ha ejecutado una operación *find* sobre la ruta que se muestra en la figura 10-45. Después de la operación *find*, la ruta se comprimirá, lo cual se muestra en la figura 10-46. Aunque el rango de x_1 sigue siendo 1, repentinamente el rango de su padre es 150. Así, ahora el nivel de x_1 ha aumentado de 1 a 4.

FIGURA 10-45 Nodos crédito y nodos débito.



	Rango	Nivel	Crédito/débito
x_{11}	150	0	Crédito
x_{10}	18	2	Crédito
x_9	17	1	Crédito
x_8	13	4	Crédito
x_7	12	1	Débito
x_6	10	1	Débito
x_5	7	2	Débito
x_4	5	1	Débito
x_3	3	3	Crédito
x_2	2	1	Débito
x_1	1	2	Débito

FIGURA 10-46 La ruta de la figura 10-45 después de una operación find.



En general, para un nodo en una ruta *find*, después de una operación *find*, tendrá un nuevo nodo padre, y éste debe tener un rango mucho más alto. Se demostrará que si x está en el nivel i y es un nodo débito y $rank(x)$ está en el bloque $(i - 1, j')$ entonces el nuevo padre, que en realidad es la raíz del árbol, se tendrá un rango que está en el bloque $(i - 1, j')$ donde $j' > j$.

Suponga que el nivel de un nodo x es i y que $rank(x)$ está en el bloque $(i - 1, j)$. Considere que el rango de $p(x)$, el padre de x, está en el bloque $(i - 1, j')$. Entonces, ciertamente $j' \geq j$ porque $rank(p(x)) \geq rank(x)$. Además, $j' \neq j$; en caso contrario, x está en el nivel $i - 1$. Así, se concluye que si el nivel del nodo x es i y $rank(x)$ está en el bloque $(i - 1, j')$, entonces el rango de $p(x)$, está en el bloque $(i - 1, j')$, donde $j' > j$.

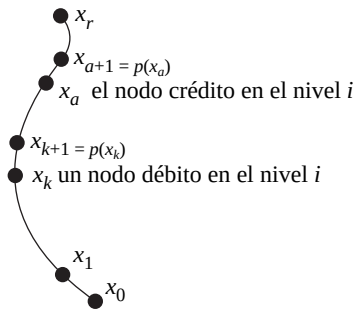
Considere la figura 10-47, donde x_{k+1} es el padre de x_k y x_k es un nodo débito en el nivel i. Debido a que x_k es un nodo débito, por definición debe haber un nodo crédito; por ejemplo x_a, entre x_k y x_r, la raíz de la ruta. Sea x_{a+1} el padre de x_a. Luego, considere que los rangos de x_k, x_{k+1}, x_a, x_{a+1} y x_r están en los bloques $(i - 1, j_1)$, $(i - 1, j_2)$, $(i - 1, j_3)$, $(i - 1, j_4)$ e $(i - 1, j_5)$, respectivamente, como se muestra a continuación.

	rank
x_k	$(i - 1, j_1)$
x_{k+1}	$(i - 1, j_2)$
x_a	$(i - 1, j_3)$
x_{a+1}	$(i - 1, j_4)$
x_r	$(i - 1, j_5)$

Luego, debido a que el nivel de x_k es i, se tiene $j_2 > j_1$.
Debido a que x_a está entre x_{k+1} y x_r, se tiene $j_3 \geq j_2$.
Debido a que el nivel de x_a también es i, se tiene $j_4 > j_3$.
Finalmente, se tiene

$j_5 \geq j_4$.

FIGURA 10-47 Los rangos de los nodos en una ruta.



En resumen, se tiene

$j_1 < j_3 < j_5$.

El análisis anterior responde un caso general. Es posible que $x_a = x_{k+1}$ y $x_r = x_{a+1}$. En este caso, sólo hay tres nodos relevantes para nuestro análisis; a saber, x_k, x_{k+1} y x_r. Es fácil demostrar que, para el caso general y para casos especiales, lo siguiente es cierto: sea x_k un nodo débito en el nivel i. Sean x_{k+1} su nodo padre y x_r el nodo raíz. Sean $rank(x_k)$, $rank(x_{k+1})$ y $rank(x_r)$ que están en los bloques $(i-1, j)$, $(i-1, j')$ e $(i-1, j'')$, respectivamente. Entonces $j < j' < j''$, lo cual se muestra en la figura 10-48.

FIGURA 10-48 Los rangos de x_k, x_{k+1} y x_r en el nivel $i-1$.

	$rank(x_k)$		$rank(x_{k+1})$		$rank(x_r)$	
	$i-1, j$	…	$i-1, j'$	…	$i-1, j''$	

Después de una operación *find*, cada nodo en una ruta, excepto el nodo raíz, se convierte en un nodo hoja debido a la heurística de compresión de ruta. Un nodo hoja, por definición, es un nodo crédito. Así, un nodo débito se convierte en un nodo crédito después de una operación *find*. Seguirá siendo un nodo crédito en tanto no haya unión de operaciones. Después de una operación link, puede volver a ser un nodo débito.

Para un nodo débito, cada operación *find* lo obliga a tener un nuevo nodo padre cuyo rango es mayor que el rango de su nodo padre previo. Además, como ya se demostró, el del nuevo nodo padre está en un bloque de nivel más alto $(i-1)$. Sea b_i el número de bloques en el nivel $(i-1)$ cuya intersección con el bloque (i, j) es no vacía. Por ejemplo, $b_{22} = 12$ y $b_{30} = 2$. Así, se dice que después de realizar $b_{ij} - 1$ operaciones *find* que atraviesan x mientras x es un nodo crédito cada vez, $rank(x)$ y $rank(p(x))$ están en bloque a diferente nivel i. O bien, con más precisión, el nivel de x aumenta a $i+1$ después de $b_{ij} - 1$ cambios de nodo débito. También puede afirmarse que después de un máximo de $b_{ij} - 1$ cambios de nodo débito, el nivel de x incrementa por uno.

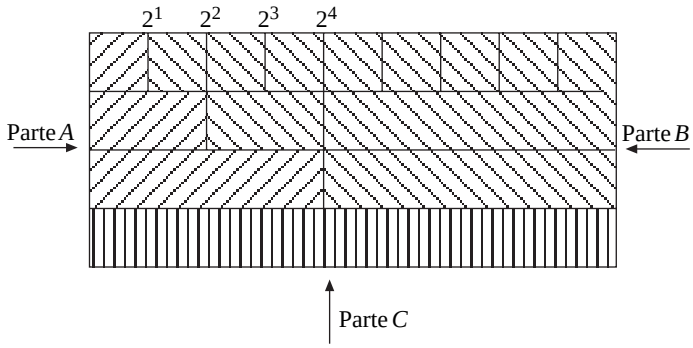
Sea n_{ij} el número total de nodos cuyos rangos están en el bloque (i, j). El número total de veces que las operaciones *find* pueden atravesar nodos débito es a lo sumo

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

En los siguientes párrafos se mostrará cómo puede obtenerse una cota superior de Q. Esta deducción es más bien complicada. De hecho, si el lector sólo está interesado en los principios básicos del análisis amortizado, puede omitir esta parte del análisis.

Nuevamente se considerará el diagrama de partición en la figura 10-44. Este diagrama puede dividirse en tres partes, que se muestran en la figura 10-49.

FIGURA 10-49 División del diagrama de partición de la figura 10-44**.**



Así, se tiene

$$
\begin{aligned}
Q &= \sum_{i=1}^{\alpha(m,n)+1} \sum_{j\geq 0} n_{ij}(b_{ij}-1) \\
&= \sum_{i=1}^{\alpha(m,n)} n_{i0}(b_{i0}-1) && \text{(Parte } A\text{)} \\
&\quad + \sum_{i=1}^{\alpha(m,n)} \sum_{j\geq 1} n_{ij}(b_{ij}-1) && \text{(Parte } B\text{)} \\
&\quad + n_{\alpha(m,n)+1,0}(b_{\alpha(m,n)+1,0}-1). && \text{(Parte } C\text{)}
\end{aligned}
$$

Primero se calculará la parte A. Por definición,

$$
\begin{aligned}
block(i, 0) &= [0, \ldots, A(i, 1) - 1] \\
&= [0, \ldots, A(i-1, 2) - 1] \\
&= [0, \ldots, A(i-1, 1), \ldots, A(i-1, 2) - 1].
\end{aligned}
$$

Esto significa que *block*(*i*, 0) cubre dos bloques de nivel (*i* – 1); a saber, el bloque (*i* – 1, 0) y el bloque (*i* – 1, 1). En consecuencia, se tiene

$$b_{i0} = 2.$$

Por otra parte, $n_{i0} \leq n$. Así, para la parte *A*,

$$\begin{aligned}\sum_{i=1}^{\alpha(m,n)} n_{i0}(b_{i0} - 1) &\leq \sum_{i=1}^{\alpha(m,n)} n(2-1) \\ &= n\alpha(m, n).\end{aligned}$$

La siguiente tarea es calcular la parte *B*, que es

$$\sum_{i=1}^{\alpha(m,n)} \sum_{j \geq 1} n_{ij}(b_{ij} - 1).$$

Primero se obtiene una cota superior para n_{ij}. Por la propiedad 4 antes analizada, se tiene

$$\begin{aligned}n_{ij} &\leq \sum_{k=A(i,j)}^{A(i,j+1)-1} n/2^k \\ &\leq \sum_{k \geq A(i,j)} n/2^k \\ &\leq 2n/2^{A(i,j)} \\ &= n/2^{A(i,j)-1}.\end{aligned} \qquad \textbf{(10-1)}$$

Para b_{ij}, $1 \leq i \leq \alpha(m, n)$ y $j \geq 1$, en general, también se tiene una cota superior.

$$\begin{aligned}block(i, j) &= [A(i, j), \ldots, A(i, j + 1) - 1] \\ &= [A(i - 1, A(i, j - 1)), \ldots, A(i - 1, A(i, j)) - 1] \\ &= [A(i - 1, A(i, j - 1)), A(i - 1, A(i, j - 1) + 1), \ldots, \\ &\quad A(i - 1, A(i, j)) - 1].\end{aligned}$$

Esto significa que para $1 \leq i \leq \alpha(m, n)$ y $j \geq 1$, *block*(*i*, *j*) cubren bloques de nivel $A(i, j) - A(i, j - 1)$ $(i - 1)$. Así, en este caso,

$$b_{ij} \leq A(i, j). \qquad \textbf{(10-2)}$$

Al sustituir (10-1) y (10-2) en la parte *B*, se tiene

$$\begin{aligned}
&\sum_{i=1}^{\alpha(m,n)} \sum_{j\geq 1} n_{ij}(b_{ij}-1) \\
&\quad \leq \sum_{i=1}^{\alpha(m,n)} \sum_{j\geq 1} n_{ij} b_{ij} \\
&\quad \leq \sum_{i=1}^{\alpha(m,n)} \sum_{j\geq 1} (n/2^{A(i,j)-1}) A(i,j) \\
&\quad = n \sum_{i=1}^{\alpha(m,n)} \sum_{j\geq 1} A(i,j)/2^{A(i,j)-1}.
\end{aligned}$$

Sea $t = A(i, j)$. Se tiene

$$\begin{aligned}
&\sum_{i=1}^{\alpha(m,n)} \sum_{j\geq 1} n_{ij}(b_{ij}-1) \\
&\quad \leq n \sum_{i=1}^{\alpha(m,n)} \sum_{\substack{t=A(i,j) \\ j\geq 1}} t/2^{t-1} \\
&\quad = n \sum_{i=1}^{\alpha(m,n)} ((A(i,1)/2^{A(i,1)-1}) + (A(i,2)/2^{A(i,2)-1}) + \cdots) \\
&\quad \leq n \sum_{i=1}^{\alpha(m,n)} ((A(i,1)/2^{A(i,1)-1}) + ((A(i,1)+1)/2^{A(i,1)}) \\
&\qquad + ((A(i,1)+2)/2^{A(i,1)+1}) + \cdots).
\end{aligned}$$

Sea $a = A(i, 1)$. Así, se ha encontrado el valor de lo siguiente:

$$S_1 = (a)/2^{a-1} + (a+1)/2^a + (a+2)/2^{a+1} + \cdots \qquad \textbf{(10-3)}$$

$$2S_1 = a/2^{a-2} + (a+1)/2^{a-1} + (a+2)/2^a + \cdots \qquad \textbf{(10-4)}$$

Al tomar (10-4) y (10-3) se tiene

$$\begin{aligned}
S_1 &= a/2^{a-2} + (1/2^{a-1} + 1/2^a + 1/2^{a+1} + \cdots) \\
&= a/2^{a-2} + 1/2^{a-2} = (a+1)/2^{a-2}.
\end{aligned}$$

Así,

$$\begin{aligned} n & \sum_{i=1}^{\alpha(m,n)} ((A(i,1)/2^{A(i,1)-1}) + ((A(i,1)+1)/2^{A(i,1)}) \\ & + ((A(i,1)+2)/2^{A(i,1)+1}) + \cdots) \\ & = n \sum_{i=1}^{\alpha(m,n)} (A(i,1)+1)/2^{A(i,1)-2} \\ & \le n((A(1,1)+1)/2^{A(1,1)-2} + (A(2,1)+1)/2^{A(2,1)-2} \\ & \quad + (A(3,1)+1)/2^{A(3,1)-2} + \cdots) \\ & \le n((2+1)/2^{2-2} + (3+1)/2^{3-2} + (4+1)/2^{4-2} + \cdots) \\ & = n \sum_{t\ge 2} (t+1)/2^{t-2}. \end{aligned}$$

Resulta fácil demostrar que

$$\begin{aligned} \sum_{t\ge 2} (t+1)/2^{t-2} &= 3/2^{-1} + (1/2^0 + 1/2^1 + 1/2^2 + \cdots) \\ &= 6 + 2 \\ &= 8. \end{aligned}$$

Así, para la parte B se tiene

$$\sum_{i=1}^{\alpha(m,n)} \sum_{j\ge 1} n_{ij}(b_{ij} - 1) \le 8n.$$

Finalmente, para la parte C,

$$n_{\alpha(m,n)+1,0}(b_{\alpha(m,n)+1,0} - 1) \le n_{\alpha(m,n)+1,0} b_{\alpha(m,n)+1,0}. \qquad \textbf{(10-5)}$$

$$n_{\alpha(m,n)+1,0} \le n. \qquad \textbf{(10-6)}$$

$$\begin{aligned} & block(\alpha(m,n)+1,0) \\ & \quad = [0, \ldots, A(\alpha(m,n), \lfloor m/n \rfloor) - 1] \\ & \quad = [0, \ldots, A(\alpha(m,n),1), \ldots, A(\alpha(m,n),2), \ldots, A(\alpha(m,n), \lfloor m/n \rfloor) - 1]. \end{aligned}$$

Esto significa que $block(\alpha\lfloor m/n \rfloor + 1, 0)$ cubre bloques de nivel $m/n(i-1)$. Así,

$$b_{\alpha(m,n)+1,0} = \lfloor m/n \rfloor. \qquad \textbf{(10-7)}$$

Al sustituir (10-6) y (10-7) en (10-5), para la parte C se tiene

$$n_{\alpha(m,n)+1,0}(b_{\alpha(m,n)+1,0} - 1) \leq n\lfloor m/n \rfloor \leq m.$$

En resumen, para Q,

$$\begin{aligned} Q &\leq n\alpha(m, n) + 8n + m \\ &= (\alpha(m, n) + 8)n + m. \end{aligned}$$

Observe que el tiempo total gastado por las m operaciones *find* es igual a la suma del número de nodos crédito atravesados por las operaciones *find* y el número de nodos débito atravesados por las operaciones *find*. Sólo hay dos tipos de nodos: nodos crédito y nodos débito. El número de nodos débito atravesados está acotado por

$$(\alpha(m, n) + 2)m$$

y el número de nodos crédito atravesados está acotado por

$$(\alpha(m, n) + 8)n + m.$$

Así, se concluye que el tiempo medio utilizado por una operación *find* es

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

Debido a que $\alpha(m, n)$ es casi una constante, el análisis amortizado indica que el tiempo medio utilizado por una operación *find* es una constante.

10-7 ANÁLISIS AMORTIZADO DE ALGUNOS ALGORITMOS DE PROGRAMACIÓN DE DISCOS (DISK SCHEDULING)

El problema de programación de discos es un problema interesante e importante desde el punto de vista práctico en Ciencias de la Computación. Considere un simple disco. Los datos están almacenados en varias pistas. En todo momento hay una secuencia de peticiones para recuperar datos. Este conjunto de peticiones se denomina lista de espera y las peticiones se denominan peticiones en espera. El problema consiste en seleccionar una de las peticiones y atenderla.

Por ejemplo, suponga que hay una secuencia de peticiones para recuperar datos en las pistas 16, 2, 14, 5, 21, respectivamente, y que la cabeza del disco inicialmente está en la pista 0. También suponga que el tiempo solicitado para moverse de la pista i a la pista j es $|i - j|$. A continuación se mostrará cómo distintos algoritmos de programación de discos producen resultados diferentes.

Primero se considera el algoritmo primero en salir-primero en servir (FCFS: first-come first-serve). La cabeza del disco viene primero a la pista 16, luego a la 2, luego a la 14 y así sucesivamente. El tiempo total usado para atender estas peticiones es $|0 - 16| + |16 - 2| + |2 - 14| + |14 - 5| + |5 - 21| = 16 + 14 + 12 + 9 + 16 = 67$.

Suponga que se usa otro algoritmo, denominado algoritmo búsqueda más corta primero (SSTF: shortest-seek-time-first). En este algoritmo, la cabeza del disco siempre se desplaza a la pista más cercana. Así, primero se mueve a la pista 2, luego a la 5, luego a la 4 y así sucesivamente. Así, el tiempo total usado para atender estas peticiones es $|0 - 2| + |2 - 5| + |5 - 14| + |14 - 16| + |16 - 21| = 2 + 3 + 9 + 2 + 5 = 21$.

En esta sección se supone que a medida que se atienden las peticiones, siguen llegando nuevas. Se supondrá que las peticiones en espera constan de m peticiones. Cualquier petición fuera de estas m peticiones es ignorada. En otras palabras, el número máximo de peticiones consideradas es m. Por otra parte, se supone que hay por lo menos W peticiones, donde $W \geq 2$. El número total de pistas es $Q + 1$. Sea t_i el tiempo que se utiliza en el i-ésimo servicio. Esencialmente, se tiene interés en calcular $\sum_{i=1}^{m} t_i$. Como es de esperar, se encontrará una cota superior para $\sum_{i=1}^{m} t_i$. Como ya se hizo, sea $a_i(x) = t_i(x) + \phi_i(x) - \phi_{i-1}(x)$, donde x denota un algoritmo particular, ϕ_i denota el potencial después del i-ésimo servicio de la petición y $a_i(x)$ denota el tiempo amortizado del i-ésimo servicio.

Se tiene

$$\begin{aligned}\sum_{i=1}^{m} a_i(x) &= \sum_{i=1}^{m} (t_i(x) - \phi_i(x) - \phi_{i-1}(x)) \\ &= \sum_{i=1}^{m} t_i(x) + \phi_m(x) - \phi_0(x) \\ \sum_{i=1}^{m} t_i(x) &= \sum_{i=1}^{m} a_i(x) + \phi_0(x) - \phi_m(x).\end{aligned}$$

Nuestra estrategia para encontrar una cota superior de $\sum_{i=1}^{m} t_i(x)$ consiste en encontrar una cota superior de $a_i(x)$. Sea esta cota superior $A(x)$. Entonces

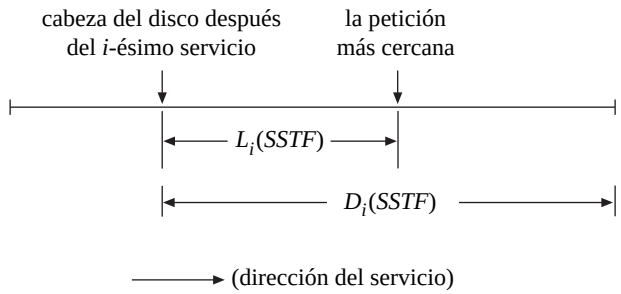
$$\sum_{i=1}^{m} t_i(x) \leq mA(x) + \phi_0(x) - \phi_m(x).$$

En el resto del apartado se analizará cómo puede usarse el análisis amortizado para estudiar los desempeños de dos algoritmos de programación de discos: el algoritmo SSTF y el algoritmo SCAN.

Análisis del algoritmo shortest-seek-time-first (SSTF: búsqueda más corta primero)

En este algoritmo se atiende la petición más cercana en la lista de espera. Considere la situación después del *i*-ésimo servicio, $1 \leq i \leq m$. Sea $N_i(SSTF)$ el número de peticiones en espera que están en la dirección del servicio presente. Sean $L_i(SSTF)$ la distancia entre la cabeza del disco y la petición más cercana y $D_i(SSTF)$ el número de pistas en la dirección de servicio. Las definiciones de $L_i(SSTF)$ y $D_i(SSTF)$ se muestran en la figura 10-50.

FIGURA 10-50 Definiciones de $L_i(SSTF)$ y $D_i(SSTF)$.



La función potencial para el algoritmo *SSTF* puede definirse como sigue:

$$\phi_i(SSTF) = \begin{cases} L_i(SSTF) & \text{si } N_i(SSTF) = 1 \\ \min\{L_i(SSTF), D_i(SSTF)/2\} & \text{si } N_i(SSTF) > 1. \end{cases} \quad \textbf{(10-8)}$$

Con base en (10-8), es fácil demostrar que

$$\phi_i(SSTF) \geq 0. \quad \textbf{(10-9)}$$

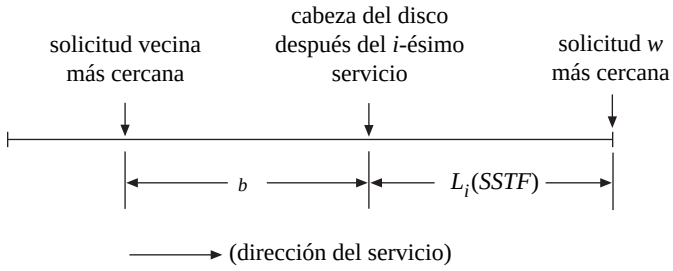
A continuación se demostrará que

$$\phi_i(SSTF) \leq Q/2. \quad \textbf{(10-10)}$$

Se consideran dos casos:

Caso 1. $N_i(SSTF) = 1$.
Debido a que $W \geq 2$, como se supuso, hay por lo menos una solicitud en la dirección opuesta, que se observa en la figura 10-51. Sea b la distancia entre la cabeza del disco y la solicitud vecina más cercana en la dirección opuesta. Entonces $b + L_i(SSTF) \leq Q$. No obstante, $L_i(SSTF) \leq b$; en caso contrario, se invierte la dirección de servicio.

FIGURA 10-51 El caso de $N_i(SSTF) = 1$.



En consecuencia,

$$2L_i(SSTF) \leq L_i(SSTF) + b \leq Q.$$

O bien,

$$L_i(SSTF) \leq Q/2.$$

Esto significa que

$$\phi_i(SSTF) = L_i(SSTF) \leq Q/2.$$

debido a las definiciones de $\phi_i(SSTF)$ según se indica en (10-8).

Caso 2. $N_i(SSTF) \geq 2$.
Debido a que $D_i(SSTF) \leq Q$, $D_i(SSTF)/2 \leq Q/2$. En consecuencia, según (10-8),

$$\begin{aligned}\phi_i(SSTF) &= \min\{L_i(SSTF),\ D_i(SSTF)/2\} \\ &\leq D_i(SSTF)/2 \\ &\leq Q/2\end{aligned}$$

Resulta fácil encontrar otra cota superior de $\phi_i(SSTF)$; a saber,

$$\phi_i(SSTF) \leq L_i(SSTF). \tag{10-11}$$

Las ecuaciones anteriores se usarán para encontrar una cota superior de

$$a_i(SSTF) = t_i(SSTF) + \phi_i(SSTF) - \phi_{i-1}(SSTF).$$

Para hacer lo anterior se considerarán los dos casos siguientes:

Caso 1. $N_{i-1}(SSTF) = 1$
En este caso, por (10-8) se tiene

$$\phi_{i-1}(SSTF) = L_{i-1}(SSTF) = t_i(SSTF). \tag{10-12}$$

Así,

$$\begin{aligned} a_i(SSTF) &= t_i(SSTF) + \phi_i(SSTF) - \phi_{i-1}(SSTF) \\ &= t_i(SSTF) + \phi_i(SSTF) - L_{i-1}(SSTF) \\ &\leq t_i(SSTF) + Q/2 - t_i(SSTF) \qquad \text{(por (10−10) y (10−12))} \\ &= Q/2. \end{aligned}$$

Caso 2. $N_{i-1}(SSTF) \geq 2$.
De nuevo se tienen dos subcasos.

Caso 2.1 $N_{i-1}(SSTF) \geq 2$ y $L_{i-1}(SSTF) \leq D_{i-1}(SSTF)/2$.
En este caso, según (10-8),

$$\phi_{i-1}(SSTF) = \min\{L_{i-1}(SSTF), D_{i-1}(SSTF)/2\} \leq L_{i-1}(SSTF).$$

Así, puede demostrarse, como se hizo en el caso 1, que

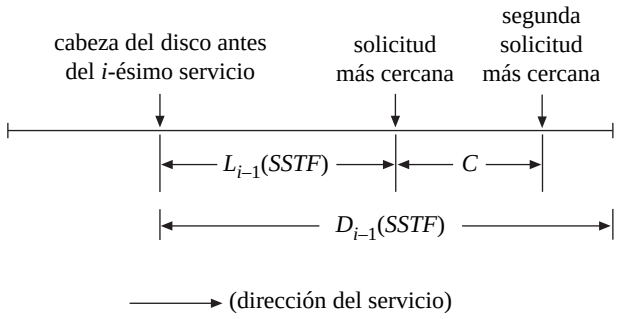
$$a_i(SSTF) \leq Q/2.$$

Caso 2.2 $N_{i-1}(SSTF) > 1$ y $L_i(SSTF) > D_{i-1}(SSTF)/2$.
En este caso, según (10-9),

$$\phi_{i-1}(SSTF) = D_{i-1}(SSTF)/2.$$

Debido a que $N_{i-1}(SSTF) > 1$, en la dirección de servicio debe haber algunas solicitudes, diferentes a la más cercana. Sea c la distancia entre la solicitud más cercana y la segunda solicitud más próxima en la dirección de servicio. En la figura 10-52 se describe esta situación.

FIGURA 10-52 La situación para $N_{i-1}(SSTF) > 1$ y $L_{i-1}(SSTF) > D_{i-1}(SSTF)/2$.



Debido a que se usa el algoritmo *SSTF*, $L_i(SSTF) \leq c$. Así, se tiene

$$L_i(SSTF) \leq c \leq D_{i-1}(SSTF) - L_{i-1}(SSTF). \quad \textbf{(10-13)}$$

Entonces

$$\begin{aligned}
a_i(SSTF) &= t_i(SSTF) + \phi_i(SSTF) - \phi_{i-1}(SSTF) \\
&\leq t_i(SSTF) + L_i(SSTF) - \phi_{i-1}(SSTF) \quad \text{(por (10-11))} \\
&\leq t_i(SSTF) + (D_{i-1}(SSTF) - L_{i-1}(SSTF)) - \phi_{i-1}(SSTF) \quad \text{(por (10-13))} \\
&= L_{i-1}(SSTF) + D_{i-1}(SSTF) - L_{i-1}(SSTF) - \phi_{i-1}(SSTF) \\
&\qquad \text{(debido a que } t_i(SSTF) = L_{i-1}(SSTF)\text{)} \\
&= D_{i-1}(SSTF) - \phi_{i-1}(SSTF) \\
&= D_{i-1}(SSTF)/2 \quad \text{(debido a que } \phi_{i-1}(SSTF) = D_{i-1}(SSTF)/2\text{)} \\
&\leq Q/2. \qquad \text{(debido a que } D_{i-1}(SSTF) \leq Q\text{)}
\end{aligned}$$

Al combinar los casos 1 y 2, se tiene

$$a_i(SSTF) \leq Q/2. \quad \textbf{(10-14)}$$

Al combinar (10-14), (10-9) y (10-10), es posible encontrar una cota superior de

$$\sum_{i=1}^{m} t_i(SSTF):$$

$$\sum_{i=1}^{m} t_i(SSTF) \le \sum_{i=1}^{m} a_i(SSTF) + \phi_0(SSTF) - \phi_m(SSTF).$$

Debido a que el máximo de $\phi_0(SSTF)$ es $Q/2$ y el mínimo de $\phi_m(SSTF)$ es 0, se tiene

$$\begin{aligned}\sum_{i=1}^{m} t_i(SSTF) &\le \sum_{i=1}^{m} a_i(SSTF) + \phi_0(SSTF) - \phi_m(SSTF)\\ &\le mQ/2 + Q/2 - 0\\ &= (m+1)Q/2.\end{aligned} \qquad \textbf{(10-15)}$$

La ecuación (10-15) indica que para el algoritmo *SSTF*, el tiempo total consumido por m solicitudes es $(m + 1)Q/2$. O bien, de manera equivalente, el tiempo medio consumido por una solicitud para este algoritmo es aproximadamente $Q/2$.

Para demostrar que ya no es posible restringir aún más la cota superior $(m + 1)Q/2$, se considerará una secuencia de m solicitudes que se encuentran respectivamente en pistas como sigue:

$$(Q/2,\ Q,\ (0,\ Q/2)^{(m-3)/2},\ 0)$$

donde x^y significa $\overbrace{(x, x, \ldots, x)}^{y}$. Suponga que el número de solicitudes en espera es 2 en cualquier instante y que $(m - 3)$ es divisible entre 2. En este caso, la cabeza del disco oscila entre la pista $Q/2$ y 0 como sigue:

$$((Q/2,\ 0)^{(m-1)/2},\ Q).$$

Así, el tiempo total de servicio de esta solicitud es

$$(Q/2 + Q/2)(m - 1)/2 + Q = (m + 1)Q/2.$$

Esto significa que ya no es posible restringir más la cota superior expresada en (10-15).

Análisis amortizado del algoritmo SCAN

Como puede verse en el análisis previo, el algoritmo SSTF puede obligar a que la cabeza del disco oscile. El algoritmo SCAN evita este problema al escoger la solicitud más cercana en la dirección de intercambio presente. Entonces, si la cabeza del disco

se mueve en una dirección, debe continuar haciéndolo hasta que en esta dirección no haya ninguna solicitud más. Entonces la cabeza del disco cambia de dirección.

Considere la situación después del i-ésimo servicio. A continuación se definen dos términos, $N_i(SCAN)$ y $D_i(SCAN)$ como sigue: Si la dirección de intercambio no cambia en el $(i + 1)$-ésimo servicio, entonces $N_i(SCAN)$ y $D_i(SCAN)$ se definen como: el número de peticiones a las que se ha servido, y la distancia que pudo haberse movido la cabeza del disco en el intercambio actual; en caso contrario, tanto $N_i(SCAN)$ como $D_i(SCAN)$ se igualan a cero. $N_0(SCAN)$ y $D_0(SCAN)$ son cero. La función potencial para SCAN puede definirse como

$$\phi_i(SCAN) = N_i(SCAN)Q/W - D_i(SCAN). \quad \textbf{(10-16)}$$

Para SSTF, se demostró que $a_i(SSTF) \leq Q/2$. Para SCAN, se demostrará que $a_i(SCAN) \leq Q/W$.

Considere el i-ésimo servicio y sea $t_i(SCAN)$ el tiempo para ejecutar el i-ésimo servicio. Nuevamente hay dos casos a considerar.

Caso 1. El $(i + 1)$-ésimo servicio no modifica la dirección de intercambio. Resulta evidente que, en este caso, se tiene

$$\begin{aligned}
N_i(SCAN) &= N_{i-1}(SCAN) + 1 \text{ y} \\
D_i(SCAN) &= D_{i-1}(SCAN) + t_i(SCAN) \\
a_i(SCAN) &= t_i(SCAN) + \phi_i(SCAN) - \phi_{i-1}(SCAN) \\
&= t_i(SCAN) + (N_i(SCAN)Q/W - D_i(SCAN)) \\
&\quad - (N_{i-1}(SCAN)Q/W - D_{i-1}(SCAN)) \\
&= t_i(SCAN) + ((N_{i-1}(SCAN) + 1)Q/W - (D_{i-1}(SCAN) \\
&\quad + t_i(SCAN)) - (N_{i-1}(SCAN)Q/W - D_{i-1}(SCAN)) \\
&= Q/W.
\end{aligned}$$

Caso 2. La dirección de intercambio se modifica a partir del $(i + 1)$-ésimo servicio. En este caso, $N_i(SCAN) = D_i(SCAN)$ y $\phi_i(SCAN) = 0$.

Debido a que la cabeza del disco inicialmente estaba en la pista 0 y a que por hipótesis el mínimo de la secuencia en espera es W, entonces el mínimo de solicitudes servidas en un intercambio también es W. Es decir, $N_{i-1}(SCAN) > (W - 1)$. Así,

$$\begin{aligned}
a_i(SCAN) &\leq t_i(SCAN) - ((W - 1)Q/W - D_{i-1}(SCAN)) \\
&\leq (t_i(SCAN) + D_{i-1}(SCAN) - Q) + Q/W.
\end{aligned}$$

Debido a que la distancia máxima que puede moverse el disco en un intercambio es Q,

$$t_i(SCAN) + D_{i-1}(SCAN) \leq Q.$$

En consecuencia,

$$a_i(SCAN) \leq Q/W. \tag{10-17}$$

El análisis anterior muestra que

$$a_i(SCAN) \leq A(SCAN) = Q/W,$$

donde $A(SCAN)$ es una cota superior de $a_i(SCAN)$.

Ahora se tiene

$$\begin{aligned}\sum_{i=1}^{m} t_i(SCAN) &\leq \sum_{i=1}^{m} a_i(SCAN) + \phi_0(SCAN) - \phi_m(SCAN) \\ &\leq mQ/W + \phi_0(SCAN) - \phi_m(SCAN).\end{aligned}$$

Pero $\phi_0(SCAN) = 0$. Así,

$$\sum_{i=1}^{m} t_i(SCAN) \leq mQ/W - \phi_m(SCAN).$$

Para estimar $\phi_m(SCAN)$ nuevamente pueden considerarse dos casos:

Caso 1. $N_m(SCAN) = D_m(SCAN) = 0.$ (10-18)

En este caso, $\phi_m(SCAN) = 0$.

Caso 2. Uno de $N_m(SCAN)$ y $D_m(SCAN)$ es diferente de cero.
En este caso, $N_m(SCAN) \geq 1$ y $D_m(SCAN) \leq Q$.
Así,

$$\begin{aligned}\phi_m(SCAN) &= N_m(SCAN)Q/W - D_m(SCAN) \\ &\geq Q/W - Q.\end{aligned} \tag{10-19}$$

Se combinarán (10-18) y (10-19) observando que

$$Q/W - Q \leq 0.$$

Se concluye que

$$\phi_m(SCAN) \geq Q/W - Q. \tag{10-20}$$

Finalmente, se tiene

$$\begin{aligned}\sum_{i=1}^{m} t_i(SCAN) &\leq mQ/W - \phi_m(SCAN)\\ &= mQ/W - (Q/W - Q)\\ &\leq (m-1)Q/W + Q.\end{aligned} \tag{10-21}$$

La fórmula (10-21) indica que el tiempo total consumido por estas m solicitudes no puede ser mayor que $(m-1)Q/W + Q$. El tiempo medio consumido por una solicitud para el algoritmo SCAN no es mayor que $((m-1)Q/W + Q)/m$.

Como se hizo antes, puede demostrarse que ya no es posible restringir más la cota superior $(m-1)Q/W + Q$. Esto se demuestra al considerar una secuencia de m solicitudes, que se encuentran respectivamente en las pistas

$$((Q^4, 0^4)^{(m-1)/8}, Q)$$

y se supone que el número de solicitudes en espera es 4 en cualquier instante. Sea $W = 4$. Suponga que $(m-1)$ es divisible entre 8. Entonces, esta secuencia de solicitudes también debe programarse y procesarse por la secuencia

$$((Q^4, 0^4)^{(m-1)/8}, Q).$$

El tiempo total de servicio para esta solicitud es

$$((m-1)/8)2Q + Q = ((m-1)/(2W))2Q + Q = (m-1)Q/W + Q.$$

Lo anterior muestra que ya no es posible restringir más la cota superior del tiempo total expresada en (10-21).

Ahora, este apartado concluye con la comparación de (10-15) con (10-21). Con base en (10-15) se tiene

$$t_{prom}(SSTF) = \sum_{i=1}^{m} t_i(SSTF)/m \leq (m+1)Q/(2m). \tag{10-22}$$

Cuando $m \to \infty$ se tiene

$$t_{prom}(SSTF) \leq Q/2. \tag{10-23}$$

Con base en (10-21) puede demostrarse que

$$t_{prom}(SCAN) \leq Q/W. \tag{10-24}$$

Debido a que $W \geq 2$, es posible concluir que

$$t_{prom}(SCAN) \leq t_{prom}(SSTF)$$

mediante el análisis amortizado de estos dos algoritmos.

10-8 LOS RESULTADOS EXPERIMENTALES

El análisis amortizado a menudo implica mucha manipulación matemática. Nunca es intuitivamente claro para muchos investigadores que así debe ser el resultado. Por ejemplo, no es en absoluto evidente para una persona común y corriente que el desempeño promedio del algoritmo de la unión de conjuntos disjuntos es casi constante. En consecuencia, se implementó el algoritmo de la unión de conjuntos disjuntos. Se usó un conjunto de 10 000 elementos. El que la siguiente instrucción sea *find* o link está determinada por un generador de números aleatorios. El programa se escribió en Turbo Pascal y se ejecutó en una PC IBM. En la tabla 10-9 se resumen los resultados.

TABLA 10-9 Resultados experimentales del análisis amortizado del algoritmo para la unión de conjuntos disjuntos.

Núm. de operaciones	Tiempo total (msg)	Tiempo promedio (μsg)
2 000	100	50
3 000	160	53
4 000	210	53
5 000	270	54
7 000	380	54
9 000	490	54
11 000	600	55
13 000	710	55
14 000	760	54
15 000	820	55

Con base en los resultados experimentales puede verse que el análisis amortizado pronostica con seguridad el comportamiento del algoritmo para la unión de conjuntos disjuntos. Para una secuencia de operaciones, el tiempo total implicado aumenta. No obstante, el tiempo promedio por operación es una constante que es pronosticada por el análisis amortizado. Para una secuencia de operaciones, el tiempo total aumenta. Sin embargo, el tiempo medio por operación es una constante pronosticada por el análisis amortizado.

10-9 NOTAS Y REFERENCIAS

La expresión análisis amortizado apareció por primera vez en Tarjan (1985). No obstante, ya antes de 1985 este concepto era utilizado de manera implícita por los investigadores. Por ejemplo, el análisis de árboles 2-3 en Brown y Tarjan (1980) y de los árboles-B débiles (weak B-trees) en Huddleston y Melhorn (1982) ya utiliza este concepto, aunque entonces no se usaba "amortizado".

El análisis amortizado de heaps sesgados puede encontrarse en Sleator y Tarjan (1986). Mehlhorn y Tsakalidis (1986) proporcionaron un análisis amortizado del árbol-AVL. El análisis amortizado de heurísticas de búsqueda secuencial autoorganizada apareció en Bentley y McGeoch (1985). Para el análisis amortizado de heaps apareados, consulte Fredman, Sedgewick, Sleator y Tarjan (1986). El análisis amortizado del algoritmo para la unión de conjuntos disjuntos puede encontrarse en Tarjan (1983) y en Tarjan y Van Leeuwen (1984).

El análisis amortizado suele usarse dentro de un algoritmo que implica una estructura de datos pertinente y una secuencia de operaciones aplicadas repetidamente sobre esta estructura de datos. Por ejemplo, en Fu y Lee (1991), se requiere una estructura de datos de modo que los árboles pueden manipularse dinámica y eficientemente. Es decir, se ejecutarán muchas operaciones con árboles, como la revisión de un borde, la inserción de un borde, etc. En un caso así, es posible usar el árbol dinámico (Sleator y Tarjan, 1983) y el análisis es un análisis amortizado.

El análisis amortizado puede aplicarse para analizar algunas estrategias prácticas para una secuencia de operaciones. Chen, Yang y Lee (1989) aplicaron análisis amortizado para estudiar algunas políticas de programación de discos.

Muy pocos libros de texto abordan el concepto de análisis amortizado, aunque en la obra de Purdom y Brown (1985a) y en la de Tarjan (1983) puede encontrarse una discusión sobre el análisis amortizado.

10-10 BIBLIOGRAFÍA ADICIONAL

Para un repaso y discusión introductorios del análisis amortizado, consulte Tarjan (1985). Todos los siguientes elementos son de reciente aparición y se recomiendan

para profundizar el conocimiento sobre el tema: Bent, Sleator y Tarjan (1985); Eppstein, Galil, Italiano y Spencer (1996); Ferragina (1997); Henzinger (1995); Italiano (1986); Karlin, Manasse, Rudolph y Sleator (1988); Kingston (1986); Makinen (1987); Sleator y Tarjan (1983); Sleator y Tarjan (1985a); Sleator y Tarjan (1985b); Tarjan y Van Wyk (1988), y Westbrook y Tarjan (1989).

Ejercicios

10.1 Para el problema de la pila que se abordó en el apartado 10-1, demuestre que la cota superior es 2 sin usar una función potencial. Proporcione un ejemplo para demostrar que esta cota superior también está restringida.

10.2 Suponga que hay una persona cuyo único ingreso es su salario mensual, que es k unidades al mes. Esta persona puede, no obstante, gastar cualquier cantidad de dinero en la medida en que tenga suficientes fondos en su cuenta bancaria. La persona deposita mensualmente k unidades de su ingreso. ¿Puede realizar un análisis amortizado sobre este comportamiento? (Defina su propio problema. Observe que la persona no puede retirar grandes cantidades de dinero todo el tiempo.)

10.3 El análisis amortizado implica de alguna manera que la estructura de datos implicada cuente con algún mecanismo autoorganizativo. En otras palabras, cuando se vuelve demasiado mala, hay una posibilidad de que después se vuelva buena. En este sentido, ¿puede analizarse la técnica de hashing usando análisis amortizado? Investigue este tema. Quizá llegue a publicar algunos artículos.

10.4 Seleccione algún algoritmo presentado en este capítulo e impleméntelo. Realice algunos experimentos para ver si tiene sentido el análisis amortizado.

10.5 Lea la obra de Sleator y Tarjan (1983) sobre las estructuras de datos de un árbol dinámico.

capítulo

11

Algoritmos aleatorios

El concepto de algoritmos aleatorios (o probabilísticos) es relativamente nuevo. En todos los algoritmos presentados hasta ahora, cada paso del algoritmo es determinístico. Es decir, nunca, en medio de la ejecución de un algoritmo, se hace una elección arbitraria. En los algoritmos aleatorios, que se presentarán en este capítulo, se hacen decisiones arbitrarias; o lo que es lo mismo, algunas acciones se realizan al azar.

Debido a que algunas acciones se ejecutan al azar, los algoritmos aleatorios que se presentan más adelante poseen las siguientes propiedades:

1. En el caso de problemas de optimización, un algoritmo aleatorio proporciona una solución óptima. No obstante, como en el algoritmo se realizan acciones al azar, la complejidad temporal de un algoritmo de optimización aleatorio también es aleatoria. Entonces, la complejidad temporal del caso promedio de un algoritmo de optimización aleatorio es más importante que su complejidad temporal en el peor caso.
2. En cuanto a los problemas de decisión, un algoritmo aleatorio comete errores de vez en cuando. Sin embargo, puede afirmarse que la probabilidad de producir soluciones erróneas es excesivamente pequeña; en caso contrario, el algoritmo aleatorio no es de utilidad.

11-1 Un algoritmo aleatorio para resolver el problema del par más cercano

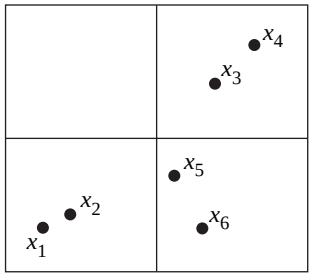
El problema del par más cercano se abordó en el capítulo 4. Quedó demostrado que puede resolverse con el método divide y vencerás en tiempo $O(n \log n)$. Esta complejidad temporal es para los peores casos. En esta sección se demostrará que existe un algoritmo aleatorio y que la complejidad temporal de usar este algoritmo para resolver el problema del par más cercano es $O(n)$.

Sean $x_1, x_2, \ldots, x_n$ n puntos en el plano bidimensional. El problema del par más cercano consiste en encontrar el par más cercano x_i y x_j de modo que la distancia entre x_i y x_j sea la más corta de todas las distancias entre todos los pares de puntos posibles. Un método directo para resolver este problema es evaluar todas las distancias $n(n-1)/2$ y encontrar su mínimo.

La idea principal de un algoritmo aleatorio se basa en la siguiente observación: *Si dos puntos x_i y x_j son distantes entre sí, entonces probablemente su distancia no es la más corta, de modo que es posible ignorarla.* Con esta idea, el algoritmo aleatorio primero hace una partición de los puntos en varios grupos acumulados de modo que los puntos en cada acumulación están próximos entre sí. Sólo entonces se calculan las distancias entre puntos del mismo grupo de acumulación.

Considere la figura 11-1. Como puede verse en esa figura, hay seis puntos. Si estos puntos se separan en los grupos (o clusters) $S_1 = \{x_1, x_2\}$, $S_2 = \{x_5, x_6\}$ y $S_3 = \{x_3, x_4\}$, entonces sólo se calculan tres distancias; a saber, $d(x_1, x_2)$, $d(x_5, x_6)$ y $d(x_3, x_4)$. Luego de hacer lo anterior se encuentra el mínimo de estas tres distancias. En caso de que no se separe en grupos, es necesario calcular $(6 \cdot 5)/2 = 15$ distancias.

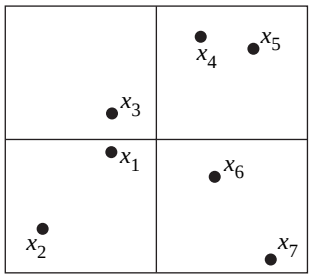
FIGURA 11-1 Partición de los puntos.



Por supuesto, este análisis es bastante engañoso porque no hay garantía de que la estrategia funcione. De hecho, la estrategia puede considerarse como una estrategia "divide sin vencer" (divide-without-conquer). Hay un proceso de división, pero no uno de fusión. Se considerará la figura 11-2. Ahí puede verse que el par más cercano es $\{x_1, x_3\}$. No obstante, estos puntos se han dividido en dos grupos distintos.

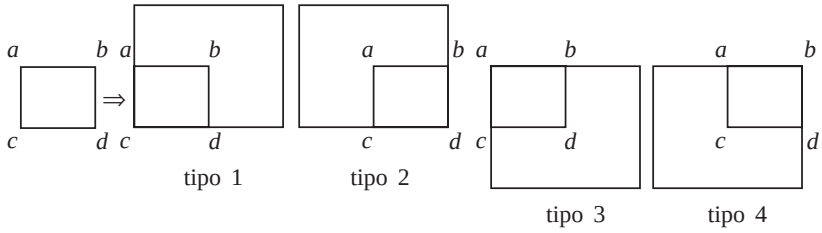
Si la partición es tal que todo el espacio se divide en cuadrados cuyos lados miden δ, que no es menor que la distancia más corta, entonces después de calcular las distancias entre todos los puntos que se encuentran dentro del grupo, es posible duplicar la

FIGURA 11-2 Caso para mostrar la importancia de las distancias entre grupos.



longitud del lado del cuadrado y obtener cuadrados más grandes. La distancia más corta debe estar dentro de uno de estos cuadrados agrandados. En la figura 11-3 se muestra cómo cuatro cuadrados agrandados corresponden a un cuadrado original. Cada cuadrado agrandado pertenece a cierto tipo, lo cual se indica en la figura 11-3.

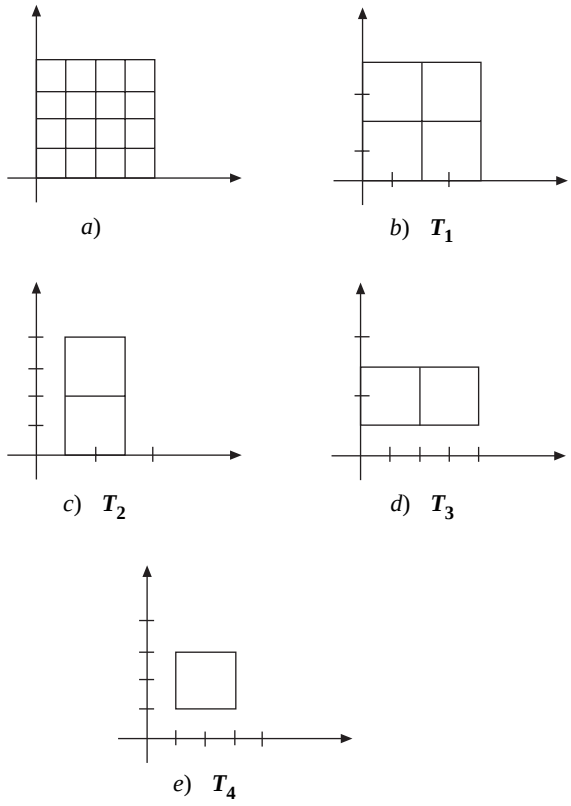
FIGURA 11-3 Obtención de cuatro cuadrados agrandados.



Suponga que todo el espacio ya se ha dividido en cuadrados de lado δ. Entonces el agrandamiento de estos cuadrados induce cuatro conjuntos de cuadrados agrandados, denotados por T_1, T_2, T_3 y T_4, que corresponden a los cuadrados tipo 1, tipo 2, tipo 3 y tipo 4, respectivamente. En la figura 11-4 se ilustra un caso típico.

Por supuesto, la cuestión crítica consiste en encontrar el tamaño de malla (o retícula) idóneo δ. Si δ es demasiado grande, el cuadrado original es muy grande y entonces es necesario calcular un gran número de distancias. De hecho, si δ es muy grande, entonces casi no hay división y el problema se convierte en el problema original. Por otra parte, δ no puede ser demasiado pequeño porque no es posible que sea menor que la distancia más corta. En el algoritmo aleatorio, se selecciona al azar un subconjunto de puntos y se encuentra la distancia más corta en este subconjunto de puntos. La distancia más corta se convierte en la δ buscada.

FIGURA 11-4 Cuatro conjuntos de cuadrados agrandados.



Algoritmo 11-1 □ Un algoritmo aleatorio para encontrar un par más cercano

Input: Un conjunto S integrado por n elementos $x_1, x_2, \ldots, x_n$, donde $S \subseteq R^2$.

Output: El par más cercano en S.

Paso 1. Escoger aleatoriamente un conjunto $S_1 = \{x_{i1}, x_{i2}, \ldots, x_{im}\}$, donde $m = n^{\frac{2}{3}}$. Encontrar el par más cercano de S_1 y denotar por δ la distancia entre este par de puntos.

Paso 2. Construir un conjunto de cuadrados T con tamaño de malla δ.

Paso 3. Construir cuatro conjuntos de cuadrados T_1, T_2, T_3 y T_4 obtenidos a partir de T al duplicar el tamaño de malla a 2δ.

Paso 4. Para todo T_i, encontrar la descomposición inducida $S = S_1^{(i)} \cup S_2^{(i)} \cup \cdots \cup S_j^{(i)}$, $1 \leq i \leq 4$, donde $S_j^{(i)}$ es una intersección no vacía de S con un cuadrado de T_i.

Paso 5. Para todo $x_p, x_q, \in S_j^{(i)}$, calcular $d(x_p, x_q)$. Sean x_a y x_b el par de puntos con la distancia más corta de estos pares. Regresar x_a y x_b como el par más cercano.

Ejemplo 11-1

Se tiene un conjunto S de 27 puntos que se muestra en la figura 11-5. En el Paso 1, aleatoriamente se escogen $27^{\frac{2}{3}} = 9$ elementos $x_1, x_2, \ldots, x_9$. Puede verse que el par más cercano es (x_1, x_2). Luego se usa la distancia δ entre x_1 y x_2 como el tamaño de la malla para construir 36 cuadrados, según requiere el Paso 2. Hay cuatro conjuntos de cuadrados T_1, T_2, T_3 y T_4:

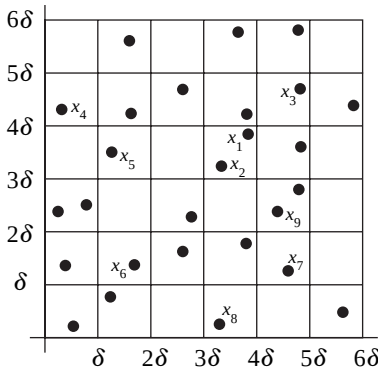
$$T_1 = \{[0 : 2\delta, 0 : 2\delta], [2\delta : 4\delta, 0 : 2\delta], [4\delta : 6\delta, 0 : 2\delta], \ldots, [4\delta : 6\delta, 4\delta : 6\delta]\}.$$

$$T_2 = \{[\delta : 3\delta, 0 : 2\delta], [3\delta : 5\delta, 0 : 2\delta] \ldots, [3\delta : 5\delta, 4\delta : 6\delta]\}.$$

$$T_3 = \{[0 : 2\delta, \delta : 3\delta], [2\delta : 4\delta, \delta : 3\delta], [4\delta : 6\delta, \delta : 3\delta], \ldots, [4\delta : 6\delta, 3\delta : 5\delta]\}.$$

$$T_4 = \{[\delta : 3\delta, \delta : 3\delta], [3\delta : 5\delta, \delta : 3\delta], \ldots, [3\delta : 5\delta, 3\delta : 5\delta]\}.$$

FIGURA 11-5 Ejemplo que ilustra el algoritmo aleatorio del par más cercano.



El número total de cálculos de distancias mutuas es

$$N(T_1): \; C_2^4 + C_2^3 + C_2^2 + C_2^3 + C_2^3 + C_2^3 + C_2^3 + C_2^3 + C_2^3 = 28.$$

$$N(T_2): \; C_2^3 + C_2^3 + C_2^2 + C_2^5 + C_2^3 + C_2^4 = 26.$$

$$N(T_3): \; C_2^4 + C_2^3 + C_2^3 + C_2^3 + C_2^4 + C_2^3 = 24.$$

$$N(T_4): \; C_2^3 + C_2^4 + C_2^3 + C_2^5 = 22.$$

De los 28 + 24 + 22 + 26 = 100 pares, se encuentra que el más cercano está a menos de $[3\delta : 5\delta, 3\delta : 5\delta]$.

11.2 EL DESEMPEÑO PROMEDIO DEL PROBLEMA ALEATORIO DEL PAR MÁS CERCANO

En el problema aleatorio del par más cercano presentado en la sección previa, el primer paso consiste en encontrar el par más cercano de entre $n^{\frac{2}{3}}$ puntos. Tal par más cercano puede encontrarse al aplicar recurrentemente este algoritmo aleatorio. Es decir, al azar se escogen $(n^{\frac{2}{3}})^{\frac{2}{3}} = n^{\frac{4}{9}}$ puntos del conjunto original. Para encontrar el par más cercano de todos estos $n^{\frac{4}{9}}$ puntos, puede aplicarse el método directo que requiere $(n^{\frac{4}{9}})^2 = n^{\frac{8}{9}} < n$ cálculos de distancia. Sin embargo, esto no significa que el Paso 1 puede ejecutarse en tiempo $O(n)$. Por el momento se suspenderá este análisis de la complejidad temporal del Paso 1, aunque después se demostrará que, en efecto, la ejecución del Paso 1 requiere tiempo $O(n)$.

Resulta evidente que tanto el Paso 2 como el Paso 3 pueden llevarse a cabo en tiempo $O(n)$. El Paso 4 se ejecuta aplicando la técnica de hashing, con la que es fácil decidir cuáles puntos están en qué cuadrado. Esto significa que el Paso 4 puede efectuarse en tiempo $O(n)$.

El número esperado de cálculos de distancia en el Paso 5 no es en absoluto fácil de determinar. En realidad, no se cuenta con una fórmula para este número esperado de cálculos de distancias. De hecho, se demostrará que la probabilidad de que el número esperado de cálculos de distancias en el Paso 5 es $1 - 2e^{-cn^{\frac{1}{6}}}$, que tiende rápidamente a 1 cuando n crece. O bien, para plantearlo de otra forma, la probabilidad de que el número esperado de cálculos de distancias sea $O(n)$ es muy alta.

¿Por qué es posible concluir lo anterior? Se observa que en nuestro algoritmo aleatorio del par más cercano el tamaño de la malla es δ. Esta partición se denota por T y el número total de cálculos de distancias, por $N(T)$. Después se demostrará que existe una partición particular, denominada T_0, cuyo tamaño de malla es δ_0 con las propiedades siguientes:

1. $n \leq N(T_0) \leq C_0 n$.
2. La probabilidad de que $\delta < \sqrt{2}\,\delta_0$ es $1 - 2e^{-cn^{\frac{1}{6}}}$, la cual es muy alta.

Más tarde se demostrará por qué existe esta partición. Por ahora, simplemente se supondrá que este hecho es cierto y se procede a partir de ahí.

Imagine que el tamaño de malla se cuadruplica de δ_0 a $4\delta_0$. Así se inducen 16 conjuntos de cuadrados. Estos conjuntos se denotarán por T_i, $i = 1, 2, \ldots, 16$. Debido a que la probabilidad de que $\delta \leq \sqrt{2}\ \delta_0$ es $1 - 2e^{-cn^{\frac{1}{6}}}$, la probabilidad de que cada cuadrado en T caiga en por lo menos un cuadrado de T_i, $i = 1, 2, \ldots, 16$ es $1 - 2e^{-cn^{\frac{1}{6}}}$. Sea $N(T_i)$ el número total de cálculos de distancias para la partición T_i. Entonces la probabilidad de que

$$N(T) \leq \sum_{i=1}^{16} N(T_i)$$

sea verdadera es $1 - 2e^{-cn^{\frac{1}{6}}}$.

A continuación se calculará $N(T)$. Se empieza a partir de $N(T_i)$. Cada cuadrado en T_i es 16 veces mayor que un cuadrado en T_0. Considere que el cuadrado en T_0 con el mayor número de elementos tiene k elementos. Sea S_{ij} el conjunto de los 16 cuadrados en T_0 que pertenece al j-ésimo cuadrado de T_i. Considere que el cuadrado en S_{ij} con el mayor número de elementos tiene k_{ij} elementos. El número total de cálculos de distancias en T_0 es mayor que $\sum_j k_{ij}(k_{ij} - 1)/2$. Es decir, $\sum_j k_{ij}(k_{ij} - 1)/2 \leq N(T_0) \leq C_0 n$. Y el número total de cálculos de distancias en el j-ésimo cuadrado de T_i es menor que $16k_{ij}(16k_{ij} - 1)/2$. Así, $N(T_i) \leq \sum_j 16k_{ij}(16k_{ij} - 1)/2 \leq C_i n$, donde C_i es una constante. En consecuencia, $N(T) \leq \sum_{i=1}^{16} N(T_i) = O(n)$ con probabilidad $1 - 2e^{-cn^{\frac{1}{6}}}$. Cuando n es grande, $N(T) \leq O(n)$ con probabilidad 1, cuando $e^{-cn^{\frac{1}{6}}}$ tiende a cero rápidamente.

A continuación se explicará por qué puede concluirse que existe una partición con las dos propiedades mencionadas. El razonamiento es más bien complicado. Antes de proporcionar la línea de razonamiento principal, primero se definirán algunos términos.

Sea D una partición de un conjunto de puntos. Es decir, $S = S_1 \cup S_2 \cup \cdots \cup S_r$ y $S_i \cap S_j = \varnothing$ si $i \neq j$. Si $T \subseteq S$ es una elección de m elementos, entonces T se denomina éxito en D si por lo menos dos elementos de T se escogieron de la misma parte S_i de la partición para alguna i. Si D' es otra partición de S, se dice que D domina a D' si para toda m se tiene que la probabilidad de éxito en D con una elección de m elementos es mayor o igual al éxito en D' con una elección de m elementos.

Según la definición anterior, puede verse fácilmente que las siguientes afirmaciones son verdaderas:

1. La partición (2, 2, 2) domina a (3, 1, 1, 1). Esto significa que cualquier partición de un conjunto de seis elementos en tres pares domina a cualquier partición del mismo conjunto por una tripleta y tres sencillos. ¿Por qué domina? Es muy fácil comprender la razón. Para la partición (2, 2, 2), hay una probabilidad mucho mayor de que dos puntos se extraigan del mismo cuadrado que la partición (3, 1, 1, 1) porque sólo hay un cuadrado con un elemento.
2. La partición (3, 3) domina a (4, 1, 1).
3. La partición (4, 4) domina a (5, 1, 1, 1).
4. La partición (p, q), $p \geq 5$, $q \geq 5$ domina a $(\ell, 1, 1, \ldots, 1)$, donde el número de unos es $p + q - \ell$, $\ell(\ell - 1) \leq p(p - 1) + q(q - 1)\ell \leq (\ell + 1)$.

Puede verse que si la partición D domina a la partición D', entonces el número de cálculos de distancias en D' es menor o igual a los cálculos de distancias en D.

Sea $N(D)$ el número de cálculos de distancias requeridos en D. Según las afirmaciones y el análisis anteriores, resulta evidente que para toda partición D de cualquier conjunto finito S de puntos, existe otra partición D' del mismo conjunto tal que $\lambda N(D) \leq N(D')$, donde D domina a D' y todos los conjuntos de D' con una excepción están integrados por un elemento, donde λ es un número positivo menor o igual a 1.

Se supondrá que D es una partición del conjunto S, $|S| = n$ y $n \leq N(D)$. Sea D' una partición tal que $S = H_1 \bigcup H_2 \cdots \bigcup H_{k'}$ que es dominada por D, donde $|H_1| = b$, $|H_i| = 1$ para $i = 2, 3, \ldots, k'$ y $\lambda n \leq N(D')$. Esto implica que $\lambda n \leq b(b - 1)/2$. Así, se tiene $b \geq \sqrt{2\lambda n}$. Sea $c = \sqrt{2\lambda}$, se tiene $b \geq c\sqrt{n}$.

Para cada selección de puntos, la probabilidad de que este punto no se extraiga de H_1 es $1 - b/n \leq 1 - c/\sqrt{n}$. Suponga que de S se seleccionan aleatoriamente $n^{\frac{2}{3}}$ puntos. La probabilidad de que ninguno está en H_1 es menor que

$$\left(1-\frac{c}{\sqrt{n}}\right)^{n^{\frac{2}{3}}} = \left(\left(1-\frac{c}{\sqrt{n}}\right)^{\sqrt{n}}\right)^{n^{\frac{1}{6}}} = e^{-cn^{\frac{1}{6}}}$$

Cuando de S se seleccionan aleatoriamente $n^{\frac{2}{3}}$ *puntos, la probabilidad de que por lo menos dos de ellos se extraigan de* H_1 *es mayor que* $1-2e^{-cn^{\frac{1}{6}}}$. *Debido a que D domina a D′, puede concluirse que si los puntos se escogen aleatoriamente de S, en-*

tonces la probabilidad de que por lo menos dos puntos se extraigan del mismo conjunto de D es por lo menos $\mu(n) = 1 - 2e^{cn^{\frac{1}{6}}}$, *donde c es una constante.*

Sigue habiendo un problema: ¿Existe una partición T_0 tal que $n \leq N(T_0) \leq C_0 n$, donde C_0 es una constante? Esta partición puede encontrarse aplicando el siguiente algoritmo.

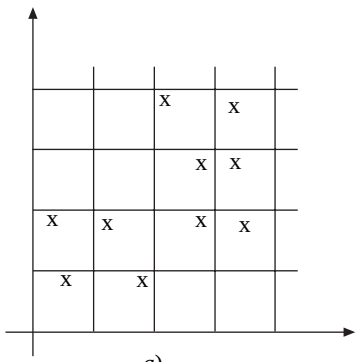
Algoritmo 11-2 □ Algoritmo de partición

Input: Un conjunto S integrado por n puntos.
Output: Una partición T_0 donde $n \leq N(T_0) \leq C_0 n$.
Paso 1. Encontrar una partición T con tamaño de malla suficientemente pequeño de modo que cada cuadrado en T contenga cuando mucho un punto de S y ningún punto de S esté sobre ninguna de las líneas de la malla. Así, $N(T) = 0$.
Paso 2. Mientras $N(T) < n$, duplicar el tamaño de la malla de T.

A continuación se explicará el significado del algoritmo 11-2. El primer paso de este algoritmo divide todo el conjunto de puntos en cuadrados de modo que cada cuadrado contiene a lo sumo un punto. Por supuesto, en este caso $N(T)$ es igual a 0 y no es de utilidad para nuestros fines. En consecuencia, gradualmente se duplica el tamaño del cuadrado hasta que se encuentra la primera partición tal que $N(T_0) \geq n$.

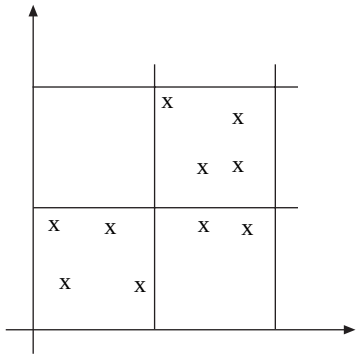
Considere la figura 11-6, donde $n = 10$. En la figura 11-6*a*) se muestra la partición inicial que es resultado del Paso 1 del algoritmo 11-1. Suponga que se duplica el tamaño de la malla. Se obtienen las particiones que se muestran en la figura 11-6*b*). Así, el número total de distancias calculadas es $N(T) = 4 \times 3/2 + 2 \times 1/2 + 4 \times 3/2 = 6 + 1 + 6 = 13 > n = 10$. Entonces, esta partición particular constituye una T_0 deseable. Observe que en este caso se tiene $C_0 = 1.3$.

FIGURA 11-6 Ejemplo que ilustra el algoritmo 11-1.



a)

FIGURA 11-6 (*continuación*)



b)

Sea δ_0 el tamaño de la malla de T_0. Para cualquier conjunto S_a con $n^{\frac{2}{3}}$ puntos escogidos al azar, si en S_a existen dos puntos que están dentro del mismo cuadrado de T_0, entonces la distancia más corta en S_a es menor o igual a $\sqrt{2}\delta_0$ y esta desigualdad se cumple con una probabilidad mayor que $\mu(n)$. En consecuencia, se ha demostrado que existe un T_0 crítico cuyo tamaño es δ_0 con las propiedades siguientes:

1. $n \leq N(T_0) \leq C_0 n$.
2. La probabilidad de que $\delta \leq \sqrt{2}\,\delta_0$ es $1 - 2e^{-cn^{\frac{1}{6}}}$, que es muy alta, donde δ es la distancia más corta en un conjunto de $n^{\frac{2}{3}}$ puntos elegidos al azar.

Así se ha explicado entonces que en el algoritmo 11-1 para ejecutar el Paso 5 se requieren $O(n)$ pasos. Observe que el Paso 1 es un paso recurrente. Sin embargo, como el Paso 5 es un paso dominante, puede concluirse que todo el algoritmo aleatorio del par más cercano es un algoritmo $O(n)$.

11-3 UN ALGORITMO ALEATORIO PARA PROBAR SI UN NÚMERO ES PRIMO

El problema del número primo consiste en determinar si un entero positivo es primo o no. Este problema es muy difícil y no fue sino hasta 2004 que se descubrió un algoritmo de tiempo polinomial para resolver este problema. En esta sección se presentará un algoritmo aleatorio. Este algoritmo aleatorio ejecuta una secuencia de m pruebas. Si alguna de ellas tiene éxito, se concluye que el número en cuestión es un número com-

puesto y se garantiza que esta conclusión es absolutamente correcta. Por otra parte, si todas estas m pruebas fracasan, entonces se concluye que el número es primo. Sin embargo, en ese caso no se tiene certeza. Esta conclusión es correcta con probabilidad $1 - 2^{-m}$. Por lo tanto, si m es suficientemente grande, se puede llegar a esa conclusión con gran confianza.

Algoritmo 11-3 □ Un algoritmo aleatorio para probar si un número es primo

Input: Un número entero N y un parámetro m, $m \geq \lceil \log_2 \varepsilon \rceil$.
Output: Si N es primo o no, con probabilidad $1 - \varepsilon = 1 - 2^{-m}$.
Paso 1. Escoger aleatoriamente m números $b_1, b_2, \ldots, b_m$, $1 < b_1, b_2, \ldots, b_m < N$.
Paso 2. Para toda b_i, probar si $W(b_i)$ se cumple, donde $W(b_i)$ se define como sigue:

1. $b_i^{N-1} \not\equiv 1 \bmod N$.

o bien, 2. $\exists j$ tal que $\dfrac{N-1}{2^j} = k$ es un entero y el máximo común divisor de $b_i^k - 1$ y N es menor que N y mayor que 1.

Si se cumple cualquier $W(b_i)$, entonces regresar N como número compuesto; en caso contrario, regresar N como primo.

A continuación, este algoritmo aleatorio se ilustrará con algunos ejemplos. Considere $N = 12$. Suponga que se escogen 2, 3 y 7. $2^{12-1} = 2048 \not\equiv 1 \bmod 12$. En consecuencia, se concluye que 12 es un número compuesto.

Considere $N = 11$. Suponga que se escogen 2, 5 y 7.

$b_1 = 2$: $2^{11-1} = 1024 \equiv 1 \bmod 11$.

Sea $j = 1$. $(N - 1)/2^j = (11 - 1)/2^1 = 10/2 = 5$.
Esta $j = 1$ es la única j donde $(N - 1)/2^j$ es un entero.
Sin embargo, el máximo común divisor de $2^5 - 1 = 31$ y 11 es 1.

$W(2)$ no se cumple.

$b_2 = 5$: $5^{11-1} = 5^{10} = 9765625 \equiv 1 \bmod 11$.

Como ya se analizó, $j = 1$ es la única j que hace entero a $(N - 1)/2^j = k = 5$.

$b_2^k = 5^5 = 3125.$

El máximo común divisor de $5^5 - 1 = 3124$ y 11 es 11. De nuevo, $W(5)$ no se cumple.

$b_3 = 7$: $7^{11-1} = 282475249 \equiv 1 \bmod 11.$

De nuevo, sea $j = 1$.

$b_3^k = 7^5 = 16807.$

El máximo común divisor de $7^5 - 1 = 16806$ y 11 es 1.

$W(7)$ no se cumple.

Puede concluirse que 11 es primo, con una probabilidad de certidumbre igual a $1 - 2^{-3} = 1 - 1/8 = 7/8$.

Si también se escoge 3, puede demostrarse que también $W(3)$ fracasa. Ahora se tiene $m = 4$ y la probabilidad de certeza se ha incrementado a $1 - 2^{-4} = 15/16$.

Para cualquier N, si m es 10, entonces la probabilidad de que el resultado sea correcto es $1 - 2^{-10}$, que casi es igual a 1.

El hecho de que este algoritmo aleatorio para probar si un número es primo es correcto se basa en el siguiente teorema.

TEOREMA 11-1

1. Si $W(b)$ se cumple para cualquier $1 < b < N$, N es un número compuesto.
2. Si N es compuesto, $\dfrac{N-1}{2} \leq |\{b \mid 1 \leq b < N,\ W(b) \text{ se cumple}\}|$.

Con base en el teorema anterior, se sabe que si N es compuesto, entonces por lo menos la mitad de los b_i ($b_i < N$) tienen la propiedad de que $W(b)$ se cumple. Si se encuentra que sólo existe una b_i tal que $W(b_i)$ se cumple, entonces N debe ser compuesto. Si ninguna de las $W(b_i)$ se cumple, para las mb_i, entonces según el teorema 11-1 la probabilidad de que N sea compuesto es $(\frac{1}{2})m$. En consecuencia, la probabilidad de que N sea primo es mayor que $1 - 2^{-m}$.

11-4 UN ALGORITMO ALEATORIO PARA EL APAREAMIENTO DE PATRONES

En esta sección se presentará un algoritmo aleatorio para el apareamiento de patrones. Este algoritmo puede usarse para resolver el siguiente problema: Dados una cadena

patrón X de longitud n y una cadena de texto Y de longitud m, $m \geq n$, encontrar la primera ocurrencia de X como una subcadena consecutiva de Y. Sin pérdida de generalidad, se supone que X y Y son cadenas binarias.

Por ejemplo, sean $X = 01001$ y $Y = 1010\underline{01001}11$. Entonces es posible ver que X ocurre en Y, como se ha subrayado.

Sean los patrones X y Y, respectivamente

$$X = x_1x_2 \ldots x_n,\ x_i \in \{0, 1\}$$
y $$Y = y_1y_2 \ldots y_m,\ y_i \in \{0, 1\}.$$

Sea $Y(1) = y_1y_2 \ldots y_n$, $Y(2) = y_2y_3 \ldots y_{n+1}$ y así sucesivamente. En general, sea $Y(i) = y_iy_{i+1} \ldots y_{i+n-1}$. Entonces, ocurre un apareamiento si $X = Y(i)$ para alguna i.

Hay otra forma para comprobar X y $Y(i)$. Sea $B(s)$ el valor binario de una cadena s. Luego,

$$B(X) = x_1 2^{n-1} + x_2 2^{n-2} + \cdots + x_n, \text{ y}$$

$B(Y_i) = y_i 2^{n-1} + y_{i+1} 2^{n-2} + \cdots + y_{i+n-1}$, $1 \leq i \leq m - n + 1$. Entonces, simplemente puede comprobarse si $B(X)$ es igual a $B(Y_i)$.

El problema es que cuando n es grande, el cálculo de $B(X)$ y $B(Y_i)$ puede ser difícil. En consecuencia, a continuación se presentará un algoritmo aleatorio para resolver este problema. Debido a que se trata de un algoritmo aleatorio, es posible cometer errores.

Nuestro método consiste en calcular el residuo de $B(X)$ y un número primo P. Sea $(u)_v$ el residuo de dos enteros u y v. Resulta evidente que si $(B(X))_p \neq (B(Y_i))_p$, entonces $X \neq Y_i$, pero no viceversa. Por ejemplo, considere $X = 10110$ y $Y = 10011$.

$$B(X) = 2^4 + 2^2 + 2^1 = 16 + 4 + 2 = 22$$
$$B(Y) = 2^4 + 2^1 + 2^0 = 16 + 2 + 1 = 19.$$

Sea $P = 3$. Entonces $(B(X))_p = (22)_3 = 1$ y $(B(Y))_p = (19)_3 = 1$. Aunque $(B(X))_p = (B(Y))_p$, aún no debe concluirse que $X = Y(i)$.

Nuestro algoritmo aleatorio consta de las siguientes ideas:

1. Seleccionar k números primos $p_1, p_2, \ldots, p_k$.
2. Si para alguna j se tiene $(B(X))p_j \neq (B(Y_i))p_j$, entonces se concluye que $X \neq Y(i)$.

Si $(B(X))p_j = (B(Y))p_j$ para $j = 1, 2, \ldots, k$, entonces se concluye que $X = Y(i)$.

Se demostrará que si la conclusión es $X \neq Y(i)$, entonces se tiene absoluta certeza. Por otra parte, si se concluye que $X = Y(i)$, hay la posibilidad de que se esté cometiendo un error.

La ventaja de este método es que $(B(X))_p$ y $(B(Y_i))_p$ pueden calcularse fácilmente. En realidad no es necesario calcular $B(X)$ y $B(Y_i)$. Recuerde que

$$B(X) = x_1 2^{n-1} + x_2 2^{n-2} + \cdots + x_n.$$

Fácilmente puede verse que

$$(B(X_i))_p = ((((x_1 \cdot 2)_p + x_2)_p \cdot 2\ )_p + x_3)_p \cdot 2 + \cdots$$

De manera semejante,

$$(B(Y(i)))_p = ((((y_i \cdot 2)_p + y_{i+1})_p \ldots\ 2)_p + y_{i+2})_p \cdot 2 + \cdots$$

A través de este tipo de mecanismo, jamás es necesario preocuparse sobre si $B(X)$ o $B(Y(i))$ son números muy grandes. A continuación se presenta un ejemplo para ilustrar esta cuestión.

Sea $X = 10110$

$x_1 = 1$
$x_2 = 0$
$x_3 = 1$
$x_4 = 1$
$x_5 = 0$

Sea p igual a 3

$(x_1 \cdot 2)_p = (1 \cdot 2)_3 = (2)_3 = 2.$
$(2 + x_2)_p = (2 + 0)_3 = (2)_3 = 2.$
$(2 \cdot 2)_p = (4)_3 = 1$
$(1 + x_3)_p = (1 + 1)_3 = (2)_3 = 2.$
$(2 \cdot 2)_p = (4)_3 = 1$
$(1 + x_4)_p = (1 + 1)_3 = (2)_3 = 2.$
$(2 \cdot 2)_p = (4)_3 = 1$
$(1 + x_5)_p = (1 + 0)_3 = (1)_3 = 1.$

Esto es $(B(X))_p = (B(X))_3 = 1$.

A través del cálculo puede verse que todos los números implicados son relativamente pequeños.

A continuación se presenta el algoritmo aleatorio para el apareamiento de patrones.

Algoritmo 11-4 □ Un algoritmo aleatorio para el apareamiento de patrones

Input: Un patrón $X = x_1x_2 \ldots x_n$, un texto $Y = y_1y_2 \ldots y_m$ y un parámetro k.

Output: 1. No, no hay una subcadena consecutiva en Y que corresponda a X.
2. Sí, $Y(i) = y_iy_{i+1} \ldots y_{i+n-1}$ corresponde a X.
Si la respuesta es "No", no hay error.
Si la respuesta es "Sí", hay alguna probabilidad de cometer un error.

Paso 1. Escoger aleatoriamente k números primos $p_1, p_2, \ldots, p_k$ de $\{1, 2, \ldots, nt^2\}$, donde $t = m + n - 1$.

Paso 2. $i = 1$.

Paso 3. $j = 1$.

Paso 4. Si $B(X)p_j \neq (B(Y_i))p_j$, entonces ir al Paso 5.
Si $j = k$, regresar $Y(i)$ como respuesta.
$j = j + 1$.
Ir al Paso 3.

Paso 5. Si $i = t$, regresar "No, no hay una subcadena consecutiva en Y que corresponda a X".
$i = i + 1$.
Ir al Paso 3.

A continuación se lleva a cabo un análisis teórico de este algoritmo aleatorio. Esencialmente, se demostrará que si k es suficientemente grande, entonces la probabilidad de obtener una conclusión errónea es muy pequeña.

Quizás el lector se pregunte, una vez que se obtiene una conclusión, qué está ocurriendo. Cuando se extrae una conclusión falsa, se tienen las condiciones siguientes:

1. $B(X) \neq (B(Y_i)$
2. $(B(X))p_j = (B(Y_i))p_j$ para $j = 1, 2, \ldots, k$

Debido a las condiciones anteriores, puede afirmarse que

$$B(X) = a_j p_j + c_j. \tag{11-1}$$

y $$B(Y_i) = b_j p_j + c_j. \tag{11-2}$$

Entonces $$B(X) - B(Y_i) = (a_j - b_j)p_j. \tag{11-3}$$

La ecuación (11-3) indica que cuando se obtiene una conclusión errónea, entonces $B(X) - B(Y_i) \neq 0$ y p_j divide a $B(X) - B(Y_i)$ para todas las p_j.

Una pregunta crítica que debe plantearse es: ¿Cuántos divisores primos tiene $|B(X) - B(Y(i))|$?

Para contestar esta pregunta, debe recordarse que se tiene un patrón X con n bits y un texto con m bits. Sea Q el producto

$$\prod_{i}^{k=m-n+1} |B(X) - B(Y(i))|$$

donde p_j divide a $|B(X) - B(Y(i))|$.

Observe que p_j también divide a Q.

El valor de Q es menor que $2^{n(m-n+1)}$. Con este valor debe ser posible obtener la probabilidad de que se está cometiendo un error. Se requiere la siguiente afirmación:

Si $u \geq 29$ y $a \geq 2^u$, entonces los diferentes divisores primos de a son menores que $\pi(u)$, donde $\pi(u)$ denota el número de números primos menores que u.

No se abordarán los detalles de cómo se llegó a la afirmación anterior. Para usar esta afirmación, se observa que Q es menor que $2^{n(m-n+1)} - 2^{nt}$. En consecuencia, Q tiene menos que $\pi(nt)$ divisores primos diferentes en el supuesto de que $nt \geq 29$.

En nuestro algoritmo, p_j es un número primo seleccionado de $\{1, 2, \ldots, nt^2\}$. En consecuencia, la probabilidad de que p_j divida a Q es menor que $\pi(nt)/\pi(nt^2)$. Esto significa que la probabilidad de que p_j proporcione una respuesta errónea es menor que $\pi(nt)/\pi(nt^2)$. Debido a que se escogen k números primos, la probabilidad de que se obtenga una conclusión errónea con base en estos números es menor que $(\pi(nt)/\pi(nt^2))^k$.

Nuestro análisis se resume como sigue:

Si en el algoritmo para el apareamiento de patrones se escogen k números primos distintos, entonces la probabilidad de cometer un error es menor que en el supuesto de que $\left(\frac{\pi(nt)}{\pi(nt^2)}\right)^k$ en el supuesto de que $nt \geq 29$.

Nuestra siguiente pregunta es: ¿Cómo calcular $\pi(x)$? Se cuenta con la siguiente fórmula de estimación:

Para toda $u \geq 17$,

$$\frac{u}{\ln u} \leq \pi(u) \leq \pi(u) \leq 1.25506 \frac{u}{\ln u}.$$

Ahora puede verse que en general la probabilidad de extraer una conclusión errónea es bastante pequeña aun si k no es demasiado grande.

Se supondrá que $nt \geq 29$. Así,

$$\begin{aligned}
\frac{\pi(nt)}{\pi(nt^2)} &\leq 1.25506 \frac{nt}{\ln nt} \cdot \frac{\ln (nt^2)}{nt^2} \\
&= \frac{1.25506}{t} \left(\frac{\ln (nt^2)}{\ln (nt)} \right) \\
&= \frac{1.25506}{t} \left(\frac{\ln (nt) + \ln (t)}{\ln (nt)} \right) \\
&= \frac{1.25506}{t} \left(1 + \frac{\ln (t)}{\ln (nt)} \right).
\end{aligned}$$

Por ejemplo, sean $n = 10$ y $m = 100$. Luego,

$t = 91$.

$$\begin{aligned}
\frac{\pi(nt)}{\pi(nt^2)} &\leq \frac{1.25506}{t} \left(1 + \frac{\mathrm{n}\, t}{\ln (nt)} \right) \\
&= \frac{1.25506}{91} \left(1 + \frac{\ln(91)}{\ln (910)} \right) \\
&= 0.013792 \cdot \left(1 + \frac{4.5109}{6.8134} \right) \\
&= 0.013792 \cdot (1 + 0.6621) \\
&= 0.013792 \cdot 1.6621 \\
&= 0.0229.
\end{aligned}$$

Suponga que $k = 4$. Entonces la probabilidad de extraer una conclusión errónea es

$(0.0229)^4 \approx 2.75 * 10^{-7}$

que es muy pequeña.

11-5 UN ALGORITMO ALEATORIO PARA PRUEBAS INTERACTIVAS

Considere el problema siguiente. Ocurrió durante la época de la Guerra Fría; un agente británico MI5 (Military Intelligence Unit 5; consulte *Spy Catcher*, de P. Wright) quería hablar con un agente a quien el gobierno británico había infiltrado en la KGB durante años.

Sean *B* y *A* el agente MI5 y el espía infiltrado en la KGB, respectivamente. Ahora el problema es: ¿Puede *B* saber que *A* es el *A* verdadero y no un oficial de la KGB que asume el papel de *A*? Bien. Un método trivial para contestar lo anterior consiste en preguntar, por ejemplo, el apellido de soltera de la madre de *A*. El problema es que si *A* contesta correctamente, algún agente de la KGB puede suplantar fácilmente a *A* la próxima vez. En consecuencia, *B* puede pedir a *A* que haga algo más difícil, tan difícil que la gente común no pueda llevar a cabo. Por ejemplo, pedir a *A* que determine la satisfactibilidad de una fórmula booleana. Se supone que *A* es una persona inteligente y que sabe muy bien cómo resolver este problema NP-completo. Así, cada vez que recibe una fórmula booleana de *B*, resuelve el problema. Si la fórmula es satisfactible, envía una asignación a *B* y simplemente "NO" en caso de no ser satisfactible. *B* no es tan brillante. Sin embargo, conoce la definición de satisfactibilidad. Por lo menos puede comprobar si la asignación satisface la fórmula o no. Si *A* resuelve correctamente el problema de satisfactibilidad en cada ocasión, entonces *B* estará feliz y convencido de que *A* es el verdadero *A*.

Sin embargo, un interceptor puede imaginar poco a poco que *B* siempre envía una fórmula booleana a *A* y que si esta fórmula es satisfactible, *A* regresa una asignación satisfactible a *B*. Este interceptor empieza a estudiar métodos mecánicos de demostración de teoremas, de modo que tarde o temprano logrará suplantar a *A*.

Puede decirse que tanto *A* como *B* han revelado demasiado. Sería agradable que *B* envíe algunos datos a *A* y que después de que *A* realice algunos cálculos en este conjunto de datos, regrese algunos datos resultantes a *B*. Luego, *B* comprueba los resultados de *A* haciendo algunos cálculos con ellos. Si queda satisfecho, puede estar seguro de que *A* es la persona correcta. Aun así, un interceptor puede interceptar algunos datos. Aunque será muy difícil para él saber qué ocurre.

En esta sección de mostrará que *B* puede pedir a *A* que resuelva un problema de residuo no cuadrático y se verá, con gran interés, que los datos pueden enviarse de ida y vuelta sin revelar mucha información. Por supuesto, se trata de un algoritmo aleatorio, de modo que existe cierto grado de error.

Sean x y y dos enteros positivos, $0 < y < x$, *tales que* $gcd(x, y) = 1$. *Se define* $Z_x = \{z \mid 0 < z < x, gcd(z, x) = 1\}$. *Se dice que y es un residuo cuadrático mod x si* $y = z^2$ *mod x para alguna* $z \in Z_x$ *y que y es un residuo no cuadrático mod x en caso contrario. Además, se definen dos conjuntos:*

$QR = \{(x, y) \mid y \text{ es un residuo cuadrático mod } x\}$
$QNR = \{(x, y) \mid y \text{ es un residuo no cuadrático mod } x\}$

Por ejemplo, sean $y = 4$ y $x = 13$. Puede verse que existe una z; a saber 2, tal que $z \in Z_x$ y $y = z^2 \bmod x$. (Esto puede comprobarse fácilmente: $4 = 2^2 \bmod 13$.) Así, 4 es un residuo cuadrático mod 13. También puede demostrarse que 8 es un residuo no cuadrático mod x.

Entonces, ¿cómo pueden comunicarse A y B de modo que A resuelva el problema del residuo no cuadrático para B sin revelar demasiada información?

Pueden proceder como sigue:

1. Tanto A como B conocen el valor de x, mismo que mantienen en secreto. B conoce el valor de y.
2. Acción de *B:*
   a) Lanzar monedas para obtener m bits: $b_1, b_2, \ldots, b_m$, donde m es la longitud de la representación binaria de x.
   b) Encontrar $z_1, z_2, \ldots, z_m$, $0 < z_i < x$, de modo que $gcd(z_i, x) = 1$ para toda i.
   c) Calcular $w_1, w_2, \ldots, w_m$ a partir de $b_1, b_2, \ldots, b_m, z_1, z_2, \ldots, z_m$, como sigue:

$$w_i = z_i^2 \bmod x \text{ si } b_i = 0.$$
$$w_i = (z_i^2 \cdot y) \bmod x \text{ si } b_i = 1.$$

   d) Enviar $w_1, w_2, \ldots, w_m$ a A.
3. Acción de A:
   a) Recibir $w_1, w_2, \ldots, w_m$ de B.
   b) Establecer $c_1, c_2, \ldots, c_m$ como sigue:

$$c_i = 0 \text{ si } (x, w_i) \in QR.$$
$$c_i = 1 \text{ si } (x, w_i) \in QNR.$$

   c) Enviar $c_1, c_2, \ldots, c_m$ a B.
4. Acción de B:
   a) Recibir $c_1, c_2, \ldots, c_m$ de A.
   b) Devolver "YES, y es un residuo no cuadrático mod x" si $b_i = c_i$ para toda i. Y devolver "NO, y es un residuo cuadrático mod x" en caso contrario.

Puede demostrarse que si y es un residuo no cuadrático mod x, entonces B acepta este hecho con probabilidad $1 - 2^{-|x|}$. *Si y es un residuo cuadrático mod x, entonces B lo acepta como un residuo no cuadrático con probabilidad* $2^{-|x|}$.

Ciertamente, B puede reducir el error repitiendo el proceso. Observe que A debe ser una persona inteligente en el sentido de que es capaz de resolver el problema del residuo no cuadrático.

Así, A es un "comprobador". B es sólo un "verificador".

A continuación se proporcionan algunos ejemplos.

Suponga que $(x, y) = (13, 8)$, $|x| = 4$.

Acción de *B*:

a) Suponga que $b_1, b_2, b_3, b_4 = 1, 0, 1, 0$.

b) Suponga que $z_1, z_2, z_3, z_4 = 9, 4, 7, 10$, que son todos "primos a pares" con respecto a $x = 13$.

c) w_1, w_2, w_3, w_4 se calcula como sigue:

$b_1 = 1$, $w_1 = (z_1^2 \cdot y) \bmod x = (9^2 \cdot 8) \bmod 13 = 648 \bmod 13 = 11$

$b_2 = 0$, $w_2 = (z_2^2) \bmod x = 4^2 \bmod 13 = 16 \bmod 13 = 3$

$b_3 = 1$, $w_3 = (z_3^2 \cdot y) \bmod x = (7^2 \cdot 8) \bmod 13 = 392 \bmod 13 = 2$

$b_4 = 0$, $w_4 = (z_4^2) \bmod x = 10^2 \bmod 13 = 100 \bmod 13 = 9$

Así, $(w_1, w_2, w_3, w_4) = (11, 3, 2, 9)$.

d) Mandar (11, 3, 2, 9) a A.

Acción de *A*:

a) Recibir (11, 3, 2, 9) de B.

b) $(13, 11) \in QNR$, $c_1 = 1$.

$(13, 3) \in QR$, $c_2 = 0$.

$(13, 2) \in QNR$, $c_3 = 1$.

$(13, 9) \in QR$, $c_4 = 0$.

c) Enviar $(c_1, c_2, c_3, c_4) = (1, 0, 1, 0)$ a B.

Acción de *B*:

Debido a que $b_i = c_i$ para toda i, B acepta que 8 es un residuo no cuadrático de 13, lo que es verdadero.

Se considerará otro ejemplo.

$(x, y) = (13, 4)$, $|x| = 4$.

Acción de *B*:

a) Suponga que nuevamente $b_1, b_2, b_3, b_4 = 1, 0, 1, 0$.
b) Suponga que $z_1, z_2, z_3, z_4 = 9, 4, 7, 10$.
c) Es fácil demostrar que $(w_1, w_2, w_3, w_4) = (12, 3, 1, 9)$.
d) Mandar (12, 3, 1, 9) a *A*.

Acción de *A*:

a) Recibir (12, 3, 1, 9) de *B*.
b) $(13, 12) \in QR$, $c_1 = 0$.
$(13, 3) \in QR$, $c_2 = 0$.
$(13, 1) \in QR$, $c_3 = 0$.
$(13, 9) \in QR$, $c_4 = 0$.
c) Mandar (0, 0, 0, 0) a *B*.

Acción de *B*:

Debido a que no es cierto que $b_i = c_i$ para toda i, *B* acepta el hecho de que 4 es un residuo cuadrático mod 13.

La base teórica que sustenta este algoritmo aleatorio es la teoría de números, cuyo alcance rebasa los objetivos de este libro.

11-6 UN ALGORITMO ALEATORIO DE TIEMPO LINEAL PARA ÁRBOLES DE EXPANSIÓN MÍNIMA

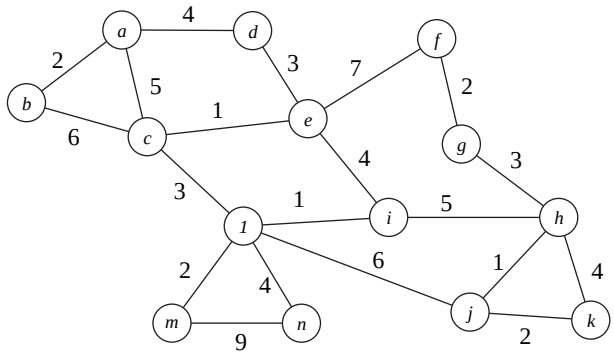
En el capítulo 3 se presentaron dos algoritmos del árbol de expansión mínimo basados en el método codicioso. Uno de estos algoritmos de árbol es el algoritmo de Kruskal, cuya complejidad temporal es $O(n^2 \log n)$, y el otro algoritmo es el algoritmo de Prim. La complejidad temporal de una versión sofisticada de este algoritmo es $O(n + m\alpha(m, n))$, donde $n(m)$ es el número de nodos (aristas) de una gráfica y $\alpha(m, n)$ es la función inversa de Ackermann. En esta sección se presentará un algoritmo aleatorio del árbol de expansión mínimo cuya complejidad temporal esperada es $O(n + m)$.

Este algoritmo se basa en el denominado "paso Boruvka", propuesto por Boruvka en 1926. El siguiente lema ilustra esta idea que subyace en el paso Boruvka:

LEMA 1: Sean V_1 y V_2 conjuntos de vértices no vacíos donde $V_1 \cup V_2 = V$ y $V_1 \cap V_2 = \phi$, y sea (v, u) la arista de peso mínimo con un punto extremo en V_1 y el otro punto extremo en V_2. Así, (v, u) debe estar contenido en el árbol de expansión mínimo en G.

El lema 1 puede representarse de otra forma como sigue: en una gráfica G, para todo nodo u, de todos los nodos incidentes en u, si la arista (u, v) es el de menor peso, entonces (u, v) debe ser una arista en el árbol de expansión mínimo de G. Es fácil demostrar el lema 1. Considere la figura 11-7. Para el nodo c, de todos los nodos incidentes en c, la arista (c, e) es el de menor peso. En consecuencia, la arista (c, e) debe incluirse en el árbol de expansión mínimo de G. De manera semejante, es posible demostrar fácilmente que también debe incluirse la arista (f, g).

FIGURA 11-7 Una gráfica.



Luego, con fundamento en el lema 1 se seleccionan todas las aristas de la gráfica en la figura 11-7 que deben incluirse en el árbol de expansión mínimo de G. Las componentes resultantes unidas se muestran en la figura 11-8. En la gráfica, en la figura 11-8, todas las líneas punteadas son aristas de componentes conectados.

Luego, todos los nodos en cada componente conectada se contraen en un nodo. Así, se tienen cinco nodos, que se muestran en la figura 11-9. Después de eliminar aristas múltiples y ciclos, el resultado se muestra en la figura 11-10.

Debido a que la gráfica resultante consta de más de un nodo, es posible aplicar nuevamente el lema 1. El resultado se muestra en la figura 11-11 y las aristas seleccionadas son (a, d), (c, l) y (g, h).

Después de contraer los nodos en cada una de las componentes conectadas, ahora se tienen dos nodos que se muestran en la figura 11-12.

FIGURA 11-8 Selección de las aristas en el paso Boruvka.

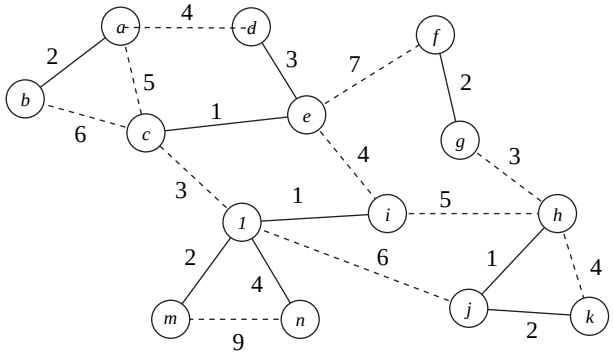


FIGURA 11-9 Construcción de los nodos.

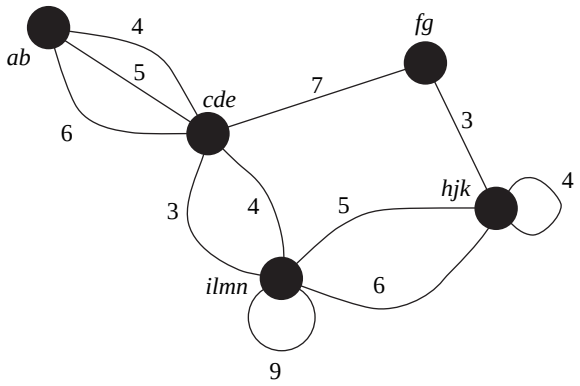


FIGURA 11-10 Resultado de la aplicación del primer paso Boruvka.

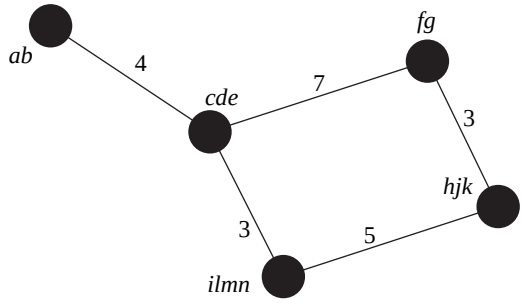


FIGURA 11-11 La segunda selección de aristas.

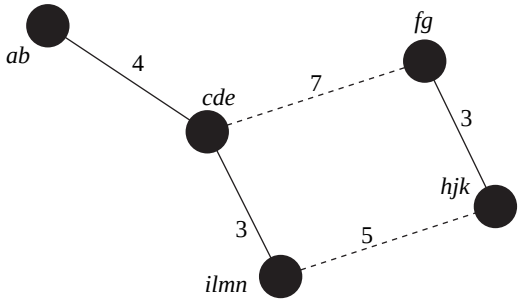
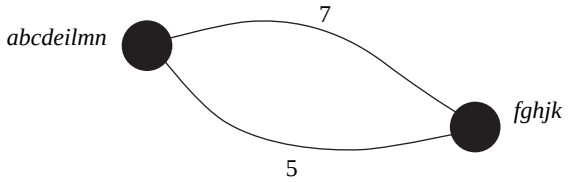
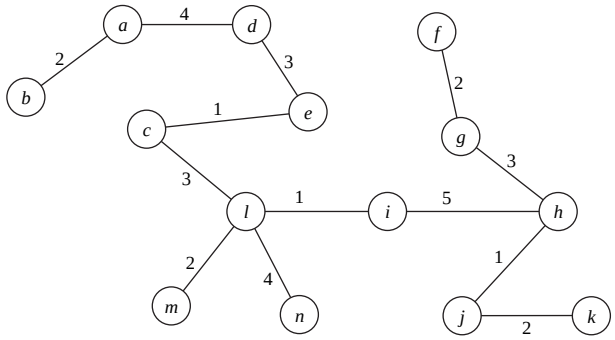


FIGURA 11-12 La segunda construcción de aristas.



De nuevo, después de eliminar las aristas múltiples y seleccionar la arista (*i*, *h*), ahora es posible contraer todas las aristas en un nodo. Se ha completado el proceso. Todas las aristas seleccionadas constituyen el árbol de expansión mínimo que se muestra en la figura 11-13.

FIGURA 11-13 El árbol de expansión mínimo obtenido en el paso Boruvka.



El algoritmo de Boruvka para encontrar el árbol de expansión mínimo aplica de manera recurrente el paso Boruvka hasta que la gráfica resultante se reduce a un solo vértice. Sean una gráfica $G(V, E)$ la entrada al paso Boruvka y la salida, una gráfica $G'(V', E')$. A continuación se describe el paso Boruvka.

El paso Boruvka

1. Para todo nodo u, encontrar el arista (u, v) de menor peso unido a u. Encontrar todas las componentes unidas determinadas por las aristas marcadas.
2. Contraer en un solo vértice toda componente unida determinada por las aristas marcadas. Sea $G'(V', E')$. Eliminar las aristas múltiples y los ciclos.

La complejidad temporal para un paso Boruvka es $O(n + m)$, donde $|V| = n$ y $|E| = m$. Debido a que G es conexa, $m > n$. Por lo tanto, $O(n + m) = O(m)$. Debido a que toda componente unida determinada por las aristas marcadas contiene por lo menos dos vértices, después de la ejecución de cada paso Boruvka el número de aristas restantes es menor que la mitad del número original. Por lo tanto, el número total de pasos Boruvka ejecutados es $O(\log n)$. La complejidad temporal del algoritmo de Boruvka es $O(m \log n)$.

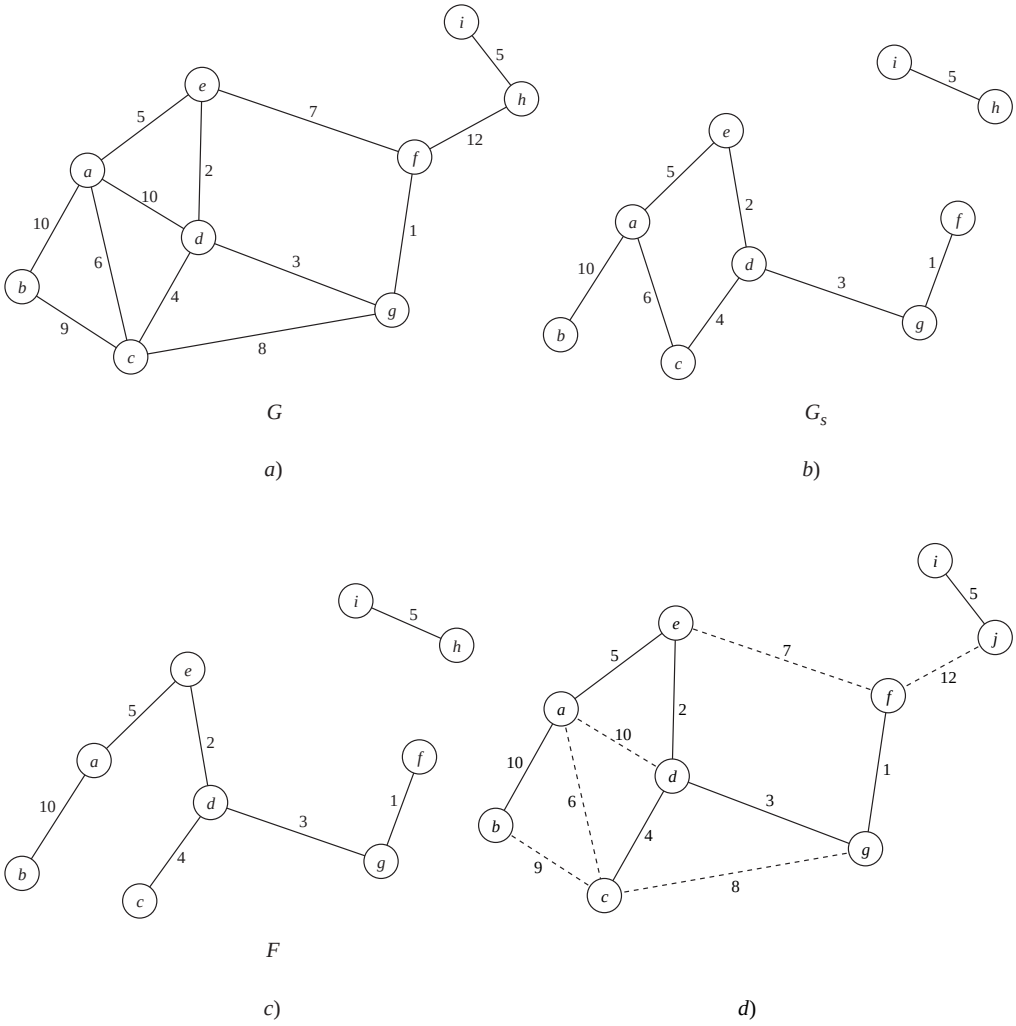
Para usar de manera eficiente el paso Boruvka, es necesario aplicar un nuevo concepto. Considere la figura 11-14. En la figura 11-14*b*), la gráfica G_s es una subgráfica de la gráfica G en la figura 11-14*a*). En la figura 11-14*c*) se muestra un bosque de expansión mínimo F de G_s. En la figura 11-14*d*), el árbol de expansión mínimo F está encajado en la gráfica original G. Todas las aristas que no son aristas de F ahora son aristas punteadas. Se considerará la arista (e, f). El peso de (e, f) es 7. Sin embargo, en el bosque F hay una ruta entre e y f. A saber, $(e, d) \rightarrow (d, g) \rightarrow (g, f)$. El peso de (e, f) es mayor que el peso máximo de las aristas en esta ruta. Según un lema que se presentará a continuación, la arista (e, f) no puede ser una arista de un árbol de expansión mínimo de G. Antes de presentar el lema, se definirá el término F-pesado.

Sean $w(x, y)$ el peso de la arista (x, y) en G, G_s una subgráfica de G y F un bosque de expansión mínimo de G_s. Sea $w_F(x, y)$ el peso mínimo de una arista en la ruta que une a x y y en F. Si x y y no están unidos en F, sea $w_F(x, y) = \infty$. Se dice que la arista (x, y) es F-pesada (F-ligera) con respecto a F si $w(x, y) > w_F(x, y)(w(x, y) \leq w_F(x, y))$.

Considere la figura 11-14*d*). Puede verse que las aristas (e, f), (a, d) y (c, g) son todas F-pesadas con respecto a F. Una vez que se ha definido este nuevo concepto, puede presentarse el lema siguiente que es muy importante para utilizar el paso Boruvka.

LEMA 2: Sea G_s una subgráfica de una gráfica $G(V, E)$. Sea F un bosque de expansión mínimo de G_s. Las aristas F-pesadas en G con respecto a F no pueden ser aristas del árbol de expansión mínimo de G.

FIGURA 11-14 Aristas *F*-pesadas.



No se demostrará el lema anterior. Sin embargo, con este lema se sabe que las aristas (*e*, *f*), (*a*, *d*) y (*c*, *g*) no pueden ser aristas del árbol de expansión mínimo.

Para utilizar completamente todo el mecanismo del paso Boruvka se requiere otro lema. Se trata del lema 3.

LEMA 3: Sea *H* una subgráfica obtenida a partir de *G* al incluir cada arista de manera independiente con probabilidad *p* y sea *F* el bosque de expansión mínimo de *H*. El número esperado de aristas *F*-ligeras en *G* es cuando mucho n/p, donde *n* es el número de vértices de *G*.

A continuación se describe el algoritmo del árbol de expansión mínima.

Algoritmo 11-5 □ Un algoritmo aleatorio para el árbol de expansión mínima

Input: Una gráfica conexa G.
Output: Un árbol de expansión mínima de G.
Paso 1. Aplicar tres veces el paso Boruvka. Sea $G_1(V_1, E_1)$ la gráfica resultante. Si G_1 contiene un nodo, regresar el conjunto de aristas marcadas en el Paso 1 y salir.
Paso 2. Obtener una subgráfica H de G_1 seleccionando cada arista de manera independiente con probabilidad 1/2. Aplicar de manera recurrente el algoritmo a H para obtener un bosque de expansión mínimo F de H. Obtener una gráfica $G_2(V_2, E_2)$ eliminando todas las aristas F-pesadas en G_1 con respecto a F.
Paso 3. Aplicar de manera recursiva el algoritmo a G_2.

A continuación se analizará la complejidad temporal del algoritmo 11-5.

Sea $T(|V|, |E|)$ el tiempo de ejecución esperado del algoritmo para la gráfica $G(V, E)$.

Cada ejecución del Paso 1 requiere tiempo $O(|V| + |E|)$. Una vez que se ha ejecutado el Paso 1, se tienen $|V_1| \leq |V|/8$ y $|E_1| < |E|$. En cuanto al Paso 2, el tiempo necesario para calcular H es $O(|V_1| + |E_1|) = O(|V| + |E|)$. El tiempo necesario para calcular F es $T(|V_1|, |E_1|/2) = T(|V|/8, |E|/2)$. El tiempo necesario para eliminar todas las aristas F-pesadas es $O(|V_1| + |E_1|) = O(|V| + |E|)$. Al aplicar el lema 3, se tiene que el valor esperado para $|E_2|$ es a lo sumo $2|V_2| \leq |V|/4$. Entonces, el tiempo esperado necesario para ejecutar el Paso 3 es $T(|V_2|, |E_2|) = T(|V|/8, |V|/4)$. Sean $|V| = n$ y $|E| = m$. Entonces se tiene la siguiente fórmula de recurrencia:

$$T(n, m) \leq T(n/8, m/2) + T(n/8, n/4) + c(n + m),$$

para alguna constante c. Puede demostrarse que

$$T(n, m) \leq 2c \cdot (n + m).$$

Alentamos al lector para que compruebe esta solución al sustituir (2) en (1). Por lo tanto, el tiempo de ejecución esperado del algoritmo es $O(n + m)$.

11-7 NOTAS Y REFERENCIAS

En Maffioli (1986) puede consultarse una investigación de algoritmos aleatorios. Gill (1987) y Kurtz (1987) también revisaron los algoritmos aleatorios. Es conveniente indicar que la idea de algoritmo aleatorio tiene varios significados para personas distintas. Algunas veces denota un algoritmo que es aceptable para realizar análisis de casos promedio. Es decir, este algoritmo se comporta en forma diferente para conjuntos de datos distintos. En este libro, insistimos en que un algoritmo aleatorio es un algoritmo que usa el lanzamiento de monedas en el proceso. En otras palabras, para los mismos datos de entrada, debido al proceso aleatorio, el programa puede comportarse de manera bastante diferente.

El hecho de que el problema del par más cercano pueda resolverse mediante un algoritmo aleatorio fue propuesto por Rabin (1976). Para resultados recientes, consulte a Clarkson (1988). Para la prueba de primos usando algoritmos aleatorios, consulte a Solovay y Strassen (1977) y a Rabin (1980). El hecho de que el problema del número primo es un problema polinomial fue demostrado posteriormente por Agrawal, Kayal y Saxena (2004). El algoritmo aleatorio para el apareamiento de patrones apareció en Karp y Rabin (1987). El algoritmo aleatorio para demostraciones interactivas puede consultarse en Goldwasser, Micali y Rackoff (1988). Galil, Haber y Yung (1989) sugirieron una mejora adicional de su método. El algoritmo aleatorio del árbol de expansión mínimo puede consultarse en Karger, Klein y Tarjan (1995). El paso Boruvka puede encontrarse en Boruvka (1926) y el método para eliminar aristas F-pesadas puede encontrarse en Dixon, Rauch y Tarjan (1992).

Los algoritmos aleatorios también se analizan con todo detalle en Brassard y Bratley (1988).

11-8 BIBLIOGRAFÍA ADICIONAL

Los algoritmos aleatorios pueden clasificarse en dos tipos: secuenciales y paralelos. Aunque este libro de texto está restringido a algoritmos secuenciales, se recomiendan algunos algoritmos aleatorios paralelos.

Para algoritmos aleatorios secuenciales se recomiendan las siguientes obras: Agarwal y Sharir (1996); Anderson y Woll (1997); Chazelle, Edelsbrunner, Guibas, Sharir y Snoeyink (1993); Cheriyan y Harerup (1995); Clarkson (1987); Clarkson (1988); d'Amore y Liberatore (1994); Dyer y Frieze (1989); Goldwasser y Micali (1984); Kannan, Mount y Tayur (1995); Karger y Stein (1996); Karp (1986); Karp, Motwani y Raghavan (1988); Matousek (1991); Matousek (1995); Megiddo y Zemel (1986); Mulmuley, Vazirani y Vazirani (1987); Raghavan y Thompson (1987); Teia (1993); Ting y Yao (1994); Wenger (1997); Wu y Tang (1992); Yao (1991), y Zemel (1987).

Para algoritmos aleatorios paralelos, recomendamos la lectura de Alon, Babai e Itai (1986); Anderson (1987); Luby (1986), y Spirakis (1988).

Una gran cantidad de artículos de reciente publicación incluyen los de Aiello, Rajagopalan y Venkatesan (1998); Albers (2002); Alberts y Henzinger (1995); Arora y Brinkman (2002); Bartal y Grove (2000); Chen y Hwang (2003); Deng y Mahajan (1991); Epstein, Noga, Seiden, Sgall y Woeginger (1999); Froda (2000); Har-Peled (2000); Kleffe y Borodovsky (1992); Klein y Subramanian (1997); Leonardi, Spaccamela, Presciutti y Ros (2001); Meacham (1981), y Sgall (1996).

Ejercicios

11.1 Escriba un programa que implemente el algoritmo aleatorio para resolver el problema del par más cercano. Pruebe sus algoritmos.

11.2 Use el algoritmo aleatorio para probar si un número es primo a fin de determinar si los siguientes números son primos o no:

13, 15, 17.

11.3 Use el algoritmo aleatorio para probar el apareamiento de patrones a las dos cadenas siguientes:

$X = 0101$

$Y = 0010111$

11.4 Use el algoritmo que se presentó en la sección 11-5 para determinar si 5 es un residuo cuadrático de 13 o no. Presente un ejemplo en el que se obtenga una conclusión errónea.

11.5 Lea las secciones 8-5 y 8-6 de la obra de Brassard y Bratley 1988.

capítulo

12

Algoritmos en línea

En todos los capítulos previos, los algoritmos fueron diseñados en el supuesto de que se cuenta con todos los datos antes de ejecutar el algoritmo. Es decir, todos los problemas se resuelven juntos gracias a la información completa de los datos. Sin embargo, en la realidad, no siempre sucede así. Considere el problema de la programación del disco. Las peticiones de los servidores de discos son absolutamente desconocidas para el algoritmo. Llegan de una en una. El problema de paginación que ocurre en el diseño de sistemas operativos también es un problema en línea. Simplemente es imposible conocer las páginas a las que se accederá antes de ejecutar los programas. Cuando los datos llegan en línea, el algoritmo debe intervenir y ocuparse de cada dato recién llegado. Como no se tiene toda la información, en ese momento la intervención puede parecer correcta, aunque después quizá no lo sea. Por consiguiente, todos los algoritmos en línea son algoritmos de aproximación en el sentido de que nunca pueden garantizar la obtención de soluciones óptimas. Se considerará el problema en línea del árbol de expansión mínima. En este caso, primeramente debe eliminarse la palabra "mínimo", ya que no lo será. Por ello, se denominará problema en línea del árbol de expansión "pequeña". Un algoritmo en línea para manipular la situación puede funcionar como sigue: cada vez que llega un dato, unirlo con su vecino más cercano. Suponga que se tienen seis puntos como se muestra en la figura 12-1. Los datos llegan en el orden que se especifica. Así, un algoritmo en línea produciría un árbol de expansión como se muestra en la figura 12-2. Resulta evidente que este árbol no es óptimo. Un árbol de expansión óptimo, construido con conocimiento total de los datos, se muestra en la figura 12-3.

Debido a que un algoritmo en línea debe ser un algoritmo de aproximación, es natural que su desempeño se mida al comparar su resultado con el que se obtiene al ejecutar un algoritmo en línea óptimo. Sea $C_{onl}(C_{off})$ que denota el costo de ejecución de un algoritmo en línea (fuera de línea óptimo) sobre el mismo conjunto de datos. Si $C_{onl} \leq c \cdot C_{off} + b$ donde b es una constante, entonces se dice que la tasa de rendimiento de este algoritmo en línea es c y que el algoritmo es c-competitivo. Si no es posible que c sea más pequeño, se dice que el algoritmo es óptimo. Los algoritmos en

FIGURA 12-1 Conjunto de datos para ilustrar un algoritmo en línea del árbol de expansión.

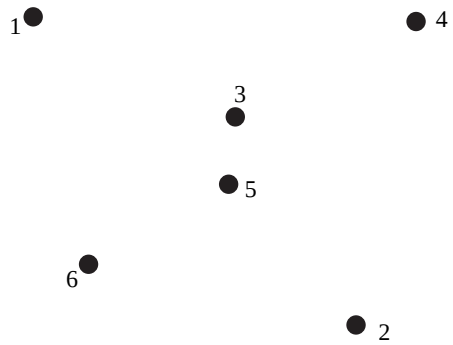


FIGURA 12-2 Árbol de expansión producido por un algoritmo en línea del árbol de expansión.

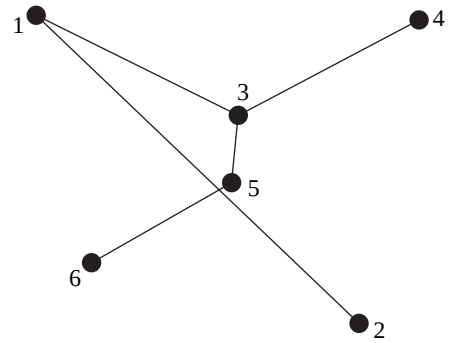
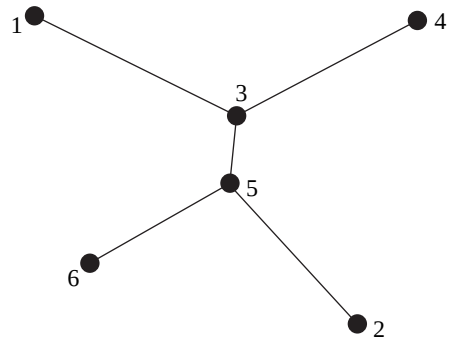


FIGURA 12-3 Árbol de expansión mínima para los datos que se muestran en la figura 12-1.



línea de ninguna manera son fáciles de diseñar, ya que el diseño debe ir acompañado de un análisis. En este capítulo se presentan varios algoritmos en línea, así como sus análisis.

12-1 EL PROBLEMA EUCLIDIANO EN LÍNEA DEL ÁRBOL DE EXPANSIÓN MÍNIMA RESUELTO CON EL MÉTODO CODICIOSO

En el problema euclidiano del árbol de expansión mínima se tiene un conjunto de puntos en un plano y la tarea consiste en construir un árbol de expansión mínima a partir de estos puntos. Para la versión en línea de este problema, los puntos se revelan uno por uno y siempre que se revela un punto es necesario emprender alguna acción para unir este punto con el árbol que ya se ha construido. Además, la acción es irreversible. Resulta evidente que los árboles construidos mediante un algoritmo en línea deben ser aproximados. El algoritmo codicioso para resolver este árbol de expansión euclidiano puede describirse como sigue: Sean $v_1, v_2, \ldots, v_{k-1}$ que se revelan y T el árbol de expansión que está construyéndose. La arista más corta entre v_k y $v_1, v_2, \ldots, v_{k-1}$ se suma a T. Se considera el conjunto de puntos que se muestra en la figura 12-4. El árbol de expansión construido con el método codicioso se muestra en la figura 12-5.

FIGURA 12-4 Un conjunto de cinco puntos.

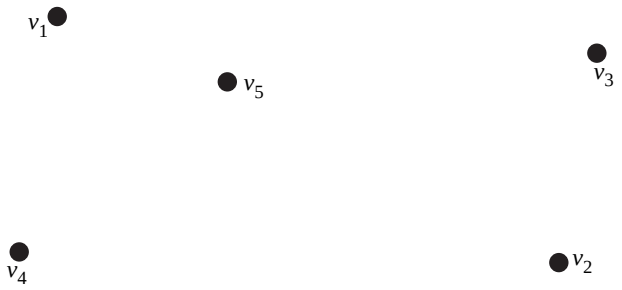
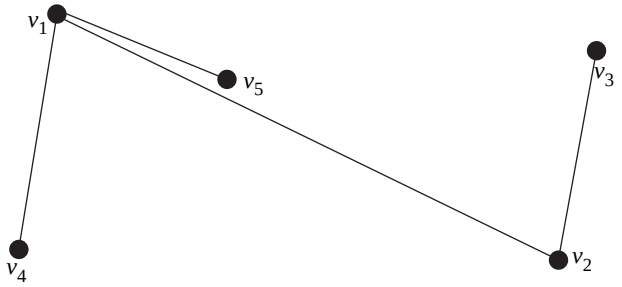


FIGURA 12-5 Árbol construido por el método codicioso.



A continuación se analizará el desempeño de este algoritmo en línea con base en el método codicioso. Sea S un conjunto de n puntos. Un árbol mínimo de Steiner en S es un árbol que conecta los puntos n en S con una longitud mínima. Sea l la longitud de un árbol mínimo de Steiner construido sobre S. Sea T_{onl} el árbol de expansión construido con el algoritmo en línea. Se demostrará que este algoritmo es $O(\log n)$ competitivo.

Esencialmente, se demostrará que en T_{onl}, la k-ésima arista más larga tiene una longitud cuando mucho igual a $2l/k$, $1 \leq k \leq n - 1$. Otra forma de plantear lo anterior es que sobre T_{onl}, hay cuando mucho $(k - 1)$ aristas cuyas longitudes son mayores que $2l/k$. Dada una secuencia de vértices que producen T_{onl}, sea S_k el conjunto de puntos cuyas sumas al árbol T_{onl} provocan que T_{onl} tenga aristas con longitudes mayores que $2l/k$. Ahora, la afirmación original se convierte en la siguiente: la cardinalidad de S_k es menor que k. Para demostrarlo, observe que por la definición del algoritmo codicioso en línea, la distancia entre cada par de puntos en S_k debe ser mayor que $2l/k$. Así, la longitud de una ruta del problema óptimo del agente viajero sobre S_k debe ser mayor que

$$|S_k| \ \frac{2l}{k}.$$

Debido a que la longitud de la ruta de un problema óptimo del agente viajero sobre un conjunto de puntos es por lo menos dos veces la longitud de un árbol mínimo de Steiner del mismo conjunto de puntos, la longitud de un árbol mínimo de Steiner sobre S_k es mayor que

$$|S_k| \ \frac{l}{k}.$$

Debido a que la longitud de un árbol mínimo de Steiner sobre S_k es menor que la longitud sobre S, se tiene que

$$|S_k| \ \frac{l}{k} < l.$$

o bien, de manera equivalente,

$$|S_k| < k.$$

Lo anterior significa que la cardinalidad de S_k es menor que k. Es decir, se ha demostrado que la afirmación original: *La longitud de la k-ésima arista más larga de T_{onl} es cuando mucho igual a 2l/k*. Así, la longitud total de T_{onl} es cuando mucho igual a

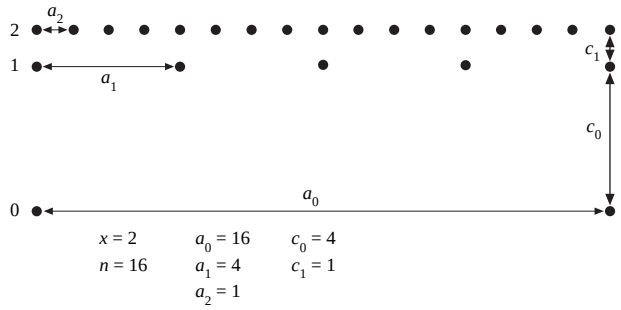
$$\sum_{k=1}^{n-1} \frac{2l}{k} = 2l \sum_{k=1}^{n-1} \frac{1}{k} = l \times (\log n).$$

Esto indica que nuestro algoritmo codicioso en línea del árbol de expansión es $O(\log n)$ competitivo. Este algoritmo, ¿es óptimo? Ciertamente no lo es, porque puede demostrarse que una cota inferior de la tasa de competitividad es $\Omega(\log n/\log\log n)$. Esto se analiza a continuación. Se encontrará una entrada σ para el problema en línea del árbol de expansión y se demostrará que no existe ningún algoritmo en línea A tal que la tasa entre la longitud del árbol construido por A con σ y la longitud del árbol de expansión mínima con σ es $\Omega(\log n/\log\log n)$. A continuación se describirá la entrada σ. Todos los puntos de la entrada σ están en una retícula. Suponga que x es un entero y que $x \geq 2$. Sea $x^{2x} = n$. Entonces $x \geq 1/2(\log n/\log\log n)$ y $n \geq 16$. La entrada consta de $x + 1$ capas de puntos, donde cada capa contiene un conjunto de puntos equidistantes sobre una recta horizontal de longitud $n = x^{2x}$. Las coordenadas de los puntos en la capa i, $0 \leq i \leq x$, son (ja_i, b_i) donde $a_i = x^{2x-2i}$, $b_i = 0$ para $i = 0$, $b_i = \sum_{k=1}^{i} a_k$ para $i \neq 0$, y para $0 \leq j \leq n/a_i$. Así, $a_0 = x^{2x} (= n)$ y $a_x = 1$. La distancia vertical entre la capa i y la capa $i + 1$ es $c_i = b_{i+1} - b_i = a_{i+1}$ para toda i. El número de puntos en la capa i es $\dfrac{n}{a_i} + 1$. El número total de puntos de entrada es

$$\begin{aligned}
\sum_{i=0}^{x}\left(\frac{n}{a_i}+1\right) &= \sum_{i=0}^{x}\left(\frac{x^{2x}}{x^{2x-2i}}+1\right)\\
&= \sum_{i=0}^{x}(x^{2i}+1)\\
&= \frac{x^{2x+2}-1}{x^2-1}+x+1\\
&= \frac{nx^2-1}{x^2-1}+x+1\\
&= n+\frac{n-1}{x^2-1}+x+1\\
&= O(n).
\end{aligned}$$

Una entrada ejemplo σ para $x = 2$ y $n = 16$ se muestra en la figura 12-6.

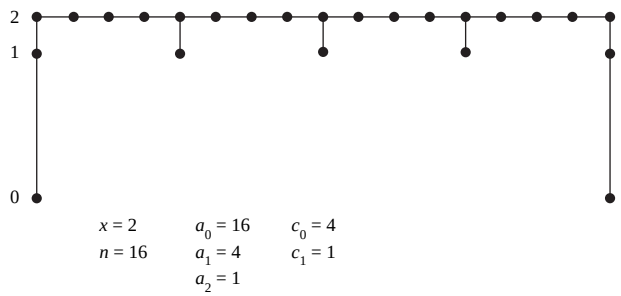
FIGURA 12-6 Entrada ejemplo σ.



Un árbol de expansión mínima de esta entrada puede construirse como sigue:

1. Cada punto en la capa x está unido horizontalmente con sus vecinos.
2. Todos los demás puntos están unidos verticalmente con sus vecinos. (Consulte la figura 12-7, donde se presenta un árbol de expansión mínima de los puntos que se muestran en la figura 12-6.)

FIGURA 12-7 Árbol de expansión mínima de los puntos en la figura 12-6.



La longitud total de este árbol es

$$n + \sum_{i=0}^{x-1} c_i \left(\frac{n}{a_i} + 1 \right) = n + \sum_{i=0}^{x-1} a_{i+1} \left(\frac{n}{a_i} + 1 \right)$$

$$= n + \sum_{i=0}^{x-1} a_{i+1} + n \sum_{i=0}^{x-1} \frac{a_{i+1}}{a_i}$$

$$= n + \sum_{i=0}^{x-1} \frac{n}{x^{2i+2}} + n \sum_{i=0}^{x-1} \frac{1}{x^2}$$

$$\leq n + n \sum_{i=0}^{x-1} \frac{1}{x^2} + n \sum_{i=0}^{x-1} \frac{1}{x^2}$$

$$= n + 2n \frac{x}{x^2}$$

$$\leq 3n.$$

Suponga que los puntos de σ están dados en línea capa por capa, desde la capa 0 y terminando con la capa x. Sea T_{i-1} la gráfica del algoritmo en línea justo antes de que éste obtenga los puntos en la capa i. Hay $\left(\frac{n}{a_i} + 1\right) \geq \frac{n}{a_i}$ puntos dispuestos en la capa i. Observe que la distancia entre dos puntos consecutivos en la capa i es a_i. Para todo punto, la longitud de cualquier arista de $T_i - T_{i-1}$ debe ser por lo menos a_i. Es decir, para la capa i, la longitud total del árbol de expansión incrementada por el algoritmo en línea es por lo menos

$$\frac{n}{a_i} \cdot a_i = n.$$

Así, la longitud total del árbol construido por el algoritmo en línea es por lo menos $n \cdot x$.

Debido a que $C_{off} \leq 3n$ y $C_{onl} \geq nx$, se tiene

$$\frac{C_{onl}}{C_{off}} \geq \frac{1}{3n} \cdot nx = \frac{1}{3} x > \frac{\log n}{6 \log \log n}.$$

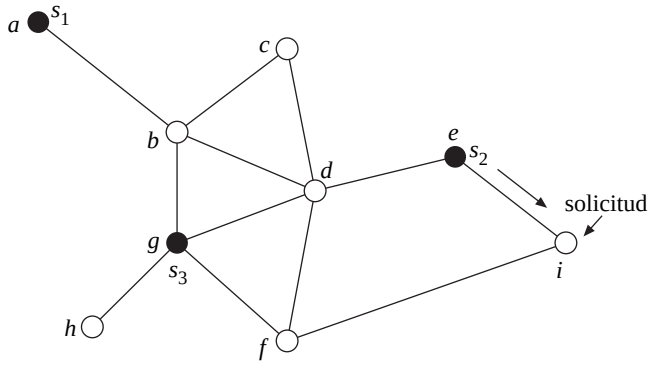
Esto significa que una cota inferior de la tasa de efectividad de este problema es $O(\log n / \log \log n)$. Debido a que nuestro algoritmo es $O(\log n)$ competitivo, no es óptimo.

12-2 EL PROBLEMA EN LÍNEA DEL *K*-ÉSIMO SERVIDOR Y UN ALGORITMO CODICIOSO PARA RESOLVER ESTE PROBLEMA DEFINIDO EN ÁRBOLES PLANOS

Se considerará el problema del k-ésimo servidor. Se cuenta con una gráfica con n vértices y donde cada arista está asociada con una longitud positiva. Considere k servidores estacionados en k vértices, donde $k < n$. Dada una secuencia de peticiones de servidores, es necesario decidir cómo mover los servidores con el fin de satisfacer las

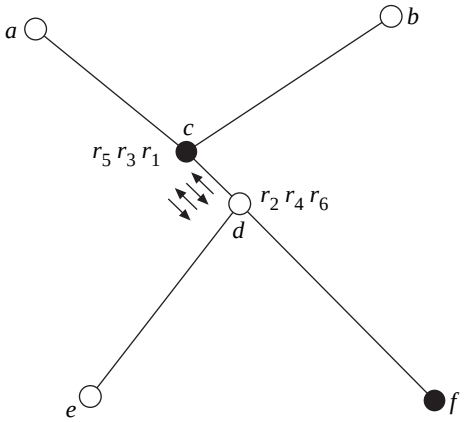
peticiones. El costo de servir una solicitud son las distancias totales entre los servidores movidos para satisfacer la petición. Considere la figura 12-8. Hay tres servidores s_1, s_2 y s_3, ubicados en *a*, *e* y *g*, respectivamente. Suponga que se tiene una solicitud en el vértice *i*. Entonces un movimiento posible es mover s_2, que actualmente se encuentra en el vértice *e*, al vértice *i* porque *e* está cerca de *i*.

FIGURA 12-8 Caso del problema de *k*-servidores.



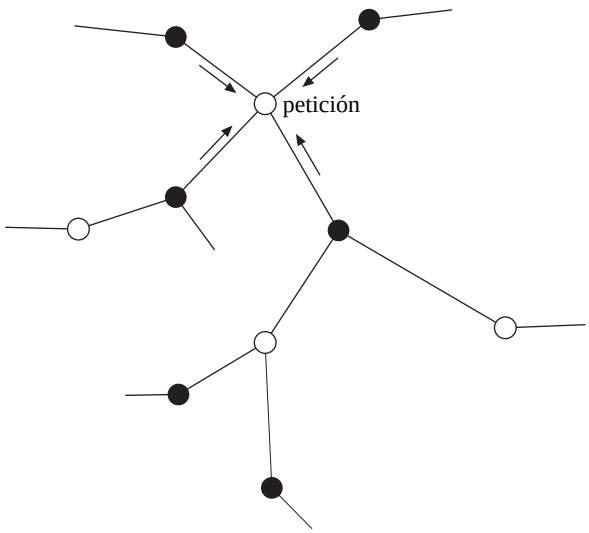
Un algoritmo en línea para resolver este problema de *k*-servidores basado en la estrategia codiciosa debe mover el servidor más cercano al vértice en que se encuentra la petición. El algoritmo en línea de corto alcance presenta un inconveniente, que se muestra en la figura 12-9.

FIGURA 12-9 Un peor caso para el algoritmo en línea de los *k*-servidores.



Suponga que en el vértice *d* hay un servidor y que la primera solicitud r_1 se encuentra en *c*. El algoritmo codicioso movería al servidor ubicado en *d* hacia el vértice *c*. Luego llega la segunda solicitud y desafortunadamente, se encuentra en *d*. Así, ahora el servidor ubicado en *c* se movería al vértice *d*. Como se ilustra en la figura 12-9, si las solicitudes se encuentran alternadamente en *c* y en *d*, entonces el servidor se movería entre *c* y *d* todo el tiempo. De hecho, como se muestra a continuación, si el servidor que está en *f* se mueve gradualmente hacia *f* y *d*, entonces este servidor finalmente se ubicará en *d* en algún tiempo y no ocurre este fenómeno de moverse continuamente entre *c* y *d*. El algoritmo codicioso modificado para resolver el problema mueve muchos servidores activos hacia el vértice en que se encuentra actualmente la solicitud. Ahora la situación se restringirá al problema de *k* servidores a un árbol plano *T*. Debido a que *T* es un árbol plano, las longitudes de arista satisfacen la desigualdad del triángulo. Sean *x* y *y* dos puntos en *T*. La distancia entre *x* y *y*, denotada por $|x, y|$, se define como la longitud de la ruta simple que va de *x* a *y* a través de *T*. Sean s_i y d_i el servidor *i* y la ubicación del servidor *i*, respectivamente. Sea el intervalo $(x, y]$ de *x* a *y* a través de *T*, excluyendo a *x*. El servidor s_i se denomina activo con respecto a alguna solicitud ubicada en *x* si en el intervalo $(d_i, x]$ no hay más servidores. El algoritmo codicioso en línea modificado de *k* servidores funciona como sigue: cuando una solicitud se encuentra en *x*, todos los servidores activos con respecto a *x* se mueven de manera continua a la misma velocidad hacia *x* hasta que uno llega a *x*. Si durante este periodo de movimiento un servidor activo se vuelve inactivo, entonces se detiene el procedimiento. En la figura 12-10 se ilustra el algoritmo codicioso mencionado, en respuesta a una solicitud.

FIGURA 12-10 El algoritmo en línea codicioso de *k* servidores modificado.

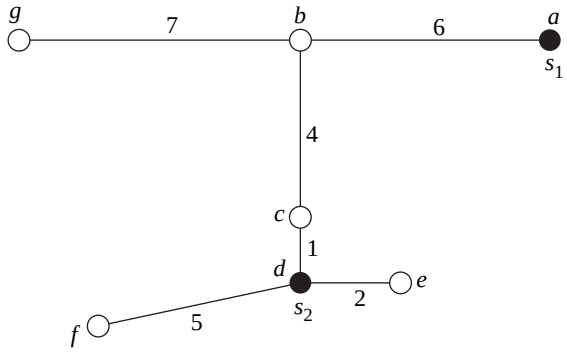


A continuación se presenta un análisis de este algoritmo. Primero se define un algoritmo en línea de *k* servidores completamente informado. Este algoritmo de *k* servidores es totalmente en línea. Es decir, después de cada solicitud, el algoritmo mueve un servidor al vértice en que se encuentra la solicitud. No obstante, debido a que el algoritmo es completamente informado, cuenta con toda la información de la secuencia de solicitudes. En consecuencia, su comportamiento es bastante distinto al de un algoritmo en línea normal que por definición no es completamente informado. De hecho, se trata de un algoritmo óptimo porque produce un resultado óptimo.

Considere la figura 12-11. Ahí hay dos servidores, s_1 y s_2, ubicados en *a* y *d*, respectivamente. Suponga que la secuencia de solicitudes es como sigue:

1. r_1 en *b*.
2. r_2 en *e*.

FIGURA 12-11 Caso que ilustra un algoritmo en línea de *k* servidores completamente informado.



Para un algoritmo en línea codicioso que no cuenta con información "completa", el movimiento de los servidores es como sigue:

1. Mover s_2 desde *d* hasta *b*. El costo es 5.
2. Mover s_2 desde *b* hasta *e*. El costo es 7.

Por consiguiente, el costo total es 12.

Para el algoritmo en línea completamente informado, éste haría lo siguiente:

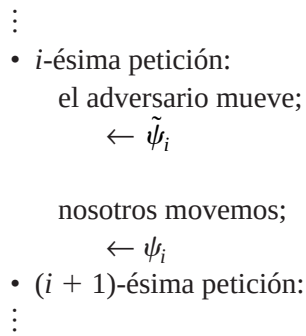
1. Mover s_1 desde *a* hasta *b*. El costo es 6.
2. Mover s_2 desde *d* hasta *e*. El costo es 2.

El costo total es de sólo 8. Este algoritmo en línea completamente informado mueve s_1 en vez de s_2 desde el principio porque sabe que r_2 se encuentra en e.

Nuestro análisis consiste en comparar el costo del algoritmo en línea codicioso modificado de k servidores con el costo del algoritmo en línea de k servidores completamente informado. Suponga que se está llevando a cabo una partida. Cada vez que llega una petición, nuestro adversario, que está completamente informado, realiza su jugada. Luego es nuestro turno. Resulta importante observar que nuestro adversario sólo mueve un servidor, mientras nosotros movemos todos los servidores activos con respecto a la petición. Así como se hizo en el análisis amortizado, se define una función potencial que transforma todas las ubicaciones de nuestros servidores y las ubicaciones de los k servidores de nuestro adversario en un número real no negativo.

Sea $\tilde{\psi}_i$ el valor de la función potencial después de que el adversario mueve en respuesta hacia la i-ésima petición y antes de que nuestro algoritmo realice cualquier movimiento para la i-ésima petición. Sea ψ_i el valor de la función potencial después de que nuestro algoritmo realiza el movimiento después de la i-ésima petición y antes de la $(i + 1)$-ésima petición. Sea ψ_0 el valor potencial de la función potencial. Estos términos pueden ilustrarse mejor como se muestra en la figura 12-12.

FIGURA 12-12 El valor de las funciones potenciales con respecto a las solicitudes.



Sean O_i y A_i el costo de nuestro algoritmo y el costo del algoritmo del adversario, respectivamente, para la i-ésima petición. Sean O y A el costo total de nuestro algoritmo y el costo del algoritmo del adversario, respectivamente, después de que se han hecho todas las peticiones. Entonces se intentará demostrar lo siguiente.

1. $\tilde{\psi}_i - \psi_{i-1} \leq \alpha A_i$, $\quad 1 \leq i \leq n$, para alguna α.
2. $\tilde{\psi}_i - \psi_i \geq \beta O_i$, $\quad 1 \leq i \leq n$, para alguna β.

Las ecuaciones anteriores pueden transformarse en:

$$\begin{aligned}
\tilde{\psi}_1 - \psi_0 &\leq \alpha A_1 \\
\psi_1 - \tilde{\psi}_1 &\leq -\beta O_1 \\
\tilde{\psi}_2 - \psi_1 &\leq \alpha A_2 \\
\psi_2 - \tilde{\psi}_2 &\leq -\beta O_2 \\
&\vdots \\
\tilde{\psi}_n - \psi_{n-1} &\leq \alpha A_n \\
\psi_n - \tilde{\psi}_n &\leq -\beta O_n.
\end{aligned}$$

Al sumar, se obtiene

$$\begin{aligned}
\psi_n - \psi_0 &\leq \alpha(A_1 + A_2 + \cdots + A_n) - \beta(O_1 + O_2 + \cdots + O_n) \\
\beta O &\leq \alpha A + \psi_0 - \psi_n.
\end{aligned}$$

Debido a que $\psi_n \geq 0$,

$$\begin{aligned}
\beta O &\leq \alpha A + \psi_0 \\
O &\leq \frac{\alpha}{\beta} A + \frac{1}{\beta} \psi_0 .
\end{aligned}$$

Ahora se define la función potencial. En cualquier caso temporal, considere que los k servidores de nuestro algoritmo están ubicados en $b_1, b_2, \ldots, b_k$, y que los k servidores del algoritmo de nuestro adversario están localizados en $a_1, a_2, \ldots, a_k$. Se define una gráfica bipartita con componentes $v_1, v_2, \ldots, v_k$ y $v_1', v_2', \ldots, v_k'$, donde $v_i(v_i')$ representa $b_i(a_i)$ y el peso de la arista (v_i, v_j') es $|b_i, a_j|$. En esta gráfica bipartita es posible ejecutar un apareamiento ponderado mínimo. Sea $L_{\text{mín}}$ el apareamiento ponderado mínimo sobre esta gráfica bipartita. La función potencial se define como

$$\psi = k|M_{\text{mín}}| + \sum_{i<j} |b_i, b_j|$$

Se considerará $\tilde{\psi}_i - \psi_{i-1}$. Debido a que sólo el adversario realiza un movimiento, sólo cambia una a_i. Por consiguiente, en ψ, el único incremento posible está relacionado con el cambio de $|M_{\text{mín}}|$, que es igual a A_i. Así, se tiene

$$\tilde{\psi}_i - \psi_{i-1} \leq kA_i \qquad \textbf{(12-1)}$$

Luego, considere $\psi - \tilde{\psi}_i$. Para la i-ésima petición, se mueven q, $q \leq k$ servidores de nuestro algoritmo. Ahora considere que cada uno de los q servidores se mueve una distancia d. Para $M_{\text{mín}}$, por lo menos una arista apareada se reduce a cero porque ambos algoritmos deben satisfacer la i-ésima petición y la distancia entre un par de v_a y v_b es cero. Esto significa que $M_{\text{mín}}$ se incrementa cuando mucho $-d + (q - 1)d = (q - 2)d$.

Luego se calcula el incremento de $\sum_{i<j} |b_i, b_j|$. Sea d_p la ubicación de uno de los servidores activos de nuestro algoritmo. Sea b_r la ubicación de un servidor tal que d_p está entre la i-ésima petición y b_r. Sea l_p el número de b_r. Debido a que el servidor localizado en d_p se mueve una distancia d, la suma de la distancia entre d_p y l_p b_r se incrementa por $(l_p - 1)d$. Sin embargo, para los otros $(k - l_p)$ servidores de nuestro algoritmo, la suma de distancias disminuye por $(k - l_p)d$. Así, la distancia total entre nuestros servidores se incrementa cuando mucho por $\sum_{p=1}^{q} (l_p - 1 + l_p - k)d$, ya que hay q servidores que se mueven para la i-ésima petición. En consecuencia, el cambio de función potencial es cuando mucho

$$k(q - 2)d + \sum_{p=1}^{q} (l_p - 1 + l_p - k)d = -qd. \quad \text{(Observe que } \sum_{p=1}^{q} l_p = k.\text{)}$$

Es decir, se tiene

$$\psi_i - \tilde{\psi}_i \leq -qd.$$

O bien, de manera equivalente,

$$\tilde{\psi}_i - \psi_i \geq O_i \qquad \textbf{(12-2)}$$

Al combinar las ecuaciones (12-1) y (12-2), se tiene

$$O \leq kA + \psi_0$$

que es lo que se buscaba. Así, nuestro algoritmo en línea para el problema de k-servidores es k-competitivo.

Se ha demostrado que el algoritmo en línea para el problema de k-servidores es k-competitivo. De manera natural surge la pregunta: nuestro algoritmo, ¿es óptimo? Es decir, ¿puede haber algún otro algoritmo en línea para el problema de k-servidores cuyo desempeño sea mejor? Por ejemplo, ¿puede haber algún algoritmo en línea $k/2$-competitivo para el problema de k-servidores? Se demostrará que esto es imposible. En otras palabras, nuestro algoritmo en línea para el problema de k-servidores es, en efecto, óptimo.

Primero se observa que nuestro algoritmo en línea se define sobre un árbol plano donde las longitudes de las aristas satisfacen la desigualdad del triángulo. Además, nuestro algoritmo mueve varios servidores, en vez de uno solo. Para efectos de demostrar que nuestro algoritmo en línea es óptimo para el problema de *k*-servidores, ahora se define un término denominado "algoritmo en línea perezoso para el problema de *k*-servidores". Un algoritmo en línea para el problema de *k*-servidores se denomina *perezoso* (lazy) si para manipular cada petición sólo mueve un servidor.

Resulta fácil darse cuenta de que un algoritmo en línea para el problema de *k*-servidores que no es perezoso puede modificarse en un algoritmo en línea perezoso para el problema de *k*-servidores sin incrementar su costo si el algoritmo se ejecuta en una gráfica que satisface la desigualdad del triángulo. Sea *D* un algoritmo en línea para el problema de *k*-servidores que no es perezoso. Suponga que *D* mueve un servidor *s* desde *v* a *w* con el fin de satisfacer cierta petición. Pero, debido a que *D* no es perezoso, antes de este paso *D* mueve a *s* desde *u* hasta *v* aun cuando en *v* no hay ninguna petición. El costo de mover *s* desde *v* a *w* es $d(u, v) + d(v, w)$, que es mayor o igual a $d(u, w)$ debido a la desigualdad del triángulo. En consecuencia, es posible eliminar el paso de mover *s* desde *u* hasta *v* sin aumentar el costo.

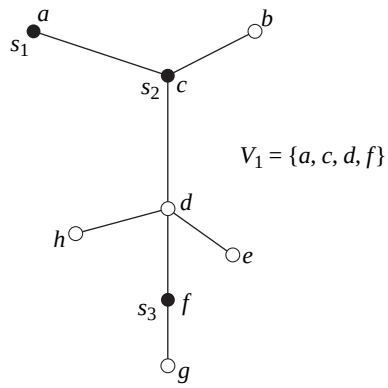
En el análisis que se presenta a continuación, se supondrá que nuestro algoritmo en línea para el problema de *k*-servidores es perezoso.

Sea *D* cualquier algoritmo en línea para el problema de *k*-servidores. Sea $G(V, E)$ la gráfica que contiene *k*-servidores localizados en *k* vértices distintos de *G*. Sea v_1 un subconjunto de *V* que contiene *k* vértices, donde los *k* servidores inicialmente están ubicados en otro vértice arbitrario *x* de *G*. Una secuencia de peticiones se define como sigue.

1. $\sigma(1)$ está ubicado en *x*.
2. $\sigma(i)$ es el único vértice en v_1 que no está cubierto por *D* en el tiempo *i* para $i \geq 1$.

Considere la figura 12-13.

FIGURA 12-13 Ejemplo para el problema de 3 servidores.



Suponga que $k = 3$ y que inicialmente los tres servidores están ubicados en a, c y f. Sea d el otro vértice escogido. Así, $v_1 = \{a, c, f, d\}$. Luego, la primera petición $\sigma(1)$ debe ser d. Suponga que D vacía c al mover s_2 desde c hasta d. Entonces $\sigma(2) = c$. Si además D mueve s_1 desde a hasta c, entonces $\sigma(3) = a$. Con este tipo de secuencia de peticiones, el costo de servir a D en $\sigma(1)$, $\sigma(2)$, …, $\sigma(t)$ es

$$C_D(\sigma, t) = \sum_{i=1}^{t} d(\sigma(i + 1), \sigma(i)).$$

Ahora se define un algoritmo en línea para k-servidores que está más informado que el algoritmo D. Este algoritmo se denomina algoritmo E. El algoritmo E está más informado porque algunas veces conoce a $\sigma(1)$. Sean v_2 $\sigma(1)$ y cualquier subconjunto de $v_1 - \sigma(1)$ con $k - 1$ vértices. Por ejemplo, para el caso anterior, $v_1 = \{a, c, f, d\}$. Debido a que $\sigma(1) = d$, v_2 puede ser $\{a, c, d\}$, $\{c, f, d\}$ o $\{a, f, d\}$. Teniendo en cuenta este subconjunto de vértices v_2, es posible definir un algoritmo en línea para k-servidores $E(v_2)$, que está más informado que D como sigue:

> Si v_2 contiene a $\sigma(1)$, no se hace nada; en caso contrario, mover el servidor que está en el vértice $\sigma(i - 1)$ hacia $\sigma(i)$ y actualizar v_2 a modo de reflejar este cambio. Inicialmente, los servidores ocupan los vértices en v_2.

Observe que en este caso, la condición inicial se ha modificado porque inicialmente un servidor está ubicado donde se encuentra $\sigma(1)$. En estas circunstancias, $E(v_2)$ no tiene que mover a ningún servidor con el fin de satisfacer la primera petición. Es por ello que se dice que $E(v_2)$ está más informado que D. El costo de $E(v_2)$ jamás es superior al de D y por consiguiente puede usarse como una cota inferior de todos los algoritmos en línea para el problema de k-servidores. Resulta fácil ver que para toda $i > 1$, v_2 siempre contiene a $\sigma(i - 1)$ cuando empieza el paso i.

De nuevo, este concepto se ilustrará con un ejemplo. Considere el caso que se muestra en la figura 12-13. Suponga que v_2 $\{a, c, d\}$ y

$$\sigma(1) = d$$
$$\sigma(2) = e$$
$$\sigma(3) = f$$
$$\sigma(4) = a.$$

Los tres servidores s_1, s_2 y s_3 inicialmente ocupan a, c y d, respectivamente. Así, $E(v_2)$ se comporta como sigue:

1. Debido a que v_2 contiene a $\sigma(1) = d$, no se hace nada.
2. Debido a que v_2 no contiene a $\sigma(2) = e$, mover a e el servidor s_3 ubicado en $\sigma(1) = d$ a e. Sea $v_2 = \{a, c, e\}$.
3. Debido a que v_2 no contiene a $\sigma(3) = f$, mover a f el servidor s_3 ubicado en $\sigma(2) = e$ a f. Sea $v_2 = \{a, c, f\}$.
4. Debido a que v_2 contiene a $\sigma(3) = a$, no se hace nada.

Ya que v_1 contiene $k + 1$ vértices, es posible que haya $\binom{k}{k-1} = k$ v_2 distintas. Así, hay k $E(v_2)$ distintas. De manera natural se plantea la pregunta: ¿Cuál $E(v_2)$ tiene el mejor desempeño? En vez de encontrar el costo del mejor algoritmo $E(v_2)$, se encontrará el costo esperado de estos algoritmos $kE(v_2)$. El costo del mejor algoritmo $E(v_2)$ es menor que el esperado.

Por la definición de los algoritmos $E(v_2)$, el paso 1 no cuesta nada. En el paso $i + 1$, (para $i \geq 1$), cada uno de estos algoritmos no hace nada con el costo o bien mueve un servidor de $\sigma(i)$ a $\sigma(i + 1)$ con costo $d(\sigma(i), \sigma(i + 1))$. De todos los k algoritmos, $\binom{k-1}{k-1} = 1$ de ellos, el único que contiene a $(\sigma(i)$, pero no a $\sigma(i + 1)$, incurre en este costo, y todos los demás no incurren en ningún costo. Así, para el paso $i + 1$, el costo total incrementado por todos estos algoritmos es $d(\sigma(i), \sigma(i + 1))$. El costo total de estos k algoritmos hasta, e incluyendo, a $\sigma(t)$ es

$$\sum_{i=1}^{t} d(\sigma(i), \sigma(i-1)).$$

Así, el costo esperado de $E(v_2)$ es

$$\begin{aligned} E(v_2) &= \frac{1}{k} \sum_{i=2}^{t} d(\sigma(i), \sigma(i-1)) \\ &= \frac{1}{k} \left(d(\sigma(2), \sigma(1)) + d(\sigma(3), \sigma(2)) + \cdots + d(\sigma(t), \sigma(t-1))\right) \\ &= \frac{1}{k} \left(\sum_{i=1}^{t} d(\sigma(i+1), \sigma(i)) - d(\sigma(t+1), \sigma(t)) \right) \\ &= \frac{1}{k} \left(C_D(\sigma, t) - d(\sigma(t+1), \sigma(t))\right) \end{aligned}$$

debido a que $C_D(\sigma, t) = \sum_{i=1}^{t} d(\sigma(i+1), \sigma(i))$.

El costo del mejor algoritmo $E(v_2)$ debe ser menor que este costo esperado. Sea $E(v_2')$ dicho costo. El costo de $E(v_2')$ se denota por $C_{E(v_2')}$. Así,

$$C_{E(v_2)} \geq C_{E(v_2')}.$$

Esto significa que

$$\frac{1}{k} C_D(\sigma, t) - \frac{1}{k} d(\sigma(t+1), \sigma(t)) \geq C_{E(v_2')}(\sigma, t).$$

Es decir,

$$C_D(\sigma, t) \geq k \cdot C_{E(v_2')}(\sigma, t).$$

o bien, de manera equivalente,

$$\frac{C_D(\sigma, t)}{C_{E(v_2')}(\sigma, t)} \geq k.$$

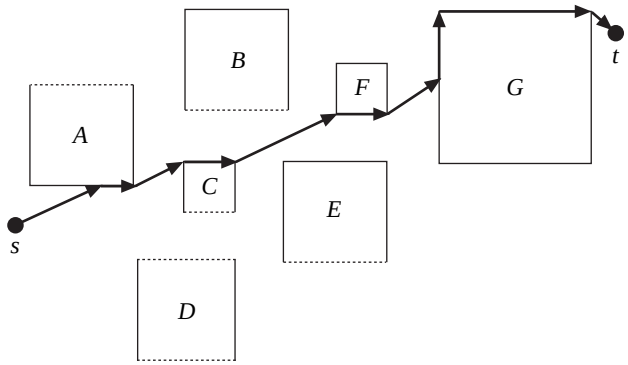
Las ecuaciones anteriores indican que para cualquier algoritmo en línea para k servidores que no está informado en absoluto, existe un algoritmo en línea para k servidores más informado cuyo costo es k veces menor. Esto demuestra que el algoritmo en línea codicioso para k servidores definido sobre árboles planos no puede ser mejor.

12-3 UN ALGORITMO EN LÍNEA PARA EL RECORRIDO DE OBSTÁCULOS BASADO EN LA ESTRATEGIA DEL EQUILIBRIO (O BALANCE)

Se considerará el problema del recorrido de obstáculos. Se tiene un conjunto de obstáculos cuadrados. Todos los lados de estos cuadrados son paralelos a los ejes. La longitud de cada lado de los cuadrados es menor que o igual a 1. Hay un punto de partida, denotado por *s*, y un punto meta, denotado por *t*. La tarea consiste en encontrar la ruta más corta de *s* a *t* que evite los obstáculos. En la figura 12-14 se ilustra un ejemplo típico.

El algoritmo que se presenta aquí para resolver este problema es un algoritmo en línea. Es decir, no se tiene toda la imagen de todos los obstáculos. Así, resulta imposible obtener una solución óptima. El buscador empieza desde *s* y utiliza muchas reglas heurísticas para guiarse. Estas heurísticas constituyen el algoritmo en línea. En el resto de esta sección se describirán tales reglas heurísticas.

FIGURA 12-14 Ejemplo del problema del recorrido.

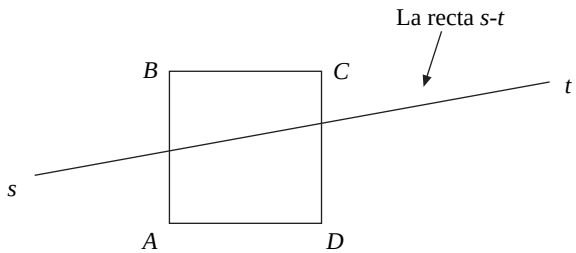


Primero se observa que la recta que une *s* y *t* es la mínima distancia geométrica posible entre *s* y *t*. Entonces, esta distancia, denotada por *d*, sirve como una cota inferior de nuestra solución. Después, las soluciones se compararán con *d*. Se demostrará que la distancia recorrida por el buscador usando nuestro algoritmo en línea no es mayor que $\frac{3}{2}d$ cuando *d* es grande.

Se supone que la recta de *s* a *t* forma un ángulo $\psi \leq \frac{\pi}{4}$ con el eje horizontal. En caso contrario, forma un ángulo $\psi' \leq \frac{\pi}{4}$ con el eje vertical y resulta evidente que nuestro algoritmo sigue siendo válido con una ligera modificación.

Sea α la dirección de *s* a *t*. A continuación se presentarán todas las reglas relacionadas con los diversos casos. Primero se verá con cuántos casos es necesario tratar. Considere la figura 12-15.

FIGURA 12-15 Un obstáculo *ABCD*.



Hay por lo menos tres casos:

Caso 1: El buscador se desplaza entre obstáculos.

Caso 2: El buscador choca con el lado horizontal de un obstáculo. Es decir, choca contra *AD*.

Caso 3: El buscador choca con el lado vertical de un obstáculo. Es decir, choca contra *AB*.

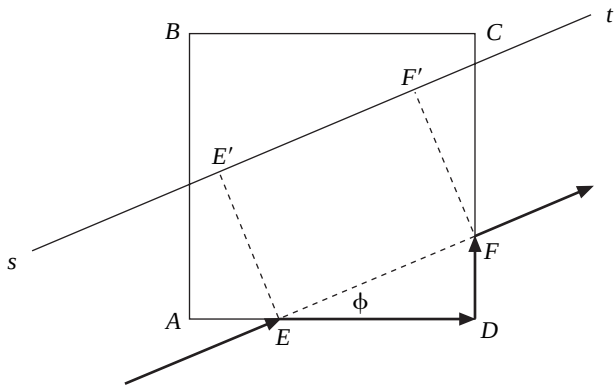
Para el caso 1 se cuenta con la regla 1.

REGLA 1: Cuando el buscador recorre los obstáculos, lo hace en la dirección α. Es decir, se desplaza como si no hubiese obstáculos.

Una vez que se tiene esta regla, puede imaginarse que hay dos buscadores, denotados por *P* y *Q*. El buscador *P* es nuestro buscador real que pasa por todos los obstáculos y luego intenta evitarlos rodeándolos. El buscador *Q* es un buscador ficticio que va del punto *s* al punto *t* a lo largo de la recta que une *s* y *t*. Se desea comparar la distancia recorrida por *P* con la distancia recorrida por *Q*.

REGLA 2: Cuando el buscador choca con el lado horizontal de un cuadrado *AD* en el punto *E*, va de *E* a *D* y luego sube hasta *F* de modo que *EF* es paralela a la recta *s*-*t*, como se ilustra en la figura 12-16. Luego de eso toma la dirección α.

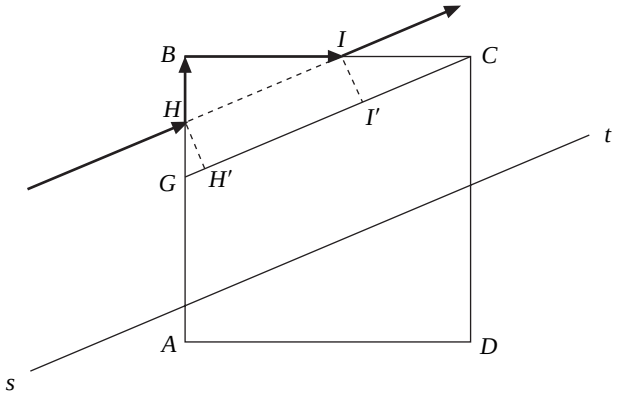
FIGURA 12-16 Choque con el lado *AD* en *E*.



Nuestro buscador P recorre la distancia $|ED| + |DF|$. El buscador ficticio Q, durante el mismo periodo, habrá recorrido la distancia $|EF|$, como se ilustra en la figura 12-16. $(|ED| + |DF|)/|EF| = \cos\psi + \text{sen}\,\psi \leq \frac{3}{2}$. Así, se concluye que si nuestro buscador choca con el lado horizontal de un cuadrado, entonces la distancia que recorre no es mayor que $\frac{3}{2}$ de la distancia recorrida por el buscador ficticio Q.

Si el buscador P choca contra el lado vertical, el caso se vuelve más complicado. Primero se traza una recta que pasa por C, en dirección α hasta que choca contra AB en G, como se ilustra en la figura 12-17.

FIGURA 12-17 Choque con el lado AB en H.



Entonces se tiene la regla 3.

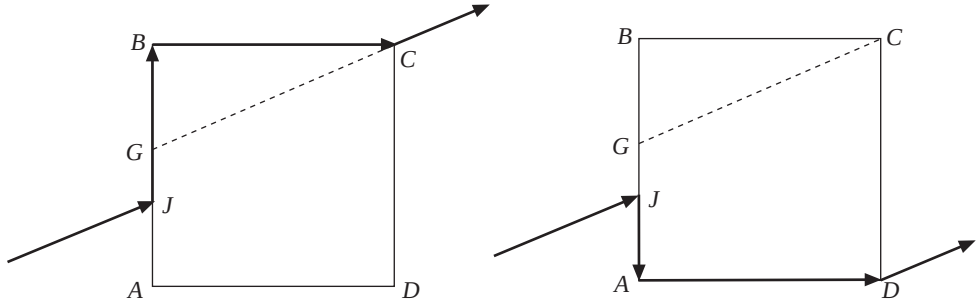
REGLA 3: Si el buscador P choca con AB dentro del intervalo BG en H, debe dirigirse hasta B, moverse hacia la derecha hasta que choca con I de modo que HI sea paralelo a la recta s-t, lo cual se ilustra en la figura 12-17. Luego de eso toma la dirección α.

De nuevo, resulta fácil demostrar que la tasa de la distancia recorrida por el buscador P y la distancia recorrida por el buscador ficticio Q no es mayor que $\frac{3}{2}$ para este caso.

Para el caso en que el buscador P choque con el intervalo AG, existen reglas más complicadas. De hecho, estas reglas constituyen el núcleo de la estrategia del equilibrio.

REGLA 4: Si el buscador P choca con AB dentro del intervalo AG, entonces sube o baja. Si sube, se dirige a B, dobla a la derecha hasta que choca con el vértice C como se muestra en la figura 12-18*a*). Si baja, se dirige hacia A, dobla a la derecha hasta que choca con el vértice D como se muestra en la figura 12-18*b*). Luego de chocar contra el vértice, toma la dirección α.

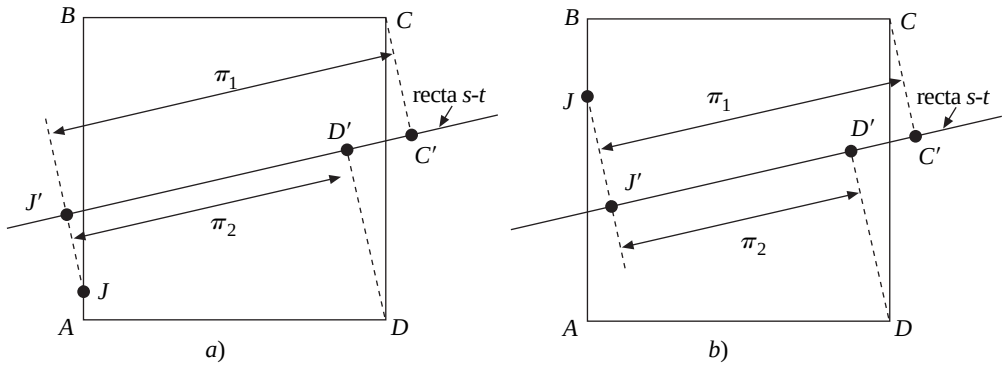
FIGURA 12-18 Dos casos para chocar con el lado AB.



La regla 4 sólo establece que el buscador sube o baja si choca con el intervalo AG. Ahora la pregunta es: ¿cómo decidir si se sube o se baja?

La distancia recorrida por P es $\tau_1 = |JB| + |BC|$ si sube o $\tau_2 = |JA| + |AD|$ si baja. Para este periodo, la distancia recorrida por nuestro buscador ficticio Q se denota por π_1 o por π_2 para subir o bajar, respectivamente. Las distancias π_1 y π_2 se obtienen al proyectar los puntos J, C y D sobre la recta s-t, lo cual se ilustra en la figura 12-19. En la figura 12-19*a*), J está abajo de la recta s-t y en la figura 12-19*b*), arriba de la recta s-t.

FIGURA 12-19 La ilustración para π_1 y π_2.



Sería deseable si tanto $\frac{\tau_1}{\pi_1}$ como $\frac{\tau_2}{\pi_2}$ no son mayores que $\frac{3}{2}$. Pero no es el caso. Considere la situación que se ilustra en la figura 12-20. Suponga que la dirección de nuestro buscador es hacia arriba. Entonces es evidente que $\frac{\tau_1}{\pi_1}$ es casi 2. De manera semejante, considere la situación que se ilustra en la figura 12-21. Si la decisión es bajar, entonces $\frac{\tau_2}{\pi_2}$ nuevamente es casi 2.

FIGURA 12-20 El caso para el peor valor de τ_1/π_1.

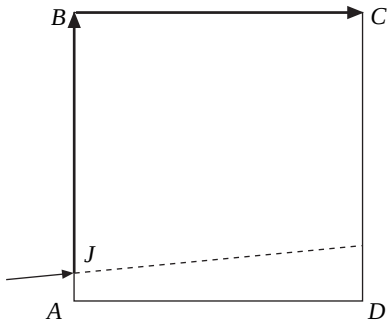
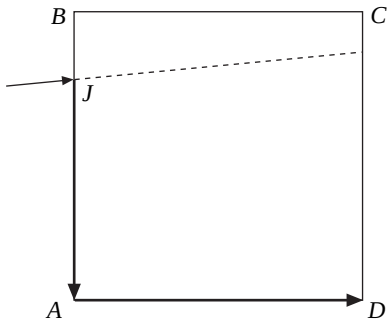


FIGURA 12-21 El caso para el peor valor de τ_2/π_2.



En 1978 se demostró que por lo menos uno de $\frac{\tau_1}{\pi_1}$ y $\frac{\tau_2}{\pi_2}$ no es mayor que $\frac{3}{2}$. Observe que el punto de choque *J* está abajo o arriba de la recta *s*-*t*. Parece que si *J* está arriba de la recta *s*-*t* y $\frac{\tau_1}{\pi_1} \leq \frac{3}{2}$, entonces es necesario desplazarse hacia arriba y si *J*

está abajo de la recta *s*-*t* y $\frac{\tau_2}{\pi_2} \leq \frac{2}{3}$, entonces es necesario desplazarse hacia abajo. No obstante, esto aún puede provocar problemas porque podría estar desplazándose continuamente hacia arriba y hacia abajo, lo cual no necesariamente es saludable, ya que podría estar alejándose cada vez más de la recta *s*-*t*.

Sea k la longitud del lado de un cuadrado, donde $k \leq 1$. El intervalo AG se divide en segmentos iguales de longitud $k/\sqrt{d}$, donde d es la longitud de la recta *s*-*t* según se definió antes. Un segmento se identifica hacia arriba si su punto más bajo satisface $\frac{\tau_1}{\pi_1} \leq \frac{3}{2}$ y se identifica hacia abajo si su punto más bajo satisface $\frac{\tau_2}{\pi_2} \leq \frac{3}{2}$. Un segmento puede identificarse hacia arriba y hacia abajo. Un segmento se denomina "puro" si sólo tiene una identificación y los demás segmentos se denominan "mixtos".

Sea J_i el punto más bajo del i-ésimo segmento y sea J_i' la intersección de la recta que va de J_i en la dirección α y el lado CD. Si el i-ésimo segmento es puro, se definen $\rho_i = |DJ_i'|/|CJ_i'|$ si este segmento sólo está identificado como arriba y $\rho_i = |CJ_i'|/|DJ_i'|$ si este segmento sólo está identificado como abajo. El algoritmo preserva un parámetro, denominado *equilibrio* (o *balance*) con todos los segmentos. Inicialmente, a todos los equilibrios se asigna el valor 0. Como se demostrará después, el *equilibrio* se utiliza para controlar el movimiento hacia arriba o hacia abajo.

REGLA 5: Suponga que el buscador P choca con el intervalo en el punto J que pertenece al i-ésimo segmento.

Caso 1: J está arriba de la recta *s*-*t*. Verificar la identificación del i-ésimo segmento. Si está identificado como abajo, moverse hacia abajo. En caso contrario, comprobar el *equilibrio*. Si $\geq \rho_i k$, moverse hacia abajo y restar $\rho_i k$ de *equilibrio*; en caso contrario, sumar k al *equilibrio* y desplazarse hacia arriba.

Caso 2: J está abajo de la recta *s*-*t*. Verificar la identificación del i-ésimo segmento. Si está identificado como arriba, moverse hacia arriba. En caso contrario, comprobar el *equilibrio*. Si $\geq \rho_i k$, moverse hacia arriba y restar $\rho_i k$ del *equilibrio*; en caso contrario, sumar k al *equilibrio* y desplazarse hacia abajo.

Finalmente, se tiene la regla que sigue.

REGLA 6: Si el buscador P choca con la misma abscisa x o con la misma ordenada y de la meta, va directamente a la meta.

Se ha presentado el algoritmo en línea para recorrer obstáculos. A continuación se analiza su desempeño. Antes que todo, es necesario recordar que si el buscador P cho-

ca con el lado *AD* o con el intervalo *BG* del lado *AB*, entonces la tasa de la distancia recorrida por *P* y la distancia recorrida por *Q* no es mayor que $\frac{3}{2}$. Así, basta restringir el análisis al caso en que se choca con el intervalo *AG*.

Por la definición de segmento mixto, no importa si el buscador *P* sube o baja, ya que $\frac{\tau_1}{\pi_1}$ y $\frac{\tau_2}{\pi_2}$ no es mayor que $\frac{3}{2}$. Así, sólo se considerará el caso en que el segmento es puro. Se considerará el *i*-ésimo segmento. Suponga que está identificado como arriba y que el buscador *P* está arriba de la recta *s*-*t*. El caso en que el *i*-ésimo segmento está identificado como abajo es semejante. Sea $a_i(b_i)$ la suma de los lados de los cuadrados para los cuales el buscador *P* choca con el *i*-ésimo segmento y sube (baja). La distancia total recorrida por el buscador *Q* con respecto al *i*-ésimo segmento es $a_i\pi_1 + b_i\pi_2$, donde las cantidades τ_1, τ_2, π_1 y π_2 son todas con respecto al punto más bajo del *i*-ésimo segmento de un cuadrado unitario. Así,

$$\frac{a_i\tau_1 + b_i\tau_2}{a_i\pi_1 + b_i\pi_2} = \frac{\frac{a_i}{b_i}\tau_1 + \tau_2}{\frac{a_i}{b_i}\pi_1 + \pi_2}.$$

Entonces, la pregunta es: ¿Cuán grande es $\frac{a_i}{b_i}$? Observe que tanto a_i como b_i están relacionados con el equilibrio usado en el algoritmo. En total, $\rho_i b_i$ se resta de *equilibrio* y a_i se suma. Debido a que *equilibrio* nunca es negativo, se tiene $\rho_i b_i \leq a_i$. Debido a que el *i*-ésimo segmento está identificado hacia arriba, por definición se tiene $\frac{\tau_1}{\pi_1} \leq \frac{3}{2}$.

Ahora se intenta encontrar una cota superior de $\frac{\rho_i\tau_1 + \tau_2}{\rho_i\pi_1 + \pi_2}$.

$$\frac{\rho_i\tau_1 + \tau_2}{\rho_i\pi_1 + \pi_2} = \frac{\frac{|DJ_i'|}{|CJ_i'|}\cdot\tau_1 + \tau_2}{\frac{|DJ_i'|}{|CJ_i'|}\cdot\pi_1 + \pi_2}$$

$$= \frac{|DJ_i'|\tau_1 + |CJ_i'|\tau_2}{|DJ_i'|\pi_1 + |CJ_i'|\pi_2}.$$

Suponga que se tiene un cuadrado unitario. El numerador

$$
\begin{aligned}
&|DJ_i|\tau_1 + |CJ_i|\tau_2 \\
&= (|AJ_i| + \tan\psi)(2 - |AJ_i|) + (1 - |AJ_i| - \tan\psi)(1 + |AJ_i|) \\
&= \frac{1}{\cos\psi}((\operatorname{sen}\psi + \cos\psi) + 2(\cos\psi - \operatorname{sen}\psi)|AJ_i| - 2\cos\psi\,(|AJ_i|)^2).
\end{aligned}
$$

Sea $|AJ_i| = y$, $\operatorname{sen}\psi = a$ y $\cos\psi = b$.

$$
\begin{aligned}
&|DJ_i'|\tau_1 + |CJ_i'|\tau_2 \\
&= \frac{1}{b}((a + b) + 2(b - a)y - 2by^2).
\end{aligned}
$$

El denominador

$$
\begin{aligned}
&|DJ_i'|\pi_1 + |CJ_i'|_{\ 2} \\
&= (|AJ_i| + \tan\psi)(\operatorname{sen}\psi + \cos\psi - |AJ_i|\operatorname{sen}\psi) \\
&\quad + (1 - |AJ_i| - \tan\psi)(\cos\psi - |AJ_i|\operatorname{sen}\psi) \\
&= \frac{1}{\cos\psi}(\operatorname{sen}^2\psi + \cos^2\psi) \\
&= \frac{1}{\cos\psi} \\
&= \frac{1}{b}.
\end{aligned}
$$

Así,

$$
\frac{|DJ_i'|\tau_1 + |CJ_i'|\tau_2}{|DJ_i'|\pi_1 + |CJ_i'|\pi_2} = (a + b) + 2(b - a)y - 2by^2.
$$

Sea $f(y) = (a + b) + 2(b - a)y - 2by^2$. Entonces

$f'(y) = 2(b - a) - 4by$.

Cuando $y = \dfrac{b-a}{2b}$, $f(y)$ se maximiza.

$$f\left(\frac{b-a}{2b}\right) = (a+b) + 2(b-a)\frac{b-a}{2b} - 2b\left(\frac{b-a}{2b}\right)^2$$
$$= (a+b) + \frac{(b-a)^2}{b} - \frac{(b-a)^2}{2b}$$
$$= (a+b) + \frac{(b-a)^2}{2b}$$
$$= \frac{2b^2 + b^2 + a^2}{2b}.$$

Ya que $a^2 + b^2 = \text{sen}^2\,\psi + \cos^2\,\psi = 1$, se tiene

$$f\left(\frac{b-a}{b}\right) = \frac{2b^2+1}{2b}.$$

Debido a que $b = \cos\,\psi$ y $\psi \leq \pi/4$, se tiene

$$f\left(\frac{b-a}{b}\right) = \frac{2b^2+1}{2b} \leq \frac{3}{2}.$$

Así,

$$\frac{\sigma_i\tau_1 + \tau_2}{\sigma_i\pi_1 + \pi_2} \leq \frac{3}{2}.$$

Resulta fácil ver que la desigualdad anterior sigue siendo verdadera para cuadrados de tamaño k (≤ 1). Así, se tiene

$$\frac{\rho_i\tau_1 + \tau_2}{\rho_i\pi_1 + \pi_2} \leq \frac{3}{2}.$$
$$2\rho_i\tau_1 + 2\tau_2 \leq 3\rho_i\pi_1 + 3\pi_2$$
$$(3\pi_1 - 2\tau_1)\rho_i \geq 2\tau_2 - 3\pi_2$$
$$(3\pi_1 - 2\tau_1)\,\frac{a_i}{b_i} \geq 2\tau_2 - 3\pi_2 \quad (\text{por } \rho_i b_i \leq a_i)$$

$$\frac{\frac{a_i}{b_i}\tau_1 + \tau_2}{\frac{a_i}{b_i}\pi_1 + \pi_2} \leq \frac{3}{2}. \quad \left(\text{por } \frac{\pi_1}{\tau_1} \leq \frac{3}{2}\right).$$

Se supone que el buscador *P* choca con el punto más bajo del *i*-ésimo segmento de un obstáculo. Si choca con un punto que está dentro del *i*-ésimo segmento de un obstáculo, hay un error $O(1/\sqrt{d})$ cada vez que ocurre esta situación. Observe que *d* es la longitud de la recta *s*-*t* y que el tamaño de cada cuadrado se ha limitado a alguna constante *c* (< 1). Así, en el peor de los casos se choca contra $O(d)$ cuadrados. Entonces, el error total es $O(\sqrt{d})$.

Luego, se requiere conocer la distancia que hay entre el buscador *P* y la recta *s*-*t* cuando el algoritmo aplica la regla 6 para llegar a *t*. Suponga que el buscador *P* está arriba de la recta *s*-*t* y que choca contra el *i*-ésimo segmento. Resulta evidente que si el *i*-ésimo segmento está identificado como abajo, entonces se vuelve más cercano a la recta *s*-*t*. Considere el caso en que el *i*-ésimo segmento está identificado como arriba y el buscador *P* sube. Por definición, las veces de suma para equilibrar el *i*-ésimo segmento es cuando mucho $\rho_i + 1$. La distancia vertical total por arriba de la recta *s*-*t* es $|CJ_i'| \times (\rho_i + 1) \leq 1$, donde $|CJ_i'|$ es la distancia vertical lejos de la recta *s*-*t* cuando el buscador *P* choca con el *i*-ésimo segmento. Debido a que en cada obstáculo hay $\sqrt{d}$ segmentos, después de aplicar la regla 6, la distancia entre el buscador *P* y la recta *s*-*t* es cuando mucho $\sqrt{d}$.

Con base en el análisis anterior se concluye que la tasa entre la distancia recorrida por el buscador *P* y el buscador *Q* no es mayor que

$$\frac{3}{2} + O\left(\frac{1}{\sqrt{d}}\right).$$

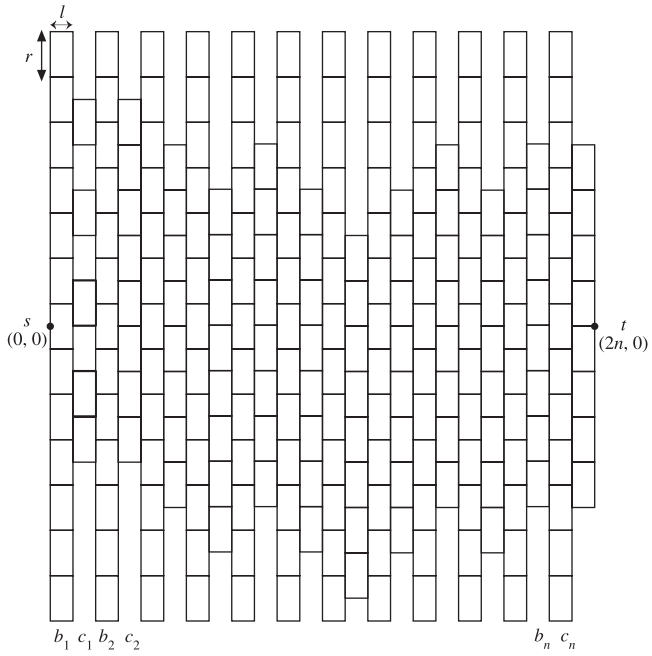
Si *d* es grande, la tasa no es mayor que $\frac{3}{2}$. Así, se concluye que la distancia total recorrida por *P* no es mayor que $\frac{3}{2}$ de la distancia recorrida por el buscador ficticio *Q*. Entonces, nuestro algoritmo es (3/2) competitivo.

En lo anterior se demostró que la tasa de competitividad de nuestro algoritmo para recorrer obstáculos es 3/2. A continuación se demostrará que esto es óptimo. Suponga que los obstáculos son rectángulos y sea *r* la tasa de la longitud al ancho de cada rectángulo. Se demostrará que es imposible tener una tasa de competitividad de cualquier algoritmo en línea para resolver el problema del recorrido de obstáculos que sea menor que $(r/2 + 1)$. En nuestro caso, se supone que todos los obstáculos son cuadrados y que *r* es igual a 1. Así, la tasa de competitividad no puede ser menor que 3/2 y el algoritmo en línea es óptimo.

Para encontrar una cota inferior de la tasa de competitividad es necesario contar con un arreglo especial de los obstáculos. En la figura 12-22 se ilustra este arreglo especial para los obstáculos. Sea *n* un entero mayor que 1. Como se muestra en la figura

12-22, las coordenadas de s y t son (0, 0) y ($2n$, 0), respectivamente. Observe que entre obstáculos hay un ancho infinitamente pequeño que los separa y que el buscador puede comprimirse a sí mismo para desplazarse entre los obstáculos. Resulta evidente que el buscador se mueve de manera horizontal o vertical.

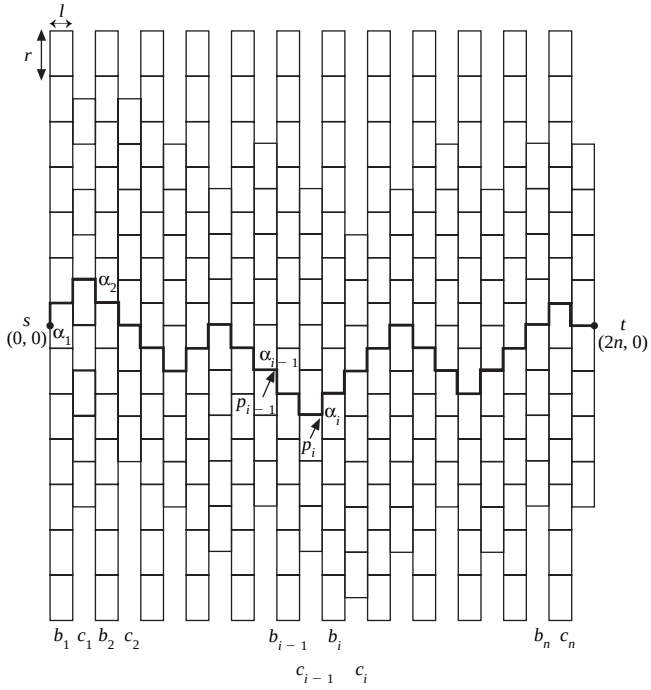
FIGURA 12-22 Arreglo especial.



Los obstáculos están dispuestos en columnas. Hay dos tipos de columnas: las columnas b y las columnas c. Cada columna b consta de infinidad de obstáculos y en cada columna c hay exactamente ocho obstáculos. Cada columna b está próxima a una columna c y cada columna c está próxima a una columna b. La i-ésima columna $b(c)$ se identifica como $b_i(c_i)$.

Para $1 \leq i \leq n$, las ubicaciones de los obstáculos en c_i están dispuestas según la forma en que el buscador choque con b_i. Sea α_i el obstáculo encontrado por el buscador P en la columna b_i. Sea p_i el punto en que el buscador P llega al obstáculo α_i en la columna b_i. Entonces los ocho obstáculos en la columna c_i están dispuestos de modo que α_i está próximo a su punto medio. En la figura 12-23 se muestra una disposición posible de los obstáculos, donde la línea gruesa indica el recorrido del buscador P.

FIGURA 12-23 Ruta para el arreglo de la figura 12-22.



Resulta evidente que cualquier longitud de la ruta más corta de p_{i-1} a p_i es por lo menos

$r + 2$.

Así, la distancia total recorrida por el buscador P es por lo menos

$(r + 2)n$.

Para $0 \leq m \leq \sqrt{n}$, la recta horizontal se define como

L_m: $y = (8m + 1/2)r$.

Observe que L_m no corta a ningún obstáculo sobre b_i, y que puede cortar a un obstáculo sobre c_j para cualquier i y j. Debido a que la distancia entre dos rectas cualquiera L_m y L'_m es por lo menos $8r$, no pueden cortar obstáculos que estén en la misma columna c_i, para $1 \leq i \leq n$. Así, $L_0, L_1, \ldots, L_{\lfloor\sqrt{n}\rfloor}$ puede cortar cuando mucho n obstáculos sobre las columnas $c_1, c_2, \ldots, c_n$. Debido a que hay $\sqrt{n}$ rectas y éstas cortan n obstáculos, en promedio, cada recta corta $n/\sqrt{n} = \sqrt{n}$ obstáculos. Existe por lo menos un

entero $m \leq \sqrt{n}$ tal que L_m corta cuando mucho $\sqrt{n}$ obstáculos sobre las columnas c_1, c_2, …, c_n. Para cualquier solución factible, la distancia horizontal es por lo menos $2n$. Cada vez que choca contra un obstáculo, recorre cr para alguna c porque r es la longitud de cada obstáculo. Así, es posible construir una ruta factible cuya longitud sea a lo sumo

$2n + cr\sqrt{n}$ para alguna c.

Finalmente, puede concluirse que la tasa de competitividad de cualquier algoritmo en línea para el problema del recorrido de obstáculos es por lo menos

$$\frac{r+n}{2n+nc\sqrt{n}}.$$

Si n es grande, esta tasa se convierte en por lo menos

$r/2 + 1$.

Si $r = 1$, esto es igual a $\frac{3}{2}$.

12-4 EL PROBLEMA DE APAREAMIENTO BIPARTITA EN LÍNEA RESUELTO POR LA ESTRATEGIA DE COMPENSACIÓN

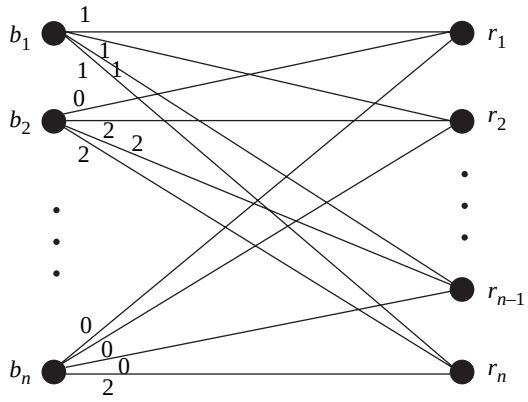
En esta sección se considera un problema de apareamiento bipartita. Se tiene una gráfica G ponderada bipartita con vértices de bipartición R y B, cada uno de cardinalidad n. Un apareamiento bipartita M es un subconjunto de E donde ningún par de aristas en el conjunto es incidente en un solo vértice y cada arista en el conjunto incide tanto en R como en B. El costo de un apareamiento se define como el peso total de las aristas en el apareamiento. El problema de apareamiento bipartita mínimo consiste en encontrar un apareamiento bipartita con el costo mínimo. En esta sección sólo se considerará el problema de apareamiento bipartita mínimo.

Nuestra versión en línea del problema de apareamiento bipartita se describe como sigue: todos los vértices en R se conocen de antemano. Luego, los vértices en B se van revelando uno por uno. Después que llega el i-ésimo vértice en B, debe aparearse con un vértice no apareado en R y esta decisión no puede cambiarse después. Nuestro objetivo es mantener bajo el costo de este apareamiento en línea. También se impone una restricción especial sobre los datos: todos los pesos de las aristas satisfacen la desigualdad del triángulo.

A continuación, primero se probará una cota inferior del costo resultante de algoritmos en línea de apareamiento bipartita mínimo. Se demostrará que el costo de cualquier algoritmo en línea de apareamiento bipartita mínimo no puede ser menor que $(2n - 1)$ veces el costo de un apareamiento fuera de línea óptimo.

Sean $b_1, \ldots, b_n$ los n vértices en B. Sea r_i, $1 \leq i \leq n$, el vértice apareado con b_i cuando aparece b_i. Sea la gráfica bipartita la que se ilustra en la figura 12-24.

FIGURA 12-24 Gráfica bipartita para probar una cota inferior para algoritmos en línea de apareamiento bipartita mínimo.



Como se muestra en la figura 12-24, esta gráfica bipartita posee las siguientes características:

1. El peso de (b_1, r_i) es 1 para toda i.
2. El peso de (b_i, r_j) es 0 para $i = 2, 3, \ldots, n$ si $j < i$.
3. El peso de (b_i, r_j) es 2 para $i = 2, 3, \ldots, n$ si $j \geq i$.

Para cualquier algoritmo en línea con $b_1, b_2, \ldots, b_n$ revelado en orden, existe el costo total de apareamiento $1 + 2(n - 1) = 2n - 1$.

Un apareamiento en línea óptimo para esta gráfica bipartita podría ser como sigue:

r_1 apareado con b_2 con costo 0.
r_2 apareado con b_3 con costo 0.
$\vdots$
r_n apareado con b_1 con costo 1.

Así, el costo óptimo para un algoritmo en línea es 1. Esto significa que ningún algoritmo en línea de apareamiento bipartita puede lograr una tasa de competitividad menor que $2n - 1$.

A continuación se ilustrará un algoritmo en línea de apareamiento bipartita. Este algoritmo de apareamiento en línea se basa en un apareamiento fuera de línea. El algoritmo se describirá por medio de un ejemplo.

Considere la figura 12-25, donde se muestra el conjunto *R*. Observe que ninguno de los nodos está identificado. El peso de la arista entre dos vértices es su distancia geométrica.

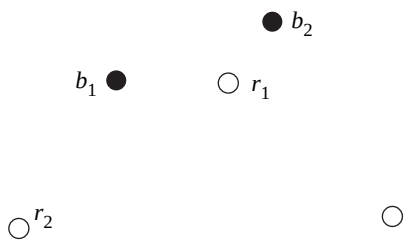
FIGURA 12-25 El conjunto *R*.

Luego, en la figura 12-26, se revela b_1. Cualquier apareamiento en línea óptimo debería encontrar el nodo en *R* más cercano a b_1. Sin pérdida de generalidad, sea r_1 este nodo. Nuestro algoritmo de apareamiento en línea aparea b_1 con r_1 justo como lo haría cualquier apareamiento en línea óptimo.

FIGURA 12-26 La revelación de b_1.

En la figura 12-27, se revela b_2. Así, cualquier apareamiento en línea óptimo aparea b_2 con r_1 y aparea b_1 con un nuevo nodo en *R*. Sin pérdida de generalidad, este nodo puede denotarse por r_2. La pregunta es ¿qué acción realizará nuestro algoritmo de apareamiento en línea?

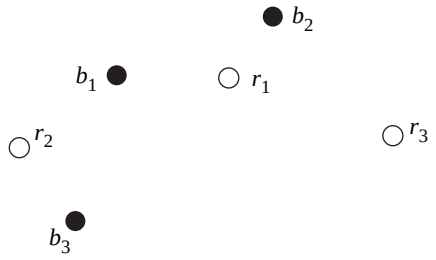
FIGURA 12-27 La revelación de b_2.



Debido a que nuestro algoritmo de apareamiento en línea ya ha apareado b_1 con r_1, este hecho ya no puede modificarse. En compensación, debido a que el nuevo nodo apareado es r_2, simplemente se aparea b_2 con r_2.

Finalmente, b_3 se revela como se muestra en la figura 12-28. Un apareamiento en línea óptimo aparearía b_1 con r_1, b_2 con r_3, el nuevo nodo agregado en R y b_3 con r_2.

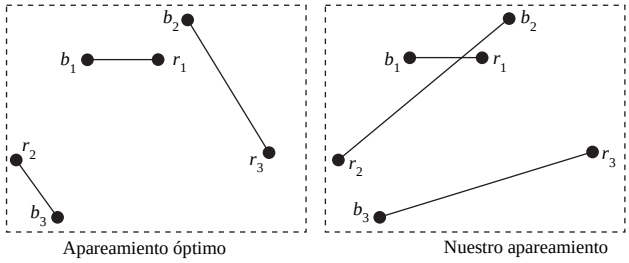
FIGURA 12-28 La revelación de b_3.



Nuestro algoritmo de apareamiento bipartita en línea observa que r_3 es un nuevo punto agregado y aparea b_3 con r_3.

En la figura 12-29 se compara el algoritmo de apareamiento óptimo fuera de línea con nuestro apareamiento en línea.

FIGURA 12-29 Comparación de apareamientos en línea y fuera de línea.



Ahora se define formalmente nuestro algoritmo en línea. Se supondrá que antes que se revele b_i, b_1, b_2, …, b_{i-1} ya se han revelado en este orden y sin pérdida de generalidad, están apareados con r_1, r_2, …, r_{i-1}, respectivamente mediante nuestro apareamiento en línea M_{i-1}. Una vez que se revela b_i, considere el apareamiento M_i', que cumple las siguientes condiciones:

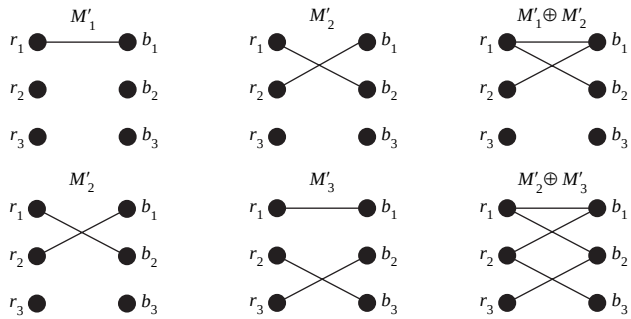
1. M_i' es un apareamiento bipartita óptimo entre $\{b_1, b_2, \ldots, b_i\}$ y $r_1, r_2, \ldots, r_i$.
2. De todos los apareamientos bipartitas óptimos entre $\{b_1, b_2, \ldots, b_i\}$ y $r_1, r_2, \ldots, r_i$, $|M_i' - M'_{i-1}|$ es el menor.

Sea R_i que denota al conjunto de elementos de R que están en M_i'. Puede demostrarse que R_i sólo agrega un elemento a R_{i-1}. Sin pérdida de generalidad, puede suponerse que r_i se agrega después que se revela b_i. Es decir, $R_i = \{r_1, r_2, \ldots, r_{i-1}, r_i\}$. Debido a que M_{i-1}, r_j está apareado con b_j para $j = 1, 2, \ldots, i - 1$, para compensar se aparea r_i con b_i en M_i.

Inicialmente, $M_1 = M_1'$.

A continuación se analizará una propiedad interesante del algoritmo en línea. Observe que para el algoritmo en línea, hay una secuencia de apareamientos M_1', M_2', …, M_n'. Ahora se considerará $M_i' \oplus M_{i+1}'$, donde cada arista está en exactamente uno de M_i' y M_{i+1}'. Para el ejemplo, los conjuntos se muestran en la figura 12-30.

FIGURA 12-30 $M_i' \oplus M_{i+1}'$.



Sean E_i y E_j dos conjuntos de aristas. Una ruta alterna (ciclo alterno) de $E_i \oplus E_j$ es una ruta simple (ciclo simple) donde las aristas alternan en E_i y E_j. Una ruta alterna maximal (ciclo alterno maximal) es una ruta alterna (ciclo alterno) que no es una subruta (subciclo) de ninguna ruta alterna (ciclo alterno). Para $M_1' \oplus M_2'$, hay exactamente una ruta alterna maximal y esta ruta tiene como punto terminal a b_2. Esta ruta es

$$r_2 \rightarrow b_1 \rightarrow r_1 \rightarrow b_2.$$

De manera semejante, en $M_2' \oplus M_3'$, hay exactamente una ruta alterna maximal que termina en b_3; a saber,

$$r_3 \rightarrow b_2 \rightarrow r_1 \rightarrow b_1 \rightarrow r_2 \rightarrow b_3.$$

A continuación se intentará demostrar que en $M'_{i-1} \oplus M'_i$ para $i = 1, 2, \ldots, n$, hay exactamente una ruta alterna maximal y que el punto terminal de esta ruta es b_1.

Sea H igual a $M'_{i-1} \oplus M'_i$. Primero se observa lo siguiente:

1. El grado de b_1 en H es 1 porque b_1 es un nuevo vértice revelado y debe aparearse con algún vértice en R.
2. Para b_j, $1 \leq j \leq i - 1$, se verá que el grado de b_j en H es 0 o 2. Esto puede verse como sigue: Si b_j se aparea con el mismo elemento en R para M'_{i-1} y para M'_i, entonces el grado de b_j en H es 0. En caso contrario, el grado de b_j en H es 2 porque una arista en M'_{i-1} y una arista en M'_i inciden en b_j.
3. Así como para cualquier r_j, el grado de r_j puede ser 0, 1 o 2. Esto puede demostrarse ya que sólo hay dos casos posibles:

   Caso 3.1: r_j no se aparea con ningún vértice tanto en M'_{i-1} como en M'_i. En este caso, el grado de r_j en H es 0.

   Caso 3.2: r_j se aparea con algún vértice. En este caso, puede demostrarse que el grado de r_j en H es 1 si se aparea con un vértice. Ya sea en M'_{i-1} o en M'_i, pero no en ambos, que el grado de r_j en H es 0 si se aparea con el mismo vértice tanto en M'_{i-1} como en M'_i, y que el grado de r_j es 2 si se aparea con dos vértices distintos en M'_{i-1} y en M'_i.

Puede verse fácilmente que en H, cada componente unida debe ser una ruta alterna maximal o un ciclo alterno maximal. Debido a que el grado de b_i es 1, debe haber una ruta alterna maximal cuyo punto terminal sea b_i. Sea P esta ruta. Se demostrará que en H no hay ninguna otra componente unida. En otras palabras, P es la única componente unida en H.

Primero se supone que H contiene una ruta alterna maximal L donde $L \neq P$. Resulta evidente que b_i no está en L porque $L \neq P$, y que P es la única ruta alterna maximal cuyo punto terminal es b_i. Debido a que todo vértice de $B_i - \{b_i\}$ tiene grado 0 o 2 y como cada vértice tiene grado 0, 1 o 2, los dos puntos terminales de L deben estar en R y el número de aristas de L debe ser par. Además, el número de vértices de L que están en R debe ser mayor por 1 que los de L que están en $B_i - \{b_i\}$. Sin pérdida de generalidad se supone que en L hay $2m$ aristas, donde $1 \leq m \leq i - 1$, y que la secuencia de vértices sobre la ruta L es $r_1, b_1, r_2, b_2, \ldots, r_m, b_m, r_{m+1}$, donde r_1 y r_{m+1} son los puntos terminales de L, $r_1, r_2, \ldots$, y r_{m+1} están en R, y $b_1, \ldots$ y b_m están en $B_i - \{b_i\}$. Sea $A_i(A_{i-1})$ que denota el conjunto de aristas en L que pertenecen a $M'_i(M'_{i-1})$. Sin pérdida de generalidad, se supone que $A_i = \{(b_1, r_1), (b_2, r_2), \ldots, (b_m, r_m)\}$ y $A_{i-1} = \{(b_1, r_2),$

$(b_2, r_3), \ldots, (b_m, r_{m+1})\}$. Observe que en H, el vértice de R con grado 1 se aparea exactamente ya sea en M_i' o en M_{i-1}'. Así, el punto terminal r_1 se aparea exactamente en M_i' y el punto terminal r_{m+1} se aparea exactamente en M_{i-1}'. En otras palabras, $M_i'(M_{i-1}')$ es el conjunto de aristas de un apareamiento de $B_i = \{b_1, b_2, \ldots, b_i\}$ ($B_{i-1} = \{b_1, b_2, \ldots, b_{i-1}\}$) con un subconjunto de $R + \{r_{m+1}\}(R - \{r_1\})$.

Debido a que A_i es el conjunto de aristas de un apareamiento de $\{b_1, \ldots, b_m\}$ con $\{r_1, \ldots, r_m\}$, $M_i' - A_i$ es el conjunto de aristas con un apareamiento de $B_i - \{b_1, \ldots, b_m\}$ con un subconjunto de $R - \{r_1, r_2, \ldots, r_m, r_{m+1}\}$. Debido a que A_{i-1} es el conjunto de aristas de un apareamiento de $\{b_1, \ldots, b_m\}$ con $\{r_2, \ldots, r_{m+1}\}$, $(M_i' - A_i) \bigcup A_{i-1}$ es el conjunto de aristas de un apareamiento de b_i con un conjunto de $R - \{r_1\}$. De manera semejante, puede demostrarse que $(M_{i-1}' - A_{i-1}) \bigcup A_i$ es el conjunto de aristas de un apareamiento de B_{i-1} con un subconjunto de $R - \{r_{m+1}\}$. Sea $W(S)$ que denota el peso total de un conjunto de aristas S. Hay tres casos posibles para L:

- $W(A_i) > W(A_{i-1})$.
  En este caso, hay un apareamiento $M_i'' = (M_i' - A_i) \bigcup A_{i-1}$ tal que $W(M_i'') < W(M_i')$. Esto contradice la minimalidad de M_i'.

- $W(A_i) < W(A_{i-1})$.
  En este caso, hay un apareamiento $M_{i-1}'' = (M_{i-1}' - A_{i-1}) \bigcup A_i$ tal que $W(M_{i-1}'') < W(M_{i-1}')$. Esto contradice la minimalidad de M_{i-1}'.

- $W(A_i) = W(A_{i-1})$.
  En este caso, hay un apareamiento $M_i'' = (M_i' - A_i) \bigcup A_{i-1}$ tal que $|M_i'' - M_{i-1}'| < |M_i' - M'_{i-1}|$. Esto contradice el hecho de que M_i' se seleccionó para minimizar el número de aristas en $M_i' - M_{i-1}'$.

Por lo tanto, no hay ninguna ruta alterna maximal, excepto P en H.

Se concluye que $M_i' \oplus M'_{i-1}$ consta de exactamente una ruta alterna maximal P que tiene a b_i como uno de sus puntos terminales.

Debido a que H consta exactamente de una ruta alterna maximal P y el grado de b_j es 0 o 2, donde $1 \leq j \leq i - 1$, hay un vértice r' de R con grado 1 como el otro punto terminal de P y cada vértice de $R - \{r'\}$ tiene grado 0 o 2.

Además, el número de aristas de P debe ser impar. Así, tanto la arista unida al punto terminal b_i como la unida al punto terminal r' deben provenir del mismo conjunto de aristas. Debido a que la arista unida a b_i es una arista de M_i', la arista unida a r' también es de M_i'. Observe que en H, el vértice de R con grado 1 está apareado exactamente en una de M_i' y M_{i-1}'. Así, r' está apareado exactamente en M_i'. En H, un vértice de

$R - \{r'\}$ con grado 0 está apareado con el mismo vértice de b_i o no está apareado ni con M'_i ni con M'_{i-1}, y un vértice de $R - \{r'\}$ con grado 2 está apareado con vértices diferentes de b_i en M'_i y M'_{i-1}. Es decir, un vértice de $R - \{r'\}$ está apareado o no lo está tanto en M'_i como en M'_{i-1}. Así, r' es el único vértice de $R_i - R_{i-1}$. Se concluye que para $1 < i \leq n$, $|R_i - R_{i-1}| = 1$.

El análisis anterior es de suma importancia porque establece los fundamentos de la estrategia de compensación. El algoritmo en línea que se presentó en esta sección sólo funciona si después de que se revela un nuevo elemento *B*, en comparación con el apareamiento óptimo previo, el nuevo apareamiento óptimo agrega sólo un nuevo elemento de *R*.

Se ha demostrado la validez de este método. A continuación se analizará el desempeño del algoritmo en línea. Se demostrará que el algoritmo es $(2n - 1)$-competitivo. Es decir, sean $C(M_n)$ y $C(M'_n)$ los costos de los apareamientos M_n y M'_n, respectivamente. Entonces

$$C(M_n) \leq (2n - 1)C(M'_n).$$

Lo anterior se demuestra por inducción. Para $i = 1$, resulta evidente que $C(M_1) = C(M'_1)$. En consecuencia,

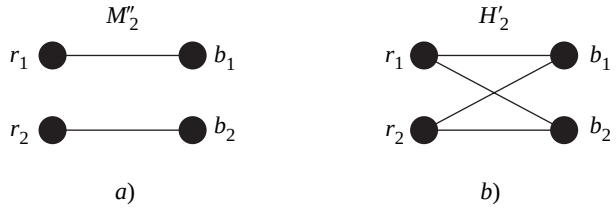
$$C(M_i) \leq (2i - 1)C(M'_i).$$

se cumple para $i = 1$.

Suponga que la fórmula anterior se cumple para $i = 1$ y que en M_i, b_i está apareado con r_j. Sean M''_i igual a $M'_{i-1} \cup (b_i, r_j)$ y H'_i igual a $M'_i \oplus M''_i$. Debido a que los apareamientos M'_i y M''_i son los apareamientos de b_i con el subconjunto R_i de R, cada vértice en H'_i tiene grado 0 o 2. Antes de proceder con la demostración se presentarán algunos ejemplos.

Para el caso que se ilustra en la figura 12-30, $M''_2 = M'_1 \cup (b_2, r_1)$ ahora se muestra en la figura 12-31*a*) y $H'_2 = M'_2 \oplus M''_2$ ahora se muestra en la figura 12-31*b*).

FIGURA 12-31 M''_2 y H'_i para el caso en la figura 12-30.



Debido a que cada vértice en H_i' tiene grado 0 o 2, para H hay dos casos posibles.

Caso 1: Todos los vértices en H_i' tienen grado 0.
En este caso, M_i' y M_i'' son exactamente lo mismo. Así, (b_i, r_j) debe estar en M_i'. Sea $d(b_i, r_j)$ la distancia entre b_i y r_j. Entonces

$$d(b_i, r_j) \leq C(M_i).$$

Debido a que $C(M_{i-1}')$ es no negativo, se tiene

$$d(b_i, r_j) \leq C(M_i') + C(M'_{i-1}).$$

Caso 2: En H hay un ciclo alterno. De nuevo, es posible demostrar que

$$d(b_i, r_j) \leq C(M_i') + C(M'_{i-1})$$

en este caso. Aquí se omite la demostración.

Debido a que $C(M_i) - C(M_{i-1}) = d(b_i, r_j)$, se tiene

$$\begin{aligned}
C(M_i) - C(M_{i-1}) &= d(b_i, r_j) \leq C(M_{i-1}') + C(M_i').\\
C(M_i) &\leq C(M_{i-1}) + C(M_{i-1}') + C(M_i')\\
&\leq (2(i-1)-1)C(M_{i-1}') + C(M_{i-1}') + C(M_i')\\
&\quad \text{(por hipótesis de inducción)}\\
&\leq (2(i-1)-1)C(M_i') + 2C(M_i') \quad (\text{para } C(M_{i-1}') \leq C(M_i'))\\
&\leq (2i-1)C(M_i').
\end{aligned}$$

Se demostró que nuestro algoritmo es $(2n - 1)$ competitivo. La pregunta es: nuestro algoritmo, ¿es óptimo? Al principio de esta sección se estableció el hecho de que ningún algoritmo de apareamiento bipartita mínimo es capaz de lograr una tasa de competitividad menor que $2n - 1$. Así que este algoritmo en línea basado en la estrategia de compensación es óptimo.

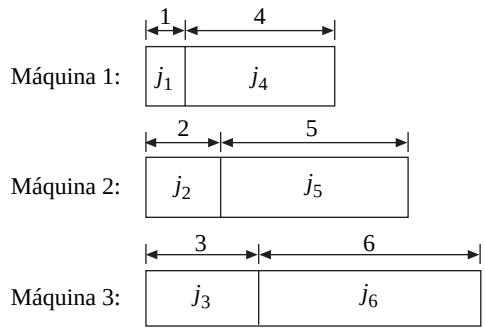
12-5 EL PROBLEMA EN LÍNEA DE LAS *M*-MÁQUINAS RESUELTO CON LA ESTRATEGIA DE MODERACIÓN

En esta sección se describirá un algoritmo basado en la estrategia de moderación para el problema en línea de programación de *m*-máquinas. Este problema se define como sigue: se cuenta con *m*-máquinas idénticas y los trabajos llegan uno por uno. El tiempo

de ejecución del *i*-ésimo trabajo se conoce cuando llega el *i*-ésimo trabajo. Tan pronto como llega un trabajo, debe asignarse de inmediato a una de las *m*-máquinas. El objetivo es programar los trabajos de manera no preferencial a las *m*-máquinas de modo que se minimice el makespan, el tiempo en que se termina (o completa) el último trabajo.

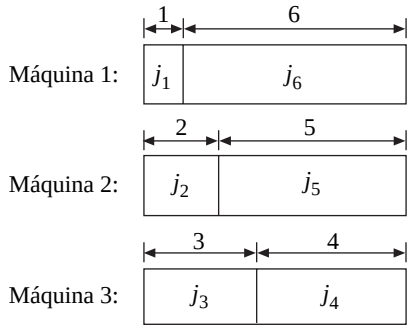
Una estrategia directa es el método codicioso, que puede ilustrarse con el siguiente ejemplo. Considere el caso en que se tienen seis trabajos, denotados por $j_1, j_2, \ldots, j_6$, y sus tiempos de ejecución son 1, 2,…, 6, respectivamente. El método codicioso asigna el trabajo recién llegado a la máquina que posee el menor tiempo total de procesamiento. La asignación final se muestra en la figura 12-32.

FIGURA 12-32 Ejemplo para el problema de programación (scheduling) de *m*-máquinas.



Como se muestra en la figura 12-32, el tiempo de terminación más largo, tan a menudo denotado como el makespan, es 9 para este caso. Este makespan puede acortarse en caso de que no se adopte el método codicioso. La programación mostrada en la figura 12-33 produce un makespan igual a 7.

FIGURA 12-33 Una mejor programación para el ejemplo de la figura 12-32.



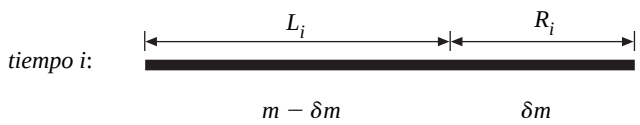
El algoritmo basado en el método codicioso se denomina algoritmo List. Se ha demostrado que este algoritmo es $\left(2-\frac{1}{m}\right)$ competitivo. A continuación, se describirá un algoritmo que usa la estrategia de moderación. El algoritmo es $\left(2-\frac{1}{70}\right)$ competitivo para $m \geq 70$. Así, este algoritmo posee un mejor desempeño que el algoritmo List cuando $m \geq 70$.

Básicamente, la estrategia de moderación no asigna un trabajo recién llegado a la máquina que tiene el menor tiempo de procesamiento. En vez de ello, intenta asignar el trabajo recién llegado a una máquina cuyo tiempo de procesamiento intermedio, que no sea demasiado breve ni demasiado largo. Debido a ello, esta estrategia se denomina estrategia de moderación.

A continuación se definirán algunos términos. Primero, sea a_i el tiempo de ejecución del i-ésimo trabajo recién llegado, donde $1 \leq i \leq n$. Suponga que $m \geq 70$. Sea $\varepsilon = \frac{1}{70}$. Sean $\delta \in [0.445 - 1/(2m), 0.445 + 1/(2m)]$ y δm números enteros. Por ejemplo, sea $m = 80$. Entonces puede hacerse $\delta = \frac{36}{80}$, que satisface la definición del intervalo de δ, y $\delta m = 36$ es un entero.

La ocupación de una máquina se define como la suma de las longitudes de los trabajos ya asignados a tal máquina. En cualquier tiempo, el algoritmo mantiene una secuencia de máquinas ordenadas de manera decreciente según sus ocupaciones actuales. En el tiempo i, es decir, cuando se han programado i trabajos, la secuencia se divide en dos subsecuencias: L_i y R_i. R_i es la subsecuencia de las δm primeras máquinas en la lista, y L_i es la subsecuencia de las $m - \delta m$ últimas máquinas, lo cual se muestra en la figura 12-34.

FIGURA 12-34 L_i y R_i.



Sean Rh_i y Lh_i, respectivamente, las secuencias de ocupaciones de L_i y R_i. Sean A_i y M_i las ocupaciones media y mínima sobre todas las m-máquinas, respectivamente, en el tiempo i. Sean $A(P)$ y $M(P)$ el promedio y el mínimo, respectivamente, de las ocupaciones en P, donde P es una secuencia de ocupaciones.

El algoritmo en línea para el problema en línea de las m-máquinas basado en la estrategia de moderación se describe a continuación:

Cuando llega el i-ésimo trabajo, colocar el $(i + 1)$-ésimo trabajo en la primera máquina en L_i, en caso de que

$$M(Lh_i) + a_{i+1} \leq (2 - \varepsilon)A(Rh_i);$$

en caso contrario, colocar el $(i + 1)$-ésimo trabajo en la primera máquina en la lista R_i, que es la que posee la menor ocupación global. En caso de ser necesario, permutar la lista de máquinas de modo que las ocupaciones permanezcan ordenadas de manera no decreciente.

Luego se demostrará que el algoritmo en línea anterior es $(2 - \varepsilon)$ competitivo. La demostración es más bien complicada, por lo que no se presentará completa. Sólo se muestran las partes fundamentales.

Sean ON_t y OPT_t los makespans de los algoritmos en línea y óptimo en el tiempo t, respectivamente. Suponga que el algoritmo no es $(2 - \varepsilon)$ competitivo. Así, existe una t tal que

$$ON_t \leq (2 - \varepsilon)OPT_t \quad \text{y} \quad ON_{t+1} > (2 - \varepsilon)OPT_{t+1}.$$

Se considerarán $M(Lh_t)$ y $A(Rh_t)$. Suponga que $M(Lh_t) + a_{t+1} \leq (2 - \varepsilon)A(Rh_t)$. Entonces el algoritmo en línea coloca a_{t+1} en la máquina más corta de L_i. Así,

$$\begin{aligned} ON_{t+1} &= M(Lh_t) + a_{t+1} \\ &\leq (2 - \varepsilon)A(Rh_t) \\ &\leq (2 - \varepsilon)A_t \\ &\leq (2 - \varepsilon)A_{t+1}. \end{aligned} \tag{12-3}$$

Debido a que OPT_i es un makespan óptimo para los trabajos $a_1, a_2, \ldots, a_i$, se tiene $m \times OPT_i \geq a_1 + a_2 + \cdots + a_i = m \times A_i$. Por lo tanto,

$$OPT_i \geq A_i. \tag{12-4}$$

Al sustituir (12-4) en (12-3), se obtiene

$$ON_{t+1} \leq (2 - \varepsilon)OPT_{t+1}. \tag{12-5}$$

Esto contradice las hipótesis. Así, $M(Lh_t) + a_{t+1} < (2 - \varepsilon)A(Rh_t)$ y el algoritmo en línea coloca a_{t+1} en la máquina que tiene la ocupación global más corta. Por lo tanto,

$$ON_{t+1} = M_t + a_{t+1}. \tag{12-6}$$

A continuación se demostrará que $M_t > (1 - \varepsilon)A_t$. Suponga que $M_t \leq (1 - \varepsilon)A_t$. Entonces

$$\begin{aligned} ON_{t+1} &= M_t + a_{t+1} \quad \text{(por la ecuación (12-6))} \\ &\leq (1 - \varepsilon)A_t + a_{t+1} \\ &\leq (1 - \varepsilon)A_{t+1} + a_{t+1}. \end{aligned}$$

Ya que $A_{t+1} \leq OPT_{t+1}$ y $a_{t+1} \leq OPT_{t+1}$

$$ON_{t+1} \leq (2 - \varepsilon)OPT_{t+1}.$$

Eso es imposible porque se ha supuesto que $ON_{t+1} > (2 - \varepsilon)OPT_{t+1}$. En consecuencia, puede concluirse que

$$M_t > (1 - \varepsilon)A_t. \tag{12-7}$$

La desigualdad (12-7) indica que la máquina más corta en el tiempo t posee una ocupación más grande que $(1 - \varepsilon)A_t$. De hecho, puede demostrarse una afirmación aún más poderosa: toda máquina en el tiempo t contiene un trabajo de longitud por lo menos igual a $\frac{1}{2(1-\varepsilon)} A_t \geq \frac{1}{2(1-\varepsilon)} M_t$. Posteriormente se presentará la demostración de esta afirmación. Mientras tanto, se supondrá que es verdadera. Entonces, rápidamente se establece el hecho de que el algoritmo es $(2 - \varepsilon)$ competitivo.

Para a_{t+1} hay dos casos posibles.

$a_{t+1} \leq (1 - \varepsilon)M_t$.

Entonces

$$\begin{aligned} ON_{t+1} &= M_t + a_{t+1} \quad \text{(por la ecuación (12-6))} \\ &\leq (2 - \varepsilon)M_t \\ &\leq (2 - \varepsilon)M_{t+1} \\ &\leq (2 - \varepsilon)OPT_{t+1}. \end{aligned}$$

Esto contradice $ON_{t+1} > (2 - \varepsilon)OPT_{t+1}$.

$a_{t+1} > (1 - \varepsilon)M_t$.

Entonces

$$a_{t+1} > (1 - \varepsilon)M_t$$
$$\geq \frac{1}{2(1-\varepsilon)} M_t. \quad (\text{por } (1 - \varepsilon) \geq \frac{1}{2(1-\varepsilon)})$$

Como toda máquina en el tiempo t contiene un trabajo de longitud igual a por lo menos $\frac{1}{2(1-\varepsilon)} M_t$ y $a_{t+1} \geq \frac{1}{2(1-\varepsilon)} M_t$, hay por lo menos $m + 1$ trabajos de longitud igual a por lo menos $\frac{1}{2(1-\varepsilon)} M_t$, dos de los cuales deben estar en la misma máquina. Así,

$$OPT_{t+1} \geq \text{máx}\left\{a_{t+1}, \frac{1}{2(1-\varepsilon)} M_t\right\}.$$

Así

$$ON_{t+1} = M_t + a_{t+1} \quad (\text{por la ecuación (12-6)})$$
$$\leq (1 - \varepsilon)OPT_{t+1} + OPT_{t+1}$$
$$(\text{por } a_{t+1} \leq OPT_t + 1 \text{ y } \frac{1}{2(1-\varepsilon)} M_t \leq OPT_{t+1})$$
$$= (2 - \varepsilon)OPT_{t+1}.$$

Esto también contradice $ON_{t+1} > (2 - \varepsilon)OPT_{t+1}$.

En cualquier caso, se llega a una contradicción. Por consiguiente, se concluye que el algoritmo en línea es $(2 - \varepsilon)$ competitivo.

A continuación se demostrará la contundente afirmación que establece que en el tiempo t, toda máquina contiene un trabajo de longitud igual a por lo menos $\frac{1}{2(1-\varepsilon)} A_t$. Para demostrarlo, observe que ya se ha demostrado que $M_t > (1 - \varepsilon)A_t$. Debido a esto, debe haber un tiempo r tal que $r < t$ y las máquinas en R_r tienen pesos de cuando mucho $(1 - \varepsilon)A_t$ en el tiempo r.

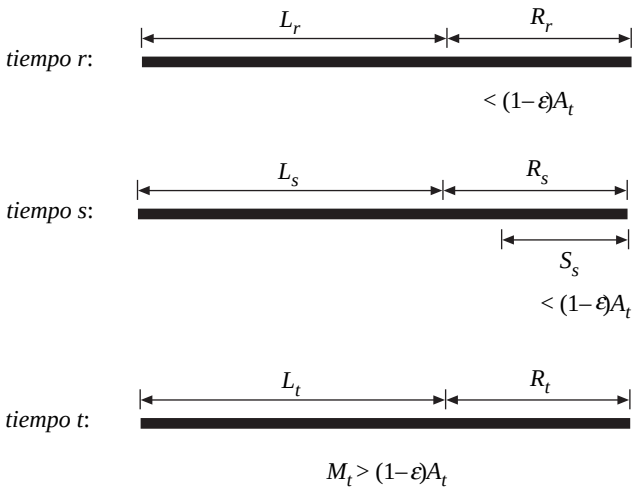
Sean $\tau \in [0.14 - 1/(2\delta m), 0.14 + 1/(2\delta m)]$ y m un entero. Sean S_i y Sh_i la secuencia de las m-máquinas de menor ocupación en el tiempo i y la secuencia de sus ocupaciones, respectivamente. Sea s el tiempo en que sólo las máquinas en S_s tienen ocupaciones de cuando mucho $(1 - \varepsilon)A_t$. Puede verse fácilmente que $r < s < t$.

Los tres tiempos críticos se resumen como sigue:

1. En el tiempo r, las máquinas en R_r tienen ocupaciones cuando mucho iguales a $(1 - \varepsilon)A_t$.
2. En el tiempo s, las máquinas en S_s tienen ocupaciones cuando mucho iguales a $(1 - \varepsilon)A_t$.
3. En el tiempo t, $M_t > (1 - \varepsilon)A_t$.

Estos tiempos se ilustran en la figura 12-35.

FIGURA 12-35 Ilustración de los tiempos r, s y t.



Además se observa que $S_s \in R_r$. Pueden hacerse las siguientes afirmaciones:

Afirmación 1. En el tiempo t, toda máquina en S_s contiene un trabajo cuyo tamaño es por lo menos igual a $\dfrac{1}{2(1-\varepsilon)} A_t$.

Afirmación 2. En el tiempo t, toda máquina en R_r, distinta a las de S_s, contiene un trabajo cuyo tamaño es por lo menos igual a $\dfrac{1}{2(1-\varepsilon)} A_t$.

Afirmación 3. En el tiempo t, toda máquina en L_r contiene un trabajo cuyo tamaño es por lo menos igual a $\dfrac{1}{2(1-\varepsilon)} A_t$.

Resulta evidente que una vez que se demuestran las tres afirmaciones anteriores, se ha demostrado que en el tiempo t cada máquina contiene un trabajo cuya longitud es por lo menos igual a $\frac{1}{2(1-\varepsilon)} A_t$. A continuación se demostrará la afirmación 1, ya que las otras dos pueden demostrarse de manera semejante.

Demostración de la afirmación 1. Debido a que toda máquina en S_s tiene una ocupación que cuando mucho es igual a $(1 - \varepsilon)A_t$ en el tiempo s, y $M_t > (1 - \varepsilon)A_t$ (desigualdad (12-7)), toda máquina en S_s debe recibir por lo menos un trabajo durante el periodo que va del tiempo s al tiempo t. Sea p una máquina en S_s. Sea el trabajo j_{i+1} el primer trabajo que recibe la máquina p después del tiempo s, donde $s < i \leq t - 1$. Según nuestro algoritmo, este trabajo siempre consiste en encender la primera máquina en L_{i+1} o la máquina más baja globalmente. Debido a que $p \in S_s$, no puede pertenecer a L_{i+1}. En consecuencia, p debe ser la máquina más baja globalmente en el tiempo $i + 1$. Así, se tiene

$$< M(Lh_i) + a_{i+1} > (2 - \varepsilon)A(Rh_i)$$

de modo que

$$a_{i+1} > (2 - \varepsilon)A(Rh_i) - M(Lh_i). \tag{12-8}$$

También se tiene

$$\begin{aligned} A_t &\geq A_i \\ &= (1 - \delta)A(Lh_i) + \delta A(Rh_i) \\ &\geq (1 - \delta)M(Lh_i) + \delta A(Rh_i). \end{aligned}$$

Esto implica que

$$M(Lh_i) \leq \frac{A_t - \delta A(Rh_i)}{1-\delta}. \tag{12-9}$$

Además,

$$\begin{aligned} A(Rh_i) &\geq A(Rh_s) \\ &= (1 - \tau)(\text{el promedio de ocupaciones en } R_s - S_s) + \tau A(Sh_s) \\ &> (1 - \tau)[(1 - \varepsilon)A_t] + \tau A(Sh_s). \end{aligned} \tag{12-10}$$

(debido a que la máquina en $R_s - S_s$ tiene una ocupación cuando menos igual a $(1 - \varepsilon)A_t$)

Luego se obtiene

$$
\begin{aligned}
a_{i+1} &> (2-\varepsilon)A(Rh_i) - M(Lh_i) \quad \text{(por la desigualdad (12-8))} \\
&> (2-\varepsilon)A(Rh_i) - \left[\frac{A_t - \delta A(Rh_i)}{1-\delta}\right] \quad \text{(por la desigualdad (12-9))} \\
&= \left(2-\varepsilon+\frac{\delta}{1-\delta}\right)A(Rh_i) - \frac{\delta}{1-\delta}A_t \\
&= \left(2-\varepsilon+\frac{\delta}{1-\delta}\right)[(1-\tau)[(1-\varepsilon)A_t] + \tau A(Sh_s)] - \frac{\delta}{1-\delta}A_t \\
&\quad \text{(por la desigualdad (12-10))} \\
&= \left\{\left[2-\varepsilon+\frac{\delta}{1-\delta}\right](1-\tau)(1-\varepsilon) - \frac{\delta}{1-\delta}\right\}A_t \\
&\quad + \left\{\left[2-\varepsilon+\frac{\delta}{1-\delta}\right]\tau\right\}A(Sh_s) \\
&> \left\{\left[2-\varepsilon+\frac{\delta}{1-\delta}\right](1-\tau)(1-\varepsilon) - \frac{\delta}{1-\delta}\right\}A_t.
\end{aligned}
$$

Debido a que $\dfrac{1}{2(1-\varepsilon)} \approx 0.51$ y

$$
\begin{aligned}
&\left(2-\varepsilon+\frac{\delta}{1-\delta}\right)(1-\tau)(1-\varepsilon) - \frac{\delta}{1-\delta} \\
&\quad \approx \left(2-\frac{1}{70}+\frac{0.455}{1-0.455}\right)(1-0.14)\left(1-\frac{1}{70}\right) - \frac{0.455}{1-0.455} \\
&\quad = 0.56,
\end{aligned}
$$

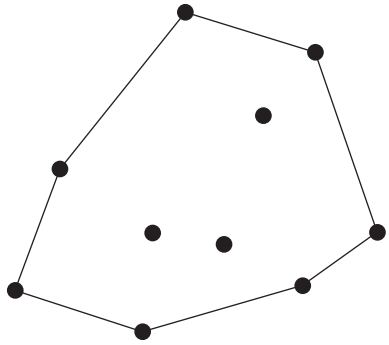
se tiene para $s < i \le t-1$,

$$
a_{i+1} > \frac{1}{2(1-\varepsilon)}A_t.
$$

12-6 ALGORITMOS EN LÍNEA BASADOS EN LA ESTRATEGIA DE ELIMINACIÓN PARA TRES PROBLEMAS DE GEOMETRÍA COMPUTACIONAL

En esta sección se presentarán algoritmos en línea para tres problemas de geometría computacional. Tales problemas son el del caso convexo, el del par más alejado y el problema con 1 centro. El problema del par más alejado consiste en lo siguiente: encontrar un par de puntos para un conjunto S de puntos en un plano de modo que la distancia entre ambos sea la más grande posible. Todos nuestros algoritmos en línea proporcionan soluciones aproximadas. Se demostrará que todas estas soluciones aproximadas están próximas a las soluciones exactas. Para estos tres problemas se usará el método de contención de rectas paralelas. Considere la figura 12-36, que contiene un convex hull.

FIGURA 12-36 Un convex hull.



Luego, suponga que se agrega un nuevo punto. Si este punto está en el interior del convex hull, no es necesario hacer nada. Esto significa que quizá sólo sea necesario recordar la frontera de este convex hull. Por supuesto, esta frontera del convex hull cambia si un nuevo punto recién llegado se encuentra fuera de la frontera.

Para formar la frontera de los puntos de entrada se usa un conjunto de rectas paralelas. Por ejemplo, en la figura 12-37, hay cuatro pares de rectas paralelas; a saber, (l_1, l_1'), (l_2, l_2'), (l_3, l_3') y (l_4, l_4'), que forman una frontera de puntos de entrada.

Siempre que llega un punto se comprueba si este punto se encuentra fuera de la frontera. En caso de estarlo, se emprende una acción; en caso contrario, sean l_i y l_i' un par de rectas paralelas que no contienen a p y sin pérdida de generalidad, sea l_i más cercano de p que l_i'. Luego se mueve l_i, sin modificar su pendiente, hasta tocar al punto p. Considere la figura 12-38. Debido a que p no está contenido en l_3 ni en l_3', l_3 se mueve hasta tocar a p.

FIGURA 12-37 Frontera formada por cuatro pares de rectas paralelas.

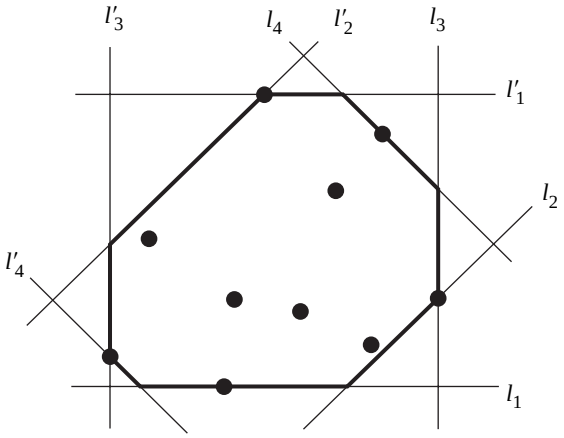
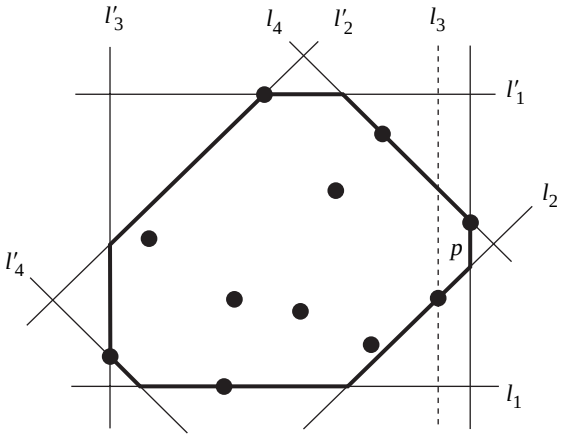


FIGURA 12-38 Movimientos de las rectas después de recibir un nuevo punto de entrada.



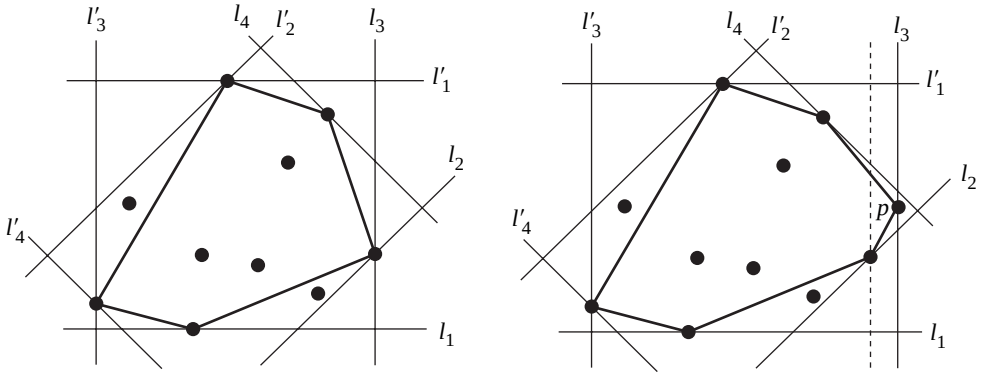
En general, considere que la frontera está formada por m pares de rectas paralelas. Las pendientes de estos m pares de rectas paralelas son 0, $\tan(\pi/m)$, $\tan(2\pi/m)$, …, $\tan((m-1)\pi/m)$, respectivamente. Estos pares de rectas paralelas deben satisfacer las siguientes condiciones:

Regla 1: Todo punto de entrada está dentro de todo par de rectas paralelas.

Regla 2: Todo par de rectas paralelas debe estar lo más próximo posible.

A continuación se presenta un algoritmo en línea para construir un convex hull aproximado. Debido a que cada recta está asociada a por lo menos un punto de entrada, estos puntos pueden unirse siguiendo el sentido del movimiento de las manecillas del reloj con el fin de formar un convex hull que funciona como nuestro convex hull aproximado. Considere la figura 12-39. El convex hull aproximado original se muestra en la figura 12-39*a*). Luego que se agrega un nuevo punto *p*, el nuevo convex hull aproximado se muestra en la figura 12-39*b*). Observe que un punto puede estar fuera del polígono convexo. Ésta es la tasa por la cual nuestro algoritmo es un algoritmo aproximado.

FIGURA 12-39 Ejemplo de un convex hull aproximado.



A continuación se presenta un algoritmo de aproximación para un convex hull aproximado, el algoritmo *A*.

Algoritmo 12-1 □ Algoritmo *A* para calcular convex hull en línea

Input: Una secuencia de puntos $p_1, p_2, \ldots$, y el número m de pares de rectas paralelas que se utilizan.

Output: Una secuencia de convex hull aproximados $a_1, a_2, \ldots$, donde a_i es un convex hull aproximado en línea que cubre los puntos $p_1, p_2, \ldots, p_i$.

Initialization: Construir m de pares de rectas paralelas con pendientes $0, \tan(\pi/m), \tan(2\pi/m), \ldots, \tan((m-1)\pi/m)$, respectivamente, y localizar todas las rectas de modo que todas se corten en el primer punto de entrada p_1. Sea $i = 1$; es decir, el punto de entrada actual es p_1.

Paso 1: Para cada uno de los m pares de rectas paralelas, si el punto p_i está entre ellos, nada cambia; en caso contrario, mover la recta más cercana hacia el punto p_i, sin cambiar su pendiente, hasta tocar p_i y asociar p_i con esta recta.

Paso 2: Construir un convex hull aproximado uniendo los $2m$ puntos con respecto a cada recta en el sentido del movimiento de las manecillas del reloj y denotar este convex hull aproximado por a_i.

Paso 3: Si ningún otro punto será la entrada, entonces detener el proceso; en caso contrario, hacer $i = i + 1$ y recibir el siguiente punto de entrada como p_i. Ir al Paso 1.

Resulta evidente que la complejidad temporal del algoritmo A es $O(mn)$, donde m es el número de pares de rectas paralelas utilizadas y n es el número de puntos de entrada. Debido a que m es fijo, el algoritmo es $O(n)$. A continuación se analizará la tasa de error de nuestros convex hull aproximados. Hay tres polígonos que resultan de interés para nuestro algoritmo:

1. Polígono E: el polígono convexo de contención formado por las $2m$ rectas. Todos los puntos de entrada están en el interior del polígono.
2. Polígono C: el convex hull de los puntos de entrada.
3. Polígono A: el polígono aproximadamente producido por el algoritmo A.

Sea $L(P)$ la longitud total del lado del polígono P. Entonces resulta evidente que

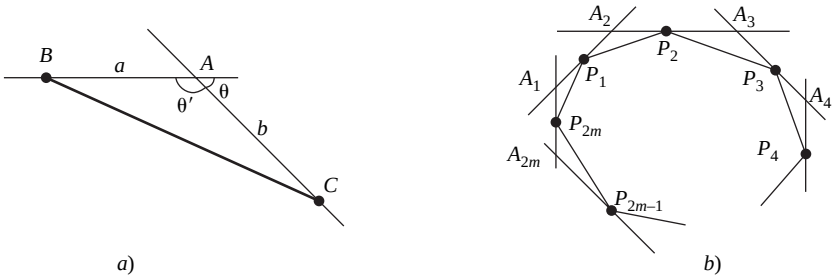
$$L(E) \geq L(C) \geq L(A).$$

La tasa de error $Err(A)$ para nuestro convex hull aproximado A en línea es

$$Err(A) = \frac{L(C) - L(A)}{L(C)}.$$

Considere la figura 12-40*a*).

FIGURA 12-40 Estimación de $Err(A)$.



a) *b*)

Se tiene lo siguiente:

$$\left(\frac{\overline{AB}+\overline{AC}}{\overline{BC}}\right)^2 = \frac{(a+b)^2}{a^2+b^2-2ab\cdot\cos\theta'}$$
$$= \frac{a^2+b^2+2ab}{a^2+b^2+2ab\cdot\cos\theta}$$
$$= \frac{(a^2+b^2)/2ab+1}{(a^2+b^2)/2ab+\cos\theta}$$
$$\leq \frac{1+1}{1+\cos\theta}$$
$$= \sec^2\frac{\theta}{2}.$$

Por lo tanto,

$$\overline{AB}+\overline{AC} \leq \sec\frac{\theta}{2}\cdot\overline{BC}.$$

Considere la figura 12-40*b*).

$$L(E) = \overline{P_{2m}A_1} + \overline{A_1P_1} + \overline{P_1A_2} + \cdots + \overline{A_{2m}P_{2m}}$$
$$\leq \sec\frac{\pi}{2m}\cdot(\overline{P_{2m}P_1} + \overline{P_1P_2} + \overline{P_2P_3} + \cdots + \overline{P_{2m-1}P_{2m}})$$
$$= \sec\frac{\pi}{2m}\cdot L(A).$$

Se tiene

$$Err(A) = \frac{L(C)-L(A)}{L(C)} \leq \frac{L(E)-L(A)}{L(A)} \leq \sec\frac{\pi}{2m} - 1.$$

La ecuación anterior indica que cuantas más rectas se usen, más baja es la tasa de error, como era de esperar. En la tabla 12-1 se muestra cómo disminuye *Err*(*A*) cuando *m* crece.

TABLA 12-1 La cota superior de la tasa de error.

m	sg $\frac{\pi}{2m}$	Cota superior de $Err(A)$
2	1.4142	0.4142
3	1.1547	0.1547
4	1.0824	0.0824
5	1.0515	0.0515
10	1.0125	0.0125
20	1.0031	0.0031

El algoritmo de aproximación en línea para el par más alejado casi siempre es el mismo que el algoritmo de aproximación en línea del convex hull que se presentó en el apartado anterior. De todos los pares de rectas paralelas, siempre se encuentra el par tal que la distancia entre las rectas es la mayor posible. Este algoritmo usa los dos puntos asociados con este par de rectas paralelas como la salida en línea del par más alejado.

Así, nuestro algoritmo en línea para el problema del par más alejado es casi exactamente el mismo que el algoritmo A, excepto en el Paso 2. Ahora se usa lo siguiente:

Paso 2: Encontrar el par de rectas paralelas cuya distancia sea la más grande. Hacer que estos dos puntos asociados con el par A de rectas paralelas en cuestión sean el par más alejado aproximado en línea A_i. Ir al Paso 3.

Considere la figura 12-41. Sean p_1 y p_2 el par exacto más alejado. Sea d_i la distancia entre el i-ésimo par de rectas paralelas. Sea $d_{\text{máx}}$ la d_i más grande para toda i. Considere la figura 12-42. Sea N_i la dirección perpendicular al i-ésimo par de rectas paralelas. Sea θ_i el ángulo pronunciado entre el par más alejado $\overline{P_1P_2}$ y N_i. Observe que $0 \leq \theta_i \leq \frac{\pi}{2}$. Sean o_1 y o_2 los dos puntos asociados con el par de rectas paralelas cuya distancia es $d_{\text{máx}}$.

FIGURA 12-41 Ejemplo de los pares más alejados exactos y aproximados.

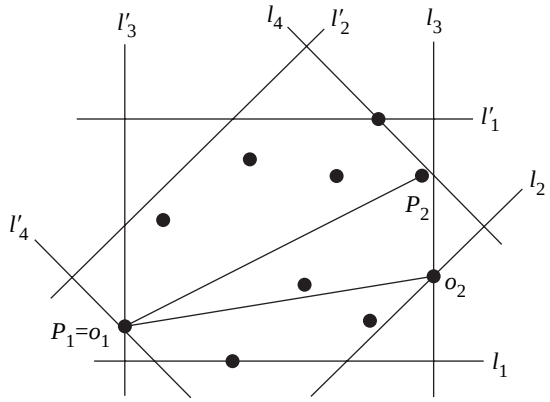
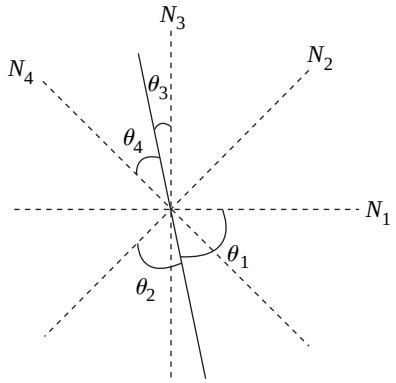


FIGURA 12-42 Ejemplo del ángulo entre el par más alejado aproximado y N_i.



Se tiene lo siguiente:

$$\begin{aligned}
\overline{P_1P_2} &\le d_i \cdot \sec\theta_i, \ \forall i \in \{1, \ldots, m\} \\
&\le \min_i \{d_i \sec\theta_i\} \\
&\le \min_i \{d_{\text{máx}} \cdot \sec\theta_i\} \\
&= d_{\text{máx}} \min_i \{\sec\theta_i\} \\
&= d_{\text{máx}} \cdot \sec(\min_i \{\theta_i\}) \\
&= d_{\text{máx}} \cdot \sec\frac{\pi}{2m} \\
&= \overline{o_1o_2} \cdot \sec\frac{\pi}{2m}.
\end{aligned}$$

Por lo tanto, se tiene lo siguiente:

$$\begin{aligned}\text{Tasa de error} &= \frac{\overline{p_1p_2} - \overline{o_1o_2}}{\overline{p_1p_2}} \\ &\leq \frac{\overline{p_1p_2} - \overline{o_1o_2}}{\overline{o_1o_2}} \\ &\leq \sec\frac{\pi}{2m} - 1.\end{aligned}$$

La cota superior de la tasa de error disminuye cuando m crece, lo cual se muestra en la tabla 12-1. El problema con 1 centro, que se presentó en el capítulo 6, se define como sigue: dado un conjunto de n puntos en el espacio euclidiano, encontrar un círculo que cubra todos estos n puntos de modo que se minimice la longitud del diámetro de este círculo. En el algoritmo A, siempre hay m pares de rectas paralelas con pendientes fijadas de manera particular. Los m pares de rectas paralelas forman un polígono contenedor convexo E. Este polígono posee las siguientes propiedades:

Propiedad 1: E cubre todos los puntos de entrada.
Propiedad 2: El número de vértices de E es cuando mucho $2m$.
Propiedad 3: Las pendientes de las aristas en E son iguales uno a uno a los de un polígono regular de $2m$ lados.
Propiedad 4: Toda arista de E contiene por lo menos un punto.

Las propiedades anteriores de E pueden usarse para construir un círculo mínimo aproximado de entrada en línea para los puntos de entrada. Nuestro algoritmo en línea para el problema con 1 centro también es casi exactamente el mismo que el algoritmo A, excepto por el Paso 2. A continuación se usa lo siguiente:

Paso 2: Encontrar los vértices del polígono convexo E; es decir, los puntos que son intersecciones de dos rectas consecutivas. Utilizar cualquier algoritmo fuera de línea con 1 centro para resolver el problema con 1 centro para el conjunto de puntos de cuando mucho $2m$ vértices de E. Deje que este círculo sea el círculo aproximado en línea A_i.

La complejidad temporal para encontrar los vértices de E requiere tiempo $O(m)$ (Propiedad 2). La complejidad temporal para encontrar un círculo que cubra $2m$ puntos es nuevamente una constante porque sólo está implicado un número constante de puntos.

Por definición, el círculo que se encuentra con nuestro algoritmo cubre el polígono contenedor convexo E y, en consecuencia por la Propiedad 1, también cubre a todos los

puntos de entrada. El problema restante es calcular cuán aceptable es el círculo mínimo aproximado en línea. Se sigue el método denominado análisis competitivo. Considere que la longitud del diámetro de un círculo C se denota por $d(C)$. Dada una secuencia de puntos, sean C_{opt} y C_{onl} el círculo mínimo y el círculo aproximado en línea que cubren todos los puntos de esta secuencia, respectivamente. Un algoritmo en línea para el problema con 1 centro es c_k-competitivo si para todas las secuencias de puntos en el plano, $d(C_{onl}) \leq c_k \cdot d(C_{opt})$. Más tarde se demostrará que c_k es $\sec \dfrac{\pi}{2m}$.

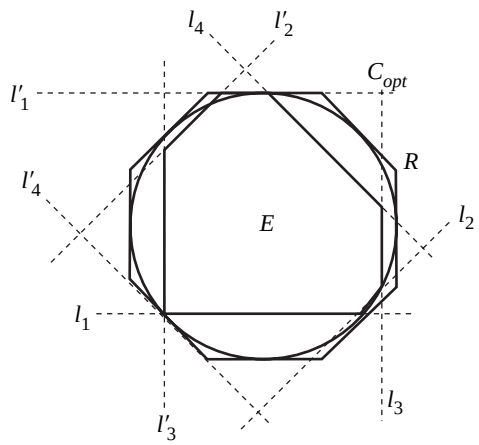
Para C_{opt}, puede construirse un polígono regular R con $2m$ lados y las mismas pendientes, ya que los $2m$ pares de rectas paralelas contienen a todos los puntos. Sea E el polígono formado por estas $2m$ rectas. Como se muestra en la figura 12-43, $E \subset R$. Sea C_R el círculo más pequeño que cubre a R. Debido a que $E \subset R$, el menor círculo que cubre a E, que es C_{onl}, debe ser menor que o igual a C_R. Debido a que C_{opt} es el círculo óptimo que cubre a todos los puntos, se tiene $d(C_{opt}) \leq d(C_{onl}) \leq d(C_R)$. Considere la figura 12-43. $\overline{OB}$ es el radio de C_{opt}. $\overline{OA}$ es el radio de C_R. Se tiene

$$d(C_{opt}) \cdot \sec \frac{\pi}{2m} = d(C_R).$$

Por lo tanto,

$$\frac{d(C_{onl})}{d(C_{opt})} \leq \frac{d(C_R)}{d(C_{opt})} = \frac{\overline{OA}}{\overline{OB}} = \sec \frac{\pi}{2m}.$$

FIGURA 12-43 Relación entre E y R.



Es decir, el algoritmo en línea presentado en esta sección es sec($\pi/(2m)$) competitivo.

12-7 UN ALGORITMO EN LÍNEA PARA EL ÁRBOL DE EXPANSIÓN BASADO EN LA ESTRATEGIA ALEATORIA

El desempeño de un algoritmo en línea con complejidades espacial y temporal bajas en cada paso de decisión es de crucial importancia para los problemas en línea, en general, ya que en aplicaciones reales (por ejemplo, sistemas de tiempo real), es necesario tomar una decisión tan pronto como sea posible una vez que llegan los datos. En consecuencia, la aleatorización del paso de decisión se convierte en una estrategia de diseño para que los algoritmos en línea satisfagan los requerimientos de baja complejidad de cálculo y temporal. En esta sección se presentará un algoritmo aleatorio simple para el problema en línea del árbol de expansión que se definió en el apartado 12-1.

Nuestro algoritmo en línea aleatorio, el algoritmo $R(m)$, para calcular un árbol euclidiano, se presenta a continuación.

Algoritmo 12-2 □ Algoritmo $R(m)$ para calcular árboles de expansión euclidianos en línea

Input: $n(\geq 3)$ puntos $v_1, \ldots, v_n$ en el espacio euclidiano
y un entero positivo $m(\leq n - 1)$

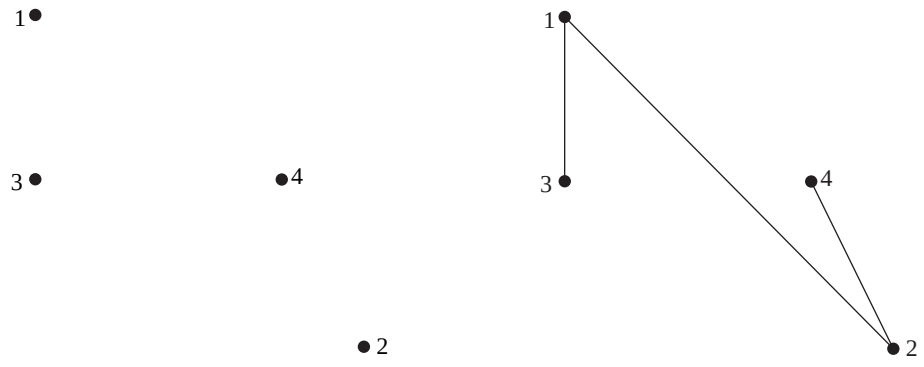
Output: Un árbol euclidiano T

```
Begin
    T = ϕ;
    input(m);
    input(v1);
    input(v2);
    Sumar a T la arista entre v1 y v2;
    For k = 3 to n do
        input(vk);
        If k ≤ m + 1 then
        Sumar a T la arista mínima entre vk y v1, …, vk−1;
        else
        Seleccionar al azar m aristas de las k − 1 aristas que hay entre vk
        y v1, …, vk−1, y sumar a T la arista mínima de las m aristas;
    Endfor
    Output(T);
End
```

El algoritmo $R(m)$ acepta los puntos uno por uno según su orden en la secuencia de entrada. Cuando llega un punto, el algoritmo toma la decisión de construir un árbol de expansión que expande los puntos proporcionados hasta el momento, y ninguna decisión se modifica después de que se ha tomado. Suponga que el punto actual que se está analizando es el i-ésimo punto de entrada (es decir, v_i). Si $2 \leq i \leq m + 1$, el algoritmo $R(m)$ suma al árbol previo la arista mínima entre el punto actual que se está analizando y los puntos analizados previos (es decir, $v_1, \ldots, v_{i-1}$) para formar un nuevo árbol de expansión; en caso contrario, suma al árbol previo la arista mínima de m aristas, que se escogen al azar de las aristas que están entre el punto actual que se está analizando y los puntos analizados previos para formar un nuevo árbol de expansión. En cada paso del análisis, se requiere tiempo $O(m)$. No es difícil percatarse que el algoritmo $R(n - 1)$ es el algoritmo codicioso determinístico antes presentado.

Suponga que se tienen cuatro puntos en el plano, como se muestra en la figura 12-44*a*), y que la secuencia de entrada de los puntos es 1, 2, 3, 4, se construirá un árbol de expansión, que se muestra en la figura 12-44*b*). En la tabla 12-2 se muestran las acciones durante la ejecución del algoritmo $R(2)$ con la secuencia de entrada 1, 2, 3, 4, donde $e(a, b)$ denota la arista entre el punto a y el punto b, y $|e(a, b)|$ denota la distancia euclidiana de $e(a, b)$.

FIGURA 12-44 Un ejemplo para el algoritmo $R(2)$.



a) Puntos de Input

b) Secuencia de Input 1, 2, 3, 4

TABLA 12-2 Acciones durante la ejecución del algoritmo $R(2)$ con la secuencia de entrada 1, 2, 3, 4.

Punto de input actual	Acción
1	Ninguna acción
2	Sumar a T $e(1, 2)$.
3	Sumar a T $e(3, 1)$.
	$(\|e(3, 1)\| = \text{mín}\{\|e(3, 1)\|, \|e(3, 2)\|\})$
4	Escoger $e(4, 1)$ y $e(4, 2)$, y sumar $e(4, 2)$ a T.
	$(\|e(4, 2)\| = \text{mín}\{\|e(4, 1)\|, \|e(4, 2)\|\})$

Un algoritmo aleatorio A en línea se denomina c-competitivo si existe una constante c tal que

$$E(C_A(\sigma)) \leq c \cdot C_{opt}(\sigma) + b$$

donde $E(C_A(\sigma))$ denota el costo esperado incurrido por el algoritmo A con entrada σ.

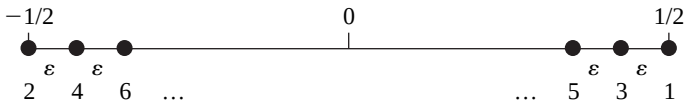
A continuación, se demostrará que si m es una constante fija, entonces la tasa de competitividad del algoritmo $R(m)$ para el problema del árbol de expansión en línea es $\Theta(n)$.

Suponga que se tiene un conjunto σ de n puntos $1, 2, 3, \ldots, n$, y que la secuencia de entrada es $1, 2, 3, \ldots, n$. Sean $D(a, b)$ la distancia entre el punto a y el punto b, y $N(a, k)$ el k-ésimo punto a más cercano entre todos los puntos proporcionados hasta el momento. La longitud esperada del árbol producido por el algoritmo $R(m)$ con entrada σ se denota por $E(L_{R(m)}(\sigma))$. Suponga que el algoritmo $R(m)$ ahora está leyendo el punto i. Si $2 \leq i \leq m + 1$, entonces el algoritmo une el punto i y su vecino más cercano de entre todos los $i - 1$ puntos proporcionados antes. No obstante, si $m + 2 \leq i \leq n$, el algoritmo une el punto i y su j-ésimo vecino más cercano de entre todos los $i - 1$ puntos proporcionados antes con probabilidad $\binom{i-1-j}{m-1} \Big/ \binom{i-1}{m}$, donde $1 \leq j \leq i - m$. Así se tiene la fórmula siguiente para $E(L_{R(m)}(\sigma))$:

$$E(L_{R(m)}(\sigma)) = \sum_{i=2}^{m+1} D(i, N(i,1)) + \sum_{i=m+2}^{n} \sum_{j=1}^{i-m} \binom{i-1-j}{m-1} \Big/ \binom{i-1}{m} \cdot D(i, N(i, j)). \quad \textbf{(12-11)}$$

Suponga que todos los n puntos están sobre una recta y que el punto i está en $\frac{(-1)^{i+1}}{2} + (-1)^i(\lceil i/2 - 1 \rceil)\varepsilon$, donde $\varepsilon > 0$ (consulte la figura 12-45).

FIGURA 12-45 Los *n* puntos sobre una recta.



Si el valor de ε es significativamente pequeño, se tiene

$$D(i, N(i, j)) = \begin{cases} 0 & \text{si } 1 \le j \le i - \lceil i/2 \rceil - 1 \\ 1 & \text{si } i \le \lceil i/2 \rceil \le j - i - 1 \end{cases}.$$

Sea σ^* los puntos de entrada anteriores. Si la secuencia de entrada es 1, 2,…, *n*, entonces por la ecuación (12-11) se obtiene

$$\begin{aligned}
&E(L_{R(m)}(\sigma^*)) \\
&= 1 + \sum_{i=m+2}^{n} \sum_{j=1-\lfloor i/2 \rfloor}^{i-m} \binom{i-1-j}{m-1} \Big/ \binom{i-1}{m} \\
&= 1 + \sum_{i=m+2}^{n} 1 \Big/ \binom{i-1}{m} \sum_{k=0}^{\lfloor i/2 \rfloor - m} \binom{m-1+k}{k} \\
&= 1 + \sum_{i=m+2}^{n} \binom{\lfloor i/2 \rfloor}{m} \Big/ \binom{i-1}{m} \quad \left(\text{por } \sum_{k=0}^{n} \binom{m+k}{k} = \binom{m+n+1}{n}\right) \\
&\ge 1 + \sum_{i=2m+1}^{n} \binom{(i-1)/2}{m} \Big/ \binom{i-1}{m} \\
&= 1 + \frac{1}{2^m} \sum_{i=2m+1}^{n} \frac{\prod_{k=\lceil m/2 \rceil + 1}^{m} (i - 2k + 1)}{\prod_{k=1}^{\lfloor m/2 \rfloor} (i - 2k)} \\
&= 1 + \frac{1}{2^m} \sum_{i=2m+1}^{n} \prod_{k=1}^{\lfloor m/2 \rfloor} \left(1 - \frac{2\lceil m/2 \rceil - 1}{i - 2k}\right).
\end{aligned}$$

Debido a que *m* es una constante, se tiene

$$\sum_{i=2m+1}^{n} \prod_{k=1}^{\lfloor m/2 \rfloor} \left(1 - \frac{2\lceil m/2 \rceil - 1}{i - 2k}\right)$$

$$> \sum_{i=2m+1}^{n} 1 - \sum_{k=1}^{\lfloor m/2 \rfloor} \frac{2\lceil m/2 \rceil - 1}{i - 2k}$$

$$= (n - 2m) - (2\lceil m/2 \rceil - 1)\left[\frac{1}{2m-1} + \frac{1}{2m-3} + \cdots + \frac{1}{2m+1-2\lceil m/2 \rceil}\right.$$

$$+ \frac{1}{2m} + \frac{1}{2m-2} + \cdots + \frac{1}{2m+2-2\lceil m/2 \rceil}$$

$$+ \frac{1}{2m-1} + \frac{1}{2m-1} + \cdots + \frac{1}{2m+3-2\lceil m/2 \rceil} + \cdots$$

$$\left. + \frac{1}{n-2} + \frac{1}{n-4} + \cdots + \frac{1}{n-2\lceil m/2 \rceil}\right]$$

$$= (n - 2m) - (2\lceil m/2 \rceil - 1)\left[\sum_{k=1}^{\lceil m/2 \rceil} (H_{n-2k} - H_{2m-2k})\right] \quad (\text{donde } H_n = \sum_{i=1}^{n} \frac{1}{i})$$

$$= n - 2m - o(n).$$

Así,

$$E(L_{R(m)}(\sigma^*))$$

$$\geq 1 + \frac{1}{2^m} \sum_{i=2m+1}^{n} \prod_{k=1}^{\lfloor m/2 \rfloor} \left(1 - \frac{2\lceil m/2 \rceil - 1}{i - 2k}\right)$$

$$= 1 + \frac{n - 2m - o(n)}{2^m}$$

Sea $L_{span}(\sigma^*)$ la longitud del árbol de expansión mínima con entrada σ^*. Resulta fácil ver que $L_{span}(\sigma^*) = 1$. En consecuencia, se obtiene

$$\frac{E(L_{R(m)}(\sigma^*))}{L_{span}(\sigma^*)} \geq 1 + \frac{n - 2m - o(n)}{2^m}.$$

Esto significa que la tasa de competitividad del algoritmo $R(m)$ para el problema en línea del árbol de expansión es $\Omega(n)$, donde n es el número de puntos y m es una constante fija.

Sea D el diámetro de n puntos. Las longitudes del árbol mínimo de Steiner y los árboles de expansión mínima son iguales o mayores que D. Debido a que la longitud de cada arista es menor o igual a D, la longitud del árbol producido por cualquier algoritmo en línea es menor o igual a $(n - 1) \cdot D$. En consecuencia, para el problema en línea del árbol de expansión, no existe ningún algoritmo en línea cuya tasa de competitividad sea mayor que $n - 1$, donde n es el número de puntos.

Con base en los resultados anteriores, puede concluirse que la tasa de competitividad del algoritmo $R(m)$ para el problema en línea del árbol de expansión es $\Theta(n)$, donde n es el número de puntos y m es una constante fija.

12-8 NOTAS Y REFERENCIAS

El problema en línea del árbol de Steiner, que es semejante al problema en línea del árbol de expansión, consiste en encontrar un árbol de Steiner en línea, donde los puntos se revelan uno por uno. En Imase y Waxman (1991) y Alon y Azar (1993), se demostró que la tasa de competitividad del algoritmo codicioso en la sección 12-1 es $\log_2 n$ para estos dos problemas. Además, Alon y Azar (1993) demostraron que las cotas inferiores de la tasa de competitividad de cualquier algoritmo en línea para los problemas del árbol de expansión y del árbol de Steiner son $\Omega(\log n/\log \log n)$.

Un problema de servidores es simétrico si la distancia del vértice i al vértice j es la misma que la distancia de j a i, y es asimétrico en caso contrario. Un algoritmo en línea para resolver el problema de k servidores opera bajo la restricción adicional de que es necesario decidir qué servidor mover para satisfacer una petición dada sin conocer cuáles son las futuras peticiones.

El problema de memoria caché, el problema de paginación, el problema del disco con dos cabezas, la búsqueda lineal, etc., pueden abstraerse como el problema de los servidores después de escoger las distancias entre los vértices y el número de servidores.

Manasse, McGeoch y Sleator (1990) introdujeron el problema de los servidores en 1990. Demostraron que k es una cota inferior de la tasa de competitividad para el problema de k servidores. El algoritmo en línea óptimo para el problema de k servidores sobre árboles planos fue propuesto en la obra de Chrobak y Larmore (1991).

El problema del recorrido de obstáculos fue propuesto por primera vez por Papadimitriou y Yannakakis (1991b). Mostraron una cota inferior de $\Omega(\sqrt{d})$ para cualquier algoritmo determinístico sobre la tasa de competitividad cuando todos los obstáculos son de lados paralelos a los ejes, donde d es la distancia entre s y t. También propusieron un algoritmo óptimo asintótico 3/2 competitivo para el que los obstáculos son cuadrados del mismo tamaño y lados paralelos a los ejes.

La desigualdad $\frac{\tau_1}{\pi_1} < \frac{2}{3}$ o $\frac{\tau_2}{\pi_2} < \frac{2}{3}$, usada en la sección 12-3, fue demostrada en Fejes (1978).

El problema de apareamiento bipartita fue propuesto en Kalyanasundaram y Pruhs (1993). El algoritmo y la cota inferior para este problema, presentado en la sección 12-4, son también de Kalyanasundaram y Pruhs (1993).

En 1996, Graham propuso un algoritmo codicioso simple, denominado List, para el problema de programación de *m*-máquinas. Graham demostró que el algoritmo List es $\left(2 - \frac{1}{m}\right)$ competitivo. El algoritmo $\left(2 - \frac{1}{70}\right)$ competitivo presentado en la sección 12-5 fue propuesto por Bartal, Fiat, Karloff y Vahra (1995) para el problema de programación de *m*-máquinas, donde $m \geq 70$. Los algoritmos en línea para los problemas de geometría del apartado 12-6 pueden consultarse en Chao (1992). El algoritmo aleatorio del apartado 12-7, para el problema en línea del árbol de expansión se analiza en Tsai y Tang (1993).

12-9 BIBLIOGRAFÍA ADICIONAL

Manasse, McGeoch y Sleator (1990) demostraron que la tasa de competitividad óptima de un algoritmo en línea determinístico para el problema simétrico de 2 servidores es igual a 2, y conjeturaron que los problemas de *k* servidores carecen de algoritmos en línea *c* competitivos para $c < k$, para cualquier espacio métrico con por lo menos $k + 1$ vértices. Koutsoupias y Papadimitriou (1995) demostraron que el algoritmo de la función trabajo para el problema de *k* servidores posee una tasa de competitividad cuando mucho igual a $2k - 1$. Hasta la fecha, la conjetura no se ha demostrado ni refutado.

Chan y Lam (1993) propusieron un algoritmo asintóticamente óptimo $(1 + r/2)$ competitivo para el problema del recorrido de obstáculos, donde cada uno de los obstáculos tiene una tasa de aspecto (la tasa de longitudes entre el lado largo y el lado corto) acotada por alguna constante *r*.

Khuller, Mitchell y Vazirani (1994) propusieron algoritmos en línea para apareamiento ponderado y matrimonios estables. El problema del apareamiento bipartita para gráficas no ponderadas fue analizado en Karp, Vazirani y Vazirani (1990) y Kao y Tate (1991).

Karger, Phillips y Torng (1994) presentaron un algoritmo para resolver el problema de programación de *m*-máquinas, que tiene una tasa de competitividad igual cuando mucho a 1.945 para toda *m* y que sobrepasa en rendimiento al algoritmo List para $m \geq 6$.

También se explotaron muchos problemas en línea interesantes, como los sistemas de tareas métricas: Borodin, Linial y Saks (1992); problemas en línea de empaque en contenedores: Csirik (1989); Lee y Lee (1985); Vliet (1992); problemas en línea de coloreado de gráficas: Vishwanathan (1992); problemas financieros en línea: El-Yaniv (1998), etcétera.

A continuación se mencionan resultados nuevos e interesantes: Adamy y Erlebach (2003); Albers (2002); Albers y Koga (1998); Albers y Leonardi (1999); Alon, Awerbuch y Azar (2003); Aspnes, Azar, Fiat, Plotkin y Waarts (1997); Awerbuch y Peleg (1995); Awerbuch y Singh (1997); Awerbuch, Azar y Meyerson (2003); Azar, Blum y Mansour (2003); Azar, Kalyanasundaram, Plotkin, Pruhs y Waarts (1997); Azar, Naor y Rom (1995); Bachrach y El-Yaniv (1997); Bareli, Berman, Fiat y Yan (1994); Bartal y Grove (2000); Berman, Charikar y Karpinski (2000); Bern, Greene, Raghunathan y Sudan (1990); Blum, Sandholm y Zinkevich (2002); Caragiannis, Kaklamanis y Papaioannou (2003); Chan (1998); Chandra y Vishwanathan (1995); Chazelle y Matousek (1996); Chekuri, Khanna y Zhu (2001); Conn y Vonholdt (1965); Coppersmith, Doyle, Raghavan y Snir (1993); Crammer y Singer (2002); El-Yaniv (1998); Epstein y Ganot (2003); Even y Shiloach (1981); Faigle, Kern y Nawjin (1999); Feldmann, Kao, Sgall y Teng (1993); Galil y Seiferas (1978); Garay, Gopal, Kutten, Mansour y Yung (1997); Goldman, Parwatikar y Suri (2000); Gupta, Konjevod y Varsamopoulos (2002); Haas y Hallerstein (1999); Halldorson (1997); Halperin, Sharir y Goldberg (2002); Irani, Shukla y Gupta (2003); Janssen, Krizanc, Narayanan y Shende (2000); Jayram, Kimbrel, Krauthgamer, Schieber y Sviridenko (2201); Kalyanasundaram y Pruhs (1993); Keogh, Chu, Hart y Pazzani (2001); Khuller, Mitchell y Vazirani (1994); Klarlund (1999); Kolman y Scheideler (2001); Koo, Lam, Ngan, Sadakane y To (2003); Kossmann, Ramsak y Rost (2002); Lee (2003a); Lee (2003b); Lueker (1998); Manacher (1975); Mandic y Cichocki (2003); Mansour y Schieber (1992); Megow y Schulz (2003); Oza y Russell (2001); Pandurangan y Upfal (2001); Peserico (2003); Pittel y Weishaar (1997); Ramesh (1995); Seiden (1999); Seiden (2002); Sgall (1996); Tamassia (1996); Tsai, Lin y Hsu (2002); Tsai, Yung y Chen (1994); Tsai, Tang y Chen (1996); Ye y Zhang (2003), y Young (2000).

Ejercicios

12.1 Considere el problema de apareamiento bipartita presentado en el apartado 12-4. Cuando llega un vértice b_i de B, se hace que b_i se aparee con el vértice más cercano sin aparear del conjunto R. Este algoritmo, ¿es $(2n - 1)$ competitivo? Demuestre su respuesta.

12.2 Dada una gráfica ponderada bipartita con vértices de partición R y B, cada uno de cardinalidad n, el problema del apareamiento bipartita máximo consiste en encontrar un apareamiento bipartita con el costo máximo. Suponga que todos los pesos de los vértices satisfacen la desigualdad del triángulo. Si todos los vértices en R se conocen de antemano y los vértices en B se revelan uno por uno, ¿cuál es la tasa de competitividad de un algoritmo codicioso que siempre aparea un vértice recién llegado con el vértice no apareado más lejano de R?

12.3 Demuestre que el algoritmo en línea para k servidores presentado en la sección 12-2 también es k competitivo para una línea.

12.4 El algoritmo para el recorrido de obstáculos que se presentó en la sección 12-3, ¿es 3/2 competitivo para obstáculos cuadrados con direcciones arbitrarias? Demuestre su respuesta.

Bibliografía

Abel, S. (1990): A Divide and Conquer Approach to Least-Squares Estimation, *IEEE Transactions on Aerospace and Electronic Systems*, vol. 26, pp. 423-427.

Adamy, U. y Erlebach, T. (2003): Online Coloring of Intervals with Bandwidth, *LNCS 2909*, pp. 1-12.

Agarwal, P. K. y Procopiuc. C. M. (2000): Approximation Algorithms for Projective Clustering, *Journal of Algorithms*, vol. 34, pp. 128-147.

Agarwal, P. K. y Sharir, M. (1996): Efficient Randomized Algorithms for Some Geometric Optimization Problems, *Discrete Computational Geometry*, vol. 16, No. 4, pp. 317-337.

Agrawal, M., Kayal, N. y Saxena, N. (2004): PRIMES is in P, *Annals of Mathematics* (en prensa).

Aho, A. V., Ganapathi, M. y Tjang, S. (1989): Code Generation Using Tree Matching and Dynamic Programming, *ACM Transactions on Programming Languages and Systems*, vol. 11, pp. 491-516.

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

Ahuja, R. K. (1988): Minimum Cost-Reliability Ratio Path Problem, *Computer and Operations Research*, vol. 15, No. 1.

Aiello, M., Rajagopalan, S. R. y Venkatesan, R. (1998): Design of Practical and Provably Good Random Number Generators, *Journal of Algorithms*, pp. 358-389.

Akiyoshi, S. y Takeaki, U. (1997): A Linear Time Algorithm for Finding a *k*-Tree Core, *Journal of Algorithms*, vol. 23, pp. 281-290.

Akutsu, T. (1996): Protein Structure Alignment Using Dynamic Programming and Iterative Improvement, *IEICE Transactions on Information and Systems*, vol. E78-D, No. 12, pp. 1629-1636.

Akutsu, T., Arimura, H. y Shimozono, S. (2000): On Approximation Algorithms for Local Multiple Alignment, *Proceedings of the Fourth Annual International Conference on Computational Molecular Biology*, ACM Press, Tokyo, pp. 1-7.

Akutsu, T. y Halldorsson, M. M. (1994): On the Approximation of Largest Common Subtrees and Largest Common Point Sets, *Lecture Notes in Computer Science*, pp. 405-413.

Akutsu, T. y Miyano, S. (1997): On the Approximation of Protein Threading, *RECOMB*, pp. 3-8.

Akutsu, T., Miyano, S. y Kuhara, S. (2003): A Simple Greedy Algorithm for Finding Functional Relations: Efficient Implementation and Average Case Analysis, *Theoretical Computer Science*, vol. 292, pp. 481-495.

Albers, S. (2002): On Randomized Online Scheduling, *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, ACM Press, Montreal, Canadá, pp. 134-143.

Albers, S. y Koga, H. (1998): New On-Line Algorithms for the Page Replication Problem, *Journal of Algorithms*, vol. 27, pp. 75-96.

Albers, S. y Leonardi, S. (1999): On-Line Algorithms, *ACM Computing Surveys (CSUR)*, vol. 31, No. 3, Article No. 4, septiembre.

Alberts, D. y Henzinger, M. R. (1995): Average Case Analysis of Dynamic Graph Algorithms, *Symposium in Discrete Algorithms*, pp. 312-321.

Aldous, D. (1989): *Probability Approximations via the Poisson Clumping Heuristic*, Springer-Verlag, Berlin.

Aleksandrov, L. y Djidjev, H. (1996): Linear Algorithms for Partitioning Embedded Graphs of Bounded Genus, *SIAM Journal on Discrete Mathematics*, vol. 9, No. 1, pp. 129-150.

Alon, N., Awerbuch, B. y Azar, Y. (2003): Session 2B: The Online Set Cover Problem, *Proceedings of the 35th ACM Symposium on Theory of Computing*, ACM Press, San Diego, California, pp. 100-105.

Alon, N. y Azar, Y. (1989): Finding an Approximate Maximum, *SIAM Journal on Computing*, vol. 18, No. 2, pp. 258-267.

Alon, N. y Azar, Y. (1993): On-Line Steiner Trees in the Euclidean Plane, *Discrete Computational Geometry*, vol. 10, No. 2, pp. 113-121.

Alon, N., Babai, L. e Itai, A. (1986): A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem, *Journal of Algorithms*, vol. 7, No. 4, pp. 567-583.

Alpert, C. J. y Kahng, A. B. (1995): Multi-Way Partitioning via Geometric Embeddings; Orderings; and Dynamic Programming, *IEEE Transactions on CAD*, vol. 14, pp. 1342-1358.

Amini, A. A., Weymouth, T. E. y Jain, R. C. (1990): Using Dynamic Programming for Solving Variational Problems in Vision, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, No. 9, pp. 855-867.

Amir, A. y Farach, M. (1995): Efficient 2-Dimensional Approximate Matching of Half-Rectangular Figures, *Information and Computation*, vol. 118, pp. 1-11.

Amir, A. y Keselman, D. (1997): Maximum Agreement Subtree in a Set of Evolutionary Trees: Metrics and Efficient Algorithms, *SIAM Journal on Computing*, vol. 26, pp. 1656-1669.

Amir, A. y Landau, G. (1991): Fast Parallel and Serial Multidimensional Approximate Array Matching, *Theoretical Computer Science*, vol. 81, pp. 97-115.

Anderson, R. (1987): A Parallel Algorithm for the Maximal Path Problem, *Combinatorica*, vol. 7, No. 4, pp. 315-326.

Anderson, R. J. y Woll, H. (1997): Algorithms for the Certified Write-All Problem, *SIAM Journal on Computing*, vol. 26, No. 5, pp. 1277-1283.

Ando, K., Fujishige, S. y Naitoh, T. (1995): A Greedy Algorithm for Minimizing a Separable Convex Function over a Finite Jump System, *Journal of the Operations Research Society of Japan*, vol. 38, pp. 362-375.

Arkin, E. M., Chiang, Y. J., Mitchell, J. S. B., Skiena, S. S. y Yang, T. C. (1999): On the Maximum Scatter Traveling Salesperson Problem, *SIAM Journal on Computing*, vol. 29, pp. 515-544.

Armen, C. y Stein, C. (1994): A 2 3/4-Approximation Algorithm for the Shortest Superstring Problem, *Technical Report* (PCS-TR94-214), Computer Science Department, Dartmouth College, Hanover, Nueva Hampshire.

Armen, C. y Stein, C. (1995): Improved Length Bounds for the Shortest Superstring Problem (Shortest Common Superstring: 2 3/4-Approximation), *Lecture Notes in Computer Science*, vol. 955, pp. 494-505.

Armen, C. y Stein, C. (1996): A 2 2/3 Approximation Algorithm for the Shortest Superstring Problem, *Lecture Notes in Computer Science*, vol. 1075, pp. 87-101.

Armen, C. y Stein, C. (1998): 2 2/3 Superstring Approximation Algorithm, *Discrete Applied Mathematics*, vol. 88, No. 1-3, pp. 29-57.

Arora, S. (1996): Polynomial Approximation Schemes for Euclidean TSP and Other Geometric Problems, *Foundations of Computer Science*, pp. 2-13.

Arora, S. y Brinkman, B. (2002): A Randomized Online Algorithm for Bandwidth Utilization, *Symposium on Discrete Algorithms*, pp. 535-539.

Arora, S., Lund, C., Motwani, R., Sudan, M. y Szegedy, M. (1998): Proof Verification and the Hardness of Approximation Problems (Prove NP-Complete Problem), *Journal of the ACM*, vol. 45, No. 3, pp. 501-555.

Arratia, R., Goldstein, L. y Gordon, L. (1989): Two Moments Suffice for Poisson Approximation: The Chen-Stein Method, *The Annals of Probability*, vol. 17, pp. 9-25.

Arratia, R., Martin, D., Reinert, G. y Waterman, M. S. (1996): Poisson Process Approximation for Sequence Repeats and Sequencing by Hybridization, *Journal of Computational Biology*, vol. 3, pp. 425-464.

Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R. y Wu, A. Y. (1998): An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions, *Journal of the ACM*, vol. 45, No. 6, pp. 891-923.

Ashenhurst, R. L. (1987): *ACM Turing Award Lectures: The First Twenty Years: 1966-1985*, ACM Press, Baltimore, Maryland.

Aspnes, J., Azar, Y., Fiat, A., Plotkin, S. y Waarts, O. (1997): On-Line Routing of Virtual Circuits with Applications to Load Balancing and Machine Scheduling, *Journal of the ACM*, vol. 44, No. 3, pp. 486-504.

Atallah, M. J. y Hambrusch, S. E. (1986): An Assignment Algorithm with Applications to Integrated Circuit Layout, *Discrete Applied Mathematics*, vol. 13, No. 1, pp. 9-22.

Auletta, V., Parente, D. y Persiano, G. (1996): Dynamic and Static Algorithms for Optimal Placement of Resources in Trees, *Theoretical Computer Science*, vol. 165, No. 2, pp. 441-461.

Ausiello, G., Crescenzi, P. y Protasi, M. (1995): Approximate Solution of NP Optimization Problems, *Theoretical Computer Science*, vol. 150, pp. 1-55.

Avis, D., Bose, P., Shermer, T. C., Snoeyink, J., Toussaint, G. y Zhu, B. (1996): On the Sectional Area of Convex Polytopes, *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, ACM Press, Philadelphia, Pennsylvania, pp. 411-412.

Avrim, B., Jiang, T., Li, M., Tromp, J. y Yannakakis, M. (1991): Linear Approximation of Shortest Superstrings, *Proceedings of the 23rd ACM Symposium on Theory of Computation*, ACM Press, Nueva Orleans, Louisiana, pp. 328-336.

Awerbuch, B., Azar, Y. y Meyerson, A. (2003): Reducing Truth-Telling Online Mechanisms to Online Optimization, *Proceedings of the 35th ACM Symposium on Theory of Computing*, ACM Press, San Diego, California, pp. 503-510.

Awerbuch, B. y Peleg, D. (1995): On-Line Tracking of Mobile Users, *Journal of the ACM*, vol. 42, No. 5, pp. 1021-1058.

Awerbuch, B. y Singh, T. (1997): On-Line Algorithms for Selective Multicast and Maximal Dense Trees, *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, ACM Press, El Paso, Texas, pp. 354-362.

Azar, Y., Blum, A. y Mansour, Y. (2003): Combining Online Algorithms for Rejection and Acceptance, *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, ACM Press, San Diego, California, pp. 159-163.

Azar, Y., Kalyanasundaram, B., Plotkin, S., Pruhs, K. y Waarts, O. (1997): On-Line Load Balancing of Temporary Tasks, *Journal of Algorithms*, vol. 22, pp. 93-110.

Azar, Y., Naor, J. y Rom, R. (1995): The Competitiveness of On-Line Assignments, *Journal of Algorithms*, vol. 18, pp. 221-237.

Bachrach, R. y El-Yaniv, R. (1997): Online List Accessing Algorithms and Their Applications: Recent Empirical Evidence, *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, Nueva Orleans, Louisiana, pp. 53-62.

Baeza-Yates, R. A. y Navarro, G. (1999): Faster Approximate String Matching, *Algorithmica*, vol. 23, No. 2, pp. 127-158.

Baeza-Yates, R. A. y Perleberg, C. H. (1992): Fast and Practical Approximate String Matching, *Lecture Notes in Computer Science*, vol. 644, pp. 185-192.

Bafna, V., Berman, P. y Fujito, T. (1999): A 2-Approximation Algorithm for the Undirected Feedback Vertex Set Problem, *SIAM Journal on Discrete Mathematics*, vol. 12, No. 3, pp. 289-297.

Bafna, V., Lawler, E. L. y Pevzner, P. A. (1997): Approximation Algorithms for Multiple Sequence Alignment, *Theoretical Computer Science*, vol. 182, pp. 233-244.

Bafna, V. y Pevzner, P. (1996): Genome Rearrangements and Sorting by Reversals (Approximation Algorithm), *SIAM Journal on Computing*, vol. 25, No. 2, pp. 272-289.

Bafna, V. y Pevzner, P. A. (1998): Sorting by Transpositions, *SIAM Journal on Discrete Mathematics*, vol. 11, No. 2, pp. 224-240.

Bagchi, A. y Mahanti, A. (1983): Search Algorithms under Different Kinds of Heuristics—A Comparative Study, *Journal of the ACM*, vol. 30, No. 1, pp. 1-21.

Baker, B. S. (1994): Approximation Algorithms for NP-Complete Problems on Planar Graphs, *Journal of the ACM*, vol. 41, No. 1.

Baker, B. S. y Coffman, E. G. Jr. (1982): A Two-Dimensional Bin-Packing Model of Preemptive FIFO Storage Allocation, *Journal of Algorithms*, vol. 3, pp. 303-316.

Baker, B. S. y Giancarlo, R. (2002): Sparse Dynamic Programming for Longest Common Subsequence from Fragments, *Journal of Algorithm*, vol. 42, pp. 231-254.

Balas, F. y Yu, C. S. (1986): Finding a Maximum Clique in an Arbitrary Graph, *SIAM Journal on Computing*, vol. 15, No. 4, pp. 1054-1068.

Bandelloni, M., Tucci, M. y Rinaldi, R. (1994): Optimal Resource Leveling Using Non-Serial Dynamic Programming, *European Journal of Operational Research*, vol. 78, pp. 162-177.

Barbu, V. (1991): The Dynamic Programming Equation for the Time Optimal Control Problem in Infinite Dimensions, *SIAM Journal on Control and Optimization*, vol. 29, pp. 445-456.

Bareli, E., Berman, P., Fiat, A. y Yan, P. (1994): Online Navigation in a Room, *Journal of Algorithms*, vol. 17, pp. 319-341.

Bartal, Y., Fiat, A., Karloff, H. y Vohra, R. (1995): New Algorithms for an Ancient Scheduling Problem, *Journal of Computer and System Sciences*, vol. 51, No. 3, pp. 359-366.

Bartal, Y. y Grove, E. (2000): The Harmonic *k*-Server Algorithm is Competitive, *Journal of the ACM*, vol. 47, No. 1, pp. 1-15.

Basse, S. y Van Gelder, A. (2000): *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, Reading, Mass.

Bein, W. W. y Brucker, P. (1986): Greedy Concepts for Network Flow Problems, *Discrete Applied Mathematics*, vol. 15, No. 2, pp. 135-144.

Bein, W. W., Brucker, P. y Tamir, A. (1985): Minimum Cost Flow Algorithm for Series-Parallel Networks, *Discrete Applied Mathematics*, vol. 10, No. 3, pp. 117-124.

Bekesi, J., Galambos, G., Pferschy, U. y Woeginger, G. (1997): Greedy Algorithms for On-Line Data Compression, *Journal of Algorithms*, vol. 25, pp. 274-289.

Bellman, R. y Dreyfus, S. E. (1962): *Applied Dynamic Programming*, Princeton University Press, Princeton, Nueva Jersey.

Ben-Asher, Y., Farchi, E. y Newman, I. (1999): Optimal Search in Trees, *SIAM Journal on Mathematics*, vol. 28, No. 6, pp. 2090-2102.

Bent, S. W., Sleator, D. D. y Tarjan, R. E. (1985): Biased Search Trees, *SIAM Journal on Computing*, vol. 14, No. 3, pp. 545-568.

Bentley, J. L. (1980): Multidimensional Divide-and-Conquer, *Communications of the ACM*, vol. 23, No. 4, pp. 214-229.

Bentley, J. L., Faust, G. M. y Preparata, F. P. (1982): Approximation Algorithms for Convex Hulls, *Communications of the ACM*, vol. 25, pp. 64-68.

Bentley, J. L. y McGeoch, C. C. (1985): Amortized Analysis of Self-Organizing Sequential Search Heuristics, *Communications of the ACM*, vol. 28, No. 4, pp. 404-411.

Bentley, J. L. y Shamos, M. I. (1978): Divide-and-Conquer for Linear Expected Time, *Information Processing Letters*, vol. 7, No. 2, pp. 87-91.

Berger, B. y Leighton, T. (1998): Protein Folding in the Hydrophobic-Hydrophilic (HP) Model is NP-Complete (Prove NP-Complete Problem), *Journal of Computational Biology*, vol. 5, No. 1, pp. 27-40.

Berger, R. (1966): The Undecidability of the Domino Problem, *Memoirs of the American Mathematical Society*, No. 66.

Berman, P., Charikar, M. y Karpinski, M. (2000): On-Line Load Balancing for Related Machines, *Journal of Algorithms*, vol. 35, pp. 108-121.

Berman, P., Hannenhalli, S. y Karpinki, M. (2001): 1.375 – Approximation Algorithm for Sorting by Reversals, *Technical Report DIMACS*, TR2001-41.

Berman, P., Karpinski, M., Larmore, L. L., Plandowski, W. y Rytter, W. (2002): On the Complexity of Pattern Matching for Highly Compressed Two-Dimensional Texts, *Journal of Computer System Sciences*, vol. 65, No. 2, pp. 332-350.

Bern, M., Greene, D., Raghunathan, A. y Sudan, M. (1990): Online Algorithms for Locating Checkpoints, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, ACM Press, Baltimore, Maryland, pp. 359-368.

Bhagavathi, D., Grosch, C. E. y Olariu, S. (1994): A Greedy Hypercube-Labeling Algorithm, *The Computer Journal*, vol. 37, pp. 124-128.

Bhattacharya, B. K., Jadhav, S., Mukhopadhyay, A. y Robert, J. M. (1994): Optimal Algorithms for Some Intersection Radius Problems, *Computing*, vol. 52, No. 3, pp. 269-279.

Blankenagel, G. y Gueting, R. H. (1990): Internal and External Algorithms for the Points-in-Regions Problem, *Algorithmica*, vol. 5, No. 2, pp. 251-276.

Blazewicz, J. y Kasprzak, M. (2003): Complexity of DNA Sequencing by Hybridization, *Theoretical Computer Science*, vol. 290, No. 3, pp. 1459-1473.

Blot, J., Fernandez de la Vega, W., Paschos, V. T. y Saad, R. (1995): Average Case Analysis of Greedy Algorithms for Optimization Problems on Set Systems, *Theoretical Computer Science*, vol. 147, No. 1-2, pp. 267-298.

Blum, A. (1994): New Approximation Algorithms for Graph Coloring, *Journal of the ACM*, vol. 41, No. 3.

Blum, A., Jiang, T., Li, M., Tromp, J. y Yannakakis, M. (1994): Linear Approximation of Shortest Superstrings, *Journal of the ACM*, vol. 41, pp. 630-647.

Blum, A., Sandholm, T. y Zinkevich, M. (2002): Online Algorithms for Market Clearing, *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, San Francisco, California, pp. 971-980.

Blum, M., Floyd, R. W., Pratt, V. R., Rivest, R. L. y Tarjan, R. E. (1972): Time Bounds for Selection, *Journal of Computer and System Sciences*, vol. 7, No. 4, pp. 448-461.

Bodlaender, H. L. (1988): The Complexity of Finding Uniform Emulations on Fixed Graphs, *Information Processing Letters*, vol. 29, No. 3, pp. 137-141.

Bodlaender, H. L. (1993): Complexity of Path-Forming Games, *Theoretical Computer Science*, vol. 110, No. 1, pp. 215-245.

Bodlaender, H. L., Downey, R. G., Fellows, M. R. y Wareham, H. T. (1995): The Parameterized Complexity of Sequence Alignment and Consensus, *Theoretical Computer Science*, vol. 147, pp. 31-54.

Bodlaender, H. L., Fellows, M. R. y Hallet, M. T. (1994): Beyond NP-Completeness for Problems of Bounded Width: Hardness for the W Hierarchy (resumen amplio), *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing*, ACM Press, Nueva York, pp. 449-458.

Boffey, T. B. y Green, J. R. (1983): Design of Electricity Supply Networks, *Discrete Applied Mathematics*, vol. 5, pp. 25-38.

Boldi, P. y Vigna, S. (1999): Complexity of Deciding Sense of Direction, *SIAM Journal on Computing*, vol. 29, No. 3, pp. 779-789.

Bonizzoni, P. y Vedova, G. D. (2001): The Complexity of Multiple Sequence Alignment with SP-Score that is a Metric, *Theoretical Computer Science*, vol. 259, pp. 63-79.

Bonizzoni, P., Vedova, G. D. y Mauri, G. (2001): Experimenting an Approximation Algorithm for the LCS, *Discrete Applied Mathematics*, vol. 110, No. 1, pp. 13-24.

Boppana, R. B., Hastad, J. y Zachos, S. (1987): Does Co-NO Have Short Interactive Proofs? *Information Processing Letters*, vol. 25, No. 2, pp. 127-132.

Boreale, M. y Trevisan, L. (2000): A Complexity Analysis of Bisimilarity for Value-Passing Processes, *Theoretical Computer Science*, vol. 238, No. 1, pp. 313-345.

Borodin, A. y El-Yaniv, R. (1998): *Online Computation and Competitive Analysis*, Cambridge University Press, Cambridge, England.

Borodin, A., Linial, N. y Saks, M. (1992): An Optimal On-line Algorithm for Metrical Task System, *Journal of the ACM*, vol. 39, No. 4, pp. 745-763.

Boros, E., Crama, Y., Hammer, P. L. y Saks, M. (1994): A Complexity Index for Satisfiability Problems, *SIAM Journal on Computing*, vol. 23, No. 1, pp. 45-49.

Boruvka, O. (1926): O Jistem Problemu Minimalmim. *Praca Moravske Prirodovedecke Spoleconosti*, vol. 3, pp. 37-58.

Bossi, A., Cocco, N. y Colussi, L. (1983): A Divide-and-Conquer Approach to General Context-Free Parsing, *Information Processing Letters*, vol. 16, No. 4, pp. 203-208.

Brassard, G. y Bratley, P. (1988): *Algorithmics: Theory and Practice*, Prentice-Hall, Englewood Cliffs, Nueva Jersey.

Breen, S., Waterman, M. S. y Zhang, N. (1985): Renewal Theory for Several Patterns, *Journal of Applied Probability*, vol. 22, pp. 228-234.

Breslauer, D., Jiang, T. y Jiang, Z. J. (1997): Rotations of Periodic Strings and Short Superstrings (2.596 – Approximation), *Algorithms*, vol. 24, No. 2, pp. 340-353.

Bridson, R. y Tang, W. P. (2001): Multiresolution Approximate Inverse Preconditioners, *SIAM Journal on Scientific Computing*, vol. 23, pp. 463-479.

Brigham, E. O. (1974): *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, Nueva Jersey.

Brown, C. A. y Purdom, P. W. Jr. (1981): An Average Time Analysis of Backtracking, *SIAM Journal on Computing*, vol. 10, pp. 583-593.

Brown, K. Q. (1979): Voronoi Diagrams from Convex Hulls, *Information Processing Letters*, vol. 9, pp. 223-228.

Brown, M. L. y Whitney, D. E. (1994): Stochastic Dynamic Programming Applied to Planning of Robot Grinding Tasks, *IEEE Transactions on Robotics and Automation*, pp. 594-604.

Brown, M. R. y Tarjan, R. E. (1980): Design and Analysis of a Data Structure for Representing Sorted Lists, *SIAM Journal on Computing*, vol. 9, No. 3, pp. 594-614.

Bruno, J., Coffman, E. G. Jr. y Sethi, R. (1974): Scheduling Independent Tasks to Reduce Mean Finishing Time, *Communications of the ACM*, vol. 17, No. 7, pp. 382-387.

Bryant, D. (1998): The Complexity of the Breakpoint Median Problem, *Technical Report CRM-2579*, pp. 1-12.

Caballero-Gil, P. (2000): New Upper Bounds on the Linear Complexity, *Computers and Mathematics with Applications*, vol. 39, No. 3, pp. 31-38.

Cai, J. Y. y Meyer, G. E. (1987): Graph Minimal Uncolorability Is DP-Complete, *SIAM Journal on Computing*, vol. 16, No. 2, pp. 259-277.

Caprara, A. (1997a): Sorting by Reversals Is Difficult, *Proceedings of the First Annual International Conference on Computational Molecular Biology*, ACM Press, Santa Fe, Nuevo México, pp. 75-83.

Caprara, A. (1997b): Sorting Permutations by Reversals and Eulerian Cycle Decompositions, *SIAM Journal on Discrete Mathematics*, pp. 1-23.

Caprara, A. (1999): Formulations and Hardness of Multiple Sorting by Reversals, *Proceedings of the 3rd Annual International Conference on Computational Molecular Biology*, ACM Press, Lyon, Francia, pp. 84-93.

Caragiannis, I., Kaklamanis, C. y Papaioannou, E. (2003): Simple On-Line Algorithms for Call Control in Cellular Networks, *Lecture Notes in Computer Science*, vol. 2909, pp. 67-80.

Cary, M. (2001): Toward Optimal ε-Approximate Nearest Neighbor Algorithms, *Journal of Algorithms*, vol. 41, pp. 417-428.

Chan, K. F. y Lam, T. W. (1993): An On-Line Algorithm for Navigating in Unknown Environment, *International Journal of Computational Geometry and Applications*, vol. 3, No. 3, pp. 227-244.

Chan, T. (1998): Deterministic Algorithms for 2-D Convex Programming and 3-D Online Linear Programming, *Journal of Algorithms*, vol. 27, pp. 147-166.

Chandra, B. y Vishwanathan, S. (1995): Constructing Reliable Communication Networks of Small Weight On-Line, *Journal of Algorithms*, vol. 18, pp. 159-175.

Chang, C. L. y Lee, R. C. T. (1973): *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, Nueva York.

Chang, K. C. y Du, H. C. (1988): Layer Assignment Problem for Three-Layer Routing, *IEEE Transactions on Computers*, vol. 37, pp. 625-632.

Chang, W. I. y Lampe, J. (1992): Theoretical and Empirical Comparisons of Approximate String Matching Algorithms, *Lecture Notes in Computer Science*, vol. 644, pp. 172-181.

Chang, W. I. y Lawler, E. L. (1994): Sublinear Approximate String Matching and Biological Applications, *Algorithmica*, vol. 12, No. 4-5, pp. 327-344.

Chao, H. S. (1992): *On-line algorithms for three computational geometry problems*. Tesis doctoral, aún sin publicar, National Tsing Hua University, Hsinchu, Taiwán.

Chao, M. T. (1985): *Probabilistic analysis and performance measurement of algorithms for the satisfiability problem*. Tesis doctoral, aún sin publicar, Case Western Reserve University, Cleveland, Ohio.

Chao, M. T. y Franco, J. (1986): Probabilistic Analysis of Two Heuristics for the 3-Satisfiability Problem, *SIAM Journal on Computing*, vol. 15, No. 4, pp. 1106-1118.

Charalambous, C. (1997): Partially Observable Nonlinear Risk-Sensitive Control Problems: Dynamic Programming and Verification Theorems, *IEEE Transactions on Automatic Control*, vol. 42, pp. 1130-1138.

Chazelle, B., Drysdale, R. L. y Lee, D. T. (1986): Computing the Largest Empty Rectangle, *SIAM Journal on Computing*, vol. 15, No. 1, pp. 300-315.

Chazelle, B., Edelsbrunner, H., Guibas, L., Sharir, M. y Snoeyink, J. (1993): Computing a Face in an Arrangement of Line Segments and Related Problems, *SIAM Journal on Computing*, vol. 22, No. 6, pp. 1286-1302.

Chazelle, B. y Matousek, J. (1996): On Linear-Time Deterministic Algorithms for Optimization Problems in Fixed Dimension, *Journal of Algorithm*, vol. 21, pp. 579-597.

Chekuri, C., Khanna, S. y Zhu, A. (2001): Algorithms for Minimizing Weighted Flow Time, *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, ACM Press, Hersonissos, Grecia, pp. 84-93.

Chen, G. H., Chern, M. S. y Jang, J. H. (1990): Pipeline Architectures for Dynamic Programming Algorithms, *Parallel Computing*, vol. 13, pp. 111-117.

Chen, G. H., Kuo, M. T. y Sheu, J. P. (1988): An Optimal Time Algorithm for Finding a Maximum Weighted Independent Set in a Tree, *BIT*, vol. 28, pp. 353-356.

Chen, J. y Miranda, A. (2001): A Polynomial Time Approximation Scheme for General Multiprocessor Job Scheduling, *SIAM Journal on Computing*, vol. 31, pp. 1-17.

Chen, L. H. Y. (1975): Poisson Approximation for Dependent Trials, *The Annals of Probability*, vol. 3, pp. 534-545.

Chen, T. S., Yang, W. P. y Lee, R. C. T. (1989): Amortized Analysis of Disk Scheduling Algorithms, *Technical Report*, Department of Information Engineering, National Chiao-Tung University, Taiwán.

Chen, W. M. y Hwang, H. K. (2003): Analysis in Distribution of Two Randomized Algorithms for Finding the Maximum in a Broadcast Communication Model, *Journal of Algorithms*, vol. 46, pp. 140-177.

Cheriyan, J. y Harerup, T. (1995): Randomized Maximum-Flow Algorithm, *SIAM Journal on Computing*, vol. 24, No. 2, pp. 203-226.

Chin, W. y Ntafos, S. (1988): Optimum Watchman Routes, *Information Processing Letters*, vol. 28, No. 1, pp. 39-44.

Chou, H. C. y Chung C. P. (1994): Optimal Multiprocessor Task Scheduling Using Dominance and Equivalence Relations, *Computer & Operations Research*, vol. 21, No. 4, pp. 463-475.

Choukhmane, E. y Franco, J. (1986): An Approximation Algorithm for the Maximum Independent Set Problem in Cubic Planar Graphs, *Networks*, vol. 16, No. 4, pp. 349-356.

Christofides, N. (1976): Worst-Case Analysis of a New Heuristic for the Traveling Salesman Problem, *Management Sciences Research Report*, No. 388.

Christos, L. y Drago, K. (1998): Quasi-Greedy Triangulations Approximating the Minimum Weight Triangulation, *Journal of Algorithms*, vol. 27, No. 2, pp. 303-338.

Chrobak, M. y Larmore, L. L. (1991): An Optimal On-Line Algorithm for *k*-Servers on Trees, *SIAM Journal of Computing*, vol. 20, No. 1, pp. 144-148.

Chu, C. y La, R. (2001): Variable-Sized Bin Packing: Tight Absolute Worst-Case Performance Ratios for Four Approximation Algorithms, *SIAM Journal on Computing*, vol. 30, pp. 2069-2083.

Chung, M. J. y Krishnamoorthy, M. S. (1988): Algorithms of Placing Recovery Points, *Information Processing Letters*, vol. 28, No. 4, pp. 177-181.

Chung, M. J., Makedon, F., Sudborough, I. H. y Turner, J. (1985): Polynomial Time Algorithms for the Min Cut Problem on Degree Restricted Trees, *SIAM Journal on Computing*, vol. 14, No. 1, pp. 158-177.

Cidon, I., Kutten, S., Mansour, Y. y Peleg, D. (1995): Greedy Packet Scheduling, *SIAM Journal on Computing*, vol. 24, pp. 148-157.

Clarkson, K. L. (1987): New Applications of Random Sampling to Computational Geometry, *Discrete and Computational Geometry*, vol. 2, pp. 195-222.

Clarkson, K. L. (1988): A Randomized Algorithm for Closest-Point Queries, *SIAM Journal on Computing*, vol. 17, No. 4, pp. 830-847.

Clarkson, K. L. (1994): An Algorithm for Approximate Closest-Point Queries, *Proceedings of the 10th Annual Symposium on Computational Geometry*, ACM Press, Stony Brook, Nueva York, pp. 160-164.

Cobbs, A. (1995): Fast Approximate Matching Using Suffix Trees, *Lecture Notes in Computer Science*, vol. 937, pp. 41-54.

Coffman, E. G., Langston, J. y Langston, M. A. (1984): A Performance Guarantee for the Greedy Set-Partitioning Algorithm, *Acta Informatica*, vol. 21, pp. 409-415.

Coffman, E. G. y Lueker, G. S. (1991): *Probabilistic Analysis of Packaging & Partitioning Algorithms*, John Wiley & Sons, Nueva York.

Colbourn, C. J., Kocay, W. L. y Stinson, D. R. (1986): Some NP-Complete Problems for Hypergraph Degree Sequences, *Discrete Applied Mathematics*, vol. 14, No. 3, pp. 239-254.

Cole, R. (1994): Tight Bounds on the Complexity of the Boyer-Moore String Matching Algorithm, *SIAM Journal on Computing*, vol. 23, No. 5, pp. 1075-1091.

Cole, R., Farach-Colton, M., Hariharan, R., Przytycka, T. y Thorup, M. (2000): An $O(n \log n)$ Algorithm for the Maximum Agreement Subtree Problem for Binary Trees, *SIAM Journal on Applied Mathematics*, vol. 30, No. 5, pp. 1385-1404.

Cole, R. y Hariharan, R. (1997): Tighter Upper Bounds on the Exact Complexity of String Matching, *SIAM Journal on Computing*, vol. 26, No. 3, pp. 803-856.

Cole, R., Hariharan, R., Paterson, M. y Zwick, U. (1995): Tighter Lower Bounds on the Exact Complexity of String Matching, *SIAM Journal on Computing*, vol. 24, No. 1, pp. 30-45.

Coleman, T. F., Edenbrandt, A. y Gilbert, J. R. (1986): Predicting Fill for Sparse Orthogonal Factorization, *Journal of the ACM*, vol. 33, No. 3, pp. 517-532.

Conn, R. y Vonholdt, R. (1965): An Online Display for the Study of Approximating Functions, *Journal of the ACM*, vol. 12, No. 3, pp. 326-349.

Cook, S. A. (1971): The Complexity of Theorem Proving Procedures, *Proceedings of the 3rd ACM Symposium on Theory of Computing*, ACM Press, Shaker Heights, Ohio, pp. 151-158.

Cooley, J. W. y Tukey, J. W. (1965): An Algorithm for the Machine Calculation of Complex Fourier Series, *Mathematics of Computation*, vol. 19, pp. 297-301.

Coppersmith, D., Doyle, P., Raghavan, P. y Snir, M. (1993): Random Walks on Weighted Graphs and Applications to On-line Algorithms, *Journal of the ACM*, vol. 40, No. 3, pp. 421-453.

Cormen, T. H. (1999): Determining an Out-of-Core FFT Decomposition Strategy for Parallel Disks by Dynamic Programming, *Algorithms for Parallel Processing*, vol. 105, pp. 307-320.

Cormen, T. H. (2001): *Introduction to Algorithms*, McGraw-Hill, Nueva York.

Cormen, T. H., Leiserson, C. E. y Rivest, R. L. (1990): *Introduction to Algorithms*, McGraw-Hill, Nueva York.

Cornuejols, C., Fisher, M. L. y Nemhauser, G. L. (1977): Location of Bank Accounts to Optimize Float: An Analytic Study of Exact and Approximate Algorithms, *Management Science*, vol. 23, No. 8, pp. 789-810.

Cowureur, C. y Bresler, Y. (2000): On the Optimality of the Backward Greedy Algorithm for the Subset Selection Problem, *SIAM Journal on Matrix Analysis and Applications*, vol. 21, pp. 797-808.

Crammer, K. y Singer, Y. (2002): Text Categorization: A New Family of Online Algorithms for Category Ranking, *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ACM Press, Tampere, Finlandia, pp. 151-158.

Crescenzi, P., Goldman, D., Papadimitriou, C., Piccolboni, A. y Yannakakis, M. (1998): On the Complexity of Protein Folding, *Journal of Computational Biology*, vol. 5, No. 3, pp. 423-465.

Csirik, J. (1989): An On-Line Algorithm For Variable-Sized Bin Packing, *Acta Informatica*, vol. 26, pp. 697-709.

Csur, M. y Kao, M. Y. (2001): Provably Fast and Accurate Recovery of Evolutionary Trees through Harmonic Greedy Triplets, *SIAM Journal on Computing*, vol. 31, pp. 306-322.

Culberson, J. C. y Rudnicki, P. (1989): A Fast Algorithm for Constructing Trees from Distance Matrices, *Information Processing Letters*, vol. 30, No. 4, pp. 215-220.

Cunningham, W. H. (1985): Optimal Attack and Reinforcement of a Network, *Journal of the ACM*, vol. 32, No. 3, pp. 549-561.

Czumaj, A., Gasieniec, L., Piotrow, M. y Rytter, W. (1994): Parallel and Sequential Approximation of Shortest Superstrings, *Lecture Notes in Computer Science*, vol. 824, pp. 95-106.

d'Amore, F. y Liberatore, V. (1994): List Update Problem and the Retrieval of Sets, *Theoretical Computer Science*, vol. 130, No. 1, pp. 101-123.

Darve, E. (2000): The Fast Multipole Method I: Error Analysis and Asymptotic Complexity, *SIAM Journal on Numerical Analysis*, vol. 38, No. 1, pp. 98-128.

Day, W. H. (1987): Computational Complexity of Inferring Phylogenies from Dissimilarity Matrices, *Bulletin of Mathematical Biology*, vol. 49, No. 4, pp. 461-467.

Decatur, S. E., Goldreich, O. y Ron, D. (1999): Computational Sample Complexity, *SIAM Journal on Computing*, vol. 29, No. 3.

Dechter, R. y Pearl, J. (1985): Generalized Best-First Search Strategies and the Optimality of A*, *Journal of the ACM*, vol. 32, No. 3, pp. 505-536.

Delcoigne, A. y Hansen, P. (1975): Sequence Comparison by Dynamic Programming, *Biometrika*, vol. 62, pp. 661-664.

Denardo, E. V. (1982): *Dynamic Programming: Model and Applications*, Prentice-Hall, Englewood Cliffs, Nueva Jersey.

Deng, X. y Mahajan, S. (1991): Infinite Games: Randomization Computability and Applications to Online Problems, *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, ACM Press, Nueva Orleans, Louisiana, pp. 289-298.

Derfel, G. y Vogl, F. (2000): Divide-and-Conquer Recurrences: Classification of Asymptotics, *Aequationes Mathematicae*, vol. 60, pp. 243-257.

Devroye, L. (2002): Limit Laws for Sums of Functions of Subtrees of Random Binary Search Trees, *SIAM Journal on Applied Mathematics*, vol. 32, No. 1, pp. 152-171.

Devroye, L. y Robson, J. M. (1995): On the Generation of Random Binary Search Trees, *SIAM Journal on Applied Mathematics*, vol. 24, No. 6, pp. 1141-1156.

Dietterich, T. G. (2000): The Divide-and-Conquer Manifesto, *Lecture Notes in Artificial Intelligence*, vol. 1968, pp. 13-26.

Dijkstra, E. W. (1959): A Note on Two Problems in Connexion with Graphs, *Numerische Mathematik*, vol. 1, pp. 269-271.

Dinitz, Y. y Nutov, Z. (1999): A 3-Approximation Algorithm for Finding Optimum 4, 5-Vertex-Connected Spanning Subgraphs, *Journal of Algorithms*, vol. 32, pp. 31-40.

Dixon, B., Rauch, M. y Tarjan, R. E. (1992): Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time, *SIAM Journal on Computing*, vol. 21, pp. 1184-1192.

Dobkin, D. P. y Lipton, R. J. (1979): On the Complexity of Computations Under Varying Sets of Primitives, *Journal of Computer and System Sciences*, vol. 18, pp. 86-91.

Dolan, A. y Aldus, J. (1993): *Networks and Algorithms: An Introductory Approach*, John Wiley & Sons, Nueva York.

Dolev, D., Lynch, N. A., Pinter, S. S., Stark, E. W. y William, E. W. (1986): Reaching Approximate Agreement in the Presence of Faults, *Journal of the ACM*, vol. 33, No. 3, pp. 499-516.

Drake, D. E. y Hougardy, S. (2003): A Simple Approximation Algorithm for the Weighted Matching Problem, *Information Processing Letters*, vol. 85, pp. 211-213.

Dreyfus, S. E. y Law, A. M. (1977): *The Art and Theory of Dynamic Programming*, Academic Press, Londres.

Du, D. Z. y Book, R. V. (1989): On Inefficient Special Cases of NP-Complete Problems, *Theoretical Computer Science*, vol. 63, No. 3, pp. 239-252.

Du, J. y Leung, J. Y. T. (1988): Scheduling Tree-Structured Tasks with Restricted Execution Times, *Information Processing Letters*, vol. 28, No. 4, pp. 183-188.

Dwyer, R. A. (1987): A Faster Divide-and-Conquer Algorithm for Constructing Delaunay Triangulations, *Algorithmics*, vol. 2, pp. 137-151.

Dyer, M. E. (1984): Linear Time Algorithm for Two- and Three-Variable Linear Programs, *SIAM Journal on Computing*, vol. 13, No. 1, pp. 31-45.

Dyer, M. E. y Frieze, A. M. (1989): Randomized Algorithm for Fixed-Dimensional Linear Programming, *Mathematical Programming*, vol. 44, No. 2, pp. 203-212.

Edelsbrunner, H. (1987): *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlín.

Edelsbrunner, H. y Guibas, L. J. (1989): Topologically Sweeping an Arrangement, *Journal of Computer and System Sciences*, vol. 38, No. 1, pp. 165-194.

Edelsbrunner, H., Maurer, H. A., Preparata, F. P., Rosenberg, A. L., Welzl, E. y Wood, D. (1982): Stabbing Line Segments, *BIT*, vol. 22, pp. 274-281.

Edwards, C. S. y Elphick, C. H. (1983): Lower Bounds for the Clique and the Chromatic Numbers of a Graph, *Discrete Applied Mathematics*, vol. 5, No. 1, pp. 51-64.

Eiter, T. y Veith, H. (2002): On the Complexity of Data Disjunctions, *Theoretical Computer Science*, vol. 288, No. 1, pp. 101-128.

Ekroot, L. y Dolinar, S. (1996): A* Decoding of Block Codes, *IEEE Transactions on Communications*, pp. 1052-1056.

ElGindy, H., Everett, H. y Toussaint, G. (1993): Slicing an Ear Using Prune-and-Search, *Pattern Recognition Letters*, vol. 14, pp. 719-722.

El-Yaniv, R. (1998): Competitive Solutions for Online Financial Problems, *ACM Computing Surveys*, vol. 30, No. 1, pp. 28-69.

El-Zahar, M. H. y Rival, I. (1985): Greedy Linear Extensions to Minimize Jumps, *Discrete Applied Mathematics*, vol. 11, No. 2, pp. 143-156.

Eppstein, D., Galil, Z., Giancarlo, R. y Italiano, G. F. (1992a): Sparse Dynamic Programming I: Linear Cost Functions, *Journal of the ACM*, vol. 39, No. 3, pp. 519-545.

Eppstein, D., Galil, Z., Giancarlo, R. y Italiano, G. F. (1992b): Sparse Dynamic Programming II: Convex and Concave Cost Functions, *Journal of the ACM*, vol. 39, pp. 516-567.

Eppstein, D., Galil, Z., Italiano, G. F. y Spencer, T. H. (1996): Separator Based Sparsication I. Planarity Testing and Minimum Spanning Trees, *Journal of Computer and System Sciences*, vol. 52, No. 1, pp. 3-27.

Epstein, L. y Ganot, A. (2003): Optimal On-line Algorithms to Minimize Makespan on Two Machines with Resource Augmentation, *Lecture Notes in Computer Science*, vol. 2909, pp. 109-122.

Epstein, L., Noga, J., Seiden, S., Sgall, J. y Woeginger, G. (1999): Randomized Online Scheduling on Two Uniform Machines, *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, Baltimore, Maryland, pp. 317-326.

Erdmann, M. (1993): Randomization for Robot Tasks: Using Dynamic Programming in the Space of Knowledge States, *Algorithmica*, vol. 10, pp. 248-291.

Erlebach, T. y Jansen, K. (1999): Efficient Implementation of an Optimal Greedy Algorithm for Wavelength Assignment in Directed Tree Networks, *ACM Journal of Experimental Algorithmics*, vol. 4.

Esko, U. (1990): A Linear Time Algorithm for Finding Approximate Shortest Common Superstrings, *Algorithmica*, vol. 5, pp. 313-323.

Evans, J. R. y Minieka, E. (1992): *Optimization Algorithms for Networks and Graphs*, 2nd ed., Marcel Dekker, Nueva York.

Even, G., Naor, J., Rao, S. y Schieber, B. (2000): Divide-and-Conquer Approximation Algorithms via Spreading Metrics, *Journal of the ACM*, vol. 47, No. 4, pp. 585-616.

Even, G., Naor, J. y Zosin, L. (2000): An 8-Approximation Algorithm for the Subset Feedback Vertex Set Problem, *SIAM Journal on Computing*, vol. 30, pp. 1231-1252.

Even, S. (1987): *Graph Algorithms*, Computer Science Press, Rockville, Maryland.

Even, S., Itai, A. y Shamir, A. (1976): On the Complexity of Timetable and Multicommodity Problems, *SIAM Journal on Computing*, vol. 5, pp. 691-703.

Even, S. y Shiloach, Y. (1981): An On-Line Edge-Deletion Problem, *Journal of the ACM*, vol. 28, No. 1, pp. 1-4.

Fagin, R. (1974): Generalized First-Order Spectra and Polynomial-Time Recognizable Sets, *SIAM-AMS Proceedings*, vol. 7, pp. 43-73.

Faigle, U. (1985): On Ordered Languages and the Optimization of Linear Functions by Greedy Algorithms, *Journal of the ACM*, vol. 32, No. 4, pp. 861-870.

Faigle, U., Kern, W. y Nawijn, W. (1999): A Greedy On-Line Algorithm for the *k*-Track Assignment Problem, *Journal of Algorithms*, vol. 31, pp. 196-210.

Farach, M. y Thorup, M. (1997): Sparse Dynamic Programming for Evolutionary Tree Comparison, *SIAM Journal on Computing*, vol. 26, pp. 210-230.

Farber, M. y Keil, J. M. (1985): Domination in Permutation Graphs, *Journal of Algorithms*, vol. 6, pp. 309-321.

Feder, T. y Motwani, R. (2002): Worst-Case Time Bounds for Coloring and Satisfiability Problems, *Journal of Algorithms*, vol. 45, pp. 192-201.

Feige, U. y Krauthgamer, R. (2002): A Polylogarithmic Approximation of the Minimum Bisection, *SIAM Journal on Computing*, vol. 31, No. 4, pp. 1090-1118.

Fejes, G. (1978): Evading Convex Discs, *Studia Science Mathematics Hungar*, vol. 13, pp. 453-461.

Feldmann, A., Kao, M., Sgall, J. y Teng, S. H. (1993): Optimal Online Scheduling of Parallel Jobs with Dependencies, *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, ACM Press, San Diego, California, pp. 642-651.

Fellows, M. R. y Langston, M. A. (1988): Processor Utilization in a Linearly Connected Parallel Processing System, *IEEE Transactions on Computers*, vol. 37, pp. 594-603.

Fernandez-Beca, D. y Williams, M. A. (1991): On Matroids and Hierarchical Graphs, *Information Processing Letters*, vol. 38, No. 3, pp. 117-121.

Ferragina, P. (1997): Dynamic Text Indexing under String Updates, *Journal of Algorithms*, vol. 22, No. 2, pp. 296-238.

Ferreira, C. E., de Souza, C. C. y Wakabayashi, Y. (2002): Rearrangement of DNA Fragments: A Branch-and-Cut Algorithm (Prove NP-Complete Problem), *Discrete Applied Mathematics*, vol. 116, pp. 161-177.

Fiat, A. y Woeginger, G. J. (1998): Online Algorithms: The State of the Art, *Lecture Notes in Computer Science*, vol. 1442.

Fischel-Ghodsian, F., Mathiowitz, G. y Smith, T. F. (1990): Alignment of Protein Sequence Using Secondary Structure: A Modified Dynamic Programming Method, *Protein Engineering*, vol. 3, No. 7, pp. 577-581.

Flajolet, P. y Prodinger, H. (1986): Register Allocation for Unary-Binary Trees, *SIAM Journal on Computing*, vol. 15, No. 3, pp. 629-640.

Floyd, R. W. (1962): Algorithm 97: Shortest Path, *Communications of the ACM*, vol. 5, No. 6, p. 345.

Floyd, R. W. y Rivest, R. L. (1975): Algorithm 489 (SELECT), *Communications of the ACM*, vol. 18, No. 3, p. 173.

Fonlupt, J. y Nachef, A. (1993): Dynamic Programming and the Graphical Traveling Salesman Problem, *Journal of the Association for Computing Machinery*, vol. 40, No. 5, pp. 1165-1187.

Foulds, L. R. y Graham, R. L. (1982): The Steiner Problem in Phylogeny is NP-Complete, *Advances Application Mathematics*, vol. 3, pp. 43-49.

Franco, J. (1984): Probabilistic Analysis of the Pure Literal Heuristic for the Satisfiability Problem, *Annals of Operations Research*, vol. 1, pp. 273-289.

Frederickson, G. N. (1984): Recursively Rotated Orders and Implicit Data Structures: A Lower Bound, *Theoretical Computer Science*, vol. 29, pp. 75-85.

Fredman, M. L. (1981): A Lower Bound on the Complexity of Orthogonal Range Queries, *Journal of the ACM*, vol. 28, No. 4, pp. 696-705.

Fredman, M. L., Sedgewick, R., Sleator, D. D. y Tarjan, R. E. (1986): The Pairing Heap: A New Form of Self-Adjusting Heap, *Algorithmica*, vol. 1, No. 1, pp. 111-129.

Friesen, D. K. y Kuhl, F. S. (1988): Analysis of a Hybrid Algorithm for Packing Unequal Bins, *SIAM Journal on Computing*, vol. 17, No. 1, pp. 23-40.

Friesen, D. K. y Langston, M. A. (1986): Variable Sized Bin Packing, *SIAM Journal on Computing*, vol. 15, No. 1, pp. 222-230.

Frieze, A. M. y Kannan, R. (1991): A Random Polynomial-Time Algorithm for Approximating the Volume of Convex Bodies, *Journal of the ACM*, vol. 38, No. 1.

Frieze, A. M., McDiarmid, C. y Reed, B. (1990): Greedy Matching on the Line, *SIAM Journal on Computing*, vol. 19, No. 4, pp. 666-672.

Froda, S. (2000): On Assessing the Performance of Randomized Algorithms, *Journal of Algorithms*, vol. 31, pp. 344-362.

Fu, H. C. (2001): Divide-and-Conquer Learning and Modular Perceptron Networks, *IEEE Transactions on Neural Networks*, vol. 12, No. 2, pp. 250-263.

Fu, J. J. y Lee, R. C. T. (1991): Minimum Spanning Trees of Moving Points in the Plane, *IEEE Transactions on Computers*, vol. 40, No. 1, pp. 113-118.

Galbiati, G., Maffioli, F. y Morrzenti, A. (1994): A Short Note on the Approximability of the Maximum Leaves Spanning Tree Problem, *Information Processing Letters*, vol. 52, pp. 45-49.

Galil, Z. y Giancarlo, R. (1989): Speeding up Dynamic Programming with Applications to Molecular Biology, *Theoretical Computer Science*, vol. 64, pp. 107-118.

Galil, Z., Haber, S. y Yung, M. (1989): Minimum-Knowledge Interactive Proofs for Decision Problems, *SIAM Journal on Computing*, vol. 18, No. 4, pp. 711-739.

Galil, Z., Hoffman, C. H., Luks, E. M., Schnorr, C. P. y Weber, A. (1987): An $O(n^3 \log n)$ Deterministic and an $O(n^3)$ Las Vagas Isomorphism Test for Trivalent Graphs, *Journal of the ACM*, vol. 34, No. 3, pp. 513-531.

Galil, Z. y Park, K. (1990): An Improved Algorithm for Approximate String Matching, *SIAM Journal on Computing*, vol. 19, pp. 989-999.

Galil, Z. y Seiferas, J. (1978): A Linear-Time On-Line Recognition Algorithm for "Palstar", *Journal of the ACM*, vol. 25, No. 1, pp. 102-111.

Galil, Z. y Park, K. (1992): Dynamic Programming with Convexity Concavity and Sparsity, *Theoretical Computer Science*, vol. 49-76.

Gallant, J., Marier, D. y Storer, J. A. (1980): On Finding Minimal Length Superstrings (Prove NP-Hard Problem), *Journal of Computer and System Sciences*, vol. 20, pp. 50-58.

Garay, J., Gopal, I., Kutten, S., Mansour, Y. y Yung, M. (1997): Efficient On-Line Call Control Algorithms, *Journal of Algorithms*, vol. 23, pp. 180-194.

Garey, M. R. y Johnson, D. S. (1976): Approximation algorithms for combinatorial problem: An annotated bibliography. En J. F. Traub (Ed.), *Algorithms and Complexity: New Directions and Recent Results*, Academic Press, Nueva York, pp. 41-52.

Garey, M. R. y Johnson, D. S. (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, California.

Geiger, D., Gupta, A., Costa, L. A. y Vlontzos, J. (1995): Dynamic Programming for Detecting Tracking and Matching Deformable Contours, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, pp. 294-302.

Gelfand, M. S. y Roytberg, M. A. (1993): Prediction of the Exon-Intron Structure by a Dynamic Programming Approach, *BioSystems*, vol. 30, pp. 173-182.

Gentleman, W. M. y Sande, G. (1966): Fast Fourier Transforms for Fun and Prot, *Proceedings of AFIPS Fall Joint Computer Conference*, vol. 29, pp. 563-578.

Giancarlo, R. y Grossi, R. (1997): Multi-Dimensional Pattern Matching with Dimensional Wildcards: Data Structures and Optimal On-Line Search Algorithms, *Journal of Algorithms*, vol. 24, pp. 223-265.

Gilbert, E. N. y Moore, E. F. (1959): Variable Length Encodings, *Bell System Technical Journal*, vol. 38, No. 4, pp. 933-968.

Gilbert, J. R., Hutchinson, J. P. y Tarjan, R. E. (1984): A Separator Theorem for Graphs of Bounded Genus, *Journal of Algorithms*, vol. 5, pp. 391-407.

Gill, I. (1987): Computational Complexity of Probabilistic Turing Machine, *SIAM Journal on Computing*, vol. 16, No. 5, pp. 852-853.

Godbole, S. S. (1973): On Efficient Computation of Matrix Chain Products, *IEEE Transactions on Computers*, vol. 22, No. 9, pp. 864-866.

Goemans, M. X. y Williamson, D. P. (1995): Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming, *Journal of the ACM*, vol. 42, No. 6, pp. 1115-1145.

Goldberg, L. A., Goldberg, P. W. y Paterson, M. (2001): The Complexity of Gene Placement (Prove NP-Complete Problem), *Journal of Algorithms*, vol. 41, pp. 225-243.

Goldman, S., Parwatikar, J. y Suri, S. (2000): Online Scheduling with Hard Deadlines, *Journal of Algorithms*, vol. 34, pp. 370-389.

Goldstein, L. y Waterman, M. S. (1987): Mapping DNA by Stochastic Relaxation (Prove NP-Complete Problem of Double Digest Problem), *Advances in Applied Mathematics*, vol. 8, pp. 194-207.

Goldwasser, S. y Micali, S. (1984): Probabilistic Encryption, *Journal of Computer and System Sciences*, vol. 28, No. 2, pp. 270-298.

Goldwasser, S., Micali, S. y Rackoff, C. (1988): The Knowledge Complexity of Interactive Proof Systems, *SIAM Journal on Computing*, vol. 18, No. 1, pp. 186-208.

Golumbic, M. C. (1980): *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, Nueva York.

Gonnet, G. H. (1983): *Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, Mass.

Gonzalez, T. F. y Lee, S. L. (1987): A 1.6 Approximation Algorithm for Routing Multiterminal Nets, *SIAM Journal on Computing*, vol. 16, No. 4, pp. 669-704.

Gonzalo, N. (2001): A Guide Tour to Approximate String Matching, *ACM*, vol. 33, pp. 31-88.

Goodman, S. y Hedetniemi, S. (1980): *Introduction to the Design and Analysis of Algorithms*, McGraw-Hill, Nueva York.

Gorodkin, J., Lyngso, R. B. y Stormo, G. D. (2001): A Mini-Greedy Algorithm for Faster Structural RNA Stem-Loop Search, *Genome Informatics*, vol. 12, pp. 184-193.

Gotieb, L. (1981): Optimal Multi-Way Search Trees, *SIAM Journal on Computing*, vol. 10, No. 3, pp. 422-433.

Gotieb, L. y Wood, D. (1981): The Construction of Optimal Multiway Search Trees and the Monotonicity Principle, *International Journal of Computer Mathematics*, vol. 9, pp. 17-24.

Gould, R. (1988): *Graph Theory*, Benjamin Cummings, Redwood City, California.

Graham, R. L. (1972): An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set, *Information Processing Letters*, vol. 1, pp. 132-133.

Grandjean, E. (1988): A Natural NP-Complete Problem with a Nontrivial Lower Bound, *SIAM Journal on Computing*, vol. 17, No. 4, pp. 786-809.

Greene, D. H. y Knuth, D. E. (1981): *Mathematics for the Analysis of Algorithms*, Birkhauser, Boston, Mass.

Grigni, M., Koutsoupias, E. y Papadimitriou, C. (1995): Approximation Scheme for Planar Graph TSP, *Foundations of Computer Science*, pp. 640-645.

Grove, E. (1995): Online Bin Packing with Lookahead, *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, San Francisco, California, pp. 430-436.

Gudmundsson, J., Levcopoulos, C. y Narasimhan, G. (2002): Fast Greedy Algorithms for Constructing Sparse Geometric Spanners, *SIAM Journal on Computing*, vol. 31, pp. 1479-1500.

Gueting, R. H. (1984): Optimal Divide-and-Conquer to Compute Measure and Contour for a Set of Iso-Rectangles, *Acta Informatica*, vol. 21, pp. 271-291.

Gueting, R. H. y Schilling, W. (1987): Practical Divide-and-Conquer Algorithm for the Rectangle Intersection Problem, *Information Sciences*, vol. 42, No. 2, pp. 95-112.

Gupta, S., Konjevod, G. y Varsamopoulos, G. (2002): A Theoretical Study of Optimization Techniques Used in Registration Area Based Location Management: Models and Online Algorithms, *Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, ACM Press, Atlanta, Georgia, pp. 72-79.

Gusfield, D. (1994): Faster Implementation of a Shortest Superstring Approximation, *Information Processing Letters*, vol. 51, pp. 271-274.

Gusfield, D. (1997): *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, Cambridge, Inglaterra.

Guting, R. H. (1984): Optimal Divide-and-Conquer to Compute Measure and Contour for a Set of Iso-Rectangles, *Acta Informatica*, vol. 21, pp. 271-291.

Haas, P. y Hellerstein, J. (1999): Ripple Joins for Online Aggregation, ACM SIGMOD Record, *Proceedings of the 1999 ACM-SIGMOD International Conference on Management of Data*, vol. 28.

Hall, N. G. y Hochbaum, D. S. (1986): A Fast Approximation Algorithm for the Multicovering Problem, *Discrete Applied Mathematics*, vol. 15, No. 1, pp. 35-40.

Halldorsson, M. (1997): Parallel and On-Line Graph Coloring, *Journal of Algorithms*, vol. 23, pp. 265-280.

Halperin, D., Sharir, M. y Goldberg, K. (2002): The 2-Center Problem with Obstacles, *Journal of Algorithms*, vol. 42, pp. 109-134.

Han, Y. S., Hartmann, C. R. P. y Chen, C. C. (1993): Efficient Priority: First Search Maximum-Likelihood Soft-Decision Decoding of Linear Block Codes, *IEEE Transactions on Information Theory*, pp. 1514-1523.

Han, Y. S., Hartmann C. R. P. y Mehrotra, K. G. (1998): Decoding Linear Block Codes Using a Priority-First Search: Performance Analysis and Suboptimal Version, *IEEE Transactions on Information Theory*, pp. 1233-1246.

Hanson, F. B. (1991): Computational Dynamic Programming on a Vector Multiprocessor, *IEEE Transactions on Automatic Control*, vol. 36, pp. 507-511.

Hariri, A. M. A. y Potts, C. N. (1983): An Algorithm for Single Machine Sequencing with Release Dates to Minimize Total Weighted Completion Time, *Discrete Applied Mathematics*, vol. 5, No. 1, pp. 99-109.

Har-Peled, S. (2000): Constructing Planar Cuttings in Theory and Practice, *SIAM Journal on Computing*, vol. 29, No. 6, pp. 2016-2039.

Hasegawa, M. y Horai, S. (1991): Time of the Deepest Root for Polymorphism in Human Mitochondrial DNA, *Journal of Molecular Evolution*, vol. 32, pp. 37-42.

Hasham, A. y Sack, J. R. (1987): Bounds for Min-Max Heaps, *BIT*, vol. 27, pp. 315-323.

Hashimoto, R. F. y Barrera J. (2003): A Greedy Algorithm for Decomposing Convex Structuring Elements, *Journal of Mathematical Imaging and Vision*, vol. 18, No. 3, pp. 269-286.

Haussmann, U. G. y Suo, W. (1995): Singular Optimal Stochastic Controls. II. Dynamic Programming, *SIAM Journal on Control and Optimization*, vol. 33, pp. 937-959.

Hayward, R. B. (1987): A Lower Bound for the Optimal Crossing-Free Hamiltonian Cycle Problem, *Discrete and Computational Geometry*, vol. 2, pp. 327-343.

Hein, J. (1989): An Optimal Algorithm to Reconstruct Trees from Additive Distance Data, *Bulletin of Mathematical Biology*, vol. 51, pp. 597-603.

Held, M. y Karp, R. M. (1962): A Dynamic Programming Approach to Sequencing Problems, *SIAM Journal on Applied Mathematics*, vol. 10, pp. 196-210.

Hell, P., Shamir, R. y Sharan, R. (2001): A Fully Dynamic Algorithm for Recognizing and Representing Proper Interval Graphs, *SIAM Journal on Computing*, vol. 31, pp. 289-305.

Henzinger, M. R. (1995): Fully Dynamic Biconnectivity in Graphs, *Algorithmica*, vol. 13, No. 6, pp. 503-538.

Hirosawa, M., Hoshida, M., Ishikawa, M. y Toya, T. (1993): MASCOT: Multiple Alignment System for Protein Sequence Based on Three-Way Dynamic Programming, *Computer Applications in the Biosciences*, vol. 9, pp. 161-167.

Hirschberg, D. S. (1975): A Linear Space Algorithm for Computing Maximal Common Subsequences, *Communications of the ACM*, vol. 18, No. 6, pp. 341-343.

Hirschberg, D. S. y Larmore, L. L. (1987): The Least Weight Subsequence Problem, *SIAM Journal on Computing*, vol. 16, No. 4, pp. 628-638.

Hoang, T. M. y Thierauf, T. (2003): The Complexity of the Characteristic and the Minimal Polynomial, *Theoretical Computer Science*, vol. 1-3, pp. 205-222.

Hoare, C. A. R. (1961): Partition (Algorithm 63), Quicksort (Algorithm 64) and Find (Algorithm 65), *Communications of the ACM*, vol. 4, No. 7, pp. 321-322.

Hoare, C. A. R. (1962): Quicksort, *Computer Journal*, vol. 5, No. 1, pp. 10-15.

Hochbaum, D. S. (1997): *Approximation Algorithms for NP-Hard Problems*, PWS Publisher, Boston.

Hochbaum, D. S. y Maass, W. (1987): Fast Approximation Algorithms for a Nonconvex Covering Problem, *Journal of Algorithms*, vol. 8, No. 3, pp. 305-323.

Hochbaum, D. S. y Shmoys, D. B. (1986): A United Approach to Approximation Algorithms for Bottleneck Problems, *Journal of the ACM*, vol. 33, No. 3, pp. 533-550.

Hochbaum, D. S. y Shmoys, D. B. (1987): Using Dual Approximation Algorithms for Scheduling Problems: Theoretical and Practical Results, *Journal of the ACM*, vol. 34, No. 1, pp. 144-162.

Hoffman, A. J. (1988): On Greedy Algorithms for Series Parallel Graphs, *Mathematical Programming*, vol. 40, No. 2, pp. 197-204.

Hofri, M. (1987): *Probabilistic Analysis of Algorithms*, Springer-Verlag, Nueva York.

Holmes, I. y Durbin, R. (1998): Dynamic Programming Alignment Accuracy, *Journal of Computational Biology*, vol. 5, pp. 493-504.

Holyer, I. (1981): The NP-Completeness of Some Edge-Partition Problems, *SIAM Journal on Computing*, vol. 10, pp. 713-717.

Homer, S. (1986): On Simple and Creative Sets in NP, *Theoretical Computer Science*, vol. 47, No. 2, pp. 169-180.

Homer, S. y Long, T. J. (1987): Honest Polynomial Degrees and P =? NP, *Theoretical Computer Science*, vol. 51, No. 3, pp. 265-280.

Horowitz, E. y Sahni, S. (1974): Computing Partitions with Applications to the Knapsack Problem, *Journal of the ACM*, vol. 21, No. 2, pp. 277-292.

Horowitz, E. y Sahni, S. (1976a): Exact and Approximate Algorithms for Scheduling Nonidentical Processors, *Journal of the ACM*, vol. 23, No. 2, pp. 317-327.

Horowitz, E. y Sahni, S. (1976b): *Fundamentals of Data Structures*, Computer Science Press, Rockville, Maryland.

Horowitz, E. y Sahni, S. (1978): *Fundamentals of Computer Algorithm*, Computer Science Press, Rockville, Maryland.

Horowitz, E., Sahni, S. y Rajasekaran, S. (1998): *Computer Algorithms*, W. H. Freeman, Nueva York.

Horton, J. D. (1987): A Polynomial-Time Algorithm to Find the Shortest Cycle Basis of a Graph, *SIAM Journal on Computing*, vol. 16, No. 2, pp. 358-366.

Horvath, E. C., Lam, S. y Sethi, R. (1977): A Level Algorithm for Preemptive Scheduling, *Journal of the ACM*, vol. 24, No. 1, pp. 32-43.

Hsiao, J. Y., Tang, C. Y. y Chang, R. S. (1993): The Summation and Bottleneck Minimization for Single Step Searching on Weighted Graphs, *Information Sciences*, vol. 74, pp. 1-28.

Hsu, W. L. (1984): Approximation Algorithms for the Assembly Line Crew Scheduling Problem, *Mathematics of Operations Research*, vol. 9, pp. 376-383.

Hsu, W. L. y Nemhauser, G. L. (1979): Easy and Hard Bottleneck Location Problems, *Discrete Applied Mathematics*, vol. 1, No. 3, pp. 209-215.

Hu, T. C. y Shing, M. T. (1982): Computation of Matrix Chain Products Part I, *SIAM Journal on Computing*, vol. 11, No. 2, pp. 362-373.

Hu, T. C. y Shing, M. T. (1984): Computation of Matrix Chain Products Part II, *SIAM Journal of Computing*, vol. 13, No. 2, pp. 228-251.

Hu, T. C. y Tucker, A. C. (1971): Optimal Computer Search Trees and Variable-Length Alphabetical Codes, *SIAM Journal on Applied Mathematics*, vol. 21, No. 4, pp. 514-532.

Hu, T. H., Tang, C. Y. y Lee, R. C. T. (1992): An Average Analysis of a Resolution Principle Algorithm in Mechanical Theorem Proving, *Annals of Mathematics and Artificial Intelligence*, vol. 6, pp. 235-252.

Huang, S. H. S., Liu, H. y Viswanathan, V. (1994): Parallel Dynamic Programming, *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, No. 3, pp. 326-328.

Huang, X. y Waterman, M. S. (1992): Dynamic Programming Algorithms for Restriction Map Comparison, *Computational Applied Biology Science*, pp. 511-520.

Huddleston, S. y Mehlhorn, K. (1982): A New Data Structure for Representing Sorted Lists, *Acta Informatica*, vol. 17, pp. 157-184.

Huffman, D. A. (1952): A Method for the Construction of Minimum-Redundancy Codes, *Proceedings of IRE*, vol. 40, pp. 1098-1101.

Huo, D. y Chang, G. J. (1994): The Provided Minimization Problem in Tree, *SIAM Journal of Computing*, vol. 23, No. 1, pp. 71-81.

Huyn, N., Dechter, R. y Pearl, J. (1980): Probabilistic Analysis of the Complexity of A*, *Artificial Intelligence*, vol. 15, pp. 241-254.

Huynh, D. T. (1986): Some Observations about the Randomness of Hard Problems, *SIAM Journal on Computing*, vol. 15, No. 4, pp. 1101-1105.

Hyafil, L. (1976): Bounds for Selection, *SIAM Journal on Computing*, vol. 5, No. 1, pp. 109-114.

Ibaraki, T. (1977): The Power of Dominance Relations in Branch-and-Bound Algorithms, *Journal of the ACM*, vol. 24, No. 2, pp. 264-279.

Ibaraki, T. y Nakamura, Y. (1994): A Dynamic Programming Method for Single Machine Scheduling, *European Journal of Operational Research*, vol. 76, p. 72.

Ibarra, O. H. y Kim, C. E. (1975): Fast Approximation Algorithms for the Knapsack and the Sum of Subset Problems, *Journal of the ACM*, vol. 22, pp. 463-468.

Imai, H. (1993): Geometric Algorithms for Linear Programming, *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, vol. E76-A, No. 3, pp. 259-264.

Imai, H., Kato, K. y Yamamoto, P. (1989): A Linear-Time Algorithm for Linear L1 Approximation of Points, *Algorithmica*, vol. 4, No. 1, pp. 77-96.

Imai, H., Lee, D. T. y Yang, C. D. (1992): 1-Segment Center Problem, *ORSA Journal on Computing*, vol. 4, No. 4, pp. 426-434.

Imase, M. y Waxman, B. M. (1991): Dynamic Steiner Tree Problem, *SIAM Journal on Discrete Mathematics*, vol. 4, No. 3, pp. 369-384.

Irani, S., Shukla, S. y Gupta, R. (2003): Online Strategies for Dynamic Power Management in Systems with Multiple Power-Saving States, *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 2, pp. 325-346.

Italiano, G. F. (1986): Amortized Efficiency of a Path Retrieval Data Structure, *Theoretical Computer Science*, vol. 48, No. 2, pp. 273-281.

Ivanov, A. G. (1984): Distinguishing an Approximate Word's Inclusion on Turing Machine in Real Time, *Izvestiia Academii Nauk USSSR, Series Math*, vol. 48, pp. 520-568.

Iwamura, K. (1993): Discrete Decision Process Model Involves Greedy Algorithm Over Greedoid, *Journal of Information and Optimization Sciences*, vol. 14, pp. 83-86.

Jadhavm, S. y Mukhopadhyay, A. (1993): Computing a Centerpoint of a Finite Planar Set of Points in Linear Time, *Proceedings of the 9th Annual Symposium on Computational Geometry*, ACM Press, San Diego, California, pp. 83-90.

Jain, K. y Vazirani, V. V. (2001): Approximation Algorithms for Metric Facility Location and *k*-Median Problems Using the Primal-Dual Schema and Lagrangian Relaxation, *Journal of the ACM*, vol. 48, No. 2.

Janssen, J., Krizanc, D., Narayanan, L. y Shende, S. (2000): Distributed Online Frequency Assignment in Cellular Networks, *Journal of Algorithms*, vol. 36, pp. 119-151.

Jayram, T., Kimbrel, T., Krauthgamer, R., Schieber, B. y Sviridenko, M. (2001): Online Server Allocation in a Server Farm via Benefit Task Systems, *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, ACM Press, Hersonissos, Grecia, pp. 540-549.

Jerrum, M. (1985): The Complexity of Finding Minimum-Length Generator Sequences, *Theoretical Computer Science*, vol. 36, pp. 265-289.

Jiang, T., Kearney, P. y Li, M. (1998): Orchestrating Quartets: Approximation and Data Correction, *Proceedings of the 39th IEEE Symposium on Foundations of Computer Science*, IEEE Press, Palo Alto, California, pp. 416-425.

Jiang, T., Kearney, P. y Li, M. (2001): A Polynomial Time Approximation Scheme For Inferring Evolutionary Trees from Quartet Topologies and Its Application, *SIAM Journal on Computing*, vol. 30, No. 6, pp. 1942-1961.

Jiang, T., Lawler, E. L. y Wang, L. (1994): Aligning Sequences via an Evolutionary Tree: Complexity and Approximation, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, ACM Press, Montreal, Quebec, pp. 760-769.

Jiang, T. y Li, M. (1995): On the Approximation of Shortest Common Supersequences and Longest Common Subsequences, *SIAM Journal on Computing*, vol. 24, No. 5, pp. 1122-1139.

Jiang, T., Wang, L. y Lawler, E. L. (1996): Approximation Algorithms for Tree Alignment with a Given Phylogeny, *Algorithmica*, vol. 16, pp. 302-315.

John, J. W. (1988): A New Lower Bound for the Set-Partitioning Problem, *SIAM Journal on Computing*, vol. 17, No. 4, pp. 640-647.

Johnson, D. S. (1973): Near-Optimal Bin Packing Algorithms, *MIT Report MAC TR-109*.

Johnson, D. S. (1974): Approximation Algorithms for Combinatorial Problems, *Journal of Computer and System Sciences*, vol. 9, pp. 256-278.

Johnson, D. S., Demars, A., Ullman, J. D., Garey, M. R. y Graham, R. L. (1974): Worst-Case Performance Bounds for Simple One-Dimensional Packing Algorithms, *SIAM Journal on Computing*, vol. 3, No. 4, pp. 299-325.

Johnson, D. S., Yanakakis, M. y Papadimitriou, C. H. (1988): On Generating All Maximal Independent Sets, *Information Processing Letters*, vol. 27, No. 3, pp. 119-123.

Johnson, H. W. y Burrus, C. S. (1983): The Design of Optimal DFT Algorithms Using Dynamic Programming, *IEEE Transactions on Acoustics Speech and Signal Processing*, vol. 31, No. 2, pp. 378-387.

Jonathan, T. (1989): Approximation Algorithms for the Shortest Common Superstring Problem, *Information and Computation*, vol. 83, pp. 1-20.

Jorma, T. y Ukkonen, E. (1988): A Greedy Approximation Algorithm for Constructing Shortest Common Superstrings, *Theoretical Computer Science*, vol. 57, pp. 131-145.

Juedes, D. W. y Lutz, J. H. (1995): The Complexity and Distribution of Hard Problems, *SIAM Journal of Computing*, vol. 24, No. 2, pp. 279-295.

Kalyanasundaram, B. y Pruhs, K. (1993): On-Line Weighted Matching, *Journal of Algorithms*, vol. 14, pp. 478-488.

Kamidoi, Y., Wakabayashi, S. y Yoshida, N. (2002): A Divide-and-Conquer Approach to the Minimum *k*-Way Cut Problem, *Algorithmica*, vol. 32, pp. 262-276.

Kannan, R., Mount, J. y Tayur, S. (1995): A Randomized Algorithm to Optimize Over Certain Convex Sets, *Mathematical Operating Research*, vol. 20, No. 3, pp. 529-549.

Kannan, S., Lawler, E. L. y Warnow, T. J. (1996): Determining the Evolutionary Tree Using Experiments, *Journal of Algorithms*, vol. 21, pp. 26-50.

Kannan, S. y Warnow, T. (1994): Inferring Evolutionary History from DNA Sequences, *SIAM Journal on Computing*, vol. 23, pp. 713-737.

Kannan, S. y Warnow, T. (1995): Tree Reconstruction from Partial Orders, *SIAM Journal on Computing*, vol. 24, pp. 511-519.

Kantabutra, V. (1994): Linear-Time Near-Optimum-Length Triangulation Algorithm for Convex Polygons, *Journal of Computer and System Sciences*, vol. 49, No. 2, pp. 325-333.

Kao, E. P. C. y Queyranne, M. (1982): On Dynamic Programming Methods for Assembly Line Balancing, *Operations Research*, vol. 30, No. 2, pp. 375-390.

Kao, M. Y., Ma, Y., Sipser, M. y Yin, Y. (1998): Optimal Constructions of Hybrid Algorithms, *Journal of Algorithms*, vol. 29, pp. 142-164.

Kao, M. Y. y Tate, S. R. (1991): Online Matching with Blocked Input, *Information Processing Letters*, vol. 38, pp. 113-116.

Kaplan, H. y Shamir, R. (1994): On the Complexity of DNA Physical Mapping, *Advances in Applied Mathematics*, vol. 15, pp. 251-261.

Karger, D. R., Klein, P. N. y Tarjan, R. E. (1995): Randomized Linear-Time Algorithm to Find Minimum Spanning Trees, *Journal of the Association for Computing Machinery*, vol. 42, No. 2, pp. 321-328.

Karger, D. R., Phillips, S. y Torng, E. (1994): A Better Algorithm for an Ancient Scheduling Problem, *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, Arlington, Virginia, pp. 132-140.

Karger, D. R. y Stein, C. (1996): New Approach to the Minimum Cut Problem, *Journal of the Association for Computing Machinery*, vol. 43, No. 4, pp. 601-640.

Karkkainen, J., Navarro, G. y Ukkonen, E. (2000): Approximate String Matching over Ziv-Lempel Compressed Text, *Lecture Notes in Computer Science*, vol. 1848, pp. 195-209.

Karlin, A. R., Manasse, M. S., Rudolph, L. y Sleator, D. D. (1988): Competitive Snoopy Caching, *Algorithmica*, vol. 3, No. 1, pp. 79-119.

Karoui, N. E. y Quenez, M. C. (1995): Dynamic Programming and Pricing of Contingent Claims in an Incomplete Market, *SIAM Journal on Control and Optimization*, 1995, pp. 27-66.

Karp, R. M. (1972): Reducibility among Combinatorial Problems, in R. Miller y J. Thatcher (Eds.), *Complexity of Computer Computations* (pp. 85-103), Plenum Press, Nueva York.

Karp, R. M. (1986): Combinatorics; Complexity and Randomness, *Communications of the ACM*, vol. 29, No. 2, 1986, pp. 98-109.

Karp, R. M. (1994): Probabilistic Recurrence Relations, *Journal of the Association for Computing Machinery*, vol. 41, No. 6, 1994, pp. 1136-1150.

Karp, R. M., Montwani, R. y Raghavan, P. (1988): Deferred Data Structuring, *SIAM Journal on Computing*, vol. 17, No. 5, 1988, pp. 883-902.

Karp, R. M. y Pearl, J. (1983): Searching for an Optimal Path in a Tree with Random Costs, *Artificial Intelligence*, vol. 21, 1983, pp. 99-116.

Karp, R. M. y Rabin, M. O. (1987): Efficient Randomized Pattern-Matching Algorithms, *IBM Journal of Research and Development*, vol. 31, 1987, pp. 249-260.

Karp, R. M., Vazirani, U. V. y Vazirani, V. V. (1990): An Optimal Algorithm for On-Line Bipartite Matching, *Proceedings of the 22nd ACM Symposium on Theory of Computing*, ACM Press, Baltimore, Maryland, pp. 352-358.

Kearney, P., Hayward, R. B. y Meijer, H. (1997): Inferring Evolutionary Trees from Ordinal Data, *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, Nueva Orleans, Louisiana, pp. 418-426.

Kececioglu, J. D. (1991): *Exact and approximate algorithms for sequence recognition problems in molecular biology*. Tesis doctoral, aún sin publicar, University of Arizona.

Kececioglu, J. D. y Myers, W. E. (1995a): Exact and Approximation Algorithms for the Sequence Reconstruction Problem, *Algorithmica*, vol. 13, pp. 7-51.

Kececioglu, J. D. y Myers, W. E. (1995b): Combinatorial Algorithms for DNA Sequence Assembly, *Algorithmica*, vol. 13, pp. 7-51.

Kececioglu, J. D. y Sankoff, D. (1993): Exact and Approximation Algorithms for the Inversion Distance between Two Chromosomes, *Lecture Notes in Computer Science*, vol. 684, pp. 87-105.

Kececioglu, J. D. y Sankoff, D. (1995): Exact and Approximate Algorithms for Sorting by Reversals with Application to Genome Rearrangement, *Algorithmica*, vol. 13, pp. 180-210.

Keogh, E., Chu, S., Hart, D. y Pazzani, M. (2001): An Online Algorithm for Segmenting Time Series, *IEEE Computer Science Press*, pp. 289-296.

Khachian, L. G. (1979): A Polynomial Algorithm for Linear Programming, *Doklady Akademii Nauk*, USSR, vol. 244, No. 5, pp. 1093-1096. Traducido en *Soviet Math. Doklady*, vol. 20, pp. 191-194.

Khuller, S., Mitchell, S. y Vazirani, V. V. (1994): On-Line Algorithms for Weighted Bipartite Matching and Stable Marriage, *Theoretical Computer Science*, vol. 127, No. 2, pp. 255-267.

Kilpelainen P. y Mannila H. (1995): Ordered and Unordered Tree Inclusion, *SIAM Journal on Computing*, vol. 24, No. 2, pp. 340-356.

Kim, D. S., Yoo, K. H., Chwa, K. Y. y Shin, S. Y. (1998): Efficient Algorithms for Computing a Complete Visibility Region in Three-Dimensional Space, *Algorithmica*, vol. 20, pp. 201-225.

Kimura, M. (1979): The Neutral Theory of Molecular Evolution, *Scientific American*, vol. 241, pp. 98-126.

King, T. (1992): *Dynamic Data Structures: Theory and Applications*, Academic Press, Londres.

Kingston, J. H. (1986): The Amortized Complexity of Henriksen's Algorithm, *BIT*, vol. 26, No. 2, pp. 156-163.

Kirousis, L. M. y Papadimitriou, C. H. (1988): The Complexity of Recognizing Polyhedral Scenes, *Journal of Computer and System Sciences*, vol. 37, No. 1, pp. 14-38.

Kirpatrick, D. y Snoeyink, J. (1993): Tentative Prune-and-Search for Computing Fixed-Points with Applications to Geometric Computation, *Proceedings of the Ninth Annual Symposium on Computational Geometry*, ACM Press, San Diego, California, pp. 133-142.

Kirpatrick, D. y Snoeyink, J. (1995): Tentative Prune-and-Search for Computing Fixed-Points with Applications to Geometric Computation, *Fundamenta Informaticae*, vol. 22, pp. 353-370.

Kirschenhofer, P., Prodinger, H. y Szpankowski, W. (1994): Digital Search Trees Again Revisited: The Internal Path Length Perspective, *SIAM Journal on Applied Mathematics*, vol. 23, No. 3, pp. 598-616.

Klarlund, N. (1999): An *n* log *n* Algorithm for Online BDD Refinement, *Journal of Algorithms*, vol. 32, pp. 133-154.

Klawe, M. M. (1985): A Tight Bound for Black and White Pebbles on the Pyramid, *Journal of the ACM*, vol. 32, No. 1, pp. 218-228.

Kleffe, J. y Borodovsky, M. (1992): First and Second Moment of Counts of Words in Random Texts Generated by Markov Chains, *Computer Applications in the Biosciences*, vol. 8, pp. 433-441.

Klein, C. M. (1995): A Submodular Approach to Discrete Dynamic Programming, *European Journal of Operational Research*, vol. 80, pp. 145-155.

Klein, P. y Subramanian, S. (1997): A Randomized Parallel Algorithm for Single-Source Shortest Paths, *Journal of Algorithm*, vol. 25, pp. 205-220.

Kleinberg, J. y Tardos, E. (2002): Approximation Algorithms for Classification Problems with Pairwise Relationships: Metric Labeling and Markov Random Fields, *Journal of the ACM*, vol. 49, No. 5, pp. 616-639.

Knuth, D. E. (1969): The Art of Computer Programming, *Fundamental Algorithms*, vol. 1, p. 634.

Knuth, D. E. (1971): Optimum Binary Search Trees, *Acta Informatica*, vol. 1, pp. 14-25.

Knuth, D. E. (1973): *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass.

Ko, M. T., Lee, R. C. T. y Chang, J. S. (1990): An Optimal Approximation Algorithm for the Rectilinear *m*-Center Problem, *Algorithmica*, vol. 5, pp. 341-352.

Kolliopoulos, S. G. y Stein, C. (2002): Approximation Algorithms for Single-Source Unsplittable Flow, *SIAM Journal on Computing*, vol. 31, No. 3, pp. 919-946.

Kolman, P. y Scheideler, C. (2001): Simple On-Line Algorithms for the Maximum Disjoint Paths Problem, *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, ACM Press, Crete Island, Grecia, pp. 38-47.

Kontogiannis, S. (2002): Lower Bounds and Competitive Algorithms for Online Scheduling of Unit-Size Tasks to Related Machines, *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, ACM Press, Montreal, Canada, pp. 124-133.

Koo, C. Y., Lam, T. W., Ngan, T. W., Sadakane, K. y To, K. K. (2003): On-line Scheduling with Tight Deadlines, *Theoretical Computer Science*, vol. 295, 2003, pp. 1-12.

Korte, B. y Louasz, L. (1984): Greedoids: A Structural Framework for the Greedy Algorithms, in W. R. Pulleybland (Ed.), *Progress in Combinatorial Optimization* (pp. 221-243), Academic Press, Londres.

Kortsarz, G. y Peleg, D.(1995): Approximation Algorithms for Minimum-Time Broadcast, *SIAM Journal on Discrete Mathematics*, vol. 8, pp. 407-421.

Kossmann, D., Ramsak, F. y Rost, S. (2002): Shooting Stars in the Sky: An Online Algorithm for Skyline Queries, *Proceedings of the 28th VLDB Conference*, VLDB Endowment, Hong Kong, pp. 275-286.

Kostreva, M. M. y Wiecek, M. M. (1993): Time Dependency in Multiple Objective Dynamic Programming, *Journal of Mathematical Analysis and Applications*, vol. 173, pp. 289-307.

Kou, L., Markowsky, G. y Berman, L. (1981): A Fast Algorithm for Steiner Trees, *Acta Informatica*, vol. 15, pp. 141-145.

Koutsoupias, E. y Nanavati, A. (2003): Online Matching Problem on a Line, *Lecture Notes in Computer Science*, vol. 2909, pp. 179-191.

Koutsoupias, E. y Papadimitriou, C. H. (1995): On the *k*-Server Conjecture, *Journal of the ACM*, vol. 42, No. 5, pp. 971-983.

Kozen, D. C. (1997): *The Design and Analysis of Algorithms*, Springer-Verlag, Nueva York.

Kozhukhin, C. G. y Pevzner, P. A. (1994): Genome Inhomogeneity Is Determined Mainly by WW and SS Dinucleotides, *Computer Applications in the Biosciences*, pp. 145-151.

Krarup, J. y Pruzan, P. (1986): Assessment Approximate Algorithms: The Error Measures's Crucial Role, *BIT*, vol. 26, No. 3, pp. 284-294.

Krivanek, M. y Moravek, J. (1986): NP-Hard Problems in Hierarchical-Tree Clustering, *Acta Informatica*, vol. 23, No. 3, pp. 311-323.

Krogh, A., Brown, M., Mian, I. S., Sjolander, K. y Haussler, D. (1994): Hidden Markov Models in Computational Biology: Applications to Protein Modeling, *Journal of Molecular Biology*, vol. 235, pp. 1501-1531.

Kronsjö, L. I. (1987): *Algorithms: Their Complexity and Efficiency*, John Wiley & Sons, Nueva York.

Krumke, S. O., Marathe, M. V. y Ravi, S. S. (2001): Models and Approximation Algorithms for Channel Assignment in Radio Networks, *Wireless Networks*, vol. 7, No. 6, pp. 575-584.

Kruskal, J. B. Jr. (1956): On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem, *Proceedings of the American Mathematical Society*, vol. 7, No. 1, pp. 48-50.

Kryazhimskiy, A. V. y Savinov V. B. (1995): Travelling-Salesman Problem with Moving Objects, *Journal of Computer and System Sciences*, vol. 33, No. 3, pp. 144-148.

Kucera, L. (1991): *Combinatorial Algorithms*, IOP Publishing, Philadelphia.

Kumar, S., Kiran, R. y Pandu, C. (1987): Linear Space Algorithms for the LCS Problem, *Acta Informatica*, vol. 24, No. 3, pp. 353-362.

Kung, H. T., Luccio, F. y Preparata, F. P. (1975): On Finding the Maxima of a Set of Vectors, *Journal of the ACM*, vol. 22, No. 4, pp. 469-476.

Kurtz, S. A. (1987): A Note on Randomized Polynomial Time, *SIAM Journal on Computing*, vol. 16, No. 5, pp. 852-853.

Lai, T. W. y Wood, D. (1998): Adaptive Heuristics for Binary Search Trees and Constant Linkage Cost, *SIAM Journal on Applied Mathematics*, vol. 27, No. 6, pp. 1564-1591.

Landau, G. M. y Schmidt, J. P. (1993): An Algorithm for Approximate Tandem Repeats, *Lecture Notes in Computer Science*, vol. 684, pp. 120-133.

Landau, G. M. y Vishkin, U. (1989): Fast Parallel and Serial Approximate String Matching, *Journal of Algorithms*, vol. 10, pp. 157-169.

Langston, M. A. (1982): Improved 0=1-Interchange Scheduling, *BIT*, vol. 22, pp. 282-290.

Lapaugh, A. S. (1980): Algorithms for Integrated Circuit Layout: Analytic Approach, *Computer Science*, pp. 155-169.

Laquer, H. T. (1981): Asymptotic Limits for a Two-Dimensional Recursion, *Studies in Applied Mathematics*, pp. 271-277.

Larson, P. A. (1984): Analysis of Hashing with Chaining in the Prime Area, *Journal of Algorithms*, vol. 5, pp. 36-47.

Lathrop, R. H. (1994): The Protein Threading Problem with Sequence Amino Acid Interaction Preferences Is NP-Complete (Prove NP-Complete Problem), *Protein Engineering*, vol. 7, pp. 1059-1068.

Lau, H. T. (1991): *Algorithms on Graphs*, TAB Books, Blue Ridge Summit, Filadelfia.

Lawler, E. L. (1976): *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, Nueva York.

Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H. G. y Shmoys, D. B. (1985): *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, John Wiley & Sons, Nueva York.

Lawler, E. L. y Moore, J. (1969): A Functional Equation and Its Application to Resource Allocation and Sequencing Problems, *Management Science*, vol. 16, No. 1, pp. 77-84.

Lawler, E. L. y Wood, D. (1966): Branch-and-Bound Methods: A Survey, *Operations Research*, vol. 14, pp. 699-719.

Lee, C. C. y Lee, D. T. (1985): A Simple On-Line Bin-Packing Algorithm, *Journal of the Association for Computing Machinery*, vol. 32, No. 3, pp. 562-572.

Lee, C. T. y Sheu, C. Y. (1992): A Divide-and-Conquer Approach with Heuristics of Motion Planning for a Cartesian Manipulator, *IEEE Transactions on Systems, Man and Cybernetics*, vol. 22, No. 5, pp. 929-944.

Lee, D. T. (1982): On *k*-Nearest Neighbor Voronoi Diagrams in the Plane, *IEEE Transactions on Computers*, vol. C-31, pp. 478-487.

Lee, D. T. y Lin, A. K. (1986): Computational Complexity of Art Gallery Problems, *IEEE Transactions on Information Theory*, vol. IT-32, No. 2, pp. 276-282.

Lee, D. T. y Preparata, F. P. (1984): Computational Geometry: A Survey, *IEEE Transactions on Computers*, vol. C-33, pp. 1072-1101.

Lee, J. (2003a): Online Deadline Scheduling: Team Adversary and Restart, *Lecture Notes in Computer Science*, vol. 2909, pp. 206-213.

Lee, J. (2003b): Online Deadline Scheduling: Multiple Machines and Randomization, *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures*, ACM Press, San Diego, California, pp. 19-23.

Leighton, T. y Rao, S. (1999): Multicommodity Max-Flow Min-Cut Theorems and Their Use in Designing Approximation Algorithms, *Journal of the ACM*, vol. 46, No. 6, pp. 787-832.

Lent, J. y Mahmoud, H. M. (1996): On Tree-Growing Search Strategies, *The Annals of Applied Probability*, vol. 6, No. 4, pp. 1284-1302.

Leonardi, S., Spaccamela, A. M., Presciutti, A. y Ros, A. (2001): On-Line Randomized Call Control Revisited, *SIAM Journal on Computing*, vol. 31, pp. 86-112.

Leoncini, M., Manzini, G. y Margara, L. (1999): Parallel Complexity of Numerically Accurate Linear System Solvers, *SIAM Journal on Computing*, vol. 28, No. 6, pp. 2030-2058.

Levcopoulos, C. y Lingas, A. (1987): On Approximation Behavior of the Greedy Triangulation for Convex Polygons, *Algorithmica*, vol. 2, pp. 175-193.

Levin, L. A. (1986): Average Case Complete Problem, *SIAM Journal on Computing*, vol. 15, No. 1, pp. 285-286.

Lew, W. y Mahmoud, H. M. (1992): The Joint Distribution of Elastic Buckets in Multiway Search Trees, *SIAM Journal on Applied Mathematics*, vol. 23, No. 5, pp. 1050-1074.

Lewandowski, G., Condon, A. y Bach, E. (1996): Asynchronous Analysis of Parallel Dynamic Programming Algorithms, *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 425-438.

Lewis, H. R. y Denenberg, L. (1991): *Data Structures and Their Algorithms*, Harper Collins, Nueva York.

Liang, S. Y. (1985): *Parallel algorithm for a personnel assignment problem*. Tesis de maestría, aún sin publicar, National Tsing Hua University, Hsinchu, Taiwán.

Liang, Y. D. (1994): On the Feedback Vertex Set Problem in Permutation Graphs, *Information Processing Letters*, vol. 52, No. 3, pp. 123-129.

Liao, L. Z. y Shoemaker, C. A. (1991): Convergence in Unconstrained Discrete-time Differential Dynamic Programming, *IEEE Transactions Automatic Control*, vol. 36, pp. 692-706.

Liaw, B. C. y Lee, R. C. T. (1994): Optimal Algorithm to Solve the Minimum Weakly Cooperative Guards Problem for 1-Spiral Polygons, *Information Processing Letters*, vol. 52, No. 2, pp. 69-75.

Lin, C. K., Fan, K. C. y Lee, F. T. (1993): On-line Recognition by Deviation-Expansion Model and Dynamic Programming Matching, *Pattern Recognition*, vol. 26, No. 2, pp. 259-268.

Lin, G., Chen, Z. Z., Jiang, T. y Wen, J. (2002): The Longest Common Subsequence Problem for Sequences with Nested Arc Annotations (Prove NP-Complete Problem), *Journal of Computer and System Sciences*, vol. 65, pp. 465- 480.

Lipton, R. J. (1995): Using DNA to Solve NP-Complete Problems, *Science*, vol. 268, pp. 542-545.

Little, J. D. C., Murty, K. G., Sweeney, D. W. y Karel, C. (1963): An Algorithm for the Traveling Salesman Problem, *Operations Researc*h, vol. 11, pp. 972-989.

Littman, M. L., Cassandra, A. R. y Kaelbling, L. P. (1996): An Algorithm for Probabilistic Planning: Efficient Dynamic-Programming Updates in Partially Observable Markov Decision Processes, *Artificial Intelligence*, vol. 76, pp. 239-286.

Liu, C. L. (1985): *Elements of Discrete Mathematic*s, McGraw-Hill, Nueva York.

Liu, J. (2002): On Adaptive Agentlets for Distributed Divide-and-Conquer: A Dynamical Systems Approach, *IEEE Transactions on Systems, Man and Cybernetics*, vol. 32, No. 2, pp. 214-227.

Lo, V., Rajopadhye, S., Telle, J. A. y Zhong, X. (1996): Parallel Divide and Conquer on Meshes, *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, No. 10, pp. 1049-1058.

Long, T. J. y Selman, A. L. (1986): Relativizing Complexity Classes with Sparse Oracles, *Journal of the ACM*, vol. 33, No. 3, pp. 618-627.

Lopez, J. y Zapata, E. (1994): Unified Architecture for Divide and Conquer Based Tridiagonal System Solvers, *IEEE Transactions on Computers*, vol. 43, No. 12, pp. 1413-1425.

Louchard, G., Szpankowski, W. y Tang, J. (1999): Average Profile of the Generalized Digital Search Tree and the Generalized Lempel—Ziv Algorithm, *SIAM Journal on Applied Mathematics*, vol. 28, No. 3, pp. 904-934.

Lovasz, L., Naor, M., Newman, I. y Wigderson, A. (1995): Search Problems in the Decision Tree Model, *SIAM Journal on Applied Mathematics*, vol. 8, No. 1, pp. 119-132.

Luby, M. (1986): A Simple Parallel Algorithm for the Maximal Independent Set Problem, *SIAM Journal on Computing*, vol. 15, No. 4, pp. 1036-1053.

Lueker, G. (1998): Average-Case Analysis of Off-Line and On-Line Knapsack Problems, *Journal of Algorithms*, vol. 29, pp. 277-305.

Lyngso, R. B. y Pedersen, C. N. S. (2000): Pseudoknots in RNA Secondary Structure (Prove NP-Complete Problem), *ACM*, pp. 201-209.

Ma, B., Li, M. y Zhang, L. (2000): From Gene Trees to Species Trees, *SIAM Journal on Applied Mathematics*, vol. 30, No. 3, pp. 729-752.

Maes, M. (1990): On a Cyclic String-to-String Correction Problem, *Information Processing Letters*, vol. 35, pp. 73-78.

Maffioli, F. (1986): Randomized Algorithm in Combinatorial Optimization: A Survey, *Discrete Applied Mathematics*, vol. 14, No. 2, junio, pp. 157-170.

Maggs, B. M. y Sitaraman, R. K. (1999): Simple Algorithms for Routing on Butterfly Networks with Bounded Queues, *SIAM Journal on Computing*, vol. 28, pp. 984-1003.

Maier, D. (1978): The Complexity of Some Problems on Subsequences and Supersequences, *Journal of the ACM*, vol. 25, pp. 322-336.

Maier, D. y Storer, J. A. (1978): A Note on the Complexity of the Superstring Problem, *Proceedings of the 12th Conference on Information Sciences and Systems (CISS)*, The Johns Hopkins University, Baltimore, Maryland, pp. 52-56.

Makinen, E. (1987): On Top-Down Splaying, *BIT*, vol. 27, No. 3, pp. 330-339.

Manacher, G. (1975): A New Linear-Time "On-line" Algorithm for Finding the Smallest Initial Palindrome of a String, *Journal of the ACM*, vol. 22, No. 3, pp. 346-351.

Manasse, M., McGeoch, L. y Sleator, D. (1990): Competitive Algorithms for Server Problems, *Journal of Algorithms*, vol. 11, No. 2, pp. 208-230.

Manber, U. (1989): *Introduction to Algorithms: A Creative Approach*, Addison-Wesley, Reading, Mass.

Mandic, D. y Cichocki, A. (2003): An Online Algorithm for Blind Extraction of Sources with Different Dynamical Structures, *Proceedings of the 4th International Symposium on Independent Component Analysis and Blind Signal Separation (ICA2003)*, Nara, Japón, pp. 645-650.

Mandrioli, D. y Ghezzi, C. (1987): *Theoretical Foundations of Computer Science*, John Wiley & Sons, Nueva York.

Maniezzo, V. (1998): Exact and Approximate Nondeterministic Tree-Search Procedures for the Quadratic Assignment Problem, *Research Report CSR 98-1*.

Mansour, Y. y Schieber, B. (1992): The Intractability of Bounded Protocols for Online Sequence Transmission over Non-FIFO Channels, *Journal of the ACM*, vol. 39, No. 4, pp. 783-799.

Marion, J. Y. (2003): Analysing the Implicit Complexity of Programs, *Information and Computation*, vol. 183, No. 1, pp. 2-18.

Martello, S. y Toth, P. (1990): *Knapsack Problem Algorithms & Computer Implementations*, John Wiley & Sons, Nueva York.

Martin, G. L. y Talley, J. (1995): Recognizing Handwritten Phrases from U. S. Census Forms by Combining Neural Networks and Dynamic Programming, *Journal of Artificial Neural Networks*, vol. 2, pp. 167-193.

Martinez, C. y Roura, S. (2001): Optimal Sampling Strategies in Quicksort and Quickselect, *SIAM Journal on Computing*, vol. 31, No. 3, pp. 683-705.

Matousek, J. (1991): Randomized Optimal Algorithm for Slope Selection, *Information Processing Letters*, vol. 39, No. 4, pp. 183-187.

Matousek, J. (1995): On Enclosing *K* Points by a Circle, *Information Processing Letters*, vol. 53, No. 4, pp. 217-221.

Matousek, J. (1996): Derandomization in Computational Geometry, *Journal of Algorithms*, vol. 20, pp. 545-580.

Mauri, G., Pavesi, G. y Piccolboni, A. (1999): Approximation Algorithms for Protein Folding Prediction, *Proceedings of the 10th Annual Symposium on Discrete Algorithms*, SIAM, Baltimore, Maryland, pp. 945-946.

McDiarmid, C. (1988): Average-Case Lower Bounds for Searching, *SIAM Journal on Computing*, vol. 17, No. 5, pp. 1044-1060.

McHugh, J. A. (1990): *Algorithmic Graph Theory*, Prentice-Hall, London.

Meacham, C. A. (1981): A Probability Measure for Character Compatibility, *Mathematical Biosciences*, vol. 57, pp. 1-18.

Megiddo, N. (1983): Linear-Time Algorithm for Linear Programming in R3 and Related Problems, *SIAM Journal on Computing*, vol. 12, No. 4, pp. 759-776.

Megiddo, N. (1984): Linear Programming in Linear Time When the Dimension Is Fixed, *Journal of the ACM*, vol. 31, No. 1, pp. 114-127.

Megiddo, N. (1985): Note Partitioning with Two Lines in the Plane, *Journal of Algorithms*, vol. 6, No. 3, pp. 430-433.

Megiddo, N. y Supowit, K. J. (1984): On the Complexity of Some Common Geometric Location Problems, *SIAM Journal on Computing*, vol. 13, No. 1, pp. 182-196.

Megiddo, N. y Zemel, E. (1986): An $O(n \log n)$ Randomizing Algorithm for the Weighted Euclidean 1-Center Problem, *Journal of Algorithms*, vol. 7, No. 3, pp. 358-368.

Megow, N. y Schulz, A. (2003): Scheduling to Minimize Average Completion Time Revisited: Deterministic On-Line Algorithms, *Lecture Notes in Computer Science*, vol. 2909, pp. 227-234.

Mehlhorn, K. (1984): *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*, Springer-Verlag, Berlin.

Mehlhorn, K. (1984): *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag, Berlín.

Mehlhorn, K. (1987): *Data Structures & Algorithms: Sorting and Searching*, Springer-Verlag, Nueva York.

Mehlhorn, K. (1988): A Faster Approximation Algorithm for the Steiner Problems in Graphs, *Information Processing Letters*, vol. 27, No. 3, pp. 125-128.

Mehlhorn, K., Naher, S. y Alt, H. (1988): A Lower Bound on the Complexity of the Union-Split-Find Problem, *SIAM Journal on Computing*, vol. 17, No. 6, pp. 1093-1102.

Mehlhorn, K. y Tsakalidis, A. (1986): An Amortized Analysis of Insertions into AVL-Trees, *SIAM Journal on Computing*, vol. 15, No. 1, pp. 22-33.

Meijer, H. y Rappaport, D. (1992): Computing the Minimum Weight Triangulation of a Set of Linearly Ordered Points, *Information Processing Letters*, vol. 42, No. 1, pp. 35-38.

Meleis, W. M. (2001): Dual-Issue Scheduling for Binary Trees with Spills and Pipelined Loads, *SIAM Journal on Applied Mathematics*, vol. 30, No. 6, pp. 1921-1941.

Melnik, S. y Garcia-Molina, H. (2002): Divide-and-Conquer Algorithm for Computing Set Containment Joins, *Lecture Notes in Computer Science*, vol. 2287, pp. 427-444.

Merlet, N. y Zerubia, J. (1996): New Prospects in Line Detection by Dynamic Programming, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, pp. 426-431.

Messinger, E., Rowe, A. y Henry, R. (1991): A Divide-and-Conquer Algorithm for the Automatic Layout of Large Directed Graphs, *IEEE Transactions on Systems, Man and Cybernetics*, vol. 21, No. 1, pp. 1-12.

Miller, G. L. y Teng, S. H. (1999): The Dynamic Parallel Complexity of Computational Circuits, *SIAM Journal on Computing*, vol. 28, No. 5, pp. 1664-1688.

Minoux, M. (1986): *Mathematical Programming: Theory and Algorithms*, John-Wiley & Sons, Nueva York.

Mitten, L. (1970): Branch-and-Bound Methods: General Formulation and Properties, *Operations Research*, vol. 18, pp. 24-34.

Mohamed, M. y Gader, P. (1996): Handwritten Word Recognition Using Segmentation-Free Hidden Markov Modeling and Segmentation-Based Dynamic Programming Techniques, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, pp. 548-554.

Monien, B. y Sudborough, I. H. (1988): Min Cut Is NP-Complete for Edge Weighted Trees, *Theoretical Computer Science*, vol. 58, No. 1-3, pp. 209-229.

Monier, L. (1980): Combinatorial Solutions of Multidimensional Divide-and-Conquer Recurrences, *Journal of Algorithms*, vol. 1, pp. 69-74.

Moor, O. de (1994): Categories Relations and Dynamic Programming, *Mathematical Structures in Computer Science*, vol. 4, pp. 33-69.

Moran, S. (1981): General Approximation Algorithms for Some Arithmetical Combinatorial Problems, *Theoretical Computer Science*, vol. 14, pp. 289-303.

Moran, S., Snir, M. y Manber, U. (1985): Applications of Ramsey's Theorem to Decision Tree Complexity, *Journal of the ACM*, vol. 32, pp. 938-949.

Moret, B. M. E. y Shapiro, H. D. (1991): *Algorithms from P to NP*, Benjamin Cummings, Redwood City, California.

Morin, T. y Marsten, R. E. (1976): Branch-and-Bound Strategies for Dynamic Programming, *Operations Research*, vol. 24, pp. 611-627.

Motta, M. y Rampazzo, F. (1996): Dynamic Programming for Nonlinear Systems Driven by Ordinary and Impulsive Controls, *SIAM Journal on Control and Optimization*, vol. 34, pp. 199-225.

Motwani, R. y Raghavan, P. (1995): *Randomized Algorithms*, Cambridge University Press, Cambridge, Inglaterra.

Mulmuley, K. (1998): *Computational Geometry: An Introduction through Randomized Algorithms*, Prentice-Hall, Englewoods Cliffs, Nueva Jersey.

Mulmuley, K., Vazirani, U. V. y Vazirani, V. V. (1987): Matching Is as Easy as Matrix Inversion, *Combinatorica*, vol. 7, No. 1, pp. 105-113.

Murgolo, F. D. (1987): An Efficient Approximation Scheme for Variable-Sized Bin Packing, *SIAM Journal on Computing*, vol. 16, No. 1, pp. 149-161.

Myers, E. y Miller, W. (1989): Approximate Matching of Regular Expression, *Bulletin of Mathematical Biology*, vol. 51, pp. 5-37.

Myers, E. W. (1994): A Sublinear Algorithm for Approximate Keyword Searching, *Algorithmica*, vol. 12, No. 4-5, pp. 345-374.

Myoupo, J. F. (1992): Synthesizing Linear Systolic Arrays for Dynamic Programming Problems, *Parallel Processing Letters*, vol. 2, pp. 97-110.

Nakayama, H., Nishizeki, T. y Saito, N. (1985): Lower Bounds for Combinatorial Problems on Graphs, *Journal of Algorithms*, vol. 6, pp. 393-399.

Naor, M. y Ruah, S. (2001): On the Decisional Complexity of Problems Over the Reals, *Information and Computation*, vol. 167, No. 1, pp. 27-45.

Nau, D. S., Kumar, V. y Kanal, L. (1984): General Branch and Bound and Its Relation to A* and AO*, *Artificial Intelligence*, vol. 23, pp. 29-58.

Neapolitan, R. E. y Naimipour, K. (1996): *Foundations of Algorithms*, D.C. Heath and Company, Lexington, Mass.

Neddleman, S. B. y Wunsch, C. D. (1970): A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins, *Journal of Molecular Biology*, vol. 48, pp. 443-453.

Nemhauser, G. L. (1966): *Introduction to Dynamic Programming*, John Wiley & Sons, Nueva York.

Nemhauser, G. L. y Ullman, Z. (1969) Discrete Dynamic Programming and Capital Allocation, *Management Science*, vol. 15, No. 9, pp. 494-505.

Neogi, R. y Saha, A. (1995): Embedded Parallel Divide-and-Conquer Video Decompression Algorithm and Architecture for HDTV Applications, *IEEE Transactions on Consumer Electronics*, vol. 41, No. 1, pp. 160-171.

Ney, H. (1984): The Use of a One-Stage Dynamic Programming Algorithm for Connected Word Recognition, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 2, pp. 263-271.

Ney, H. (1991): Dynamic Programming Parsing for Context-Free Grammars in Continuous Speech Recognition, *IEEE Transactions on Signal Processing*, vol. 39, pp. 336-340.

Nilsson, N. J. (1980): *Principles of Artificial Intelligence*, Tioga Publishing Company, Palo Alto, California.

Nishizeki, T., Asano, T. y Watanabe, T. (1983): An Approximation Algorithm for the Hamiltonian Walk Problem on a Maximal Planar Graph, *Discrete Applied Mathematics*, vol. 5, No. 2, pp. 211-222.

Nishizeki, T. y Chiba, N. (1988): *Planar Graphs: Theory and Algorithms*, Elsevier, Ámsterdam.

Novak, E. y Wozniakowski, H. (2000): Complexity of Linear Problems with a Fixed Output Basis, *Journal of Complexity*, vol. 16, No. 1, pp. 333-362.

Nuyts, J., Suetens, P., Oosterlinck, A., Roo, M. De y Mortelmans, L. (1991): Delineation of ECT Images Using Global Constraints and Dynamic Programming, *IEEE Transactions on Medical Imaging*, vol. 10, No. 4, pp. 489-498.

Ohno, S., Wolf, U. y Atkin, N. B. (1968): Evolution from Fish to Mammals by Gene Duplication, *Hereditas*, vol. 59, pp. 708-713.

Ohta, Y. y Kanade, T. (1985): Stereo by Intra- and Inter-Scanline Search Using Dynamic Programming, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 7, pp. 139-154.

Oishi, Y. y Sugihara, K. (1995): Topology-Oriented Divide-and-Conquer Algorithm for Voronoi Diagrams, *Graphical Models and Image Processing*, vol. 57, No. 4, pp. 303-314.

Omura, J. K. (1969): On the Viterbi Decoding Algorithm, *IEEE Transactions on Information Theory*, pp. 177-179.

O'Rourke, J. (1987): *Art Gallery Theorems and Algorithms*, Oxford University Press, Cambridge, Inglaterra.

O'Rourke, J. (1998): *Computational Geometry in C*, Cambridge University Press, Cambridge, Inglaterra.

Orponen, P. y Mannila, H. (1987): On Approximation Preserving Reductions: Complete Problems and Robust Measures, *Technical Report C-1987-28*.

Ouyang, Z. y Shahidehpour, S. M. (1992): Hybrid Artificial Neural Network-Dynamic Programming Approach to Unit Commitment, *IEEE Transactions on Power Systems*, vol. 7, pp. 236-242.

Owolabi, O. y McGregor, D. R. (1988): Fast Approximate String Matching, *Software Practice and Experience*, vol. 18, pp. 387-393.

Oza, N. y Russell, S. (2001): Experimental Comparisons of Online and Batch Versions of Bagging and Boosting, *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM Press, San Francisco, California, pp. 359-364.

Ozden, M. (1988): A Solution Procedure for General Knapsack Problems with a Few Constraints, *Computers and Operations Research*, vol. 15, No. 2, pp. 145-156.

Pach, J. (1993): *New Trends in Discrete and Computational Geometry*, Springer-Verlag, Nueva York.

Pacholski, L., Szwast, W. y Tendera, L. (2000): Complexity Results for First-Order Two-Variable Logic with Counting, *SIAM Journal on Computing*, vol. 29, No. 4, pp. 1083-1117.

Pandurangan, G. y Upfal, E. (2001): Can Entropy Characterize Performance of Online Algorithms? *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, Washington, DC, pp. 727-734.

Papadimitriou, C. H. (1977): The Euclidean TSP is NP-Complete, *Theoretical Computer Science*, vol. 4, pp. 237-244.

Papadimitriou, C. H. (1981): Worst-Case and Probabilistic Analysis of a Geometric Location Problem, *SIAM Journal on Computing*, vol. 10, No. 3, pp. 542-557.

Papadimitriou, C. H. (1994): *Computational Complexity*, Addison-Wesley, Reading, Mass.

Papadimitriou, C. H. y Steiglitz, K. (1982): *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, Nueva Jersey.

Papadimitriou, C. H. y Yannakakis, M. (1991a): Optimization; Approximation; and Complexity Classes, *Journal of Computer and System Sciences*, vol. 43, pp. 425-440.

Papadimitriou, C. H. y Yannakakis, M. (1991b): Shortest Paths without a Map, *Theoretical Computer Science*, vol. 84, pp. 127-150.

Papadimitriou, C. H. y Yannakakis, M. (1992): The Traveling Salesman Problem with Distances One and Two, *Mathematics of Operations Research*, vol. 18, No. 1, pp. 1-11.

Papadopoulou, E. y Lee, D. T. (1998): A New Approach for the Geodesic Voronoi Diagram of Points in a Simple Polygon and Other Restricted Polygonal Domains, *Algorithmica*, vol. 20, pp. 319-352.

Parida, L., Floratos, A. y Rigoutsos, I. (1999): An Approximation Algorithm for Alignment of Multiple Sequences Using Motif Discovery, *Journal of Combinatorial Optimization*, vol. 3, No. 2-3, pp. 247-275.

Park, J. K. (1991): Special Case of the *n*-Vertex Traveling-Salesman Problem That Can Be Solved in $O(n)$ Time, *Information Processing Letters*, vol. 40, No. 5, pp. 247-254.

Parker, R. G. y Rardin, R. L. (1984): Guaranteed Performance Heuristic for the Bottleneck Traveling Salesperson Problem, *Operations Research Letters*, vol. 2, No. 6, pp. 269-272.

Pearl, J. (1983): Knowledge Versus Search: A Quantitative Analysis Using A*, *Artificial Intelligence*, vol. 20, pp. 1-13.

Pearson, W. R. y Miller, W. (1992): Dynamic Programming Algorithms for Biological Sequence Comparison, *Methods in Enzymology*, vol. 210, pp. 575-601.

Pe'er, I. y Shamir, R. (1998): The Median Problems for Breakpoints are NP-Complete, *Electronic Colloquium on Computational Complexity*, vol. 5, No. 71, pp. 1-15.

Pe'er, I. y Shamir, R. (2000): Approximation Algorithms for the Median Problem in the Breakpoint Model, en D. Sankoff y J. H. Nadeau (Eds.), *Comparative Genomics: Empirical and Analytical Approaches to Gene Order Dynamics, Map Alignment and the Evolution of Gene Families*, Kluwer Academic Press, Dordrecht, Holanda.

Peleg, D. y Rubinovich, V. (2000): A Near-Tight Lower Bound on the Time Complexity of Distributed Minimum-Weight Spanning Tree Construction, *SIAM Journal on Applied Mathematics*, vol. 30, No. 5, pp. 1427-1442.

Peng, S., Stephens, A. B. y Yesha, Y. (1993): Algorithms for a Core and *k*-Tree Core of a Tree, *Journal of Algorithms*, vol. 15, pp. 143-159.

Penny D., Hendy, M. D. y Steel, M. (1992): Progress with Methods for Constructing Evolutionary Trees, *Trends in Ecology and Evolution*, vol. 7, No. 3, pp. 73-79.

Perl, Y. (1984): Optimum Split Trees, *Journal of Algorithms*, vol. 5, pp. 367-374.

Peserico, E. (2003): Online Paging with Arbitrary Associativity, *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, Baltimore, Maryland, pp. 555-564.

Petr, S. (1996): A Tight Analysis of the Greedy Algorithm for Set Cover, *ACM*, pp. 435-441.

Pevzner, P. A. (1992): Multiple Alignment; Communication Cost; and Graph Matching, *SIAM Journal on Applied Mathematics*, vol. 52, No. 6, pp. 1763-1779.

Pevzner, P. A. (2000): *Computational Molecular Biology: An Algorithmic Approach*, The MIT Press, Boston.

Pevzner, P. A. y Waterman, M. S. (1995): Multiple Filtration and Approximate Pattern Matching, *Algorithmica*, vol. 13, No. 1-2, pp. 135-154.

Pierce, N. A. y Winfree, E. (2002): Protein Design Is NP-Hard (Prove NP-Complete Problem), *Protein Engineering*, vol. 15, No. 10, pp. 779-782.

Pittel, B. y Weishaar, R. (1997): On-Line Coloring of Sparse Random Graphs and Random Trees, *Journal of Algorithms*, vol. 23, pp. 195-205.

Pohl, I. (1972): A Sorting Problem and Its Complexity, *Communications of the ACM*, vol. 15, No. 6, pp. 462-463.

Ponzio, S. J., Radhakrishnan, J. y Venkatesh, S. (2001): The Communication Complexity of Pointer Chasing, *Journal of Computer and System Sciences*, vol. 62, No. 2, pp. 323-355.

Preparata, F. P. y Hong, S. J. (1977): Convex Hulls of Finite Sets of Points in Two and Three Dimensions, *Communications of the ACM*, vol. 2, No. 20, pp. 87-93.

Preparata, F. P. y Shamos, M. I. (1985): *Computational Geometry: An Introduction*, Springer-Verlag, Nueva York.

Prim, R. C. (1957): Shortest Connection Networks and Some Generalizations, *Bell System Technical Journal*, pp. 1389-1401.

Promel, H. J. y Steger, A. (2000): A New Approximation Algorithm for the Steiner Tree Problem with Performance Ratio 5/3, *Journal of Algorithms*, vol. 36, pp. 89-101.

Purdom, P. W. Jr. y Brown, C. A. (1985a): *The Analysis of Algorithms*, Holt, Rinehart and Winston, Nueva York.

Purdom, P. W. Jr. y Brown, C. A. (1985b): The Pure Literal Rule and Polynomial Average Time, *SIAM Journal on Computing*, vol. 14, No. 4, pp. 943-953.

Rabin, M. O. (1976): Probabilistic Algorithm. In J. F. Traub (Ed.), *Algorithms and Complexity: New Directions and Recent Results* (pp. 21-39), Academic Press, Nueva York.

Raghavachari, B. y Veerasamy, J. (1999): A 3/2-Approximation Algorithm for the Mixed Postman Problem, *SIAM Journal on Discrete Mathematics*, vol. 12, No. 4, pp. 425-433.

Raghavan, P. (1988): Probabilistic Construction of Deterministic Algorithms: Approximating Packing Integer Programs, *Journal of Computer and System Sciences*, vol. 37, No. 2, pp. 130-143.

Raghavan, P. y Thompson C. (1987): Randomized Rounding: A Technique for Provably Good Algorithms and Algorithmic Proofs, *Combinatorica*, vol. 7, No. 4, pp. 365-374.

Ramanan, P., Deogun, J. S. y Liu, C. L. (1984): A Personnel Assignment Problem, *Journal of Algorithms*, vol. 5, No. 1, pp. 132-144.

Ramesh, H. (1995): On Traversing Layered Graphs On-Line, *Journal of Algorithms*, vol. 18, pp. 480-512.

Rangan, C. P. (1983): On the Minimum Number of Additions Required to Compute a Quadratic Form, *Journal of Algorithms*, vol. 4, pp. 282-285.

Reingold, E. M., Nievergelt, J. y Deo, N. (1977): *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, Englewood Cliffs, Nueva Jersey.

Reingold, E. M. y Supowit, K. J. (1983): Probabilistic Analysis of Divide-and-Conquer Heuristics for Minimum Weighted Euclidean Matching, *Networks*, vol. 13, No. 1, pp. 49-66.

Rival, I. y Zaguia, N. (1987): Greedy Linear Extensions with Constraints, *Discrete Mathematics*, vol. 63, No. 2, pp. 249-260.

Rivas, E. y Eddy, S. R. (1999): A Dynamic Programming Algorithm for RNA Structure Prediction Including Pseudoknots, *Journal of Molecular Biology*, vol. 285, pp. 2053-2068.

Rivest, L. R. (1995): *Game Tree Searching by Min/Max Approximation*, MIT Laboratory for Computer Science, Cambridge, Mass.

Robinson, D. F. y Foulds, L. R. (1981): Comparison of Phylogenetic Tree, *Mathematical Biosciences*, vol. 53, pp. 131-147.

Robinson, J. A. (1965): Machine Oriented Logic Based on the Resolution Principle, *Journal of the ACM*, vol. 12, No. 1, pp. 23-41.

Rosenkrantz, D. J., Stearns, R. E. y Lewis, P. M. (1977): An Analysis of Several Heuristics for the Traveling Salesman Problem, *SIAM Journal on Computing*, vol. 6, pp. 563-581.

Rosenthal, A. (1982): Dynamic Programming Is Optimal for Nonserial Optimization Problems, *SIAM Journal on Computing*, vol. 11, No. 1, pp. 47-59.

Rosler, U. (2001): On the Analysis of Stochastic Divide and Conquer Algorithms, *Algorithmica*, vol. 29, pp. 238-261.

Rosler, U. y Ruschendorf, L. (2001): The Contraction Method for Recursive Algorithms, *Algorithmica*, vol. 29, pp. 3-33.

Roura, S. (2001): Improved Master Theorems for Divide-and-Conquer Recurrences, *Journal of the ACM*, vol. 48, No. 2, pp. 170-205.

Rzhetsky, A. y Nei, M. (1992): A Simple Method for Estimating and Testing Minimum-Evolution Tree, *Molecular Biology and Evolution*, vol. 9, pp. 945-967.

Rzhetsky, A. y Nei, M. (1992): Statistical Properties of the Ordinary Least-Squares; Generalized Least-Squares; and Minimum-Evolution Methods of Phylogenetic Inference, *Journal of Molecular Evolution*, vol. 35, pp. 367-375.

Sahni, S. (1976): Algorithm for Scheduling Independent Tasks, *Journal of the ACM*, vol. 23, No. 1, pp. 116-127.

Sahni, S. (1977): General Techniques for Combinatorial Approximation, *Operations Research*, vol. 25, pp. 920-936.

Sahni, S. y Gonzalez, T. (1976): P-Complete Approximation Problems, *Journal of the ACM*, vol. 23, pp. 555-565.

Sahni, S. y Wu, S. Y. (1988): Two NP-Hard Interchangeable Terminal Problems, *IEEE Transactions on CAD*, vol. 7, No. 4, pp. 467-471.

Sakoe, H. y Chiba, S. (1978): Dynamic Programming Algorithm Optimization for Spoken Word Recognition, *IEEE Transactions on Acoustics Speech and Signal Processing*, vol. 27, pp. 43-49.

Santis, A. D. y Persiano, G. (1994): Tight Upper and Lower Bounds on the Path Length of Binary Trees, *SIAM Journal on Applied Mathematics*, vol. 23, No. 1, pp. 12-24.

Sarrafzadeh, M. (1987): Channel-Routing Problem in the Knock-Knee Mode Is NP-Complete, *IEEE Transactions on CAD*, vol. CAD-6, No. 4, pp. 503-506.

Schmidt, J. (1998): All Highest Scoring Paths in Weighted Grid Graphs and Their Application to Finding All Approximate Repeats in Strings, *SIAM Journal on Computing*, vol. 27, No. 4, pp. 972-992.

Schwartz, E. S. (1964): An Optimal Encoding with Minimum Longest Code and Total Number of Digits, *Information and Control*, vol. 7, No. 1, pp. 37-44.

Sedgewick, R. y Flajolet, D. (1996): *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, Mass.

Seiden, S. (1999): A Guessing Game and Randomized Online Algorithms, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, ACM Press, Portland, Oregon, pp. 592-601.

Seiden, S. (2002): On the Online Bin Packing Problem, *Journal of the ACM*, vol. 49, No. 5, septiembre, pp. 640-671.

Sekhon, G. S. (1982): Dynamic Programming Interpretation of Construction-Type Plant Layout Algorithms and Some Results, *Computer Aided Design*, vol. 14, No. 3, pp. 141-144.

Sen, S y Sherali, H. D. (1985): A Branch and Bound Algorithm for Extreme Point Mathematical Programming Problem, *Discrete Applied Mathematics*, vol. 11, No. 3, pp. 265-280.

Setubal, J. y Meidanis, J. (1997): *Introduction to Computational Biology*, PWS Publishing, Boston, Mass.

Sgall, J. (1996): Randomized On-Line Scheduling of Parallel Jobs, *Journal of Algorithms*, vol. 21, pp. 149-175.

Shaffer, C. A. (2001): *A Practical Introduction to Data Structures and Algorithm Analysis*, Prentice-Hall, Englewood Cliffs, Nueva Jersey.

Shamos, M. I. (1978): *Computational geometry*. Disertación doctoral, aún sin publicar, Yale University.

Shamos, M. I. y Hoey, D. (1975): Closest-Point Problems, *Proceedings of the Sixteenth Annual IEEE Symposium on Foundations of Computer Science*, IEEE Press, Washington, DC, pp. 151-162.

Shamos, M. I. y Hoey, D. (1976): Geometric Intersection Problems, *Proceedings of the Seventeenth Annual IEEE Symposium on Foundations of Computer Science*, IEEE Press, Washington, DC, pp. 208-215.

Shmueli, O. e Itai, A. (1987): Complexity of Views: Tree and Cyclic Schemas, *SIAM Journal on Computing*, vol. 16, No. 1, pp. 17-37.

Shreesh, J., Asish, M. y Binay, B. (1996): An Optimal Algorithm for the Intersection Radius of a Set of Convex Polygons, *Journal of Algorithms*, vol. 20, No. 2, pp. 244-267.

Simon, R. y Lee, R. C. T. (1971): On the Optimal Solutions to AND/OR Series-Parallel Graphs, *Journal of the ACM*, vol. 18, No. 3, pp. 354-372.

Slavik, P. (1997): A Tight Analysis of the Greedy Algorithm for Set Cover, *Journal of Algorithms*, vol. 25, pp. 237-254.

Sleator, D. D. y Tarjan, R. E. (1983): A Data Structure for Dynamic Trees, *Journal of Computer and System Sciences*, vol. 26, No. 3, pp. 362-391.

Sleator, D. D. y Tarjan, R. E. (1985a): Amortized Efficiency of List Update and Paging Rules, *Communications of the ACM*, vol. 28, No. 2, pp. 202-208.

Sleator, D. D. y Tarjan, R. E. (1985b): Self-Adjusting Binary Search Trees, *Journal of the ACM*, vol. 32, No. 3, pp. 652-686.

Sleator, D. D. y Tarjan, R. E. (1986): Self-Adjusting Heaps, *SIAM Journal on Computing*, vol. 15, No. 1, febrero, pp. 52-69.

Smith, D. (1984): Random Trees and the Analysis of Branch-and-Bound Procedures, *Journal of the ACM*, vol. 31, No. 1, pp. 163-188.

Smith, J. D. (1989): *Design and Analysis of Algorithms*, PWS Publishing, Boston, Mass.

Snyder, E. E. y Stormo, G. D. (1993): Identification of Coding Regions in Genomic DNA Sequences: An Application of Dynamic Programming and Neural Networks, *Nucleic Acids Research*, vol. 21, No. 3, pp. 607-613.

Solovay, R. y Strassen, V. (1977): A Fast Monte-Carlo Test for Primality, *SIAM Journal on Computing*, vol. 6, No. 1, pp. 84-85.

Spirakis, P. (1988): Optimal Parallel Randomized Algorithm for Sparse Addition and Identication, *Information and Computation*, vol. 76, No. 1, pp. 1-12.

Srimani, P. K. (1989): Probabilistic Analysis of Output Cost of a Heuristic Search Algorithm, *Information Sciences*, vol. 47, pp. 53-62.

Srinivasan, A. (1999): Improved Approximation Guarantees for Packing and Covering Integer Programs, *SIAM Journal on Computing*, vol. 29, pp. 648-670.

Srinivasan, A. y Teo, C. P. (2001): A Constant-Factor Approximation Algorithm for Packet Routing and Balancing Local vs. Global Criteria, *SIAM Journal on Computing*, vol. 30, pp. 2051-2068.

Steel, M. A. (1992): The Complexity of Reconstructing Trees from Qualitative Characters and Subtrees, *Journal of Classification*, vol. 9, pp. 91-116.

Steele, J. M. (1986): An Efron-Stein Inequality for Nonsymmetric Statistics, *Annals of Statistics*, vol. 14, pp. 753-758.

Stewart, G. W. (1999): The QLP Approximation to the Singular Value Decomposition, *SIAM Journal on Scientific Computing*, vol. 20, pp. 1336-1348.

Stoneking, M., Jorde, L. B., Bhatia, K. y Wilson, A. C. (1990): Geographic Variation in Human Mitochondrial DNA from Papua Nueva Guinea, *Genetics*, vol. 124, pp. 717-733.

Storer, J. A. (1977): NP-Completeness Results Concerning Data Compression (Prove NP-Complete Problem), *Technical Report 233*, Princeton University, Princeton, Nueva Jersey.

Strassen, V. (1969): Gaussian Elimination Is Not Optimal, *Numerische Mathematik*, vol. 13, pp. 354-356.

Stringer, C. B. y Andrews, P. (1988): Genetic and Fossil Evidence for the Origin of Modern Humans, *Science*, vol. 239, pp. 1263-1268.

Sutton, R. S. (1990): Integrated Architectures for Learning, Planning and Reacting Based on Approximating Dynamic Programming, *Proceedings of the Seventh International Conference on Machine Learning*, Morgan Kaufmann Publishers Inc., San Francisco, California, pp. 216-224.

Sweedyk, E. S. (1995): *A 2 1/2 approximation algorithm for shortest common superstring*. Tesis doctoral, aún sin publicar, University of California.

Sykora, O. y Vrto, I. (1993): Edge Separators for Graphs of Bounded Genus with Applications, *Theoretical Computer Science*, vol. 112, No. 2, pp. 419-429.

Szpankowski, W. (2001): *Average Case Analysis of Algorithms on Sequences*, John Wiley & Sons, Nueva York.

Tamassia, R. (1996): On-line Planar Graph Embedding, *Journal of Algorithms*, vol. 21, pp. 201-239.

Tang, C. Y., Buehrer, D. J. y Lee, R. C. T. (1985): On the Complexity of Some Multi-Attribute File Design Problems, *Information Systems*, vol. 10, No. 1, pp. 21-25.

Tarhio, J. y Ukkonen, E. (1986): A Greedy Algorithm for Constructing Shortest Common Superstrings, *Lecture Notes in Computer Science*, vol. 233, pp. 602-610.

Tarhio, J. y Ukkonen, E. (1988): A Greedy Approximation Algorithm for Constructing Shortest Common Superstrings, *Theoretical Computer Science*, vol. 57, pp. 131-145.

Tarjan, R. E. (1983): Data Structures and Network Algorithms, *SIAM*, vol. 29.

Tarjan, R. E. (1985): Amortized Computational Complexity, *SIAM Journal on Algebraic Discrete Methods*, vol. 6, No. 2, pp. 306-318.

Tarjan, R. E. (1987): Algorithm Design, *Communications of the ACM*, vol. 30, No. 3, pp. 204-213.

Tarjan, R. E. y Van Leeuwen, J. (1984): Worst Case Analysis of Set Union Algorithms, *Journal of the ACM*, vol. 31, No. 2, pp. 245-281.

Tarjan, R. E. y Van Wyk, C. J. (1988): An $O(n \log \log n)$-Time Algorithm for Triangulating a Simple Polygon, *SIAM Journal on Computing*, vol. 17, No. 1, pp. 143-178.

Tataru, D. (1992): *Viscosity Solutions for the Dynamic Programming Equations, Applied Mathematics and Optimization*, vol. 25, pp. 109-126.

Tatman, J. A. y Shachter, R. D. (1990): Dynamic Programming and Influence Diagrams, *IEEE Transactions on Systems, Man and Cybernetics*, vol. 20, pp. 365-379.

Tatsuya, A. (2000): Dynamic Programming Algorithms for RNA Secondary Structure Prediction with Pseudoknots, *Discrete Applied Mathematics*, vol. 104, pp. 45-62.

Teia, B. (1993): Lower Bound for Randomized List Update Algorithms, *Information Processing Letters*, vol. 47, No. 1, pp. 5-9.

Teillaud, M. (1993): *Towards Dynamic Randomized Algorithms in Computational Geometry*, Springer-Verlag, Nueva York.

Thomassen, C. (1997): On the Complexity of Finding a Minimum Cycle Cover of a Graph, *SIAM Journal on Computing*, vol. 26, No. 3, pp. 675-677.

Thulasiraman, K. y Swamy, M. N. S. (1992): *Graphs: Theory and Algorithms*, John Wiley & Sons, Nueva York.

Tidball, M. M. y Atman, E. (1996): Approximations in Dynamic Zero-Sum Games I, *SIAM Journal on Control and Optimization*, vol. 34, No. 1, pp. 311-328.

Ting, H. F. y Yao, A. C. (1994): Randomized Algorithm for Finding Maximum with $O((\log n)^2)$, *Information Processing Letters*, vol. 49, No. 1, pp. 39-43.

Tisseur, F. y Dongarra, J. (1999): A Parallel Divide and Conquer Algorithm for the Symmetric Eigenvalue Problem on Distributed Memory Architectures, *SIAM Journal on Scientific Computing*, vol. 20, No. 6, pp. 2223-2236.

Tomasz, L. (1998): A Greedy Algorithm Estimating the Height of Random Trees, *SIAM Journal on Discrete Mathematics*, vol. 11, pp. 318-329.

Tong, C. S. y Wong, M. (2002): Adaptive Approximate Nearest Neighbor Search for Fractal Image Compression, *IEEE Transactions on Image Processing*, vol. 11, No. 6, pp. 605-615.

Traub, J. F. y Wozniakowski, H. (1984): On the Optimal Solution of Large Linear Systems, *Journal of the ACM*, vol. 31, No. 3, pp. 545-549.

Trevisan, L. (2001): Non-Approximability Results for Optimization Problems on Bounded Degree Instances, *ACM*, pp. 453-461.

Tsai, C. J. y Katsaggelos, A. K. (1999): Dense Disparity Estimation with a Divide-and-Conquer Disparity Space Image Technique, *IEEE Transactions on Multimedia*, vol. 1, No. 1, pp. 18-29.

Tsai, K. H. y Hsu, W. L. (1993): Fast Algorithms for the Dominating Set Problem on Permutation Graphs, *Algorithmica*, vol. 9, No. 6, pp. 601-614.

Tsai, K. H. y Lee, D. T. (1997): K Best Cuts for Circular-Arc Graphs, *Algorithmica*, vol. 18, pp. 198-216.

Tsai, Y. T., Lin, Y. T. y Hsu, F. R. (2002): The On-Line First-Fit Algorithm for Radio Frequency Assignment Problem, *Information Processing Letters*, vol. 84, No. 4, pp. 195-199.

Tsai, Y. T. y Tang, C. Y. (1993): The Competitiveness of Randomized Algorithms for Online Steiner Tree and On-Line Spanning Tree Problems, *Information Processing Letters*, vol. 48, pp. 177-182.

Tsai, Y. T., Tang, C. Y. y Chen, Y. Y. (1994): Average Performance of a Greedy Algorithm for On-Line Minimum Matching Problem on Euclidean Space, *Information Processing Letters*, vol. 51, pp. 275-282.

Tsai, Y. T., Tang, C. Y. y Chen, Y. Y. (1996): An Average Case Analysis of a Greedy Algorithm for the On-Line Steiner Tree Problem, *Computers and Mathematics with Applications*, vol. 31, No. 11, pp. 121-131.

Turner, J. S. (1989): Approximation Algorithms for the Shortest Common Superstring Problem, *Information and Computation*, vol. 83, pp. 1-20.

Ukkonen, E. (1985a): Algorithms for Approximate String Matching, *Information and Control*, vol. 64, pp. 10-118.

Ukkonen, E. (1985b): Finding Approximate Patterns in Strings, *Journal of Algorithms*, vol. 6, pp. 132-137.

Ukkonen, E. (1990): A Linear Time Algorithms for Finding Approximate Shortest Common Superstrings, *Algorithmica*, vol. 5, pp. 313-323.

Ukkonen, E. (1992): Approximate String-Matching with Q-Grams and Maximal Matches, *Theoretical Computer Science*, vol. 92, pp. 191-211.

Unger, R. y Moult, J. (1993): Finding the Lowest Free Energy Conformation of a Protein Is an NP-Hard Problem: Proof and Implications (Prove NP-Complete Problem), *Bulletin of Mathematical Biology*, vol. 55, pp. 1183-1198.

Uspensky, V. y Semenov, A. (1993): *Algorithms: Main Ideas and Applications*, Kluwer Press, Norwell, Mass.

Vaidya, P. M. (1988): Minimum Spanning Trees in *k*-Dimensional Space, *SIAM Journal on Computing*, vol. 17, No. 3, pp. 572-582.

Valiant, L. G. y Vazirani, V. V. (1986): NP Is as Easy as Detecting Unique Solutions, *Theoretical Computer Science*, vol. 47, No. 1, pp. 85-93.

Van Leeuwen, J. (1990): *Handbook of Theoretical Computer Science: Volume A: Algorithms and Complexity*, Elsevier, Ámsterdam.

Vazirani, V. V. (2001): *Approximation Algorithms*, Springer-Verlag, Nueva York.

Verma, R. M. (1997): General Techniques for Analyzing Recursive Algorithms with Applications, *SIAM Journal on Computing*, vol. 26, No. 2, pp. 568-581.

Veroy, B. S. (1988): Average Complexity of Divide-and-Conquer Algorithms, *Information Processing Letters*, vol. 29, No. 6, pp. 319-326.

Vintsyuk, T. K. (1968): Speech Discrimination by Dynamic Programming, *Cybernetics*, vol. 4, No. 1, pp. 52-57.

Vishwanathan, S. (1992): Randomized Online Graph Coloring, *Journal of Algorithms*, vol. 13, pp. 657-669.

Viterbi, A. J. (1967): Error Bounds for Convolutional Codes and an Asymptotically Optimal Decoding Algorithm, *IEEE Transactions on Information Theory*, pp. 260-269.

Vliet, A. (1992): An Improved Lower Bound for On-Line Bin Packing Algorithms, *Information Processing Letters*, vol. 43, No. 5, pp. 277-284.

Von Haeselerm A., Blum, B., Simpson, L., Strum, N. y Waterman, M. S. (1992): Computer Methods for Locating Kinetoplastid Cryptogenes, *Nucleic Acids Research*, vol. 20, pp. 2717-2724.

Voronoi, G. (1908): Nouvelles Applications des Parameters Continus a la Theorie des Formes Quadratiques. Deuxieme M'emoire: Recherches Sur les Parall'eloedres Primitifs, *J. Reine Angew. Math.*, vol. 134, pp. 198-287.

Vyugin, M. V. y V'yugin, V. V. (2002): On Complexity of Easy Predictable Sequences, *Information and Computation*, vol. 178, No. 1, pp. 241-252.

Wah. B. W. y Yu, C. F. (1985): Stochastic Modeling of Branch-and-Bound Algorithms with Best-First Search, *IEEE Transactions on Software Engineering*, vol. SE-11, No. 9, pp. 922-934.

Walsh, T. R. (1984): How Evenly Should One Divide to Conquer Quickly? *Information Processing Letters*, vol. 19, No. 4, pp. 203-208.

Wang, B. F. (1997): Tighter Bounds on the Solution of a Divide-and-Conquer Maximum Recurrence, *Journal of Algorithms*, vol. 23, pp. 329-344.

Wang, B. F. (2000): Tight Bounds on the Solutions of Multidimensional Divide-and-Conquer Maximum Recurrences, *Theoretical Computer Science*, vol. 242, pp. 377-401.

Wang, D. W. y Kuo, Y. S. (1988): A Study of Two Geometric Location Problems, *Information Processing Letters*, vol. 28, No. 6, pp. 281-286.

Wang, J. S. y Lee, R. C. T. (1990): An Efficient Channel Routing Problem to Yield an Optimal Solution, *IEEE Transactions on Computers*, vol. 39, No. 7, pp. 957-962.

Wang, J. T. L., Zhang, K., Jeong, K. y Shasha, D. (1994): A System for Approximate Tree Matching, *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, No. 4, pp. 559-571, 1041-4347.

Wang, L. y Gusfield, D. (1997): Improved Approximation Algorithms for Tree Alignment, *Journal of Algorithms*, vol. 25, No. 2, pp. 255-273.

Wang, L. y Jiang, T. (1994): On the Complexity of Multiple Sequence Alignment, *Journal of Computational Biology*, vol. 1, No. 4, pp. 337-348.

Wang, L., Jiang, T. y Gusfield, D. (2000): A More Efficient Approximation Scheme for Tree Alignment, *SIAM Journal on Applied Mathematics*, vol. 30, No. 1, pp. 283-299.

Wang, L., Jiang, T. y Lawler, E. L. (1996): Approximation Algorithms for Tree Alignment with a Given Phylogeny, *Algorithmica*, vol. 16, pp. 302-315.

Wang, X., He, L., Tang, Y. y Wee, W. G. (2003): A Divide and Conquer Deformable Contour Method with a Model Based Searching Algorithm, *IEEE Transactions on Systems, Man and Cybernetics*, vol. 33, No. 5, pp. 738-751.

Wareham, H. T. (1995): A Simplified Proof of the NP- and MAX SNP-Hardness of Multiple Sequence Tree Alignments (Prove NP-Complete Problem), *Journal of Computational Biology*, vol. 2, No. 4.

Waterman, M. S. (1995): *Introduction to Computational Biology: Maps, Sequences and Genomes*, Chapman & Hall/CRC, Nueva York.

Waterman, M. S. y Smith, T. F. (1978): RNA Secondary Structure: A Complete Mathematical Analysis, *Mathematical Bioscience*, vol. 42, pp. 257-266.

Waterman, M. S. y Smith, T. F. (1986): Rapid Dynamic Programming Algorithms for RNA Secondary Structure, *Advances in Applied Mathematics*, vol. 7, pp. 455-464.

Waterman, M. S. y Vingron, M. (1994): Sequence Comparison Significance and Poisson Approximation, *Statistical Science*, vol. 2, pp. 367-381.

Weide, B. (1977): A Survey of Analysis Techniques for Discrete Algorithms, *ACM Computing Surveys*, vol. 9, No. 4, pp. 291-313.

Weiss, M. A. (1992): *Data Structures and Algorithm Analysis*, Benjamin Cummings, Redwood City, California.

Wenger, R. (1997): Randomized Quickhull, *Algorithmica*, vol. 17, No. 3, pp. 322-329.

Westbrook, J. y Tarjan, R. E. (1989): Amortized Analysis of Algorithms for Set Union with Backtracking, *SIAM Journal on Computing*, vol. 18, No. 1, pp. 1-11.

Wilf, H. S. (1986): *Algorithms & Complexity*, Prentice-Hall, Engelwood Cliffs, Nueva Jersey.

Williams, J. W. J. (1964): Heapsort: Algorithm 232, *Communications of the ACM*, vol. 7, pp. 347-348.

Wood, D. (1993): *Data Structures, Algorithms and Performance*, Addison-Wesley, Reading, Mass.

Wright, A. H. (1994): Approximate String Matching Using Withinword Parallelism, *Software: Practice and Experience*, vol. 24, pp. 337-362.

Wu, B. Y., Lancia, G., Bafna, V., Chao, K. M., Ravi, R. y Tang, C. Y. (2000): A Polynomial-Time Approximation Scheme for Minimum Routing Cost Spanning Trees, *SIAM Journal on Computing*, vol. 29, No. 3, pp. 761-778.

Wu, L. C. y Tang, C. Y. (1992): Solving the Satisability Problem by Using Randomized Approach, *Information Processing Letters*, vol. 41, No. 4, pp. 187-190.

Wu, Q. S., Chao, K. M. y Lee, R. C. T. (1998): The NPO-Completeness of the Longest Hamiltonian Cycle Problem, *Information Processing Letters*, vol. 65, pp. 119-123.

Wu, S. y Manber, U. (1992): Fast Text Searching Allowing Errors, *Communications of the ACM*, vol. 35, pp. 83-90.

Wu, S. y Myers, G. (1996): A Subquadratic Algorithm for Approximate Limited Expression Matching, *Algorithmica*, vol. 15, pp. 50-67.

Wu, T. (1996): A Segment-Based Dynamic Programming Algorithm for Predicting Gene Structure, *Journal of Computational Biology*, vol. 3, pp. 375-394.

Wu, Y. F., Widmayer, P. y Wong, C. K. (1986): A Faster Approximation Algorithm for the Steiner Problem in Graphs, *Acta Informatica*, vol. 23, No. 2, pp. 223-229.

Xu, S. (1990): *Dynamic programming algorithms for alignment hyperplanes*. Tesis de maestría, aún sin publicar, University of Southern California.

Yagle, A. E. (1998): Divide-and-Conquer 2-D Phase Retrieval Using Subband Decomposition and Filter Banks, *IEEE Transactions on Signal Processing*, vol. 46, No. 4, pp. 1152-1154.

Yang, C. I., Wang, J. S. y Lee, R. C. T. (1989): A Branch-and-Bound Algorithm to Solve the Equal-Execution-Time Job Scheduling Problem with Precedence Constraint and Prole, *Computers and Operations Research*, vol. 16, No. 3, pp. 257-269.

Yannakakis, M. (1985): A Polynomial Algorithm for the Min-Cut Linear Arrangement of Trees, *Journal of the Association for Computing Machinery*, vol. 32, No. 4, pp. 950-988.

Yannakakis, M. (1989): Embedding Planar Graphs in Four Pages, *Journal of Computer and System Sciences*, vol. 38, No. 1, pp. 36-67.

Yao, A. C. (1981): Should Tables be Sorted, *Journal of the ACM*, vol. 28, No. 3, pp. 615-628.

Yao, A. C. (1985): On the Complexity of Maintaining Partial Sums, *SIAM Journal on Computing*, vol. 14, No. 2, pp. 277-288.

Yao, A. C. (1991): Lower Bounds to Randomized Algorithms for Graph Properties, *Journal of Computer and System Sciences*, vol. 42, No. 3, pp. 267-287.

Ye, D. y Zhang, G. (2003): On-Line Extensible Bin Packing with Unequal Bin Sizes, *Lecture Notes in Computer Science*, vol. 2909, pp. 235-247.

Yen, C. C. y Lee, R. C. T. (1990): The Weighted Perfect Domination Problem, *Information Processing Letters*, vol. 35, pp. 295-299.

Yen, C. C. y Lee, R. C. T. (1994): Linear Time Algorithm to Solve the Weighted Perfect Domination Problem in Series-Parallel Graphs, *European Journal of Operational Research*, vol. 73, No. 1, pp. 192-198.

Yen, C. K. y Tang, C. Y. (1995): An Optimal Algorithm for Solving the Searchlight Guarding Problem on Weighted Trees, *Information Sciences*, vol. 87, pp. 79-105.

Yen, F. M. y Kuo, S. Y. (1997): Variable Ordering for Ordered Binary Decision Diagrams by a Divide-and-Conquer Approach, *IEE Proceedings: Computer and Digital Techniques*, vol. 144, No. 5, pp. 261-266.

Yoo, J., Smith, K. F. y Gopalarkishnan, G. (1997): A Fast Parallel Squarer Based on Divide-and-Conquer, *IEEE Journal of Solid-State Circuits*, vol. 32, No. 6, pp. 909-912.

Young, N. (2000): On-Line Paging Against Adversarially Biased Random Inputs, *Journal of Algorithms*, vol. 37, pp. 218-235.

Younger, D. H. (1967): Recognition and Parsing of Context-Free Languages in Time n^3, *Information and Control*, vol. 10, No. 2, pp. 189-208.

Zelikovsky, A. (1993): An 11/6 Approximation Algorithm for the Steiner Tree Problem in Graph, *Information Processing Letters*, vol. 46, pp. 317-323.

Zemel, E. (1987): A Linear Randomized Algorithm for Searching Rank Functions, *Algorithmica*, vol. 2, No. 1, pp. 81-90.

Zhang, K. y Jiang, T. (1994): Some Max SNP-Hard Results Concerning Unordered Labeled Trees, *Information Processing Letters*, vol. 49, pp. 249-254.

Zhang, Z., Schwartz, S., Wagner, L. y Miller, W. (2003): A Greedy Algorithm for Aligning DNA Sequences, *Journal of Computational Biology*, vol. 7, pp. 203-214.

Zuker, M. (1989): The Use of Dynamic Programming Algorithms in RNA Secondary Structure Prediction, en M. S. Waterman (Ed.), *Mathematical Methods for DNA Sequences* (pp. 159-185), CRC Press, Boca Raton, Florida.

Lista de figuras

Capítulo 1

Capítulo 2

Capítulo 3

Capítulo 4

Capítulo 5

Capítulo 6

Capítulo 7

Capítulo 8

Capítulo 9

Capítulo 10

Capítulo 11

Capítulo 12

Índice onomástico

Índice analítico